EFFICIENT SUPERNODAL SPARSE CHOLESKY FACTORIZATION

By

ADRIAN MASCARENHAS

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

To my wife, Andrea, for all her love, support and encouragement.

ACKNOWLEDGMENTS

I wish to thank Dr. Timothy A. Davis for giving me an opportunity to work under him as a research assistant, which has eventually led to this thesis. He has been very supportive and helpful, and this thesis would not be possible without his supervision, guidance and encouragement. I would also like to thank Dr. William Hager and Dr. Baba Vemuri for serving on my thesis committee. Most importantly, I would like to thank God for always being there for me.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

EFFICIENT SUPERNODAL SPARSE CHOLESKY FACTORIZATION

By

Adrian Mascarenhas

May 2002

Chair: Dr. Timothy A. Davis
Department: Computer and Information Science of Engineering

Many scientific and engineering applications today make use of large sparse symmetric positive definite systems. Cholesky factorization is commonly used to solve such systems. A major factor that limits performance of the Cholesky factorization is the cost associated with moving data between memory and the processor. An algorithm that uses supernodes to reduce this cost is presented. A supernode is a collection of interspersed columns (within the Cholesky factor) that have similar sparsity structure. This block matrix computation improves performance by reducing the amount of indirect addressing and memory traffic.

CHAPTER 1
INTRODUCTION

Sparse Cholesky factorization followed by forward and backward triangular solutions is commonly used to solve large sparse symmetric positive definite systems of linear equations. It is the bottleneck in a wide range of computations, from domains such as structural analysis, computational fluid dynamics, device and process simulation and electric network problems. The goal is to minimize the time required for sparse Cholesky factorization.

Overview of Sparse Cholesky Factorization

For any $n \times n$ symmetric positive definite matrix **A**, its Cholesky factor **L** is the lower triangular matrix with positive diagonal such that $\mathbf{A}=\mathbf{L}\mathbf{L}^{\mathbf{T}}$ or $\mathbf{A}=\mathbf{L}\mathbf{D}\mathbf{L}^{\mathbf{T}}$ where **D** is a diagonal matrix. When **A** is sparse, it will generally suffer some fill during the computation of **L**; that is, some of the zero elements in A will become nonzero elements in **L**. To reduce time and storage requirements, only the nonzero positions of **L** are stored and operated on during sparse Cholesky factorization. Determination of the nonzero positions of **L** is often called *symbolic factorization* (terminology introduced by George and Liu [5]) while the actual computation of **L** is referred to as *numerical factorization*. The major bottleneck in sparse factorization is not the number of floating-point operations, but rather the cost of fetching data from the main memory. The goal is to decrease the number of memory-to-register transfers executed and to improve the program's cache behavior to decrease the cost of each transfer. An algorithm that uses *supernodes* to reduce this cost is discussed in this thesis. A supernode is a collection of

interspersed columns (within the Cholesky factor) that have similar sparsity structure. Block matrix computation using supernodes improves performance by reducing the amount of indirect addressing and memory traffic.

Thesis Objectives

A supernodal approach to solving sparse Cholesky factorization of the matrix $\mathbf{AA}^{\mathbf{T}}$ is presented, where **A** is *m*-by-*n*. The matrix $\mathbf{AA}^{\mathbf{T}}$ must be symmetric and positive definite. The technique developed for the matrix $\mathbf{AA}^{\mathbf{T}}$ can be extended to any symmetric positive definite matrix **A**. This particular form of Cholesky factorization ($\mathbf{AA}^{\mathbf{T}} = \mathbf{LDL}^{\mathbf{T}}$) arises in the LP Dual Active Set Algorithm (LPDASA) [7] for solving linear programming problems. The algorithms presented here were developed for use by the LPDASA method. The sparse Cholesky factorization consists of the following two major stages:

- **Symbolic factorization.** The pattern of the nonzeros in the Cholesky factor **L** is computed. There is no numerical computation.
- **Numerical factorization.** The Cholesky factor **L** is numerically computed. There are three different methods for the numerical factorization. These are as follows:
  - Column-column numerical factorization
  - Supernode-column numerical factorization
  - Supernode-supernode numerical factorization

The supernodal approach for solving sparse Cholesky factorization was discussed by several researchers [1,9–12]. They defined supernodes as a set of contiguous columns that have the same sparsity structure. Having the columns of a supernode be contiguous can be a bottleneck for various applications where Matrix **A** is continuously being modified. An example of such an application is the Linear Program Dual Active Set Algorithm (LPDASA) [7], where Matrix **A** corresponds to the basic variables in the

current basis of the linear program. In successive iterations, variables are brought in and out of the basis, leading to changes of the form $\mathbf{AA^T} + \sigma\mathbf{ww^T}$. On an update, $\mathbf{AA^T} + \mathbf{ww^T}$, new entries are added causing supernodes to merge (2 columns now have the same pattern) or split (2 columns had the same pattern, but one gets updated and the other does not). The same occurs during a downdate, $\mathbf{AA^T} - \mathbf{ww^T}$. This results in columns that are no longer contiguous but have the same sparsity structure. The main objective of this thesis is to show that supernodal methods for solving the Cholesky factorization, where a supernode contains columns that need not be contiguous but have the same sparsity structure, have a much better performance than the simplicial column-column factorization method.

Organization of Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 contains the mathematical background of the Cholesky factorization and also explains the notation used in this thesis.
- Chapter 3 describes the algorithm for the *symbolic factorization* and its implementation.
- Chapter 4 describes the algorithm and the implementation of the *column-column numerical factorization*.
- Chapter 5 contains the definition of a supernode and an algorithm for constructing the supernodal elimination tree.
- Chapter 6 describes the algorithm and the implementation of *supernode-column numerical factorization*.
- Chapter 7 describes the algorithm and the implementation of *supernode-supernode numerical factorization*.
- Chapter 8 presents performance results for all three 3 methods of *numerical factorization*.
- Chapter 9 gives a conclusion of the results with a discussion of future work.

- Appendix A contains the source code for *symbolic factorization* in MATLAB.
- Appendix B contains the source code for *column-column numerical factorization* in C.
- Appendix C contains the source code for *supernode-column numerical factorization* in C.

CHAPTER 2
BACKGROUND

This chapter explains the theory behind the dense Cholesky factorization and some related graph theory concepts.

Cholesky Factorization

Golub and Loan [6] gave the algorithms for the dense Cholesky factorization in their book. They describe the theory behind $\mathbf{A} = \mathbf{LL}^T$ and $\mathbf{A}=\mathbf{LDL}^T$ Cholesky factorization. Although the implementation of the algorithms was done for the $\mathbf{A}=\mathbf{LDL}^T$ Cholesky factorization, the reader is presented with the algorithm for $\mathbf{A} = \mathbf{LL}^T$ Cholesky factorization for a better understanding of the left-looking Cholesky algorithm. The techniques described in this thesis are applicable to both forms of the Cholesky factorization. These algorithms are discussed in this section.

Dense $\mathbf{LL}^T$ Cholesky Factorization

The Cholesky factor **L** of a positive symmetric definite matrix **A** is a lower triangular matrix such that $\mathbf{A} = \mathbf{LL}^T$. Using MATLAB notation, let `A(:,j)` represent entire column j of **A**. Similarly, let `L(:,j)` represent the entire column j of **L**. The equation

$$\texttt{A(:,j)} = \sum_{k=1}^{j} \texttt{L(j,k)}\ \texttt{L(:,k)}$$

holds true for every column j of **A**. Using this equation the column j of **L** can be computed as

$$\texttt{L(j,j)}\ \texttt{L(:,j)} = \texttt{A (:,j)} - \sum_{k=1}^{j-1} \texttt{L(j,k)}\ \texttt{L(:,k)} \equiv \texttt{v}.$$

If the first j-1 columns of **L** are known, then v is computable. It follows by the above equation that `L(j:n, j) = v (j:n) / √v (j)` . This is a scaled gaxpy[1] operation. A gaxpy-based method for computing the Cholesky factorization in MATLAB notation is shown in Figure 2-1.

```
for j = 1:n
   v (j:n) = A(j:n, j)
   for k = 1:j-1
      v(j:n) = v(j:n) - L (j,k) * L(j:n,k)
   end
   L(j:n,j) = v(j:n) / √v(j)
end
```

Figure 2-1 Algorithm for computing dense $\mathbf{LL}^T$ Cholesky factorization

In Figure 2-1, it can be seen that the computation of the column j of **L** looks at all (j-1) columns to the left of it, and so this approach is called the **Left Looking** approach. There are other approaches to solving the Cholesky factorization such as the **Right Looking** and the **Multifrontal** method, but this thesis focuses on just the improving the performance of the Left Looking method.

Note that when computing a column j of **L** one need not look at all the columns to the left of j in **L**, but only those columns to the left of j that have `L(j,k)` as nonzero, that is, only those columns that have a nonzero value in the row j. So when computing j, a link list of all the columns that have nonzeros in row j is used.

Ng and Peyton [11] identified two regions in this algorithm where performance can be improved:

- **cmod (j,k)**. This is the modification of column j by a multiple of column k. In Figure 2-1, this corresponds to the operation of `v(j:n)=v(j:n)-L(j,k)*L(j:n,k)` inside the inner loop.

[1] Gaxpy : It is a mnemonic for "general A x plus y, that is, $z = y + Ax$ $x \in R^n$, $y \in R^n$, $A \in R^{n \times n}$.

- **cdiv (j)**. This is the scaling of column j. In Figure 2-1, this corresponds to the operation of `L(j:n,j) = v(j:n) / √v(j)` inside the outer loop.

Dense LDL^T Cholesky Factorization

This is a variation of the **A=LL^T** Cholesky factorization where the diagonal is stored in **D**. Golub and Loan [6] described the algorithm in MATLAB notation as in Figure 2-2.

```
for j = 1:n
   for i = 1:j-1
      v(i) = L(j,i) * d(i)
   end
   v (j) = A(j, j)  - L(j,1:j-1) * v(1:j-1)
   d(j) = v(j)
   L(j+1:n,j) = (A(j+1:n,j) - L (j+1:n,1:j-1) * v(1:j-1)) / d(j)
end
```

Figure 2-2 Algorithm for computing **LDL^T** Cholesky factorization

In Figure 2-2, the scalars are computed and stored in the array **v**. For each column k where k = 1:j-1, the scalar is `L(j,k)*d(k)`. This scalar is also referred to as `l_jk_times_d_k` in the remainder of this thesis.

Here too, only those columns to the left of j that have a nonzero value in row j will be used to compute column j. There are two regions in this algorithm where one can improve the performance:

- **cmod (j,k)**. This is the modification of column j by a multiple of column k. In Figure 2-2, this corresponds to the operation of `A(j+1:n,j)-L(j+1,1:j-1)*v(1:j-1)` where k lies in the range of 1: j-1.

- **cdiv (j)**. This is the scaling of column j. In Figure 2-2, this corresponds to the operation of `cmod(j,k)/d(j)`, that is, the division of the result of `cmod(j,k)` by `d(j)`.

Graph Theory

Davis and Hager [4] discuss the graph theory concepts. The notations used by them are used for the remainder of this thesis. The concepts of parent and elimination tree are discussed in this section.

Pattern of L

The pattern of **L** is denoted by $\mathcal{L}$ where $\mathcal{L}$ is a collection of patterns:

$$\mathbf{L} = \{L_1, L_2, \ldots, L_m\},$$

where the nonzero pattern of column j of **L** is denoted,

$$\mathcal{L}_j = \{i : l_{ij} \neq 0\}.$$

Some of these predicted nonzeros may be zero due to the numerical cancellation during the factorization process. The statement "$l_{ij} \neq 0$" means that l_{ij} is *symbolically* nonzero. $|\mathcal{L}|$ denotes the sum of the sizes of the sets it contains. Also, the notation $\mathcal{L}_{j,k}$ is used to refer to the pattern of the $\mathbf{L}_{j,k}$ entry.

Pattern of A

The pattern of **A** is denoted by $\mathcal{A}$ where $\mathcal{A}$ is the collection of patterns:

$$\mathbf{A} = \{A_1, A_2, \ldots, A_m\},$$

where the nonzero pattern of column j of **A** is denoted

$$\mathcal{A}_j = \{i : a_{ij} \neq 0\}.$$

|A| denotes the sum of the sizes of the sets it contains.

Parent Map

Parent map Π [8] is used to define the *elimination tree*. For any node j, Π (j) is the row index of the first nonzero element in column j of **L** beneath the diagonal element:

$$\Pi(j) = \min \mathcal{L}_j \setminus \{j\},$$

where "min X" denotes the smallest element of X:

$$\min X = \min_{i \in X} i.$$

The min of the empty set is zero. Note that $j < \Pi(j)$ except in the case where the diagonal element in column j is the only nonzero element.

Children Multifunction

Children multifunction is the inverse Π^{-1} of the parent map. That is, the children of node k are the set defined by

$$\Pi^{-1}(k) = \{j: \Pi(j)=k\}.$$

Ancestors

The *ancestors* of a node j, denoted $\mathcal{P}(j)$, are the set of successive parents:

$$P(j) = \{j, \Pi(j), \Pi(\Pi(j)), ..\} = \{\Pi^0(j), \Pi^1(j), \Pi^2(j), ...\}$$

Elimination Tree

The sequence of nodes $j, \Pi(j), \Pi(\Pi(j)), \ldots$ forming $\mathcal{P}(k)$ is called the *path* from j to the associated tree *root*. An *elimination tree* is the collection of all such paths leading to a root. An *elimination forest* is the set of all such trees. If column j of **L** has only one nonzero element, the diagonal element, then j will be the root of a separate tree. In most cases, there is a single tree having root *m* and so in the remainder of the thesis, the term *elimination tree* is used even if there is more than one tree forming a *forest*.

An example that explains the concepts of parent, child and elimination tree is shown in Figure 2-3. In this example, one can see that the pattern of nonzeros for column 1 is $\mathcal{L}_1=\{1,3\}$, and so the parent of the node for column 1is $\Pi(1) = 3$. For this example, the pattern of nonzeros and parent of each column is shown in Table 2-1. The pattern for column 8 is $\mathcal{L}_8=\{8\}$, so the parent for column 8 is $\Pi(8) = 0$, which means that node 8 has no parent and so is the root of the tree.

The *elimination tree* shows the column dependencies and the order in which the columns must be computed during the computation of **L**. For instance in order to compute any node, all of its children nodes must be computed first. This order has to be

strictly maintained. The column representing a child node will always lie to the left of the column representing its parent node and is thus consistent with the *left-looking* approach described above in the previous section. The elimination tree is used to obtain the pattern or set of nonzeros of all the columns within **L**. This process of finding the nonzero pattern of the Cholesky factor **L** is called as *symbolic factorization* which is discussed in Chapter 3.

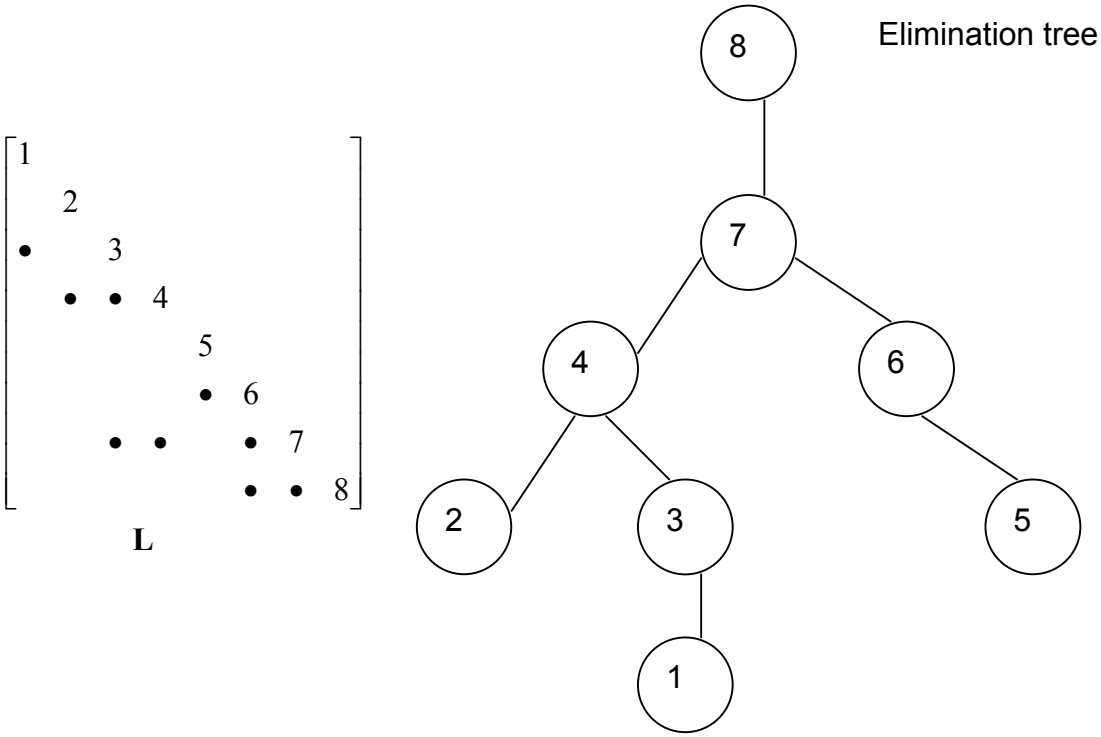

Figure 2-3 Sample matrix **L** and its elimination tree

Table 2-1. Nonzero pattern and parent for each column of sample matrix **L**

j	$\mathcal{L}_j$	**Π (j)**
1	{1,3}	3
2	{2,4}	4
3	{3,4,7}	4
4	{4,7}	7
5	{5,6}	6
6	{6,7,8}	7
7	{7,8}	8
8	{8}	0 (root)

Note that the nodes in the elimination tree are ordered in a non-decreasing order, that is, the parent node has a higher index than any of its children. This ensures the left-looking computation of the method, by making sure that before any parent is evaluated, all of its children, that is, all columns to the left of it, are evaluated first.

CHAPTER 3
SYMBOLIC FACTORIZATION

It is often desirable to determine the structure of **L** before computing it numerically, since the information allows a data structure to be set up prior to the numerical factorization. Then numerical factorization can proceed with a fixed storage structure. The determination of the structure or pattern of nonzeros of the Cholesky factor **L** is called as the *symbolic factorization* of **A** [5]. Numerical computation is done only on the nonzeros contained in the symbolic factor. This symbolic factor is denoted as $\mathcal{L}$.

Algorithm for Symbolic Factorization

The algorithm for the symbolic factorization of a matrix of the form $\mathbf{AA}^{\mathbf{T}}$ as discussed by Davis and Hager [4] is shown in Figure 3-1.

$\Pi(j) = 0$ for each j
for $j = 1$ to m do

$$\mathcal{L}_j = \{j\} \cup \left(\bigcup_{c \ni \Pi^{-1}(j)} \mathcal{L}_c \setminus \{c\} \right) \cup \left(\bigcup_{\min \mathcal{A}_k = j} \mathcal{A}_k \right)$$

$$\Pi(j) = \min \mathcal{L}_j \setminus \{j\}$$

end for

Figure 3-1 Algorithm for symbolic factorization

In Figure 3-1, it can be seen that the pattern of column j of **L** is the union of the pattern of column j of $\mathbf{AA}^{\mathbf{T}}$ and the patterns of the child nodes of node j. The pattern of column j of $\mathbf{AA}^{\mathbf{T}}$ is computed by taking the union of the patterns of the columns of **A** whose first nonzero element is j. The *elimination tree*, connecting each child to its parent, is easily formed during the symbolic factorization. This complete algorithm can be done

in $O(|\mathcal{L}| + |\mathcal{A}|)$ time[1]. Observe that the pattern of the parent of node j contains all the entries in the pattern of column j except j itself. See Appendix A for the implementation of the above algorithm in MATLAB.

