Sparse Gaussian Elimination on High Performance Computers

by

Xiaoye S. Li

B.S. (Tsinghua University) 1986
M.S., M.A. (Penn State University) 1990

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

James W. Demmel, Chair
Katherine A. Yelick
John R. Gilbert
Phillip Colella

1996

Sparse Gaussian Elimination on High Performance Computers

Abstract

Sparse Gaussian Elimination on High Performance Computers

by

Xiaoye S. Li
Doctor of Philosophy in Computer Science

University of California at Berkeley

James W. Demmel, Chair

This dissertation presents new techniques for solving large sparse unsymmetric linear systems on high performance computers, using Gaussian elimination with partial pivoting. The efficiencies of the new algorithms are demonstrated for matrices from various fields and for a variety of high performance machines.

In the first part we discuss optimizations of a sequential algorithm to exploit the memory hierarchies that exist in most RISC-based superscalar computers. We begin with the left-looking supernode-column algorithm by Eisenstat, Gilbert and Liu, which includes Eisenstat and Liu's symmetric structural reduction for fast symbolic factorization. Our key contribution is to develop both numeric and symbolic schemes to perform supernode-panel updates to achieve better data reuse in cache and floating-point registers. A further refinement, a two-dimensional matrix partitioning scheme, enhances performance for large matrices or machines with small caches. We conduct extensive performance evaluations on several recent superscalar architectures, such as the IBM RS/6000-590, MIPS R8000 and DEC Alpha 21164, and show that our new algorithm is much faster than its predecessors. The advantage is particularly evident for large problems. In addition, we develop a detailed model to systematically choose a set of blocking parameters in the algorithm.

The second part focuses on the design, implementation and performance analysis of a shared memory parallel algorithm based on our new serial algorithm. We parallelize the computation along the column dimension of the matrix, assigning one block of columns (a panel) to a processor. The parallel algorithm retains the serial algorithm's ability to reuse cached data. We develop a dynamic scheduling mechanism to schedule tasks onto available processors. One merit of this approach is the ability to balance work load automatically. The algorithm attempts to schedule independent tasks to different processors. When this is not possible in the later stage of factorization, a *pipeline* approach is used to coordinate dependent computations. We demonstrate that the new parallel algorithm is very efficient on shared memory machines with modest numbers of processors, such as the SGI Power Challenge, DEC AlphaServer 8400, and Cray C90/J90. We also develop performance models to study available concurrency and identify performance bottlenecks.

James W. Demmel
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my advisor Jim Demmel for his guidance through my trials of graduate study at Berkeley. Jim has always been available for discussions and providing insightful opinions so that I could pursue research in the right direction. I greatly appreciate his enthusiasm in many subject areas in scientific computing, and his ability to view an idea from many different perspectives. His broad knowledge in both mathematics and computer science has helped my research in numerous ways.

Much of the work in this dissertation benefited from discussions with John Gilbert of Xerox Palo Alto Research Center. I have learned from him many graph algorithms and combinatorial tools required to manipulate sparse matrices. I am truly grateful to him for passing on to me his knowledge and long-time experience in sparse matrix algorithms and software.

I thank Beresford Parlett and Kathy Yelick, not only for serving on my dissertation committee, but also for their introducing me to the field of sparse matrix computations in their classes. Since then, I developed a great interest in this area, and discovered open problems which lead to the research work in this dissertation. I also thank Phil Colella for serving on my qualifying exam and dissertation committee.

I thank Esmond Ng of Oak Ridge National Lab for correspondences on the issues of nonzero structure prediction, which helped design the memory management scheme discussed in Chapter 5. Ed Rothberg of Silicon Graphics not only provided me access to the SGI Power Challenge, but also helped improve performance of our algorithm. In particular, he inspired me to develop the matrix-two-vector-multiply kernel discussed in Chapter 4.

It has been a great pleasure to work with the members of the LAPACK team at Berkeley. Graduate life would be less interesting and productive without their company in the office.

I truly thank my husband Suganda for his understanding, patience, and constant support throughout my graduate study.

Finally, I acknowledge support from NSF grant ASC–9313958, DOE grant DE–FG03–94ER25219, UT Subcontract No. ORA7453.02 from ARPA Contract No. DAAH04-95-1-0077, DOE grant DE–FG03–94ER25206, NSF Infrastructure grants CDA–8722788 and CDA–9401156, and support from Xerox Corporation through a summer internship at Xerox PARC.

Chapter 1

Introduction

We investigate new techniques in direct methods for solving large sparse nonsymmetric linear systems of equations. Such linear systems arise in diverse areas such as sparse eigenvalue computation, solving discrete finite-element problems, device and circuit simulation, linear programming, chemical engineering, and fluid dynamics modeling. These demanding and important applications can immediately benefit by any improvements in linear equation solvers.

The motivation for this research is two-fold. First, existing sparse algorithms and codes are much slower than their dense counterparts, especially on modern RISC workstations. These machines typically have multiple pipelined functional units, pipelined floating-point units, and fast but relatively small cache memory to hide the main memory access latency. These workstations provide a cost-effective way to achieve high performance and are more widely available than traditional supercomputers. The emergence of these novel architectures has motivated the redesign of linear algebra software for dense matrices, such as the well-known LAPACK library [7]. The earlier algorithms used in LINPACK and EISPACK are inefficient because they often spend more time moving data than doing useful floating-point operations. One of the chief improvements of many new algorithms in LAPACK is to use *block* matrix operations, whose improved data locality permits exploitation of cache and multiple functional units. Significant performance gains have been observed over the unblocked algorithms [7]. The analogous algorithmic improvements are much harder for sparse matrix algorithms, because of their irregular data structures and memory access patterns. This thesis will address this issue by studying one class of such algorithms, sparse LU factorization. In essence, our new algorithm identifies and exploits the dense blocks (*supernodes*) that emerge during the sparse LU factorization.

Our second motivation is to exploit parallelism in order to solve the ever larger linear systems arising in practice. Twenty years ago, the day-to-day linear systems people wanted to solve usually had only tens or hundreds of unknowns (see Table 1.6.1 in Duff et al. [36]). In a more recent and widely used Harwell-Boeing collection of sparse matrices [37], the largest nonsymmetric system has about 5000 unknowns. Today, it is not uncommon to encounter systems involving 50,000 unknowns, for example, from three-dimensional simulations. Examples of several large matrices will be used in our study (see Table 4.1). Solving such systems is made possible by faster processors and larger and cheaper main memory.

However, efficient algorithms are crucial to take advantage of the new architectures. Furthermore, parallel processing capabilities enlarge the problem domains under consideration. We will primarily focus on the runtime and storage efficiency of the new algorithms on large problems, in particular, the largest ones that can fit in the main memory.

In parallel processing, computational power can be multiplied by connecting tens or hundreds of off-the-shelf processors. Memory access locality is even more important for high performance and scalability than on a sequential machine. For example, on a bus-connected system, frequent access to globally shared memory by different processors will saturate the shared bus. On a distributed memory machine, processors communicate to each other by explicitly sending messages. Accessing non-local data by receiving a message is often orders of magnitude more costly than accessing local memory. It is vital to design our algorithms to be aware of the non-uniform memory access times inherent in all machines. We will show that the concept of locality plays a central role in designing and implementing efficient algorithms on modern high performance architectures.

The remainder of the dissertation is organized as follows. Chapter 2 briefly reviews the existing factorization techniques used to solve general sparse unsymmetric linear systems of equations. It discusses and compares primary algorithms used, which are called left-looking, right-looking and multifrontal. It also identifies potential for improvements. Chapter 3 introduces the fundamental concepts used in sparse column methods, including the column elimination tree and unsymmetric supernodes. Column methods are the main focus of this thesis. In Chapter 4 we study various techniques to improve an existing sequential algorithm, and quantify the performance gains on a variety of machines. In Chapter 5 we develop a parallel algorithm for shared memory machines, and demonstrate its efficiency on a set of parallel computers. We also develop a theoretical model to predict maximum speedup attainable by the algorithm. Finally, Chapter 6 summarizes the results from this research and discusses the future extensions of this work. Our main contributions are (1) new techniques to enhance the performance of an existing sequential algorithm for hierarchical memory machines; (2) a new parallel algorithm and its performance analysis and modeling for shared memory machines; and (3) portable software for a variety of high-performance computers.

Chapter 2

Sparse Solvers Using Direct Factorizations

In this chapter, we give a brief overview of the algorithms and software that use direct factorization, or Gaussian elimination, to solve a large sparse linear system of equations $Ax = b$. In a direct method, the matrix A is first decomposed into the product of lower triangular matrix L and upper triangular matrix U (or L^T if A is symmetric positive definite). Then the two triangular systems $Ly = b$ and $Ux = y$ are solved to obtain the solution x. In this solution process, the LU factorization usually dominates the execution time. Over the years a great deal of research effort has been devoted to finding efficient ways to perform this factorization. Although many different algorithms exist, the generic Gaussian elimination algorithm can be written as the following three nested loops:

$$
\begin{array}{l}
\textbf{for} \ \underline{\hspace{2cm}} \ \textbf{do} \\
\quad \textbf{for} \ \underline{\hspace{2cm}} \ \textbf{do} \\
\quad \quad \textbf{for} \ \underline{\hspace{2cm}} \ \textbf{do} \\
\quad \quad \quad a_{ij} \leftarrow a_{ij} - (a_{ik} * a_{kj})/a_{kk} \ ; \\
\quad \quad \textbf{end for}; \\
\quad \textbf{end for}; \\
\textbf{end for};
\end{array}
\tag{2.1}
$$

The loop indices have variable names i, j, and k, but they will have different ranges. Six possible permutations of i, j and k are possible in the three nested loops. Dongarra et al. [29] studied the performance impact of each permutation for dense LU factorization algorithms on vector pipeline machines. Although the generic algorithm is very simple, significant complications in its actual implementation arise from sparsity, the need for numerical pivoting and diverse computer architectures. In the update in Equation (2.1), an a_{ij} that was originally zero will become nonzero if both a_{ik} and a_{kj} are nonzero. This new nonzero entry is called *fill*. The fills incur more floating-point arithmetic and more storage. Also in Equation (2.1), if the pivot a_{kk} at step k is too small, the updated entry may become large in magnitude. The large element will cause instability when it is added to other smaller entries of A. It is therefore crucial to find a good elimination ordering, corresponding to permuting the rows and/or columns of the original A into $\bar{A} = PAQ$, so that when factorizing $\bar{A}$ the number of fills introduced is minimal and the element growth is small. Besides preserving sparsity and maintaining numerical stability, the efficiency of an algorithm also depends very much on how the i-j-k loops are organized around Equation (2.1), and on the data structures used to manipulate sparsity.

The problem of solving sparse symmetric positive definite systems has been studied extensively and now is fairly well understood. For such a system, the pivots can be chosen down the diagonal in any order without losing numerical stability. Therefore, the solution process can be divided into four distinct phases:

1. Finding a good ordering P so that the lower triangular Cholesky factor L of $PAP^T = LL^T$ suffers little fill;

2. Symbolic factorization to determine the nonzero structure of L;

3. Numeric factorization to compute L;

4. Solution of $Ly = b$ and $L^T x = y$.

In the first phase, although it is computationally expensive (NP-hard) to find an optimal P in terms of minimizing fills, many heuristics have been used successfully in practice, such as variants of minimum degree orderings [3, 10, 38, 52] and various dissection orderings based on graph partitioning [15, 58, 92] or hybrid approaches [13, 18, 76].

Two important data structures have been introduced in efficient implementations of the Cholesky factorization. One is the *elimination tree* and another is the *supernode*. The elimination tree [100] is defined for the Cholesky factor L. Each node in the tree corresponds to one row/column of the matrix. The edges in the tree can be succinctly represented by the following $parent[*]$ vector:

$$parent[j] = \min \{ \ i > j \mid l_{ij} \neq 0 \ \} \ .$$

In graph-theoretic terms, the elimination tree is simply the transitive reduction [2] of the directed graph $G(L^T)$,[1] see Liu [83]. It is the minimal subgraph of G that preserves paths and provides the smallest possible description of column dependencies in the Cholesky factor. Liu [83] discusses the use of elimination trees in various aspects of sparse algorithms, including reordering, symbolic and numeric factorizations, and parallel elimination.

The supernode structure has long been recognized and employed in enhancing the efficiency of both the minimum degree ordering [40, 48] and the symbolic factorization [102]. A supernode is a set of contiguous columns in the Cholesky factor L that share essentially the same sparsity structure. More recently, supernodes have also been introduced in numeric factorization and triangular solution, in order to make better use of vector registers or cache memory. Indeed, supernodal [12] and multifrontal [41] elimination allow the use of dense vector operations for nearly all of the floating-point computation, thus reducing the symbolic overhead in numeric factorization to a smaller fraction. Overall, the Megaflop

[1] The directed graph of a square matrix has n vertices corresponding to n rows/columns. An edge from i to j indicates a nonzero in row i and column j of the matrix.

rates of modern sparse Cholesky codes are comparable to those of dense solvers [87, 95] for some classes of problems, such as those with a great deal of fill.

The triangular solution in phase 4 is also dictated by the elimination structure, where the forward substitution ($Ly = b$) proceeds from the leaves toward the root of the tree and the back substitution ($L^T x = y$) proceeds from the root toward the leaves. Since triangular solution requires many fewer floating-point operations than the factorization, the time it takes constitutes a small fraction of the total time, typically under 5% in a sequential algorithm.

For general unsymmetric systems, where pivoting is required to stabilize the underlying algorithm, progress has been less satisfactory than with Cholesky factorization. A major distinction from symmetric positive definite systems is that the nonzero structure of the factored matrices cannot be determined in advance of the numeric factorization. So both symbolic and numeric factorizations must interleave. In addition, some algorithms include a column pivoting strategy to preserve sparsity during the elimination process, mixing the ordering, symbolic and numeric phases altogether.

Gilbert and Liu [68] introduced *elimination dags* (directed acyclic graphs), or edags for short, to study the structure changes during unsymmetric LU factorization. The edags are transitive reductions of the graphs $G(L^T)$ and $G(U)$. Since L and U usually have different structures, the edags for L and U are often distinct. Furthermore, one node may have more than one parent in the edag, in contrast to the tree structure. The dags characterize the triangular factors L and U in the same way that the elimination tree characterizes the Cholesky factor. They are the minimal subgraphs of $G(L^T)$ and $G(U)$ that preserve paths and also provide the smallest possible description of the column dependencies during unsymmetric elimination.

Recent research on unsymmetric systems has concentrated on two basic approaches: submatrix-based (also called right-looking) methods and column-based (also called left-looking) methods.[2] Submatrix methods use k in the outer loop for Equation (2.1). They typically use a combination of some form of Markowitz ordering [86] and numerical threshold pivoting [36] to choose the pivot element from the uneliminated submatrix. To illustrate this, let us assume that the first $k-1$ stages of Gaussian elimination have been completed. We may partition A in the blocked form

$$A = \begin{pmatrix} A_{KK} & A_{K\tilde{K}} \\ A_{\tilde{K}K} & A_{\tilde{K}\tilde{K}} \end{pmatrix},$$

where $K = (1:k-1)$, $\tilde{K} = (k:n)$, and A_{KK} is nonsingular.[3] Then the factored form of A can be written as

$$A = \begin{pmatrix} L_{KK} & 0 \\ L_{\tilde{K}K} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & R_{\tilde{K}\tilde{K}} \end{pmatrix} \begin{pmatrix} U_{KK} & U_{K\tilde{K}} \\ 0 & I \end{pmatrix}.$$

[2] Row methods are exactly analogous to column methods, and codes of both sorts exist. We will use column terminology in this thesis; those who prefer rows may interchange the terms throughout.

[3] We use Matlab notation for integer ranges and submatrices: $r:s$ or $(r:s)$ is the range of integers $(r, r+1, \ldots, s)$. If I and J are sets of integers, then $A(I,J)$ is the submatrix of A with rows whose indices are from I and with columns whose indices are from J. $A(:,J)$ abbreviates $A(1:n, J)$. $nnz(A)$ denotes the number of nonzeros in A.

The reduced submatrix $R_{\tilde{K}\tilde{K}} = A_{\tilde{K}\tilde{K}} - A_{\tilde{K}K}A_{KK}^{-1}A_{K\tilde{K}}$ is known as the *Schur complement* of A_{KK} [71]. For $R_{\tilde{K}\tilde{K}}$, let r_i denote the number of entries in row i, and let c_j denote the number of entries in column j. The *Markowitz count* associated with entry (i,j) is defined as $(r_i - 1)(c_j - 1)$. Now, at stage k of the elimination, the pivot a_{ij} is selected from $R_{\tilde{K}\tilde{K}}$ to minimize the Markowitz count among those candidates satisfying the following numerical threshold criterion

$$|a_{ij}| \geq \mu \max_{l \geq k} |a_{lj}| \,, \tag{2.2}$$

where $\mu \in (0,1]$ is a threshold parameter. Although the use of threshold parameter μ permits more element growth than classical partial pivoting, it gives the Markowitz ordering more flexibility in selecting pivots to control fill. Based on this idea, some variations on the criteria for selecting pivots have been proposed to balance numerical stability and preservation of sparsity. The reader may consult Chapter 7 of Duff et al. [36] for a thorough treatment of this subject.

Multifrontal approaches [6, 21, 32] are essentially variations of the submatrix methods. At each stage of the elimination, the update operations for the Schur complement are not applied directly to the target columns of the trailing submatrix. Instead, they are accumulated as a sequence of partial update matrices, which are passed through each level of the elimination tree (or elimination dag in the unsymmetric case) until finally they are incorporated into the destination columns. Multifrontal methods have proven to be more efficient than the pure right-looking methods for several reasons. These include the ability to use dense matrix operations on the frontal matrices, reduced indirect addressing, and localization of memory references. However, multifrontal methods require more working storage to store the frontal matrices than a pure right-looking algorithm. They also require more data movement between the working storage and the target storage for the L and U factors. Furthermore, working storage management is particularly hard in a parallel formulation, because the stack-based organization used for efficiency in the sequential algorithm severely limits the degree of parallelism. More sophisticated parallel schemes were used by Amestoy and Duff [6, 35], which came with nontrivial runtime overhead.

Recent submatrix codes include MA48 [33], Amestoy and Duff's symmetric pattern multifrontal code MUPS [5], and Davis and Duff's unsymmetric multifrontal code UMFPACK [21, 23].

Column methods, by contrast, take j as the outer loop for Equation (2.1) and typically use classical partial pivoting. The pivot is chosen from the current column according to numerical considerations alone; the columns may be preordered before factorization to preserve sparsity. Figure 2.1 sketches a generic left-looking column LU factorization. Notice that the bulk of the numeric computation occurs in column-column updates ("col-col update" on line 5), or, to use BLAS terminology [30], in sparse AXPYs.

Column methods have the advantage that preordering the columns for sparsity is completely separate from the factorization, just as in the symmetric positive definite case. However, symbolic factorization cannot be separated from numeric factorization, because the nonzero structures of the factors depend on the numerical pivoting choices. Thus, column codes must do some symbolic factorization at each stage; typically this amounts to predicting the structure of each column of the factors immediately before computing

1. **for** column $j = 1$ **to** n **do**
2. $\quad f = A(:, j)$;
3. $\quad$ Symbolic factorization: determine which columns of L will update f;
4. $\quad$ **for** each updating column $r < j$ in topological order **do**
5. $\quad\quad$ Col-col update: $f = f - f(r) \cdot L(:, r)$;
6. $\quad$ **end for**;
7. $\quad$ Pivot: interchange $f(j)$ and $f(k)$, where $|f(k)| = \max |f(j\colon n)|$;
8. $\quad$ Separate L and U: $U(1\colon j, j) = f(1\colon j)$; $\quad L(j\colon n, j) = f(j\colon n)$;
9. $\quad$ Scale: $L(j\colon n, j) = L(j\colon n, j)/L(j, j)$;
10. **end for**;

Figure 2.1: Left-looking LU factorization with column-column updates.

it (line 3 in Figure 2.1). Pivot search is confined within one column, which can be done inexpensively. One disadvantage of the column methods is that, unlike Markowitz ordering, they do not reorder the columns dynamically, so the fills may be greater.

An early example of such a code is Sherman's NSPIV [103] (which is actually a row code). Gilbert and Peierls [64] showed how to use depth-first search and topological ordering to obtain the structure of each factor column. This gives a column code that runs in total time proportional to the number of floating-point operations, unlike earlier partial pivoting codes. We shall refer to their code as GP in our performance study in Chapter 4. Eisenstat and Liu [44] designed a pruning technique to reduce the amount of structural information required for the symbolic factorization, which we will describe further in Section 4.3. The result was that the time and space for symbolic factorization were typically reduced to a small fraction of the entire factorization. This improved GP code is referred to as GP-Mod. GP and GP-Mod are used in Matlab 1992 and 1996, respectively.

In view of the success of supernodal techniques for symmetric matrices, it is natural to consider the use of supernodes to enhance the performance of unsymmetric solvers. One difficulty is that, unlike the symmetric case, supernodal structure cannot be determined in advance but rather emerges depending on pivoting choices during the factorization. Eisenstat, Gilbert and Liu [45] discussed how to detect supernodes dynamically. In Chapter 4 we will review their approach and quantify the performance gains.

There have been debates about whether submatrix methods are preferable to column methods, or vice versa. Their memory reference patterns are markedly different. Heath, Ng and Peyton [75] gave a thorough survey of many distinctions between left-looking and right-looking sparse Cholesky factorization algorithms. On uniprocessor machines for in-memory problems, Ng and Peyton [89] and Rothberg [99] conducted extensive experiments with sparse Cholesky factorization, and concluded that the supernodal left-looking algorithm is somewhat better than the multifrontal approach both in runtime and working storage requirement. Gupta and Kumar [72] developed a two-dimensional multifrontal Cholesky algorithm on 1024 nodes of the Cray T3D, and achieved up to 20 Gflops factorization rate for one problem. Rothberg [97] developed a two-dimensional block oriented right-looking Cholesky algorithm, and achieved up to 1.7 Gflops factorization rate on 128 nodes of the Intel Paragon. Rothberg and Schreiber [98] further improved its performance by better block mapping, and achieved up to 3.2 Gflops factorization rate on 196 nodes of the Intel Paragon.

No comprehensive comparisons have yet been made for the unsymmetric LU factorization algorithms. In this case it is even harder to make fair comparisons because, in addition to the considerations above, the trade-off between numerical stability and sparsity plays an important role and depends very much on the input matrices. Although detailed comparisons are valuable to identify the "best" algorithm (or the best combination), they are beyond the scope of this thesis, and will remain as part of our future work. See Chapter 6 for a list of available sparse codes. The goal of this thesis is to make the column algorithm as fast as possible on a variety of high performance architectures and for a variety of problems. From now on, we will focus exclusively on the left-looking column methods.

Chapter 3

Fundamentals of Sparse Column Methods

In this chapter, we present several important data structures and tools used throughout this thesis. We do not intend to review all the basics of sparse matrix computations, such as matrix representation, nonzero manipulation, and graph-theoretic terminology. For that purpose, George and Liu [51] and Duff et al. [36] serve as excellent sources. We hereby confine ourselves only to the most relevant concepts. In Section 3.1, we elaborate on the roles of row interchanges (partial pivoting) and column interchanges to maintain numerical stability and to preserve sparsity. Section 3.2 introduces unsymmetric supernodes, which are essential in order to use higher level BLAS [27, 28]. Section 3.3 gives the definition and properties of the column elimination tree, which is an important tool to assist in the sparse LU factorization, particularly in a parallel setting. Both unsymmetric supernodes and the column elimination tree are generalizations of their symmetric counterparts.

3.1 Row and column permutations

To solve a linear system

$$Ax = b \; , \tag{3.1}$$

we first use Gaussian elimination to transform A into a lower triangular matrix L and an upper triangular matrix U. In this section, we examine the possible row and/or column permutations associated with a sparse Gaussian elimination algorithm. In effect, we perform the following decomposition

$$PAQ^T = LU \; , \tag{3.2}$$

where P and Q are permutation matrices that reorder the rows and columns of A, respectively. In general, P and Q are different. With the factorization (3.2) at hand, the solution of Equation (3.1) is then the same as the solution x of the transformed system

$$PAQ^TQx = Pb \; . \tag{3.3}$$

Equation (3.3) is solved by a forward substitution $Ly = Pb$ for y, a back-substitution $Uz = y$ for z, and finally a permutation $x = Q^T z$ for x. In the next two subsections, we will study the purpose of applying P and Q.

3.1.1 Partial pivoting

It is well known that for the special class of problems where A is symmetric and positive definite, pivots can be chosen down the diagonal in order [71, Chapter 5]. The factorization thus obtained can be written as $A = LL^T$, which is known as Cholesky decomposition.

For general unsymmetric A, however, it is possible to encounter arbitrarily small pivots on the diagonal. If we still pivot on the diagonal, large element growth may occur, yielding an unstable algorithm. This problem can be alleviated by pivoting on the element with largest magnitude in each column, interchanging rows when needed. This process is called partial pivoting. The efficacy of partial pivoting (as opposed to the more costly complete pivoting) to maintain numerical stability is well studied in a large body of literature; for example, see [71, Chapter 4].

When partial pivoting is incorporated, Gaussian elimination can be written as

$$M_{n-1}P_{n-1}M_{n-2}P_{n-2}\dots M_1P_1A = U \;, \tag{3.4}$$

where P_k is an elementary permutation matrix representing the row interchange at step k, M_k corresponds to the k-th Gauss transformation. It is easy to see that we can rewrite Equation (3.4) as

$$PA = LU \;, \tag{3.5}$$

where $P = P_{n-1}\dots P_1$, $L = P(P_1L_1\dots P_{n-1}L_{n-1})$, and $L_k = M_k^{-1}$ is a unit lower triangular matrix with its kth column containing the multipliers at step k.

It is fairly straightforward to implement a dense partial pivoting code. For a sparse matrix, however, off-diagonal pivoting is tremendously difficulty to implement mainly due to the following reason. The nonzero patterns in L and U depend on the row interchanges and cannot be predetermined precisely based solely on the structure of A. This can be best illustrated by the following example given by Gilbert [66]. Let the structure of A be

$$\begin{pmatrix} 1 & & \\ \bullet & 2 & \\ \bullet & & 3 \end{pmatrix} .$$

Depending on the relative magnitudes of the nonzero entries, pivoting could cause the structure of U to be any of the four outcomes:

$$\begin{pmatrix} 1 & & \\ & 2 & \\ & & 3 \end{pmatrix}, \quad \begin{pmatrix} 1 & \bullet & \\ & 2 & \bullet \\ & & 3 \end{pmatrix}, \quad \begin{pmatrix} 1 & \bullet & \\ & 2 & \\ & & 3 \end{pmatrix}, \quad \begin{pmatrix} 1 & & \bullet \\ & 2 & \bullet \\ & & 3 \end{pmatrix} .$$

In consequence, the symbolic nonzero structure prediction cannot be treated as a separate process completely decoupled from numerical factorization, as is normally done in the sparse Cholesky factorization. For an efficient sparse partial pivoting code, we must design both efficient numerical kernels and fast symbolic algorithms. These dual goals carry over to parallel algorithms as well.

3.1.2 Ordering for sparsity

Arranging the equations and variables in an appropriate order so that L and U suffer low fill is an important issue, because low fill implies fewer floating-point operations and low storage requirement.

A canonical example is a symmetric arrow matrix shown below

$$\begin{pmatrix} 1 & \bullet & \bullet & \bullet \\ \bullet & 2 & & \\ \bullet & & 3 & \\ \bullet & & & 4 \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & & & \bullet \\ & 2 & & \bullet \\ & & 3 & \bullet \\ \bullet & \bullet & \bullet & 4 \end{pmatrix},$$

where the original order on the left results in full L and U but the new order on the right preserves all zeros, provided the diagonal entries are numerically acceptable.

In the symmetric positive definite case, an ordering algorithm works only on the graph of A and a sequence of elimination graphs [51] thereafter. It does not need to know the numerical values of A. In the unsymmetric case, however, the elimination graph at each step changes with the numerical pivot selection, as we saw in the previous section. The question arises whether it is still possible to choose a fill-reducing ordering before the factorization begins. The answer is partially positive. The essence of our approach is based on a result proved by George and Ng [56]. Let L_c denote the symbolic Cholesky factor of the normal equations matrix A^TA, in the absence of coincidental numerical cancellation.[1] They showed that, if L is stored as $P_1L_1 \ldots P_{n-1}L_{n-1}$, the structure of L is contained in the structure of L_c, and the structure of U is contained in the structure of L_c^T. This is true regardless of the numerical partial pivoting (row permutation P in Equation (3.5)). It is thus desirable to choose an ordering Q such that the Cholesky factor of QA^TAQ^T suffers little fill. Once a good Q is obtained, it is then applied to the columns of A *before LU factorization.* One would expect that factoring the reordered matrix AQ^T tends to produce less fill in both L and U compared to factoring the original A.

In principle, any ordering heuristic used in the symmetric case can be applied to A^TA to arrive at Q. The column minimum degree algorithm used in Matlab [62] is the first efficient implementation of the minimum degree algorithm on A^TA *without explicitly forming the nonzero structure of* A^TA. In recent work of Davis et al. [24], better minimum degree algorithms for A^TA are under investigation that will improve both fill and runtime.

To summarize, in our column factorization methods, the row permutation P is used to maintain numerical stability and is obtained in the course of elimination. The

[1]Throughout the thesis, when we refer to the structure of a matrix, such as L and U, we always ignore numerical cancellation. This applies to both reordering phase and the symbolic algorithms to be discussed in Section 4.3.

column permutation Q is used to control sparsity and is computed and applied prior to the factorization. With these two permutations, the actual factorization performed is what we see in Equation (3.2) in the beginning of this section.

3.2 Unsymmetric supernodes

The idea of a supernode is to group together columns with the same nonzero structure, so they can be treated as a dense matrix for storage and computation. Supernodes were originally used for sparse Cholesky factorization; the first published results are by Ashcraft, Grimes, Lewis, Peyton, and Simon [12]. In the factorization $A = LL^T$, a supernode is a range $(r\colon s)$ of columns of L with the same nonzero structure below the diagonal; that is, $L(r\colon s, r\colon s)$ is full lower triangular and every row of $L(s+1\colon n, r\colon s)$ is either full or zero. (In Cholesky, supernodes need not consist of contiguous columns, but we will consider only contiguous supernodes.)

Ng and Peyton [87] analyzed the effect of supernodes in Cholesky factorization on modern uniprocessor machines with memory hierarchies and vector or superscalar hardware. We use Figure 3.1 to illustrate all the benefits from supernodes. All the updates from columns of the supernode $(r_1 : s_1)$ can be summed into a packed dense vector before one single sparse update is performed. This reduces indirect addressing, and allows the inner loops to be unrolled. In effect, a sequence of column-column updates is replaced by a supernode-column update (loops 5–9). This so-called "sup-col update" can be implemented using a call to a standard dense BLAS-2 matrix-vector multiplication kernel [27]. This idea can be further extended to supernode-supernode updates ("sup-sup update", loops 2–12), which can be implemented using a BLAS-3 dense matrix-matrix kernel [28]. Sup-sup update can reduce memory traffic by an order of magnitude, because a supernode in the cache can participate in multiple column updates. Ng and Peyton reported that a sparse Cholesky algorithm based on sup-sup updates typically runs 2.5 to 4.5 times as fast as a col-col algorithm. Indeed, supernodes have become a standard tool in sparse Cholesky factorization [12, 87, 95, 105].

To sum up, supernodes as the source of updates (line 4) help because:

1. The inner loop (line 6) over rows i has no indirect addressing. (Sparse BLAS-1 is replaced by dense BLAS-1.)

2. The outer loop (line 5) over columns k in the supernode can be unrolled to save memory references. (BLAS-1 is replaced by BLAS-2.)

Supernodes as the destination of updates (line 1) help because:

3. Elements of the source supernode can be reused in multiple columns j of the destination supernode to reduce cache misses. (BLAS-2 is replaced by BLAS-3.)

Supernodes in sparse Cholesky can be determined during symbolic factorization, before the numeric factorization begins. However, in sparse LU, the nonzero structure cannot be predicted before numeric factorization, so we must identify supernodes on the fly. Furthermore, since the factors L and U are no longer transposes of each other, we must generalize the definition of a supernode.

```
for each destination supernode (r2 : s2) do
    for j = r2 to s2 do
        f = A(j : n, j);
        for each source supernode (r1 : s1) < (r2 : s2) with L(j, r1 : s1) ≠ 0 do
            for k = r1 to s1 do
                for i = j to n with L(i, k) ≠ 0 do
                    f = f − L(i, k) · L(j, k);
                end for;
            end for;
        end for;
        L(j : n, j) = f;
    end for;
    Inner factorization for L(r2 : n, r2 : s2);
end for;
```

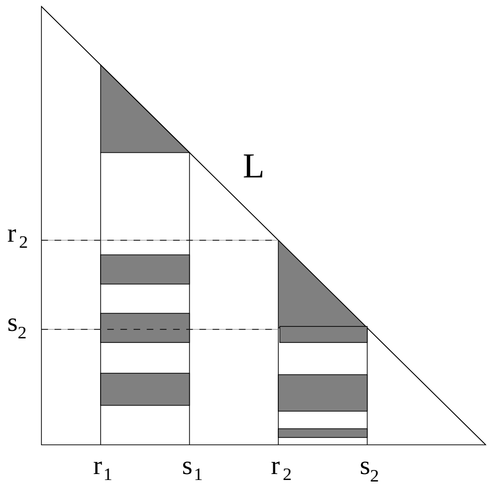


Figure 3.1: Left-looking Cholesky factorization with supernode-supernode updates.

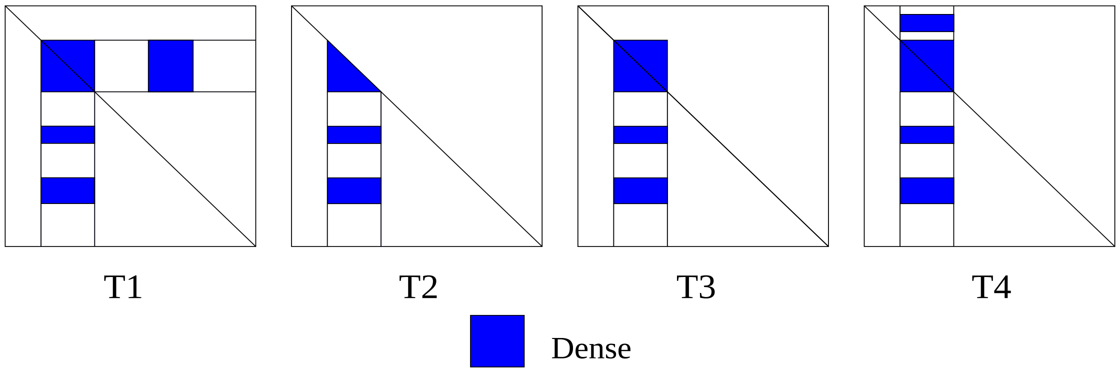


Figure 3.2: Four possible types of unsymmetric supernodes.

3.2.1 Definition of a supernode

Eisenstat, Gilbert and Liu [45] considered several possible ways to generalize the symmetric definition of supernodes to unsymmetric factorization. Here, we present all their characterizations and choose the most appropriate one to use. We define $F = L + U - I$ to be the *filled matrix* containing both L and U, where $PAQ^T = LU$. Figure 3.2 is a schematic of definitions T1 through T4.

T1. Same row and column structures: A supernode is a range $(r\colon s)$ of columns of L and rows of U, such that the diagonal block $F(r\colon s, r\colon s)$ is full, and outside that block all the columns of L in the range have the same structure and all the rows of U in the range have the same structure. T1 supernodes make it possible to do sup-sup updates, realizing the same three benefits enjoyed by Cholesky.

T2. Same column structure in L: A supernode is a range $(r\colon s)$ of columns of L with the triangular diagonal block full and the same structure below the diagonal block. T2 supernodes allow sup-col updates, realizing the first two benefits.

T3. Same column structure in L, full diagonal block in U: A supernode is a range $(r\colon s)$ of columns of L and U, such that the diagonal block $F(r\colon s, r\colon s)$ is full, and below the diagonal block the columns of L have the same structure. T3 supernodes allow sup-col updates, like T2. In addition, if the storage for a supernode is organized as for a two-dimensional array (for BLAS-2 or BLAS-3 calls), T3 supernodes do not waste any space in the diagonal block of U.

T4. Same column structure in L and U: A supernode is a range $(r\colon s)$ of columns of L and U where all columns of $F(:, r : s)$ have identical structure. (Since the diagonal is nonzero, the diagonal block must be full.) T4 supernodes allow sup-col updates, and also simplify storage of L and U.

T5. Supernodes of A^TA: A supernode is a range $(r\colon s)$ of columns of L corresponding to a Cholesky supernode of the symmetric matrix A^TA. T5 supernodes are motivated by

	T1	T2	T3	T4
median	0.236	0.345	0.326	0.006
mean	0.284	0.365	0.342	0.052

Table 3.1: Fraction of nonzeros not in first column of supernode.

George and Ng's observation [59] that (with suitable representations) the structures of L and U in the unsymmetric factorization $PA = LU$ are contained in the structure of the Cholesky factor of $A^T A$. In unsymmetric LU, these supernodes themselves are sparse, so we would waste time and space operating on them. Thus we do not consider them further.

Supernodes are only useful if they actually occur in practice. The occurrence of symmetric supernodes is related to the clique structure of the chordal graph of the Cholesky factor, which arises because of fill during the factorization. Unsymmetric supernodes seem harder to characterize, but they also are related to dense submatrices arising from fill. Eisenstat et al. [45] measured the supernodes according to each definition for 126 unsymmetric matrices from the Harwell-Boeing sparse matrix test collection [31] under various column orderings. Table 3.1 tabulates the results from their measurements. It shows, for each definition, the fraction of nonzeros of L that are not in the first column of a supernode; this measures how much row index storage is saved by using supernodes. Corresponding values for symmetric supernodes for the symmetric Harwell-Boeing structural analysis problems usually range from about 0.5 to 0.9. Larger numbers are better, indicating larger supernodes. We reject T4 supernodes as being too rare to make up for the simplicity of their storage scheme. T1 supernodes allow BLAS-3 updates, but as we will see in Section 4.2 we can get most of their cache advantage with the more common T2 or T3 supernodes by using sup-panel updates. Thus we conclude that either T2 or T3 is the best choice. Our code uses T2, which gives slightly larger superndoes than T3 at a small extra cost in storage, because we store the triangular matrix in full array, with the upper diagonal entries padded with the elements from U.

Figure 3.3 shows a sample matrix, and the nonzero structure of its factors with no pivoting. Using definition T2, this matrix has four supernodes: $\{1, 2\}$, $\{3\}$, $\{4, 5, 6\}$, and $\{7, 8, 9, 10\}$. For example, in columns 4, 5, and 6 the diagonal blocks of L and U are full, and the columns of L all have nonzeros in rows 8 and 9. By definition T3, the matrix has five supernodes: $\{1, 2\}$, $\{3\}$, $\{4, 5, 6\}$, $\{7\}$, and $\{8, 9, 10\}$. Column 7 fails to join $\{8, 9, 10\}$ as a T3 supernode because u_{78} is zero.

3.2.2 Storage of supernodes

A standard way to organize storage for a sparse matrix is as a one-dimensional array of nonzero values in column major order, plus integer arrays giving row numbers and column starting positions. This is called compressed column storage, and is also the storage scheme used in the Harwell-Boeing collection. Figure 3.4 illustrates the storage for the first three columns of the sample matrix A in Figure 3.3.

$$
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & & \bullet & & & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & & & \bullet & & & \\
\bullet & & & & & 6 & & & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & & \bullet & & & & & & & \\
& & & \bullet & & \bullet & & & 9 & \\
& & \bullet & & & & \bullet & & \bullet & 10
\end{pmatrix}
\qquad
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & \bullet & \bullet & & \bullet & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & 5 & \bullet & \bullet & & \bullet & \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \bullet & \bullet & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 8 & \bullet & \bullet \\
& & & \bullet & \bullet & \bullet & \bullet & \bullet & 9 & \bullet \\
& & \bullet & & & & \bullet & \bullet & \bullet & 10
\end{pmatrix}
$$

Original matrix A — Factors $F = L + U - I$

Figure 3.3: A sample matrix and its LU factors. Diagonal elements a_{55} and a_{88} are zero.

Values	a_{11} a_{21} a_{61} a_{81}	a_{12} a_{22}	a_{13} a_{33} a_{83} $a_{10,3}$	$\bullet\ \bullet\ \bullet$							
Row Subscripts	1 2 6 8	1 2	1 3 8 10	$\bullet\ \bullet\ \bullet$							
Column Pointers	1	5	7	11	15	16	20	24	26	31	

Figure 3.4: Compressed column storage for a sample sparse matrix.

$$
\begin{array}{r}
1\\ 2\\ 3\\ 4\\ 5\\ 6\\ 7\\ 8\\ 9\\ 10
\end{array}
\begin{pmatrix}
s_1 & s_1 & u_3 & & & u_6 & & & & \\
s_1 & s_1 & u_3 & u_4 & & u_6 & & u_8 & & \\
 & & s_2 & & & & u_7 & u_8 & & \\
 & & & s_3 & s_3 & s_3 & & & u_9 & \\
 & & & s_3 & s_3 & s_3 & u_7 & & u_9 & \\
s_1 & s_1 & s_2 & s_3 & s_3 & s_3 & u_7 & u_8 & u_9 & \\
 & & & & & & s_4 & & s_4 & s_4 \\
s_1 & s_1 & s_2 & s_3 & s_3 & s_3 & s_4 & s_4 & s_4 & s_4 \\
 & & & s_3 & s_3 & s_3 & s_4 & s_4 & s_4 & s_4 \\
 & & s_2 & & & & s_4 & s_4 & s_4 & s_4
\end{pmatrix}
$$

Figure 3.5: Supernodal structure (by definition T2) of the factors of the sample matrix.

