PARALLEL UNSYMMETRIC-PATTEN MULTIFRONTAL SPARSE LU WITH COLUMN PREORDERING

HAIM AVRON, GIL SHKLARSKI, AND SIVAN TOLEDO

Abstract. We present a new parallel sparse LU factorization algorithm and code. The algorithm uses a column-preordering partial-pivoting unsymmetric-pattern multifrontal approach. Our baseline sequential algorithm is based on UMFPACK 4 but is somewhat simpler and is often somewhat faster than UMFPACK version 4.0. Our parallel algorithm is designed for shared-memory machines with a small or moderate number of processors (we tested it on up to 32 processors). We experimentally compare our algorithm with SuperLU_MT, an existing shared-memory sparse LU factorization with partial pivoting. SuperLU_MT scales better than our new algorithm, but our algorithm is more reliable and is usually faster in absolute (on up to 16 processors; we were not able to run SuperLU_MT on 32). More specifically, on large matrices our algorithm is always faster on up to 4 processors, and is usually faster on 8 and 16. The main contribution of this paper is showing that the column-preordering partial-pivoting unsymmetric-pattern multifrontal approach, developed as a sequential algorithm by Davis in several recent versions of UMFPACK, can be effectively parallelized.

1. Introduction

We present a new parallel sparse partial-pivoting LU factorization algorithm. The experience of designers and implementors of sparse LU algorithms has been that a single algorithm usually cannot perform well on machines ranging from uniprocessors to small parallel computers to massively-parallel computers. For example, the SuperLU family of algorithms consists of three different algorithms, one for uniprocessors [14], one for shared-memory multiprocessors [15], and one for distributed-memory multiprocessors [33]. We chose to focus on one class of target machines, shared-memory parallel computers with 1-32 processors.

The factorization of a general matrix into triangular factors often requires some form of pivoting (row and/or column exchanges) in order to avoid numerical instability. Three classes of pivoting techniques have been proposed for sparse LU factorizations. Our algorithm belongs to the class of *partial-pivoting* algorithms. At each elimination step, these algorithms examine the numerical values in the next column to be eliminated, and perform a row exchange that brings a matrix entry with a large absolute value to the diagonal of that column. So-called *static-pivoting* algorithms, such as [33], prepermute the rows to bring large elements to the diagonal.

Static pivoting is a heuristic that may lead to numerical instability because an element that was large in the original matrix may become tiny during the elimination process. However, static pivoting often works well, especially when coupled with iterative refinement. Static pivoting allows more detailed planning of the scheduling of a parallel algorithm, because the row permutation is known before the numerical factorization begins. Finally, *delayed-pivoting* algorithms, such as [30], perform both row and column exchanges during the numerical factorization. These algorithm precompute a column ordering, and for each column, a set of potential pivot rows. During the elimination of a column the algorithm examines the elements in the potential rows that have not been used as pivot rows. If one of them is large enough, a row exchange is performed and the column is eliminated. If all of them are too small, the elimination of the column is delayed. This corresponds to a column exchange. During the column exchange, the set of potential rows for that column is usually expanded.

We chose to use partial pivoting for two reasons. First, partial pivoting, especially when performed strictly (the largest element in absolute value is brought to the diagonal), is numerically very reliable. In particular, static-pivoting algorithms sometimes fail on matrices that partial-pivoting algorithms can factor successfully. Second, partial pivoting without column exchanges allows the algorithm to select a column preordering. Preordering the columns can provide a-priori guarantees on fill [24, 28, 4]; delayed pivoting algorithms provide no such guarantees. Although delayed-pivoting has been shown to work well in practice, in theory the factors may fill completely.

We note that sparse partial-pivoting algorithms have another advantage: they can be implemented so that the total number of operations that they perform is proportional to the number of arithmetic operations required [27] (the number of arithmetic operations depends only on the non-zero structure of the input matrix and of the factors). Our algorithm is not implemented that way. We chose to use data structures for which this property does not necessarily hold, but which lead to faster performance in practice.