Data Structures for Symbolic Factorization

There are two main data structures used during the symbolic factorization. These are for storing the matrices **A** and **L**, and for storing the elimination tree.

Data Structure for Storing Matrices L and A

Both the matrix **A** and its Cholesky factor **L** are sparse, that is, they have a large number of zero entries. So rather than storing these matrices as two-dimensional arrays, only the nonzero entries and their locations are stored using a packed data structure. Also, since the Cholesky factorization described in this thesis uses a left-looking column approach, these matrices are stored by columns rather than by rows. The Cholesky factor **L** is represented by the following data structure:

- **ln**. The number of columns in **L**, which is equal to *m* of the *m*-by-*n* matrix **A**.
- **Li**. An array of length $|\mathcal{L}|$ for storing the row indices of nonzero entries in each column.
- **Lx**. An array of length $|\mathcal{L}|$ for storing the actual nonzero values within each column.
- **Lp**. An array of length **ln**, which contains the indices within the other two arrays (**Li** and **Lx**) where each column begins.
- **Lnz**. An array of length **ln** containing the number of nonzeros in each column.

An example is shown in Figure 3-2. In this Figure, **Lp** stores the indices within **Li** and **Lp** where each column begins. **Li** and **Lp** have a one-to-one correspondence. **Li**

[1] Asymptotic complexity notation O is defined in [9]. We write $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that to the right of n_0, the value of f(n) always lies on or below cg(n), that is, $0 \le f(n) \le cg(n)$ for all $n > n_0$.

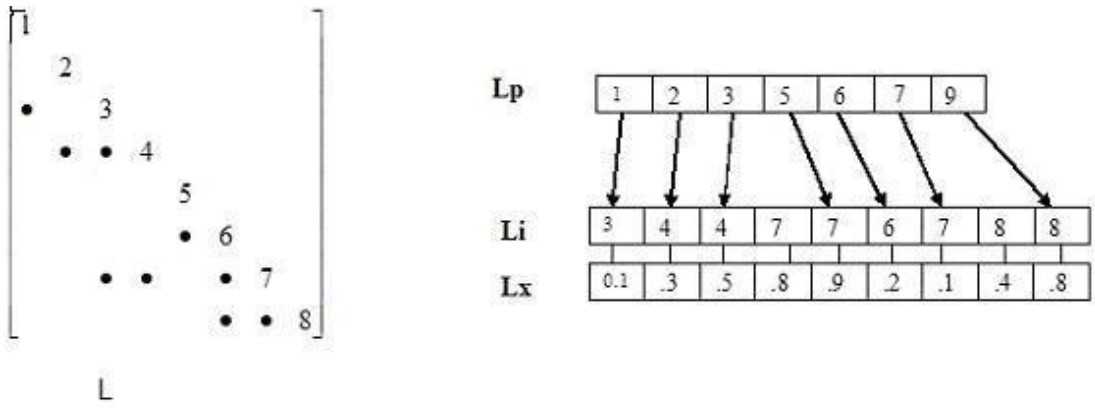

Figure 3-2 Example of data structures (**Lx**,**Li**,**Lp**) for storing **L**

contains the row indices of all nonzeros in each column of **L** and **Lx** contains the actual values for the nonzeros. For example, column 3 starts at index 3 (`Lp [3] = 3`) in arrays **Ai** and **Ax**.

Note that the diagonal is not stored in **L**, but in a separate array **D** of length **ln**. The data structures used for storing matrix **A** is similar to that of **L**, and it contains the diagonal. These are **Ai**, **Ax**, **Ap**, and **Anz**. Although $\mathbf{AA}^{\mathbf{T}}$ is factorized, storage space can be saved by just storing **A** instead of $\mathbf{AA}^{\mathbf{T}}$.

Data Structure for Storing Elimination Tree

The elimination tree is stored in a data structure called **Parent** which is a single dimensional array of length **ln**. Each entry in the **Parent** array contains the parent node for that particular node.

The **Parent** map for the elimination tree shown in Figure 2-3 is shown below:

3	4	4	7	6	7	8	0

The parent of node 1 is Parent [1] = 3, the parent of node 2 is Parent [2] = 4 and so on. Node 8 has no parent and so is the root of the elimination tree.

CHAPTER 4
COLUMN-COLUMN CHOLESKY FACTORIZATION

This is the simplicial form of *numerical factorization* and is the base on which further improvement in performance can be achieved. The column-column Cholesky factorization is an implementation of the algorithm for $\mathbf{A=LDL^T}$ factorization described in Chapter 2. The main kernel of the factorization, that is, the `cmod (j,k)` and `cdiv (k)` operations are column-based. The algorithm and implementation are discussed in the next sections.

Algorithm

George and Liu [5] discuss this algorithm in their book, in the SPARSPAK code for sparse Cholesky factorization. The algorithm can be expressed in terms of the following two major subtasks:

- **cmod (j,k)**. modification of column j by a multiple of column k, k < j. In algorithm shown in Figure 2-2, the `cmod (j,k)` corresponds to the operation of `A(j+1:n,j) - L (j+1:n,1:j-1) v(1:j-1)` where k is in the range 1:j-1.
- **cdiv (j)**. division of column j by a scalar. In Figure 2-2, this corresponds to the operation of `cmod (j,k)/d(j)`, that is, the division of the result of `cmod (j,k)` by `d(j)`.

A high level description of the left-looking column-column algorithm is shown in Figure 4-1. In this figure, it can be seen that `cmod (j,k)` is used to update target column j with all those columns to the left of j that have a nonzero value in row j. So for each target column j, a list of all columns k called **List** is maintained such $\mathcal{L}_{j,k} \neq 0$, rather than checking each column during the numerical computation. To construct the **List**, symbolic factor $\mathcal{L}$ is used, that is, the pattern of **L** stored in **Li** is used.

```
for each col j in L do
  for k such that ℒ_{j,k} ≠ 0 do
      Cmod (j,k)
  end for
  Cdiv (j)
end for
```

Figure 4-1 Algorithm1 column-column Cholesky factorization

The algorithm is now modified to accommodate the **List** as shown in Figure 4-2.

```
for each col j in L do
   for each k in List (j) do
      Cmod (j,k)
      add k to List ( next (j,k))
   end for
   Cdiv (j)
   add j to List (Π(j))
end for
```

Figure 4-2 Algorithm2 column-column Cholesky factorization

The algorithm in Figure 4-2, contains two new data structures. These are as follows:

- **List (j)**. $\{ k : \mathcal{L}_{j,k} \neq 0 \}$
- **next (j,k)**. min $\{i: i \in \mathcal{L}_k \wedge i > j\}$, that is, row index in column k that is immediately greater than j. This is used to place the column k in the list of the next column that it is going to update.

Also, now that the pattern of **L** is known from the symbolic factorization, the computation can be restricted to only the predicted nonzero locations of **L**. This reduces the number of floating point of operations as computation for the predicted zero locations is eliminated. Maintaining the **List** takes O ($|\mathcal{L}|$) time. Also, the entire algorithm can be done in O ($\Sigma |\mathcal{L}_j|^2$) time. This is clear from the algorithm in Chapter 2, where it can be seen that for each column of **L**, every nonzero entry is multiplied by every other nonzero entry in that column so as to update all the other columns of **L** that correspond to the nonzero row indices within the updating column.

Implementation Details

There are two data structures **First** and **List**, and two methods **cmod** and **cdiv** to be discussed.

First

First is an array of size equal to the number of columns of **L**, that is, **ln**. For each column k, `First [k]` contains the index within **Li** of the first entry in column k that can be used to update target columns; that is, if j is the target column to be updated then `Li [First [k]]` should be equal to j. After updating a target column, `First [k]` is incremented by 1 to point to the next nonzero entry in column k, that is, the next column that column k is going to update.

List (j)

Two arrays are used to implement this list, **Link** and the **First** arrays. **Link** holds the list of all the columns that are to affect the target column j, that is, all those columns that have a nonzero in row j. **Link** is an array of size equal to the number of columns in **L**. All entries of **Link** are initially set to EMPTY. `Link [j]` contains the head of the list for column j. Since a left-looking approach is used, each column j is updated by columns k where k < j. In terms of **Link** data structure, this means that when column j is to be computed, only the entries in **Link** following `Link [j]` will be used to store the heads of the lists. The columns before j will already have been computed and so no longer require to stop their lists. So the entries before `Link [j]` are then used to store the remainder of the lists for the columns after j. Thus a single data structure can be used to store lists for all columns of **L**, thereby conserving space. The rest of the list is stored by having each entry in **Link** contain the next element of the list. For example, if the list for column j

was a set of columns $\{k_1, k_2, k_3\}$ where k_1 is the head of the list, then the entire list would be stored in **Link** as,

$$\texttt{Link [j]} = k_1,$$
$$\texttt{Link [}k_1\texttt{]} = k_2,$$
$$\texttt{Link [}k_2\texttt{]} = k_3, \text{ and}$$
$$\texttt{Link [}k_3\texttt{]} = \texttt{EMPTY}.$$

Note that $k_1 < j$, $k_2 < j$, and $k_3 < j$.

Each column within the list of column j, after updating column j, is then stored in the list of the next column that it is going to update. The algorithm for creating and accessing the list in C-style notation is shown in Figure 4-3.

```
for ( k = Link [j] ;  k != EMPTY ; k = nextk)
{
    nextk = Link [k] ;

    update j with k ;
    First [k] ++;

    /* put k on the list of next j  */
    nextj = Li [First [k]];
    Link [k] = Link [nextj];
    Link [nextj] = k;
}
```

Figure 4-3 Algorithm for creating and accessing `List (j)`

Cmod (j,k)

This is the innermost kernel of the factorization where a multiple of column k is used to update target column j. A work vector **W** of size equal to the number of columns in **L**, that is, **ln**, is used to accumulate the updates of all the columns k that update target column j. The updates scattered in **W** are then gathered into **Lx** at the appropriate location for column j. As seen in the algorithm shown in Figure 2-2, each entry in column k is first multiplied by a scalar **v** where `v = L (j,k) * D (k) = l_jk_times_d_k.` The

result is then accumulated into the **W** at the location corresponding to the row index of that nonzero entry. The **cmod** operation in C-style notation is shown in Figure 4-4.

```
p = First [k];
p2 = Lp [k] + Lnz [k];

for (; p < p2; p++)
{
   W [Li [p]] = W [Li [p]] - Lx [p] * l_jk_times_d_k;
}
```

Figure 4-4 Scattering for `cmod (j,k)` into **W**

Cdiv (j)

As shown in the algorithm in Figure 2-2, the updates scattered into **W** are divided by the diagonal entry of column j which is stored in **D**. The results are then gathered into **Lx** as shown in Figure 4-5 which is in C-style notation.

```
p = First [j];
p2 = p + Lnz [j];
for (; p < p2;p++)
{
  i = Li [p] ;
  Lx [p] = W [i] / D [j];
  W[i] = 0.0 ;
}
```

Figure 4-5 Gathering **W** into **Lx**

As seen in Figure 4-5, the row index of each nonzero in column j is used to index into **W**. The corresponding value in **W** is then stored into **Lx** at the appropriate location for column j. Also by resetting the corresponding entry in **W** to zero, only used locations are reinitialized. Entire **W** does not have to be reinitialized to zero.

Analysis of Column-Column Factorization

This is the simplicial algorithm where a single column updates another column. Thus indirect referencing has to be done for every such update. Also once a column is loaded into the cache, it is used to update only a single column. However there may be

other columns which it can update. In such cases, it will have to be reloaded again into the cache. This can affect performance.

Implementation of Column-Column Numerical Factorization

See Appendix B for the implementation of the column-column Numerical factorization. This method has been implemented in C and was run in MATLAB using mexfunction interfaces.

CHAPTER 5
SUPERNODAL THEORY

This chapter contains the definition of a supernode and an algorithm for finding the supernodal elimination tree, which is used to compute the Cholesky factor **L** in supernode-column and supernode-supernode factorization methods described in Chapter 6 and Chapter 7 respectively.

Supernode Definition

In this thesis, a supernode is defined as a set of columns that have a similar sparsity structure and that may or may not be contiguous. The supernodal approach for solving sparse Cholesky factorization was discussed by several researchers [1,9–12]. They defined supernodes to be a set of contiguous columns that have the same sparsity structure. The condition that the columns of a supernode be contiguous can be a bottleneck for various applications where the matrix **A** is continuously being modified. An example of such an application is the Linear Program Dual Active Set Algorithm (LP DASA) [7]. So in this thesis, a supernode is given the flexibility to contain non-contiguous columns.

Liu, Ng and Peyton [9] define a supernode as a contiguous block of columns of **L**, {p,p+1,….,p+q-1}, such that, Struct ($L_{*,p}$) = Struct ($L_{*,p+q-1}$) $\cup$ {p+1,…,p+q-1} where Struct basically means the nonzero pattern of column p. In the notation used in this thesis, $\mathcal{L}_j$ = Struct ($L_{*,j}$). It is easy to show that for $p \le i \le p+q-2$, by induction. Thus, the columns of the supernode have a dense diagonal block and have identical structure below the row

p+q-1. This definition is extended in this thesis to allow columns within the supernode to be non-contiguous.

An equivalent definition for a supernode is a collection of columns within **L**, where for every parent-child relationship between the column-nodes the following relation holds:

$$|\mathcal{L}_{parent}| = |\mathcal{L}_{child}| - 1 \quad \text{or} \quad \mathcal{L}_c \setminus \{c\} = \mathcal{L}_p$$

This relation says that a child column-node will have one extra non-nonzero entry than its parent, the extra entry being the diagonal entry of that child column-node. Thus a supernode contains a subtree, wherein every parent-child pair of nodes hold the above relation. Also, the above relation is made obvious from the symbolic factorization where the pattern of a parent node is obtained by taking a union of the patterns corresponding to its children nodes. Thus the pattern of every descendant of a particular node is a subset of the pattern of that node. So if follows that for every parent-child pair in a supernode, the relation $\mathcal{L}_c \setminus \{c\} \subseteq \mathcal{L}_p$ would hold.

The main advantage of supernodes is that since the supernodal columns have a similar sparsity structure, indirect referencing is greatly reduced when updating the target column j. Note that if any one column within a supernode affects column j, then all the columns within that supernode will also affect j. So, rather than doing indirect referencing for every column within the supernode that affects j, the updates for the entire supernode can be accumulated into a single dense work vector and then a single indirect reference can be done for that supernode to the target column j. This brings about a significant performance gain as compared to the simplicial column-column numerical factorization method.

Size-Limit of Supernode

A supernode that is too large in size might not fit into cache causing cache misses. This can affect performance. To avoid this, supernode size is limited to a certain maximum number of columns. In this thesis, this maximum limit is set to 32 columns. If a supernode contains more than 32 columns, then it is further split into supernodes each containing no more than 32 columns.

Supernodal Elimination Tree

Supernodal elimination tree is a tree that determines the order in which the supernodes must be computed. If the *elimination tree* shows the column dependencies, then the *supernodal elimination tree* determines the supernodal dependencies.

Every parent-child pair within a supernode holds the relation $|\mathcal{L}_{\text{parent}}| = |\mathcal{L}_{\text{child}}| - 1$. Now a parent may have more than one child that satisfies this relationship. Only one of such children can be chosen to be a part of the supernode containing the parent. Adding more than one child means including siblings which will obviously violate the required parent-child condition of the columns within the supernode.

The elimination tree is now modified by combining the parent-child nodes that hold the supernodal relation into a single node. By doing so, the elimination tree is transformed into the supernodal elimination tree where every node is now a supernode consisting of one or more nodes. A supernode consisting of just one node is a called singleton supernode. The order of elimination of the nodes within the elimination tree must be maintained and not disturbed within the newly created supernodal elimination tree. That is before computing a supernode, all the children of all the nodes within that supernode must be computed first. It is for this reason, that when two or more children satisfy the supernodal criteria, the higher numbered child-node is chosen to be a part of

the supernode that contains the parent-node. This preserves the order of elimination within the supernodal elimination tree.

The algorithm for finding the supernodal elimination tree is shown in Figure 5-1.

```
for c = 1 to n
  j = Π(c);
  if ( |L_c| - 1 = |L_j| ) then
     Schild (j) = c;
  end if
end for

mark (1..n) = false ;

for j = n downto 1
  if j  is unmarked then
     no of supernodes ++;
     add j to supernode;
     while Schild (j) is unmarked  and SupernodeSize < maxSize
         j = Schild (j);
         mark (j) = true;
         add j to supernode;
     end while
  end if
end for
```

Figure 5-1 Algorithm to find supernodal elimination tree.

In Figure 5-1, in first loop iteration, a node's largest numbered child that has a supernodal relationship with it is determined. This is to maintain the correct order for elimination as discussed above. In the second loop iteration, the list of nodes is traversed from top to down, iteratively adding a node and its superchild to the collection, till it no longer has a superchild. The collection then forms a complete supernode. Initially all nodes are unmarked and get marked when added to a supernode. This algorithm takes O(ln) time where **ln** is the number of columns of **L**. Note that, the number of supernodes can never be greater than **ln**.

An example of Cholesky factor **L** and its supernodes is shown in Figure 5-2. In this figure, supernode 1 is a singleton supernode. Note that both children of node 10, that

is node 8 and 9, satisfy the supernodal relation. But the larger child, that is, node 9, is chosen to be a part of the supernode.

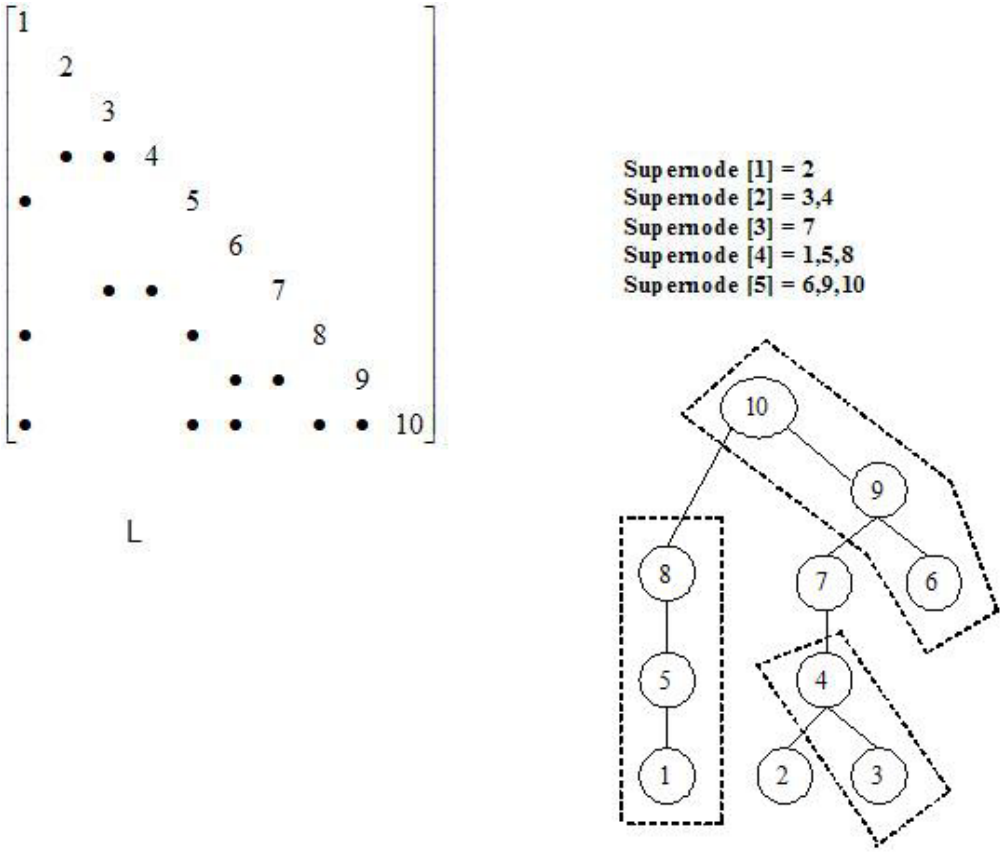

Figure 5-2 Example of Cholesky factor **L** and its supernodal elimination tree

Two arrays, **Slist** and **Sp** are used to store the supernodal elimination tree . Both are single dimensional arrays of size equal to the number of columns in Cholesky factor **L**, that is, **ln**. **Slist** stores supernodes sequentially in the non-decreasing order. **Sp** is the supernode pointer list which contains indices within **Slist** where each supernode begins. It would contain entries equal to the number of supernodes.