We use this layout for both L and U, but with a slight modification: we store the entire square diagonal block of each supernode as part of L, including both the strict lower triangle of values from L and the upper triangle of values from U. We store this square block as if it were completely full (it is full in T3 supernodes, but its upper triangle may contain zeros in T2 supernodes). This allows us to address each supernode as a two-dimensional array in calls to BLAS routines. In other words, if columns $(r\colon s)$ form a supernode, then all the nonzeros in $F(r\colon n, r\colon s)$ are stored as a single dense two-dimensional array. This also lets us save some storage for row indices: only the indices of nonzero rows outside the diagonal block need be stored, and the structures of all columns within a supernode can be described by one set of row indices. This is similar to the effect of compressed subscripts in the symmetric case [102].

We represent the part of U outside the supernodal blocks with a compressed column storage: the values are stored by columns, with a companion integer array the same size to store row indices; another array of n integers indicates the start of each column.

Figure 3.5 shows the structure of the factors in the example from Figure 3.3, with s_k denoting a nonzero in the k-th supernode and u_k denoting a nonzero in the k-th column of U outside the supernodal block. Figure 3.6 shows the storage layout. (We omit the indexing vectors that point to the beginning of each supernode and the beginning of each column of U.)

3.3 Column elimination tree

Since our definition requires the columns of a supernode to be contiguous, we should get larger supernodes if we bring together columns of L with the same nonzero structure. But the column ordering is fixed, for sparsity, before numeric factorization; what can we do?

In Cholesky factorization, the so-called fundamental supernodes can be made contiguous by permuting the matrix (symmetrically) according to a *postorder* on its elimination tree [11]. This is because each fundamental supernode corresponds to a chain of nodes in

$$
\begin{array}{c}
 \\
 \\
 \\
6 \\
8
\end{array}
\begin{array}{c}
\begin{array}{cc} 1 & 2 \end{array} \\
\begin{pmatrix}
s_1 & s_1 \\
s_1 & s_1 \\
s_1 & s_1 \\
s_1 & s_1
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c}
 \\
 \\
6 \\
8 \\
10
\end{array}
\begin{array}{c}
3 \\
\begin{pmatrix}
s_2 \\
s_2 \\
s_2 \\
s_2
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c}
 \\
 \\
 \\
 \\
8 \\
9
\end{array}
\begin{array}{c}
\begin{array}{ccc} 4 & 5 & 6 \end{array} \\
\begin{pmatrix}
s_3 & s_3 & s_3 \\
s_3 & s_3 & s_3 \\
s_3 & s_3 & s_3 \\
s_3 & s_3 & s_3 \\
s_3 & s_3 & s_3
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c}
\begin{array}{cccc} 7 & 8 & 9 & 10 \end{array} \\
\begin{pmatrix}
s_4 & 0 & s_4 & s_4 \\
s_4 & s_4 & s_4 & s_4 \\
s_4 & s_4 & s_4 & s_4 \\
s_4 & s_4 & s_4 & s_4
\end{pmatrix}
\end{array}
$$

Supernodal blocks (stored in column-major order)

Row Subscripts	1	2	2	1	2	3	5	6	2	3	6	4	5	6
	u_3	u_3	u_4	u_6	u_6	u_7	u_7	u_7	u_8	u_8	u_8	u_9	u_9	u_9

Off-supernodal nonzeros in columns of U

Figure 3.6: Storage layout for factors of the sample matrix, using T2 supernodes.

the tree, and in a postordering of the tree, the nodes within every subtree of the elimination tree will be numbered consecutively. This postorder is an example of what Liu calls an equivalent reordering [80], which does not change the sparsity of the Cholesky factor L, nor the amount of arithmetic to compute L. (Liu proved that any topological ordering, which numbers the children nodes before their parent node, is an equivalent reordering of the given matrix.) The postordered elimination tree can also be used to locate the supernodes before the numeric factorization.

We proceed similarly for the unsymmetric case. Here the appropriate analog of the symmetric elimination tree is the *column elimination tree*, or column etree for short. The vertices of this tree are the integers 1 through n, representing the columns of A. The column etree of A is the (symmetric) elimination tree of A^TA provided there is no cancellation in computing A^TA. More specifically, if L_c denotes the Cholesky factor of A^TA, then the parent of vertex j is the row index i of the first nonzero entry below the diagonal of column $L_c(:,j)$. The column etree can be computed from A in time almost linear in the number of nonzeros of A by a variation of an algorithm of Liu [80].

The following theorem says that the column etree represents potential dependencies among columns in LU factorization, and that for strong Hall matrices (that is, they cannot be permuted to nontrivial block triangular forms), no stronger information is obtainable from the nonzero structure of A. Note that column i updates column j in LU factorization if and only if $u_{ij} \neq 0$.

Theorem 1 (Column Elimination Tree) [63] *Let A be a square, nonsingular, possibly unsymmetric matrix, and let $PA = LU$ be any factorization of A with pivoting by row interchanges. Let T be the column elimination tree of A.*

1. *If vertex i is an ancestor of vertex j in T, then $i \geq j$.*

2. *If $l_{ij} \neq 0$, then vertex i is an ancestor of vertex j in T.*

3. *If $u_{ij} \neq 0$, then vertex j is an ancestor of vertex i in T.*

4. *Suppose in addition that A is strong Hall. If vertex j is the parent of vertex i in T, then there is some choice of values for the nonzeros of A that makes $u_{ij} \neq 0$ when the factorization $PA = LU$ is computed with partial pivoting.*

Just as a postorder on the symmetric elimination tree brings together symmetric supernodes, we expect a postorder on the column etree to bring together unsymmetric supernodes. Thus, before we factor the matrix, we compute its column etree and permute the matrix columns according to a postorder on the tree. The following theorem, due to Gilbert [67], shows that this does not change the factorization in any essential way.

Theorem 2 *Let A be a matrix with column etree T. Let π be a permutation such that whenever $\pi(i)$ is an ancestor of $\pi(j)$ in T, we have $i \geq j$. Let P be the permutation matrix such that $\pi = P \cdot (1{:}\, n)^T$. Let $\bar{A} = PAP^T$.*

1. *$\bar{A} = A(\pi, \pi)$.*

2. *The column etree $\bar{T}$ of $\bar{A}$ is isomorphic to T; in particular, relabeling each node i of $\bar{T}$ as $\pi(i)$ yields T.*

3. *Suppose in addition that $\bar{A}$ has an LU factorization without pivoting, $\bar{A} = \bar{L}\bar{U}$. Then $P^T\bar{L}P$ and $P^T\bar{U}P$ are respectively unit lower triangular and upper triangular, so $A = (P^T\bar{L}P)(P^T\bar{U}P)$ is also an LU factorization.*

Remark: Liu [80] attributes to F. Peters a result similar to part (3) for the symmetric positive definite case, concerning the Cholesky factor and the (usual, symmetric) elimination tree. For completeness, we give the proof by Gilbert as follows.

Proof: Part (1) is immediate from the definition of P. Part (2) follows from Corollary 6.2 in Liu [80], with the symmetric structure of the column intersection graph of our matrix A taking the place of Liu's symmetric matrix A. (Liu exhibits the isomorphism explicitly in the proof of his Theorem 6.1.)

Now we prove part (3). We have $a_{\pi(i)\pi(j)} = \bar{a}_{ij}$ for all i and j. Write $L = P^T\bar{L}P$ and $U = P^T\bar{U}P$, so that $l_{\pi(i)\pi(j)} = \bar{l}_{ij}$ and $u_{\pi(i)\pi(j)} = \bar{u}_{ij}$. Then $A = LU$; we need only show that L and U are triangular.

Consider a nonzero $u_{\pi(i)\pi(j)}$ of U. In the triangular factorization $\bar{A} = \bar{L}\bar{U}$, element $\bar{u}_{ij}$ is equal to $u_{\pi(i)\pi(j)}$ and is therefore nonzero. By part (3) of Theorem 1, then, j is an ancestor of i in $\bar{T}$. By the isomorphism between $\bar{T}$ and T, this implies that $\pi(j)$ is an ancestor of $\pi(i)$ in T. Then it follows from part (1) of Theorem 1 that $\pi(j) \geq \pi(i)$. Thus every nonzero of U is on or above the diagonal, so U is upper triangular. A similar argument shows that every nonzero of L is on or below the diagonal, so L is lower triangular. The diagonal elements of L are a permutation of those of $\bar{L}$, so they are all equal to 1. □

Since the triangular factors of A are just permutations of the triangular factors of PAP^T, they have the same number of nonzeros. ($nnz(L) = nnz(\bar{L})$ and $nnz(U) = nnz(\bar{U})$.) Indeed, they require the same arithmetic to compute; the only possible difference is the order of updates. If addition for updates is commutative and associative, this implies that with partial pivoting (i,j) is a legal pivot in $\bar{A}$ iff $(\pi(i),\pi(j))$ is a legal pivot in A. In floating-point arithmetic, the different order of updates could conceivably change the pivot sequence. Thus we have the following corollary.

Corollary 1 *Let π be a postorder on the column elimination tree of A, let P_1 be any permutation matrix, and let P_2 be the permutation matrix with $\pi = P_2 \cdot (1\colon n)^T$. If $P_1AP_2^T = LU$ is an LU factorization, then so is $(P_2^T P_1)A = (P_2^T LP_2)(P_2^T UP_2)$. In exact arithmetic, the former is an LU factorization with partial pivoting of AP_2^T if and only if the latter is an LU factorization with partial pivoting of A.*

This corollary says that an LU code can permute the columns of its input matrix by postorder on the column etree, and then fold the column permutation into the row permutation on output. Thus our code has the option of returning either four matrices P_1, P_2, L, and U (with $P_1AP_2^T = LU$), or just the three matrices $P_2^T P_1$, $P_2^T LP_2$, and $P_2^T UP_2$, which are a row permutation and two triangular matrices. The advantage of returning all four matrices is that the columns of each supernode are contiguous in L, which permits the use of a BLAS-2 supernodal triangular solve for the forward-substitution phase of a linear system solver. The supernodes are not contiguous in $P_2^T LP_2$.

We note that in the symmetric positive definite case, the elimination tree has long been employed as a major task scheduling model to design parallel sparse Cholesky factorization. At a large-grained level, "parallel pivots" can be chosen from the disjoint subtrees and eliminated simultaneously by different processors. At a fine-grained level, more than one processor cooperates to eliminate one pivot; this is necessary at later stages of the elimination. In the unsymmetric case, the column etree can play a similar role. However, there exist some subtle differences between the two computational models, which we will illustrate in Chapter 5 when we study parallel LU factorization.

3.4 Artificial supernodes

We have explored various ways of allowing sparsity in a supernode. We experimented with both T2 and T3 supernodes, and found that T2 supernodes (those with only nested column structures in L) are slightly larger than T3 supernodes and give slightly better performance. Our code uses T2 at a small extra cost in storage.

We observe that, for most matrices, the average size of a supernode is only about 2 to 3 columns (though a few supernodes are much larger). A large percentage of supernodes consist of only a single column, many of which are leaves of the column etree. Therefore we have devised a scheme to merge groups of columns at the fringe of the etree into *artificial supernodes* regardless of their row structures. A parameter r controls the granularity of the merge. Our merge rule is: node i is merged with its parent node j when the subtree rooted at j has at most r nodes. This may introduce some logical zeros. In practice, the best values of r are generally between 4 and 8, and yield improvements in running time of 5% to 15%. For such values of r, the extra storage needed to store the logical zeros is very small for all our test matrices.

Artificial supernodes are a special case of *relaxed supernodes*, which were used in the context of multifrontal methods for symmetric systems [11, 40]. Ashcraft and Grimes allow a small number of zeros in the structure of any supernode, thus relaxing the condition that the columns must have strictly nested structures. It would be possible to use this idea in the unsymmetric code as well, though we have not experimented with it. Relaxed supernodes could be constructed either on the fly (by relaxing the nonzero count condition described in Section 4.3.3), or by preprocessing the column etree to identify small subtrees that we would merge into supernodes.

Chapter 4

Supernode-Panel Sparse Factorization with Partial Pivoting

In this chapter, we show how to modify the column-column algorithm to use supernode-column updates and supernode-panel updates. Sections 4.1 and 4.2 describe the organization of the numerical kernels in the supernodal algorithms. Section 4.3 describes the symbolic factorization that determines which supernodes update which columns and produce fills in the factored matrices, identifies the boundaries between supernodes, and also performs the symmetric structure reduction. Following the description of the algorithms, we present performance results obtained on several high performance machines, including IBM RS6000-590, MIPS R8000, and DEC Alpha 21164. Our test matrices were collected from diverse application fields with varied characteristics. We also analyze at great length the performance of the new algorithm.

4.1 Supernode-column updates

Eisenstat et al. [45] first introduced the supernode-column algorithm, as formulated in Figure 4.1. We refer to this code as SupCol. The only difference from the column-column algorithm (Figure 2.1) is that all the updates to a column from a single supernode are done together. Consider a supernode $(r\colon s)$ that updates column j. The coefficients of the updates are the values from a segment of column j of U, namely $U(r\colon s, j)$. The nonzero structure of such a segment is particularly simple: all the nonzeros are contiguous, and follow all the zeros (as proved in Corollary 2, to appear in Section 4.3.1). Thus, if k $(r \leq k \leq s)$ is the index of the first nonzero row in $U(r\colon s, j)$, the updates to column j from supernode $(r\colon s)$ come from columns k through s. Since the supernode is stored as a dense matrix, these updates can be performed by a dense lower triangular solve (with the matrix $L(k\colon s, k\colon s)$) and a dense matrix-vector multiplication (with the matrix $L(s+1\colon n, k\colon s)$). As described in Section 4.3, the symbolic phase determines the value of k, that is, the position of the first nonzero in the segment $U(r\colon s, j)$.

The advantages of using supernode-column updates are similar to those in the

for column $j = 1$ **to** n **do**
    $f = A(:, j)$;
    Symbolic factorization: determine which supernodes of L will update f;
    Determine whether j belongs to the same supernode as $j-1$;
    for each updating supernode $(r\colon s) < j$ in topological order **do**
        Apply supernode-column update to column j:
            $f(r\colon s) = L(r\colon s, r\colon s)^{-1} \cdot f(r\colon s)$; /* Now $f(r\colon s) = U(r\colon s, j)$ */
            $f(s+1\colon n) = f(s+1\colon n) - L(s+1\colon n, r\colon s) \cdot f(r\colon s)$;
    end for;
    Pivot: interchange $f(j)$ and $f(m)$, where $|f(m)| = \max|f(j\colon n)|$;
    Separate L and U: $U(1\colon j, j) = f(1\colon j)$; $L(j\colon n, j) = f(j\colon n)$;
    Scale: $L(j\colon n, j) = L(j\colon n, j)/L(j, j)$;
    Prune symbolic structure based on column j;
end for;

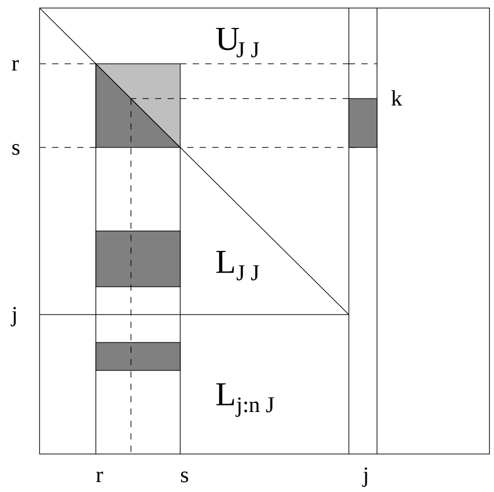


Figure 4.1: LU factorization with supernode-column updates. $J = 1\colon j-1$.

symmetric case [87]. Efficient BLAS-2 matrix-vector kernels can be used for the triangular solve and matrix-vector multiply. Furthermore, all the updates from the supernodal columns can be collected in a temporary packed vector before doing a single scatter into a full-length working array of size n, called SPA (for *sparse accumulator* [62]). This reduces the amount of indirect addressing. The use of the SPA allows random access to the entries of the active column. The scatter operations are buried in lines 7 and 8 in Figure 4.1.

4.2 Supernode-panel updates

We can improve the supernode-column algorithm further on machines with a memory hierarchy by changing the data access pattern. The data we are accessing in the inner loop (lines 5–9 in Figure 4.1) includes the destination column j and all the updating supernodes $(r\colon s)$ to the left of column j. Column j is accessed many times, while each supernode $(r\colon s)$ is used only once. In practice, the number of nonzero elements in column j is much less than that in the updating supernodes. Therefore, the access pattern given by this loop provides little opportunity to reuse cached supernodes. In particular, the same supernode $(r\colon s)$ may be needed to update both columns j and $j+1$. But when we factor the $(j+1)$-st column (in the next iteration of the outer loop), we will have to fetch supernode $(r\colon s)$ again from memory, instead of from cache (unless the supernodes are small compared to the cache).

To exploit memory locality, we factor several columns (say w of them) at a time in the outer loop, so that one updating supernode $(r\colon s)$ can be used to update as many of the w columns as possible. We refer to these w consecutive columns as a *panel*, to differentiate them from a supernode; the row structures of these columns may not be correlated in any fashion, and the boundaries between panels may be different from those between supernodes. The new method requires rewriting the doubly nested loop as the triple loop shown in Figure 4.2. This is analogous to loop tiling techniques used in optimizing compilers to improve cache behavior for two-dimensional arrays with regular stride. It is also somewhat analogous to the supernode-supernode updates that Ng and Peyton [87], and Rothberg and Gupta [95] have used in symmetric Cholesky factorization.

The structure of each supernode-column update is the same as in the supernode-column algorithm. For each supernode $(r\colon s)$ to the left of column j, if $u_{kj} \neq 0$ for some $r \leq k \leq s$, then $u_{ij} \neq 0$ for all $k \leq i \leq s$. Therefore, the nonzero structure of the panel of U consists of dense column segments that are row-wise separated by supernodal boundaries, as in Figure 4.2. Thus, it is sufficient for the symbolic factorization algorithm to record only the first nonzero position of each column segment. As detailed in Section 4.3.4, symbolic factorization is applied to all the columns in a panel at once, before the numeric-factorization loop over all the updating supernodes.

In a dense factorization, the entire supernode-panel update in lines 3–7 of Figure 4.2 would be implemented as two BLAS-3 calls: a dense triangular solve with w right-hand sides, followed by a dense matrix-matrix multiply. In the sparse case, this is not possible, because the different supernode-column updates begin at different positions k within the supernode, and the submatrix $U(r\colon s, j\colon j+w-1)$ is not dense. Thus the sparse supernode-panel algorithm still calls the level-2 BLAS. However, we get similar cache ben-

```
for column j = 1 to n step w do
    Symbolic factor: determine which supernodes will update
                any of F(:, j: j + w − 1);
    for each updating supernode (r: s) < j in topological order do
        for column jj = j to j + w − 1 do
            Apply supernode-column update to column jj;
        end for;
    end for;
    Inner factorization:
        Apply the sup-col algorithm on columns and supernode within the panel;
end for;
```

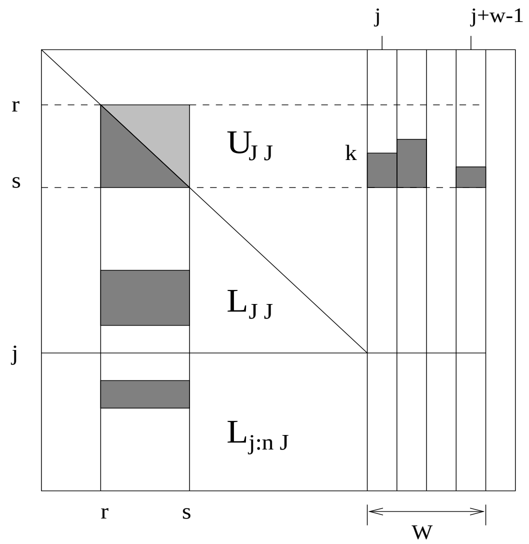


Figure 4.2: The supernode-panel algorithm, with column-wise blocking. $J = 1\colon j-1$.

efits to the level-3 BLAS, at the cost of doing the loop reorganization ourselves. Thus we call the kernel of this algorithm a "BLAS-$2\frac{1}{2}$" method.

In the doubly nested loop 3–7, the ideal circumstance is that all w columns in the panel require updates from supernode $(r{:}\, s)$. Then this supernode will be used w times before it is forced out of the cache. There is a trade-off between insufficient reuse (if w is too small) and cache thrashing (if w is too large). For this scheme to work efficiently, we need to reduce two kinds of cache misses. One is due to cache capacity; that is, we must keep w small enough that all the data accessed in this doubly nested loop fit in cache. Another is due to cache conflict between the source supernode and the destination panel, even though the cache is in principle large enough to hold both. This has to do with the memory layout of the two data structures and the cache block-placement policy. One possibility to avoid this cache conflict is to copy the source supernode and the destination panel into buffers to perform the update.

4.2.1 Outer and inner factorization

At the end of the supernode-panel update (line 7), columns j through $j+w-1$ of L and U have received all their updates from columns to the left of j. We call this the *outer factorization*. What remains is to apply updates that come from columns within the panel. This amounts to forming the LU factorization of the panel itself (in columns $(j{:}\, j+w-1)$, and rows $(j{:}\, n)$). This *inner factorization* is performed by the supernode-column algorithm, almost exactly as shown in Figure 4.1. The inner factorization includes a columnwise symbolic factorization just as in the supernode-column algorithm. The inner factorization also includes the supernode identification, partial pivoting, and symmetric structure reduction for the entire algorithm.

4.2.2 Reducing cache misses by row-wise blocking

Our first experiments with the supernode-panel algorithm showed speedups of around 20–30% for some medium-sized problems. However, the improvement for large matrices was often only a few percent. We now study the reasons and remedies for this.

To implement loops 3–7 in Figure 4.2, we first expand the nonzeros of the panel columns of A into an n by w full working array, that is, the SPA. An n by 1 full array stores the results of the BLAS operations, and the updates are scattered into the SPA. At the end of panel factorization, the data in the SPA are copied into storage for L and U. Although increasing the panel size w gives more opportunity for data reuse, it also increases the size of the active data set that must fit into cache. The supernode-panel update loop accesses the following data (which we call the *working set*):

- the nonzeros in the updating supernode $L(r{:}\, n, r{:}\, s)$.
- the n by w SPA structure, and an n by 1 full array.

By instrumenting the code, we found that the working sets of large matrices are much larger than the cache size. Hence, cache thrashing limits performance.

We experimented with a scheme suggested by Rothberg [96], in which the SPA has only as many rows as the number of nonzero rows in the panel (as predicted by symbolic

1. **for** $j = 1$ **to** n **step** w **do**
2. $\quad\cdots$
3. $\quad$ **for** each updating supernode $(r\colon s) < j$ in topological order **do**
4. $\quad\quad$ Apply triangular solves to $A(r\colon s, j\colon j+w-1)$ using $L(r\colon s, r\colon s)$;
5. $\quad\quad$ **for** each row block B in $L(s+1\colon n, r\colon s)$ **do**
6. $\quad\quad\quad$ **for** $jj = j$ **to** $j + w - 1$ **do**
7. $\quad\quad\quad\quad$ Multiply $B \cdot U(r\colon s, jj)$, and scatter into SPA$(:, jj)$;
8. $\quad\quad\quad$ **end for**;
9. $\quad\quad$ **end for**;
10. $\quad$ **end for**;
11. $\quad\cdots$

12 **end for**;

Figure 4.3: The supernode-panel algorithm, with 2-D blocking.

factorization), and an extra indirection array of size n is used to address the SPA. Unfortunately, the cost incurred by double indirection is significant, and this scheme was not as effective as the two-dimensional blocking method we now describe.

We implemented a row-wise blocking scheme on top of the column-wise blocking in the supernode-panel update, see Figure 4.3. The 2-D blocking adds another level of looping between the two loops in lines 3 and 4 of Figure 4.2. This partitions the supernodes (and the SPA structure) into block rows. Then each block row of the updating supernode is used for up to w partial matrix-vector multiplies, which are pushed all the way through into the SPA before the next block row of the supernode is accessed. The active data set accessed in the inner loops is thus much smaller than in the 1-D scheme. The key performance gains come from loops 5–9, where each row block is reused as much as possible before the next row block is brought into the cache. The innermost loop is still a dense-matrix vector multiply, performed by a BLAS-2 kernel.

4.2.3 Combining 1-D and 2-D blocking

The 2-D blocking works well when the rectangular supernodal matrix $L(r\colon n, r\colon s)$ is large in both dimensions. If all of $L(r\colon n, r\colon s)$ can fit into cache, then the row-wise blocking gives no benefit, but still incurs overhead for setting up loop variables, skipping the empty loop body, and so on. This overhead can be nearly 10% for some of the sparser problems in our test suite. Thus we have devised a hybrid update algorithm that uses either the 1-D or 2-D partitioning scheme, depending on the size of each updating supernode. The decision is made at run-time, as shown in Figure 4.4. Note that Figure 4.4 is identical to Figure 4.3, if line 5 in Figure 4.3 is implemented appropriately. The overhead is limited to the test at line 4 of Figure 4.4. It turns out that this hybrid scheme works better than either 1-D or 2-D codes for many problems. Therefore, this is the algorithm we use in the ultimate code, which we call it SuperLU. In Section 4.6.3 we will discuss in more detail what we mean by "large" in the test on line 4.

```
for j = 1 to n step w do
    ...
    for each updating supernode (r:s) < j in topological order do
        if supernode (r:s) is large then /* use 2-D blocking */
            Apply triangular solves to A(r:s, j:j+w-1) using L(r:s, r:s);
            for each row block B in L(s+1:n, r:s) do
                for jj = j to j+w-1 do
                    Multiply B · U(r:s, jj), and scatter into SPA(:, jj);
                end for;
            end for;
        else /* use 1-D blocking */
            for jj = j to j+w-1 do
                Apply triangular solve to A(r:s, jj) using L(r:s, r:s);
                Multiply L(s+1:n, r:s) · U(r:s, jj), and scatter into SPA(:, jj);
            end for;
        end if;
    end for;
    ...
end for;
```

Figure 4.4: The supernode-panel algorithm, with both 1-D and 2-D blocking.

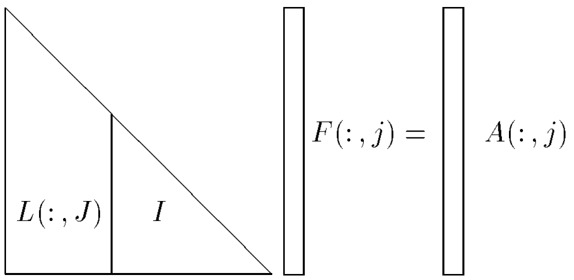
4.3 Symbolic factorization

Symbolic factorization is the process that determines the nonzero structure of the triangular factors L and U from the nonzero structure of the matrix A. This in turn determines which columns of L update each column j of the factors (namely, those columns r for which $u_{rj} \neq 0$), and also which columns of L can be combined into supernodes.

Without numeric pivoting, e.g., in Cholesky factorization or for diagonally dominant matrices, the complete symbolic factorization can be performed before any numeric factorization. Partial pivoting, however, requires that the numeric and symbolic factorizations be interleaved. The supernode-column algorithm performs symbolic factorization for each column just before it is computed, as described in Section 4.3.1. The supernode-panel algorithm performs symbolic factorization for an entire panel at once, as described in Section 4.3.4.

4.3.1 Column depth-first search

From the numeric factorization algorithm, it is clear that the structure of column $F(:,j)$ ($F = L + U - I$) depends on the structure of column $A(:,j)$ of the original matrix and on the structure of $L(:,J)$, where $J = 1{:}j-1$. Indeed, $F(:,j)$ has the same structure as the solution vector for the following triangular system [64]:



A straightforward way to compute the structure of $F(:,j)$ from the structures of $L(:,J)$ and $A(:,j)$ is to simulate the numerical computation. A less expensive way is to use the following characterization in terms of paths in the directed graph of $L(:,J)$.

For any matrix M, the notation $i \xrightarrow{M} j$ means that there is an edge from i to j in the directed graph of M, that is, $m_{ij} \neq 0$. Edges in the directed graph of M are directed from rows to columns. The notation $i \stackrel{M}{\Longrightarrow} j$ means that there is a directed path from i to j in the directed graph of M. Such a path may have length zero; that is, $i \stackrel{M}{\Longrightarrow} i$ holds if $m_{ii} \neq 0$.

Theorem 3 [60] *f_{ij} is nonzero (equivalently, $i \xrightarrow{F} j$) if and only if $i \stackrel{L(:,J)}{\Longrightarrow} k \xrightarrow{A} j$ for some $k \leq i$.*

This result implies that the symbolic factorization of column j can be obtained as follows. Consider the nonzeros in $A(:,j)$ as a subset of the vertices of the directed graph $G = G(L(:,J)^T)$, which is the reverse of the directed graph of $L(:,J)$. The nonzero positions of $F(:,j)$ are then given by the vertices reachable by paths from this set in G.

We use the graph of L^T here because of the convention that edges are directed from rows to columns. Since L is actually stored by columns, our data structure gives precisely the adjacency information for G. Therefore, we can determine the structure of column j of L and U by traversing G from the set of starting nodes given by the structure of $A(:,j)$.

The traversal of G determines the structure of $U(:,j)$, which in turn determines the columns of L that will participate in updates to column j in the numerical factorization. These updates must be applied in an order consistent with a topological ordering of G. We use depth-first search to perform the traversal, which makes it possible to generate a topological order (specifically, reverse postorder) on the nonzeros of $U(:,j)$ as they are located [64].

Another consequence of the path theorem is the following corollary. It says that if we divide each column of U into *segments*, one per supernode, then within each segment the column of U just consists of a consecutive sequence of nonzeros. Thus we need only keep track of the position of the first nonzero in each segment.

Corollary 2 *Let $(r\!:s)$ be a supernode (of either type T2 or T3) in the factorization $PA = LU$. Suppose u_{kj} is nonzero for some j with $r \le k \le s$. Then $u_{ij} \ne 0$ for all i with $k \le i \le s$.*

Proof: Let $k \le i \le s$. Since $u_{kj} \ne 0$, we have $k \stackrel{L(:,J)}{\Longrightarrow} m \stackrel{A}{\longrightarrow} j$ for some $m \le k$ by Theorem 3. Now l_{ik} is in the diagonal block of the supernode, and hence is nonzero. Thus $i \stackrel{L(:,J)}{\longrightarrow} k$, so $i \stackrel{L(:,J)}{\Longrightarrow} m \stackrel{A}{\longrightarrow} j$, whence u_{ij} is nonzero by Theorem 3. □

4.3.2 Pruning the symbolic structure

To speed up the depth-first search traversals, Eisenstat and Liu [43, 44] and Gilbert and Liu [61] have explored the idea of using a reduced graph in place of $G = G(L(:,J)^T)$. Any graph H can be substituted for G, provided that $i \stackrel{H}{\Longrightarrow} j$ if and only if $i \stackrel{G}{\Longrightarrow} j$. The traversals are more efficient if H has fewer edges; but any gain in efficiency must be traded off against the cost of computing H.

An extreme choice of H is the *elimination dag* [61], which is the transitive reduction of G, or the minimal subgraph of G that preserves paths. However, the elimination dag is expensive to compute. The *symmetric reduction* [43] is a subgraph that has (in general) fewer edges than G but more edges than the elimination dag, and that is much less expensive than the latter to compute. The symmetric reduction takes advantage of symmetry in the structure of the filled matrix F; if F is completely symmetric, it is just the symmetric elimination tree. The symmetric reduction of $G(L(:,J)^T)$ is obtained by removing all nonzero l_{rs} for which $l_{ts}u_{st} \ne 0$ for some $t < \min(r,j)$. Eisenstat and Liu [44] give an efficient method to compute the symmetric reduction during symbolic factorization, and demonstrate experimentally that it significantly reduces the total factorization time when used in an algorithm that does column-column updates.

Our supernodal code uses symmetric reduction to speed up its symbolic factorization. Take the sample matrix used in Figure 3.3, Figure 4.5 illustrates symmetric reduction in the presence of supernodes. We use S to represent the supernodal structure of $L(:,J)^T$,

$$
\begin{array}{cccc}
\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & \\
\bullet & 2 & \bullet & \bullet & & \bullet & \\
& & 3 & & & & \bullet \\
& & & 4 & \bullet & \bullet & \\
& & & \bullet & 5 & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \bullet \\
& & & & & & 7 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
& & & \bullet & \bullet & \bullet & \bullet \\
& & \bullet & & & & \bullet
\end{pmatrix}
&
\begin{pmatrix}
1 & & & & & & \\
\bullet & 2 & & & & & \\
& & 3 & & & & \\
& & & 4 & & & \\
& & & \bullet & 5 & & \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \\
& & & & & & 7 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
& & & \bullet & \bullet & \bullet & \bullet \\
& & \bullet & & & & \bullet
\end{pmatrix}
&
\begin{pmatrix}
1 & & & \\
2 & & & \\
& 3 & & \\
& & 4 & \\
& & 5 & \\
\bullet & \bullet & 6 & \\
& & & 7 \\
\bullet & \bullet & \bullet & \bullet \\
& & \bullet & \bullet \\
& \bullet & & \bullet
\end{pmatrix}
&
\begin{pmatrix}
1 & & & \\
2 & & & \\
& 3 & & \\
& & 4 & \\
& & 5 & \\
\bullet & \bullet & 6 & \\
& & & 7 \\
\cdot & \bullet & \bullet & \bullet \\
& & \bullet & \bullet \\
& \bullet & & \bullet
\end{pmatrix}
\\
\text{Filled matrix} & \text{Lower triangle} & \text{Supernodal} & \text{Reduced} \\
F(:,J) & G & S & R
\end{array}
$$

Figure 4.5: Supernodal and symmetrically reduced structures.

$$
\begin{pmatrix}
1 & & & \\
2 & & & \\
& 3 & & \\
& & 4 & \\
& & 5 & \\
\bullet & \bullet & 6 & \\
& & & 7 \\
\cdot & \bullet & \bullet & \bullet \\
& & \bullet & \bullet \\
& \bullet & & \bullet
\end{pmatrix}
\begin{pmatrix}
\\
\bullet \\
\bullet \\
\\
\\
\circ \\
\\
\circ \\
\circ \\
\circ
\end{pmatrix}
\Longrightarrow
\begin{pmatrix}
1 & & & & \\
2 & & & & \\
& 3 & & & \\
& & 4 & & \\
& & 5 & & \\
\bullet & \bullet & 6 & & \\
& & & 7 & \\
\cdot & \bullet & \bullet & \bullet & 8 \\
& & \cdot & \bullet & \bullet \\
& \cdot & & \bullet & \bullet
\end{pmatrix}
$$

Figure 4.6: One step of symbolic factorization in the reduced structure.

and R to represent the symmetric reduction of S. It is this R that we use in our code. Note that the edges of the graph of R are directed from columns of L to rows of L.

In the figure, the small dot "$\cdot$" indicates an entry in S that was pruned from R by symmetric reduction. The $(8, 2)$ entry was pruned due to the symmetric nonzero pair $(6, 2)$ and $(2, 6)$. The figure shows the current state of the reduced structure based on the first seven columns of the filled matrix.

It is instructive to follow this example through one more column to see how symbolic factorization is carried out in the reduced supernodal structure. Consider the symbolic step for column 8. Suppose a_{28} and a_{38} are nonzero. The other nonzeros in column 8 of the factor are generated by paths in the reduced supernodal structure (we just show one possible path for each nonzero):

$$8 \xrightarrow{A^T} 2 \xrightarrow{R} 6,$$

$$8 \xrightarrow{A^T} 3 \xrightarrow{R} 8,$$

$$8 \xrightarrow{A^T} 2 \xrightarrow{R} 6 \xrightarrow{R} 9,$$

$$8 \xrightarrow{A^T} 3 \xrightarrow{R} 10,$$

Figure 4.6 shows the reduced supernodal structure before and after column 8. In column 8 of A, the original nonzeros are shown as "$\bullet$" and the filled nonzeros are shown as "$\circ$". Once the structure of column 8 of U is known, more symmetric reduction is possible. The entry $l_{10,3}$ is pruned due to the symmetric nonzeros in l_{83} and u_{38}. Also, l_{96} is pruned by l_{86} and u_{68}. Figure 4.6 shows the new structure.

The supernodal symbolic factorization relies on the path characterization in Theorem 3 and on the path-preserving property of symmetric reduction. In effect, we use the modified path condition

$$i \xrightarrow{A^T} \xRightarrow{R} j$$

on the symmetrically-reduced supernodal structure R of $L(:,J)^T$.

Finally we note that only the adjacency list of the last column in each supernode needs to be stored. We call this last column the representative of the supernode. In this example, the representatives are 2, 3, 6, and 8. Now the depth-first search traversal and the symmetric pruning work on the adjacency lists of the representative columns, that is, the supernodal graph instead of the nodal one.

4.3.3 Detecting supernodes

Since supernodes consist of contiguous columns of L, we can decide at the end of each symbolic factorization step whether the new column j belongs to the same supernode as column $j-1$.

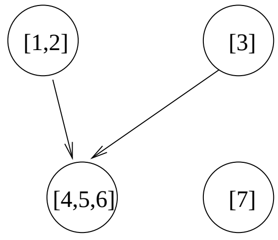
For T2 supernodes, the test is straightforward. During symbolic factorization, we test whether $L(:,j) \subseteq L(:,j-1)$ (where the containment applies to the set of nonzero indices). At the end of the symbolic factorization step, we test whether $nnz(L(:,j)) = nnz(L(:,j-1)) - 1$. Column j joins column $j-1$'s supernode if and only if both tests are passed.

T3 supernodes also require the diagonal block of U to be full. To check this, it suffices to check whether the single element u_{rj} is nonzero, where r is the first column index of the supernode. This follows from Corollary 2: $u_{rj} \neq 0$ implies that $u_{ij} \neq 0$ for all $r \leq i \leq j$. Indeed, we can even omit the test $L(:,j) \subseteq L(:,j-1)$ for T3 supernodes. If $u_{rj} \neq 0$, then $u_{j-1,j} \neq 0$, which means that column $j-1$ updates column j, which implies $L(:,j) \subseteq L(:,j-1)$. Thus we have proved the following.

Theorem 4 *Suppose a T3 supernode begins with column r and extends at least through column $j-1$. Column j belongs to this supernode if and only if $u_{rj} \neq 0$ and $nnz(L(:,j)) = nnz(L(:,j-1)) - 1$.*

For either T2 or T3 supernodes, it is straightforward to implement the relaxed versions discussed in Section 3.3. Also, since the main benefits of supernodes come when they fit into the cache, we impose a maximum size for a supernode.

$$
\begin{pmatrix}
1 & & & \\
2 & & & \\
 & 3 & & \\
 & & 4 & \\
 & & 5 & \\
\bullet & \bullet & 6 & \\
 & & & 7 \\
 & \bullet & \bullet & \bullet \\
 & & \bullet & \bullet \\
 & \bullet & & \bullet
\end{pmatrix}
\qquad
\begin{pmatrix}
 & \\
\bullet & \\
\bullet & \\
 & \bullet \\
 & \circ \\
\circ & \bullet \\
 & \bullet \\
\circ & \circ \\
\circ & \bullet \\
\circ & \bullet
\end{pmatrix}
$$



Reduced supernodal R $\quad A(:,8{:}9)$

Figure 4.7: The supernodal directed graph corresponding to $L(1{:}7, 1{:}7)^T$.

4.3.4 Panel depth-first search

The supernode-panel algorithm consists of an outer factorization (applying updates from supernodes to the active panel) and an inner factorization (applying supernode-column updates within the active panel). Each has its own symbolic factorization. The outer symbolic factorization happens once per panel, and determines two things: (1) a single column structure, which is the union of the structures of the panel columns, and (2) which supernodes update each column of the panel, and in what order. This is the information that the supernode-panel update loop in Figure 4.2 needs.

The inner symbolic factorization happens once for each column of the panel, interleaved column by column with the inner numeric factorization. In addition to determining the nonzero structure of the active column and which supernodes within the panel will update the active column, the inner symbolic factorization is also responsible for forming supernodes (that is, for deciding whether the active column will start a new supernode) and for symmetric structural pruning. The inner symbolic factorization is, therefore, exactly the supernode-column symbolic factorization described above.