The decision to use partial pivoting left us with a choice between two families of algorithms: left-looking and multifrontal. The most sophisticated left-looking algorithm today is SuperLU [14, 16], a followup to earlier algorithms, GP [27] and SupCol [19], all of which use partial pivoting. The most sophisticated unsymmetric multifrontal algorithm today is UMFPACK version 4.x [9]. Several other multifrontal algorithms, like WSMP [30, 31], earlier versions of UMFPACK [6, 7], and MA41U [1], do not combine partial pivoting with column preordering, so they are not relevant to us. We decided to focus on the multifrontal family, for two reasons. First, comparisons between SuperLU and UMFPACK indicate that the latter is often faster, and rarely significantly slower. In particular, comparisons between SuperLU and UMFPACK 4, made by the author of UMFPACK, indicate that it is much faster than SupreLU [9]. Comparisons between SuperLU and UMFPACK 3, made by two teams not associated with either code, indicate that UMFPACK is usually faster [1, 31]. These comparisons motivated us to try to parallelize the partial-pivoting unsymmetric-pattern multifrontal approach. The second reason for choosing a multifrontal approach is that there is already a shared-memory parallel version of SuperLU, called SuperLU_MT [15], so parallelizing UMFPACK would shed additional light on the difference between the two approaches, whereas another parallel left-looking algorithm would probably not contribute much to our understanding.

Can the partial-pivoting unsymmetric-pattern multifrontal algorithm be parallelized, and in particular, would such an algorithm be more effective than a parallel left-looking algorithm? This is the question that our research addresses.

This paper shows that the answer to this question is affirmative. The partial-pivoting unsymmetric-pattern multifrontal algorithm can be parallelized, and the resulting algorithm performs better on small-to-moderate shared-memory multiprocessors than SuperLU_MT.

We have conducted our research in two stages. In the first stage, we designed and implemented a sequential partial-pivoting unsymmetric-pattern multifrontal LU factorization. We refined and tuned the algorithm until it matched or bettered the performance of UMFPACK (under some restrictions that we explain later). In the process of doing so, we have simplified the UMFPACK algorithm fairly significantly, and we have introduced one significant improvement to the sequential algorithm. Obviously, we designed and implemented this sequential version with parallelization in mind. At the end of this stage, our algorithm was not only simpler than UMFPACK, but outperformed it[1] on most of the larger matrices.

In the second stage, we parallelized the algorithm. During this stage we again refined the algorithm, mainly in order to obtain as much parallelism as possible without increasing the total work. Our main benchmark code at this stage was SuperLU_MT. At the end of this stage, our algorithm performed significantly better than SuperLU_MT on most matrices and on most processor numbers up to 32.

The rest of the paper is organized has follows. Section 2 provides some necessary background. Section 3 presents the partial-pivoting unsymmetric-pattern multifrontal algorithm. The material in that section is not new, but the presentation is. Section 4 presents our new algorithm. Extensive experimental results are given in section 5. We present our conclusions in Section 6.

2. Background

This section provides some background material. We begin with a formal description of the LU factorization algorithm with partial pivoting, mainly

[1]These comparisons are with UMFPACK version 4.0. During our research, Tim Davis has produced two additional versions, 4.1 and 4.3; to provide a stable baseline to our research, we kept using version 4.0.

```
[L, U, p] = dense_lu(A)
$A^{(0)} \longleftarrow A$
for $j \longleftarrow 1\colon n$
    $p(j) \longleftarrow \arg\max_{\bar{p}(1\colon j-1)} \left| A^{(j-1)}_{\bar{p}(1\colon j-1),j} \right|$
    $L_{\bar{p}(1\colon j-1),j} \longleftarrow A^{(j-1)}_{\bar{p}(1\colon j-1),j} / A^{(j-1)}_{p(j),j}$
    $U_{j,j\colon n} \longleftarrow A^{(j-1)}_{p(j),j\colon n}$
    $A^{(j)}_{\bar{p}(1\colon j),j+1\colon n} \longleftarrow A^{(j-1)}_{\bar{p}(1\colon j),j+1\colon n} - L_{\bar{p}(1\colon j),j} U_{j,j+1\colon n}$
end
```

FIGURE 2.1. Dense LU factorization with partial pivoting. At the end of the algorithm, L itself is not triangular, but $L_{p(1\colon n),1:n}$ is.

in order to establish the notation. We then define the column elimination tree and state its properties. The last part of the section briefly described the parallel programming language that we use.

2.1. **Dense LU Factorization with Partial Pivoting.**

The algorithm works by factoring one column and one row in every step. We assume that the columns of the matrix have already been preordered. Therefore, column j is always factored in step j. The row that the algorithm factors in step j depends on the numerical values in the reduced matrix. We denote that row by $p(j)$.

We use MATLAB colon notation for contiguous sets of integers, $i\colon j = \{i, i+1, \ldots, j-1, j\}$. For an ordered set s of column indices, we denote by $p(s)$ their map under p. The complement of the row set $p(s)$ is defined to be $\bar{p}(s) = 1\colon n \setminus p(s)$. In particular, the set $p(1\colon j)$ denotes the ordered set of rows that have been factored during steps 1 through j, and $\bar{p}(1\colon j)$ denotes the unordered set of yet-unfactored rows at the end of step j.