The supernodal elimination tree shown in Figure 5-2 is stored in **Slist** and **Sp** as shown in Figure 5-3. The size of a supernode **J** is equal to (`Sp [J+1] – Sp [J]`). The columns within supernode **J** can be accessed as shown in Figure 5-4.

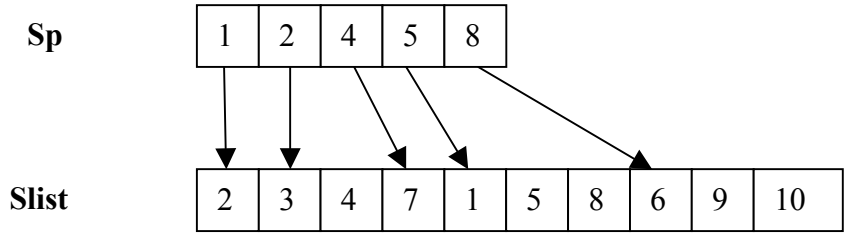

Figure 5-3 Example of **Slist** and **Sp**

The first column within a supernode is called the representative column of that supernode as it is the largest column within that supernode. The pattern of a supernode is given by the pattern of the representative column. The representative column of supernode **J** is `Slist [Sp [J]]` and is also called **Head_of_J**. Similarly, the representative column of supernode **K** is called **Head_of_K**.

```
for (j = Sp [J] ; j < Sp [J+1] ; j++)
    Slist [j] ;
end for
```

Figure 5-4 Accessing columns within supernode **J**

CHAPTER 6
SUPERNODE-COLUMN CHOLESKY FACTORIZATION

This chapter contains the algorithm and implementation details of the supernode-column numerical Cholesky factorization. This method uses supernodes which are discussed in Chapter 5. Supernodes reduce the amount of indirect referencing and improve performance. The main kernel of the factorization, that is, the **cmod** operation is supernode-column based where a supernode updates the target column. The algorithm and implementation are discussed in the next sections.

Algorithm

The notation for supernodes are bold capital letters **S**,**J** and **K**, which represent the set of columns contained in the supernode. Let j be any column in supernode **J** and k be any column in supernode **K**. The algorithm can be expressed in terms of the following major subtasks:

- **cmod (j**,**K)**. modification of column j by a multiple of all columns k within supernode **K**. Note that, k < j. In algorithm shown in Figure 2-2, the `cmod (j,K)` corresponds to the operation of `A(j+1:n,j) - L (j+1:n,1:j-1) v(1:j-1)`. Here, `L(j+1:n,1:j-1)` is replaced with `L (j+1:n, 1: J-1)` . **K** is in the range 1:J-1.
- **cmod (j**,**J**). modification of column j by a multiple of all of the columns to the left of j in supernode **J**. In the algorithm shown in Figure 2-2, the `cmod (j,J)` corresponds to the operation of `A(j+1:n,j) - L (j+1:n,1:j-1) v(1:j-1)`. Here, `L(j+1:n,1:j-1)` is replaced with `L (j+1:n, J (1:j-1))` .
- **cdiv (j)**. division of column j by a scalar. In Figure 2-2, this corresponds to the operation of `cmod (j,k)/d(j)` , that is, the division of the result of `cmod(j,k)` by `d(j)`.

A high level description of the left-looking supernode-column algorithm is shown in Figure 6-1.

```
for J = 1 to N do
   for j ∈ J do
      for K such that ℒ_{j,K} ≠ 0 do
         cmod (j,K)
      end for
      cmod (j,J)
      cdiv (j)
   end for
end for
```

Figure 6-1 Algorithm1 supernode-column Cholesky factorization

In Figure 6-1, it can be seen that `cmod (j,K)` is used to update target column j with all those supernodes to the left of j that have a nonzero value in row j. So for each target column j, a list of all supernodes called **List** is maintained such $\mathcal{L}_{j,K} \neq 0$. To construct the **List**, the symbolic factor $\mathcal{L}$ is used, that is, the pattern of **L** stored in **Li** is used. This **List** is created and maintained in linear time. The algorithm is now modified to accommodate the **List** as shown in Figure 6-2:

```
for J = 1 to N do
   for j ∈ J do
       for K ∈ List (j) do
           cmod (j,K)
           add K to List (next (j,K))
       end for
       cmod (j,J)
       cdiv (j)
   end for
   add J to List (next (jlast,J))
end for
```

Figure 6-2 Algorithm2 supernode-column Cholesky factorization

The algorithm shown in Figure 6-2, contains two new data structures. These are as follows:

- **List (j)**. { **K**: $\mathcal{L}_{j,K} \neq 0$ }, it is a list of supernodes.
- **next (j,K)**. min {i: i ϵ $\mathcal{L}_K \wedge$ i > j}, that is, row index in pattern of supernode **K** that is immediately greater than j. This is used to place the supernode **K** in the list of the next column that it is going to update. Note that `jlast` is the last column within supernode **J**.

Also, now that the pattern of **L** is known from the symbolic factorization, the computation is restricted to only the predicted nonzero locations of **L**, thus reducing the number of floating point of operations. Maintaining the **List** takes O ($|\mathcal{L}|$) time. The entire algorithm can be done in O ($\Sigma\ |\mathcal{L}_j|^2$) time.

Implementation Details

There are two data structures **First** and **List**, and two methods **cmod** and **cdiv** to be discussed.

First

First is an array of size equal to the number of supernodes of **L**. For each supernode **K**, `First [K]` contains the index within **Li** of the first entry in the representative column of **K** (**Head_of_K**) that can be used to update target columns; that is, if j is the target column to be updated then `Li [First [K]]` should be equal to j. After updating a target column, `First [K]` is incremented by 1 to point to the next nonzero entry in column **Head_of_K**, that is, the next column that the supernode **K** is going to update.

List (j)

Three arrays are used to implement this list, **Link**, **Lnext** and the **First** arrays. **Link** contains the heads of lists of all the supernodes that are to affect the target column j, that is, lists of all those supernodes that have a nonzero in row j. Only the heads are stored in **Link**. The remainder of the list is stored in **Lnext**. **Link** is an array of size equal to the number of columns in **L**, while **Lnext** is of size equal to the number of supernodes in **L**. All entries of **Link** and **Lnext** are initially set to EMPTY. `Link [j]` contains the head of the list for column j. The rest of the list is stored by having each entry in **Lnext** contain the next element of the list. For example, if the list for column j was a set of

supernodes $\{K_1, K_2, K_3\}$ where K_1 is the head of the list, then the entire list would be stored in **Link** and **Lnext** as,

$$\texttt{Link [j]} = K_1,$$

$$\texttt{Lnext [}K_1\texttt{]} = K_2,$$

$$\texttt{Lnext [}K_2\texttt{]} = K_3, \text{ and}$$

$$\texttt{Lnext [}K_3\texttt{]} = \texttt{EMPTY}.$$

Each supernode within the list of column j, after updating column j, is then stored in the list of the next column that it is going to update. The algorithm for creating and accessing the list in C-style notation is shown in Figure 6-3.

```
for ( K = Link [j] ;  K != EMPTY ; K = nextK)
{
    nextK = Lnext [K] ;

    update j with K;
    First [K] ++;

    /* put K on the list of next j  */
    nextj = Li [First [K]];
    Lnext [K] = Link [nextj];
    Link [nextj] = K;
}
```

Figure 6-3 Algorithm for creating and accessing `List (j)`

Note that two arrays are used to store the **List**, that is, **Link** and **Lnext**, compared to a single array **Link** for the column-column Cholesky factorization. The reason is that a supernode contains a collection of columns which may or may not be contiguous. The elimination is by the sequential order of supernodes, compared to the sequential elimination of columns as in the column-column method. So in the supernode-column method, it is unlikely that when column j is the target column, all columns before j would have been computed, leaving the entries in **Link** before `Link [j]` reusable for storing the remainder of lists. For this purpose, only the heads of lists are stored in **Link** and the

remainder of the lists in the **Lnext**. This is an overhead that is not present in the column-column numerical factorization method.

Cmod (j,K)

This is the innermost kernel of the factorization where multiples of the columns k within supernode **K** is used to update target column j. A dense work vector **Wx** of size equal to the length of the supernode, is used to accumulate the updates of all the columns with the supernode **K**. This dense operation exploits the similarity of the sparsity structure of the columns within the supernode. The results from **Wx** are then scattered into work vector **W** of size equal to the number of columns in **L**, that is, **ln**. **W** is used to accumulate the updates of all the supernodes that update target column j. The updates scattered in **W** are then gathered into **Lx** at the appropriate locations for column j. Thus the amount of indirect referencing to column j is reduced to just one for each supernode. This improves performance by eliminating the time required to do indirect referencing for each column within the supernode. As seen in the algorithm shown in Figure 2-2, each entry in column k within supernode **K** is first multiplied by a scalar **v** where

```
v = L (j,k) * D (k) = l_jk_times_d_k
```

The **cmod** operation in C-style notation is shown in Figure 6-4.

If the size of the supernode **K** is one, that is, if it is a singleton supernode, then the overhead of using dense work vector **Wx** is avoided and the updates are directly stored into the work vector **W**. This is nothing but a column-column update. This is a special case of the supernode–column method when the size of the supernode is one.

Cmod (j,J)

The target column j within supernode **J** is updated by the columns within the supernode **J** itself that are to the left of column j. Since the diagonal block of a supernode

is always dense, all columns to the left of j within supernode **J** will update column j. Now since all columns share the same sparsity structure, so no indirect referencing is required.

```
for k ∈ K do
     Wx = Wx + Lx_k * l_jk_times _d_k;
end for

/* scatter  Wx into W */
p = First [K];
p2 = p + Lnz [Head of K];
for (i = 0; p < p2;p++,i++)
     W [Li [p]] = W [Li [p]] - Wx [i];
end for

/* gather  W into Lx */
p = Lp [j];
p2 = p + Lnz [j];
for ( ; p < p2 ; p++)
    i = Li [p];
    Lx [p] = W [i];
    W [i] = 0.0;
end for
```

Figure 6-4 Algorithm for `cmod (j,K)`

So, rather than using the work vector **W**, all updates are done directly on **Lx** at the appropriate locations for column j. For this, current results from the work vector **W** need to be gathered into **Lx** first and this is done at the end of the `cmod (j,K)` stage.

The algorithm for `cmod (j,J)` in C-style notation is shown in Figure6-5.

```
for j1 ∈ J and j1 < j
     Lx_j  = Lx_j  - Lx_j1 * l_jk_times_d_k;
end for
```

Figure 6-5 Algorithm for `cmod (j,J)`

Cdiv (j)

As shown in the algorithm in Figure 2-2, the updates for column j that are gathered into **Lx** are divided by the diagonal entry of column j which is stored in **D**.

The algorithm for `cdiv (j)` in C-style notation is shown in Figure 6-6.

```
p = Lp [j];
p2 = p + Lnz [j];
for (; p < p2; p++)
     Lx [p]  = Lx [p]/D [j];
end for
```

Figure 6-6 Algorithm for `cdiv(j)`

BLAS

BLAS stands for Basic Linear Algebra Subroutines. BLAS2 are used for doing matrix-vector operations and scaling operations over a block of contiguous columns. Since the supernodes contain non-contiguous columns, the BLAS cannot be used to do matrix-vector operations. BLAS1 for vector operations can be used such as the **daxpy_** and the **dscal_** .

Analysis of Supernode-Column Cholesky Factorization

This method reduces the amount of indirect referencing significantly by exploiting the similarity of the sparsity structure of the columns within each supernode. This is clear by the usage of **Wx** to accumulate the updates of all columns within the supernode **K** and then updating the target column j. However, there is more room for improvement in the area of cache reusability. When supernode **K** is loaded into cache, it is used to update only a single column j within supernode **J** and after that it may be removed from the cache. However it may also update some other columns within the supernode **J** and would have to be reloaded into cache again. This is an area where further improvement in performance can be achieved.

CHAPTER 7
SUPERNODE-SUPERNODE CHOLESKY FACTORIZATION

This chapter contains the algorithm and implementation details of the supernode-supernode numerical Cholesky factorization. This method uses supernodes which are discussed in Chapter 5. Supernodes reduce the amount of indirect referencing and improve cache behavior. The main kernel of the factorization, that is, the **cmod** operation is supernode-supernode based where a supernode updates the target supernode. The algorithm and implementation are discussed in the next sections.

Algorithm

The notation for supernodes are capital letters **S**,**J** and **K**. Let j be any column in supernode **J** and k be any column in supernode **K**. Also, note that the columns within the supernode may be non-contiguous.The algorithm can be expressed in terms of the following major subtasks:

- **cmod (J**,**K)** . modification of a collection of columns within supernode **J** by a multiple of all the columns within supernode **K**. Note that **K** < **J**. The intersection set containing the columns in supernode **J** to be updated by supernode **K** must be determined. This set is obtained by taking an intersection of the row indices of supernode **K** with the columns of supernode **J**. Also, performance can be improved by using block matrix methods to do the computation of `cmod (J,K)`.
- **cdiv (J)**. Each column within supernode **J** has to be updated by the columns to the left of it and then it has to be scaled. This is nothing but a Cholesky factorization of the diagonal block of supernode **J**. Also, since the diagonal block is dense, the algorithm shown in Figure 2-2 for the dense $\mathbf{A=LDL^T}$ Cholesky factorization can be used.

A high level description of the left-looking supernode-supernode numerical factorization algorithm is shown in Figure 7-1.

```
for J = 1 to N do
   for K such that ℒJ,K ≠ 0 do
       cmod (J,K)
   end for
   cdiv (J)
end for
```

Figure 7-1 Algorithm1 supernode-supernode Cholesky factorization

In Figure 7-1, it can be seen that `cmod (J,K)` is used to update target supernode **J** with all those supernodes to the left of **J** that have a nonzero value in any of the rows corresponding to columns of **J**. So for each target supernode **J**, a list of all supernodes called **List** is maintained such $\mathcal{L}_{J,K} \neq 0$. To construct the **List**, the symbolic factor $\mathcal{L}$ is used, that is, the pattern of **L** stored in **Li** is used. The algorithm is now modified to accommodate the **List** as shown in Figure 7-2.

```
for J = 1 to N do
   for K ∈ List (J) do
      cmod (J,K)
      add K to List (next (jlast,K))
   end for
   cdiv (J)
   add J to List (next (jlast,J))
end for
```

Figure 7-2 Algorithm2 supernode-supernode Cholesky factorization

The algorithm in Figure 7-2, contains two new structures. These are as follows:

- **List (J)**. $\{ K : \mathcal{L}_{J,K} \neq 0 \}$, it is a list of supernodes for supernode **J**. It is a union of the supernode lists of all the columns within the supernode **J**.
- **next (j,K)**. $J' \ni (\min \{i: i \in \mathcal{L}_K \wedge i > j\} \in J')$. This returns the supernode that contains the column corresponding to the row index in supernode **K** that is immediately greater than j. For this, a mapping between the columns and the supernodes is required. This map is called the **Col_to_Supernode_Map**. `next(j,K)` uses the **Col_to_Supernode_Map** for getting the supernode that contains the next target column to be updated by supernode **K**. Note that `jlast` is the last column within supernode **J**.

Also, now that the pattern of **L** is known from the symbolic factorization, the computation can be restricted to only the predicted nonzero locations of **L**, thus reducing

the number of floating point of operations. Maintaining the **List** takes O ($|\mathcal{L}|$) time. The entire algorithm can be done in O ($\Sigma\ |\mathcal{L}_j|^2$) time.

The work storage used for the `cmod (J,K)` is different from the work vector used in the earlier two methods of numerical factorization. The work storage is no longer a single dimensional array, but a two dimensional work array **Wx** of size *m*–by-*n*, where *m* is the maximum length of a column in the Cholesky factor **L** and *n* is the maximum width of the supernode. Now for `cmod (J,K)`, **Wx** is used as a dense two dimensional work array. Only a portion of **Wx** of size *mm*-by-*nn* will be used for each supernode **J** where *mm* is the length of the representative column of supernode **J** and *nn* is the width of the supernode **J**. Supernode **J** and **K** need to be mapped to **Wx**. For this, a data structure called **Map** is used.

The algorithm is further modified as shown in Figure7.3.

```
Wx [max_column_length, max_supernode_size] = 0
for J = 1 to N do
    scatter J's relative indices into Map
    for K ∈ List (J) do
       compute intersection set of columns of J to be updated by K;
       Wx = cmod (J,K) ; (uses Map to assemble into Wx)
       add K to List (next (jlast,K))
    end for
    cdiv (J)
    add J to List (next (jlast,J))
end for
```

Figure 7-3 Algorithm3 supernode-supernode Cholesky factorization

Implementation Details

There are four data structures that are used. These are **First**, **Col_to_Snode_Map**, **List** and **Map**. There are two methods that are implemented. These are **cmod** and **cdiv**. The implementation details of each of these data structures and the methods are discussed in detail in this section.

Wx

This is the dense work array for storing the updates to supernode **J**. It is two-dimensional, but is stored linearly by rows. The entry at `Wx [i] [j]` can be accessed using `Wx [ i * WxCol + j]` where `WxCol` is the width of the supernode **J**.

First

First is an array of size equal to the number of supernodes of **L**. For each supernode **K**, `First [K]` contains the index within **Li** of the first entry in the representative column of **K** (**Head_of_K**) that can be used to update target columns; that is, if j is the target column to be updated then `Li [First [K]]` should be equal to j.

After updating a target column, `First [K]` is incremented by 1 to point to the next nonzero entry in column **Head_of_K**, that is, the entry corresponding to the next column that the supernode **K** is going to update.