The outer symbolic factorization must determine the structures of columns j to $j+w-1$, i.e., the structure of the whole panel, and also a topological order for $U(1{:}j, j{:}j+w-1)$ *en masse*. To this end, we developed an efficient panel depth-first search scheme, which is a slight modification of the column DFS. The panel depth-first search algorithm maintains a single postorder DFS list for all w columns of the panel. Let us call this the *PO* list, which is initially empty. The algorithm invokes the column depth-search procedure for each column from j to $j+w-1$. In the column DFS, each time the search backtracks from a vertex, that vertex is appended to the *PO* list. In the panel DFS, however, the vertex is appended to the *PO* list *only if it is not already on the list.* This gives a single list that includes every position that is nonzero in any panel column, just once; and the entire list is in (reverse) topological order. Thus the order of updates specified by the list is acceptable for each of the w individual panel columns.

We illustrate the idea in Figure 4.7, using the sample matrix from Figure 4.5 and 4.6 and a panel of width two. The first seven columns have been factored, and the current panel consists of columns 8 and 9. In the panel, nonzeros of A are shown as "$\bullet$" and fill in F is shown as "$\circ$". The depth-first search for column 8 starts from vertices 2 and 3. After that search is finished, the panel postorder list is $PO = (6, 2, 3)$. Now the depth-first search for column 9 starts from vertices 6 and 7 (not 4, since 6 is the representative vertex for the supernode containing column 4). This DFS only appends 7 to the PO list, because 6 is already on the list. Thus, the final list for this panel is $PO = (6, 2, 3, 7)$. The postorder list of column 8 is $(6, 2, 3)$ and the postorder list of column 9 is $(6, 7)$, which are simply two sublists of the panel PO list. The topological order is the reverse of PO, or $(7, 3, 2, 6)$. In the loop of line 3 of Figure 4.2, we follow this topological order to schedule the updating supernodes and perform numeric updates to columns of the panel.

4.4 Test matrices

To evaluate our algorithms, we have collected matrices from various sources, with their characteristics summarized in Table 4.1.

Some of the matrices are from the Harwell-Boeing collection [31]. Many of the larger matrices are from the ftp site maintained by Tim Davis of the University of Florida. Those matrices are as follows. MEMPLUS is a circuit simulation matrix from Steve Hamm of Motorola. RDIST1 is a reactive distillation problem in chemical process separation calculations, provided by Stephen Zitney at Cray Research, Inc. SHYY161 is derived from a direct, fully-coupled method for solving the Navier-Stokes equations for viscous flow calculations, provided by Wei Shyy of the University of Florida. GOODWIN is a finite element matrix in a nonlinear solver for a fluid mechanics problem, provided by Ralph Goodwin of the University of Illinois at Urbana-Champaign. VENKAT01, INACCURA and RAEFSKY3/4 were provided by Horst Simon of NASA. VENKAT01 comes from an implicit 2-D Euler solver for an unstructured grid in a flow simulation. RAEFSKY3 is from a fluid structure interaction turbulence problem. RAEFSKY4 is from a buckling problem for a container model. BAI is from solving an unsymmetric eigenvalue problem, provided by Zhaojun Bai of the University of Kentucky. EX11 is from a 3-D steady flow calculation in the SPARSKIT collection maintained by Yousef Saad at University of Minnesota. WANG3 is from solving a coupled nonlinear PDE system in a 3-D ($30 \times 30 \times 30$ uniform mesh) semiconductor device simulation, as provided by Song Wang of the University of New South Wales, Sydney. VAVASIS3 is an unsymmetric augmented matrix for a 2-D PDE with highly varying coefficients [109]. DENSE1000 is a dense 1000×1000 random matrix.

The matrices are sorted in increasing order of $flops/nnz(F)$, the ratio of the number of floating-point operations to the number of nonzeros $nnz(F)$ in the factored matrix $F = U + L - I$. The reason for this order will be described in more detail in section 4.6.

This thesis does not address the performance of column preordering for sparsity. We simply use the existing ordering algorithms provided by Matlab [62]. For all matrices, except 1, 14 and 21, the columns were permuted by Matlab's minimum degree ordering of $A^T A$, also known as "column minimum degree" ordering. However, this ordering produces tremendous amount of fill for matrices 1, 14 and 21, because it only attempts to minimize the upper bound on the actual fill (Section 3.1.2), and the upper bounds are too loose in these

	Matrix	s	Rows	Nonzeros	Nonzeros/row
1	MEMPLUS	.983	17758	99147	5.6
2	GEMAT11	.002	4929	33185	6.7
3	RDIST1	.062	4134	9408	2.3
4	ORANI678	.073	2529	90158	35.6
5	MCFE	.709	765	24382	31.8
6	LNSP3937	.869	3937	25407	6.5
7	LNS3937	.869	3937	25407	6.5
8	SHERMAN5	.780	3312	20793	6.3
9	JPWH991	.947	991	6027	6.1
10	SHERMAN3	1.000	5005	20033	4.0
11	ORSREG1	1.000	2205	14133	6.4
12	SAYLR4	1.000	3564	22316	6.3
13	SHYY161	.769	76480	329762	4.3
14	GOODWIN	.642	7320	324772	44.4
15	VENKAT01	1.000	62424	1717792	27.5
16	INACCURA	1.000	16146	1015156	62.9
17	BAI	.947	23560	460598	19.6
18	DENSE1000	1.000	1000	1000000	1000
19	RAEFSKY3	1.000	21200	1488768	70.2
20	EX11	1.000	16614	1096948	66.0
21	WANG3	1.000	26064	177168	6.8
22	RAEFSKY4	1.000	19779	1316789	66.6
23	VAVASIS3	.001	41092	1683902	41.0

Table 4.1: Characteristics of the test matrices. Structural symmetry s is defined to be the fraction of the nonzeros matched by nonzeros in symmetric locations. None of the matrices are numerically symmetric.

cases. When these three matrices were symmetrically permuted by Matlab's symmetric minimum degree ordering on $A + A^T$, the amount of fill is much smaller than using column minimum degree ordering.

4.5 Performance on an IBM RS/6000-590

In this section we carry out numerical experiments on an IBM RS/6000-590 to demonstrate the efficiency of our new code, SuperLU. The CPU clock rate of this machine is 66.5 MHz. The processor has two branch units, two fixed-point units, and two floating-point units, which can all operate in parallel if there are no dependencies. In particular, each FPU can perform two operations (a multiply and an add or subtract) at every cycle. Thus, the peak floating-point performance is 266 Mflops. The data cache is of size 256 KB with 256-byte lines, and is 4-way set-associative with LRU replacement policy. There is a separate 32 KB instruction cache. The size of the main memory is 768 MB. The SuperLU algorithm is implemented in C, using double precision arithmetic; we use the AIX xlc compiler with -O3 optimization.

In the inner loops of our sparse code, we call the two dense BLAS-2 routines DTRSV (triangular solve) and DGEMV (matrix-vector multiply) provided in the IBM ESSL library [77], whose BLAS-3 matrix-matrix multiply routine (DGEMM) achieves about 250 Mflops when the dimension of the matrix is larger than 60 [1]. In our sparse algorithm, we find that DGEMV typically accounts for more than 80% of the floating-point operations, as depicted in Figure 4.8. This percentage is higher than 95% for many matrices. Our measurements reveal that for typical dimensions arising from the benchmark matrices, DGEMV achieves roughly 235 Mflops if the data are from cache. If the data are fetched from main memory, this rate can drop by factors of 2 to 3.

The BLAS speed is clearly an upper bound on the overall factorization rate. However, because symbolic manipulation usually takes a nontrivial amount of time, this bound could be much higher than the actual sparse code performance. Figure 4.9 shows the fraction of the total factorization spent in numeric computation. For matrices 1 and 2, the program spent less than 35% of its time in the numeric part. Compared to the others, these two matrices are sparser, have less fill, and have smaller supernodes, so our supernodal techniques are less applicable. Matrix 2 is also highly unsymmetric, which makes the symmetric structural reduction technique less effective. However, it is important to note that the execution times for these two matrices are small. We also note that neither symbolic nor numeric time dominates the other, which means the symbolic algorithms are efficient as well.

Table 4.2 presents the absolute performance of the SuperLU code on this system. All floating point computations are in double precision. The third column gives the nonzero growth factor in F. For larger and denser matrices such as $17-21$, we achieve between 110 and 125 Mflops, which is about half of the machine peak. These matrices take much longer to factor, which could be a serious bottleneck in an iterative simulation process. Our techniques are successful in reducing the solution times for this type of problem.

For a dense 1000×1000 matrix, our code achieves 117 Mflops. This compares with 168 Mflops reported in the LAPACK manual [8] on a matrix of this size. That is

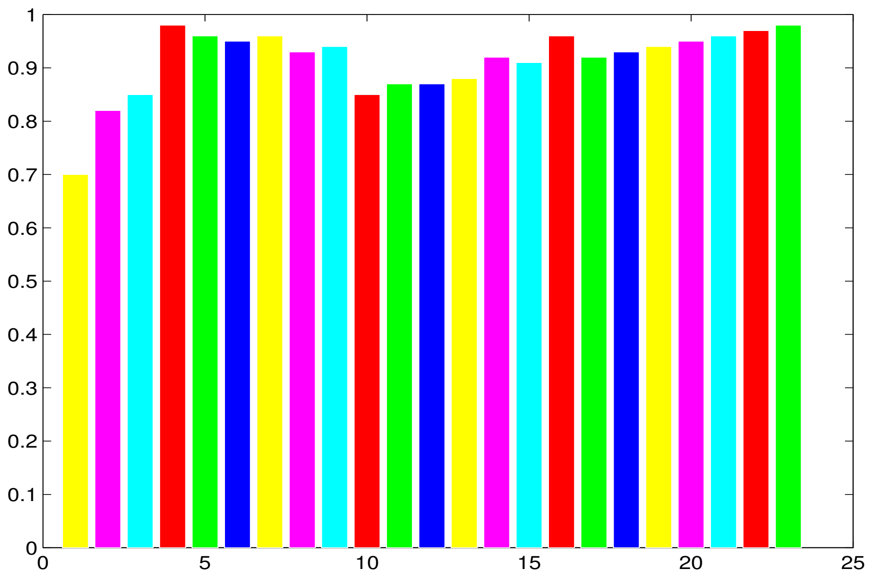


Figure 4.8: Percentage of the flops spent in DGEMV routine.

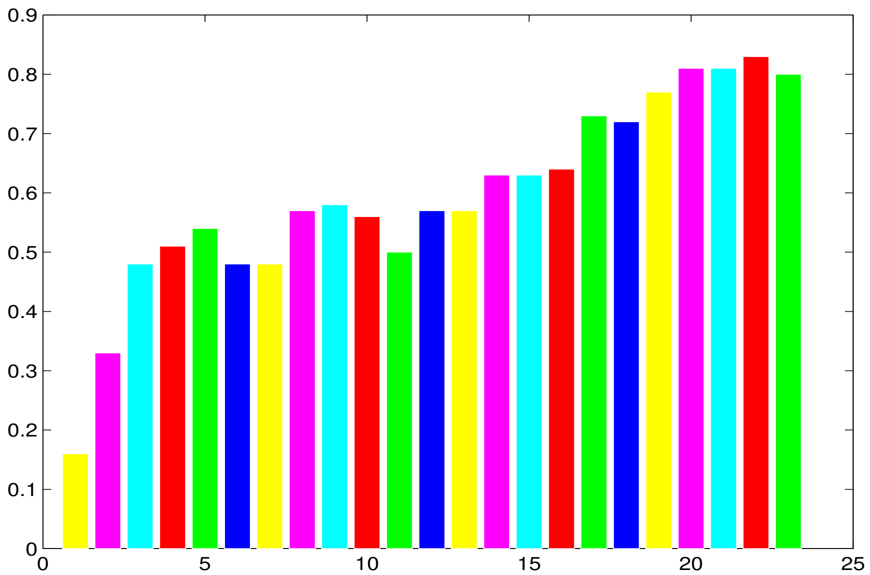


Figure 4.9: Percentage of the runtime spent in numeric factorization on an IBM RS/6000-590.

	Matrix	$nnz(F)$	$\frac{nnz(F)}{nnz(A)}$	#flops (10^6)	Seconds	Mflops
1	MEMPLUS	140388	1.4	1.8	0.57	3.08
2	GEMAT11	93370	2.8	1.5	0.27	5.64
3	RDIST1	338624	3.6	12.9	0.96	13.47
4	ORANI678	280788	3.1	14.9	1.11	13.48
5	MCFE	69053	2.8	4.1	0.24	17.42
6	LNSP3937	427600	16.8	38.9	1.50	25.97
7	LNS3937	449346	17.7	44.8	1.65	27.16
8	SHERMAN5	249199	12.0	25.2	0.82	30.78
9	JPWH991	140746	23.4	18.0	0.52	34.57
10	SHERMAN3	433376	21.6	60.6	1.37	44.24
11	ORSREG1	402478	28.5	59.8	1.21	49.42
12	SAYLR4	654908	29.3	104.8	2.18	48.07
13	SHYY161	7634810	23.2	1571.6	25.42	61.83
14	GOODWIN	3109585	9.6	665.1	12.55	52.99
15	VENKAT01	12987004	7.6	3219.9	42.99	74.90
16	INACCURA	9941478	9.8	4118.7	67.73	60.81
17	BAI	13986992	30.4	6363.7	75.91	83.83
18	DENSE1000	1000000	1.0	666.2	5.68	117.28
19	RAEFSKY3	17544134	11.8	12118.7	107.60	112.62
20	EX11	26207974	23.8	26814.5	247.05	108.54
21	WANG3	13287108	74.9	14557.5	116.58	124.86
22	RAEFSKY4	26678597	20.3	31283.4	263.13	118.89
23	VAVASIS3	49192880	29.2	89209.3	786.94	113.36

Table 4.2: Performance of SuperLU on an IBM RS/6000-590.

to say, when input matrix is dense, our sparse code achieves roughly 70% efficiency of a state-of-the-art dense code.

4.6 Understanding cache behavior and parameters

In this section, we analyze the behavior of SuperLU in detail. We wish to understand when our algorithm is significantly faster than other algorithms. We would like performance-predicting variable(s) that depend on "intrinsic" properties of the problem, such as the sparsity structure, rather than algorithmic details and machine characteristics. We begin by analyzing the speedups of the enhanced codes over the base GP implementation. Figures 4.10, 4.11 and 4.12 depict the speedups and the characteristics of the matrices, with panel size $w = 8$.

4.6.1 How much cache reuse can we expect?

As discussed in Section 4.2, the supernode-panel algorithm gets its primary gains from improved data locality, by reusing a cached supernode several times. To understand how much cache reuse we can hope for, we computed two statistics: *ops-per-nz* and *ops-per-ref*. After defining these statistics carefully, we discuss which more successfully measures reuse.

Ops-per-nz is simply calculated as $\#flops/nnz(F)$, the total number of floating point operations per nonzero in the filled matrix F. If there were perfect cache behavior, i.e., one cache miss per data item (ignoring the effect of cache line size), then *ops-per-nz* would exactly measure the amount of work per cache miss. In reality, *ops-per-nz* is an upper bound on the reuse. Note that *ops-per-nz* depends only on the fact that we are performing Gaussian elimination with partial pivoting, not on algorithmic or machine details. *Ops-per-nz* is a natural measure of potential data reuse, because it has long been used to distinguish among the different levels of BLAS, for example, for an $n \times n$ matrix-matrix multiplication (BLAS-3) versus matrix-vector multiplication (BLAS-2).

In contrast, *ops-per-ref* provides a lower bound on cache reuse, and does depend on the details of the SuperLU algorithm. *Ops-per-ref* looks at each supernode-panel update separately, and assumes that all the associated data is outside the cache before beginning the update. This pessimistic assumption limits *ops-per-ref* to twice the panel size, $2\,w$.

Now we define *ops-per-ref* more carefully. Consider a single update from supernode $(r\colon s)$ to panel $(j\colon j + w - 1)$. Depending on the panel's nonzero structure, each entry in the updating supernode is used to update from 1 to w panel columns. Thus each entry in the updating supernode participates in between 0 and $2\,w$ floating point operations during a supernode-panel update. We assume that the supernode entry is brought into cache from main memory exactly once for the entire sup-panel update, if it is used at all. Thus, during a single sup-panel update, each entry accessed in the updating supernode accounts for between 2 and $2\,w$ *operations per reference*. The *ops-per-ref* statistic is the average of this number over all entries in all sup-panel updates. It measures how many times the average supernode entry is used each time it is brought into cache from main memory. *Ops-per-ref* ranges from 2 to $2\,w$, with larger values indicating better cache use. If there is

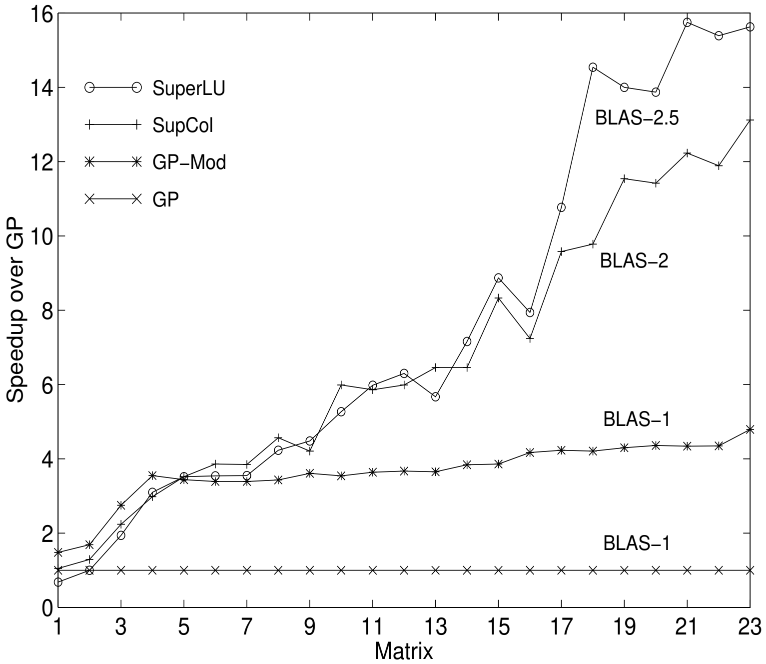


Figure 4.10: Speedups of each enhancement over GP code, on an IBM RS/6000-590.

little correlation between the row structures of the columns in each panel, *ops-per-ref* will be small; if there is perfect correlation, as in a dense matrix, it will be close to $2\,w$.

Now we describe how we compute the average *ops-per-ref* for the entire factorization. For each updating supernode $(r\colon s)$ and each panel $(j\colon j+w-1)$ (see Figure 4.2), define

$$ksmin = \min_{j \le jj < j+w,\ r \le i \le s} \{ i \mid A(i,jj) \neq 0 \}.$$

Then $nnz(L(r\colon n, ksmin\colon s))$ entries of the supernode are referenced in the sup-panel update. The dense triangular solve in column jj of the update takes $(s-ks+1)\cdot(s-ks)$ flops, where $ks = min_{r \le i \le s}\{i \mid A(i,jj) \neq 0\}$. The matrix-vector multiply uses $2\cdot(s-ks+1)\cdot nnz(L(s+1\colon n, s))$ flops. We count both additions and multiplications. For all panel updates, we sum the memory reference counts and sum the flop counts, then divide the second sum by the first to arrive at an average *ops-per-ref*.

Now we compare the predictive powers of *ops-per-nz* (Figure 4.11 (a)) and *ops-per-ref* (Figure 4.11 (b)) in predicting speedup (Figure 4.10). The superiority of *ops-per-nz* is evident; it is much more strongly correlated with the speedup of SuperLU than *ops-per-ref*. This is good news, because *ops-per-nz* measures the best case reuse, and *ops-per-ref* the worst case. But neither statistic captures all the variation in the performance. In future work, we hope to use a hardware monitor to measure the exact cache reuse rate. (This data could also be obtained from a simulator, but the matrices we are interested in are much too large for a simulator to be viable.)

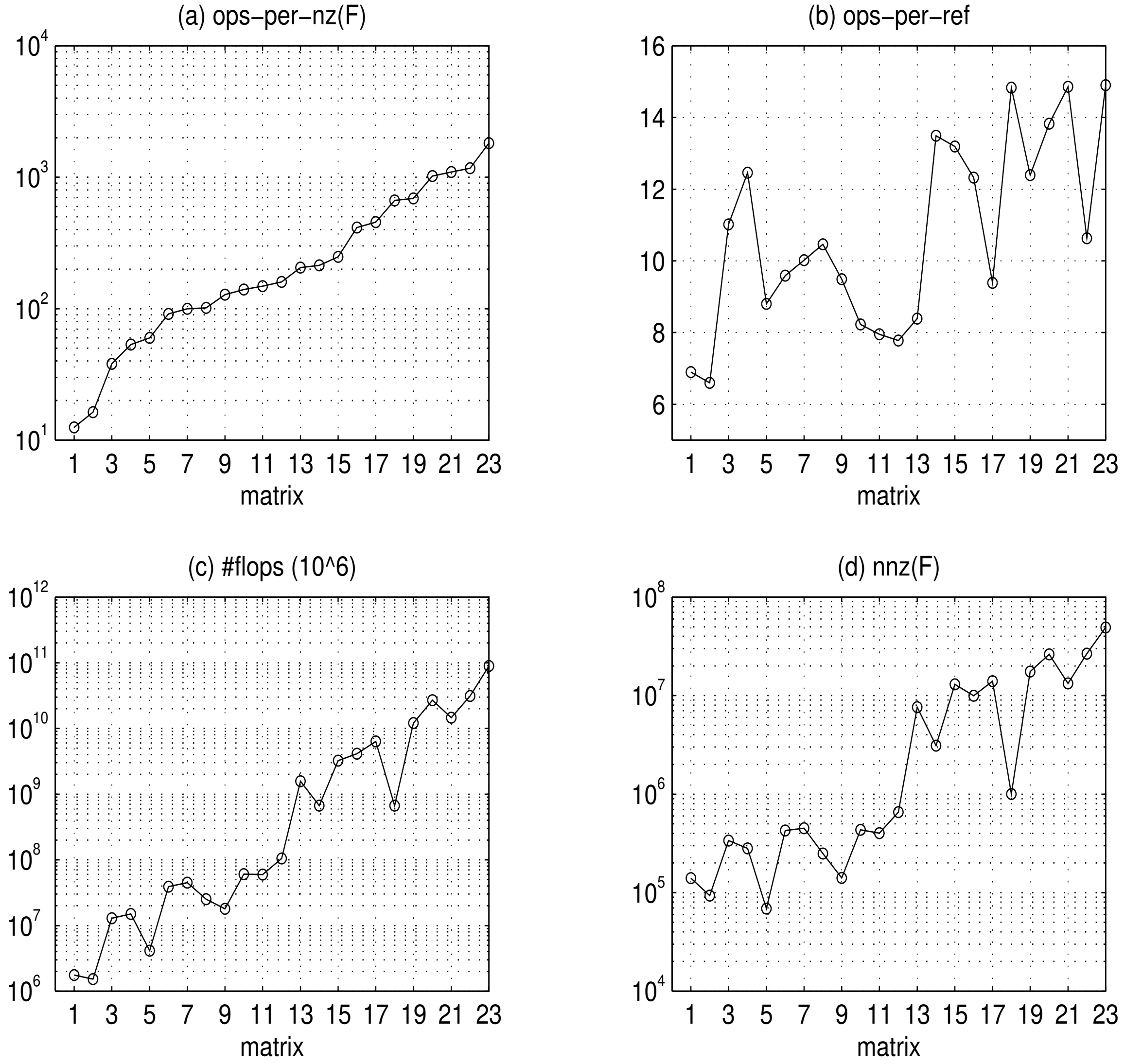


Figure 4.11: Some characteristics of the matrices.

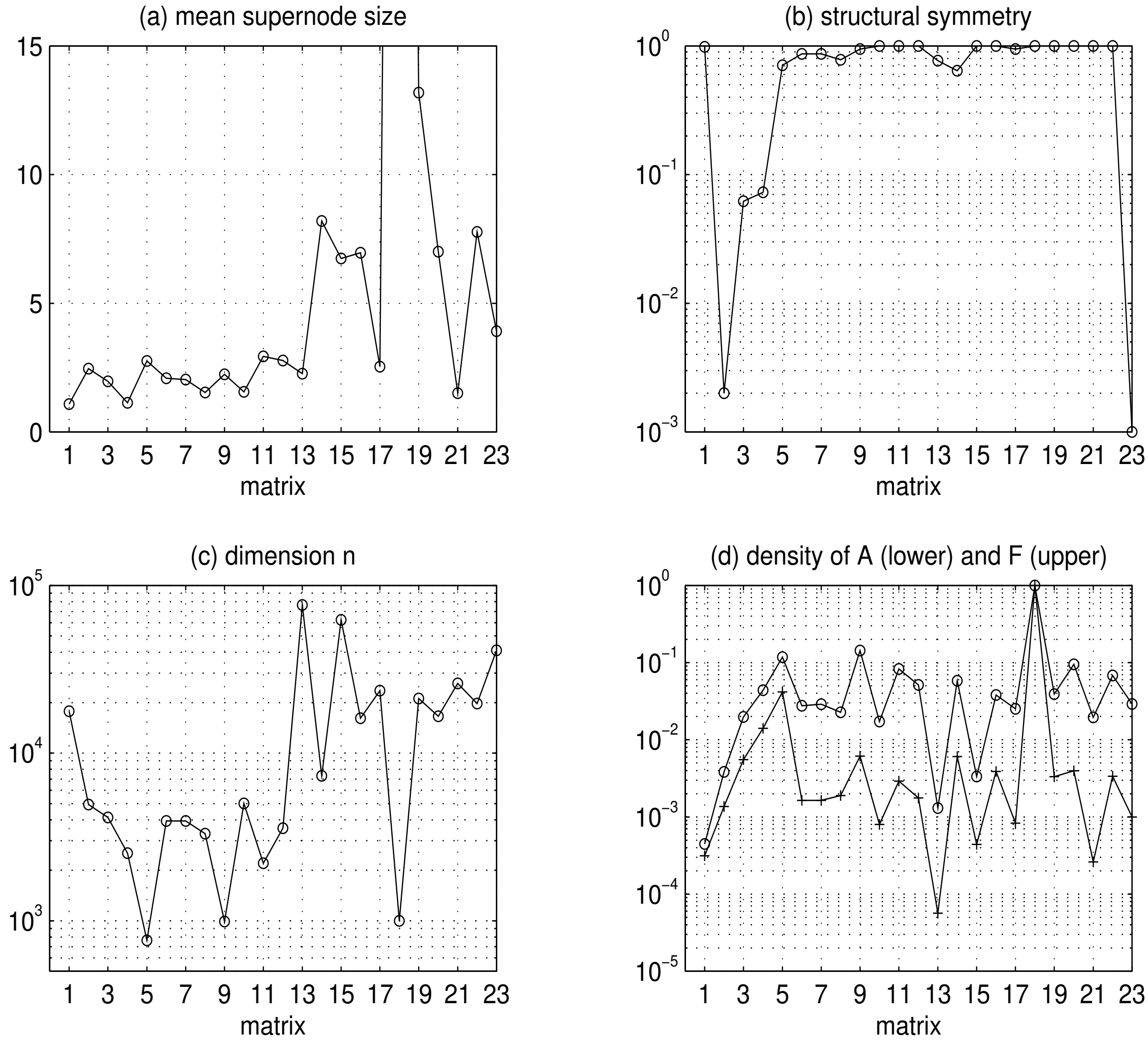


Figure 4.12: Some intrinsic properties of the matrices.

4.6.2 How large are the supernodes?

The supernode size determines the size of the matrix to be passed to matrix-vector multiply and other BLAS-2 routines in our algorithm. Figure 4.12(a) shows the average number of columns in the supernodes of the matrices, after amalgamating the relaxed supernodes at the bottom of the column etree (Section 3.4). The average size is usually quite small.

More important than average size is the distribution of supernode sizes. In sparse Gaussian elimination, more fill tends to occur in the later stages. Usually there is a large percentage of small supernodes corresponding to the leaves of the column etree, even after amalgamation. Larger supernodes appear nearer the root. In Figure 4.13 we plot the histograms of the supernode size for four matrices chosen to exhibit a wide range of behavior. In the figure, the number at the bottom of each bar is the smallest supernode size in that bin. The mark "o" at the bottom of a bin indicates zero occurrences; otherwise, a "$*$" is put at the bottom of a bin. Relaxed supernodes of granularity $r = 4$ are used. Matrix 1 has 16378 supernodes, all but one of which have less than 12 columns; the single large supernode, with 115 columns, is the dense submatrix at the bottom right corner of F. Matrix 14 has more supernodes distributed over a wider spectrum; it has 13 supernodes with 54 to 59 columns. This matrix gives greater speedups over the non-supernodal codes.

Figure 4.12 also plots three other properties of each matrix: structural symmetry, dimension, and density. None of them have any significant correlation with the performance. The effectiveness of symmetric reduction depends on F being structurally symmetric, which depends on the choice of pivots. So, structural symmetry of A does not give any useful information.

We note that the speedup achieved by the dense 1000×1000 problem (matrix 18) show the best performance gain over SupCol, because this matrix has large supernodes and exhibits ideal data reuse. It achieves a speedup of 1.43 on the RS/6000-590. The gain for any sparse matrix should be smaller than this on this machine.

4.6.3 Blocking parameters

In our hybrid blocking algorithm (Figure 4.4), we need to select appropriate values for the parameters that describe the two-dimensional data blocking: panel width w, maximum supernode size t, and row block size b, see Figure 4.14. The key considerations are that the active data we access in the inner loop (the *working set*) should fit into the cache, and that the matrices presented to the BLAS-2 routine DGEMV should be the sizes and shapes for which that routine is optimized. Here we describe in detail the methodology we used to choose parameters for the IBM RS/6000. The methodology can be employed on other machines as well, with the block sizes adapted to the cache sizes.

- DGEMV **optimization.** As indicated in Figure 4.8, the majority of the floating-point operations are in the matrix-vector multiply. The dimensions (m, n) of the matrices in calls to DGEMV vary greatly depending on the supernode dimensions. Very often, the supernode is a tall and skinny matrix, that is, $m \gg n$. We measured the DGEMV Mflops rate for various m and n, and present a contour plot in the (m, n) plane in Figure 4.15(a). Each contour represents a constant Mflops rate. The

(a) Matrix 1: 17758 rows, 16378 supernodes (b) Matrix 2: 4929 rows, 2002 supernodes

(c) Matrix 3: 4134 rows, 2099 supernodes (d) Matrix 14: 7320 rows, 893 supernodes

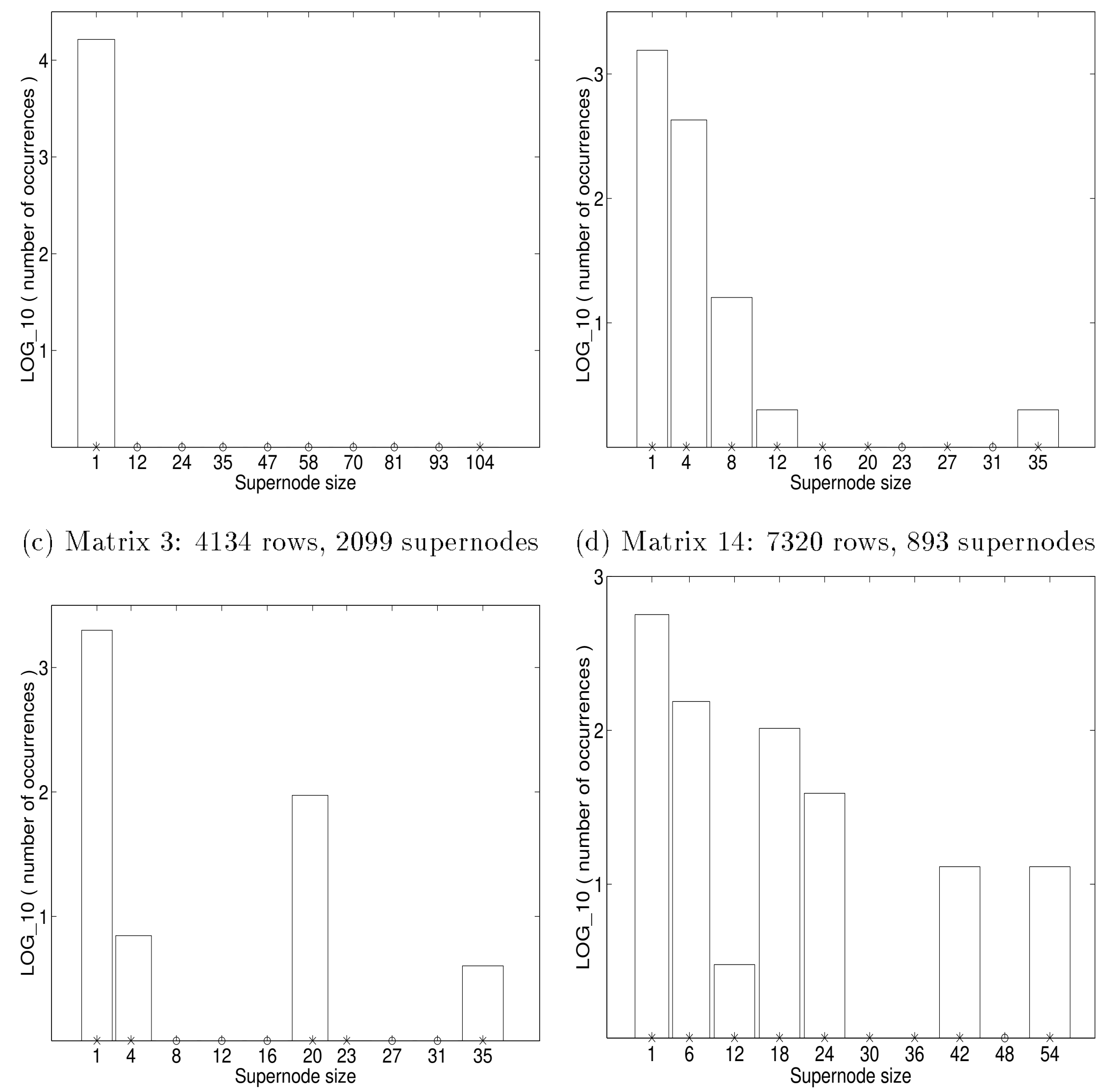


Figure 4.13: Distribution of supernode size for four matrices.

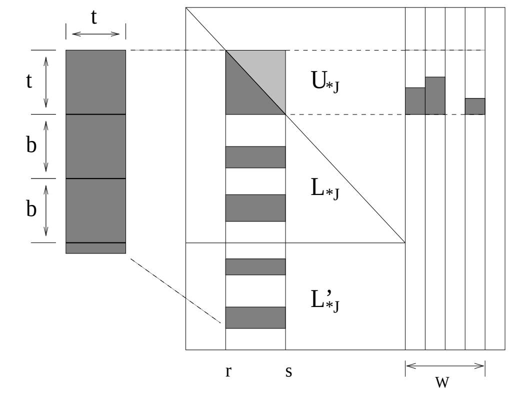


Figure 4.14: Parameters of the working set in the 2-D algorithm.

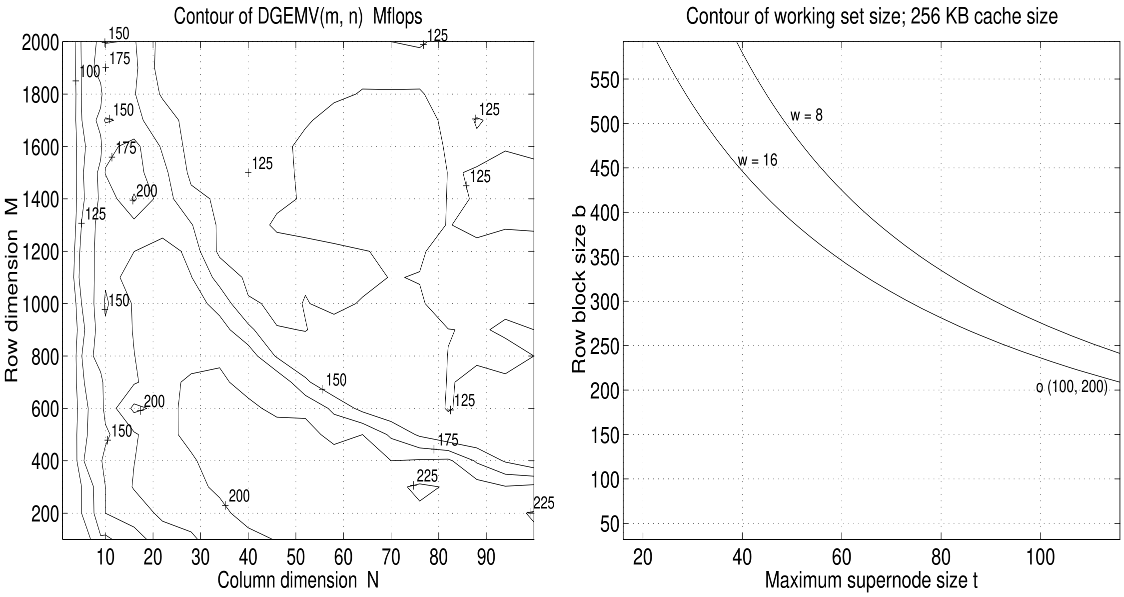


Figure 4.15: (a) Contour plot of DGEMV performance. (b) Contour plot of working set in 2-D algorithm.

dashed curve represents $mn = 32\mathrm{K}$ double floats, or a cache capacity of 256 KB. In the optimum region, we achieve more than 200 Mflops; outside this region, performance drops either because the matrices exceed the cache capacity, or because the column dimension n is too small.

- **Working set.** By studying the data access pattern in the inner loop of the 2-D algorithm, lines (7–9) in Figure 4.4, we find that the working set size is the following function of w, t, and b, as shown in Figure 4.14:

$$WS = \underbrace{b \times t}_{\text{row block from supernode}} + \underbrace{(t+b) \times w}_{\text{vectors in matrix-vector multiply}} + \underbrace{b \times w}_{\text{part of SPA structure}} .$$

  In Figure 4.15(b), we fix two w values, and plot the contour lines for $WS = 32\mathrm{K}$ in the (t, b) plane. If the point (t, b) is below the contour curve, then the working set can fit in a cache of 32K double floats, or 256 KB.

Based on this analysis, we use the following rules to set the parameters.

First we choose w, the width of the panel in columns. Larger panels mean more reuse of cached data in the outer factorization, but also mean that the inner factorization (by the sup-col algorithm) must be applied to larger matrices. We find empirically that the best choice for w is between 8 and 16. Performance tends to degrade for larger w.

Next we choose b, the number of rows per block, and t, the maximum number of columns in a supernode. Recall that b and t are upper bounds on the row and column dimensions of the call to DGEMV. We choose $t = 100$ and $b \approx 200$, which guarantees that the working set fits in cache (per Figure 4.15(b)), and that we can hope to be near the optimum region of DGEMV performance (per Figure 4.15(a)).

Recall that b is relevant only when we use row-wise blocking, that is, when the test "**if** supernode $(r\colon s)$ is large" succeeds at line 4 of Figure 4.4. This implies that the 2-D scheme adds overhead only if the updating supernode is large. In the actual code, the test for a large supernode is

if $ncol > 40$ **and** $nrow > b$ **then** the supernode is large,

where $nrow$ is the number of dense rows below the diagonal block of the supernode, $ncol$ is the number of dense columns of the supernode updating the panel, i.e., $ncol = s - r + 1$. In practice, this choice usually gives the best performance.

The best choice of the parameters w, t, and b depends on the machine architecture and on the BLAS implementation, but it is largely independent of the matrix structure. Thus we do not expect each user of SuperLU to choose values for these parameters. Instead, our library code provides an inquiry function that returns the parameter values, much in the spirit of the LAPACK environment routine ILAENV. The machine-independent defaults will often give satisfactory performance. The methodology we have described here for the RS/6000 can serve as a guide for users who want to modify the inquiry function to give optimal performance for particular computer systems.

4.7 Register reuse

Commercial microprocessor performance has increased dramatically, largely driven by CMOS fabrication technology improvements. It is now common to see superscalar processors capable of executing up to four scalar instructions in every cycle. Given multiple floating-point units operating in parallel, the peak speed is attainable only if all the source operands to the FPUs are in registers upon execution. In such systems, not only is cache reuse important, but sufficient reuse of data in registers becomes vital as well. Register reuse can reduce the load/store frequency and bandwidth requirement between registers and cache. We will illustrate this by studying in more detail the performance of DGEMV and of the complete factorization on the two different superscalar microprocessors.

4.7.1 Performance on the MIPS R8000

As seen from Section 4.5, the IBM RS/6000-590 (POWER2 family) has a peak potential performance of four floating-point operations per cycle. To allow for such a sustained rate, the IBM POWER2 provides unique *quad* load/store instructions, which double the effective bandwidth between floating-point registers (FPRs) and cache. This quad load/store capability together with a large factor of inner loop unrolling are the keys for the matrix-vector kernel to achieve nearly peak performance. The code segment in Figure 4.16 for DGEMV ($y \leftarrow y + A * x$) explains the reason. Here, the x_j's and y_i's denote the FPRs to temporarily hold the respective values from vectors x and y. Loop 4–10 performs multiplication of a R-by-C block of matrix A with a C-by-1 subvector of x. The column-wise block size C can be chosen to minimize the finite cache and TLB effects. The row-wise block size R (or unrolling factor) depends on the number of available FPRs. For example, $R = 24$ is appropriate for the IBM POWER2, because there are 32 floating-point registers. In the multiply-add instructions of the innermost loop 6–9 (which should be fully unrolled in the actual code), all operands except the two entries from A are already in the FPRs. Since $A(I+i,j)$ and $A(I+i+1,j)$ are stored contiguously in memory, only one quad load is needed to load both entries into the two FPRs, and to feed both FPUs at the peak rate of two multiply-adds in every cycle. That is essentially how the DGEMV routine in the IBM ESSL library achieves close to peak performance.