Factoring column j and row $p(j)$ corresponds to the elimination of the jth unknown from a linear system of equations using equation $p(j)$. The elimination step expresses the jth unknown as a linear combination of the remaining unknowns, and eliminates j by substituting the symbolic expression for j in all the remaining equations. Therefore, the remaining equations must be updated. The submatrix corresponding to the reduced equations is called the *reduced matrix*, and we denote it by $A^{(j)} = A^{(j)}_{\bar{p}(1\colon j),j+1\colon n}$. The reduced matrix is an $(n-j)$-by-$(n-j)$ matrix, with column indices starting at $j+1$ and with row indices $\bar{p}(1\colon j)$. We also denote $A^{(0)} = A$.

The full algorithm is presented in Figure 2.1 using this notation.

2.2. **Column Preordering.** The rows and columns of a linear system $Ax = b$ are unordered, because the equations and variables are unordered. But when A is sparse, re-ordering the rows and columns of the system prior to the factorization of A can have a dramatic effect on the number of nonzeros in the LU factors of A. However, if the rows and columns are re-ordered arbitrarily, a factorization may not exist, or the algorithm may become unstable. Partial pivoting solves this problem, but it requires that the row permutation be determined dynamically during the factorization. This still allows the algorithm to permute the columns arbitrarily to reduce fill.

There are two approaches to the selection of the column permutation. One approach is to construct the column permutation dynamically during the numerical factorization. In step j, the algorithm first selects the next column $q(j)$ to be factored, and then selects the pivot row. The goal in the selection of $q(j)$ is to produce as little fill as possible in the reduced matrix. Early versions of UMFPACK use this approach [6, 7]. These algorithms maintain an approximation of the number of nonzeros in each row and column of the reduced matrix, and a column with a small approximate nonzero count is selected as $q(j)$ (the exact criteria is more complex, but uses this idea).

In the other approach, a column permutation is computed before the factorization begins. The permutation is typically constructed so as to minimize the fill in the Cholesky factor R of $A^T A$ [4, 24, 28], because the fill in R bounds from above the fill in L and U for any selection of pivot rows. Another popular method, COLAMD [11], uses a heuristic that selects the column ordering using an approximation of their order during the factorization. A precomputed permutation may not be optimal (even if it is optimal for R) because it ignores actual pivot row selections. On the other hand, the fact that the nonzero structure of L and U are contained in that of R allows the factorization algorithm to precompute useful structural information, before the numerical factorization begins. In particular, the algorithm can identify columns that can be eliminated concurrently.

Delaying the construction of the column permutation until the numerical factorization allows columns to be selected for elimination using complete information about the structure of the reduced matrix (this information is often represented only implicitly, so it is not always easy to use). On the other hand, constructing the column permutation during the factorization rules out almost any pre-estimation of the nonzero structure of the factors. In particular, this approach does not allow a preprocessing algorithm to identify columns that can be eliminated concurrently. Another potential disadvantage of late column selection is the fact that greedy heuristics are used in such algorithms, whereas column preordering algorithms can use preordering algorithms with provable theoretical bounds [4, 24, 28]. Some algorithms combine column preordering with slight dynamic modifications to the precomputed ordering [9].

2.3. **The Column Elimination Tree.** When the column ordering is known in advance (before the numerical factorization begins), the factorization algorithm can quickly compute a data structure that captures information about all potential dependences in the numerical factorization process. This data structure is called the *column elimination tree*; our algorithm uses it for several purposes.

The column elimination tree of A is the symmetric elimination tree [34] of $A^T A$ under the assumption that no numerical cancellation occurs during the formation of $A^T A$. The column elimination tree can be computed in time almost linear in the number of nonzeros in A [22]. Our algorithm relies on the following properties of the column elimination tree.

Theorem 2.1. *(from* [26]*) 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 etree 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 (that is, A cannot be permuted to a nontrivial block triangular form). 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.*

2.4. **Parallel Programming with Cilk.** We have implemented the algorithm in Cilk [20, 39], a programming environment that supports a fairly minimal parallel extension of the C programming language. Cilk programs use a specialized run-time system that performs the scheduling of the computation using a fixed number of operating-system threads.

The key constructs of the Cilk language are illustrated in Figure 2.2. The `spawn` keyword declares that the function call that follows can be executed concurrently with the calling function. The operating-system thread that spawns a computation always suspends the calling function (saving its state on the stack) and executes the spawned function. In most cases, when the spawned function returns, the calling function is still waiting on the stack and its execution is resumed by the same thread that suspended it. But if, during the execution of the spawned function, another thread becomes idle, it may steal the activation frame of the calling function from the stack and resume its execution concurrently with the spawned function. The `sync` keyword is the main synchronization mechanism. It suspends the execution of a function until all the functions that it has spawned return.