Col_to_Supernode_Map

This is an integer array of size equal to number of columns in **L**. It maps columns to the supernodes that contain them. The mapping algorithm is shown in Figure 7-4.

```
for J = 1 to Number_of_Supernodes
   for j ∈ J
      Col_to_Snode_Map [j] = J
   end for
end for
```

Figure 7-4 Algorithm to construct **Col_to_Supernode_Map**

List (J)

Two arrays are used to implement this list, **Link** and the **First** arrays. **Link** holds the list of all the supernodes that are to affect the target supernode **J**, that is, all those supernodes that have a nonzero in any of the rows corresponding to the columns of supernode **J**. **Link** is an array of maximum size equal to the number of columns in **L**. All

entries of **Link** are initially set to EMPTY. `Link [J]` contains the head of the list for supernode **J**. Since a left-looking approach is used, each supernode **J** is updated by supernodes **K** where **K**< **J**. In terms of **Link** data structure, this means that when supernode **J** is to be computed, only the entries in **Link** following `Link [J]` will be used to store the heads of the lists. The supernodes before **J** will already have been computed and so no longer require storing their lists. So the entries before `Link [J]` are then used to store remainder of the lists for the supernodes after **J**. Thus a single data structure can be used to store lists for all supernodes of **L**, thereby conserving space. The rest of the list is stored by having each entry in **Link** contain the next element of the list. For example, if the list for supernode **J** was a set of supernodes $\{K_1, K_2, K_3\}$ where K_1 is the head of the list, then the entire list would be stored in **Link** as,

$$\text{Link } [J] = K_1,$$
$$\text{Link } [K_1] = K_2,$$
$$\text{Link } [K_2] = K_3, \text{ and}$$
$$\text{Link } [K_3] = \text{EMPTY}.$$

Note that $K_1 < j$, $K_2 < j$, and $K_3 < j$.

Each supernode within the list of supernode **J**, after updating supernode **J**, is then stored in the list of the supernode that contains the next column that it is going to update. The mapping of the column to the supernode is done using the **Col_to_Supernode_Map**. The algorithm for creating and accessing the list in C-style notation is shown in Figure 7-5. In this figure, `nextj` is the next column (within Cholesky factor **L**) to be updated by supernode **K** and `nextJ` is the supernode that contains `nextj`. The supernode **K** after updating supernode **J** is then placed in the list for supernode `nextJ`. In this manner the list of supernodes for each target supernode is constructed dynamically.

```
for ( K = Link [J] ;  K != EMPTY ; K = nextK)
{
   nextK = Link [K] ;

   update J with K;
   First [K]++;

   /* put K on the list of nextJ */
   nextj = Li [First [K]];
   nextJ = Col_to_Supernode_Map [nextj];
   Link [K] = Link [nextJ];
   Link [nextJ] = K;
}
```

Figure 7-5 Algorithm to create and access `List (J)`

Map

This is used to map supernode **J** and supernode **K** to the dense work array **Wx**. Map is a single dimensional integer array that maps the nonzero row indices of the representative column of supernode **J** (**Head_of_J**) to the first *mm* rows of **Wx**, where *mm* is the number of nonzeros in **Head_of_J**. It also maps the columns of supernode **J** to the first *nn* columns of **Wx**, where *nn* is the width of supernode **J**. As the pattern of supernode **K** is a subset of the pattern for supernode **J**, **Map** can be used to map supernode **K** to the work vector **Wx**. The algorithm for computing **Map** in C-style notation is shown in Figure7-6.

```
p = Lp [Head_of_J];
p2 = p + Lnz [Head_of_J];

Map [Head_of_J] = 0;
for (i = 1; p < p2; p++,i++)
  Map [Li [p]] = i * WxCol;
end for
```

Figure 7-6 Algorithm to construct **Map**

In Figure7-6, `WxCol` is the width of supernode **J** and is used to traverse through the rows as **Wx** is linearly stored by rows. The boundaries for the nonzero pattern of a column are represented by `p` and `p2`.

Cmod (J,K)

This is the outermost kernel of the supernode-supernode numerical Cholesky factorization. A two dimensional work vector **Wx** of size p–by-q, where p is the maximum length of a column in the Cholesky factor **L** and q is the maximum width of a supernode. Only a portion of **Wx** of size *pp*-by-*qq* will be used for any given **cmod (J,K)** where *pp* is the length of the representative column of supernode **J** (**Head_of_J**) and *qq* is the width of the supernode **J**. The supernodes **J** and **K** are mapped to **Wx** by the **Map**. The work vector **Wx** is used to accumulate the updates of all the supernodes **K** that update target supernode **J**. The results from **Wx** are eventually gathered into **Lx** into the appropriate locations for all the columns j within the supernode **J**.

Intersection set

This set contains the columns within supernode **J** that are to be updated by supernode **K**. It is the intersection between the row indices of supernode **K** with the columns within supernode **J**. The algorithm for finding the intersection set in C-style notation is shown in Figure 7-7.

```
jlast = Slist [Spp [J+1] -1];
p = First [k];
p2 = p + Lnz [Head_of_K];
for (i=0; p<p2 && Li [p] <= jlast; p++,i++)
{
    Intersect [i] = Li [p];
}
```

Figure 7-7 Algorithm to construct Intersection set

In Figure 7-7, it can be seen that it is not required to check the entire row indices of supernode **K** for intersection. This is because as the row indices of supernode **K** are in ascending order, there would be no intersection for row indices that are greater than `jlast`.

Block matrix computation of cmod (J,K)

Both supernode **J** and supernode **K** are blocks containing more than one column. So block matrix computation is employed to improve performance by better cache reusability. The benefit of using block matrix computation is a better ratio of the cache misses to floating point operations. The supernode **J** is trapezoidal in shape. So for the diagonal part, the computation is done as in the supernode-column method. For the remaining rectangular block of supernode **J**, the block matrix method is used.

The method **mMKN** is called to do the block matrix computation. The arguments passed to this method are as follows:

- **M**. `Lp [Head_of_K] + Lnz [Head_of_K] – First [K]`. This is the length of the block of supernode **K** that is used to modify the rectangular block of supernode **J**
- **K**. `Sp [K+1] – Sp [K]`. This is the width of supernode **K**.
- **N**. This is the length of the intersection set.
- **pk**. An array of double pointers that point to the columns within supernode **K**. The array size is equal to the width of supernode **K**. The pointers are used to reduce the amount of indirect referencing for the columns of supernode **K**, which may or may not be contiguous.
- **ik**. A pointer to the row indices of supernode **K**. It initially points to `Li [First [K]]`.
- **B**. B is of size 32-by-32, where 32 is the maximum width of a supernode. This contains the scalar values (`l_jk_times_d_k`) that are used to create the multiples of the columns of supernode **K** which are then used to update supernode **J**. First, a copy is made into **B** of the real values of supernode **K** in **Lx** corresponding to the columns in the intersection set. These are then multiplied with the appropriate diagonal entries to form the `l_jk_times_d_k` entries.
- **Wx**. The dense work vector which stores the updates to supernode **J**.
- **IntersectMap**. This is the map for the intersection set. It maps the columns within the intersection set to the columns of **Wx**. It is computed using both **Map** and the **Intersection** set.
- **Map**. see description for Map in above section.

The high level description of the entire block matrix algorithm is shown in Figure 7-8:

```
for i = 1:4:M  ; steps of 4
   for j = 1:4:N ; steps of 4
      C[4][4] = 0;
      for k = 1:2:K ; steps of 2
          Cij = Cij + Aik * Bkj;
      end for
      Wx = C[4][4] using Map;
   end for
end for
```

Figure 7-8 Algorithm for block matrix computation of `cmod (J,K)`

In Figure7-8, it can be seen that the innermost kernel is of size 4-by-4 and is stored in the array **C**. For ease of understanding, the block of supernode **K** is referred to as an array **A** of size **M**–by–**K**. The product of a 4-by-2 block of **A** and a 2-by-4 block of **B** is computed and stored in **C** in the innermost kernel. Thus the supernode **J** is updated in chunks of 4-by-4 and stored in **Wx**.

The implementation of the **mMKN** method in C-style notation is shown in Figure 7-9.

```
mMKN (M,K,N,pk,ik, B,Wx,IntersectMap,Map)
{
      for (i = 0 ; i< M ; i+ = 4)
            m4KN (K,N,pk,ik,B,Wx,IntersectMap,Map);
      end for
      switch ( M % 4)
      {
            case 0 : break ;
            case 1 : m1KN (...) ; break;
            case 2: m2KN (...) ; break;
            case 3: m3KN (...) ; break;
      }
}
```

Figure 7-9 Implementation of **mMKN** method

In Figure 7-9, it can be seen that the iteration at the outermost loop is in the direction of **M**, that is, along the length of the block of supernode **K** in steps of 4. In each

iteration, a call to the method **m4KN** is made and the same arguments are passed to it. Special cases when the step size is less than 4 are handled by the methods **m1KN**, **m2KN**, **m3KN**. These methods are similar to **m4KN**, the only difference is that **M** < 4.

The implementation of the **m4KN** method in C-style notation is shown in Figure 7-10.

```
m4KN (K,N,pk,ik,B,Wx,IntersectMap,Map)
{
       C[4][4] ;
       for (i = 0 ; i< N ; i+ = 4)
            m4K4 (K,pk,ik,B,C,j);
            Wx = C[4][4] ;   /* using Map */
      end for
      switch ( M % 4)
      {
           case 0 : break ;
           case 1 : m1K1 (...) ; Wx = C[4][4] ;break;
           case 2: m2K2 (...) ; Wx = C[4][4] ;break;
           case 3: m3K3 (...) ;Wx = C[4][4] ; break;
      }
}
```

Figure 7-10 Implementation of **m4KN** method

In Figure 7-10, it can be seen that the iteration at the loop is in the direction of **N**, that is, along the width of **B**, in steps of 4. In each iteration, a call to the method **m4K4** is made and the arguments are passed to it along with work array **C** and the column-index of the current **B** block. Special cases when the step size is less than 4 are handled by the methods **m1K1**, **m2K2**, **m3K3**. These methods are similar to **m4K4**, the only difference is that **N** < 4. The result from **C** is stored into **Wx** using the **Map** and **IntersectMap**.

The implementation of the **m4K4** method in C-style notation is shown in Figure 7-11. In this figure, it can be seen that the iteration at the loop is in the direction of **K**, that is, along the width of supernode **K**, in steps of 2. In each iteration, a call to the method **m424** is made and the arguments are passed to it along with work array **C** and the

row and column indices of the current **B** block. The special case when the step size is less than 2 is handled by the method **m414**. This method is similar to **m4K4**, the only difference is that **K<**1.

```
m4K4 (K,pk,ik,B,C,Bn)
{
       C[4][4] = 0;
       for (i = 0 ; i< K ; i+ = 2)
            m424 (pk[i],pk[i+1],B,C,Bn,k);
       end for
       if ( K % 2) == 1
             m414 (...);
       end if
}
```

Figure 7-11 Implementation of **m4K4** method

The implementation of the **m424** method in C-style notation is shown in Figure 7-12.

```
m424 (pk0,pk1,ik, B,C,Bn,Bk)
{

    for (i = 0 ; i < 4 ; i++)
    {
      for (j = 0; j< 4; j++)
      {
         C[i][j] = C[i][j]+(*pk0) * B [Bk][Bn+j] + (*pk1)* B [Bk+1] [Bn+j];
      }
      pk0++;
      pk1++;
    }
}
```

Figure 7-12 Implementation of **m424** method

In Figure 7-12, the product is computed using the **pk** pointers and indices into **B**. This kernel is further optimized by loop unrolling thereby reducing the pipeline stalls. This is the innermost kernel of the entire block matrix computation of the supernode-supernode method.

Cdiv (J)

Each column within **J** has to be updated by the columns to the left of it and then it has to be scaled. This is nothing but a Cholesky factorization of the diagonal block of **J**. Also, since the diagonal block is dense, the algorithm shown in Figure 2-2 for the dense $\mathbf{A=LDL^T}$ Cholesky factorization is used. **Wx** now serves as the matrix **A** to be factored. Also, instead of just scaling the diagonal block, the scaling is done for the entire block of **Wx**.

Special Case of Singleton Supernodes

When the target supernode **J** is of width 1, that is, it is a singleton supernode, the block computation results in an overhead and can adversely affect performance. So in this special case, the supernode-column method of numerical factorization is employed. The description of this method is given in Chapter 6.

Analysis of Supernode-Supernode Cholesky Factorization

This method reduces the amount of indirect referencing significantly by exploiting the similarity of the sparsity structure of the columns within each supernode. It also improves the cache behavior by improving cache reusability as each supernode is now updating more than one column within the target supernode. However the overhead in the case of singleton target supernodes must be taken care of by appropriately switching over to the supernode-column method for this special case. There is an overhead of function calls to do the block computation, but this overhead is not significant in comparison to the performance gain achieved by better cache reusability.

The number of floating point operations for the block computation is 2***M*K*N** where the factor of 2 is due to the addition and multiplication in the innermost loop. The total number of load-store operations is the sum of **M*K** (to load **A**), **K*N** (to load **B**)

and **M*N***2 (to load and store **Wx**). The ratio of floating point operations to memory accesses for the supernode-supernode method is

$$\frac{2MKN}{MK + KN + MN2}$$

When K = 32 and N=32 ,this ratio is approximately 20, that is, for every memory access, 20 floating point of operations can be performed. Compare this with a column oriented method, where K=1 and N=1. The ratio then is approximately 0.66. For the supernode-column method, where K=32 and N =1, the ratio is approximately 1.88. Thus block computation gives better performance.

Implementation of Supernode-Supernode Numerical Factorization

See Appendix C for the implementation of the supernode-supernode numerical factorization. This method has been implemented in C and was run in MATLAB using mexfunction interfaces.

CHAPTER 8
EXPERIMENTAL RESULTS

This chapter compares the performance of the supernodal methods with the simplicial column-column method of numerical factorization. The three methods compared are as follows:

- **Column-column**. This is the simplicial column-column numerical Cholesky factorization. The complete description of this method is given in Chapter 4.
- **Supernode-column**. This is the supernode-column numerical Cholesky factorization. The complete description of this method is given in Chapter 6.
- **Supernode-supernode**. This is the supernode-supernode numerical Cholesky factorization method. The complete description of this method is given in Chapter 7. This method is further optimized by reducing the overhead of block computation when the target supernode **J** is of size 1, that is, it is a singleton supernode. In this special case, the supernode-column method is used. The main goal is to show that this supernodal method has better performance than the other two methods.

Test Design

The tests were performed on **shine**.**cise**.**ufl**.**edu** at the **CISE** department, **University of Florida**. This is a **Sun Ultra 80** workstation with the following configuration:

- Machine Hardware: sun4u
- OS version: SunOS 5.8
- Processor type: sparc
- Hardware: SUNW, Ultra-80
- Main Memory: 4 GB
- Processors: Four 450 MHz UltraSparc II with 4 MB L2 cache (only one processor was used).

Two sets of test matrices were used. These are as follows:

- **LPNetlibMat** collection of sparse matrices collected by Dr. Tim Davis, CISE, University of Florida.
- **University of Florida's Sparse Matrix Collection**. This is accessible to anyone from the URL http://www.cise.ufl.edu/research/sparse/mat/.

All the test cases were run in MATLAB using mexfunction interfaces. Each test matrix **A** is first reordered using the COLAMD column approximate minimum degree ordering algorithm [3] so that the Cholesky factorization of $(\mathbf{A(:,P)})*(\mathbf{A(:,P)})^{\mathbf{T}}$ is sparser than that of $\mathbf{A}*\mathbf{A}^{\mathbf{T}}$. **P** is the permutation vector computed by the COLAMD algorithm. Each of the sparse Cholesky factorization methods first computes $\mathbf{AA}^{\mathbf{T}}$ and then computes its Cholesky factor **L**.

Test Results

The test results for the two collections of sparse matrices are presented in Table 8-1 and Table 8-2. The following notations are used:

- **m**. number of rows in A.
- **n**. number of columns in A.
- **nnz (A)**. number of nonzeros in A. This is equal to $|\mathcal{A}|$.
- **nnz (L)**. number of nonzeros in L. This is equal to $|\mathcal{L}|$.
- **Flop count**. Floating point operation count. This is approximately equal to $\sum_{j} |\mathcal{L}_j|^2$
- **Wall time**. Wall clock time in seconds.
- **MFLOPS**. Million Floating Point Operations Per Second. This is computed by dividing the flop count by wall time, and then dividing by 10^6.
- **ColCol/SupCol**. Ratio of the wall time of the column-column method to the wall time of the supernode-column method.
- **ColCol/SupSup**. Ratio of the wall time of the column-column method to the wall time of the supernode-supernode method.

All tables are sorted in the increasing order of **Flop count**. Table 8-1 contains the results for the LPNetlibMat collection of sparse matrices. From this table, it can be seen that there is very little performance gain for small matrices and that the column-column method performs as well, if not better, than the supernodal methods. The reason for this is that for small matrices, the supernodes are small and also the computation of supernodes is an overhead. However, as the flop count increases, the performance improvement becomes more prominent. For larger matrices, that is, matrices having flop counts greater than 10^9, more and more columns combine to form supernodes. The amount of indirect referencing is reduced and cache reusability is improved. This is evident from problem 33 onwards. Problem 35, `qap12`, runs roughly 2.7 times faster. Problem 38, `qap15`, runs 3.38 times faster. Thus the objective of this thesis is satisfied for large matrices.

Table 8-1. Test results for LPNetlibMat collection of sparse matrices

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
1.	perold	625	1506	6148	26425	1842527	0.030	61.99	0.025	74.00	0.027	67.67	1.19	1.09
2.	pds_02	2953	7716	16571	44486	2252018	0.038	59.22	0.035	63.76	0.042	53.16	1.08	0.90
3.	80bau3b	2262	12061	23264	44263	2564535	0.045	56.51	0.043	60.20	0.051	50.24	1.07	0.89
4.	woodw	1098	8418	37487	45438	2576792	0.046	55.42	0.045	57.54	0.054	47.37	1.04	0.85
5.	25fv47	821	1876	10705	39498	3043644	0.048	63.63	0.042	71.91	0.045	67.83	1.13	1.07
6.	truss	1000	8806	27836	51082	3046918	0.047	64.40	0.044	68.61	0.055	55.47	1.07	0.86
7.	wood1p	244	2595	70216	24266	3185980	0.081	39.26	0.074	43.13	0.069	46.43	1.10	1.18
8.	osa_07	1118	25067	144812	59306	3758202	0.103	36.53	0.112	33.45	0.147	25.52	0.92	0.70
9.	ken_11	14694	21349	49058	133221	4080983	0.098	41.55	0.114	35.80	0.138	29.54	0.86	0.71
10.	stocfor3	16675	23541	72721	234325	4532223	0.098	46.07	0.102	44.23	0.180	25.13	0.96	0.55
11.	greenbea	2392	5598	31070	79425	4558593	0.078	58.17	0.073	62.75	0.092	49.29	1.08	0.85
12.	greenbeb	2392	5598	31070	79425	4558593	0.085	53.74	0.074	61.53	0.095	47.86	1.15	0.89
13.	pilot_ja	940	2267	14977	54544	5871290	0.088	66.76	0.082	71.70	0.081	72.20	1.07	1.08
14.	pilotnov	975	2446	13331	55648	5935500	0.093	64.04	0.082	72.77	0.081	73.02	1.14	1.14
15.	osa_14	2337	54797	317097	121898	7246262	0.293	24.76	0.312	23.21	0.340	21.31	0.94	0.86
16.	cycle	1903	3371	21234	92074	8404256	0.134	62.57	0.117	71.69	0.120	69.86	1.15	1.12
17.	d6cube	415	6184	37704	52449	9358881	0.143	65.63	0.137	68.44	0.123	76.15	1.04	1.16
18.	bnl2	2324	4486	14996	81139	11292209	0.174	64.79	0.151	74.64	0.142	79.61	1.15	1.23
19.	osa_30	4350	104374	604488	226104	13080274	0.684	19.11	0.713	18.34	0.771	16.97	0.96	0.89
20.	degen3	1503	2604	25432	123379	16403831	0.308	53.30	0.247	66.53	0.244	67.14	1.25	1.26
21.	ken_13	28632	42659	97246	353474	16827776	0.376	44.75	0.372	45.26	0.417	40.38	1.01	0.90
22.	osa_60	10280	243246	1408073	519037	28780915	1.995	14.43	1.943	14.81	1.869	15.40	1.03	1.07
23.	d2q06c	2171	5831	33081	175037	34972257	0.580	60.31	0.465	75.19	0.406	86.07	1.25	1.43
24.	pilot	1441	4860	44375	180658	37737090	0.611	61.79	0.559	67.50	0.537	70.30	1.09	1.14
25.	qap8	912	1632	7296	204290	74089806	1.219	60.79	0.978	75.77	0.824	89.89	1.25	1.48
26.	fit1p	627	1677	9868	196878	82360630	1.244	66.20	1.114	73.95	0.882	93.40	1.12	1.41

Table 8-1 continued.