The quad load/store data path on the IBM POWER2 is a nice feature but is very expensive to implement from hardware point of view, and not many RISC processors provide this capability. We now study another high performance architecture, the MIPS R8000 chip set, to see what the peak DGEMV performance is provided that the source matrix A is in cache. The CPU has a clock frequency of 90 MHz. By four-way superscalar implementation, the processor can dispatch up to four instructions per cycle, including two integer and two floating-point instructions. The two floating-point instructions can be a pair of multiply-add (MADD) instructions, resulting in a throughput of up to four floating-point operations per cycle and 360 Mflops peak rate. However, unlike the IBM POWER2, there is no quad load/store instruction. Each load can supply only one 64-bit double-word per cycle. Therefore, the DGEMV inner loop of Figure 4.16 is now limited by load/store bandwidth. In the vendor-supplied BLAS library, the DGEMV routine achieves at most 210 Mflops, roughly 58% of the machine peak. We are therefore motivated to design a

```
for J = 1 to n, step C, do
    for I = 1 to n, step R, do
        for i = 0 to R - 1 do y_i = y(I + i);
        for j = J to J + C - 1 do
            x_j = x(j);
            for i = 0 to R - 1, step 2, do
                y_i = y_i + A(I + i, j) * x_j;
                y_{i+1} = y_{i+1} + A(I + i + 1, j) * x_j;
            endfor;
        endfor;
    endfor;
endfor;
```

Figure 4.16: A code segment to implement DGEMV.

new kernel, which we call DGEMV2, that multiplies a matrix with two vectors altogether ($y1 \leftarrow y1 + A * x1$, $y2 \leftarrow y2 + A * x2$). In addition to matrix A, DGEMV2 also takes two source vectors $x1, x2$ and two destination vectors $y1, y2$. The DGEMV2 routine can be written as in Figure 4.17.

Again, we assume that $x1_j$'s and $x2_j$'s are the FPRs holding the elements from the two source vectors, $y1$ and $y2$ are the two FPRs holding the elements from the two destination vectors. Then the load requirement in the inner loop 9–11 is only one 64-bit double-word for $A(i, j)$. This one load can supply two MADDs or four floating-point operations per cycle. Since the R8000 processor has 32 FPRs, the level of unrolling factor C can be set to 8. That amounts to 16 FPRs used by the source vectors $x1$ and $x2$, 2 FPRs used by the destination vectors $y1$ and $y2$, and 8 FPRs used by the elements from A. With this level of unrolling, a block column computation in the loop 6–15 performs 32 floating-point operations, does 8 loads for the A entries, 2 loads and 2 stores for the y entries. In particular, each $A(i, j)$ is reused across four floating-point operations. Compared with the 2-flops-per-load ratio in DGEMV, the DGEMV2 kernel certainly reuses registers better and lessens the load/store requirement. This kernel uncovers another half of the peak speed which is not achievable by DGEMV. Figure 4.18 shows our measurement of the performance of DGEMV2, and the vendor optimized BLAS routines DGEMM and DGEMV. For the C routine implementing DGEMV2, we use the `cc` compiler with options `-O3 -mips4 -64 -OPT:alias=restrict` in order for the compiler to generate most efficient code. From the figure we see that DGEMV2 achieves over 95% of the machine peak for matrices of wide range of dimensions. It sometimes performs better than the vendor-supplied DGEMM routine. When the matrix size exceeds cache capacity (4 MB), its performance degrades substantially, but is still better than DGEMV. Overall, DGEMV2 is roughly 70% faster than DGEMV when the matrix fits in the cache.

Now we need to alter the *LU* factorization algorithm in order to call DGEMV2. Recall that the supernode-panel update involves two matrices A and B, with A being (part

```
for J = 1 to n, step C, do
    for j = J to J + C − 1 do
        x1_j = x1(j);   x2_j = x2(j);
    endfor;
    for i = 1 to n do
        y1 = y1(i);   y2 = y2(i);  /* load */
        for j = J to J + C − 1 do
            y1 = y1 + A(i, j) ∗ x1_j;
            y2 = y2 + A(i, j) ∗ x2_j;
        endfor;
        y1(i) = y1;   y2(i) = y2;  /* store */
    endfor;
endfor;
```

Figure 4.17: A code segment to implement DGEMV2.

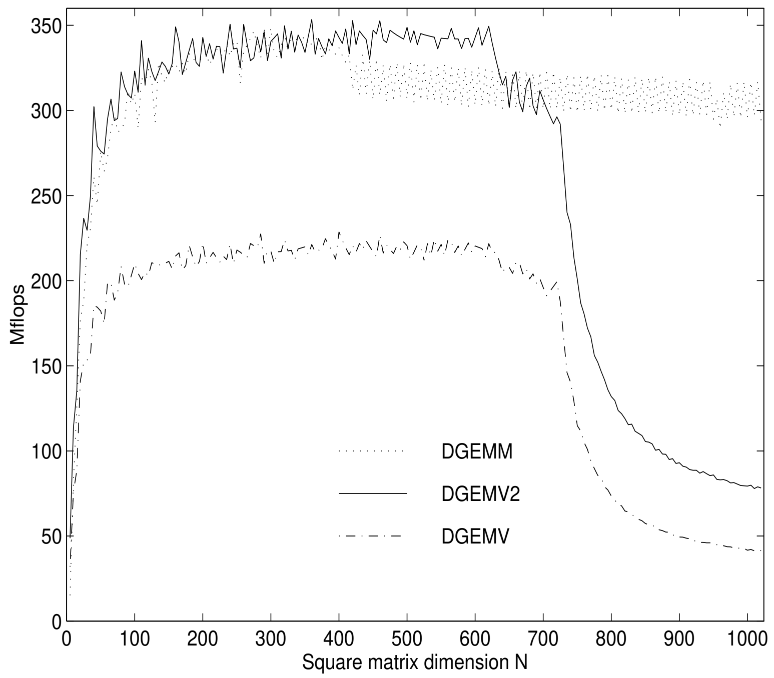
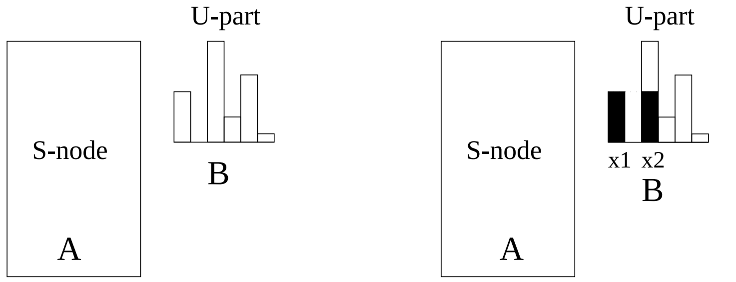


Figure 4.18: Measurement of the double-precision DGEMV2, DGEMV and DGEMM on the MIPS R8000.



(a) DGEMV kernel (b) DGEMV2 kernel

Figure 4.19: Supernode-panel update using DGEMV or DGEMV2 kernels.

of) a supernode and B being several column segments from U, in skyline form, as shown in Figure 4.19 (a). In general the vectors in matrix B are of different lengths. As illustrated in Figure 4.19 (b), we can call DGEMV2 with the pair of adjacent vectors, $x1$ and $x2$, using the length of the shorter vector $x1$. For the other part of the longer vector $x2$, we still use DGEMV with the corresponding columns in matrix A. A more elaborate scheme might be to pair up the vectors in decreasing order of their lengths, so that vectors of greater lengths are used in DGEMV2. However, this requires sorting and may be costly to implement.

Table 4.3 shows the overall factorization rate on a MIPS R8000 when using the two different kernels DGEMV and DGEMV2. The second to last column shows the improvement over DGEMV. Depending on how much opportunity there is to use DGEMV2, some matrices achieve better speedup. The average performance gain is about 25%.

4.7.2 Performance on the DEC Alpha 21164

In this subsection, we study another superscalar architecture, the DEC Alpha 21164, to see whether the DGEMV2 kernel is helpful to improve the performance. The CPU has a clock frequency of 300 MHz, and is capable of issuing four instructions per cycle. The processor has one floating-point add pipeline and one floating-point multiply pipeline, with a throughput of two floating-point operations per cycle. The peak floating-point rate is therefore 600 Mflops. Inside the chip is an 8 KB direct-mapped Level 1 instruction cache, and an 8 KB direct-mapped write-through Level 1 data cache. Also on the chip there is a 96 KB 3-way set-associative write-back Level 2 unified cache. Off the chip there is a 4 MB direct-mapped Level 3 cache. This organization of multiple levels of (small) caches makes it somewhat more difficult to achieve good performance than on the machines with simpler cache systems. By measurement, the DGEMM routine from DEC's DXML library achieves at most about 350 Mflops, far from the peak.

We implemented a DGEMV2 routine using Fortran 77, and compiled it with `-O5 -fast -tune ev5`. Figure 4.20 shows our measurement of the performance of DGEMV2 and the vendor optimized DGEMV. Surprisingly, DGEMV2 is not drastically faster than DGEMV, as we saw on the MIPS R8000. For the matrices that fit into the L2 cache, the gain of DGEMV2 over DGEMV is only about 40 Mflops. One possible explanation is that our hand-coded DGEMV2 is not adequately optimized for this architecture. Further

	Matrix	DGEMV Mflops	DGEMV2 Mflops	Speedup	DGEMV2 Seconds
1	Memplus	2.40	2.47	1.03	0.71
2	Gemat11	4.36	5.87	1.35	0.26
3	Rdist1	12.53	13.17	1.05	0.98
4	Orani678	12.27	13.01	1.06	1.15
5	Mcfe	15.92	17.99	1.13	0.23
6	Lnsp3937	21.90	28.88	1.32	1.35
7	Lns3937	22.54	30.10	1.34	1.49
8	Sherman5	27.43	35.55	1.30	0.71
9	Jpwh991	29.96	37.45	1.25	0.48
10	Sherman3	46.98	54.60	1.16	1.11
11	Orsreg1	48.63	59.23	1.22	1.01
12	Saylr4	49.90	60.22	1.21	1.74
13	Shyy161	58.87	66.95	1.34	23.48
14	Goodwin	49.82	72.30	1.45	9.20
15	Venkat01	77.93	105.71	1.36	30.46
16	Inaccura	54.49	86.44	1.59	47.65
17	Bai	92.04	107.03	1.16	59.46
18	Dense1000	133.77	177.64	1.33	3.75
19	Raefsky3	119.64	154.55	1.29	78.41
20	Ex11	109.86	129.75	1.18	205.90
21	Wang3	136.86	168.99	1.24	86.14
22	Raefsky4	116.38	142.88	1.23	218.95
23	Vavasis3	114.12	127.01	1.11	702.38

Table 4.3: Factorization rate in Mflops and time in seconds with two different kernels on a MIPS R8000.

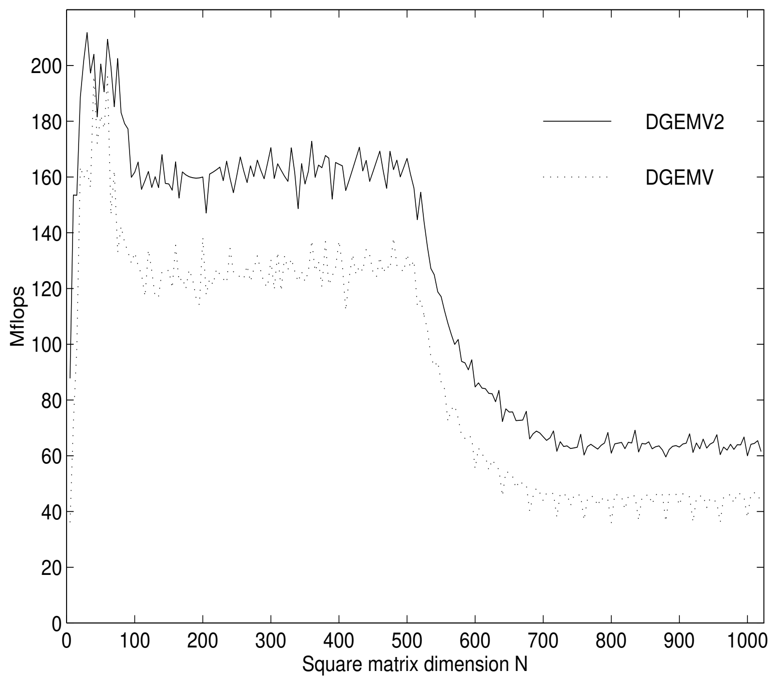


Figure 4.20: Measurement of the double-precision DGEMV2 and DGEMV on the DEC Alpha 21164.

investigation is needed to improve DGEMV2 speed.

Table 4.4 presents the overall LU factorization performance on this machine. As anticipated, the performance gain of using DGEMV2 kernel is rather moderate. For large matrices, the speedups are between 10% and 15%. For most smaller matrices the speedups are below 10%.

4.8 Comparison with previous column LU factorization algorithms

In this section, we compare the performance of SuperLU with three of its predecessors, including GP by Gilbert and Peierls [64] (Figure 2.1), GP-Mod by Eisenstat and Liu [44] (Chapter 2, and Section 4.3.2). and SupCol by Eisenstat, Gilbert and Liu [45] (Figure 4.1). GP and GP-Mod are written in Fortran; SupCol and SuperLU are written in C. (Matlab contains C implementations of GP and GP-Mod [62], which we did not test here.)

We benchmarked the above four codes on six high-end workstations from four vendors, whose characteristics are tabulated in Table 4.5. The instruction caches, if separate from the data cache, are not listed in the table. The blocking parameters for SuperLU are chosen according to the size of data cache, and are reported in each comparison table. In most cases, the on-chip L1 caches are fairly small, so we use either L2 cache or the off-chip cache as reference. Most DGEMM and DGEMV Mflop rates were measured using vendor-

	Matrix	DGEMV Mflops	DGEMV2 Mflops	Speedup	DGEMV2 Seconds
1	Memplus	4.39	4.58	1.04	0.38
2	Gemat11	9.16	10.17	1.11	0.15
3	Rdist1	22.78	23.47	1.03	0.55
4	Orani678	20.88	23.63	1.13	0.63
5	Mcfe	31.04	31.04	1.00	0.13
6	Lnsp3937	39.65	42.53	1.07	0.92
7	Lns3937	40.17	43.41	1.08	1.03
8	Sherman5	47.32	50.48	1.07	0.50
9	Jpwh991	51.36	53.93	1.05	0.33
10	Sherman3	60.60	64.93	1.07	0.93
11	Orsreg1	66.47	69.02	1.04	0.87
12	Saylr4	60.46	67.61	1.10	1.55
13	Shyy161	65.49	70.38	1.07	22.33
14	Goodwin	66.85	70.51	1.05	9.43
15	Venkat01	85.00	95.53	1.13	33.57
16	Inaccura	68.50	75.05	1.09	54.88
17	Bai	83.65	95.26	1.14	66.80
18	Dense1000	110.72	139.75	1.26	4.75
19	Raefsky3	100.60	113.92	1.13	106.42
20	Ex11	96.26	110.74	1.15	241.53
21	Wang3	105.09	121.48	1.15	119.77
22	Raefsky4	97.17	110.18	1.13	283.77
23	Vavasis3	93.63	108.11	1.15	825.37

Table 4.4: Factorization rate in Mflops and time in seconds with two different kernels on a DEC Alpha 21164.

	Clock MHz	On-chip Cache	External Cache	#Issues 1 cycle	Peak Mflops	DGEMM Mflops	DGEMV Mflops
RS/6000-590	66.5	256 KB		6	266	250	235
MIPS R8000	90	16 KB	4 MB	4	360	340	210
Alpha 21064	200	8 KB	512 KB	2	200	120	60
Alpha 21164	300	8 KB-L1 96 KB-L2	4 MB	4	600	350	135
Sparc 20	60	16 KB	1 MB	3	60	55*	–
UltraSparc-I	143	16 KB	512 KB	4	286	227*	–

Table 4.5: Machines used to compare various column LU codes.

supplied BLAS libraries. When the vendors do not supply a BLAS library, we report the results from PHiPAC [16], with an asterisk (*) beside such a number. For some machines, PHiPAC is often faster than the vendor-supplied DGEMM.

Because of physical memory limits on the Alpha 21064, the Sparc 20 and the UltraSparc-I, some large problems could not be tested.

For the Fortran codes, we use Fortran 77 compilers; for the C codes, we use ANSI C compilers. In all cases, we use highest possible optimization provided by each compiler. Both SupCol and SuperLU call Level 2 BLAS routines. For the RS/6000-590, we use the BLAS routines from IBM's ESSL library. For both Alphas, we use the BLAS routines from DEC's DXML library. There are no vendor supplied BLAS libraries on the Sparcs, so we use our own routines implemented in C.

Tables 4.6 through 4.11 present the results of comparisons on the six machines. In all these tables, the column labeled "GP" gives the raw factorization times in seconds of the GP column-column code. The numbers in each successive column are speedups achieved by the corresponding enhancement over GP. Thus, for example, a speedup of 2 means that the running time was half that of GP. The numbers in the last two rows of each table show the average speedup and its standard deviation. We make the following observations from these results.

The symmetric structure pruning in GP-Mod is very effective in reducing the graph search time. This significantly decreases the symbolic factorization time in the GP code. It achieves speedup in all problems, on all machines. Its average speedup on the RS/6000 is 3.64, the highest among all the machines.

Supernodes restrict the search to the supernodal graph, and allow the numeric kernels to employ dense BLAS-2 operations. The effects are not as dramatic as the pruning technique. For matrices $1-3$, the runtimes are actually longer than GP-Mod. This is because supernodes are often small in the sparser matrices.

Supernode-panel update reduces the cache miss rate and exploit dense substructures in the factor F. For problems without much structure, the gain is often offset by various implementation overheads. However, the advantage of SuperLU over SupCol becomes significant for larger or denser problems, or on machines with small cache. This is in part because for small problems or large caches, when SupCol factors consecutive columns,

	Matrix	GP (Seconds)	GP-Mod	SupCol	SuperLU
1	Memplus	0.40	1.48	1.05	0.68
2	Gemat11	0.27	1.69	1.29	1.00
3	Rdist1	1.90	2.75	2.24	1.94
4	Orani678	13.86	3.55	2.98	3.10
5	Mcfe	1.55	3.44	3.52	3.52
6	Lnsp3937	7.11	3.39	3.86	3.54
7	Lns3937	7.77	3.39	3.85	3.55
8	Sherman5	3.98	3.43	4.57	4.23
9	Jpwh991	2.78	3.61	4.21	4.48
10	Sherman3	7.43	3.54	5.99	5.27
11	Orsreg1	8.73	3.64	5.86	5.98
12	Saylr4	17.51	3.67	5.99	6.30
13	Shyy161	163.14	3.65	6.46	5.67
14	Goodwin	90.63	3.84	6.46	7.16
15	Venkat01	355.50	3.86	8.33	8.87
16	Inaccura	544.91	4.17	7.24	7.94
17	Bai	823.47	4.23	9.58	10.47
18	Dense1000	83.48	4.21	10.22	14.54
19	Raefsky3	1571.63	4.30	11.54	14.00
20	Ex11	3439.41	4.36	11.42	13.87
21	Wang3	1841.27	4.34	12.23	15.75
22	Raefsky4	3968.16	4.35	11.89	15.39
23	Vavasis3	12342.97	4.79	13.11	15.63
Mean			3.64	6.67	7.52
Std			0.79	3.69	5.04

Table 4.6: Speedups achieved by each enhancement over the GP column-column code, on a RS/6000-590. The blocking parameters for SuperLU are $w = 8, t = 100$ and $b = 200$.

	Matrix	GP (Seconds)	GP-Mod	SupCol	SuperLU
1	MEMPLUS	0.42	1.51	1.10	0.59
2	GEMAT11	0.29	1.77	1.61	1.11
3	RDIST1	2.03	2.58	2.07	2.07
4	ORANI678	2.26	2.61	1.61	1.96
5	MCFE	0.60	2.93	2.73	2.61
6	LNSP3937	5.13	3.23	4.17	3.80
7	LNS3937	5.74	3.32	4.22	3.85
8	SHERMAN5	3.70	3.38	5.37	5.22
9	JPWH991	2.50	3.63	4.81	5.21
10	SHERMAN3	8.73	3.78	8.08	7.87
11	ORSREG1	8.18	3.72	7.24	8.10
12	SAYLR4	14.92	3.67	7.65	8.58
13	SHYY161	235.77	3.24	7.11	10.04
14	GOODWIN	103.66	3.45	8.87	11.27
15	VENKAT01	524.46	2.95	8.51	17.22
16	INACCURA	720.86	2.93	6.36	15.13
17	BAI	1095.30	2.95	7.28	18.42
18	DENSE1000	113.28	3.34	11.99	30.21
19	RAEFSKY3	2263.80	2.88	6.54	28.87
20	EX11	5302.74	2.96	6.44	25.75
21	WANG3	2710.19	2.80	6.31	31.46
22	RAEFSKY4	6005.72	2.85	6.29	27.44
Mean			3.02	5.74	12.13
Std			0.57	2.75	10.48

Table 4.7: Speedups achieved by each enhancement over the GP column-column code, on a MIPS R8000. The blocking parameters for SuperLU are $w = 16, t = 100$ and $b = 800$.

	Matrix	GP (Seconds)	GP-Mod	SupCol	SuperLU
1	MEMPLUS	0.42	1.19	.98	.55
2	GEMAT11	0.28	1.31	1.10	.78
3	RDIST1	1.81	1.65	1.37	1.27
4	ORANI678	18.16	1.80	1.61	1.85
5	MCFE	1.63	1.90	2.12	2.09
6	LNSP3937	8.27	1.84	2.25	2.35
7	LNS3937	9.25	1.81	2.24	2.33
8	SHERMAN5	4.55	1.81	2.63	2.79
9	JPWH991	3.40	1.92	2.46	2.79
10	SHERMAN3	9.63	1.84	3.23	3.54
11	ORSREG1	11.35	1.82	3.09	2.64
12	SAYLR4	24.48	1.78	3.13	4.19
13	SHYY161	249.83	1.80	3.43	3.84
14	GOODWIN	115.40	1.77	2.82	4.19
18	DENSE1000	117.21	1.83	3.60	6.45
Mean			1.74	2.40	2.85
Std			0.21	0.84	1.53

Table 4.8: Speedups achieved by each enhancement over the GP column-column code, on a DEC Alpha 21064. The blocking parameters for SuperLU are $w = 8, t = 100$ and $b = 400$.

all (or most) of the source updating supernodes are likely to fit into cache, which means SupCol already achieves data reuse to some extent. This may be best illustrated by the results on the DEC Alpha 21164 (Table 4.9). For the six large matrices $18 - 23$, SuperLU achieves more than a factor of 2 speedup over SupCol. This machine differs from the others in that it has multilevel caches, with each cache having rather small capacity. This deep cache organization makes it easier for SupCol to experience cache thrashing than it is on a large flat cache. On the MIPS R8000, the large matrices achieve more than 4-fold speedup. This is partly due to better cache reuse, and partly due to better register reuse realized by DGEMV2 kernel.

With more and more sophisticated techniques introduced, the added complications in the code increase the the runtime overhead to some extent. This overhead can show up prominently in small or sparse problems. The two supernodal codes are particularly sensitive to the characteristic of the problems. This can be seen from the large standard deviations of their average speedups.

In practical applications, matrices are of varying size and sparsity, a natural question to ask is whether we can provide “black box” software that can choose the best algorithm based on characteristics of the matrix. This still remains a challenge in software engineering and deserves future investigation. The chief difficulty is that we cannot make the decision simply by looking at matrix itself. Take matrix 1 as an example. This matrix is large in dimension, and fairly sparse. More importantly, it remains sparse after factor-

	Matrix	GP (Seconds)	GP-Mod	SupCol	SuperLU
1	Memplus	0.17	1.25	1.01	0.45
2	Gemat11	0.13	1.54	1.26	0.84
3	Rdist1	0.80	1.76	1.77	1.45
4	Orani678	0.92	1.74	1.47	1.45
5	Mcfe	0.24	1.71	2.01	1.85
6	Lnsp3937	2.09	1.93	2.61	2.27
7	Lns3937	2.33	1.94	2.59	2.27
8	Sherman5	1.50	1.92	3.13	3.00
9	Jpwh991	1.06	2.14	3.20	3.20
10	Sherman3	3.65	2.10	4.06	3.93
11	Orsreg1	3.41	2.07	3.87	3.91
12	Saylr4	6.73	2.05	4.01	4.34
13	Shyy161	102.19	1.81	3.97	4.58
14	Goodwin	46.18	1.92	3.84	4.90
15	Venkat01	235.01	1.71	4.08	7.00
16	Inaccura	333.24	1.72	3.48	6.07
17	Bai	497.36	1.68	4.03	7.45
18	Dense1000	49.29	1.82	4.82	10.38
19	Raefsky3	1065.88	1.68	4.00	10.02
20	Ex11	1563.17	1.73	4.12	10.61
21	Wang3	1324.79	1.74	3.92	11.06
22	Raefsky4	2939.42	1.73	3.96	10.36
23	Vavasis3	9477.62	1.83	4.51	11.48
Mean			1.80	3.29	5.34
Std			0.20	1.10	3.69

Table 4.9: Speedups achieved by each enhancement over the GP column-column code, on a DEC Alpha 21164. The blocking parameters for SuperLU are $w = 16, t = 50$ and $b = 100$.

ization (only 1.4 fill factor, which is actually good for sparse code). There is no gain from introducing supernodes. Unfortunately, we know this only after the factorization.

4.9 Working storage requirement

In this section, we analyze the storage efficiency of the new panel algorithm. Apart from the data structures required to store the factored matrices L and U, a certain amount of working storage is also needed to facilitate the factorization process. Because of the inevitable fill-ins in direct factorization algorithms, memory is almost always at a premium. The resource limitation preventing the solution of large problems is often memory, not CPU hours. Therefore, a low working storage requirement is an important criterion to judge a solver's efficacy.

	Matrix	GP (Seconds)	GP-Mod	SupCol	SuperLU
1	Memplus	0.86	1.19	1.25	.75
2	Gemat11	0.57	1.32	1.71	1.09
3	Rdist1	3.77	1.64	1.65	1.58
4	Orani678	29.13	1.86	1.78	1.81
5	Mcfe	3.18	1.80	2.16	2.32
6	Lnsp3937	14.68	1.82	2.36	2.33
7	Lns3937	16.29	1.84	2.47	2.27
8	Sherman5	8.12	1.82	2.74	2.81
9	Jpwh991	5.74	1.85	2.39	2.58
10	Sherman3	16.04	1.90	3.19	3.09
11	Orsreg1	18.81	1.89	3.09	3.20
12	Saylr4	38.72	1.95	3.09	3.32
13	Shyy161	442.48	2.08	3.47	3.55
14	Goodwin	195.06	1.89	3.02	3.91
18	Dense1000	195.08	1.96	3.13	4.89
Mean			1.78	2.49	2.63
Std			0.24	0.68	1.09

Table 4.10: Speedups achieved by each enhancement over the GP column-column code, on a Sparc 20. The blocking parameters for SuperLU are $w = 8, t = 100$ and $b = 400$.

	Matrix	GP (Seconds)	GP-Mod	SupCol	SuperLU
1	Memplus	0.36	1.17	1.08	0.58
2	Gemat11	0.23	1.27	1.16	0.93
3	Rdist1	1.53	1.69	1.56	1.46
4	Orani678	1.86	1.64	1.25	1.33
5	Mcfe	0.52	1.97	1.85	1.92
6	Lnsp3937	4.26	1.86	2.16	2.24
7	Lns3937	4.89	1.94	2.11	2.33
8	Sherman5	3.15	1.94	2.28	3.03
9	Jpwh991	2.32	2.18	2.47	3.09
10	Sherman3	7.73	2.01	2.84	3.59
11	Orsreg1	7.2	1.97	2.69	3.52
12	Saylr4	13.88	1.96	2.52	3.84
13	Shyy161	188.72	1.91	3.01	3.43
14	Goodwin	89.30	1.89	2.62	4.41
18	Dense1000	94.77	2.05	3.33	4.25
Mean			1.83	2.19	2.66
Std			0.28	0.69	1.22

Table 4.11: Speedups achieved by each enhancement over the GP column-column code, on an UltraSparc-I. The blocking parameters for SuperLU are $w = 8, t = 100$ and $b = 400$.

In our supernode-panel factorization approach, the working storage is allocated during the factorization of a single panel, including both its outer and inner factorizations. The same working storage is then used repeatedly by factorizations across different panels. The working storage consists of two parts, where one part is used by symbolic factorization, and another part is used by numerical factorization. In symbolic factorization, five integer n-vectors are used in the column and panel depth-first search (Sections 4.3.1 and 4.3.4), where n is the order of the matrix. One integer n-vector is used to record the topological order obtained from the depth-first traversal. We use an n-by-w integer array to keep track of the position of the first nonzero of each supernodal segment in U, for all the columns in a panel of width w. An n-vector of integers is used as pointers pointing into the adjacency list of L, representing the pruned subgraph of L (Section 4.3.2). During the outer factorization, we use an n-by-w integer array to temporarily record the row indices of the nonzeros filled in the panel and below the U part. This is obtained by panel depth-first search, and is used immediately by the inner factorization. Thus, the total integer storage equals $n \times (7 + 2\ w)$.

In numerical factorization, an n-by-w floating-point SPA is used to allow random access to the entries in the active panel. Another floating-point temporary array is employed to store the results of BLAS calls. The size of this array is determined by the blocking parameters, and is calculated as $(t + b) \times w$, with t, b and w being illustrated in Figure 4.14. The total floating-point storage is $n \times w + (t + b) \times w$. Note that all the above working storage is reclaimed when the factorization is completed.

Table 4.12 reports the statistics on working storage usage. We compare the working storage requirement with the LU storage in two different ways. In the third column of the table, the total working storage divided by the number of bytes used by the factor matrix $F = L + U - I$ is shown. We include both integer and floating-point storage for F. We assume that an integer occupies 4 bytes, and a double precision floating-point number occupies 8 bytes. In the last column of the table only floating-point working storage is considered. Here, the floating-point working storage divided by the number of nonzeros in F is shown.

It can be seen from this table that the working storage requirement for smaller and sparser problems is relatively high, such as matrices 1 and 2. Since their L and U factors only occupy small amount of memory, memory is not a bottleneck for this type of problem. For large problems where L and U take up more than tens of Megabytes, the working storage usually represents only a few percent of the LU storage. Thus, being free from undue memory usage, the new algorithm is capable of solving the largest problems that can fit into core memory.

4.10 Supernodal triangular solves

Not only the factorization can benefit from supernodes; so can the triangular solution. Since our data structures and storage layouts for L and U are different, the lower and upper triangular solves are implemented differently. Here, we assume that the right-hand side vector is full.

Figure 4.21 shows the forward substitution procedure to solve a lower triangular system. Since L has supernode structures, both the triangular solve at line 3 and the matrix-

	Matrix	*LU* storage (MB)	Fraction of *LU* storage	Fraction of *LU* floats
1	Memplus	1.75	1.74	1.45
2	Gemat11	1.03	.82	.56
3	Rdist1	3.77	.19	.12
4	Orani678	3.34	.13	.09
5	Mcfe	0.73	.19	.12
6	Lnsp3937	4.59	.15	.09
7	Lns3937	4.83	.14	.08
8	Sherman5	2.65	.21	.13
9	Jpwh991	1.47	.12	.07
10	Sherman3	4.38	.19	.12
11	Orsreg1	4.04	.09	.06
12	Saylr4	6.60	.09	.05
13	Shyy161	78.04	.16	.10
14	Goodwin	33.96	.03	.02
15	Venkat01	129.83	.08	.05
16	Inaccura	105.18	.02	.02
17	Bai	142.61	.03	.02
18	Dense1000	9.85	.01	.01
19	Raefsky3	178.79	.02	.01
20	Ex11	275.50	.01	.01
21	Wang3	133.54	.03	.02
22	Raefsky4	267.59	.01	.01
23	Vavasis3	510.86	.01	.01

Table 4.12: Working storage requirement as compared with the storage needed for L and U. The blocking parameter settings are: $w = 8$, $t = 100$, and $b = 200$.

```
x = b;
for each supernode (r : s) in increasing order do
    x(r : s) = L(r : s, r : s)^-1 · x(r : s);
    x(s + 1 : n) = x(s + 1 : n) − L(s + 1 : n, r : s) · x(r : s);
end for;
```

Figure 4.21: Forward substitution to solve for x in $Lx = b$.

```
x = b;
for each supernode (r : s) in decreasing order do
    x(r : s) = U(r : s, r : s)^-1 · x(r : s);
    for j = r to s do
        x(1 : r − 1) = x(1 : r − 1) − x(j) · U(1 : r − 1, j);
    end for;
end for;
```

Figure 4.22: Back substitution to solve for x in $Ux = b$.

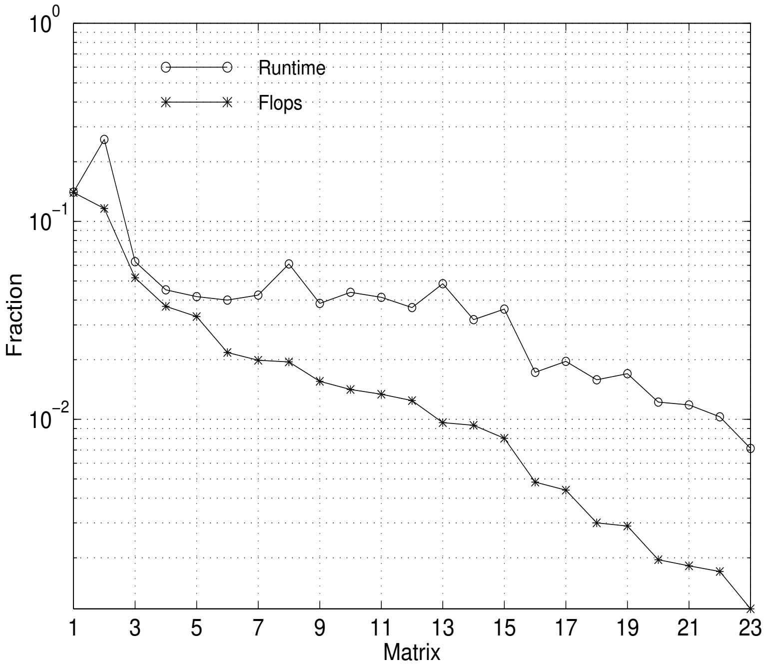


Figure 4.23: Fraction of the floating-point operations and runtime in the triangular solves over the LU factorization. Runtime is gathered from a RS/6000-590.

vector update at line 4 can call dense BLAS-2 routines. Moreover, if there are multiple right-hand side vectors, we can call the corresponding BLAS-3 routines, respectively.

Figure 4.22 shows the back substitution procedure to solve the upper triangular system $Ux = b$. Recall that diagonal blocks in U are stored together with the rectangular supernodes, so the triangular solve at line 3 can call a BLAS-2 routine. But since different columns of U usually have different structures, the update kernel at line 5 can only be a BLAS-1 operation.

Finally we note that at line 4 of Figure 4.21 and line 5 of Figure 4.22, we must first perform the respective dense operations in temporary arrays, then scatter the results into the destination vector x.

Figure 4.23 shows the fraction of the floating-point operations and the runtime of the solve phase as compared to the sequential LU factorization. Each is solved with only one right-hand side vector. Note that the percentage of the flops is usually lower than the percentage of the runtime. This is because each floating-point operation in the solve phase takes longer time than in the factorization.

4.11 Conclusions

Our starting point in this chapter was the supernode-column LU factorization algorithm developed by Eisenstat et al. [45]. Based on this, we designed both symbolic and numeric algorithms to perform the supernode-panel updates, in order to achieve better

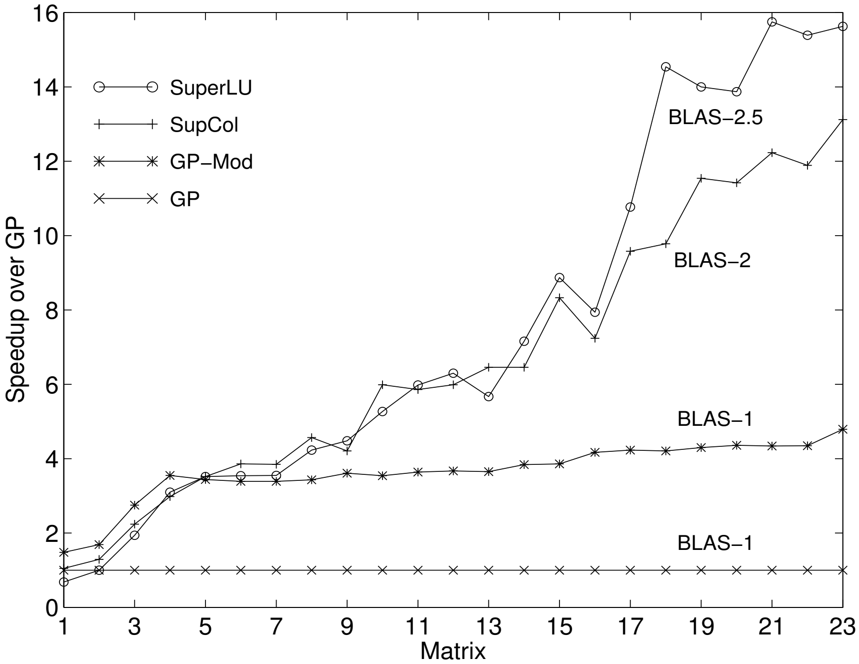


Figure 4.24: SuperLU speedup over previous codes on an IBM RS/6000-590.

data reuse. For the new code, SuperLU, we have conducted careful performance studies on several high performance machines. We studied both runtime and working storage efficiency.

Figures 4.24 [1] to 4.26 summarize, in graphical form, the improvement of SuperLU over the earlier codes on three cache-based superscalar machines. Each matrix has a "figure of merit": the ratio of floating-point operations to the (minmum possible) number of memory references. This figure limits the performance one can hope to achieve on a particular matrix. SuperLU delivers high performance matrices with high figures of merit. For large problems, SuperLU achieves more than 2-fold and 4-fold speedups over SupCol on the DEC Alpha 21164 and SGI MIPS R8000, respectively.

Figure 4.27 summarizes SuperLU factorization rate in flops-per-cycle on the three platforms. We give the respective peak flops-per-cycle figure in parentheses after each machine name. For large sparse matrices, we see that SuperLU achieves up to 40% of the peak floating-point rate on both RS/6000-590 and MIPS R8000. Given a dense matrix of size 1000-by-1000, SuperLU achieves roughly 70% of the efficiency of the dense *LU* factorization code implemented in LAPACK; this is the consequence of employing both symmetric reduction and the efficient "BLAS-2$\frac{1}{2}$" numeric kernel. The 30% efficiency loss is due to the time spent in symbolic factorization and indirect addressing, which cannot be eliminated in any sparse code. On the Alpha 21164, we achieve no more than 25% of the peak; this is somewhat disappointing. On the other hand, we note that DGEMM achieves only about 58% of the peak (Table 4.5). Table 4.13 gives a summary of the absolute factorization performance achieved by SuperLU on the three machines. For a problem from

[1]Figure 4.24 is identical to Figure 4.10

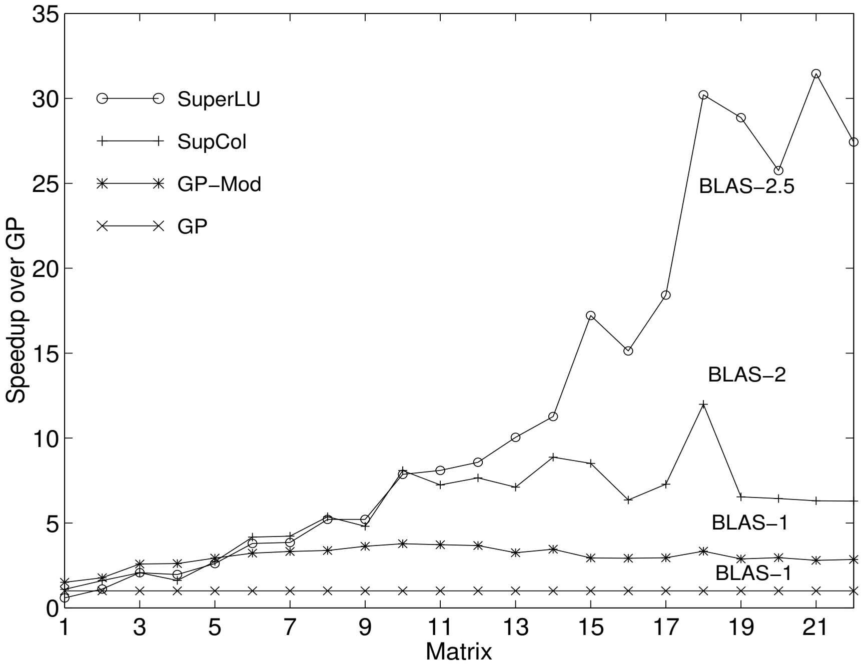


Figure 4.25: SuperLU speedup over previous codes on a MIPS R8000.