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
27.	pilot87	2030	6680	74949	422518	168779430	2.571	65.64	2.281	73.99	2.063	81.80	1.13	1.25
28.	pds_06	9881	29351	63220	593816	208033998	3.298	63.08	2.882	72.19	2.474	84.08	1.14	1.33
29.	cre_b	9648	77137	260785	775319	220519635	3.855	57.20	3.232	68.24	2.690	81.97	1.19	1.43
30.	cre_d	8926	73948	246614	763531	221294227	4.204	52.64	3.210	68.94	2.719	81.40	1.31	1.55
31.	maros_r7	3136	9408	144848	853785	240685851	3.711	64.86	4.205	57.23	4.628	52.01	0.88	0.80
32.	ken_18	105127	154699	358171	2229341	248618965	7.060	35.22	6.451	38.54	4.918	50.55	1.09	1.44
33.	pds_10	16558	49932	107605	1633555	948647731	21.882	43.35	15.530	61.09	11.174	84.90	1.41	1.96
34.	dfl001	6071	12230	35632	1544399	1254493929	28.080	44.68	21.021	59.68	13.602	92.23	1.34	2.06
35.	qap12	3192	8856	38304	2407591	3.09E+09	89.219	34.61	61.525	50.19	32.700	94.43	1.45	2.73
36.	pds_20	33874	108175	232647	6827078	8.60E+09	268.143	32.06	195.422	44.00	97.281	88.38	1.37	2.76
37.	fit2p	3000	13525	50284	4501500	9.00E+09	291.288	30.91	210.501	42.78	91.907	97.97	1.38	3.17
38.	qap15	6330	22275	94950	9342597	2.40E+10	851.630	28.20	606.610	39.60	251.903	95.35	1.40	3.38

Table 8-2 contains the results for the University of Florida's collection of sparse matrices. Column 2 lists the matrices in the form of Group Name/Problem Name.

The results from this table conclusively prove that the supernodal method performs better than the simplicial column-column method. All the matrices listed are flop counts greater than 10^8. The largest gain is seen in problem number 160, *HB/psmigr_1*, where the supernode-supernode method runs 3.61 times faster taking just 118 seconds while the column-column method takes 427 seconds. The average gain is found to be in the range of 2 to 3.5 times faster.

Also, in most of the cases, the supernode-supernode method has better performance than the supernode-column method. This is due to the block computation that improves performance by better cache reusability.

Table 8-2. Test results for University of Florida's collection of sparse matrices

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
1.	Gset/G10	800	800	38352	319787	170109669	3.253	52.295	2.304	73.840	1.781	95.520	1.41	1.83
2.	Gset/G6	800	800	38352	320063	170494375	2.588	65.879	2.330	73.166	1.812	94.092	1.11	1.43
3.	Gset/G9	800	800	38352	320059	170494653	2.583	66.007	2.276	74.918	1.815	93.942	1.13	1.42
4.	Gset/G7	800	800	38352	320063	170496207	3.430	49.712	2.318	73.540	1.786	95.477	1.48	1.92
5.	Gset/G8	800	800	38352	320106	170560116	3.041	56.093	2.294	74.361	1.793	95.125	1.33	1.70
6.	Gset/G5	800	800	38352	320247	170751731	2.603	65.605	2.299	74.281	1.778	96.014	1.13	1.46
7.	Gset/G1	800	800	38352	320280	170801062	2.943	58.040	2.308	74.001	1.777	96.097	1.28	1.66
8.	Gset/G2	800	800	38352	320285	170808797	3.292	51.881	2.263	75.486	1.811	94.339	1.45	1.82
9.	Gset/G4	800	800	38352	320303	170835833	2.556	66.846	2.320	73.631	1.776	96.189	1.10	1.44
10.	Gset/G3	800	800	38352	320308	170843792	2.931	58.282	2.285	74.752	1.799	94.983	1.28	1.63
11.	FIDAP/ex31	3909	3909	91223	741347	176685509	2.623	67.360	2.567	68.821	2.142	82.474	1.02	1.22
12.	Nemeth/nemeth21	9506	9506	1173746	1297263	177936811	5.182	34.337	5.125	34.717	5.345	33.293	1.01	0.97
13.	Wang/wang1	2903	2903	19093	590936	189525380	3.545	53.463	2.648	71.581	2.036	93.086	1.34	1.74
14.	Wang/wang2	2903	2903	19093	590936	189525380	3.407	55.632	2.635	71.932	2.119	89.421	1.29	1.61
15.	FIDAP/ex14	3251	3251	65875	723046	200078644	3.448	58.019	2.685	74.506	2.281	87.719	1.28	1.51
16.	Mallya/lhr04c	4101	4101	82682	758440	201947790	3.664	55.111	3.346	60.362	2.983	67.694	1.10	1.23
17.	LPnetlib/lp_pds_06	9881	29351	63220	593816	208033998	3.468	59.987	2.945	70.632	2.456	84.707	1.18	1.41
18.	FIDAP/ex26	2163	2163	74464	630748	209531094	3.149	66.530	2.839	73.807	2.325	90.109	1.11	1.35
19.	LPnetlib/lp_cre_b	9648	77137	260785	775319	220519635	4.448	49.578	3.207	68.761	2.671	82.564	1.39	1.67
20.	LPnetlib/lp_cre_d	8926	73948	246614	763531	221294227	4.506	49.115	3.260	67.881	2.731	81.028	1.38	1.65
21.	Okunbor/aft01	8205	8205	125567	1234184	227699204	3.816	59.664	3.229	70.522	3.257	69.901	1.18	1.17
22.	ATandT/onetone2	36057	36057	222596	1376334	229340616	4.047	56.667	3.234	70.916	2.872	79.852	1.25	1.41
23.	Simon/raefsky5	6316	6316	167178	943697	229940327	3.873	59.370	3.097	74.258	2.606	88.245	1.25	1.49
24.	Gset/G53	1000	1000	11828	407151	231247217	3.955	58.473	3.058	75.627	2.429	95.202	1.29	1.63
25.	Gset/G54	1000	1000	11832	411578	235878096	4.585	51.445	3.139	75.139	2.543	92.769	1.46	1.80
26.	DRIVCAV/cavity17	4562	4562	131735	943772	236532566	3.696	63.996	3.242	72.962	2.785	84.924	1.14	1.33

Table 8-2. continued

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
27.	DRIVCAV/cavity19	4562	4562	131735	943772	236532566	4.245	55.719	3.262	72.516	2.765	85.542	1.30	1.54
28.	DRIVCAV/cavity21	4562	4562	131735	943772	236532566	4.235	55.852	3.274	72.253	2.759	85.738	1.29	1.54
29.	DRIVCAV/cavity23	4562	4562	131735	943772	236532566	3.636	65.049	3.255	72.671	2.762	85.635	1.12	1.32
30.	DRIVCAV/cavity25	4562	4562	131735	943772	236532566	3.777	62.620	3.265	72.453	2.775	85.249	1.16	1.36
31.	Simon/raefsky6	3402	3402	130371	730376	236723568	4.890	48.407	3.673	64.447	2.611	90.677	1.33	1.87
32.	Gset/G52	1000	1000	11832	412074	237068622	4.119	57.556	3.076	77.082	2.483	95.472	1.34	1.66
33.	Gset/G51	1000	1000	11818	415051	239259885	4.230	56.566	3.217	74.370	2.595	92.206	1.31	1.63
34.	LPnetlib/lp_maros_r7	3136	9408	144848	853785	240685851	3.812	63.136	4.296	56.023	4.633	51.954	0.89	0.82
35.	TOKAMAK/utm5940	5940	5940	83842	1004940	242156356	4.471	54.157	3.347	72.358	2.815	86.024	1.34	1.59
36.	LPnetlib/lp_ken_18	105127	154699	358171	2229341	248618965	7.326	33.935	6.502	38.237	4.894	50.802	1.13	1.50
37.	Nemeth/nemeth22	9506	9506	1358832	1538058	250327160	7.733	32.372	7.283	34.373	8.023	31.203	1.06	0.96
38.	Okunbor/aft02	8184	8184	127762	1326827	265748075	4.658	57.051	3.741	71.036	3.709	71.651	1.25	1.26
39.	DRIVCAV/cavity18	4562	4562	138040	1054903	274463825	4.288	64.009	3.813	71.989	3.202	85.720	1.12	1.34
40.	DRIVCAV/cavity20	4562	4562	138040	1054903	274463825	4.307	63.731	3.836	71.548	3.189	86.062	1.12	1.35
41.	DRIVCAV/cavity22	4562	4562	138040	1054903	274463825	4.245	64.649	3.838	71.506	3.176	86.414	1.11	1.34
42.	DRIVCAV/cavity24	4562	4562	138040	1054903	274463825	4.902	55.985	3.823	71.791	3.178	86.351	1.28	1.54
43.	DRIVCAV/cavity26	4562	4562	138040	1054903	274463825	4.962	55.314	3.834	71.596	3.175	86.447	1.29	1.56
44.	DRIVCAV/cavity16	4562	4562	137887	1066633	280896845	4.849	57.926	3.785	74.216	3.239	86.727	1.28	1.50
45.	Grund/meg1	2904	2904	58142	649377	283105887	4.785	59.161	3.943	71.806	3.132	90.388	1.21	1.53
46.	Gset/G44	1000	1000	19980	475029	301694341	5.675	53.160	3.965	76.084	3.166	95.304	1.43	1.79
47.	Gset/G47	1000	1000	19980	475563	302000165	5.665	53.311	4.033	74.879	3.139	96.223	1.40	1.80
48.	Gset/G45	1000	1000	19980	476028	302723998	5.521	54.833	4.077	74.247	3.154	95.970	1.35	1.75
49.	Gset/G43	1000	1000	19980	477342	304185672	5.504	55.270	4.023	75.618	3.184	95.527	1.37	1.73
50.	Gset/G46	1000	1000	19980	477516	304273252	5.484	55.480	4.011	75.857	3.158	96.358	1.37	1.74
51.	Bai/qc2534	2534	2534	463360	858144	305227088	6.968	43.805	6.618	46.120	6.456	47.277	1.05	1.08
52.	Nasa/skirt	12598	12598	196520	1523961	311701613	4.989	62.479	4.339	71.843	4.189	74.417	1.15	1.19
53.	Nemeth/nemeth23	9506	9506	1506810	1725881	315479391	9.730	32.423	8.873	35.556	8.716	36.194	1.10	1.12
54.	Nemeth/nemeth24	9506	9506	1506550	1749670	324244980	8.786	36.904	8.567	37.847	8.601	37.697	1.03	1.02
55.	Nemeth/nemeth25	9506	9506	1511758	1749823	324304803	8.769	36.981	8.575	37.822	8.531	38.017	1.02	1.03

Table 8-2. continued

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
56.	Nemeth/nemeth26	9506	9506	1511760	1749823	324304803	8.786	36.910	9.898	32.764	8.944	36.258	0.89	0.98
57.	Averous/epb1	14734	14734	95053	1624166	332499130	5.011	66.348	4.390	75.748	3.731	89.123	1.14	1.34
58.	Mallya/lhr07	7337	7337	154660	1488879	399877411	6.708	59.615	5.799	68.960	5.063	78.985	1.16	1.32
59.	Boeing/nasa2910	2910	2910	174296	1048038	426161336	7.686	55.448	6.385	66.742	4.958	85.957	1.20	1.55
60.	Nasa/nasa2910	2910	2910	174296	1098730	458755010	8.932	51.361	6.434	71.299	5.039	91.046	1.39	1.77
61.	HB/bcsstk13	2003	2003	83883	848609	464061305	9.689	47.897	6.588	70.439	5.140	90.284	1.47	1.88
62.	HB/bcsstk28	4410	4410	219024	1297720	473030614	11.251	42.045	8.297	57.013	7.093	66.689	1.36	1.59
63.	Mallya/lhr07c	7337	7337	156508	1625798	487049380	9.775	49.826	7.765	62.724	7.309	66.639	1.26	1.34
64.	FIDAP/ex8	3096	3096	90841	1151433	549048309	10.953	50.127	7.845	69.987	6.013	91.306	1.40	1.82
65.	Boeing/nasa4704	4704	4704	104756	1306234	549561684	10.979	50.055	8.145	67.470	6.452	85.180	1.35	1.70
66.	Mallya/lhr10	10672	10672	228395	2171771	556076329	10.184	54.601	7.937	70.065	7.341	75.748	1.28	1.39
67.	Nasa/nasa4704	4704	4704	104756	1320652	560924686	10.245	54.751	8.136	68.947	6.465	86.762	1.26	1.58
68.	Mallya/lhr10c	10672	10672	232633	2230723	607887597	11.147	54.533	9.966	60.997	8.697	69.895	1.12	1.28
69.	Hollinger/g7jac020	5850	5850	42568	1210411	610432605	11.420	53.451	8.728	69.943	6.364	95.924	1.31	1.79
70.	Hollinger/g7jac020sc	5850	5850	42568	1210413	610432931	11.174	54.631	8.653	70.544	6.434	94.876	1.29	1.74
71.	HB/bcsstk15	3948	3948	117816	1449877	624715699	12.420	50.300	9.134	68.396	7.596	82.247	1.36	1.64
72.	Averous/epb3	84617	84617	463625	4868600	635672400	9.918	64.092	8.584	74.051	7.770	81.810	1.16	1.28
73.	Mallya/lhr11	10964	10964	231806	2610715	838488207	17.395	48.202	13.419	62.484	11.731	71.478	1.30	1.48
74.	Mallya/lhr11c	10964	10964	233741	2708223	908962325	18.268	49.756	14.330	63.433	11.859	76.651	1.27	1.54
75.	LPnetlib/lp_pds_10	16558	49932	107605	1633555	948647731	21.615	43.889	15.601	60.806	11.262	84.231	1.39	1.92
76.	HB/bcsstk16	4884	4884	290378	2019675	961598107	16.539	58.142	14.981	64.189	13.580	70.812	1.10	1.22
77.	Boeing/crystk01	4875	4875	315891	2148267	1035304495	19.980	51.816	15.626	66.255	14.039	73.744	1.28	1.42
78.	Mallya/lhr14c	14270	14270	307858	3394088	1054889610	21.161	49.851	18.233	57.856	15.944	66.162	1.16	1.33
79.	HB/bcsstk17	10974	10974	428650	3095965	1101809421	19.752	55.783	15.849	69.520	13.924	79.131	1.25	1.42
80.	LPnetlib/lpi_cplex1	3005	5224	10947	1139265	1135195805	25.341	44.798	18.149	62.549	11.387	99.691	1.40	2.23
81.	Mallya/lhr14	14270	14270	305750	3617331	1252808831	31.719	39.497	24.969	50.174	22.263	56.274	1.27	1.42
82.	LPnetlib/lp_dfl001	6071	12230	35632	1544399	1254493929	28.214	44.464	21.106	59.437	13.596	92.266	1.34	2.08
83.	Nasa/pwt	36519	36519	326107	4999500	1262399344	23.138	54.560	17.201	73.392	14.027	89.998	1.35	1.65

Table 8-2. continued

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
84.	FIDAP/ex40	7740	7740	456188	3107372	1283261286	23.970	53.535	19.189	66.875	17.754	72.279	1.25	1.35
85.	Averous/epb2	25228	25228	175027	3711441	1294790649	24.386	53.096	18.888	68.550	13.705	94.476	1.29	1.78
86.	Grund/meg4	5860	5860	25258	1500034	1350252050	32.528	41.510	22.298	60.555	14.167	95.307	1.46	2.30
87.	Bomhof/circuit_2	4510	4510	21199	1652270	1383486656	31.682	43.668	20.931	66.096	13.918	99.400	1.51	2.28
88.	HB/orani678	2529	2529	90158	1540627	1417599413	43.535	32.562	32.844	43.161	22.913	61.868	1.33	1.90
89.	Mallya/lhr17c	17576	17576	381975	4385695	1447817515	31.436	46.056	26.640	54.348	23.050	62.811	1.18	1.36
90.	Hamm/scircuit	170998	170998	958936	8418483	1487398273	24.585	60.501	20.820	71.442	17.889	83.147	1.18	1.37
91.	Boeing/bcsstk38	8032	8032	355460	3072775	1533240723	31.879	48.095	22.753	67.386	17.625	86.992	1.40	1.81
92.	Mallya/lhr17	17576	17576	379761	4477627	1538108643	35.828	42.931	31.620	48.644	28.513	53.944	1.13	1.26
93.	Hollinger/jan99jac020	6774	6774	33744	2039668	1558251222	34.655	44.965	25.450	61.228	16.568	94.050	1.36	2.09
94.	Hollinger/jan99jac020sc	6774	6774	33744	2039735	1558308261	37.775	41.253	25.686	60.668	16.371	95.187	1.47	2.31
95.	LPnetlib/lpi_ceria3d	3576	4400	21178	1590872	1561353042	39.223	39.807	29.752	52.478	21.077	74.080	1.32	1.86
96.	HB/bcsstk18	11948	11948	149090	3082230	1574762140	31.001	50.797	23.781	66.219	16.916	93.095	1.30	1.83
97.	Garon/garon2	13535	13535	373235	4324300	1672354266	34.173	48.937	25.193	66.383	19.829	84.337	1.36	1.72
98.	Gset/G40	2000	2000	23532	1599906	1838023916	50.527	36.377	33.455	54.940	19.271	95.380	1.51	2.62
99.	Gset/G36	2000	2000	23532	1605105	1844233063	47.477	38.845	33.468	55.105	19.215	95.979	1.42	2.47
100.	Gset/G42	2000	2000	23558	1609297	1853136197	50.247	36.881	33.687	55.011	19.528	94.896	1.49	2.57
101.	Gset/G41	2000	2000	23570	1606373	1853367465	44.957	41.225	33.715	54.971	19.376	95.653	1.33	2.32
102.	Gset/G39	2000	2000	23556	1608712	1854078664	51.052	36.317	33.917	54.665	19.619	94.503	1.51	2.60
103.	Gset/G37	2000	2000	23570	1612649	1859738409	51.160	36.351	33.732	55.133	19.522	95.262	1.52	2.62
104.	Gset/G38	2000	2000	23558	1615872	1860311384	50.763	36.647	34.440	54.015	19.515	95.328	1.47	2.60
105.	Gset/G35	2000	2000	23556	1614921	1861710845	50.412	36.930	33.715	55.219	19.644	94.774	1.50	2.57
106.	Boeing/bcsstm36	23052	23052	320606	4004046	1907288086	37.816	50.435	29.426	64.816	21.866	87.226	1.29	1.73
107.	Shyy/shyy161	76480	76480	329762	6570517	1912358951	50.137	38.143	39.184	48.804	32.678	58.522	1.28	1.53
108.	Goodwin/goodwin	7320	7320	324772	3695060	2180890000	43.796	49.797	33.438	65.222	23.203	93.991	1.31	1.89
109.	Brethour/coater2	9540	9540	207308	4140284	2287428000	45.637	50.122	34.634	66.046	24.803	92.225	1.32	1.84
110.	HB/bcsstk25	15439	15439	252241	5231323	2319027000	53.125	43.653	38.641	60.014	30.205	76.775	1.37	1.76
111.	Gset/G31	2000	2000	39980	1874295	2376445000	63.961	37.155	45.815	51.871	24.895	95.460	1.40	2.57