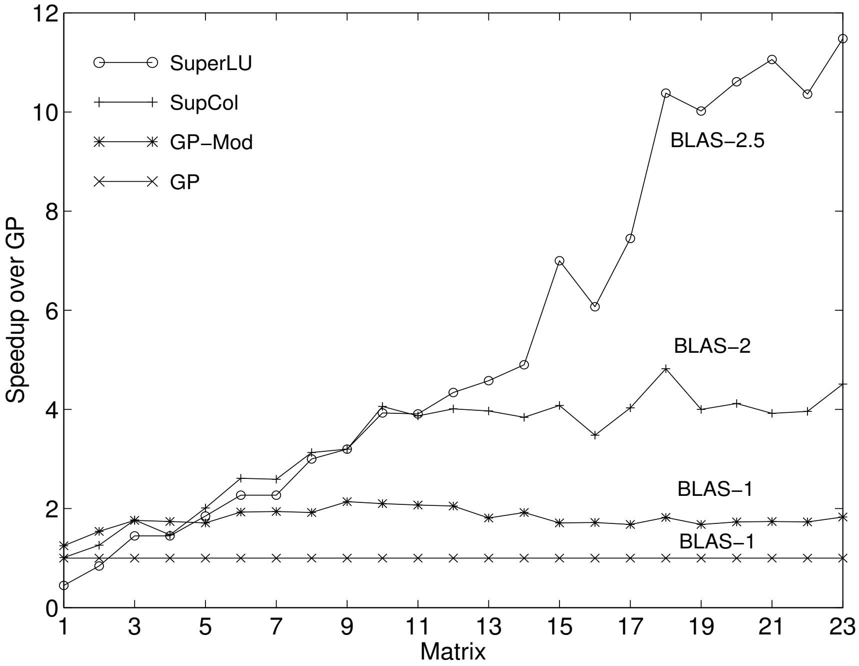


Figure 4.26: SuperLU speedup over previous codes on a DEC Alpha 21164.

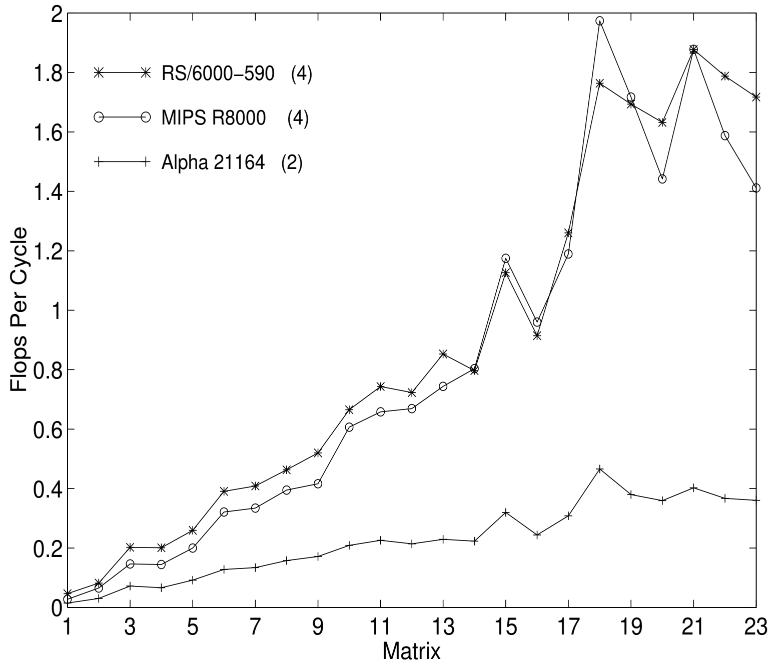


Figure 4.27: SuperLU factorization rate in flops/cycle, on the three platforms.

a 3-D semiconductor device simulation (matrix 21), the raw Mflop rates are 125 on the RS/6000-590, 169 on the MIPS R8000, and 121 on the Alpha 21164.

In addition to the *LU* factorization algorithm described in this chapter, we have developed a suite of supporting routines to solve general sparse linear systems. The complete SuperLU package includes condition number estimation, iterative refinement of solutions, and componentwise error bounds for the refined solutions [9]. These are all based on the dense matrix routines in LAPACK [8]. In addition, SuperLU includes a Matlab mex-file interface, so that our factor and solve routines can be called as alternatives to those built into Matlab.

We reported an earlier version of these results in a technical report [25].

	Matrix	RS/6000-590 Seconds	RS/6000-590 Mflops	MIPS R8000 Seconds	MIPS R8000 Mflops	Alpha 21164 Seconds	Alpha 21164 Mflops
1	MEMPLUS	0.57	3.08	0.71	2.47	0.38	4.58
2	GEMAT11	0.27	5.64	0.26	5.87	0.15	10.17
3	RDIST1	0.96	13.47	0.98	13.17	0.55	23.47
4	ORANI678	1.11	13.48	1.15	13.01	0.63	23.63
5	MCFE	0.24	17.42	0.23	17.99	0.13	31.04
6	LNSP3937	1.50	25.97	1.35	28.88	0.92	42.53
7	LNS3937	1.65	27.16	1.49	30.10	1.03	43.41
8	SHERMAN5	0.82	30.78	0.71	35.55	0.50	50.48
9	JPWH991	0.52	34.57	0.48	37.45	0.33	53.93
10	SHERMAN3	1.37	44.24	1.11	54.60	0.93	64.93
11	ORSREG1	1.21	49.42	1.01	59.23	0.87	69.02
12	SAYLR4	2.18	48.07	1.74	60.22	1.55	67.61
13	SHYY161	25.42	61.83	23.48	66.95	22.33	70.38
14	GOODWIN	12.55	52.99	9.20	72.30	9.43	70.51
15	VENKAT01	42.99	74.90	30.46	105.71	33.57	95.53
16	INACCURA	67.73	60.81	47.65	86.44	54.88	75.05
17	BAI	75.91	83.83	59.46	107.03	66.80	95.26
18	DENSE1000	5.68	117.28	3.75	177.64	4.75	139.75
19	RAEFSKY3	107.60	112.62	78.41	154.55	106.42	113.92
20	EX11	247.05	108.54	205.90	129.75	241.53	110.74
21	WANG3	116.58	124.86	86.14	168.99	119.77	121.48
22	RAEFSKY4	263.13	118.89	218.95	142.88	283.77	110.18
23	VAVASIS3	786.94	113.36	702.38	127.01	825.37	108.11

Table 4.13: Factorization time in seconds and rate in Mflops on the RS/6000-590, the MIPS R8000 and the Alpha 21164.

Chapter 5

A Parallel Supernode-Panel Algorithm

In this chapter we study an efficient parallel algorithm based on our left-looking blocking algorithm discussed in Chapter 4. The primary objective of this chapter is to achieve good efficiency on shared memory systems with a modest number of processors. Examples of such commercially popular machines include Sun SPARCcenter 2000 [107], SGI Power Challenge [104], DEC AlphaServer 8400 [46], and Cray C90/J90 [110, 111]. In addition to demonstrating the efficiency of our parallel algorithm on these machines, we also study the (theoretical) upper bound on performance of this algorithm.

Several methods have been proposed to perform sparse Cholesky factorization [49, 73, 90] and sparse *LU* factorization [6, 57, 65] on shared memory machines. A common practice is to organize the program as a self-scheduling loop, interacting with a global pool of tasks that are ready to be executed. Each processor repeatedly takes a task from the pool, executes it, and puts new ready task(s) in the pool. This pool-of-tasks approach has the merit of balancing work load automatically even for tasks with large variance in granularity. There is no notion of *ownership* of tasks or submatrices by processors – the assignment of tasks to processors is completely dynamic, depending on the execution speed of the individual processors. Our scheduling algorithm employs this model as well. Our parallel algorithm resembles the supernode-panel sparse Cholesky factorization studied in [73] in the way we define the basic tasks and the computational primitives. The way we handle the coordination of the dependent tasks is reminiscent of the approach used by Gilbert [65] in his column-wise sparse *LU* factorization. However, our algorithm represents a non-trivial extension to the earlier work in that we have incorporated several new mechanisms, such as unsymmetric supernodes and symmetric structure reduction.

We begin this chapter by looking at the architectural features and the programming environments of several shared memory machines which we use in our study. We focus on the features that are most relevant to the design and performance of our algorithm. In Section 5.2 we describe our basic strategies in parallelization, such as where we shall find parallelism in the problem, and how we shall define the individual tasks. Section 5.3 sketches the high-level parallel scheduling algorithm. Section 5.4 describes specific implementation details and shows the design choices we have made in different components of the algorithm, such as handling the dependent tasks, and memory management. In Section 5.5 we show the parallel performance (or *speedup*) achieved by the test matrices on various platforms. Both time and space efficiency will be illustrated. Finally, in Section 5.8 we establish a PRAM model to predict theoretical upper bound on speedups attainable by the underlying algorithm.

5.1 Shared memory machines

Many of the shared memory machines belong to the category of symmetric multiprocessing systems (SMP). By symmetric multiprocessing we mean that all processors in the system have the same computational power, and that they all have the same access latency to any location in the globally shared main memory. The processor-memory-I/O interconnect is often implemented using a shared bus, or some high speed switch. Main memory is usually configured in multiple logical units. There is no penalty for accessing a memory unit which is physically distant from the processor because all units are equidistant in an architectural sense. The provision of quick memory access is ensured by offering extensive interleaving. Individual DRAMs are incapable of providing data on a continuous basis. After each access, the chip must spend time recovering before permitting the next access. In interleaved memory schemes, the memory is subdivided into several independent memory banks and the addresses distributed across these banks. Memory performance is increased by arranging for one bank to supply data while other banks are recovering. So multiple memory components can operate in parallel.

The shared memory model often offers fine-grain, low latency access to remote data, which is a nice feature for our application. The major sources of overhead in shared memory programs are bus and memory contention due to sharing data. On modern cache-based SMP systems, variants of invalidation-based cache coherence protocols are often implemented in hardware. An update to a local copy of the shared block requires that every other copy must be either updated or invalidated. This may generate a lot of bus traffic. To maintain data integrity in globally shared data structures, it is necessary to serialize the concurrent accesses by different processors to a *critical section*, such as a segment of code that modifies a shared data structure. This mutually exclusive access is guaranteed by using locks on mutual exclusion variables (mutex variables). There are two types of performance problems associated with the mutual exclusion: (1) contention for a mutex variable because the critical region is too large; (2) overhead of lock acquisition even if no other processor is holding the lock. It is important to minimize the use of critical sections to obtain the best performance.

5.1.1 The Sun SPARCcenter 2000

Each processor in the Sun SPARCcenter 2000 [107] is a SuperSPARC microprocessor rated at 50 MHz. The processor is capable of executing up to three independent instructions per clock cycle. The on-chip cache consists of a 5-way set associative, 16 KB data cache, and a 5-way set associative, 20 KB instruction cache; both caches are physically addressed and operate in write-through mode. There is an external unified data and instruction cache associated with each processor, of size 1 MB, that is physically-addressed and direct-mapped. The external cache always operates in write-back mode. This integrated processor module can be easily upgraded with newer generation microprocessors to capture the most recent advances in processor technology. The parallel machine used in our study has 4 processors.

Dual high-speed packet-switched buses, called XDBuses, are used as interconnect. The packet-switched design permits split phase transactions of bus requests and their corresponding replies, and so enjoys higher bus utilization than circuit-switching. The dual buses provide 500 MB/sec effective data transfer bandwidth.

To access multiple processors, we use a user-level multithread library implemented in the Solaris 2.x operating system [106]. In this model, the lightweight user-level threads within a single UNIX process are multiplexed on top of kernel-supported threads. Synchronization and context switching of the user-level threads are accomplished rapidly, without entering the OS kernel.

5.1.2 The SGI Power Challenge

A 64-bit MIPS R8000 microprocessor and MIPS R8010 floating-point chip are used for each processor of the Power Challenge [104]. The chip set delivers peak performance of 360 MIPS and 360 double-precision Mflops with a clock frequency of 90 MHz. This processor was used in Chapter 4, see Table 4.5. Each processor contains a 16 KB direct-mapped level-one data cache in the integer unit (IU). This small on-chip cache allows fast access for integer loads and stores and helps the IU to accomplish fast integer and address calculations. A large 4 MB four-way set associative off-chip cache, called the data streaming cache, serves as a second-level cache for integer data and instructions, and as a first-level cache for floating-point data. Floating-point loads and stores bypass the on-chip cache and communicate with the large off-chip cache directly. The data streaming cache is pipelined to allow for continuous access by the floating-point functional units. Total cache bandwidth is 1.2 GB/sec, or two 64-bit double words per cycle. The cache line size is 512 bytes (64 double words). The parallel machine has 16 processors, and we use 12 processors in our experiments.

The memory subsystem consists of several memory modules. Each module is further divided into several banks. Memory is interleaved on cache line boundaries. The system used in our study has 2 GB of main memory, and is 4-way interleaved.

The multiprocessor system uses the POWERpath-2 interconnect. This bus structure provides cache-coherent communication between processors, main memory, and I/O. The address (40-bit) and data (256-bit) buses are separate. Read transactions are split: independent address and data transactions can occur simultaneously, creating a pipeline effect. A sustained transfer rate is 1.2 GB/sec, or two 64-bit double words per cycle.

The MIPS Power C compiler enables multiprocessing directives to ease parallel programming development. The system provides hardware support for fast synchronization operations, such as fork and join, semaphores and locks. These allow for efficient fine-grain parallel processing.

5.1.3 The DEC AlphaServer 8400

The DEC AlphaServer 8400 is based on the 64-bit Alpha 21164 microprocessor and the AlphaServer 8000-series platform architecture [46]. The clock frequency of the processor is 300 MHz with peak floating-point rate 600 Mflops. Each microprocessor has its own independent caches, including an 8 KB instruction cache, an 8 KB data cache, a 96 KB write-back second-level cache, and a 4 MB tertiary cache. This processor was used in Chapter 4, see Table 4.5. The parallel system used in our study has 8 processors.

Main memory is divided into multiple modules and supports between 2-way and 8-way interleaving. The system used in our study has a 4 GB of main memory.

The interconnect features separate address and data buses. With the emphasis on the low memory latency and the advantages of simple bused system, a wide (256-bit) and high frequency bus is used for the data path. The address bus supports a 40-bit address space. The system bus operates at 75 MHz which when applied to the 256-bit data path, produces a peak bandwidth of 2.4 GBytes/sec. However, a sustainable bandwidth is 1.6 GBytes/sec.

In the parallel program development, we use the pthread interface provided by DECthreads, Digital's multithreading run-time library [26]. The pthread interface implements a version of the POSIX 1003.1c API draft standard for multithreaded programming [91]; thus, the code will be easily portable to future systems. Similar to the Solaris threads model, multiple threads execute concurrently within (and share) a single address space. On the DEC, the multithreaded program is capable of utilizing multiple processors if the operating system supports kernel threads.

5.1.4 The Cray C90/J90

The Cray C90 [110] and J90 [111] are Cray Research's two series of vector supercomputers. The J90 series is the latest entry-level supercomputing system that is designed to address low price and high performance. Both systems have multiple processors, in which each processor is a vector machine. On each processor, high performance is achieved through *vectorization* – a version of the Single Instruction Multiple Data (SIMD) parallel processing technique. Unlike scalar processing, which requires a separate instruction cycle for each operation, vector processing requires only one instruction to carry out the same operation on an entire list of operands. The maximum number of processors for the C90 and J90 are 16 and 32, respectively.

In each processor of C90 and J90, the scalar chip is responsible for scalar processing and control for both the scalar and vector processors (VU chip). The scalar chip contains floating-point functional units, 32-word instruction buffers, and scalar and address registers. The VU chip contains the vector registers and vector (segmented) functional units. The vector registers are the operational registers for the vector operations. The vector registers can be loaded from memory, from the functional units, from other vector registers, or even from scalar registers. The segmented functional units divide an operation into distinct suboperations, each requiring one clock period to complete.

Central memory is highly interleaved. It is organized into eight sections for the 4 CPUs on a module. Each section is made up of eight subsections, which are further broken

Machine	Processor	CPUs	Bus Bandwidth	Read Latency	Memory Size	Programming Model
Sun	SuperSPARC	4	500 MB/s	1200 ns	196 MB	Solaris thread
SGI	MIPS R8000	16	1.2 GB/s	252 ns	2 GB	Parallel C
DEC	Alpha 21164	8	1.6 GB/s	260 ns	4 GB	pthread
Cray	C90	8	245.8 GB/s	96 ns	640 MB	microtasking
Cray	J90	16	51.2 GB/s	330 ns	640 MB	microtasking

Table 5.1: Characteristics of the parallel machines used in our study.

down into separate banks. There are altogether 1024 memory banks. Each word has 8 bytes, or 64 bits, of data. All integer values occupy a full 64-bit word. All floating-point values use Cray single-precision (64-bit) representation and 64-bit arithmetic hardware.

The clock speed of the C90 is 240 MHz. Each processor can produce four floating-point results per cycle, two adds and two multiplies, resulting in 960 Mflops peak on vector code.

The clock speed of the J90 is 100 MHz. The processor can produce two floating-point results per cycle, an add and a multiply, resulting in 200 Mflops peak on vector code. The peak scalar code performance is 100 MIPS. The vector register length is 64 words. Each scalar chip contains an 1 KB of 2-way, set-associative cache.

On both C90 and J90, the Cray C compiler provides a user-directed tasking (also called microtasking) capability to use multiple processors. In our C program, we insert the `taskloop` directive for the top level scheduling loop. The taskloop construct allows different iterations of a loop to be executed on different processors. Synchronization primitives are supported on both machines.

Table 5.1 summarizes the configurations and several key parameters of the five parallel systems. In the column "Bus Bandwidth" we report the effective or sustainable bandwidth. In "Read Latency" we report the minimum amount of time it takes a processor to fetch a piece of data from memory into a register in response to a load instruction.

5.2 Parallel strategies

Two crucial issues must be addressed in designing a parallel algorithm. One is to exploit as much concurrency as the problem presents to us. Another is to maintain a sufficient level of per-processor efficiency by choosing an appropriate granularity of task as a single scheduling unit. Care must be taken to strike a good balance between sustained amount of concurrency and per-processor performance. Before describing the detailed algorithm, we first address the above two issues in the context of our supernode-panel algorithm.

5.2.1 Parallelism

In dense linear algebra software, such as LAPACK [7], parallelism can simply rely on the parallel BLAS routines. So the sequential and shared memory parallel code are identical, except that the BLAS implementations differ. However, for sparse matrix factorizations, parallelism in the dense matrix kernels is quite limited, because the dense submatrices are typically small.

We can exploit two sources of parallelism in the sparse LU factorization. The coarse level parallelism comes from the sparsity of the matrix, and is exposed to us by the column elimination tree of A (see Section 3.3). Recall that each node in the elimination tree corresponds to one column in the matrix, so we will use "node" and "column" interchangeably.

In symmetric sparse Cholesky factorization, the elimination tree describes accurate column dependencies. A column will update its parent column and a subset of its ancestor columns along the path leading to the root. However, columns from two different subtrees never modify each other during the elimination process. This implies that the factorization of the independent subtrees can proceed concurrently. Almost all parallel sparse Cholesky factorization algorithms take advantage of this type of parallelism, referred to as tree or task parallelism.

In unsymmetric LU factorization with partial pivoting, we also wish to determine column dependencies prior to the factorization. It has been shown in a series of studies [50, 54, 63, 65] that the column elimination tree gives the information about all potential dependencies. We herein simply state the most relevant results. The interested reader can consult Gilbert and Ng [63] for a complete and rigorous treatment of this topic. Recall that column i of L and/or U modifies column j if and only if $u_{ij} \neq 0$. Part 3 of Theorem 1 implies that the columns in different subtrees do not update one another. Furthermore, the columns in independent subtrees can be computed without referring to any common memory, because the columns they depend on have completely disjoint row indices (Theorem 3.2 in [65]).

In general we cannot predict the nonzero structure of U precisely before the factorization, because the pivoting choice and hence the exact nonzero structure depend on numerical values. The column elimination tree can overestimate the true column dependencies. A typical example is

$$\mathrm{A} = \begin{pmatrix} 1 & \bullet & \bullet & \bullet \\ & 2 & & \\ & & 3 & \\ & & & 4 \end{pmatrix},$$

in which $A^T A$ is symbolically full, so the column elimination tree is a single chain. But regardless of the numeric values in the entries, matrix A has a trivial LU decomposition, and no column will update any other column. Despite the possible overestimate, part 4 of Theorem 1 says that if A is strong Hall, this dependency is the strongest information obtainable from the structure of A alone. (The example matrix is not strong Hall.) Therefore we will live with some pessimism in scheduling independent tasks. (In Section 5.2.2 we will be concrete about task definition.)

For a matrix that is not strong Hall, we might be able to improve the quality of the estimate by permuting the matrix to a block upper triangular form (called the *Dulmage-Mendelsohn decomposition*) [94], in which each diagonal block is strong Hall. Then, we only need to factorize the diagonal blocks.

Having studied the parallelism arising from different subtrees, we now turn our attention to the dependent columns, that is, the columns having ancestor-descendant relations. When the elimination process proceeds to a stage where there are more processors than independent subtrees, we need to make sure all processors effectively work on dependent columns. Thus the second level of parallelism comes from *pipelining* the computations of the dependent columns.

Consider a simple situation with only two processors. Processor 1 gets a task *Task 1* containing column j, processor 2 gets another task *Task 2* containing column k, and node j is a descendant of node k in the elimination tree. The (potential) dependency says only that *Task 2* cannot finish its execution before *Task 1* finishes. However, processor 2 can start *Task 2* right away with the computations not involving column j; this includes performing the symbolic structure prediction and accumulating the numeric updates using the finished columns that are descendants in the tree. After processor 2 has finished the other part of the computation, it has to wait for *Task 1* to finish. (If *Task 1* is already finished at this moment, processor 2 does not waste any time waiting.) Then processor 2 will predict the new fills and perform numeric updates that may result from the finished columns in *Task 1*. In this way, both processors do useful work concurrently while still preserving the precedence constraint. Note that we assume the updates can be done in any order. This could give different numerical result and is therefore not a straightforward parallelization of the sequential algorithm.

The pipelining scheme above can be generalized to an arbitrary number of processors. When a processor obtains a panel, it uses appropriate data structures to keep track of the currently unfinished descendants of this panel. The processor first performs updates from the computed descendants, waits for the others to finish, and finally performs any updates that may come from the just-finished descendants. Although this pipelining mechanism is complicated to implement, it is essential to achieve higher concurrency. This is because, in most problems, a large percentage of the computation occurs at a few top levels of the etree, where there are fewer branches than processors. An extreme example is a dense matrix, the elimination tree of which is a single chain. Here, all parallelism must come from pipelining.

5.2.2 Panel tasks

As studied in Chapter 4, the introduction of *supernodes* and *panels* makes the computational kernels highly efficient. Recall that a panel differs from a supernode in that we do not require the row structures of its constituent columns to be the same (although the more similar, the better opportunity for data reuse). We may view supernodes as blocks intrinsic to the problem, whereas panels arise from algorithmic blocking. The size of each panel can be set before factorization, but supernode boundaries must be identified dynamically. To retain the serial algorithm's ability to reuse data in cache and registers, we treat the factorization of one panel as a unit task to be scheduled; it computes the part of U and the part of L for all columns within this panel. More specifically, a panel task comprises two distinct subtasks. The first corresponds to the outer factorization, which accumulates the updates from the descendant supernodes. The second subtask is to perform the panel's inner factorization, which factors one column at a time, including supernode

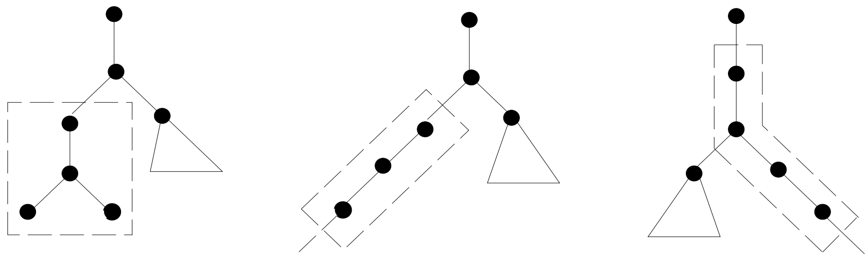


Figure 5.1: Panel definition. (a) relaxed supernodes at the bottom of the elimination tree; (b) consecutive columns from part of one branch of the elimination tree; (c) consecutive columns from more than one branch of the elimination tree.

detection, partial pivoting, and symmetric pruning. For this simple algorithm, we do not exploit potential parallelism within a panel factorization.

A panel consists simply of a set of consecutive columns in the matrix. Since the parallel algorithm uses the column elimination tree as the main scheduling tool, it is worth studying the relationship between the panels and the structure of the column elimination tree. We assume that the columns of the matrix are ordered according to a postorder on the elimination tree. Recall that the sequential algorithm takes the panel size w as an input parameter. It tries to factorize w consecutive columns at a time. There might exist panels of size smaller than w. This happens whenever two columns j and $j+1$ come from different subtrees. Then there is no benefit from grouping j and $j+1$ into one panel, because their respective sets of updating supernodes are disjoint. In this case, column $j+1$ will start a different panel. Pictorially, panels can be classified into three types, depending on where they are located in the elimination tree, as illustrated in Figure 5.1.

In the parallel algorithm, panels of type (a) and (b) are easy to handle. To deal with type (c) panels, the pipelining scheme requires complex data structures to keep track of the busy descendant panels, because each panel may contain columns from different branches of the tree. We need to identify all the frontier busy columns hanging off the different subtrees. We realize that the cost of this book-keeping and the cost associated with the complicated control logic would be enormous.

To simplify this matter, we have redefined the panels so that type (c) panels do not occur. We will let a panel stop before a node (column) that has more than one branch in the elimination tree. Every branching node necessarily starts a new panel. Under this restriction, the busy descendant panels, except type (a) panels, always form one path in the elimination tree. If a processor needs to wait for, and later perform, the updates from the busy panels, it can simply walk up the path in the etree starting from the most distant busy descendant(s). By this new definition of panels, there will be more panels of smaller sizes. The question arises whether this will hurt performance. We studied the distribution of floating-point operations on different panel sizes for all of our test matrices, and observed that usually more than 95% of the floating-point operations are performed in the panels of largest size, and these panels tend to occur at a few topmost levels of the elimination tree. Thus, panels of small sizes normally do not represent much computation. On unipro-

```
Slave_worker()
newp = NULL;
while ( there are more panels ) do
    oldp = newp;
    Scheduler( oldp, newp, Q );
    if ( newp is a relaxed supernode ) then
        relaxed_supernode_factor( newp );
    else
        panel_symbolic_factor( newp );
            – determine which supernodes will update panel newp;
            – skip all BUSY panels/supernodes;
        panel_numeric_factor( newp );
            – accumulate updates from the DONE supernodes, updating newp;
            – wait for the BUSY supernodes to become DONE, then predict
              new fills and accumulate more updates to newp;
        inner_factorization( newp ); /* independent from other processors */
            – perform supernode-column update within the panel;
            – perform row pivoting;
            – detect supernode boundary;
            – perform symmetric structure pruning;
    end if;
end while;
```

Figure 5.2: The parallel scheduling loop to be executed on each processor.

cessors, we see almost identical performance using the earlier and the new definitions of panels. Therefore, we believe that this restriction on panels simplies the parallel scheduling algorithm with no performance compromise on individual processors.

5.3 The asynchronous scheduling algorithm

Having described the parallelism and basic computational tasks, we are now in a position to describe the parallel factorization algorithm. This section presents the organization of the scheduling algorithm, with more implementation details to appear in Section 5.4. Our scheduling approach used some techniques from the parallel algorithm developed by Gilbert [65], which was based on the sequential GP algorithm. Figure 5.2 sketches the top level scheduling loop. Each processor executes this loop until its termination criterion is met, that is, all panels have been factorized.

The parallel algorithm maintains a central priority queue of tasks (panels), denoted by Q, that are ready to be executed by any free processor. The content of this task queue can be accessible and altered by any processor. At any moment during the elimination, a panel is tagged with a certain state, such as READY, BUSY, or DONE. Every processor repeatedly asks the scheduler (at line 4) for a panel task in the queue. The `Scheduler()` routine implements a priority-based scheduling policy described below. The input argument *oldp* denotes the panel that was just finished by this processor. The output argument *newp* is a newly selected panel to be factorized by this processor. The selection preference is as follows:

(1) The scheduler first checks whether all the children of *oldp*'s parent panel, say *parent*, are DONE. If so, *parent* now becomes a new leaf and is immediately assigned to *newp* on the same processor.

(2) If *parent* still has unfinished children, the scheduler next attempts to take from Q a panel which can be computed without pipelining, that is, an initial leaf panel.

(3) If no more leaf panels exist, the scheduler will take a panel that has some BUSY descendant panels currently being worked by other processors. Then the new panel must be computed by this processor in a pipelined fashion.

One may argue that (1) and (2) should be reversed in priority. Choosing to eliminate the immediately available parent first is primarily concerned with locality of reference. Since a just-finished panel is likely to update its parent or other ancestors in the etree, it is advantageous to schedule its parent and other ancestors on the same processor.

To implement the above priority scheme, the task queue Q is initialized with the leaf panels, that is, the relaxed supernodes, which are marked as READY. Later on, `Scheduler()` may add more panels at the tail of Q. This happens when all the children of *newp*'s parent, *parent*, are BUSY; *parent* is then enqueued into Q and is marked as eligible for pipelining. By rule (1), some panel in the middle of the queue may be taken when all its children are DONE. This may happen even before all the initial leaf panels are finished. All the intermediate leaf panels are taken in this way. By rule (2) and (3), `Scheduler()` removes tasks from the head of Q.

It is worth noting that the executions of different processors are completely asynchronous. There is no global barrier; the only synchronization occurs at line 13 in Figure 5.2, where a processor stalls when it waits for some BUSY updating supernode to finish. As soon as this BUSY supernode is finished, all the processors waiting on this supernode are awakened to proceed. This type of synchronization is commonly referred to as *event notification*. Since the newly finished supernode may produce new fills to the waiting panels, the symbolic mechanism is required to discover and accommodate these new fills.

We use the SPMD (Single Program Multiple Data) parallel programming style, in which a single program text is executed by all processors. At the program level, multiple concurrent (logical) *threads* are created for the scheduling loop `Slave_worker()`. Scheduling these threads on available physical processors is done by the operating system or runtime library. Thread migration between processors is usually invisible to us.

5.4 Implementation details

In this section we present assorted implementation details. The details in this section are not required for understanding later sections of this thesis.

5.4.1 Linear pipelining

In the most general pipeline approach, a panel, say p, can begin execution as soon as all its children are either DONE or BUSY. To implement this scheme, we must record all the busy descendants, possibly from many different branches down the etree. This bookkeeping can be very expensive. Instead, we have implemented a simpler pipeline mechanism, in which a panel p is allowed to start pipeline execution only if it has exactly one busy child. Under this rule, the busy panels must always form a single path in the etree. We therefore call this *linear pipelining*. Let d be the first (lowest numbered) panel in this busy chain. That is, all the children of d are finished but d is still busy. Then the processor working on p simply walks up the etree from d, waits for all the columns between d and p to finish, and accumulates new updates from those columns along the path. The only bookkeeping required by each processor is to record d, the most distant busy panel in this linear chain.

5.4.2 Symmetric pruning

Symmetric pruning [43, 44] was discussed in Section 4.3.2 for the sequential algorithm. The idea is to use a graph H with fewer edges than $G(L^T)$ to represent the structure of L. Traversing H gives the same reachable set as does traversing G, but is less expensive. As shown in Section 4.5, this technique is very effective in reducing the symbolic factorization time. Therefore, we want to retain this technique in the parallel algorithm.

In the sequential algorithm, in addition to the adjacency structure for G, there is another adjacency structure to represent the reduced graph H. For each supernode, since the row indices are the same among the columns, we only store the row indices of the first column for G and the row indices of the last column for H. (If we use only one adjacency list for each supernode, since pivoting may have reordered the rows so that the pruned and unpruned rows are intermingled in the original row order, it is then necessary to reorder all of L and A to account for it.) Figure 5.3 illustrates the storage layout for the adjacency lists of G and H of a sample matrix (also see Figures 3.3 and 3.5). Array `Lsub[*]` stores the row subscripts. `G_ptr[*]` points to the beginning of each supernode in array `Lsub[*]`. `H_ptr[*]` points to the pruned location of each supernode in array `Lsub[*]`. Using `G_ptr` and `H_ptr` together can locate the adjacency list for each supernode in H. This matrix has four supernodes: {1,2}, {3}, {4,5,6}, and {7,8,9,10}. The adjacency lists for G and H are interleaved by supernodes in the global memory `Lsub[*]`. For a singleton supernode, such as {3}, only one adjacency list is used for both G and H. The storage for the adjacency structure of H is reclaimed at the end of the factorization.

The pruning procedure works on the adjacency lists for H. Each adjacency list of a supernode (actually only the last column in the supernode) is pruned at the position of the first symmetric nonzero pair in the factored matrix F, as indicated by the small "$\cdot$" in the figure. Both column DFS (Section 4.3.1) and panel DFS (Section 4.3.4) traverse the adjacency structure of H, as given by `H_ptr[*]` in Figure 5.3.

In the parallel algorithm, contention occurs when one processor is performing DFS using H's adjacency list of column j (a READ operation), while another processor is pruning the structure of column j, since pruning will reorder the row indices in the list (a MODIFY

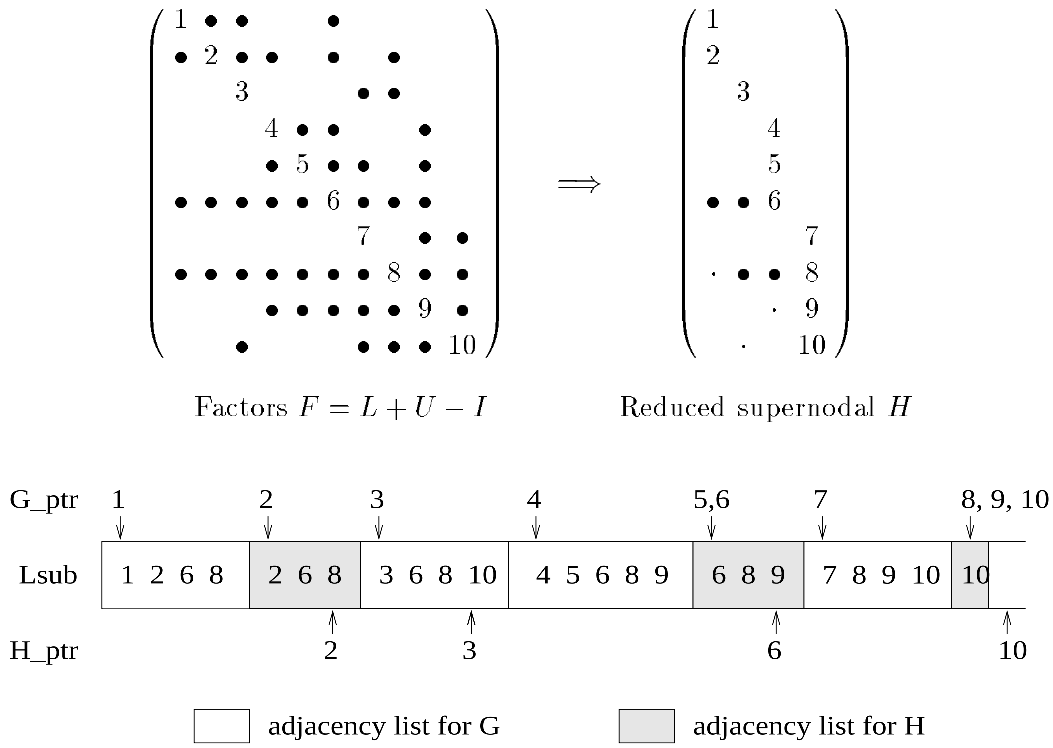


Figure 5.3: Storage layout for the adjacency structures of G and H.

operation). There are two possible solutions to avoid this contention. The first solution is to associate one mutex lock with each adjacency list of H. A processor acquires the lock before it prunes the list and releases the lock thereafter. Similarly, a processor uses the lock when performing DFS on the list. Although the critical section for pruning can be very short, the critical section for DFS may be very long, because the list must be locked until the entire depth-first search starting from all nodes in the list is completed. During this period, all the other processors attempting to prune the list or to traverse the list will be blocked. Therefore this approach may incur too much overhead, and the benefit of pruning may be completely offset by the cost of locking.

We now describe a better algorithm that is free from locking. We will use both graphs H and G to facilitate the depth-first search. Recall that each adjacency list is pruned only *once* throughout the factorization. We will associate with each list a status bit indicating whether it is pruned or not. Once a list is pruned, all the subsequent traversals on the list involve only READ operations, and hence do not require locking. If the search procedure reaches a list of H that is not yet been pruned, we will direct the search procedure to traverse the list of the corresponding column in G. So when the search algorithm reaches column j, it does the following:

```
if column j is pruned then
    continue search from nodes in the H-list of column j;
else
    continue search from nodes in the G-list of column j;
endif
```

In order for this scheme to work, we need to maintain two copies of an identical list for each singleton supernode. This incurs a little more working storage requirement than the sequential algorithm.

Since H is generally a subgraph of G, the depth-first search algorithms in the parallel code may traverse more edges than those in the sequential code. This is because in the parallel algorithm, a supernode may be pruned later than in the sequential algorithm. However, because of the effectiveness of symmetric reduction, very often the search still uses the pruned list in H. So it is likely that the time spent in the slight extra search in the G-lists is much less than that when using the locking mechanism. Figure 5.4 shows the relative size of the reduced supernodal graph H, and Figure 5.5 shows the fraction of the number of searches that use the H-lists. The numbers in both figures are collected on a single processor Alpha 21164.

5.4.3 Supernode storage using nonzero column counts in QR factorization

Recall that we have used two blocking structures in our algorithm, which are panels and supernodes. A panel differs from a supernode in that we do not require the row structures of its constituent columns to be the same (although the more similar the better the chance of data reuse). We may view supernodes as blocks intrinsic to the problem, whereas panels arise from algorithmic blocking.

The reader may refer to Section 3.2 for our definition of T2 supernodes and the storage scheme used to store supernodes in memory. It is important to store the columns

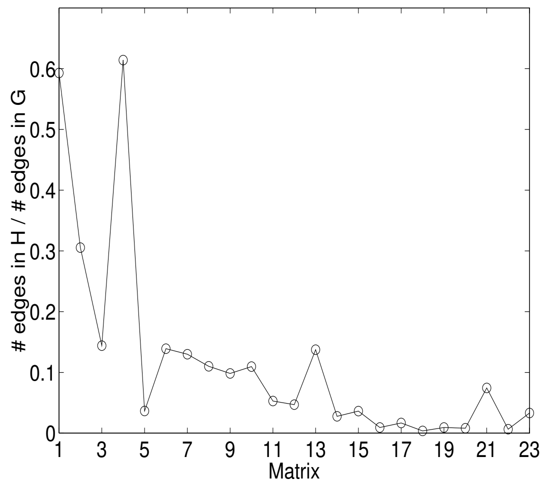


Figure 5.4: Number of edges in H versus number of edges in G.

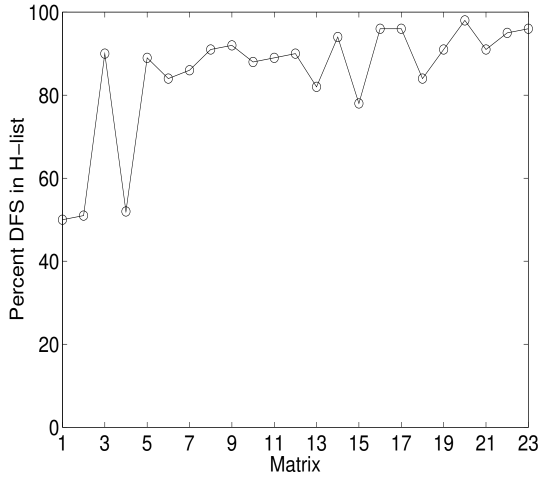


Figure 5.5: Percent of the depth-first search on adjacency lists in H.

of a supernode *consecutively* in memory, so that we can call BLAS routines directly in-place without paying the cost of copying the columns into contiguous memory. Although this contiguity is easy to achieve in a sequential code, it poses problems in the parallel algorithm.

Consider the scenario of parallel execution depicted in Figure 5.6. According to the order of the finishing times specified in the figure, panel {3,4} will be stored in memory first, followed by panel {1,2}, and then followed by panel {5,6}. The supernode {3,4,5,6} is thus separated by the panel {1,2} in memory. The major difficulty comes from the fact that the supernodal structure emerges dynamically as the factorization proceeds, so we cannot statically calculate the amount of storage required by each supernode. Another difficulty is that panels and supernodes can overlap in many different ways.