Table 8-2. continued

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
112.	Gset/G26	2000	2000	39980	1875211	2377664000	67.386	35.284	45.545	52.205	24.754	96.052	1.48	2.72
113.	Gset/G27	2000	2000	39980	1876405	2379489000	66.650	35.701	45.644	52.131	24.646	96.546	1.46	2.70
114.	Gset/G22	2000	2000	39980	1877908	2381447000	61.996	38.413	45.520	52.316	24.697	96.427	1.36	2.51
115.	Gset/G29	2000	2000	39980	1879194	2384794000	62.426	38.202	45.485	52.431	24.720	96.471	1.37	2.53
116.	Gset/G28	2000	2000	39980	1879207	2385058000	64.490	36.984	46.119	51.716	24.957	95.568	1.40	2.58
117.	Gset/G24	2000	2000	39980	1880150	2386298000	67.208	35.506	45.868	52.025	24.738	96.464	1.47	2.72
118.	Gset/G23	2000	2000	39980	1880542	2387103000	64.595	36.955	45.548	52.408	24.935	95.734	1.42	2.59
119.	Gset/G30	2000	2000	39980	1883275	2396120000	63.192	37.918	45.546	52.608	24.904	96.213	1.39	2.54
120.	Gset/G25	2000	2000	39980	1884011	2397097000	68.206	35.145	45.921	52.200	24.973	95.987	1.49	2.73
121.	Graham/graham1	9035	9035	335472	4144723	2469852000	58.567	42.171	40.371	61.178	26.489	93.241	1.45	2.21
122.	Mallya/lhr34c	35152	35152	764014	8716749	2872392000	654.847	4.386	639.996	4.488	654.688	4.387	1.02	1.00
123.	Boeing/crystm02	13965	13965	322905	5438616	2874462000	115.264	24.938	100.896	28.489	92.635	31.030	1.14	1.24
124.	Mallya/lhr34	35152	35152	746972	8242179	3058159000	56.643	53.990	45.996	66.488	35.975	85.007	1.23	1.57
125.	LPnetlib/lp_qap12	3192	8856	38304	2407591	3087899000	79.632	38.777	61.432	50.265	32.322	95.535	1.30	2.46
126.	Rothberg/struct4	4350	4350	237798	3321077	3096405000	83.181	37.225	60.163	51.467	35.387	87.502	1.38	2.35
127.	HB/bcsstk29	13992	13992	619488	6143522	3701704000	93.482	39.598	73.840	50.132	52.193	70.923	1.27	1.79
128.	ATandT/onetone1	36057	36057	335552	5017027	3754836000	96.019	39.105	70.801	53.033	41.074	91.415	1.36	2.34
129.	Hollinger/mark3jac020	9129	9129	52883	3681913	3777894000	106.487	35.477	72.399	52.182	39.022	96.815	1.47	2.73
130.	Qaplib/lp_nug12	3192	8856	38304	2713965	3897213000	112.026	34.788	79.715	48.889	40.393	96.483	1.41	2.77
131.	Simon/olafu	16146	16146	1015156	7347770	4303513000	84.724	50.795	70.721	60.852	56.331	76.397	1.20	1.50
132.	Simon/raefsky1	3242	3242	293409	3422969	4310133000	122.783	35.104	90.204	47.782	55.344	77.879	1.36	2.22
133.	Simon/raefsky2	3242	3242	293551	3424313	4311574000	123.463	34.922	90.999	47.381	54.986	78.413	1.36	2.25
134.	Hollinger/g7jac040	11790	11790	107383	4544093	4481829000	115.122	38.931	88.527	50.627	46.728	95.913	1.30	2.46
135.	Hollinger/g7jac040sc	11790	11790	107383	4544097	4481831000	130.517	34.339	88.068	50.891	46.641	96.092	1.48	2.80
136.	Mulvey/pfinan512	74752	74752	596992	15377219	5195197000	85.829	60.529	72.168	71.988	58.314	89.090	1.19	1.47
137.	Mulvey/finan512	74752	74752	596992	15382253	5195893000	85.426	60.824	71.075	73.104	58.506	88.809	1.20	1.46
138.	Cote/mplate	5962	5962	142190	4640465	5244537000	153.641	34.135	106.799	49.107	55.689	94.175	1.44	2.76
139.	Mallya/lhr71c	70304	70304	1528092	17506866	5791794000	1274.009	4.546	1244.647	4.653	1274.651	4.544	1.02	1.00

Table 8-2. continued

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
140.	Bomhof/circuit_1	2624	2624	35823	3371039	5834597000	171.487	34.024	129.136	45.182	59.314	98.368	1.33	2.89
141.	Mallya/lhr71	70304	70304	1494006	16426629	6074660000	125.157	48.536	91.430	66.440	71.807	84.597	1.37	1.74
142.	Bai/af23560	23560	23560	460598	11113062	6725722000	137.914	48.767	106.059	63.415	76.416	88.015	1.30	1.80
143.	HB/bcsstk33	8738	8738	591904	7063307	7191355000	225.907	31.833	180.477	39.846	115.526	62.249	1.25	1.96
144.	Goodwin/rim	22560	22560	1014951	12539614	7899598000	168.447	46.897	125.288	63.052	84.048	93.989	1.34	2.00
145.	Boeing/crystm03	24696	24696	583770	12126954	7959582000	757.712	10.505	730.646	10.894	697.881	11.405	1.04	1.09
146.	Simon/venkat01	62424	62424	1717792	20928396	8368295000	161.096	51.946	137.175	61.005	110.032	76.053	1.17	1.46
147.	Simon/venkat50	62424	62424	1717777	20938444	8375653000	158.082	52.983	135.781	61.685	110.151	76.038	1.16	1.44
148.	Simon/venkat25	62424	62424	1717763	20986556	8434109000	178.687	47.200	136.769	61.667	110.786	76.130	1.31	1.61
149.	Boeing/pcrystk02	13965	13965	968583	10581232	8448026000	195.257	43.266	147.950	57.101	125.779	67.166	1.32	1.55
150.	Hamm/hcircuit	105676	105676	513072	10042070	8504487000	257.074	33.082	175.713	48.400	88.737	95.839	1.46	2.90
151.	LPnetlib/lp_pds_20	33874	108175	232647	6827078	8597795000	266.394	32.275	193.859	44.351	97.914	87.810	1.37	2.72
152.	Boeing/crystk02	13965	13965	968583	10693425	8618372000	175.604	49.078	148.227	58.143	126.029	68.384	1.18	1.39
153.	Rothberg/struct3	53570	53570	1173694	18310916	8647195000	197.803	43.716	146.910	58.860	114.173	75.738	1.35	1.73
154.	Bova/rma10	46835	46835	2329092	16984768	8731659000	180.274	48.436	140.912	61.966	106.258	82.174	1.28	1.70
155.	LPnetlib/lp_fit2p	3000	13525	50284	4501500	9004500000	292.896	30.743	208.224	43.244	91.946	97.932	1.41	3.19
156.	Hollinger/g7jac050sc	14760	14760	145157	7140185	9330335000	285.102	32.726	204.642	45.593	97.036	96.154	1.39	2.94
157.	Gset/G56	5000	5000	24996	4732117	9381531000	292.330	32.092	221.523	42.350	97.896	95.832	1.32	2.99
158.	Gset/G55	5000	5000	24996	4732160	9381574000	276.865	33.885	223.153	42.041	97.451	96.270	1.24	2.84
159.	HB/psmigr_2	3140	3140	540022	4919808	10273540000	350.835	29.283	268.154	38.312	117.952	87.099	1.31	2.97
160.	HB/psmigr_1	3140	3140	543160	4919818	10273580000	427.466	24.034	267.662	38.383	118.558	86.655	1.60	3.61
161.	HB/psmigr_3	3140	3140	543160	4919818	10273580000	351.945	29.191	269.261	38.155	118.117	86.978	1.31	2.98
162.	Hollinger/mark3jac040	18289	18289	106803	9066438	11060620000	328.624	33.657	229.932	48.104	114.393	96.689	1.43	2.87
163.	Hollinger/jan99jac040	13694	13694	72734	8622550	13940050000	449.697	30.999	321.030	43.423	144.772	96.290	1.40	3.11
164.	Hollinger/jan99jac040sc	13694	13694	72734	8622567	13940060000	446.690	31.207	326.138	42.743	146.535	95.131	1.37	3.05
165.	Boeing/msc10848	10848	10848	1229776	12305436	15149390000	444.361	34.092	313.221	48.366	163.239	92.805	1.42	2.72
166.	Hamm/memplus	17758	17758	99147	8425062	15490000000	529.188	29.271	379.717	40.794	160.856	96.297	1.39	3.29
167.	Simon/raefsky3	21200	21200	1488768	17243112	15892560000	399.429	39.788	288.577	55.072	192.717	82.466	1.38	2.07

Table 8-2. continued

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
168.	Hollinger/mark3jac060	27449	27449	160723	14166500	16904030000	490.097	34.491	348.329	48.529	174.843	96.681	1.41	2.80
169.	Vavasis/av41092	41092	41092	1683902	18270558	18366600000	565.145	32.499	398.189	46.125	204.424	89.845	1.42	2.76
170.	Hollinger/g7jac060	17730	17730	183325	10861304	18446460000	567.243	32.519	429.857	42.913	190.564	96.799	1.32	2.98
171.	Hollinger/g7jac060sc	17730	17730	183325	10861308	18446470000	565.457	32.622	427.197	43.180	190.587	96.788	1.32	2.97
172.	Zhao/Zhao1	33861	33861	166453	17495241	19950720000	602.258	33.127	423.641	47.093	207.343	96.221	1.42	2.90
173.	HB/bcsstk30	28924	28924	2043492	22445143	21049860000	724.263	29.064	529.288	39.770	292.793	71.893	1.37	2.47
174.	Boeing/pcrystk03	24696	24696	1751178	23086574	22805530000	548.315	41.592	459.979	49.580	349.848	65.187	1.19	1.57
175.	Boeing/crystk03	24696	24696	1751178	23259357	23099770000	550.409	41.968	460.847	50.125	350.200	65.962	1.19	1.57
176.	LPnetlib/lp_qap15	6330	22275	94950	9342597	24019620000	729.420	32.930	600.900	39.973	252.201	95.240	1.21	2.89
177.	Gset/G61	7000	7000	34296	8979461	24681460000	875.049	28.206	633.231	38.977	256.636	96.173	1.38	3.41
178.	Gset/G60	7000	7000	34296	8979541	24681680000	875.450	28.193	633.397	38.967	256.934	96.062	1.38	3.41
179.	Hollinger/mark3jac080	36609	36609	214643	20613574	27314350000	757.383	36.064	583.054	46.847	287.324	95.065	1.30	2.64
180.	Hollinger/g7jac080	23670	23670	259648	15669651	28794400000	948.473	30.359	685.252	42.020	297.416	96.815	1.38	3.19
181.	Hollinger/g7jac080sc	23670	23670	259648	15669656	28794400000	905.237	31.809	686.944	41.917	299.107	96.268	1.32	3.03
182.	Gset/G59	5000	5000	59140	9981679	29024400000	1021.634	28.410	790.764	36.704	303.843	95.524	1.29	3.36
183.	Gset/G58	5000	5000	59140	10002737	29084430000	1053.358	27.611	756.885	38.426	303.596	95.800	1.39	3.47
184.	HB/bcsstk31	35588	35588	1181416	24451314	29490740000	1247.392	23.642	910.942	32.374	406.700	72.512	1.37	3.07
185.	Qaplib/lp_nug15	6330	22275	94950	10732658	31052350000	1096.489	28.320	786.917	39.461	323.385	96.023	1.39	3.39
186.	Hollinger/mark3jac100	45769	45769	268563	26806307	36822480000	1128.636	32.626	802.982	45.857	382.885	96.171	1.41	2.95
187.	Wang/wang3	26064	26064	177168	21574595	38715380000	1209.351	32.013	918.913	42.132	405.571	95.459	1.32	2.98
188.	Hollinger/mark3jac120	54929	54929	322483	31137276	40519030000	1174.421	34.501	859.876	47.122	419.129	96.674	1.37	2.80
189.	Hollinger/jan99jac060	20614	20614	111903	18336070	41533620000	1454.943	28.547	1017.623	40.814	429.375	96.730	1.43	3.39
190.	Hollinger/jan99jac060sc	20614	20614	111903	18336126	41533690000	1436.237	28.918	1017.520	40.819	429.977	96.595	1.41	3.34
191.	Wang/wang4	26068	26068	177196	22732723	45355570000	1518.457	29.870	1099.338	41.257	473.740	95.739	1.38	3.21
192.	Nasa/nasasrb	54870	54870	2677324	48435319	52334471145	1921.888	27.231	1280.019	40.886	734.449	71.257	1.50	2.62
193.	Hollinger/g7jac100	29610	29610	335972	24015227	54977160000	1684.484	32.637	1375.645	39.965	571.634	96.176	1.22	2.95
194.	Hollinger/g7jac100sc	29610	29610	335972	24015230	54977170000	1795.315	30.623	1364.166	40.301	570.660	96.340	1.32	3.15
195.	Hollinger/mark3jac140	64089	64089	376395	39399002	57531669512	1849.425	31.108	1283.443	44.826	596.607	96.431	1.44	3.10

Table 8-2. continued

No	Problem	m	n	nnz(A)	nnz(L)	flop	Col - Col		Sup-Col		Sup-Sup		ColCol/ SupCol	ColCol/ SupSup
							wall	MFLOPS	wall	MFLOPS	wall	MFLOPS		
196.	Bomhof/circuit_3	12127	12127	48137	16410265	61347520000	2204.868	27.824	1607.291	38.168	620.787	98.822	1.37	3.55
197.	Hollinger/jan99jac080	27534	27534	151063	26665966	63467570000	1947.108	32.596	1571.188	40.395	655.117	96.880	1.24	2.97
198.	Hollinger/jan99jac080sc	27534	27534	151063	26666077	63467810000	2211.139	28.704	1569.065	40.449	657.747	96.493	1.41	3.36
199.	Gset/G64	7000	7000	82918	19509120	79735630000	2956.030	26.974	2235.549	35.667	832.551	95.773	1.32	3.55
200.	Gset/G63	7000	7000	82918	19546685	79879530000	2762.487	28.916	2161.287	36.959	836.140	95.534	1.28	3.30
201.	Simon/bbmat	38744	38744	1771722	48340616	81102354486	2672.973	30.342	1989.278	40.770	899.042	90.210	1.34	2.97
202.	Hollinger/g7jac120	35550	35550	412306	31923992	82034936754	2886.934	28.416	2080.726	39.426	852.448	96.235	1.39	3.39
203.	Hollinger/g7jac120sc	35550	35550	412306	31923992	82034936754	2888.288	28.403	2081.670	39.408	852.228	96.259	1.39	3.39
204.	Hollinger/g7jac140	41490	41490	488633	39338350	1.01059E+11	3590.847	28.143	2561.638	39.451	1060.150	95.325	1.40	3.39
205.	Hollinger/g7jac140sc	41490	41490	488633	39338350	1.01059E+11	3588.180	28.164	2563.804	39.418	1061.272	95.224	1.40	3.38
206.	Hollinger/jan99jac100	34454	34454	190224	37744175	1.02127E+11	3416.191	29.895	2584.148	39.521	1072.911	95.187	1.32	3.18
207.	Hollinger/jan99jac100sc	34454	34454	190224	37744175	1.02127E+11	3198.297	31.932	2582.730	39.542	1065.001	95.894	1.24	3.00
208.	Hollinger/jan99jac120	41374	41374	229385	48385209	1.3588E+11	4836.205	28.096	3460.230	39.269	1417.700	95.846	1.40	3.41
209.	Hollinger/jan99jac120sc	41374	41374	229385	48385209	1.3588E+11	4850.367	28.014	3456.805	39.308	1417.664	95.848	1.40	3.42
210.	Ronis/xenon1	48600	48600	1181120	57274787	1.36106E+11	4908.634	27.728	3456.457	39.377	1411.972	96.394	1.42	3.48
211.	Hollinger/g7jac160	47430	47430	564952	50057053	1.4722E+11	5317.605	27.685	3845.311	38.286	1551.527	94.887	1.38	3.43
212.	Hollinger/g7jac160sc	47430	47430	564952	50057053	1.4722E+11	5304.352	27.755	3830.373	38.435	1547.406	95.140	1.38	3.43
213.	ATandT/twotone	120750	120750	1206265	54135558	1.4964E+11	5067.283	29.531	3959.697	37.791	1535.921	97.427	1.28	3.30
214.	Li/pli	22695	22695	1350309	49861690	1.58264E+11	5093.441	31.072	4178.776	37.873	1775.458	89.140	1.22	2.87
215.	Hollinger/g7jac180sc	53370	53370	641290	56321164	1.63043E+11	5523.093	29.520	4208.354	38.743	1707.456	95.489	1.31	3.23
216.	Hollinger/g7jac200sc	59310	59310	717620	63537939	1.83407E+11	6168.445	29.733	4716.108	38.890	1912.268	95.911	1.31	3.23
217.	Qaplib/lp_nug20	15240	72600	304800	63193703	4.49828E+11	16554.380	27.173	12261.373	36.687	5592.126	80.440	1.35	2.96

CHAPTER 9
CONCLUSION AND FUTURE WORK

The experimental results conclusively show that the supernode-supernode numerical Cholesky factorization gives a much better performance then the simplicial column-column numerical factorization even when the supernodal columns are interspersed. The performance improvement is approximately 2 to 3.5 times for large matrices. Thus supernodes can be used to exploit the similarity within the sparsity structure to reduce indirect referencing and to improve cache reusability. This technique can be used in a variety of applications where the time requirement is critical.

There are several other implementations of the supernodal method such as SPOOLES. A performance comparison with these other implementations is left for future work. Supernodal updates and downdates to the sparse Cholesky factor is an area of further study. Better heuristics for determining the supernodal elimination tree can be researched.