One immediate solution is not to allow any supernode to cross the boundary of a panel. In other words, the leading column of a panel is always treated as the beginning of a new supernode. Thus a panel can possibly be subdivided into more than one supernode, but not vice versa. In such circumstances, the columns of a supernode can be easily guaranteed to be contiguous in memory because they are part of a panel and assigned to a single processor by the scheduler. Each processor simply stores a (partial) ongoing supernode in its local temporary store, and copies the whole supernode into the global data structure as soon as it is finished.

This restricted definition of supernodes would mean that the maximum size of supernodes would be bounded by the panel size. As discussed in Chapter 4, for best performance, we would like to have large supernodes but relatively small panels. These conflicting demands make it hard to find one good size for both supernodes and panels. We conducted an experiment with this scheme for the sequential algorithm. Figure 5.7 shows the uniprocessor performance loss with varying panel size (i.e., the maximum size of supernodes). For large matrices, say matrices $12-21$, the smaller panels and supernodes result in more performance loss. For example, when panel size is $w = 16$, the slowdown can

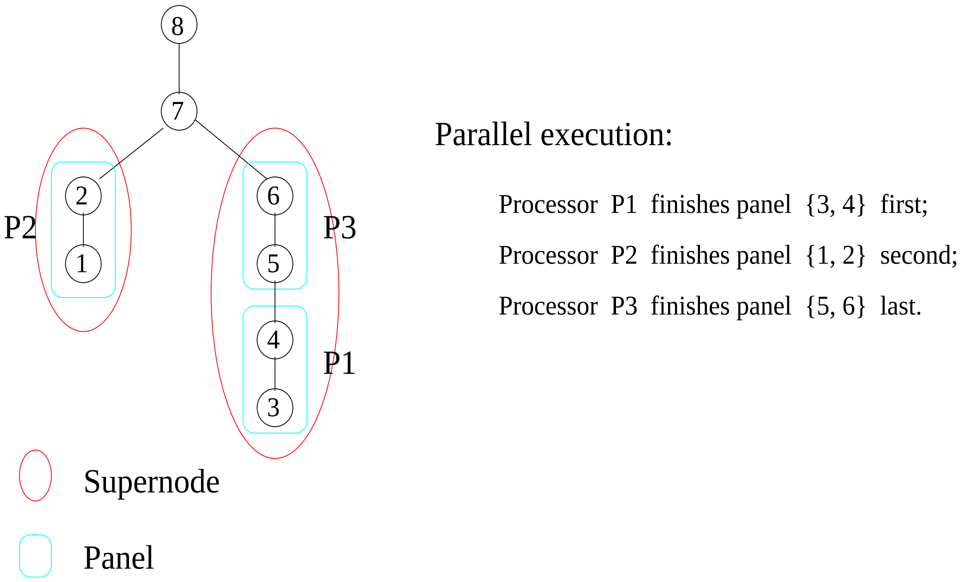


Figure 5.6: A snapshot of parallel execution.

be as large as 20% to 68%. Even for large panel sizes, such as $w = 48$, the slowdown is still between 5% and 20%. However, in the parallel algorithm, such large panels give rise to too large a task granularity and severely limit the level of concurrency in the parallel algorithm. We therefore feel that this simple solution is not satisfactory. Instead, we seek a solution that does not impose any restriction on the relation between panels and supernodes, and that allows us to vary the size of panels and supernodes independently in order to better trade off concurrency and single-processor efficiency.

Our second proposed solution is to allocate space that is an upper bound on the actual storage needed by each supernode in the L factor, *irrespective of the numerical pivoting choice*. Then there will always be space to store supernode columns as they are computed. Note that after Gaussian elimination with partial pivoting, we can write $A = P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} U$. We define L as the unit lower triangular matrix whose i-th column is the i-th column of L_i, such that $L - I = \sum_i (L_i - I)$.[1] Let L_c be the result of forming $A^T A$ symbolically and then performing symbolic Cholesky factorization. (i.e., in the absence of conincidental numerical cancellation.) We shall make use of the following structure containment properties in our storage scheme. Here we only quote the results without proof.

Theorem 5 [54] *Let A be a nonsingular matrix with nonzero diagonal. If L_c is the symbolic Cholesky factor described above, and L and U are the triangular factors of A represented as above, then $Struct(L+U) \subseteq Struct(L_c + L_c^T)$.*

Theorem 6 [50, 55] *Consider the QR factorization $A = QR$ using Householder transformations. Let H be the symbolic Householder matrix consisting of the sequence of House-*

[1]This L is different from the $\hat{L}$ in $PA = \hat{L}U$. Both L and $\hat{L}$ contain the same nonzero values, but in different positions. In this section, L is used as a data structure for storing $\hat{L}$.

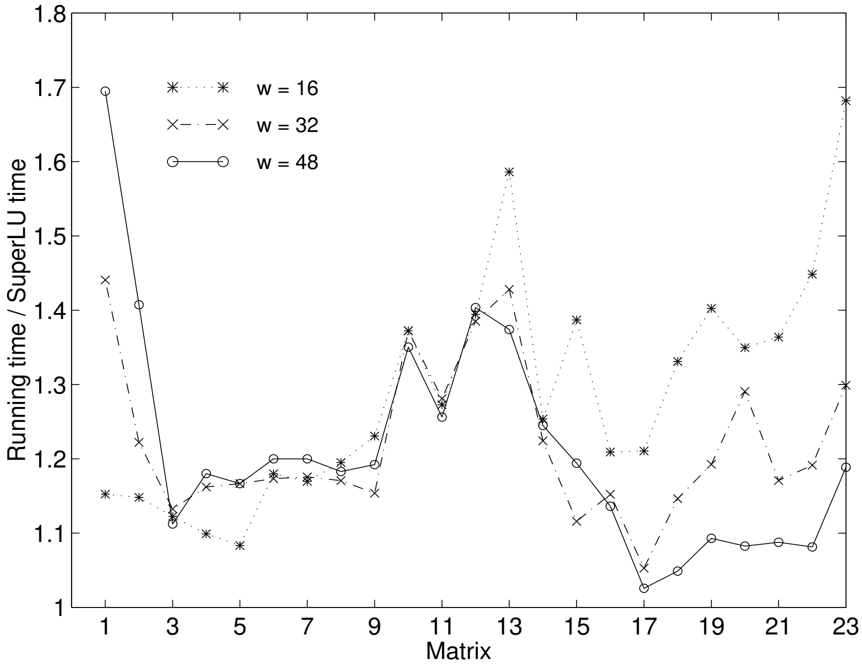


Figure 5.7: The sequential runtime penalty for requiring that a leading column of a panel also starts a new supernode. The times are measured on the RS/6000-590.

holder vectors as used to represent the factored form of Q. If A is a nonsingular matrix with nonzero diagonal, and L and U are the triangular factors of A represented as above, then $Struct(L) \subseteq Struct(H)$, and $Struct(U) \subseteq Struct(R)$.

Theorem 7 [20] *Suppose A has full (column) rank. If L_c is the symbolic factor described above, then $Struct(R^T) \subseteq Struct(L_c)$. Furthermore, if A is strong Hall, then at most an arbitrarily small perturbation in the nonzero values of A is needed to achieve $Struct(R^T) = Struct(L_c)$.*

In what follows, we describe how these upper bounds can facilitate our storage management for the L supernodes. First, we need a notion of *fundamental* supernode, which was introduced by Ashcraft and Grimes [11] for symmetric matrices. In a fundamental supernode, every column except the last is an only child in the elimination tree. Liu et al. [84] gave several reasons why fundamental supernodes are appropriate, one of which is that the set of fundamental supernodes are the same regardless of the particular postordering. For consistency, we now also impose this restriction on the supernodes in L,[2] L_c and H, respectively. For convenience, let S_L denote the fundamental supernodes in the L factor, S_{L_c} denote the fundamental supernodes in the symbolic Cholesky factor L_c, and S_H denote the fundamental supernodes in the symbolic Householder matrix H. In the following, we shall omit the word "fundamental" when it is clear.

[2]The sequential SuperLU does not have this restriction on supernodes.

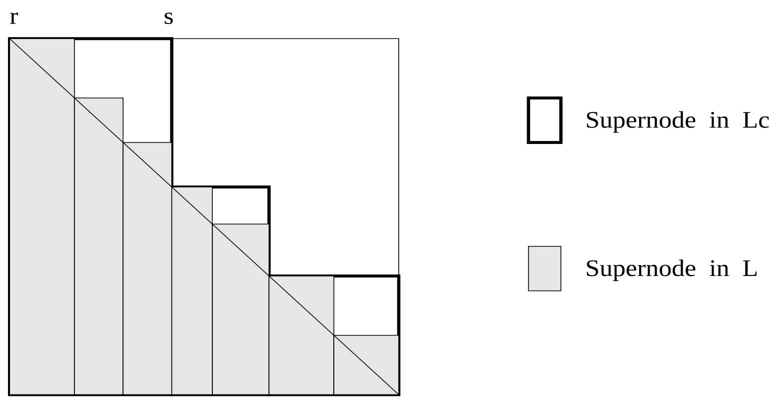


Figure 5.8: Bound the L supernode storage using the supernodes in L_c.

Our code breaks the L supernode at the boundary of an L_c (or H) supernode, forcing the L supernode to be contained in the L_c (or H) supernode. In fact, if we use fundamental L supernodes and ignore numerical cancellation (which we must do anyway for symmetric pruning), we can show that an L supernode is always contained in an L_c (or H) supernode [69]. Our objective is to allocate storage based on number of nonzeros in either S_{L_c} or S_H, so that this storage is sufficiently large to hold S_L. Figure 5.8 illustrates the idea of using S_{L_c} as a bound. Two supernodes in S_L from different branches of the elimination tree will go to their corresponding memory locations of the enclosing supernodes in S_{L_c}. For those S_L supernodes occurred in the same S_{L_c} supernode, even if their panels are assigned to different processors, the scheduling algorithm guarantees that the panels (and hence the supernodes) are finished in the order of increasing column numbers. So the columns of each S_L supernode are contiguous in the storage of the S_{L_c} supernode.

To determine the storage for S_{L_c}, what we need is an efficient algorithm to compute the column counts $nnz(L_{c_{*j}})$ for L_c. We also need to identify the first vertex of each supernode in S_{L_c}. Then the number of nonzeros in each supernode is simply the product of the column count of the first vertex and the number of columns in the supernode. To compute $nnz(L_{c_{*j}})$ and S_{L_c}, we can apply the supernodal count algorithm for sparse Cholesky factor [70] to A^TA. However, forming the structure of A^TA may be expensive and A^TA may be much denser than A. To achieve the needed level of efficiency, Gilbert, Ng and Peyton [69] suggested ways to modify their Cholesky-column-count algorithm [70] to work with the structure of A without explicitly forming A^TA. The running time of this algorithm is $O(m\ \alpha(m,n))$, where $m = nnz(A)$ and $\alpha(m,n)$ is the slowly-growing inverse of Ackermann's function coming from disjoint set union operations.

For H, we need the following crucial result by George, Liu and Ng ([50], Theorem 2.1): Each row set $Struct(H_{i*})$ consists of all the vertices on a path in the column elimination tree from f_i to the smaller of i or the root of the elimination subtree containing f_i, where f_i is the column subscript of the first nonzero in row i of A. Thus the column counts $nnz(H_{*j})$ can be obtained using a simple variant of Cholesky-column-count algorithm, with time complexity $O(nnz(A))$. It can be easily incorporated into the column count algorithm for L_c. Furthermore, the first vertices of the fundamental supernodes in H are characterized by the following theorem, which we established through discussions with Ng [88].

Theorem 8 *Vertex j is the first vertex in a fundamental supernode of H if and only if vertex j has two or more children in the column elimination tree T, or j is the column subscript of the first nonzero in some row of A.*

Proof: It is clear by definition that if vertex j has two or more children in T it must be the first node of a fundamental supernode. Therefore we only need to prove the second case in which vertex j has exactly one child $j-1$.

"if" part: Let a_{ij} be the first nonzero in row i of A. Then we must have $h_{i,j-1} = 0$, for otherwise, there exists a path in T associated with $Struct(H_{*i})$ from some k $(\leq j-1)$ to i, and $a_{ik} \neq 0$. This leads to the contradiction that $a_{ik} = 0,$ for all $k < j$. It follows that $Struct(H_{*j}) \not\subseteq Struct(H_{*,j-1})$; thus j must start a new supernode.

"only if" part: Assume that node j is the first node of its fundamental supernode, implying that $Struct(H_{*j}) \neq Struct(H_{*,j-1}) - \{j-1\}$. Then there exists a row i such that $h_{i,j-1} = 0$ and $h_{ij} \neq 0$. If there is an $a_{ik} \neq 0$ with $k \leq j-1$, then $h_{i,j-1}$ cannot be zero, because $j-1$ is on the path from k to i in T. This leads to a contradiction. Therefore we must have $a_{ik} \neq 0$ for all $k \leq j-1$, that is, a_{ij} is the first nonzero in row i of A. □

Finding the first nonzeros in each row (hence S_H) takes time $O(nnz(A))$. In summary, the combined QR-column-count algorithm takes $Struct(A)$ and the postordered T as inputs, and computes $nnz(L_{c_{*j}})$, S_{L_c}, $nnz(H_{*j})$ and S_H. The complexity of the algorithm is almost linear in $nnz(A)$. In practice, it is as fast as computing the column elimination tree T. Table 5.2 reports the respective runtimes of the etree algorithm and the QR-column-count algorithm. In both the etree and QR-column-count algorithms, the disjoint set union operations are implemented using path halving and no union by rank. (see Gilbert et al. [70] for details.)

One remaining issue yet to be addressed is what we should do if the static storage given by an upper bound structure is far too generous than actually needed. We developed a dynamic approach to better capture the structural changes of the LU factors during Gaussian elimination. It overcomes the inefficiency of the purely static analysis based solely on $Struct(A)$. In this scheme, we still use either the supernode partition S_{L_c} or S_H as found in their respective upper bound structure. For brevity, we will use B to represent either L_c or H, and use the notation S_{bnd} to denote either S_{L_c} or S_H, because the method works the same way using either S_{L_c} or S_H. Unlike the static scheme, which uses the column counts $nnz(L_{c_{*j}})$ or $nnz(H_{*j})$, we dynamically compute the column count for the first column of each supernode as follows. When a processor obtains a panel that includes the first column of some supernode $B(:, r:s)$ in S_{bnd}, the processor invokes a search procedure on the directed graph $G(L(:, 1:r-1)^T)$, using the nonzeros in $A(:, r:s)$, to determine the union of the row structures in the submatrix $(r:n, r:s)$. We use the notation $D(r:n, r:s)$ to denote this structure. It is true that

$$Struct((L+U)(r:n, r:s)) \subseteq Struct(D(r:n, r:s)) \subseteq Struct(B(r:n, r:s)) \,. \qquad (5.1)$$

The search procedure is analogous to (yet simpler than) the panel DFS described in Section 4.3.4; now we only want to determine the count for the column $D(r:n, r)$, without the nonzero structure or the topological order of the updates. Then we use the product

	Matrix	T_{etree}	T_{cnt}	T_{cnt}/T_{etree}
1	Memplus	0.070	0.101	1.4
2	Gemat11	0.023	0.029	1.3
3	Rdist1	0.059	0.056	0.9
4	Orani678	0.015	0.015	1.0
5	Mcfe	0.014	0.018	1.3
6	Lnsp3937	0.018	0.025	1.4
7	Lns3937	0.017	0.025	1.4
8	Sherman5	0.014	0.021	1.5
9	Jpwh991	0.004	0.006	1.4
10	Sherman3	0.057	0.078	1.4
11	Orsreg1	0.010	0.013	1.4
12	Saylr4	0.015	0.022	1.4
13	Shyy161	0.241	0.371	1.5
14	Goodwin	1.081	1.003	0.9
15	Venkat01	0.202	0.176	0.9
16	Inaccura	0.629	0.547	0.9
17	Bai	0.300	0.380	1.3
18	Dense1000	0.610	0.430	0.7
19	Raefsky3	0.945	0.735	0.8
20	Ex11	0.680	0.640	0.9
21	Wang3	0.124	0.169	1.4
22	Raefsky4	0.832	0.702	0.9
23	Vavasis3	1.055	1.262	1.2

Table 5.2: Running time in seconds of the etree (T_{etree}) and QR-column-count (T_{cnt}) algorithms on an IBM RS/6000-590.

	Matrix	Static $\frac{nnz(S_L)}{nnz(S_{L_c})}$	Static $\frac{nnz(S_L)}{nnz(S_H)}$	Dynamic $\frac{nnz(S_L)}{nnz(S_{L_c})}$	Dynamic $\frac{nnz(S_L)}{nnz(S_H)}$
1	MEMPLUS	$<$ .01	.04	.23	.68
2	GEMAT11	.52	.85	.87	.90
3	RDIST1	.48	.72	.64	.73
4	ORANI678	.11	.56	.48	.90
5	MCFE	.41	.73	.61	.89
6	LNSP3937	.41	.84	.75	.92
7	LNS3937	.42	.86	.77	.94
8	SHERMAN5	.50	.92	.82	.96
9	JPWH991	.52	.88	.81	.94
10	SHERMAN3	.57	.89	.91	.91
11	ORSREG1	.57	.90	.91	.92
12	SAYLR4	.53	.89	.89	.92
13	SHYY161	.54	.91	.91	.92
14	GOODWIN	.35	.95	.86	.98
15	VENKAT01	.07	.11	.69	.74
16	INACCURA	.47	.96	.97	.99
17	BAI	.53	.95	.96	.97
18	DENSE1000	1.00	1.00	1.00	1.00
19	RAEFSKY3	.57	.99	.98	.99
20	EX11	.56	.99	.99	1.00
21	WANG3	.09	.14	.86	.89
22	RAEFSKY4	.57	.99	.99	.99
23	VAVASIS3	.64	.95	.98	.98

Table 5.3: Supernode storage utilization by various upper bounds. The notations $nnz(S_L)$, $nnz(S_{L_c})$ and $nnz(S_H)$ denote the number of nonzeros in the supernodes of L, L_c and H, respectively.

of $nnz(D(r:n,r))$ and $s-r+1$ to allocate storage for the L supernodes within columns r and s. Since $nnz(L(r:n,r)) \leq nnz(D(r:n,r)) \leq nnz(B(r:n,r))$, the dynamic storage bound so obtained is usually tighter than the static bound.

The storage utilizations for the supernodes in S_L are tabulated in Table 5.3. The utilization is calculated as the ratio of the actual number of nonzeros in the supernodes of the L factor to the number of nonzeros in the supernodes of an upper bound structure. When collecting this data, the maximum supernode size t was set to 64. The results in the table lead to the following observations.

In both static and dynamic schemes, the bounds using H are tighter than those using L_c. The difference is especially large in the static schemes. Note that this observation is consistent with what George and Ng observed for a set of smaller test problems in [55]. For most matrices, the storage utilizations using the static bound by H are quite high; they are often greater than 70% and are over 85% for 14 out of the 21 problems.

In the static scheme using H, the storage utilizations for matrices 1, 14 and 21 are only 4%, 11% and 14% respectively. The dynamic schemes certainly overcome the low utilizations. When using H in the dynamic scheme, the utilizations now become 68%, 74% and 89% for those three problems. These percentage utilizations are quite satisfactory. For other problems, the dynamic approaches also result in higher utilizations.

The runtime overhead associated with the dynamic schemes is usually between 2% and 15% on the RS/6000-590. From these experiments, we conclude that the static scheme using H often gives a tight enough storage bound for S_L. For some problems, such as matrices 15 and 21, the dynamic scheme must be employed to achieve better storage utilization. Then the program will suffer from a certain amount of slowdown. Our code tries the static scheme first and switches to the dynamic scheme only if the static scheme predicts too much space.

5.5 Parallel performance

In this section we demonstrate efficiency of our algorithm on real machines. We first look at some distinct features of the Crays and their impact on our algorithm.

The Cray architecture is quite different from the other cache-based systems which we studied in Chapter 4. First, it has much larger bandwidth between memory and CPU (Table 5.1), and the memory organization is flat. This implies that cache reuse is not an issue. As shown in Figures 5.9 and 5.10, performance of SGEMV and SGEMM do not differ significantly. Once the matrix dimension exceeds the vector register length (128 and 64 words, respectively), performance remains roughly the same. In fact, with varied panel size w for SuperLU, we found that $w = 1$ gives the best performance. Secondly, a scalar code on this machine runs significantly slower than a vectorized code. In our code, the depth-first search algorithm is not vectorizable, and takes large percentage of the total factorization time.

Figure 5.11 shows the percentage of the total runtime spent in the depth-first search on single processors of the three parallel machines. On the Cray J90, because of the slow depth-first search, the Mflop rate is rather low. For dense matrix 18, about 40% of the total time is spent in the depth-first search alone, despite the fact that the symmetrically reduced graph is simply the graph of a tridiagonal matrix. SuperLU achieves only 65 out of 200 peak Mflops, while LAPACK achieves 165 Mflops. For large sparse problems, SuperLU achieves at most 89 Mflops. Figure 5.11 suggests that Alpha 21164 has the best integer performance, relative to floating-point performance. (or the worst floating-point performance relative to integer performance.)

5.5.1 Speedup

Tables 5.4 through 5.8 report the speedups of the parallel algorithm on the five platforms, with number of threads varied. Because of memory limits we could not test all problems on the SPARCcenter 2000. The speedup is measured against the best sequential runtime achieved by SuperLU on a single processor of each parallel machine.

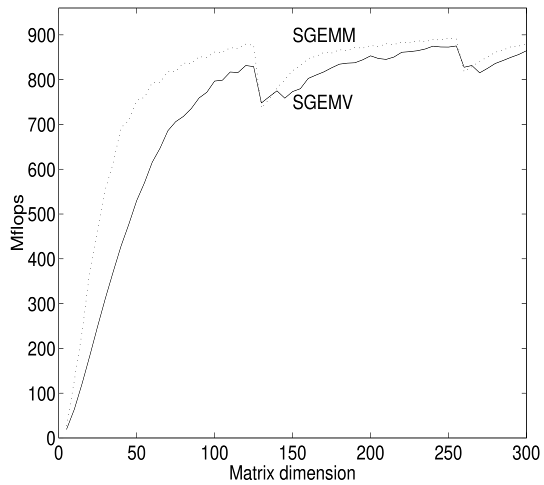


Figure 5.9: BLAS performance on single processor Cray C90.

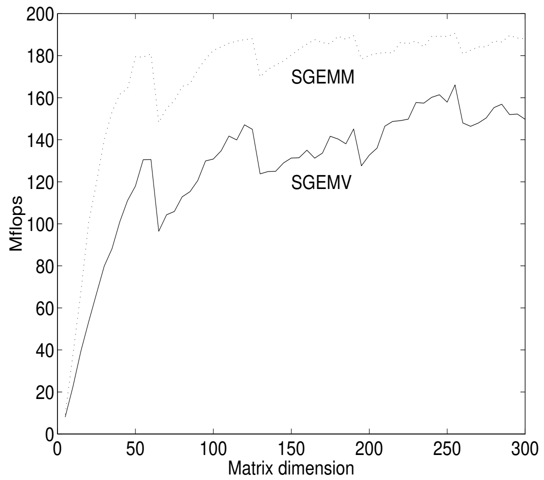


Figure 5.10: BLAS performance on single processor Cray J90.

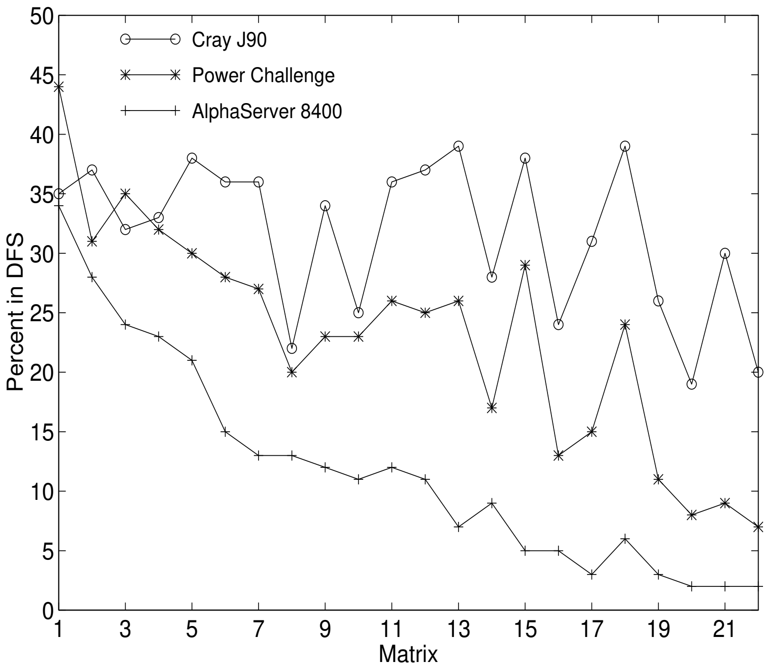


Figure 5.11: Percent of time spent in depth-first search on single processors.

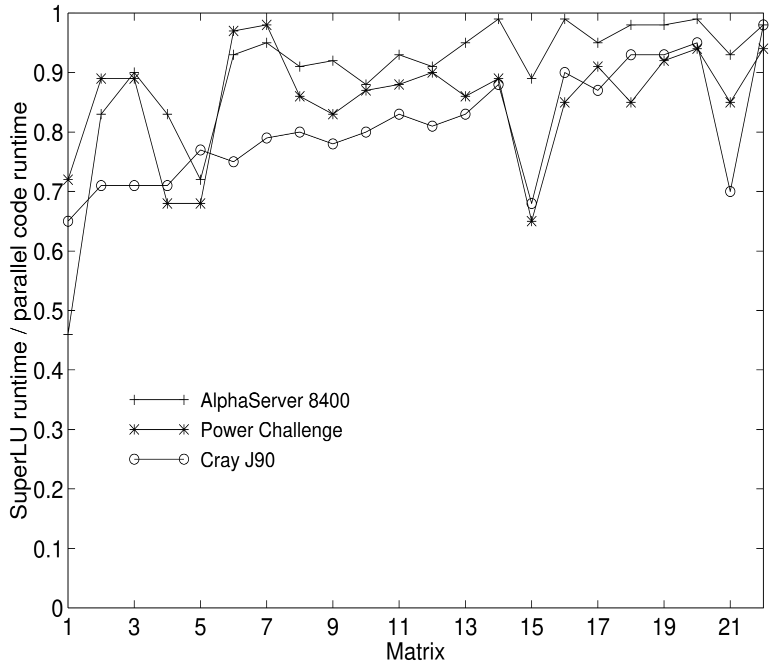


Figure 5.12: Overhead of the parallel code on a single processor, compared to SuperLU.

The column labeled "$P = 1$" illustrates the overhead in the parallel code when compared with the sequential code, using the same blocking parameters. This is also depicted in Figure 5.12. The structure of the parallel code, when run on a single processor, does not differ much from sequential SuperLU, except that a global task queue and various locks are involved. The extra work in the parallel code is purely integer arithmetic. Figure 5.12 also suggests that the Alpha 21164 has the best integer performance. Matrices 15 and 21 experience more overhead in the parallel code than the other large matrices. This is because we must use the dynamic memory allocation scheme developed in Section 5.4.3. The static upper bounds on supernodes storage are too loose for these two problems (Table 5.3).

The last two columns in each table show the factorization time and Megaflop rate, respectively, corresponding to the largest number of processors used. In order to achieve higher degree of concurrency, the panel size (w) and maximum size of a supernode ($maxsup$) for "$P > 1$" are set smaller than those used for "$P = 1$".

5.5.2 Working storage requirement

As in the sequential algorithm, parallel factorization requires a certain amount of working storage, and perhaps much more. In the shared memory parallel model, multiple threads share heap storage, static storage, and code, all residing in main memory. Each thread, upon execution, is allocated a private stack and has its own register set. Our program does not use many stack variables, so the stack size for each thread need not be very large. All working storage is allocated via `malloc()` from the heap. The working storage consists of two parts, where one part is shared among all threads, and another part

	Matrix	$P = 1$	$P = 2$	$P = 4$	Seconds	Mflops
1	MEMPLUS	0.44	0.82	0.74	2.35	1
2	GEMAT11	0.77	1.25	1.51	0.47	3
3	RDIST1	0.86	1.92	1.82	1.71	8
4	ORANI678	0.71	1.24	2.08	1.98	8
5	MCFE	0.79	1.38	2.00	0.45	9
6	LNSP3937	0.96	1.85	2.03	2.26	18
7	LNS3937	0.92	1.73	3.09	2.41	19
8	SHERMAN5	0.83	1.70	2.81	1.26	20
9	JPWH991	0.77	1.56	2.77	0.84	22
10	SHERMAN3	0.90	1.74	2.92	2.77	22
11	ORSREG1	0.89	1.75	3.17	2.27	27
12	SAYLR4	0.88	1.76	3.10	4.17	25
13	SHYY161	0.90	1.82	3.25	59.55	26
15	GOODWIN	0.92	1.86	3.61	20.50	33
18	DENSE1000	0.97	1.96	3.64	16.39	41
Mean		0.83	1.62	2.64		
Std		0.13	0.32	0.83		

Table 5.4: Speedup, factorization time and Mflop rate on a 4-CPU SPARCcenter 2000.

is local to each thread. The shared working storage is mainly used to facilitate the central scheduling activity and memory management. It includes:

- one integer array of size p used as the task queue, where p is the total number of panels;
- one bit vector of size n to mark whether a column is busy;
- four integer arrays of size n to record the status of each panel;
- one integer array of size n to record a column's most distant busy column down the etree during pipelining;
- three integer arrays of size n to implement storage layout for supernodes (Section 5.4.3).

The local working storage used by each thread is very similar to that used by sequential SuperLU, that is, all that is necessary to factorize one single panel. It includes:

- eight integer arrays of size n to perform the panel and column depth-first search;
- one n-by-w integer array to keep track of the position of the first nonzero of each supernodal segment in U;
- one n-by-w integer array to temporarily store the row subscripts of the nonzeros filled in the panel;

	Matrix	$P=1$	$P=4$	$P=8$	$P=12$	Seconds	Mflops
1	MEMPLUS	0.72	1.73	1.73	1.69	0.42	4
2	GEMAT11	0.89	1.86	2.36	3.71	0.07	22
3	RDIST1	0.89	1.66	1.56	2.23	0.44	32
4	ORANI678	0.68	1.72	2.40	2.56	0.45	33
5	MCFE	0.68	1.92	2.09	3.29	0.07	59
6	LNSP3937	0.97	3.00	3.65	3.86	0.35	122
7	LNS3937	0.98	2.98	3.92	3.73	0.40	117
8	SHERMAN5	0.86	2.29	3.09	3.09	0.23	111
9	JPWH991	0.83	2.40	3.43	5.33	0.09	205
10	SHERMAN3	0.87	2.36	2.78	2.78	0.40	157
11	ORSREG1	0.88	2.67	2.73	2.97	0.34	180
12	SAYLR4	0.90	2.81	3.48	4.58	0.38	284
13	SHYY161	0.86	2.71	3.54	5.06	4.64	332
14	GOODWIN	0.89	3.45	5.17	5.90	1.56	433
15	VENKAT01	0.65	1.72	2.00	1.98	15.37	209
16	INACCURA	0.85	2.77	4.14	5.00	9.53	438
17	BAI	0.91	2.98	5.10	6.70	8.87	722
18	DENSE1000	0.85	2.64	3.32	4.17	0.90	740
19	RAEFSKY3	0.92	3.07	5.62	6.91	11.35	1070
20	EX11	0.94	3.23	5.96	7.64	26.95	1046
21	WANG3	0.85	2.20	3.39	4.03	21.37	681
22	RAEFSKY4	0.94	3.05	5.17	6.52	33.57	936
23	VAVASIS3	0.91	3.58	6.06	6.69	105.06	862
Mean		0.86	2.56	3.59	4.37		
Std		0.09	0.59	1.36	1.73		

Table 5.5: Speedup, factorization time and Mflop rate on a 12-CPU SGI Power Challenge.

	Matrix	$P=1$	$P=2$	$P=4$	$P=6$	$P=8$	Seconds	Mflops
1	MEMPLUS	0.46	0.79	0.79	0.78	0.64	0.59	3
2	GEMAT11	0.83	1.63	1.88	1.88	1.88	0.08	20
3	RDIST1	0.90	1.98	2.10	1.77	1.77	0.31	40
4	ORANI678	0.83	1.29	2.00	2.33	2.42	0.26	57
5	MCFE	0.72	1.80	3.00	2.17	2.17	0.06	66
6	LNSP3937	0.93	1.94	3.19	3.68	3.68	0.25	159
7	LNS3937	0.95	1.83	3.08	3.81	4.12	0.25	187
8	SHERMAN5	0.91	1.89	2.89	2.94	2.94	0.17	151
9	JPWH991	0.92	1.89	3.00	3.30	3.00	0.11	178
10	SHERMAN3	0.88	1.83	2.72	2.74	2.74	0.34	180
11	ORSREG1	0.93	1.88	2.93	3.35	3.35	0.26	231
12	SAYLR4	0.91	1.98	3.20	3.78	4.08	0.38	276
13	SHYY161	0.95	1.93	3.23	4.21	4.79	4.66	334
14	GOODWIN	0.99	1.98	3.68	5.39	6.33	1.49	453
15	VENKAT01	0.89	1.92	2.95	3.04	3.16	10.62	303
16	INACCURA	0.99	1.83	3.08	4.15	5.02	10.94	380
17	BAI	0.95	1.98	3.72	5.03	5.77	11.58	553
18	DENSE1000	0.98	1.86	3.35	4.32	4.80	0.99	675
19	RAEFSKY3	0.98	1.98	3.81	3.16	3.61	28.65	422
20	EX11	0.99	1.98	3.76	5.56	7.06	34.23	781
21	WANG3	0.93	1.98	3.69	4.75	5.61	21.36	682
22	RAEFSKY4	0.98	1.98	3.81	5.44	6.63	42.79	734
23	VAVASIS3	0.96	1.97	3.69	5.28	6.64	124.24	724
Mean		0.92	1.74	2.89	3.59	4.01		
Std		0.13	0.28	0.81	1.31	1.77		

Table 5.6: Speedup, factorization time and Mflop rate on an 8-CPU DEC AlphaServer 8400.

	Matrix	$P=1$	$P=2$	$P=4$	$P=6$	$P=8$	Seconds	Mflops
1	MEMPLUS	0.66	0.75	0.74	0.72	0.71	1.24	2
2	GEMAT11	0.76	1.36	2.27	3.09	3.40	0.10	15
3	RDIST1	0.71	1.98	2.41	2.41	2.31	0.48	34
4	ORANI678	0.72	1.24	2.22	2.91	3.20	0.41	37
5	MCFE	0.69	1.25	1.82	2.00	2.00	0.10	43
6	LNSP3937	0.78	1.51	2.77	2.84	4.41	0.27	151
7	LNS3937	0.78	1.51	2.95	3.97	4.23	0.30	156
8	SHERMAN5	0.77	1.49	2.90	3.59	4.07	0.15	170
9	JPWH991	0.78	1.52	2.50	3.18	2.92	0.12	164
10	SHERMAN3	0.79	1.48	2.53	2.97	2.97	0.29	214
11	ORSREG1	0.80	1.53	2.69	3.25	3.55	0.22	278
12	SAYLR4	0.83	1.58	3.05	3.85	3.97	0.33	318
13	SHYY161	0.80	1.50	2.87	3.87	4.86	3.29	477
14	GOODWIN	0.84	1.65	3.31	4.83	6.59	0.99	682
15	VENKAT01	0.70	1.28	1.65	1.73	1.74	14.04	229
16	INACCURA	0.86	1.70	3.19	4.38	5.21	5.18	807
17	BAI	0.84	1.63	3.22	4.56	4.89	6.24	1035
18	DENSE1000	0.95	1.86	2.95	3.30	3.55	0.71	943
19	RAEFSKY3	0.91	1.74	3.45	4.77	5.83	6.17	1977
20	EX11	0.90	1.65	3.21	5.02	6.53	10.37	2583
21	WANG3	0.78	1.48	1.82	2.31	2.32	14.62	996
22	RAEFSKY4	0.92	1.80	3.43	4.60	5.46	13.13	2399
Mean		0.80	1.53	2.63	3.42	3.85		
Std		0.08	0.27	0.67	1.11	1.55		

Table 5.7: Speedup, factorization time and Mflop rate on an 8-CPU Cray C90.

	Matrix	$P=1$	$P=4$	$P=8$	$P=12$	$P=16$	Seconds	Mflops
1	MEMPLUS	0.65	0.94	0.98	0.97	0.76	3.67	1
2	GEMAT11	0.71	2.44	4.38	5.25	5.83	0.18	8
3	RDIST1	0.71	2.86	2.88	2.71	2.39	1.53	10
4	ORANI678	0.71	2.07	3.11	3.82	3.85	1.13	13
5	MCFE	0.77	2.21	2.70	2.70	2.52	0.29	15
6	LNSP3937	0.75	2.87	4.91	6.21	6.39	0.66	62
7	LNS3937	0.79	2.75	4.63	5.41	5.41	0.83	58
8	SHERMAN5	0.80	2.91	4.64	5.07	5.32	0.41	63
9	JPWH991	0.78	2.72	3.57	3.68	3.38	0.37	49
10	SHERMAN3	0.80	2.63	3.49	3.42	3.31	0.96	66
11	ORSREG1	0.83	2.83	3.88	4.22	4.16	0.70	89
12	SAYLR4	0.81	2.91	4.26	4.82	4.82	0.99	108
13	SHYY161	0.83	2.92	5.30	6.94	7.47	8.06	196
14	GOODWIN	0.88	3.32	6.66	10.02	12.81	1.94	354
15	VENKAT01	0.68	1.84	1.96	1.98	1.90	47.34	68
16	INACCURA	0.90	3.26	5.55	6.64	7.39	15.09	277
17	BAI	0.87	3.22	5.98	7.55	8.49	15.05	431
18	DENSE1000	0.93	2.84	3.79	3.92	3.91	2.61	256
19	RAEFSKY3	0.93	3.38	6.20	7.69	8.43	19.03	641
20	EX11	0.95	3.56	6.53	9.47	10.17	32.48	831
21	WANG3	0.77	2.53	3.21	3.14	3.06	50.42	288
22	RAEFSKY4	0.98	3.54	5.87	7.36	8.12	43.54	723
Mean		0.81	2.75	4.29	5.13	5.45		
Std		0.09	0.60	1.51	2.38	2.97		

Table 5.8: Speedup, factorization time and Mflop rate on a 16-CPU Cray J90.

- one n-by-w real array used as the SPA.
- one scratch space of size $(t + b) \times w$ to help BLAS calls. See Figure 4.14 for the definition of t, b and w.

This amount of local storage should be multiplied by P, where P is the number of threads created. Thus the working storage grows affinely with respect to P, and this algorithm, albeit efficient, is hard to scale up from a memory point of view.

To put this in perspective, Table 5.9 compares the working storage requirement with the actual LU storage. The last two columns report the amount of working storage as a fraction of the total LU storage in Megabytes, for 1 and 8 threads, respectively. It is clear that for $P = 8$, the working storage requirement can be comparable to the LU storage for small problems. For large problems, working storage is typically 10% to 20% of the LU storage. Matrix 13 is exceptionally bad: it is a matrix of medium size for which the required working storage is more than LU storage. Since we would not use multiple processors on the small problems anyway, so the overall working storage requirement is quite small.

5.6 Overheads in parallelization

In this section we quantify all the overheads associated with our parallel algorithm. The overhead mainly comes from four sources: the reduced per-processor efficiency due to smaller granularity of unit tasks, accessing critical sections via locks, orchestrating the dependent tasks via event notification, and load imbalance. The purpose of this section is to understand how much time is spent in each part of the parallel algorithm and explain the speedups we saw in Section 5.5.1.

5.6.1 Decreased per-processor performance due to smaller blocking

The first overhead is due to the necessity to reduce the blocking parameters in order to achieve more concurrency. Recall that two blocking parameters affect performance: panel size (w) and maximum size of a supernode ($maxsup$). For better per-processor performance, we prefer larger values. On the other hand, the large granularity of unit tasks limits the degree of concurrency.

On the Cray J90, this trade-off is not so important, because a small w ($w = 1$) is good for the sequential algorithm. We therefore also use $w = 1$ in the parallel algorithm. When varying the value of $maxsup$, we find that performance is quite robust in the range between 16 and 64.