APPENDIX A
MATLAB SOURCE CODE FOR SYMBOLIC FACTORIZATION

```
function [L,Parent] = symfact4(A)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%symfact4 is a function that returns the pattern for the Cholesky factor of AA'

%input parameters :
%    A : m * n sparse matrix. AA' is to be factorized symbolically.

%output parameters :
%    L : m * m sparse matrix having pattern for Cholesky factor L of AA'.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[m,n] = size(A);       % %get dimensions of A

Parent = zeros (1,m) ; %% store parents of each column of L

AC = zeros (1,n) ;     %% row vector for storing min nz row index of
                       %% each col of A

for j = 1:n
    AC (j) = min (find (A (:,j))) ;
end

%% set L to an identity matrix of size m * m
L = speye(m);


for j = 1:m

    %% Lj = Lj U (min Ak)
    for k = find (AC == j)
     L(j+1:m,j) = L(j+1:m,j) | A(j+1:m,k);
    end

    %% Lj = Lj U Lc
    for c = find (Parent == j)
     L(j+1:m,j) = L(j+1:m,j) | L(j+1:m,c) ;
    end

    %% store parent of j in Parent
    p = min (find (L (j+1:m,j))) ;
    if ~isempty (p)
       Parent (j) = p + j ;
    end

end
```

APPENDIX B
SOURCE CODE FOR COLUMN-COLUMN FACTORIZATION IN C

```
/* ========================================================================== */
/* === ldlcc ================================================================ */
/* ========================================================================== */

/*
    Numerical LDL' factorization.

    Matlab calling syntax is:

       [Lx, D] = ldlcc (A, Basis, sigma, Lp, Li, Lnz) ;

    Author : Dr. Tim Davis (davis@cise.ufl.edu), University of Florida.

    The factorization is:

       Af = A (:, basis)
       B = sigma*I + Af*Af'
       LDL' = B

    The matrix L is stored in 0-based column form.
    A is a Matlab sparse matrix, which is held in 0-based column form.
    A must be in packed form, with sorted columns.

    The pattern of L has already been computed, and is in Lp, Li, and Lnz.
    This routine just computes Lx and D. All the other arguments are not
    modified. L->m is not used. The columns of L must be sorted.

    The diagonal of L is not stored (L is unit diagonal).

    Returns TRUE if the factorization was successful, FALSE otherwise.

    This routine performs a number of flops exactly equal to:

       sum (for each column j of Af) of (Anz (j)^2 + Anz (j)), to form B
       +
       sum (for each column j of L ) of (Lnz (j)^2 + 3*Lnz (j)), to factorize B

    where Anz (j) is the number of nonzeros in column j of Af,
    and Lnz (j) is the number of nonzero in column j of L below the diagonal.
*/

/* ========================================================================== */

#include "sparse.h"

PUBLIC int ldlcc
(
    Sparse_Matrix *A,       /* in packed form, with sorted columns */
    Sparse_Matrix *L,       /* with sorted columns */
    Set *Basis,             /* basis set, uses Basis->list [0..Basis->size-1]*/
    double sigma,           /* normally 1e-12 for LPDASA */

    double D [],            /* D: diagonal of the matrix D */

    double W [],            /* W [0..n-1] */
    int    Link [],         /* Link [0..n-1] */
    int    First [],        /* First [0..n-1] */

    /* for walking through Af transpose: */
    int    Ahead [],        /* Ahead [0..L->ncol-1], Ahead [j] is head of */
```

```
                        /* link list of cols of A that apply to col j */
                        /* of L */
    int    Anext [],    /* Anext [0..A->ncol-1], remainder of */
                        /* link list */
    int    Aleft []     /* Aleft [0..A->ncol-1], Aleft [j] points to */
                        /* remainder of col j */
                        /* Aleft [j] is EMPTY if j is not in Basis */
)
{
    /* ========================================================================= */
    /* ==== local variables ==================================================== */
    /* ========================================================================= */

    int i, j, p, p2, k, nextk, nextj, *Ai, *Ap,
       ln, *Li, *Lp, *Lnz, *List, size ;
    double l_jk, d_j, a_jk, l_jk_times_d_k, *Ax, *Lx ;

    Ai = A->i ;
    Ax = A->x ;
    Ap = A->p ;

    Li = L->i ;
    Lx = L->x ;
    Lp = L->p ;
    Lnz = L->nz ;
    ln = L->ncol ;

    List = Basis->list ;
    size = Basis->size ;

    /* ========================================================================= */
    /* ==== initializations ==================================================== */
    /* ========================================================================= */

    for (j = 0 ; j < ln ; j++)
    {
       Ahead [j] = EMPTY ;
       Link [j] = EMPTY ;
       W [j] = 0.0 ;
    }

    /* place each column of the basis set on the link list corresponding to */
    /* the smallest row index in that column */

    for (i = 0 ; i < size ; i++)
    {
       k = List [i] ;
       p = Ap [k] ;
       Aleft [k] = p ;
       j = Ai [p] ;
       Anext [k] = Ahead [j] ;
       Ahead [j] = k ;
    }

    /* ========================================================================= */
    /* === compute LDL' factorization, left to right =========================== */
    /* ========================================================================= */

    for (j = 0 ; j < ln ; j++)
    {

       /* ========================================================================= */
       /* === compute jth column of sigma*I + Af*Af' =========================== */
       /* ========================================================================= */

       W [j] = sigma ;

       /* for each nonzero A (j,k) in row j of A do */
       for (k = Ahead [j] ; k != EMPTY ; k = nextk)
       {
           /* determine next column of A that modifies column j of B */
           nextk = Anext [k] ;
```

```
    /* find the A (j,k) entry in column k of A */
    p = Aleft [k] ;

    /* compute W (j) += A (j,k) * A (j,k) */
    a_jk = Ax [p] ;
    W [j] += a_jk * a_jk ;

    /* compute with the remainder of A (j+1,k) */
    p++ ;
    p2 = Ap [k+1] ;
    if (p < p2)
    {
       /* place column k on link list of next row of A */
       nextj = Ai [p] ;
       Anext [k] = Ahead [nextj] ;
       Ahead [nextj] = k ;

       /* advance for the next row of A */
       Aleft [k] = p ;

       /* compute W (j+1:ln) += A (j+1:ln,k) * A (j,k) */
       for ( ; p < p2 ; p++)
       {
           W [Ai [p]] += Ax [p] * a_jk ;
       }
    }
}

/* ========================================================================== */
/* === compute jth column of L (unscaled) =================================== */
/* ========================================================================== */

/* for each nonzero L (j,k) in row j of L (:,1:j-1) do */
for (k = Link [j] ; k != EMPTY ; k = nextk)
{
    /* determine next column of L that modifies column j of L */
    nextk = Link [k] ;

    /* find the L (j,k) entry in column k of L */
    p = First [k] ;

    /* compute W (j) -= L (j,k) * L (j,k) * D (k) */
    l_jk = Lx [p] ;
    l_jk_times_d_k = l_jk * D [k] ;
    W [j] -= l_jk * l_jk_times_d_k ;

    /* compute with the remainder of L (j+1:n,k) */
    p++ ;
    p2 = Lp [k] + Lnz [k] ;
    if (p < p2)
    {
       /* place column k on link list of next column of L */
       nextj = Li [p] ;
       Link [k] = Link [nextj] ;
       Link [nextj] = k ;

       /* advance for the next column of L */
       First [k] = p ;

       /* compute W (j+1:ln) -= L (j+1:ln,k) * L (j,k) * D (k) */

       /* non-pointer version: */
       for ( ; p < p2 ; p++)
       {
           W [Li [p]] -= Lx [p] * l_jk_times_d_k ;
       }
    }
}

/* ========================================================================== */
/* === gather the results and scale ========================================= */
```

```
        /* ========================================================================== */

        p = Lp [j] ;
        p2 = p + Lnz [j] ;
        if (W [j] <= 0)
        {
            /* matrix is not symmetric positive definite */
            return (FALSE) ;
        }

        /* compute the diagonal, D (j,j) */
        d_j = W [j] ;
        D [j] = d_j ;
        W [j] = 0.0 ;

        /* gather and scale L (j+1:n,j) */
        if (p < p2)
        {

            /* prepare column j to modify its parent */
            nextj = Li [p] ;
            Link [j] = Link [nextj] ;
            Link [nextj] = j ;

            /* advance to the first offdiagonal entry in column j of L */
            First [j] = p ;

            /* gather column j of L, and divide by D (j,j) */
            for ( ; p < p2 ; p++)
            {
               i = Li [p] ;
               Lx [p] = W [i] / d_j ;
               W [i] = 0.0 ;
            }
        }


    }

    return (TRUE) ;
}
```