On the Power Challenge and AlphaServer 8400, we observe more dramatic differences with varied blockings. Figure 5.13 and 5.14 illustrate this loss of efficiency for several large problems on single processors of the two machines, Power Challenge and AlphaServer 8400, respectively. In this experiment, the parallel code is run on single processors with two different settings of w and $maxsup$. Figure 5.13 shows, on a single processor Power Challenge, the ratio of the runtime with the best blocking for 1 CPU ($w = 24, maxsup = 64$) to the runtime with the best blocking for 12 CPUs ($w = 12, maxsup = 48$). Figure 5.14 shows the analogous ratio for the 8-CPU AlphaServer 8400. On the Power Challenge, the

	Matrix	LU storage (MB)	Fraction of $P=1$	LU storage $P=8$
1	MEMPLUS	16.27	.23	1.51
2	GEMAT11	1.15	.89	5.92
3	RDIST1	3.70	.23	1.54
4	ORANI678	4.77	.11	.73
5	MCFE	0.88	.18	1.26
6	LNSP3937	4.93	.16	1.10
7	LNS3937	7.04	.12	.77
8	SHERMAN5	2.75	.25	1.66
9	JPWH991	1.58	.13	.88
10	SHERMAN3	4.68	.22	1.47
11	ORSREG1	4.23	.11	.72
12	SAYLR4	6.98	.10	.70
13	SHYY161	80.01	.19	1.31
14	GOODWIN	34.25	.04	.30
15	VENKAT01	566.09	.02	.15
16	INACCURA	106.06	.03	.21
17	BAI	145.02	.03	.22
18	DENSE1000	9.90	.02	.14
19	RAEFSKY3	183.65	.02	.16
20	EX11	277.59	.01	.08
21	WANG3	459.14	.01	.07
22	RAEFSKY4	271.28	.02	.10
23	VAVASIS3	521.75	.02	.11

Table 5.9: Working storage requirement as compared with the storage needed for L and U. The blocking parameter settings are: $w = 8$, $t = 100$, and $b = 200$.

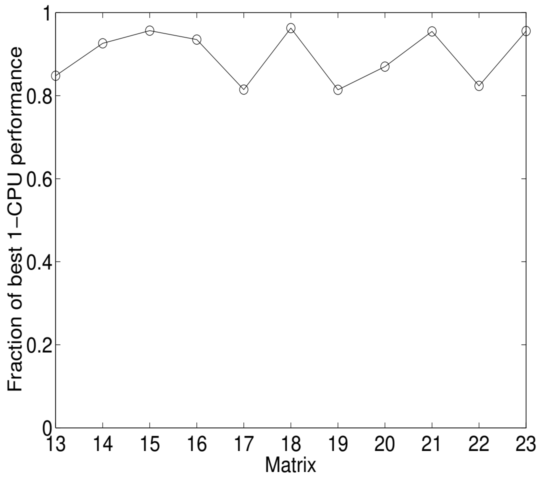


Figure 5.13: Performance of sequential code with blockings tuned for parallel code on 1-CPU Power Challenge.

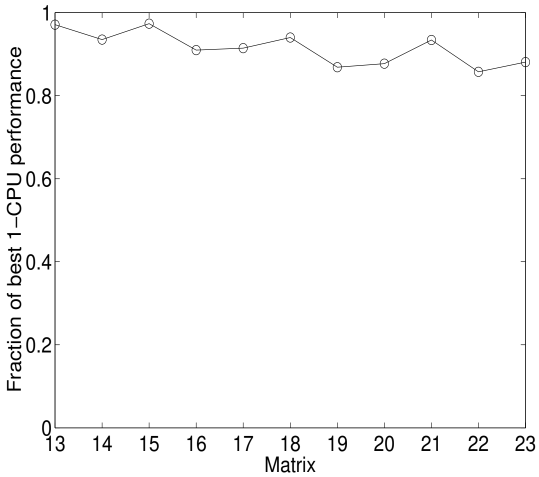


Figure 5.14: Performance of sequential code with blockings tuned for parallel code on 1-CPU AlphaServer 8400.

blocking used for best parallel performance achieves only 81% uniprocessor efficiency for matrices 17 and 19. The corresponding lowest number on the AlphaServer 8400 is 86% for matrix 22.

5.6.2 Accessing critical sections

The following program segment and shared data structures must be protected under mutual exclusion: (1) The `Scheduler()` routine can only be entered by one processor at a time, because it modifies the contents of the global task queue. (2) Every time a processor needs to copy part of L and/or U from its local working arrays into the global store, it has to call the allocator to get space. (By now, the size of each column or supernode is already known.) This amounts to one call per column of U for row subscripts (`Usub`) and values (`Uval`), and one call per supernode of L for row subscripts (`Lsub`). The storage management for the L supernodes, S_L, is discussed in Section 5.4.3. There, the static scheme does not need locking, because the storage is pre-allocated according to an upper bound estimate. The dynamic scheme, on the other hand, still requires locking. Each call to the allocator involves acquiring and relinquishing a lock, although the duration in the critical section is very short. (3) The increment of supernode number (`nsuper`) should also be protected, because different processors may detect different supernodes simultaneously.

In Table 5.10, we roughly count the number of times the program acquires and relinquishes various locks. Note that the total number of lockings performed are independent of number of processors.

A mutex variable should be declared for each critical section. Since we want to allow more than one processor to enter different critical sections simultaneously, we use five named mutex variables to guard each of the above critical regions.

To see how much cost is associated with lockings, in Table 5.11, we measured the

Critical section	Counts
call `Scheduler()`	number of panels (approx.)*
allocate `Lsub`	number of supernodes
allocate `Usub/Uval`	number of columns
allocate S_L	number of supernodes
increment `nsuper`	number of supernodes

Table 5.10: Number of lockings performed.
* Here we assume that `Scheduler()` returns a new panel upon each call.

Machine	$P=1$	$P=4$	P=8
SPARCcenter 2000	1.63 (82)	4.34 (217)	4.36 (218)
Power Challenge	1.13 (102)	1.98 (179)	2.02 (182)
AlphaServer 8400	0.98 (294)	2.26 (678)	2.71 (814)
Cray C90	1.34 (323)	1.09 (261)	1.40 (336)
Cray J90	2.67 (267)	4.17 (417)	4.42 (442)

Table 5.11: Time in microseconds (cycles) to perform a single lock and unlock.

time it takes to acquire and relinquish a lock on several platforms, with different numbers of threads P. The figure in the parenthesis is the number of clock cycles. In this small benchmark code, the critical section is simply one statement, to increment a counter. The locking and unlocking are placed around this statement. The measurement is done in a tight loop with many iterations. When there is more than one thread, this corresponds to worst-case contention, because all the threads do nothing besides competing for the lock. Note that the cost for $P > 1$ is usually more than twice that of $P = 1$, because in the latter case there is no queuing contention to obtain the lock. When there is more than one thread, the time increases slightly, but not linearly in the number of threads.

The uniprocessor slowdown observed in Figure 5.12 is partly due to the overhead incurred by using these locks, when there are no other processors competing for the locks. By multiplying the time for a single lock/unlock in Table 5.11 by the number of the lockings performed in Table 5.10, we can estimate the locking overhead. As a concrete example, let us consider a medium size matrix 13, on a single processor Cray J90. Since the sequential code performance is 26 Mflops, each lock/unlock is equivalent to roughly 69 floating-point operations. When the factorization is performed with panel size $w = 1$, the total number of lock acquisitions is 237004, which, when multiplied by 2.67 microseconds, results in about 0.64 seconds. This is less than 3% of the entire factorization time (24.85 seconds). We observe that this percentage is typical for all matrices. The locking overhead also varies with machines. For example, it is higher on the Cray J90 than on the Power Challenge and the AlphaServer 8400.

This estimate ignores time spent waiting for a processor that is in the critical section, because Table 5.11 had a trivial critical section. In our parallel LU code, most critical sections are trivial, except for calling `Scheduler()` (see Table 5.10).

5.6.3 Coordinating dependent tasks

The third source of overhead is due to insufficient parallelism in the pipelined executions of the dependent panels. Dependent panels are those that have an ancestor-descendant relation in the column etree. When a processor factoring a panel needs an update from a BUSY descendant panel, this processor simply spins, and waits for that panel to finish, as shown at line 13 in the scheduling loop of Figure 5.2. During the spin wait the processor does nothing useful. The total amount of spin wait time observed is significant in some cases, especially with a larger number of processors. For example, for matrix 16, on the 12-CPU Power Challenge, about 40% of the parallel runtime is spent spinning. The corresponding number for the dense matrix is about 58%. The dense matrix is the worst one, because the factorization of all panels must be carried out in pipelined fashion.

Figure 5.15 depicts the locking overhead from Section 5.6.2 and spinning due to dependencies on the 8-CPU Cray J90. The locking overhead also includes the possible contention from the 8 processors. In this figure, we also plot the inefficiency of the parallel algorithm. Here, $\mathit{efficiency} = \frac{\mathit{speedup}}{P}$, and $\mathit{inefficiency} = 1 - \mathit{efficiency}$. For most matrices, the spinning overhead due to dependencies is much higher than the overhead from lock acquisition. Clearly, the loss of parallel efficiency is at least as large as the percent of time spent in spin wait. The inefficiency curve captures very well the overhead curve for spin wait. In particular, for larger and denser problems, the spin wait is responsible for most of the inefficiency. For the dense matrix 18, the spin wait contributes more than 90% of the inefficiency. (Even in the presence of these overheads, the parallel efficiency of the six largest problems still exceed 70%.)

5.6.4 Load imbalance

We use a balance factor B to measure the load balance. Let f_i denote the numbers of floating-point operations performed on processor i, and P denote the numbers of processors. We define B as

$$B = \frac{\sum_i (f_i)}{P \ \max_i (f_i)} \ . \tag{5.2}$$

In words, B equals the average work load divided by the maximum work load. It is readily seen that $0 < B \leq 1$, and higher B indicates better load balance. This figure is shown in Tables 5.12 and 5.13.

If load imbalance is the sole overhead in a parallel program, the parallel execution time is simply the execution time of the slowest processor whose work load is highest.

5.6.5 Combining all overheads

In this subsection we evaluate the effect of the combined overheads on the parallel efficiency. In summary, the overheads include

(1) reduced uniprocessor performance due to smaller blocking
(2) accessing critical sections
(3) idle time (from spin wait in the panel pipeline and in the top-level scheduling loop)

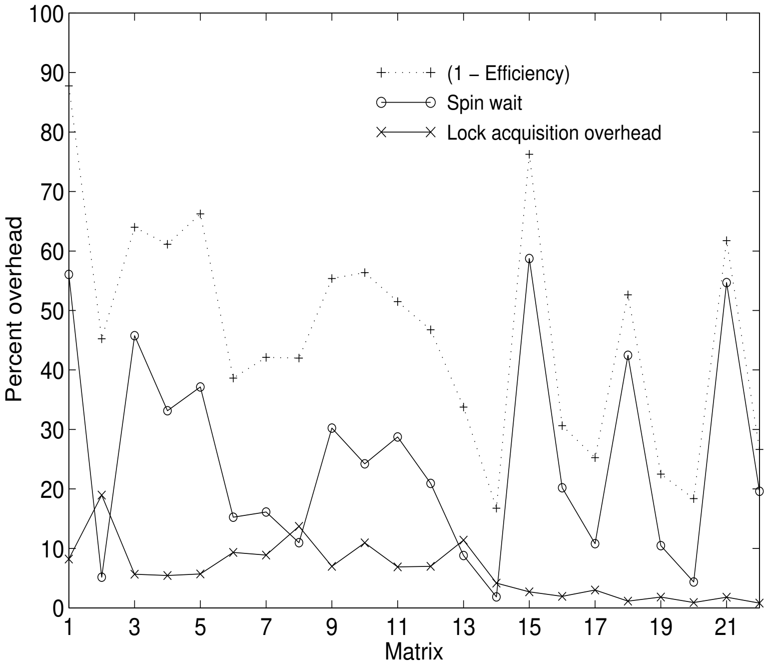


Figure 5.15: Parallel overhead in percent on an 8-CPU Cray J90.

(4) load imbalance

Overhead (1) only affects uniprocessor performance. Overhead (2) decreases both uniprocessor performance of the parallel code and parallel performance. Compared with the serial execution, the parallel execution experiences more contention for locks. But Table 5.11 and Figure 5.15 indicate that runtime does not increase significantly in the presence of contention. Therefore, we may assume that (2) only adds overhead to the uniprocessor execution. Overheads (3) and (4) exist only in the parallel computation and their magnitudes are correlated. Load imbalance may be due to poor assignment of tasks to processors, or insufficient parallelism. In either case, the processor with less work to do will sit idle. In addition, a processor may be idle due to dependencies between tasks, even when the total work performed by all processors is balanced. We introduce the following notation to denote various times:

- T_s is the best serial time obtained from SuperLU
- T_1 is the execution time of the parallel code on one processor
- T_P is the parallel execution time on P processors
- T_I is the total idle time of all processors

All the times above are measured independently. In particular, for T_I, there are two situations a processor may sit idle: one is due to spin wait in the pipeline, and another is when a processor calls `Scheduler()` (line 4 in Figure 5.2) and fails to get a panel from the scheduler. We found that, for the test matrices and the numbers of processors being considered, failure from the scheduler rarely occurs. So most of the idle time is due to pipeline waiting. The following relation holds for the parallel runtime:

$$P\,T_P \approx T_1 + T_I \,. \tag{5.3}$$

We can compute the observed efficiency (E_{actual}) and the estimated efficiency (E_{est}) as follows:

$$E_{actual} = \frac{T_s}{P\,T_P} \,. \tag{5.4}$$

$$E_{est} = \frac{T_s}{T_1 + T_I} \,. \tag{5.5}$$

We also introduce two parameters α_1 and α_p to quantify the uniprocessor and parallel overheads, respectively. α_1 and α_p are calculated based on T_s, T_1, T_P, and T_I as follows:

$$\alpha_1 = \frac{T_1 - T_s}{T_1} = 1 - \frac{T_s}{T_1} \,. \tag{5.6}$$

$$\alpha_p = \frac{T_I/P}{T_P} \,. \tag{5.7}$$

Both α_1 and α_p are in the range $[0, 1)$; α_1 shows the overhead that degrades the uniprocessor performance, while α_p shows the overhead in the parallel execution. The smaller are α_1 and α_2, the more efficient is the parallel algorithm. In Tables 5.12 and 5.13, we report E_{actual}, E_{est}, α_1, α_p, and B for the two parallel machines.

Cray J90

As mentioned in Section 5.6.1, the uniprocessor performance on the J90 does not degrade much with smaller *maxsup*, that is, overhead (1) does not exist. Therefore, $1 - \alpha_1$ can be taken as the numbers from the column labeled "$P = 1$" in Table 5.8. We gathered the statistics for α_p and B on 16 processors, as shown in Table 5.12. In the last two columns of Table 5.12, we compare the estimated efficiency by (5.5) with the actually observed efficiency E_{actual} by (5.4).

The estimated and observed efficiencies are very close. Their differences are mostly within 4%, except for matrix 20 which has a 7% difference. For most problems, the pipeline spin waiting, as reflected by α_p, is the primary cause of inefficiency. This is particularly evident for matrices 15, 18 and 21, for which 74%, 67% and 71% of the total runtime is spent in spin wait, respectively. Most problems have achieved good load balance, with exception of matrix 13.

Power Challenge

On a cache-based machine, the uniprocessor performance loss of the parallel code is a combination of performing lockings and less efficient cache utilization. Therefore, $1 - \alpha_1$ equals the product of the numbers from column labeled "$P = 1$" in Table 5.5 and the

	Matrix	α_1	α_p	B	E_{est}	E_{actual}
13	SHYY161	.17	.23	.66	.51	.47
14	GOODWIN	.12	.10	.97	.77	.80
15	VENKAT01	.32	.74	.99	.11	.12
16	INACCURA	.10	.46	.97	.45	.46
17	BAI	.13	.34	.93	.54	.53
18	DENSE1000	.07	.67	.99	.25	.25
19	RAEFSKY3	.07	.37	.96	.55	.53
20	EX11	.05	.23	.98	.71	.64
21	WANG3	.23	.71	.99	.17	.19
22	RAEFSKY4	.02	.43	.97	.53	.51

Table 5.12: Overheads and efficiencies on a 16-CPU Cray J90.

	Matrix	α_1	α_p	B	E_{est}	E_{actual}
13	SHYY161	.27	.20	.70	.39	.42
14	GOODWIN	.18	.25	.87	.49	.49
15	VENKAT01	.38	.56	.91	.20	.17
16	INACCURA	.21	.40	.88	.42	.42
17	BAI	.26	.20	.93	.55	.56
18	DENSE1000	.18	.58	.92	.30	.35
19	RAEFSKY3	.25	.16	.95	.60	.58
20	EX11	.18	.09	.98	.73	.64
21	WANG3	.19	.52	.93	.36	.34
22	RAEFSKY4	.23	.15	.95	.62	.54
23	VAVASIS3	.14	.17	.97	.68	.56

Table 5.13: Overheads and efficiencies on a 12-CPU Power Challenge.

numbers from Figure 5.13. Table 5.13 reports the statistics of α_1, α_p and B, together with the estimated and the observed efficiencies, E_{est} and E_{actual}, respectively. Again, E_{est} and E_{actual} match reasonably well, except for matrix 23, for which the gap is 12%.

Compared with J90, we observe that α_1 is much larger, because the cache plays an important role on the Power Challenge. In fact, for matrices 13, 17, 19, 20 and 22, uniprocessor performance loss is more severe than the parallel overhead. For matrices 15, 18 and 21, the parallel spin waiting is the major bottleneck. Again, load balance is usually very good, except for matrix 13.

5.7 Possible improvements

We have considered two ways to circumvent the inefficiency caused by dependencies. One is to uncover more independent panels and hence decrease the number of pipelined panels. Another is to use a more sophisticated dynamic scheduling algorithm that steals cycles from the idle processors to do useful work.

5.7.1 Independent domains

The concept of *domains* has been widely used in sparse Cholesky factorizations, especially on distributed memory machines [14, 72, 93, 97]. A domain refers to a rooted subtree of the elimination tree such that all nodes in this subtree are mapped onto the same processor to factorize. In sparse LU factorization, we may define domains similarly, but we use the column etree. The factorization within each domain does not require pipelining or cooperation among processors. Therefore, the benefit of using domains is two-fold: (1) it decreases the number of pipelined panels; (2) it improves locality. Since the etree is in postorder, a domain consists of consecutive columns in the matrix. Note that our relaxed supernodes (Section 3.4) at the bottom of the etree are in fact domains, but they are too small to warrant the above listed benefits.

The next question is how we shall find the domains. We first examine what people have done in sparse Cholesky factorizations. For a well balanced etree, often coming from a nested dissection ordering, the *subtree-to-subcube mapping* [53] is quite effective. In this method, the processors are recursively divided into two groups at each branching node of the tree, until the log P level is reached. At this level there are exactly P disjoint subtrees, or domains, each being assigned to one processor.

A generalization of this method, called *proportional mapping*, was proposed by Pothen and Sun [93], which is intended to work for any unbalanced tree. First, we compute the amount of arithmetic associated with each node when factorizing the corresponding column, and the amount of arithmetic associated with each subtree. Secondly, we traverse etree in a top-down fashion, starting with P processors at the root. When a branching node with k children is reached, the processors are divided among k children, each with a number of processors proportional to the relative amount of work required by the child subtree. This process terminates when a single processor is assigned to a single subtree.

There is a problem with proportional mapping, that is, the proportion of the number of processors determined from the work distribution of the children may not be an integral number. It is necessary to round it to an integer. This rounding may cause serious load imbalance. Geist and Ng [47] proposed a tasking scheduling method that can alleviate this problem. They relaxed the condition of finding exactly P subtrees. Instead, their algorithm may find more than P subtrees, so there will be more flexibility to assign them to the P processors with reasonable load balance. Their algorithm involves a breadth first search of the etree, cutting off the branches and applying a heuristic bin-packing algorithm to assign the set of branches to the P bins. This procedure is applied recursively until the work load across all processors meets a certain tolerance. Intuitively, if the load balance requirement is high, the algorithm will find more subtrees, each with smaller work load.

We use Geist and Ng's approach due to its generality and flexibility. To adapt their algorithm to our LU context, we experimented with the following scheme:

(1) Since we cannot pre-determine the exact amount of arithmetic, we use the column etree and the nonzero column counts of the Householder matrix H (Section 5.4.3) to arrive at an estimated amount of work for each column. This gives an upper bound the actual work.

(2) We ignore the static schedule of domains to processors; instead, we add the domains into the task queue Q. Processors get domains from Q as factorization proceeds, and hopefully load balance is automatically maintained.

In the first step we find the domains based statically only on estimated work. In the second step we dynamically schedule them onto processors. Therefore, the static load balance requirement is not so critical.

We experimented this method on the 12-CPU Power Challenge and the 16-CPU Cray J90. Unfortunately, there is no strong evidence that this pre-scheduling is very profitable. For some problems, we see improvement; while for some others, we see degradation. Neither the improvement nor the degradation is more than 5%. There may be several reasons that this method is not effective. First, LU factorization has a larger proportion of the work at the top of the etree than does Cholesky. So there is not enough of the total work in the domains at the bottom. Second, the gross estimate of floating-point operations may not be sufficiently accurate. Third, our original scheduling policy has already captured enough locality by favoring the immediate parent in the etree (Section 5.3). We believe that this type of static pre-schedule will be much more useful for our future algorithm on distributed memory machines, where a pure dynamic scheduling will be far too expensive to implement.

5.7.2 No-spin-wait scheduling

Instead of spin waiting for a BUSY panel, the processor can put the current panel back in the queue, mark the panel as "partially-done", and find another panel to work on. To implement this scheme, we may use two separate queues, say $Q1$ and $Q2$, where $Q1$ is the same task queue in Figure 5.2 and $Q2$ holds the partially-done panels. The scheduler switches between the two queues to assign panels to an available processor. It might be advantageous to give $Q2$ a higher priority, because all partially-done panels are likely to be on the critical path of the computation.

When a partially-done panel is swapped off a processor and put in $Q2$, the partial factorization result of this panel must be kept in order for the panel to resume factorization correctly. This includes all the nonzero subscripts and the numeric values in the SPA. Because of the large amount of state information that must be swapped in and out of a processor, the bookkeeping may be prohibitively expensive in practice.

When a processor stops working on a partially-done panel, the type of the panel the processor picks up will affect the overall performance. There are two scenarios:

- If the processor obtains a panel not dependent on the partially-done panel, say from a disjoint subtree, the scheme will certainly pay off. This could well happen in the beginning of the computation.
- If the processor obtains a panel which is an ancestor of the partially-done panel, the partially-done panel clearly lies on the critical path and becomes the bottleneck. This may happen in a later stage of the computation when the etree becomes narrow. In this case, having a cascading of partially-done panels implies many swappings of states, and may severely hurt performance.

We have yet to implement this method. In Chapter 6 we propose other ways to improve the efficiency of the parallel algorithm, which we think may be more effective.

5.8 A PRAM model to predict optimal speedup

Given a matrix with a fixed column ordering, we want to establish a performance model to estimate the maximum speedup attainable by the underlying algorithm, and indeed determine the limitations of algorithms based on a one-dimensional matrix partition. To this end, we will estimate a lower bound on the parallel runtime. We shall first formalize our notion of *parallel completion time* and describe how to compute it. We will then show the results of applying this model to our test matrices.

5.8.1 Method

In a parallel algorithm the total amount of work is divided among multiple processors. Because of various precedence constraints, some part of the work must be finished before some other part of the work can start. Thus, the completion time of the parallel algorithm is constrained by the amount of work that must be finished *serially* in time, i.e., the critical path. Our objective is to predict the shortest possible parallel completion time. In our model we make the following simplifying assumptions: (1) The work only includes floating-point operations, and each floating-point operation takes one unit of time. (2) There are an infinite number of processors. Whenever a task is ready, there will be a free processor to execute this task immediately. (3) Accessing memory and communication are free in this hypothetical machine. (4) We ignore various overheads associated with the actual implementation of the scheduling algorithm and the synchronizations. This model gives an optimistic estimate; therefore, we can use it to prove lower bounds on the performance of an algorithm on a real machine.

$$
\begin{pmatrix}
1 & \bullet & & & & & & & & \\
\bullet & 2 & \bullet & & & & & \bullet & & \\
 & & 3 & & & & \bullet & \bullet & & \\
 & & & 4 & \bullet & & & & \bullet & \\
 & & & \bullet & & \bullet & \bullet & & & \\
 & & & \bullet & \bullet & 6 & & & \bullet & \\
 & & & & & & 7 & & \bullet & \bullet \\
\bullet & & & & & & & & & \\
 & & & \bullet & & \bullet & & \bullet & 9 & \\
 & & \bullet & & & & & & & 10
\end{pmatrix}
\Longrightarrow
\begin{pmatrix}
1 & \bullet & & & & & & & & \\
\bullet & 2 & \bullet & & & & & \bullet & & \\
 & & 3 & & & & \bullet & \bullet & & \\
 & & & 4 & \bullet & & & & \bullet & \\
 & & & \bullet & 5 & \bullet & & & \bullet & \\
 & & & \bullet & \bullet & 6 & \bullet & & & \\
 & & & & & & 7 & & \bullet & \bullet \\
 & & & \bullet & \bullet & \bullet & \bullet & 8 & \bullet & \bullet \\
\bullet & \bullet & \bullet & & & & \bullet & \bullet & 9 & \bullet \\
 & & \bullet & & & & \bullet & \bullet & \bullet & 10
\end{pmatrix}
$$

Original A Factors $F = L + U - I$

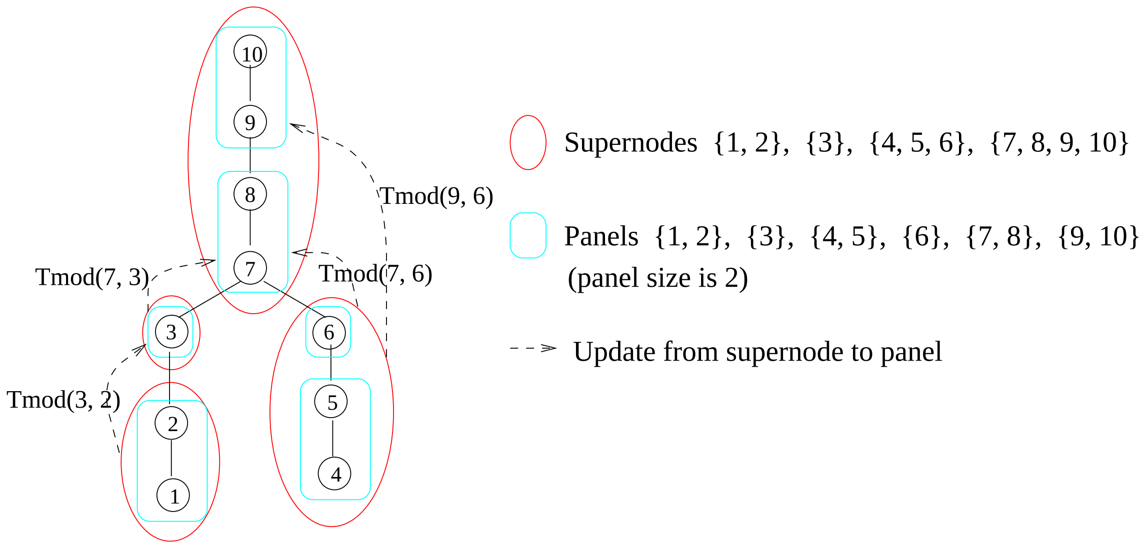


Figure 5.16: An example of computational DAG to model the factorization.

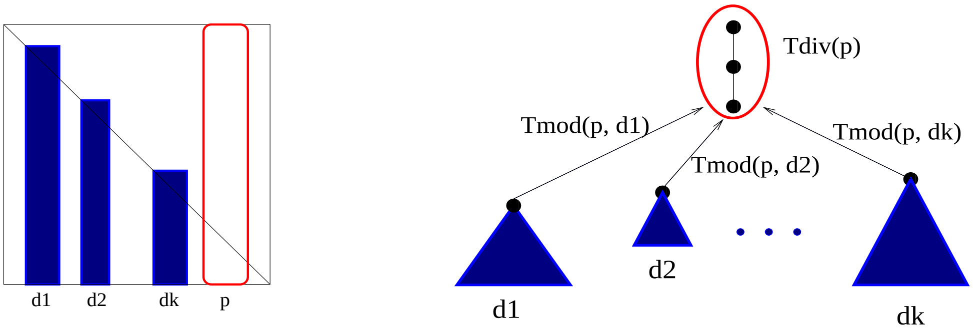


Figure 5.17: Tasks associated with panel p.

The *LU* factorization algorithm presented in Section 5.3 can be modeled by a data structure called a directed acyclic graph (DAG). Each node in the DAG corresponds to the computation of a panel. An edge directed from node s to node p corresponds to an update of panel p by supernode s. The edges also represent precedence relations between the updating supernodes and the destination panels. Figure 5.16 illustrates such a DAG for a sample matrix.

5.8.2 Model

In presenting our model, we employ the following notation:

- $T_{mod}(p, d)$:= the task of updating panel p by a descendant supernode d
- $T_{div}(p)$:= the task of performing the inner factorization of panel p
- $tmod(p, d)$:= time taken by task $T_{mod}(p, d)$
- $tdiv(p)$:= time taken by task $T_{div}(p)$
- $EST(p)$:= earliest possible starting time of $T_{div}(p)$
- $EFT(p)$:= earliest possible finishing time of $T_{div}(p)$

All times are expressed in units of floating-point operations. It is clear that for any panel p the following relation holds: $EFT(p) = EST(p) + tdiv(p)$.

According to our scheduling algorithm, each panel task $T_{panel}(p)$ is assigned to a single processor P. $T_{panel}(p)$ consists of the following two types of subtasks:

$$T_{panel}(p) := \{T_{mod}(p, d) \mid d \in \mathcal{D}\} \cup \{T_{div}(p)\} ,$$

where $\mathcal{D}$ is the set of descendant supernodes that update the destination panel p. Figure 5.17 shows the part of the DAG associated with a particular panel p.

Both T_{mod} and T_{div} are indivisible tasks, and are carried out sequentially on a processor. Clearly, T_{div} cannot start until all the T_{mod}'s have been finished. By looking at the precedence relations of these two types of tasks, we can determine the runtime of

$T_{panel}(p)$ on processor P. We will try to schedule these tasks as early as possible, in order to derive **the** minimum parallel execution time.

We first look at the tasks associated with one particular panel p, as shown in Figure 5.17. Suppose there are k descendant supernodes to update panel p, and that all the times $\{EFT(d), d \in \mathcal{D}\}$ have been computed. We schedule the tasks $\{T_{mod}(p, d), d \in \mathcal{D}\}$ to processor P in the order of $T_{mod}(p, 1), \ldots, T_{mod}(p, k)$, such that:

$$EFT(1) \leq EFT(2) \leq \ldots \leq EFT(k) \ .$$

Here, $EFT(i)$ is the finishing time of the last column of supernode i, because a supernode i cannot update any ancestor panel before its last column is completed. For convenience, we call this scheduling policy Sched-A. Then we can compute $EST(p)$ and $EFT(p)$ as follows.

1. Run the following recurrence to get the completion time of the T_{mod}'s:

   $t = 0$;
   for $i = 1$ to k
   $\quad t = \max \{ \ t, \ EFT(i) \ \} + tmod(i)$;
   endfor;

2. Set $EST(p) = t$ and $EFT(p) = t + tdiv(p)$.

In the following we will give an informal argument about the optimality of the parallel runtime resulting from Sched-A.

Theorem 9 *For panel p, scheduling the T_{mod}'s by Sched-A gives the shortest completion time.*

Proof: Processor P requires at least $\sum_{i=1}^{k} tmod(p, i)$ units of time to finish all the updates to panel p. Now suppose another scheduling strategy Sched-B starts with a task $T_{mod}(p, i), i \neq 1$. Due to the precedence constraint, $T_{mod}(p, i)$ cannot start until after time $EFT(i)$ ($\geq EFT(1)$). That means processor P will be idle during the period of $LAG := EFT(i) - EFT(1)$. Thus the amount of time to finish all the T_{mod} s will be at least $LAG + \sum_{i=1}^{k} tmod(p, i)$.

On the other hand, in Sched-A, at least some $T_{mod}(p, j), j < i$ have been scheduled in the time period LAG. Hence the amount of work left after time $EFT(i)$ is less than the work left when using Sched-B. Sched-A will give shorter finishing time than Sched-B. □

We are now ready to simulate parallel computation for the whole factorization. To begin with, the ESTs of the leaf panels in the elimination tree are initialized to zero. Various times ($tmod$ and $tdiv$ in floating-point operations) can be computed successively from the bottom of the elimination tree to the top. By applying the argument above inductively to all the panels in the DAG, with leaf panels as the basis, we can show that EFT(root panel) gives the minimum execution time. The (predicted) optimal speedup can then be computed by

$$\text{Predicted speedup} = \frac{\text{Total flops}}{EFT(\text{root panel})} \ .$$

There are several points worth noting in this model. First, because of numerical pivoting, we do not know the computational DAG in advance of the factorization; rather, the DAG is built incrementally as the factorization proceeds. Also, the floating-point operations associated with all the tasks are calculated on the fly. So this model gives an *a posteriori* estimate. Secondly, for each panel computation, the scheduling method of Sched-A requires sorting the EFT's of all the descendant supernodes that will update this panel. The cost associated with this sorting is prohibitively high, and so this method cannot be used to schedule panel updates in practice. However, we content ourselves with bounding the theoretically attainable speedup.

5.8.3 Results from the model

In this subsection, we present the optimal speedups predicted by the model for all of our test problems. The degree of parallelism (and hence speedup) is strongly dependent on the granularity of the sequential tasks. In our algorithm, there are two parameters to control task granularity: The panel size w determines the amount of work in a T_{div} task, and both w and the maximum supernode size $maxsup$ determine the amount of work in a T_{mod} task. Any large supernode of size exceeding $maxsup$ (such as in a dense matrix) is divided into smaller ones so that they fit in cache.

Table 5.14 reports the predicted speedups when varying w and $maxsup$. For a fixed value of $maxsup$, the simulated speedups decrease with increasing w. For sequential SuperLU we find empirically that the best choice for w is between 8 and 16, depending on matrices and architectures. In the parallel setting, a smaller w, say between 4 and 8, may give the best overall performance. This embodies an interesting trade-off between available concurrency and per-processor efficiency.

We now compare the results when fixing w but varying $maxsup$. In relatively sparser matrices, such as matrices 1 – 10, the actual sizes of supernodes may be much smaller than $maxsup$. The performance of such matrices are not so sensitive to $maxsup$. However, for larger and denser matrices, larger value of $maxsup$ results in poorer speedup.

Finally we note that the speedups for small matrices are very low, even with small values of w and $maxsup$. Fortunately, for large matrices such as 13 – 21, the predicted speedups are greater than 20 when $w = 8$ and $maxsup = 32$. These matrices perform more than one billion floating-point operations in the factorization. It is these matrices that require parallel processing power. The current 1-D algorithm is well suited for most of the commercially popular SMP machines, because the number of processors on these systems is usually below 20.

The height of the column etree can also be used as a crude prediction of the parallel performance. The height of a node i is defined as

$$height(i) = \begin{cases} 0, & \text{if } i \text{ is a leaf node} \\ 1 + \max\{\, height(j) \mid j \in child(i)\} & \text{otherwise} \end{cases}$$

The height of the etree is the height of the root, which represents the longest (critical) path in the etree. The computation of all the nodes along this path must be performed in succession. Therefore, the length of the critical path is a constraining factor for performance.

	Matrix	$maxsup = 32$ $w = 4$	$w = 8$	$w = 16$	$maxsup = 64$ $w = 4$	$w = 8$	$w = 16$	$height/n$
1	MEMPLUS	4.8	3.6	2.8	2.9	2.5	2.1	0.95
2	GEMAT11	7.3	5.3	4.1	6.4	4.9	3.6	0.06
3	RDIST1	4.6	3.2	2.1	4.6	3.2	2.1	0.99
4	ORANI678	42.2	28.4	16.6	42.2	28.4	16.6	0.64
5	MCFE	6.6	4.3	2.6	6.6	4.3	2.6	0.67
6	LNSP3937	23.2	15.4	9.7	23.2	15.4	9.7	0.25
7	LNS3937	24.1	15.8	9.6	22.9	15.3	9.6	0.27
8	SHERMAN5	15.8	11.4	7.5	14.0	10.7	7.2	0.20
9	JPWH991	13.4	9.7	6.4	11.3	8.3	6.0	0.46
10	SHERMAN3	12.7	9.7	7.0	8.2	6.9	5.5	0.20
11	ORSREG1	14.4	11.0	7.5	9.2	7.8	5.9	0.34
12	SAYLR4	19.8	16.1	11.0	13.1	11.4	8.6	0.29
13	SHYY161	47.9	36.2	24.1	28.1	23.8	18.1	0.04
14	GOODWIN	97.4	71.3	43.6	83.4	63.4	40.1	0.19
15	VENKAT01	22.0	20.2	17.0	14.3	14.2	13.1	0.73
16	INACCURA	62.6	43.5	26.0	44.5	33.6	22.2	0.45
17	BAI	70.9	55.3	37.2	41.4	35.7	27.4	0.20
18	DENSE1000	33.1	23.7	18.4	18.2	14.9	12.7	1.00
19	RAEFSKY3	140.2	110.6	80.8	80.4	69.6	56.5	0.21
20	EX11	106.7	83.5	58.2	61.6	53.2	41.7	0.35
21	WANG3	57.6	43.4	29.4	34.3	28.9	22.1	0.94
22	RAEFSKY4	99.1	77.1	52.0	56.3	48.5	37.3	0.33
23	VAVASIS3	176.5	133.9	90.7	106.2	89.5	68.2	0.18

Table 5.14: Optimal speedup predicted by the model, and the column etree height.

The last column of Table 5.14 shows the height of the etree over total numbers of nodes n in the etree. The larger is $height/n$, the larger the fraction of panels will be factorized in pipelined manner, resulting in poor parallelism and more synchronizations. For example, $height/n$ for matrices 1, 3, 15 and 21 are rather large. This is consistent with the relatively lower predicted speedups. However, we must note that the etree height alone is not an accurate measure of parallelism. For example, both dense matrix (18) and a tridiagonal matrix have $height/n = 1.00$, but the former possesses much more concurrency than the later.

The fundamental problem is due to over-prediction of the nonzeros when using A^TA as the analysis tool. The consequences of the over-prediction are: (1) The column etree is tall, and contains substantial false dependencies. (2) The dynamic storage scheme (Section 5.4.3) is needed to store supernodes, because the static storage bound is too loose (Table 5.3). Using the dynamic scheme increases the sequential runtime. For example, for matrix 15, the runtime increases by about 15%. In the parallel algorithm, this overhead

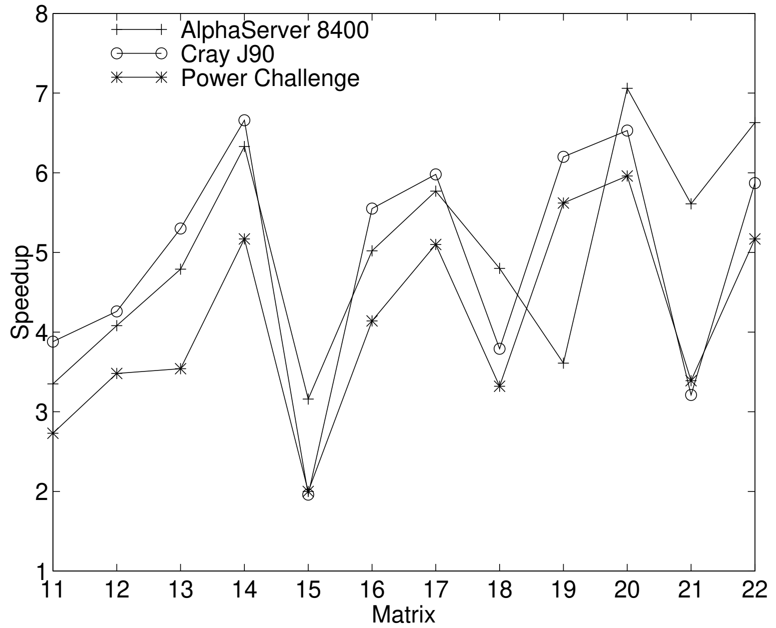


Figure 5.18: Speedups on 8 processors of the Power Challenge, the AlphaServer 8400 and the Cray J90.

also occurs on the critical path, and increases the length of it. So the combined effect gives poor performance on the real machines. On the 8-CPU Power Challenge and AlphaServer 8400, matrix 15 achieves only 2-fold and 3-fold speedups, respectively.

5.9 Conclusions

We have designed and implemented a parallel algorithm for modest size shared memory multiprocessors. The efficiency of the algorithm has been demonstrated on several parallel machines. Figure 5.18 shows the speedups on 8 processors of the three parallel machines. Figures 5.19 through 5.22 recall the factorization rate in Megaflops for six large matrices, with increasing number of processors. We believe these large problems are the primary candidates to be solved on parallel machines. In fact, the largest one in our test suite takes a little more than 0.5 GBytes memory, far less than most parallel machines have offered. Our algorithm is expected to work well for even larger problems.

For a realistic problem arising from a 3-D flow calculation (matrix 20), on the Power Challenge, the Cray C90 and J90, our parallel algorithm achieves 25% peak floating-point performance. On the AlphaServer 8400, it achieves 17% of the peak for the same problem. The respective Mflop rates are 1002, 2583, 831 and 781. These are the fastest results for the unsymmetric *LU* factorization on these powerful high-performance machines. Previous results showed much lower factorization rate because the machines used were relatively slow and the computational kernel in the earlier parallel algorithms was based on Level 1 BLAS. The closest work is the parallel symmetric pattern multifrontal factorization by Amestoy and Duff [5], also on shared memory machines. However, that approach may result in too many nonzeros and so is inefficient for unsymmetric pattern sparse matrices.

Another contribution is providing detailed performance analysis and modeling for the underlying algorithm. In particular, we identified the three main factors limiting parallel performance: (1) contention for accessing critical sections, (2) processors sitting idle due to pipeline waiting, and (3) the need to sacrifice some per-processor efficiency in order to gain more concurrency. Which factor plays more significant role depends on the relative performance of integer and floating-point arithmetic in the underlying architecture.

We have developed a theoretical model to analyze our parallel algorithm and predict the optimally attainable speedup. When comparing the theoretical prediction (Table 5.14) with the actual speedups (Figure 5.18), we find that there exists a discrepancy between the two. This is because our hypothetical machine and the optimal scheduling used in the model do not capture all the details of a real machine with real scheduling. Nevertheless, we do see a similar shape of curves in the predicted and actual speedups. That is, for the matrices predicted lower speedups, such as 11, 15, 18 and 21, the actual speedups are also lower. The model is a useful tool to help identify the inherently sequential problems with bad column orderings. The model also suggests that the panel-wise parallel algorithm, although efficient on small scale SMPs, cannot effectively utilize more than 50 processors. Our future research is to develop a more scalable algorithm for massively parallel machines.

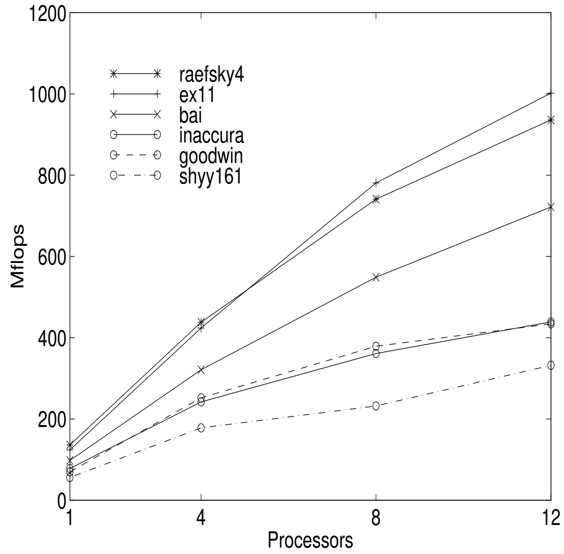


Figure 5.19: Mflop rate on a SGI Power Challenge.

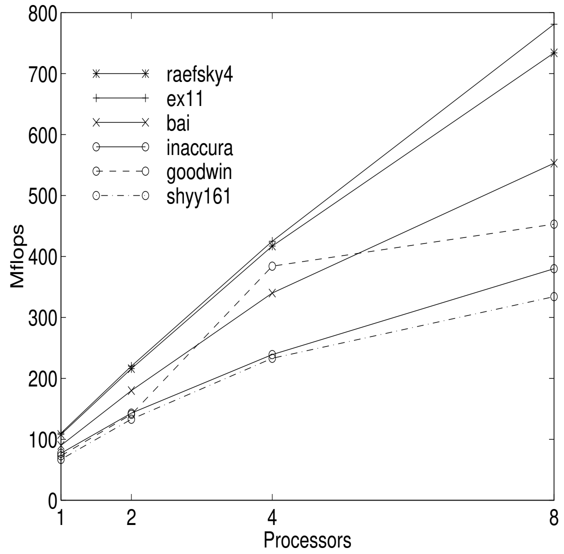


Figure 5.20: Mflop rate on a DEC AlphaServer 8400.

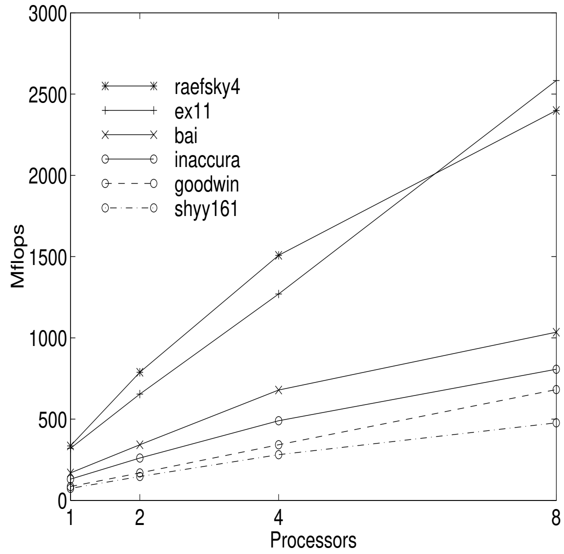


Figure 5.21: Mflop rate on a Cray C90.

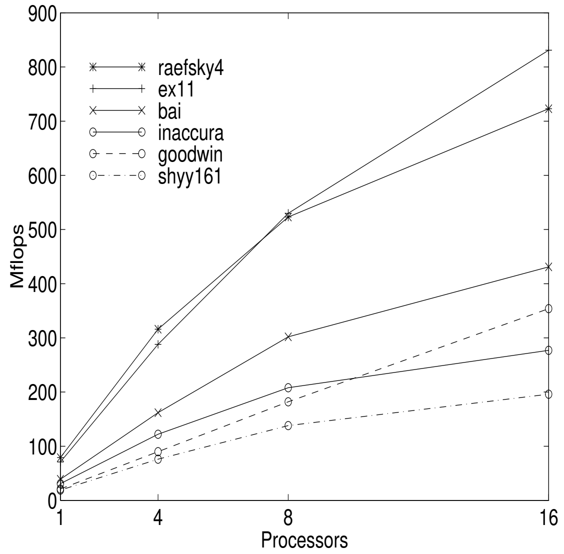


Figure 5.22: Mflop rate on a Cray J90.

Chapter 6

Conclusions and Future Directions

6.1 Summary of contributions

The main goal of this dissertation is to design, implement and analyze new techniques for sparse LU factorization of large unsymmetric matrices, and to show that the proposed methods achieve high performance on a wide range of modern architectures. In addition to algorithm design, a large part of the research lies in performance analysis and modeling, taking into account the characteristics of algorithms, computer architectures and input matrices.

Many high performance computers, the so-called superscalar architectures, have multiple pipelined functional units. Fast but relatively small cache memory is essential to support this functional parallelism. To effectively utilize this level of parallelism, the algorithm must be structured in such a way that a piece of data in registers and cache is reused sufficiently often. To this end, we designed the supernode-panel factorization algorithm in Chapter 4. Although panel factorization, as opposed to column factorization, was adopted in dense LU factorization several years ago, it is much more complicated to implement for sparse matrices. Our supernode-panel algorithm is evaluated on several cache-based superscalar machines, including IBM RS/6000-590, MIPS R8000 and DEC Alpha 21164.

The amount of performance gain over older algorithms is very much dependent on the matrix characteristics. We show that, among all matrix properties, *ops-per-nz*, has the strongest predictive power for performance gain. *Ops-per-nz* is the average number of floating point operations per nonzero in the filled matrix F, which is an upper bound on the maximum cache reuse. When the matrices have a large *ops-per-nz*, the speedups of our supernode-panel algorithm over an earlier supernode-column algorithm are more than four-fold on the MIPS R8000 and more than two-fold on the Alpha 21164. The raw factorization rates are up to 169 Mflops and 121 Mflops, respectively (see Table 4.13).

Based on our efficient supernode-panel algorithm, we developed a parallel algorithm on shared memory machines. The major difficulties are dealing with data dependencies and memory management. We designed a low overhead scheduler to dynamically schedule panel tasks on free processors. A pipeline mechanism is incorporated into the scheduler to coordinate dependent tasks. We used the nonzero structure of the Householder matrix from the QR factorization as an upper bound on the supernode storage for the LU factorization, in order to preallocate enough storage to store supernodes contiguously in memory and so to exploit locality. When this upper bound is too loose, we use a dynamic storage scheme to mitigate storage inefficiency, at the expense of some runtime overhead. Overall, our parallel algorithm is practical and easily portable across different platforms.

Understanding and predicting performance for the parallel algorithm is much more difficult than for the serial algorithm mainly because there are so many interacting factors depending on the algorithm, the machine and the matrices. Among these factors are the cost of locking, the relative speeds of integer and floating-point arithmetic, the memory organization, and the matrix characteristics. Depending on the relative speeds of the above system components, we observe different speedups on different parallel computers, even for the same input matrix (Figure 5.18). We have been able to quantify the major overheads of the algorithm on different machines. One interesting trade-off in the parallel algorithm is to use tasks of smaller granularity (and so with less potential cache reuse) in order to achieve more concurrency. On cache-based machines, such as Power Challenge and AlphaServer 8400, this may cause nontrivial performance degradation on individual processors. This trade-off is not relevant on the Cray C90 and J90 because these machines do not have caches and have much better memory performance. We also developed a theoretical model to predict optimal speedup, irrespective of the architectural details.

For matrices exhibiting sufficient parallelism, the parallel algorithm achieves up to 7-fold speedup on a 12-CPU Power Challenge, 7-fold speedup on an 8-CPU AlphaServer 8400, 6-fold speedup on an 8-CPU Cray C90, and 12-fold speedup on a 16-CPU Cray J90. All speedups are obtained when comparing with the best sequential runtime.

6.2 Future research directions

6.2.1 Sequential algorithm

Further improvements in the sequential algorithm are possible. These improvements are more likely to come from better symbolic algorithms than from the numerical part. Improvement in the symbolic part is especially important for the machines with relatively slower integer performance, such as the Cray C90 and J90.

It may be worthwhile to switch to a dense LU code at a late stage of the factorization. The dense code does not spend time on symbolic structure prediction and pruning, thus streamlining the numeric computation. It can also use BLAS-3 naturally. Eliminating symbolic computation is especially important for vector machines like Crays, because the symbolic part is hard to vectorize and runs relatively slowly. We believe that, for large matrices, the final dense submatrix will be big enough to make the switch beneficial. For example, for a 2-D $k \times k$ square grid problem ordered by nested dissection, the dimension of the final dense submatrix is $\frac{3}{2}k \times \frac{3}{2}k$; for a 3-D $k \times k \times k$ cubic grid, it is $\frac{3}{2}k^2 \times \frac{3}{2}k^2$, if pivots come from the diagonal. The Harwell code MA48 [33, 39] employs such a switch to dense code, which has a significant beneficial effect on performance.

Where is a good point to switch to dense LU? Since our algorithm is left-looking, we do not know exactly when the trailing submatrix becomes dense or nearly so. We considered using the nonzero upper bound approach discussed in Section 5.4.3. Again, we can use two matrices: the Householder matrix H (in factored form) of the QR factorization and the Cholesky factor L_c of A^TA. We know that L is contained in both H and L_c, and that H generally gives a tighter bound. Let l denote the first column of the last supernode of H or of L_c. It is reasonable to assume that the reduced submatrix $F(l:n, l:n)$, corresponding to the last supernode in either H or L_c is fairly dense. Thus, we may use a dense code when eliminating the variables from l to n. To support our argument, we gathered some statistics about the last supernode, in Figure 6.1 (for H) and Figure 6.2 (for L_c). In both figures, we plot the "density" of the last supernode, defined as $\frac{nnz(F(l:n,l:n))}{(n-l+1)^2}$, the percentage of the total floating-point operations performed when eliminating the last $n-l+1$ variables ("Flops %"), and the percentage of the variables to be eliminated by a dense code ("$(n-l+1)/n$"). If we use last supernode in H as switching criterion, $F(l:n, l:n)$ is usually more than 80% full, except for small matrices in which the "dense" part is rather small. On the other hand, if we use L_c, we will need to do more floating-point operations for more zeros. Even though these floating-point operations are cheaper in a dense code, it is not clear how the overall runtime is affected. For the large matrices 19 − 23, which is where SuperLU shows clear advantage, about the last 10% of the rows and columns during elimination are nearly dense. So those problems will probably benefit more from switching to dense LU.

Significant improvements may come from better column preordering heuristics that suffer less fill. These include column nested dissection, or a hybrid of column minimum degree and column nested dissection orderings.

6.2.2 Parallel algorithm

Finding a better elimination tree

According to our theoretical model (Section 5.8), for the input matrices we considered, the current algorithm is not likely to scale up to larger parallel machines with over a hundred processors. There are two different solutions to overcome this obstacle.

One reason for poor (optimal) speedups of some problems is due to the long (inherently sequential) critical path in the column etree, which is determined by a particular column preordering. An optimal ordering with respect to storage or arithmetic may not be necessarily optimal for parallel complexity. Since we use the column etree to drive the parallelism, it is desirable to have a short and wide etree. It is commonly accepted that minimum degree ordering, either on A^TA or $A + A^T$, tends to produce tall and narrow etrees. In the context of sparse Cholesky factorization, Liu [82] developed a reordering scheme to reduce the height of the etree (relabeling nodes in $G(A)$ as well). He used a two-step modular approach to ordering a matrix. He first applied a fill-reducing heuristic (such as minimum degree ordering) to the original matrix A, resulting in a permuted matrix $\bar{A}$. In the second step, he applied a sequence of his etree rotation operations [81] to restructure the etree so that its height is reduced to nearly the minimum. The permutation P found in the second step has the important property that the reordered matrix $P\bar{A}P^T$ suffers the same fill and requires the same number of arithmetic operations for its factorization as does $\bar{A}$. An ordering with this property is called an *equivalent reordering* in the literature. Therefore, the parallel completion time will be reduced when factoring $P\bar{A}P^T$, provided

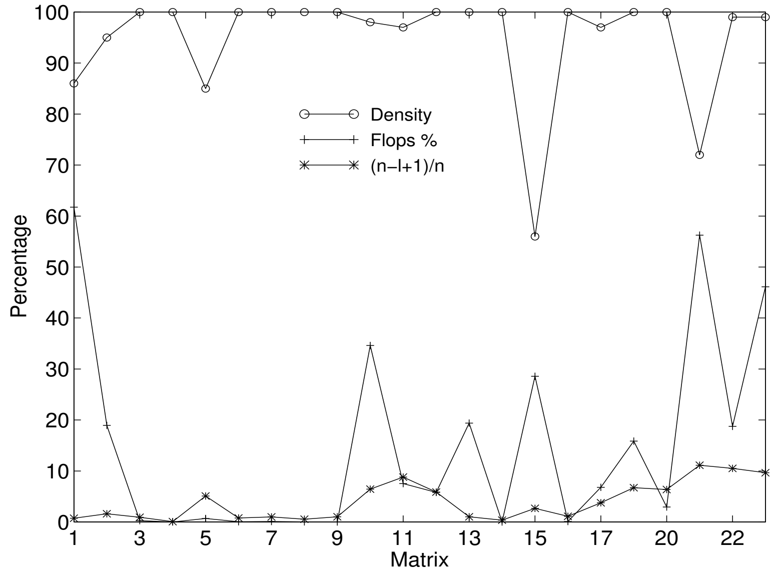


Figure 6.1: Statistics of the last supernode in H.

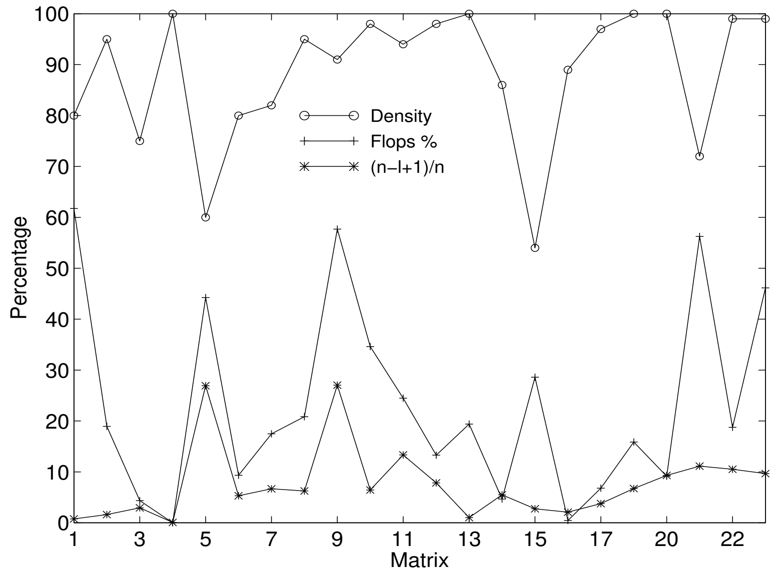


Figure 6.2: Statistics of the last supernode in L_c.

there are enough processors.

This idea may be extended to deal with the unsymmetric case, where the column etree is restructured, and the columns of A are permuted accordingly. However, without knowledge of the numerical values, it is impossible to determine whether a permutation will preserve fills and arithmetic operations for the LU factorization, so the term equivalent reordering is not well defined. At best, the same semantics of equivalent reordering may be used but applied to the Cholesky factor L_c of A^TA. This only says that the *upper bounds* of the fills and arithmetic on L and U are the same (Theorem 5 in Section 5.4.3), with no guarantees for L and U themselves. George and Ng [57] employed this technique in their parallel sparse Gaussian elimination algorithm. Their implementation makes use of the static data structure $\bar{L}$ and $\bar{U}$ obtained from a symbolic row merge algorithm. (In structure, $\bar{L}$ and $\bar{U}$ are identical to H and R respectively, by Theorem 6 in Section 5.4.3, and are upper bounds on L and U.) As they pointed out, if the structure of the column etree is changed, the number of nonzeros in $\bar{U}$ is preserved if A is irreducible, but the number of nonzeros in $\bar{L}$ may not be preserved. They did not report whether restructuring the etree improved the efficiency of their parallel algorithm.

If A is reducible, the upper bound may be very loose. After the reordering, the actual number of nonzeros in L and U and the floating-point operations may become more or fewer than before. However, the parallel runtime may possibly be decreased even though the algorithm performs more operations, simply because more concurrency is exposed. This is a promising area deserving further investigation. It is worth noting that our performance model established in Section 5.8 can be a useful tool to assess whether restructuring the etree (or some other reordering heuristic) will be effective in parallel runtime reduction. This is in fact one of our motivations for building the theoretical model in the first place.

We may also use the etree defined by $\bar{U}$ [57] instead of the column etree, which would present more concurrency than does the column etree. But we should note that it is more expensive to compute the etree of $\bar{U}$ ($O(nnz(\bar{L}) + nnz(\bar{U}))$) than to compute the column etree (almost linear in $nnz(A)$).

Using 2-D decomposition instead of 1-D decomposition

Another remedy, which we believe will be more effective than simply restructuring the etree, is to parallelize the computation along both row and column dimensions of the matrix.

Schreiber [101] modeled the lower bounds on parallel completion time of a left-looking column-oriented sparse Cholesky factorization, and concluded that a two-dimensional mapping is needed to achieve better scalability. Since then, several researchers [72, 97] have developed and demonstrated efficient and scalable 2-D distributed algorithms for sparse Cholesky. To summarize their results, there are two essential ingredients: (1) computational kernels are based on BLAS 3 in order to achieve high per-processor performance; (2) the matrix is partitioned in a 2-D fashion and mapped onto processors in a 2-D grid. In this mapping, the machine is organized as $P \times Q$ processor grid. A block row of the matrix is mapped to the same row of the processor grid, and a block column of the matrix is mapped to the same column of the processor grid. Compared with the 1-D mapping, asymptotically, both the length of the critical path and the interprocessor communication volume are reduced for a grid model problem [97].

Unlike sparse Cholesky, the following issues must be addressed in the LU factorization: (1) supernodes (blocks) emerge dynamically, but we need to determine the block boundaries and distribute matrix prior to factorization; (2) we need to parallelize the underlying symbolic algorithm to accommodate dynamic structural change; (3) processors must cooperate to perform numerical pivoting at each step.

We propose the following strategies to address the above issues.

- **block partition.** We will use the supernode boundaries in the Householder matrix H (Section 5.4.3) to partition matrix A into blocks of columns. If a block is too large, we further divide it into smaller blocks. Then, we can apply the same block partitioning to the rows of matrix A.

- **block mapping.** The global 2-D block cyclic mapping successfully used in dense algorithms [19] may cause serious load imbalance. Instead, we propose a two-phase mapping method as follows. First, we will use the column etree and arithmetic estimate based on the Householder matrix H to find independent domains and assign them to individual processors. We discussed this method in Section 5.7.1. Secondly, at the higher level of the etree outside domains, we will use a Cartesian product mapping heuristic proposed by Rothberg and Schreiber [98]. Here, we can estimate the work associated with each block using the two upper bound matrices H and L_c (Section 5.4.3). One important observation is that the mapping functions for rows and columns can be defined independently. Compared with the customary 2-D block cyclic mapping, there are two advantages associated with the independent row and column mappings: (1) there is more flexibility to statically balance the work load; (2) it can avoid heavily loading the diagonal processors in the processor grid, since the diagonal blocks tend to have more work than the off-diagonal blocks and they are now mapped to not only the diagonal processors but also the off-diagonal ones.

- **symbolic algorithm.** Although the current symbolic algorithm is very efficient in the sequential and 1-D parallel codes, it may become a performance bottleneck in the 2-D algorithm, because depth-first search does not exploit locality and is hard to parallelize. We are investigating alternative algorithms to perform structure prediction that use more localized information.

It should be noted that the 2-D algorithm proposed here is mainly to address scalability and is targeted at massively parallel machines, including distributed memory machines and clusters of SMPs. We do not expect it to replace the 1-D algorithm developed in Chapter 5 for small-scale SMPs, because the 2-D algorithm requires more synchronizations and requires more complicated data structures, and will be less efficient for small-scale SMPs.

6.2.3 Parallel triangular solves

Parallel sparse triangular solves usually attract less attention than factorization, simply because they usually perform many fewer floating-point operations and require much less time in sequential code (Figure 4.23). However, once the runtime of LU factorization is significantly reduced by parallelization, triangular solves represent a larger fraction of the total runtime. It therefore becomes more important to parallelize this phase as well, especially if there are multiple right-hand sides in the equations.

The parallel strategy is very similar to the parallel factorization algorithm. Again, we can employ the column etree to guide the parallelization, with forward substitution proceeding from the bottom of the etree upward, and back substitution proceeding in top-down fashion.

6.3 Available software

As stated earlier, the performance of a sparse code depends not only on the algorithm and architecture, but also on matrix properties, such as dimension, density, structural symmetry, etc. Some comparisons indicate that no single code or algorithm performs best for all classes of problems and for all machines. For example, we demonstrated that supernodal techniques (SuperLU) based on dense matrix kernels are very efficient for large problems, on both serial and parallel machines with superscalar or vector hardware. But for small or extremely sparse matrices, the earlier and simpler codes based on BLAS-1 kernels may be as efficient as or more efficient than SuperLU.

In this section we give an overview of the sparse codes developed recently. Our purpose is not to give a complete survey or comparison of all the sparse *LU* codes; rather, we emphasize *functionality* and *availability* of the codes. We hope that this section may serve as a brief guide for users to choose the appropriate code according to their problems nature and solution environments. So we only include the codes that are either publically available, or likely to be available from the authors.

Table 6.1 tabulates these codes. Here we simply highlight the key algorithmic features of each code. For more details and performance issues, we refer readers to the original references and a recent survey by Duff [34]. Besides sparse *LU* factorizations, Duff also summarized many other advances in sparse numerical linear algebra, including ordering, linear least-squares, and preconditioning. The new book by Björck [17] contains a complete list of algorithms and software for sparse least-squares problems.

The last colum in the table shows the availability of each code. All the serial codes are publically available, and are portable to a majority of uniprocessor platforms. Shared memory codes have achieved reasonable success in portability. Even if a code is developed on one system, it is usually not a quite difficult task to move it onto another parallel machine. For distributed memory machines, most codes are still at the research stage, and not so publically accessible. Each code usually works only on one parallel machine. So for distributed memory machines, much work remains to develop reliable, portable, and high performance sparse direct solvers. (This is in contrast to dense matrix problems, for which the ScaLAPACK library is available [19].)

In the future, it will be worthwhile to conduct direct comparisons and evaluations of some of these codes on the same machines and for the same input matrices.

Matrix Type	Name	Algorithm	Numerical Kernel	Status /Source
Serial Algorithms				
unsym.	SuperLU	LL, partial	BLAS-2.5	Pub/UCB
unsym.	UMFPACK [21, 22]	MF, Markowitz	BLAS-3	Pub/netlib
	MA38 (same as UMFPACK)			Com/HSL
unsym.	MA48 [39]	Anal: RL, Markowitz		Com/HSL
		Fact: LL, partial	BLAS-1, SD	
unsym.	SPARSE [79]	RL, Markowitz	Scalar	Pub/netlib
sym-pattern	MA41 [4]	MF, threshold	BLAS-3	Com/HSL
sym-pattern	MA42 [42]	Frontal (eqn+element)	BLAS-3	Com/HSL
sym.	MA27 [40]	MF, LDL^T	BLAS-1	Com/HSL
sym.	MA47 [38]	MF, LDL^T	BLAS-3	Com/HSL
s.p.d.	Ng & Peyton [89]	LL	BLAS-3	Pub/Author
Shared Memory Algorithms				
unsym.	SuperLU	LL, partial	BLAS-2.5	Pub/UCB
unsym.	PARASPAR [112, 113]	RL, Markowitz	BLAS-1, SD	Res/Author
sym-pattern	MUPS [6]	MF, threshold	BLAS-3	Res/Author
unsym.	George & Ng [57]	RL, partial	BLAS-1	Res/Author
s.p.d.	Gupta, Rothberg, Ng & Peyton [73]	LL	BLAS-3	Com/SGI Pub/Author
s.p.d.	SPLASH [78]	RL, 2-D block	BLAS-3	Pub/Stanford
Distributed Memory Algorithms				
unsym.	van der Stappen [108]	RL, Markowitz	Scalar	Res/Author
sym-pattern	Lucas et al. [85]	MF, no pivoting	BLAS-1	Res/Author
s.p.d.	Rothberg et al. [98]	RL, 2-D block	BLAS-3	Res/Author
s.p.d.	Gupta [72]	MF, 2-D block	BLAS-3	Res/Author
s.p.d.	CAPSS [74]	MF, full parallel (require coordinates)	BLAS-1	Pub/netlib

Table 6.1: Software to solve sparse linear systems using direct methods.

Abbreviations used in the table:

unsym. – fully unsymmetric matrices
sym-pattern – unsymmetric matrices with symmetric nonzero patterns
sym. – symmetric but possibly indefinite matrices
s.p.d – symmetric positive definite matrices
MF, LL and RL – multifrontal, left-looking and right-looking, respectively
SD – switches to a dense code on a sufficiently dense trailing submatrix
Pub – publically available and the authors may be willing to supply the code
Res – published in literature but may not be available from the authors
Com – commercial
HSL – Harwell Subroutine Library
(http://www.rl.ac.uk/departments/ccd/numerical/hsl/hsl.html)
netlib – http://www.netlib.org; netlib@www.netlib.org
UCB – http://www.cs.berkeley.edu/~xiaoye/superlu.html
Stanford – http://www-flash.stanford.edu/apps/SPLASH/

Bibliography

[1] R.C. Agarwal, F.G. Gustavson, P. Palkar, and M. Zubair. A performance analysis of the subroutines in the ESSL/LAPACK call conversion interface (CCI). IBM T.J. Watson Research Center, Yorktown Heights, 1994.

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM J. Computing*, 1:131–137, 1972.

[3] P. R. Amestoy, T. A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications*, 1996. To appear. (Also University of Florida TR-94-039).

[4] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *The International Journal of Supercomputer Applications*, 3:41–59, 1989.

[5] P. R. Amestoy and I.S. Duff. MUPS: a parallel package for solving sparse unsymmetric sets of linear equations. Technical report, CERFACS, Toulouse, France, 1994.

[6] Patrick R. Amestoy. Factorization of large unsymmetric sparse matrices based on a multifrontal approach in a multiprocessor environment. Technical Report TH/PA/91/2, CERFACS, Toulouse, France, February 1991. Ph.D thesis.

[7] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 2.0.* SIAM, Philadelphia, 1995. 324 pages.

[8] E. Anderson et al. *LAPACK User's Guide, Second Edition.* SIAM, Philadelphia, 1995.

[9] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.*, 10(2):165–190, April 1989.

[10] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM Journal on Scientific Computing*, 16:1404–1411, 1995.

[11] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Mathematical Software*, 15:291–309, 1989.

[12] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in sparse matrix methods for large sparse linear systems on vector supercomputers. *Intern. J. of Supercomputer Applications*, 1:10–30, 1987.

[13] C. Ashcraft and J. Liu. Robust ordering of sparse matrices using multisection. Technical Report ISSTECH-96-002, Boeing information and support services, 1996.

[14] Cleve Ashcraft, S.C. Eisenstat, Joseph Liu, and A.H. Sherman. A comparison of three column-based distributed sparse factorization schemes. Technical Report Research Report YALEU/DCS/RR-810, Yale University, Department of Computer Science, July 1990.

[15] S. T. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In R. F. Sincovec, D. Keyes, M. Leuze, L. Petzold, and D. Reed, editors, *Sixth SIAM conference on parallel processing for scientific computing*, pages 711–718, 1993.

[16] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.-W. Chin. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ansi c coding methodology. Computer Science Dept. Technical Report CS-96-326, University of Tennessee, Knoxville, May 1996. (LAPACK Working Note #111).

[17] Å. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, 1996.

[18] T. Chan, J. Gilbert, and S.-H. Teng. Geometric spectral partitioning. Technical Report CSL-94-15, Palo Alto Research Center, Xerox Corporation, California, 1994.

[19] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - Design issues and performance. Computer Science Dept. Technical Report CS-95-283, University of Tennessee, Knoxville, March 1995. (LAPACK Working Note #95).

[20] Thomas F. Coleman, Anders Edenbrandt, and John R. Gilbert. Predicting fill for sparse orthogonal factorization. *Journal of the Association for Computing Machinery*, 33:517–532, 1986.

[21] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report RAL 93-036, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, 1994.

[22] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. Technical Report TR-95-020, Computer and Information Sciences Department, University of Florida, 1995.

[23] Timothy A. Davis. User's guide for the unsymmetric-pattern multifrontal package (UMFPACK). Technical Report TR-93-020, Computer and Information Sciences Department, University of Florida, June 1993.

[24] Timothy A. Davis, John R. Gilbert, Esmond Ng, and Barry Peyton. Approximate minimum degree ordering for unsymmetric matrices. Talk presented at XIII Householder Symposium on Numerical Algebra, June 1996. Journal version in preparation.

[25] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W.H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, U.C. Berkeley, July 1995. (LAPACK Working Note #103).

[26] Guide to DECthreads. Digital Equipment Corporation, July 1994.

[27] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[28] J. Dongarra, J. Du Croz, Duff I., and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[29] J. Dongarra, F. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, January 1984.

[30] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of basic linear algebra subroutines. *ACM Trans. Mathematical Software*, 14:1–17, 18–32, 1988.

[31] I. S. Duff, R. Grimes, and J. Lewis. Sparse matrix test problems. *ACM Trans. Mathematical Software*, 15:1–14, 1989.

[32] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM J. Scientific and Statistical Computing*, 5:633–641, 1984.

[33] I. S. Duff and J. K. Reid. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Technical Report RAL–93–072, Rutherford Appleton Laboratory, Oxon, UK, 1993.

[34] Iain S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. Technical Report RAL-TR-96-047, Rutherford Appleton Laboratory, 1996.

[35] I.S Duff. Multiprocessing a sparse matrix code on the Alliant FX/8. *Journal of computational and applied mathematics*, 27:229–239, September 1989.

[36] I.S. Duff, I.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, London, 1986.

[37] I.S Duff, R.G Grimes, and J.G Lewis. Users' guide for the harwell-boeing sparse matrix collection (release 1). Technical Report RAL-92-086, Rutherford Appleton Laboratory, December 1992.

[38] I.S Duff and J. K. Reid. MA47, a Fortran code for direct solution of indefinite sparse symmetric linear systems. Technical Report RAL-95-001, Rutherford Appleton Laboratory, 1995.

[39] I.S Duff and J. K. Reid. The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Mathematical Software*, 22:187–226, 1996.

[40] I.S Duff and J.K Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9(3):302–325, September 1983.

[41] I.S. Duff and J.K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Mathematical Software*, 9:302–325, 1983.

[42] I.S Duff and J. A. Scott. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Trans. Mathematical Software*, 22(1):30–45, 1996.

[43] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in sparse unsymmetric symbolic factorization. *SIAM J. Matrix Analysis and Applications*, 13:202–211, 1992.

[44] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM J. Scientific and Statistical Computing*, 14:253–257, 1993.

[45] Stanley C. Eisenstat, John R. Gilbert, and Joseph W.H. Liu. A supernodal approach to a sparse partial pivoting code. In *Householder Symposium XII*, 1993.

[46] David M. Fenwick, Denis J. Foley, William B. Gist, Stephen R. VanDoren, and Daniel Wissel. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

[47] G.A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[48] A. George and D. McIntyre. On the application of the minimum degree algorithm to finite element systems. *SIAM J. Numerical Analysis*, 15:90–111, 1978.

[49] Alan George, Michael T. Heath, Joseph Liu, and Esmond Ng. Solution of sparse positive definitive systems on a shared-memory multiprocessor. *International Journal of Parallel Programming*, 15(4):309–325, 1986.

[50] Alan George, Joseph Liu, and Esmond Ng. A data structure for sparse QR and LU factorizations. *SIAM J. Sci. Stat. Comput.*, 9:100–121, 1988.

[51] Alan George and Joseph W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[52] Alan George and Joseph W. H. Liu. The evolution of the minimum degree ordering algorithms. *SIAM Review*, 31(1):1–19, March 1989.

[53] Alan George, Joseph W. H. Liu, and Esmond Ng. Communication results for parallel sparse cholesky factorization on a hypercube. *Parallel Computing*, pages 287–298, 1989.

[54] Alan George and Esmond Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Sci. Stat. Comput.*, 6(2):390–409, 1985.

[55] Alan George and Esmond Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Stat. Comput.*, 8(6):877–898, 1987.

[56] Alan George and Esmond Ng. On the complexity of sparse QR and LU factorization on finite-element matrices. *SIAM J. Sci. Stat. Comput.*, 9(5):849–861, September 1988.

[57] Alan George and Esmond Ng. Parallel sparse Gaussian elimination with partial pivoting. *Annals of Operation Research*, 22:219–240, 1990.

[58] J. George. Nested dissection of a regular finite element mesh. *SIAM J. Numerical Analysis*, 10:345–363, 1973.

[59] J. A. George and E. Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Scientific and Statistical Computing*, 6:390–409, 1985.

[60] J. R. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Analysis and Applications*, 15:62–79, 1994.

[61] J. R. Gilbert and J. W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Analysis and Applications*, 14:334–352, 1993.

[62] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: Design and implementation. *SIAM J. Matrix Analysis and Applications*, 13:333–356, 1992.

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

[64] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Scientific and Statistical Computing*, 9:862–874, 1988.

[65] John R. Gilbert. An efficient parallel sparse partial pivoting algorithm. Technical Report CMI No. 88/45052-1, Computer Science Department, University of Bergen, Norway, 8 1988.

[66] John R. Gilbert. Predicting structures in sparse matrix computations. *SIAM J. Matrix Analysis and Applications*, 15(1):62–79, January 1994.

[67] John R. Gilbert. Personal communication, 1995.

[68] John R. Gilbert and Joseph W.H. Liu. Elimination structures for unsymmetric sparse lu factors. *SIAM J. Matrix Anal. Appl.*, 14(2):334–352, April 1993.

[69] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. Computing row and column counts for sparse QR factorization. Talk presented at SIAM Symposium on Applied Linear Algebra, June 1994. Journal version in preparation.

[70] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.*, 15:1075–1091, 1994.

[71] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[72] A. Gupta and V. Kumar. Optimally scalable parallel sparse Cholesky factorization. In *Proceedings of the Seventh SIAM Conference on Parallel Proceesing for Scientific Computing*, pages 442–447. SIAM, 1995.

[73] A. Gupta, E. Rothberg, E. Ng, and B. W. Peyton. Parallel sparse Cholesky factorization algorithms for shared-memory multiprocessor systems. In R. Vichnevetsky, D. Knight, and G. Richter, editors, *Advances in Computer Methods for Partial Differential Equations–VII.* IMACS, 1992.

[74] M. T. Heath and P. Raghavan. Performance of a fully parallel sparse solver. In *Proc. Scalable High-Performance Computing Conf.*, pages 334–341, Los Alamitos,CA, 1994. IEEE.

[75] M.T. Heath, E. Ng., and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, September 1991.

[76] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. Technical Report SAND96-0868J, Sandia National Laboratories, Albuquerque, 1996.

[77] *International Business Machines Corporation Engineering and Scientific Subroutine Library, Guide and Reference.* Version 2 Release 2, Order No. SC23-0526-01, 1994.

[78] W-D. Webber J.P. Singh and A. Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, 1992.

[79] Kenneth Kundert. Sparse matrix techniques. In Albert Ruehli, editor, *Circuit Analysis, Simulation and Design.* North-Holland, 1986.

[80] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Analysis and Applications*, 11:134–172, 1990.

[81] Joseph W.H. Liu. Equivalent sparse matrix reordering by elimination tree rotations. *SIAM J. Sci. Statist. Comput.*, 9(3):424–444, May 1988.

[82] Joseph W.H. Liu. Reordering sparse matrices for parallel elimination. *Parallel Computing*, 11:73–91, 1989.

[83] Joseph W.H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, January 1990.

[84] Joseph W.H. Liu, Esmond G. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(1):242–252, January 1993.

[85] Robert F. Lucas, Tom Blank, and Jerome J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):981–991, November 1987.

[86] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.

[87] E. G. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Scientific and Statistical Computing*, 14:1034–1056, 1993.

[88] Esmond G. Ng. Personal communication, 1996.

[89] Esmond G. Ng and Barry W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14(5):1034–1056, September 1993.

[90] Esmond G. Ng and Barry W. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM J. Sci. Comput.*, 14(4):761–769, July 1993.

[91] POSIX System Application Arogram Interface: Threads extension [C Language], POSIX 1003.1c draft 4. IEEE Standards Department.

[92] A. Pothen, H. D. Simon, L. Wang, and S. Barnard. Towards a fast implementation of spectral nested dissection. In *Supercomputing, ACM Press*, pages 42–51, 1992.

[93] A. Pothen and C. Sun. A distributed multifrontal algorithm using clique trees. Tech report CTC91TR72, Cornell Theory Center, Cornell University, August 1991.

[94] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Mathematical Software*, Vol. 16:303–324, 1990.

[95] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations – exploiting the memory hierarchy. *ACM Trans. Mathematical Software*, 17:313–334, 1991.

[96] E. Rothberg and A. Gupta. An evaluation of left-looking, right-looking and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. *Int. J. High Speed Computing*, 5:537–593, 1993.

[97] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Scientific Computing*, 17(3):699–713, May 1996.

[98] Edward Rothberg and Robert Schreiber. Improved load distribution in parallel sparse cholesky factorization. In *Supercomputing*, pages 783–792, November 1994.

[99] Edward E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Dept. of Computer Science, Stanford University, December 1992.

[100] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Trans. Mathematical Software*, 8:256–276, 1982.

[101] R. Schreiber. Scalability of sparse direct solvers. In Alan George, John R. Gilbert, and Joseph W.H. Liu, editors, *Graph theory and sparse matrix computation*, pages 191–209. Springer-Verlag, New York, 1993.

[102] A. H. Sherman. *On the efficient solution of sparse systems of linear and nonlinear equations*. PhD thesis, Yale University, 1975.

[103] A. H. Sherman. Algorithm 533: NSPIV, a FORTRAN subroutine for sparse Gaussian elimination with partial pivoting. *ACM Trans. Mathematical Software*, 4:391–398, 1978.

[104] SGI Power Challenge. Silicon Graphics, 1995. Technical Report.

[105] H. Simon, P. Vu, and C. Yang. Performance of a supernodal general sparse solver on the CRAY Y-MP: 1.68 GFLOPS with autotasking. Technical Report TR SCA-TR-117, Boeing Computer Services, 1989.

[106] Solaris SunOS 5.0 multithread architecture. Sun Microsystems, Inc., November 1991. Technical White Paper.

[107] SPARCcenter 2000 architecture and implementation. Sun Microsystems, Inc., November 1993. Technical White Paper.

[108] A. Frank van der Stappen, Rob H. Bisseling, and Johannes G. G. van der Vorst. Parallel sparse LU decomposition on a mesh network of transputers. *SIAM J. Matrix Analysis and Applications*, 14(3):853–879, July 1993.

[109] S. A. Vavasis. Stable finite elements for problems with wild coefficients. Technical Report 93–1364, Department of Computer Science, Cornell University, Ithaca, NY, 1993. To appear in *SIAM J. Numerical Analysis*.

[110] The Cray C90 series. http://www.cray.com/PUBLIC/product-info/C90/. Cray Research, Inc.

[111] The Cray J90 series. http://www.cray.com/PUBLIC/product-info/J90/. Cray Research, Inc.

[112] Z. Zlatev, J. Waśniewski, P. C. Hansen, and Tz. Ostromsky. PARASPAR: a package for the solution of large linear algebraic equations on parallel computers with shared memory. Technical Report 95-10, Technical University of Denmark, September 1995.

[113] Zahari Zlatev. *Computational methods for general sparse matrices*. Kluwer Academic, Dordrecht; Boston, 1991.