APPENDIX C
SOURCE CODE FOR SUPERNODE-SUPERNODE FACTORIZATION IN C

```
/* ========================================================================== */
/* ========================== ldlSupSup ===================================== */
/* ========================================================================== */

/*
    ldlSupSup : Numerical LDL' factorization using sup-sup .

    Matlab calling syntax is:

       [Lx, D] = ldlSupSup (A, Basis, sigma, Lp, Li, Lnz,Parent,tt) ;

    Authors :
       The authors of the code are Adrian Mascarenhas and Dr. Tim Davis
       (davis@cise.ufl.edu),University of Florida.

    Date :
       March 17, 2002

    The factorization is:

    Af = A (:, basis)
    B = sigma*I + Af*Af'
    LDL' = B

    The matrix L is stored in 0-based column form.
    A is a Matlab sparse matrix, which is held in 0-based column form.
    A must be in packed form, with sorted columns.

    The pattern of L has already been computed, and is in Lp, Li, and Lnz.
    This routine just computes Lx and D. All the other arguments are not
    modified. L->m is not used. The columns of L must be sorted.

    The diagonal of L is not stored (L is unit diagonal).

    Returns TRUE if the factorization was successful, FALSE otherwise.

    This routine performs a number of flops exactly equal to:

    sum (for each column j of Af) of (Anz (j)^2 + Anz (j)), to form B
    +
    sum (for each column j of L ) of (Lnz (j)^2 + 3*Lnz (j)), to factorize B

    where Anz (j) is the number of nonzeros in column j of Af,
    and Lnz (j) is the number of nonzero in column j of L below the diagonal.


*/

/* ========================================================================== */

#include "sparse.h"

#include "mkn.h"

#include "mKn.h"

#include "mKN.h"
```

```
/* ========================================================================== */
/* ==== mMKN ROUTINE========================================================= */
/* ========================================================================== */

/*
    This routine is used to do the block computation of cmod(J,K). This is
    highest level routine of the block computation.

*/


PRIVATE void mMKN
(
    int M,                  /* length of the updating block of supernode K */
    int K,                  /* width of supernode K */
    int N,                  /* width of the intersection set */
    double *pk [],          /* pk [32]; array of pointers to the columns of
                               the supernode K*/
    int *ik,                /* pointer to row indices of supernode K */
    double B [][],          /* B[32][32],copy of l_jk_times_d_k entries stored
                               by row in 32-by-32 array */
    double Wx [],           /* Wx [(maxColLength + 1) * SnodeSizeLimit],
                               initialized to 0.0 */
    int IntersectMap [],    /* IntersectMap [32]; mapping of columns of Supernode
                              J to Wx ; */
    int Map []              /* Map [0..L->ncol-1]; mapping of row indices of
                               supernode K to Wx */

)
{
    int i = 0 ;             /* loop variable */


    /*============== ITERATE ALONG  M  IN STEPS OF 4    ==========*/

    if (M >= 4)
    {
        for (i = 0 ; (i+4) <= M ; i += 4)
        {
            m_4KN (K,N,pk,&ik,B,Wx,IntersectMap,Map) ;

        }
    }



    /* special cases when M < 4 */
    switch (M % 4)
    {
        case 0:
            break ;
        case 1:
            /* M == 1 */
            m_1KN (K,N,pk,&ik,B,Wx,IntersectMap,Map) ;
            break ;
        case 2:
            /* M == 2 */
            m_2KN (K,N,pk,&ik,B,Wx,IntersectMap,Map) ;
            break ;
        case 3:
            /* M == 3 */
            m_3KN (K,N,pk,&ik,B,Wx,IntersectMap,Map) ;
            break ;
    }
}




/* ========================================================================== */
/* ==== ldlSupSup ROUTINE==================================================== */
```

```
/* ========================================================================== */

/*
    This is the main routine that does the sparse Cholesky factorization.
*/


PUBLIC int ldlSupSup
(
    Sparse_Matrix *A,           /* in packed form, with sorted columns */
    Sparse_Matrix *L,           /* with sorted columns */
    Set *Basis,                 /* the basis set, uses
                                   Basis->list [0..Basis->size-1]*/
    double sigma,               /* normally 1e-12 for LPDASA */

    double D [],                /* D diagonal of the matrix D */

    /*============== WORKSPACE ARRAYS ========= */

    double W [],                /* W [0..L->ncol-1] */

    int    Slist [],            /* Slist [0..L->ncol-1];contains
                                    supernodes in order */
    int    Schild [],           /* Schild [0..L->ncol-1],contains maximum
                                    superchild */
    int    Sp [],               /* Sp [0..L->ncol], contains boundary pointers
                                   for supernodes in Slist*/

    int    Link [],             /* Link [0..L->ncol-1] */
    int    First [],            /* First [0..L->ncol-1] */

    int    Ahead [],            /* Ahead [0..L->ncol-1],Ahead [j] is head
                                   of link list of cols of A that apply to
                                   col j of L */
    int    Anext [],            /* Anext [0..A->ncol-1], remainder of link
                                   list */
    int    Aleft [],            /* Aleft [0..A->ncol-1], Aleft [j] points to
                                   remainder of col j */
                                /* Aleft [j] is EMPTY if j is not in Basis */
    int Col_to_Snode_Map [],    /* Col_to_Snode_Map [0..L->ncol-1], maps
                                   columns to supernodes */
    int Map [],                 /* Map [0..L->ncol-1] */
    int R [],                   /* R [0..L->ncol-1] */
    double v [],                /* v [0..L->ncol-1] */
    int SnodeSizeLimit,         /* maximum width of supernode. set to 32 */
    double Wx []                /* Wx [(maxColLength + 1) * SnodeSizeLimit],
                                   initialized to 0.0 */
)
{

    /* ============= LOCAL VARIABLES ========== */

    int i,j;                    /* loop variables */
    int p,p1,p2;                /* index pointers within Li,Lx,Ai or Ax */
    int k;
    int nextk;                  /* next column k */
    int nextK;                  /* next supernode K */
    int nextj;                  /* next column j */
    int nextJ;                  /* next supernode J */
    int *Ai;                    /* A -> i */
    int *Ap;                    /* A -> p */
    int ln;                     /* L -> ncol */
    int *Li;                    /* L -> i */
    int *Lp;                    /* L -> p */
    int *Lnz;                   /* L -> nz */
    int *List;                  /* Basis-> list */
    int size;                   /* Basis-> size */
    int roffset;                /* offset for relative index R */
    int SppJ;                   /* Spp [J] */
    int SppJ_plus_1;            /* Spp [J+1] */
    int *RR ;                   /* RR = &R [roffset] */
    int FirstOffset;            /* offset for First for columns within a
```

```
                                    supernode */
int m;
int n;
int kk ;                            /* p2 - p1 or p2 - p */
int col;                            /* col index within Wx */
int Slist_ctr;                      /* counter for Slist */
int no_of_snodes;                   /* number of supernodes */
int sp_ctr;                         /* counter for Sp */
int Head_of_K;                      /* first column of supernode K */
int Head_of_J ;                     /* first column of supernode J */
double l_jk;                        /* scalar for L (j,k) */
double inv_d_j;                     /* scalar for 1/D(j) */
double a_jk;                        /* scalar for A(j,k) */
double l_jk_times_d_k;              /* scalar for L(j,k) * D(k) */
double *Ax;                         /* A -> x */
double *Lx ;                        /* L -> x */
double *Lxx;                        /* Lxx = &Lx [] */
int scol;                           /* column within superode */
double D_scol ;                     /* scalar for D [scol] */
int jindex,jindex1,jindex2;         /* index within Slist */
int jlast ;                         /* last column within supernode J */
int J,K;                            /* supernodes */
int *Spp;                           /* Spp = &Sp [] */
int one = 1 ;
int kk_minus_1;                     /* kk - 1 */
int CurrSnodeSize ;                 /* size of supernode */
double neg_l_jk_times_d_k ;         /* - l_jk_times_d_k */
int WxCol ;                         /* width of supernode ; Spp [J-1] - Spp [J] */
double *Wcol;                       /* Wcol = &Wx [] */
double *BB ;                        /* BB = & B [] */
double dj;                          /* scalar for D (j) in dense LDL */
double L_times_v ;                  /* scalar for L*v in dense LDL */
double B [32] [32] ;                /* array for storing l_jk_times_d_k scalars */
int Intersect [32] ;                /* columns in J to be updated by K */
int IntersectMap [32] ;             /* mapping of intersection columns into Wx */
int Mm,Kk,Nn;                       /* arguments for mMKN, M=Mm,K=Kk,N=Nn */
double * pk [32] ;                  /* pointers within Lx for columns of supernode
                                       K */
int * ik ;                          /* pointer within Li for Head_of_K */
int nintersect ;                    /* size of the Intersection set */


Ai = A->i ;
Ax = A->x ;
Ap = A->p ;

Li = L->i ;
Lx = L->x ;
Lp = L->p ;
Lnz = L->nz ;
ln = L->ncol ;

List = Basis->list ;
size = Basis->size ;



/*=========================================================*/
/*          SUPERNODAL ELIMINATION TREE CONSTRUCTION       */
/*=========================================================*/

/*============= CREATING Schild LIST ====================*/

for (i = 0 ; i < ln ; i++)
{
    Schild [i] = EMPTY ;
}

for (i = 0 ; i < ln ; i++)
{
    j = L->parent[i] ;
```

```
        if ( j > 0 )
        {
            if ((Lnz[i] - Lnz[j]) == 1) Schild[j] = i ;
        }
    }

    /*================ CREATING Slist ========================*/

    Slist_ctr = ln-1 ;    /* decrementing counter for Slist */
    no_of_snodes = 0 ;
    sp_ctr = ln ;        /* decrementing counter for Sp */


#define MARKED (-100)

    for (i = ln - 1 ; i >= 0 ; i--)
    {
        if (Schild [i] != MARKED) /* If i is unmarked */
        {
            no_of_snodes++ ;
            CurrSnodeSize = 1 ;
            Slist [Slist_ctr] = i ;
            Sp [sp_ctr] = Slist_ctr + 1 ;
            Slist_ctr-- ;
            sp_ctr-- ;
            j = Schild [i] ;
            Schild [i] = MARKED ;

            while (j > 0 && CurrSnodeSize < SnodeSizeLimit)
            {
                Slist [Slist_ctr] = j ;
                CurrSnodeSize ++ ;
                Slist_ctr-- ;
                k = Schild [j] ;
                Schild [j] = MARKED ;
                j = k ;
            }
        }
    }

    Sp [sp_ctr] = Slist_ctr + 1 ;


    Spp = &Sp [sp_ctr] ; /* Make Spp pointer to point to the first
                            NON-EMPTY value of Sp */


    /*==================== CREATING Col_to_Snode_Map ============== */

    for (i = 0 ; i < no_of_snodes ; i++)
    {
        k = Spp [i+1] ;
        for (j = Spp [i] ; j < k ; j++)
        {
            Col_to_Snode_Map [Slist [j]] = i ;
        }
    }


    /* ========================================================================= */
    /* ================== INITIALIZATIONS ====================================== */
    /* ========================================================================= */

    for (j = 0 ; j < ln ; j++)
    {
        Ahead [j] = EMPTY ;
        Link [j] = EMPTY ;
        W [j] = 0.0 ;
    }

  /* place each column of the basis set on the link list corresponding to the
  smallest row index in that column */
```

```
for (i = 0 ; i < size ; i++)
{
    k = List [i] ;
    p = Ap [k] ;
    Aleft [k] = p ;
    j = Ai [p] ;
    Anext [k] = Ahead [j] ;
    Ahead [j] = k ;
}



/* ========================================================================== */
/* ======================= MAIN LOOP  ======================================= */
/* ========================================================================== */

for (J = 0 ; J < no_of_snodes ; J++)
{

    SppJ = Spp [J] ;
    SppJ_plus_1 = Spp [J+1] ;

    WxCol = SppJ_plus_1 - SppJ ; /* width of supernode J */

    Head_of_J = Slist [SppJ] ; /* First node of Supernode J */

    p = Lp [Head_of_J] ;
    p2 = p + Lnz [Head_of_J] ;


    /* WxCol == 1 do Supernode-Column
       WxCol > 1 do Supernode-Supernode */

    if (WxCol == 1)
    {
        /* ================================================================ */
        /* === SUPERNODE COLUMN (supcol) FACTORIZATION ================== */
        /* ================================================================ */


         j = Head_of_J ;

        /* =========SUP-COL: COMPUTE jth COLUMN OF sigma*I + Af*Af'===== */

        W [j] = sigma ;

        /* for each nonzero A (j,k) in row j of A do */
        for (k = Ahead [j] ; k != EMPTY ; k = nextk)
        {
            /* determine next column of A that modifies column j of B */
            nextk = Anext [k] ;

            /* find the A (j,k) entry in column k of A */
            p = Aleft [k] ;


            /* compute W (j) += A (j,k) * A (j,k) */
            a_jk = Ax [p] ;
            W [j] += a_jk * a_jk ;

            /* compute with the remainder of A (j+1,k) */
            p++ ;
            p2 = Ap [k+1] ;
            if (p < p2)
            {
                /* place column k on link list of next row of A */
                nextj = Ai [p] ;
                Anext [k] = Ahead [nextj] ;
                Ahead [nextj] = k ;

                /* advance for the next row of A */
```

```
        Aleft [k] = p ;

        /* compute W (j+1:ln) += A (j+1:ln,k) * A (j,k) */
        for ( ; p < p2 ; p++)
        {
            W [Ai [p]] += Ax [p] * a_jk ;
        }
    }
}



/* =======SUP-COL: COMPUTE jth COLUMN OF L (UNSCALED)=========== */

for (K = Link [J] ; K != EMPTY ; K = nextK)
{

    /* determine next supernode that modifies column j of L */
    nextK = Link [K] ;

    /* calculate offset for the first col in the supernode K */
    Head_of_K = Slist [Spp [K]] ;
    FirstOffset = First [K] - Lp [Head_of_K] ;

    p1 = First [K] ;
    p2 = Lp [Head_of_K] + Lnz [Head_of_K] ;

    kk = p2 - p1 ;

    jindex1 = Spp [K] ;
    jindex2 = Spp [K+1] ;

    /* check if supernode K has more than one column */
    if ((jindex2 - jindex1) > 1 && kk > 0 )
    {

        scol = Slist [jindex1] ;

        /* compute p1 for scol within K using FirstOffset */
        p1 = Lp [scol] + FirstOffset-- ;



        /* compute scalar */
        Lxx = &Lx [p1] ;
        l_jk = Lxx [0] ;
        l_jk_times_d_k = l_jk * D [scol] ;

        /* initialize Wx */
        for (k = 0 ; k < kk ; k++)
        {
            Wx [k] = Lxx [k] * l_jk_times_d_k ;
        }

        jindex1++ ;

        /* compute Wx for remaining columns in supernode K */
        for ( ; jindex1 < jindex2 ; jindex1++)
        {
            scol = Slist [jindex1] ;

            /* compute p1 for scol within K using FirstOffset */
            p1 = Lp [scol] + FirstOffset-- ;



            /* compute scalar */
            Lxx = &Lx [p1] ;
            l_jk = Lxx [0] ;
            l_jk_times_d_k = l_jk * D [scol] ;
```

```
                        /* accumulate updates into dense vector Wx */
                        for (k = 0 ; k < kk ; k++)
                        {
                            Wx [k] += Lxx [k] * l_jk_times_d_k ;
                        }
#                     }

                    /* scatter Wx [] into W */
                    p = First [K] ;
                    for (i = 0 ; i < kk ; i++, p++)
                    {
                        W [Li [p]] -= Wx [i] ;
                    }

                }
                else
                {
                 /* supernode K has only one column */
                    if ( kk > 0)
                    {
                        /* compute scalar */
                        l_jk = Lx [p1] ;
                        l_jk_times_d_k = l_jk * D [Head_of_K] ;

                        /* scatter into W [...] */
                        p = First [K] ;
                        for (i = 0 ; i < kk ; i++, p++)
                        {
                            W [Li [p]] -= Lx [p] * l_jk_times_d_k ;
                        }
                    }
                }

                /* advance for the next column of L */
                First [K]++;

                /*place supernode K on link list of next singleton supernode
                 of L */
                nextj = Li [First [K]] ;
                nextJ = Col_to_Snode_Map [nextj] ;
                Link [K] = Link [nextJ] ;
                Link [nextJ] = K ;
            }


            /* ============ SUP-COL: GATHER RESULTS SO FAR ================= */

            p = Lp [j] ;
            p2 = p + Lnz [j] ;

            D [j] = W [j] ;
            W [j] = 0.0 ;

            for ( ; p < p2 ; p++)
            {
                i = Li [p] ;
                Lx [p] = W [i] ;
                W [i] = 0.0 ;
            }


            /* =============== SUP-COL : SCALING =========================== */

            p = Lp [j] ;
            p2 = p + Lnz [j] ;
            if (D [j] <= 0)
            {
                /* matrix is not symmetric positive definite */
                return (FALSE) ;
            }

            /* scale L (j+1:n,j) */
```

```
        if (p < p2)
        {

            /* the diagonal, D (j,j) */
            inv_d_j = 1.0 / D [j] ;

            /* advance to the first offdiagonal entry in column j of L */
            First [J] = Lp [j] ;

            /* divide col j of L by D (j,j) */


            for ( ; p < p2 ; p++)
            {
                Lx [p] *= inv_d_j ;
            }
        }


    /*=========SUP-COL: PREPARE SUPERNODE J TO MODIFY ITS PARENT===== */
    p = First [J] ;
    p2 = p + Lnz [Head_of_J] ;

    if (p < p2)
    {
        nextj = Li [p] ;
        nextJ = Col_to_Snode_Map [nextj] ;
        Link [J] = Link [nextJ] ;
        Link [nextJ] = J ;
    }




}
else
{
     /* WxCol > 1 */

    /* ============================================================ */
    /* ======== SUPERNODE - SUPERNODE (supsup) FACTORIZATION ======== */
    /* ============================================================ */

    /* ================= SUP-SUP: CREATE Map FOR J ========== */

    Head_of_J = Slist [SppJ] ; /* First node of Supernode J */

    p = Lp [Head_of_J] ;
    p2 = p + Lnz [Head_of_J] ;

    Map [Head_of_J] = 0 ;
    for (i = 1 ; p < p2 ; p++, i++)
    {
        Map [Li [p]] = i * WxCol ;
    }



    /* ======== SUP-SUP: COMPUTE sigma*I + Af*Af' FOR SUPERNODE J ====== */

    for (jindex = SppJ ; jindex < SppJ_plus_1 ; jindex++)
    {
        j = Slist [jindex] ;

        W [j] = sigma ;

        /* for each nonzero A (j,k) in row j of A do */
        for (k = Ahead [j] ; k != EMPTY ; k = nextk)
        {
            /* determine next column of A that modifies column j of B */
            nextk = Anext [k] ;
```

```
            /* find the A (j,k) entry in column k of A */
            p = Aleft [k] ;


            /* compute W (j) += A (j,k) * A (j,k) */
            a_jk = Ax [p] ;
            W [j] += a_jk * a_jk ;

            /* compute with the remainder of A (j+1,k) */
            p++ ;
            p2 = Ap [k+1] ;
            if (p < p2)
            {
                /* place column k on link list of next row of A */
                nextj = Ai [p] ;
                Anext [k] = Ahead [nextj] ;
                Ahead [nextj] = k ;

                /* advance for the next row of A */
                Aleft [k] = p ;

                /* compute W (j+1:ln) += A (j+1:ln,k) * A (j,k) */
                for ( ; p < p2 ; p++)
                {
                    W [Ai [p]] += Ax [p] * a_jk ;
                }
            }
        }


        /*=============SUP-SUP: GATHER W INTO Wx ============*/
        p = Lp [j] ;
        p2 = p + Lnz [j] ;

        col = Map [j] / WxCol ;

        Wcol = Wx + col ;
        for ( ; p < p2 ; p++)
        {
            i = Li [p] ;
            Wcol [Map [i]] = W [i] ;
            W [i] = 0.0 ;
        }

        /* gather the diagonal */
        Wcol [Map [j]] = W [j] ;
        W [j] = 0.0 ;

    }




    /* ======= SUP-SUP : COMPUTE SUPERNODE J OF L (UNSCALED) =========== */

    for (K = Link [J] ; K != EMPTY ; K = nextK)
    {

        /* determine next supernode of L that modifies column j of L */
        nextK = Link [K] ;


        Head_of_K = Slist [Spp [K]] ; /* first node in supernode K */

        p = First [K] ;
        p2 = Lp [Head_of_K] + Lnz [Head_of_K] ;
```

```
/*======= SUP-SUP: CALCULATE THE INTERSECTION SET OF J With K == */

jlast = Slist [SppJ_plus_1 - 1] ;
nintersect = 0 ;
for (p = First [K], k = 0; p < p2 && Li [p] <= jlast; p++, k++ )
{
    nintersect++ ;
    i = Li [p] ;
    Intersect [k] = i ;

    /* create the mapping for the intersection set to columns
        of Wx */
    IntersectMap [k] = Map [i] / WxCol ;

    /* create Relative index R for supernode K */
    R [k] = Map [i] ;
}


/*===== SUP-SUP: COPY Jj PART OF K INTO B AND CREATE pk,ik ======*/
/*===== POINTERS                                              ====*/

jindex1 = Spp [K] ;
jindex2 = Spp [K+1] ;


/* calculate offset for First for cols in supernode K */
FirstOffset = First [K] - Lp [Head_of_K] ;
ik = &Li [First [K] + nintersect] ;

for (k=0 ; jindex1 < jindex2 ; jindex1++, k++)
{
    scol = Slist [jindex1] ;

    p1 = Lp [scol] + FirstOffset-- ;
    Lxx = &Lx [p1] ;

    /* copying into B */
    BB = & B [k][0] ;
    D_scol = D [scol] ;
    for (i=0 ; i < nintersect ; i++)
    {
        BB [i] = Lxx [i] * D_scol ;
    }

    /* setting pk, ik pointers */
    p2 = p1 + nintersect ;
    pk [k] = &Lx [p2] ;



}



/*==SUP-SUP: COMPUTATION OF DIAGONAL PART OF J using SUP-COL)====*/

roffset = 0 ;
for (jindex = 0 ; jindex < nintersect ; jindex++)
{
    j = Intersect [jindex] ;

    col = Map [j] / WxCol ;

    /* calculate offset for First for cols in supernode K */
    FirstOffset = First [K] - Lp [Head_of_K] ;

    p1 = First [K] ;
    p2 = Lp [Head_of_K] + Lnz [Head_of_K] ;
```

```
/*length of diagonal part */
kk = nintersect - jindex ;

jindex1 = Spp [K] ;
jindex2 = Spp [K+1] ;

/* check if supernode K has more than one column */
if ((jindex2 - jindex1) > 1 && kk > 0 )
{

    scol = Slist [jindex1] ;

    /* compute p1 for scol within K using FirstOffset */
    p1 = Lp [scol] + FirstOffset-- ;

    /* compute scalar */
    Lxx = &Lx [p1] ;
    l_jk = Lxx [0] ;
    l_jk_times_d_k = l_jk * D [scol] ;

    /* initialize W */
    for (k = 0 ; k < kk ; k++)
    {
        W [k] = Lxx [k] * l_jk_times_d_k ;
    }

    jindex1++ ;

    /* compute W for remaining columns in supernode K */
    for ( ; jindex1 < jindex2 ; jindex1++)
    {
        scol = Slist [jindex1] ; /*col within curr supernode*/

        /* compute p1 for scol within K using FirstOffset */
        p1 = Lp [scol] + FirstOffset-- ;

        /* compute scalar */
        Lxx = &Lx [p1] ;
        l_jk = Lxx [0] ;
        l_jk_times_d_k = l_jk * D [scol] ;


        /* accumulate the updates into a dense vector W */
        for (k = 0 ; k < kk ; k++)
        {
            W [k] += Lxx [k] * l_jk_times_d_k ;
        }
    }


    /* gather W into Wx */
    RR = &R [roffset] ;
    Wcol = Wx + col ;
    for (i = 0 ; i < kk ; i++)
    {

        Wcol [RR [i]] -= W [i] ;
        W [i] = 0.0 ;
    }

}
else
{
    /* supernode K has only one column */
    if ( kk > 0)
    {
        /* compute scalar */
        l_jk = Lx [p1] ;

        l_jk_times_d_k = l_jk * D [Head_of_K] ;
```

```
                    /* gather into Wx [...] */
                    p = First [K] ;
                    RR = &R [roffset] ;
                    Wcol = Wx + col ;
                    for (i = 0 ; i < kk ; i++, p++)
                    {
                        Wcol [RR [i]] -= Lx [p] * l_jk_times_d_k ;
                    }


                }
            }

            /* advance for the next supernodal column of L */
            p = First [K] ;
            p2 = Lp [Head_of_K] + Lnz [Head_of_K] ;
            if (p < p2)
            {
                First [K]++;
                roffset ++ ;
            }

        }

        /*=====PLACE SUPERNODE K ON Link LIST OF NEXT SUPERNODE OF L ====*/

        p = First [K] ;
        p2 = Lp [Head_of_K] + Lnz [Head_of_K] ;
        if (p < p2)
        {
            nextj = Li [First [K]] ;
            nextJ = Col_to_Snode_Map [nextj] ;
            Link [K] = Link [nextJ] ;
            Link [nextJ] = K ;
        }


        /*======= SUP-SUP: BLOCK COMPUTATION of Cmod (J,K) =============*/

        Mm = Lp [Head_of_K] + Lnz [Head_of_K] - First [K] ;
        Kk = Spp [K+1] - Spp [K] ;
        Nn = nintersect ;


        /* call to the routine mMKN, see definition before the ldlSupSup*/
        mMKN (Mm,Kk,Nn,pk,ik,B,Wx,IntersectMap,Map) ;


    }

    /* ====== SUP-SUP: DENSE A=LDL' CHOLESKY FACTORIZATION OF Wx ======= */

    /* [m,n] = size (A) where A = Wx */
    m = Lnz [Head_of_J] + 1 ; /* including the diagonal */
    n = SppJ_plus_1 - SppJ ;

    /* n == 1 -> A has only one column
        n > 1 -> A has more than one columns */
    if (n == 1)
    {
        Wcol = &Wx [WxCol] ;
        for (i = 1; i < m ;i++)
        {
            Wcol [0] /= Wx [0] ;
            Wcol += WxCol ;
        }
    }
    else
    {
        for (j = 0 ; j < n ; j++)
        {
```

```
        Wcol = &Wx [j * WxCol] ;
        for (i=0 ; i < j ; i++)
        {
            /* v(i) = L(j,i) d(i) */
             v [i] = Wcol [i] * Wx [i * WxCol + i] ;
        }

        /* d(j) = A (j,j) - L (j,1:j-1) * v(1:j-1) */
        L_times_v = 0.0 ;
        for (i = 0 ; i < j ; i++)
        {
             L_times_v += Wcol [i] * v [i] ;
        }
        Wcol [j] -= L_times_v ;
        dj = Wcol [j] ;


        /* L(j+1:n,j) = (A(j+1:n,j) - L(j+1:n,1:j-1) * v(1:j-1))/d(j)*/
        Wcol += WxCol ;
        for (i = j+1 ; i < m ; i++)
        {
            L_times_v = 0.0 ;
            for (k = 0 ; k < j ; k++)
            {
                L_times_v += Wcol [k] * v [k] ;
            }

            Wcol [j] = (Wcol [j] - L_times_v ) / dj ;
            Wcol += WxCol ;
        }

    }
}


/* ======= SUP-SUP: SCATTER Wx INTO Lx,D ========================== */

for (jindex = SppJ, col = 0; jindex < SppJ_plus_1 ; jindex++,col++)
{
    j = Slist [jindex] ;

    /* store D */
    Wcol = &Wx [col * WxCol] ;
    D [j] = Wcol [col] ;
    Wcol [col] = 0.0 ;


    if (D [j] <= 0)
    {
        /* matrix is not symmetric positive definite */
        return (FALSE) ;
    }


    p = Lp [j] ;
    p2 = p + Lnz [j] ;

    /* scatter into Lx*/
    Wcol += WxCol ;
    for (i=1 ; p < p2 ; p++,i++)
    {
        Lx [p] = Wcol [col] ;
        Wcol [col] = 0.0 ;
        Wcol += WxCol ;
    }


}


/*=========== SUP-SUP: PREPARE SUPERNODE J TO MODIFY ITS PARENT ==== */
```

```
        jlast = Slist [SppJ_plus_1 - 1] ;
        p = Lp [jlast] ;
        p2 = p + Lnz [jlast] ;

        if (p < p2)
        {
            nextj = Li [p] ;
            nextJ = Col_to_Snode_Map [nextj] ;
            Link [J] = Link [nextJ] ;
            Link [nextJ] = J ;


            First [J] = Lp [Head_of_J] + (SppJ_plus_1 - SppJ - 1) ;
        }



     }
    }

    /*==================== END OF MAIN LOOP ==============*/


  return (TRUE) ;

} /* end of file */
```

LIST OF REFERENCES

[1] C. ASHCRAFT AND R. GRIMES, *The influence of relaxed supernode partitions on the multifrontal method*, ACM Trans. of Math. Software, 15(1989), pp. 291-309.

[2] T. H. CORMEN, C. E. LEISERSON AND R. L. RIVEST, *Introduction to algorithms*, MIT press, Cambridge, MA, 1990.

[3] T. A. DAVIS, J. R. GILBERT, S. I. LARIMORE AND E. G. NG, *A column approximate minimum degree ordering algorithm*, Technical Report TR-00-005, Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 2000.

[4] T. A. DAVIS AND W. W. HAGER, *Modifying a sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 20(1999), pp. 606-627.

[5] A. GEORGE AND J. LIU, *Computer solution of large sparse positive definite systems*, Prentice-Hall, 1981.

[6] G. H. GOLUB AND C. V. LOAN, *Matrix computations*, John Hopkins University Press, 1996.

[7] W. W. HAGER, *The LP dual active set algorithm*, in High Performance Algorithms and Software in Nonlinear Optimization, R. De Leone, A. Murli, P. M. Pardalos, and G. Toraldo, eds., Kluwer, Dordrecht, 1998, pp. 243-254.

[8] J. W. H. LIU, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11(1990), pp. 134-172.

[9] J. W. H. LIU, E. G. NG AND B. W. PEYTON, *On finding supernodes for sparse matrix computations*, SIAM J. Matrix Anal. Appl., 14(1993), pp. 242-252.

[10] E. G. NG, *Supernodal symbolic Cholesky factorization on a local-memory multiprocessor*, Parallel Computing, 19(1993), pp. 153-162.

[11] E. G. NG AND B. W. PEYTON, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput., 14(1993), pp. 1034-1056.

[12] E. G. NG AND B. W. PEYTON, *A supernodal Cholesky factorization algorithm for shared-memory multiprocessor*, SIAM J. Sci. and Stat. Comput., 14(1993), pp. 761-769.

[13] E. ROTHBERG AND A. GUPTA, *Efficient sparse matrix factorization on high-performance workstations – Exploiting the memory hierarchy*, ACM Transactions on Mathematical Software, 17 (1991), pp. 313-334.

[14] E. ROTHBERG AND A. GUPTA, *Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations*, In proceedings of Supercomputing, 1990, pp. 232-243.

BIOGRAPHICAL SKETCH

Adrian Mascarenhas was born in Mumbai, India, in November 1977. He received his Bachelor of Engineering degree in Computer Engineering from the University of Mumbai (V.J.T.I.) in August 1999. He received his Master of Science degree in Computer Engineering from the University of Florida in May 2002.