Another synchronization mechanism that Cilk supports is the `inlet`. An inlet is a subfunction that spawned functions activate when they return. At most one copy of an inlet of an invocation of a function may be active at a given time. This scheduling constraint can be used to serialize the processing of values returned by spawned functions. For further details, see [20, 39] or [32].

```
cilk void mat_mult_add(int n,
                       matrix A, matrix B, matrix C) {
  if (n < blocksize) {
    mat_mult_add_kernel(n, A, B, C);
  } else {
    // Partition A into A_11, A_12, A_21, A_22
    // Partition B and C similarly
    spawn mat_mult_add(n/2,A_11,B_11,C_11);
    spawn mat_mult_add(n/2,A_11,B_12,C_12);
    spawn mat_mult_add(n/2,A_21,B_11,C_21);
    spawn mat_mult_add(n/2,A_21,B_12,C_22);
    sync; // wait for the 4 calls to return
    spawn mat_mult_add(n/2,A_12,B_21,C_11);
    spawn mat_mult_add(n/2,A_12,B_22,C_12);
    spawn mat_mult_add(n/2,A_22,B_21,C_21);
    spawn mat_mult_add(n/2,A_22,B_22,C_22);
  }
}
```

FIGURE 2.2. A simplified Cilk code for square matrix multiply-add. The code is used as an illustration of the main features of Cilk.

3. The Unsymmetric-Pattern Multifrontal Method with Column Preordering

The aim of this section is to provide a complete but easy-to-understand description of the unsymmetric-pattern multifrontal method with column preordering. Neither the unsymmetric-pattern multifrontal method itself nor its column preordering variant is new. Both have been described before [6, 5], but to better explain our improvements and our parallel strategies, we provide here a complete and easy-to-understand description of the basic method. To keep the description simple, we ignore supernodes in this chapter.

3.1. **Multifrontal Representation of the Reduced Matrix.**

In modern sparse-matrix factorizations, the reduced matrices $A^{(j)}$ are almost never represented explicitly. One possible representation for the reduced matrices, which is used by unsymmetric-pattern multifrontal algorithm, relies on the expansion

$$\begin{aligned}
A^{(j)}_{\bar{p}(1:\,j),j+1:\,n} &= A^{(j-1)}_{\bar{p}(1:\,j),j+1:\,n} - L_{\bar{p}(1:\,j),j}U_{j,j+1:\,n} \\
&= A^{(j-2)}_{\bar{p}(1:\,j),j+1:\,n} - L_{\bar{p}(1:\,j),j-1}U_{j-1,j+1:\,n} - L_{\bar{p}(1:\,j),j}U_{j,j+1:\,n} \\
&= A^{(0)}_{\bar{p}(1:\,j),j+1:\,n} - \sum_{k=1}^{j} L_{\bar{p}(1:\,j),k}U_{k,j+1:\,n} \;.
\end{aligned}$$

(we continue to use the notation introduced in section 2.1.)

Multifrontal algorithms multiply, at every step, the *L*-*U* product inside the summation, but they do not sum them up immediately. That is, the reduced matrix is always represented as a sum of the original matrix $A = A^{(0)}$, and a sum of rank-1 matrices, which are called *contribution blocks* or *update matrices.*

Using the above expansion one easily reformulates the dense algorithm given in Figure 2.1. The unsymmetric-pattern multifrontal algorithm is given in Figure 3.1. This pseudo-code, while mathematically correct, leaves out the details on how to utilize the sparsity. This utilization, which is essential for an efficient implementation, is given in the next section.

3.2. **Exploiting Sparsity.** As we explained, the reduced matrix $A^{(j)}$ is represented by a sum of the matrices $\left\{A^{(0)}_{\bar{p}(1:\,j),j+1:\,n}, F^{(1)}, F^{(2)}, ..., F^{(j-1)}\right\}$. These matrices are sparse and the algorithm must exploit that. Multifrontal algorithms use two kinds of representations for sparse matrices. The matrices A, L, and U, which are accessed by column and/or by row, are stored in compressed-column or compressed-row format. In a compressed-column format, the matrix is essentially stored as an array of compressed sparse columns. For each column in the array, the representation consists of an array of ℓ row indices and an array of ℓ nonzero values. Compressed-row format is similar, but row oriented.

Contribution blocks are kept in a more efficient data structure. A contribution block is a sparse matrix, but because it has rank 1, all of its nonzero columns have the same structure, and all of its nonzero rows have the same structure. This uniformity can be exploited in the data structure. A contribution block is represented by a two-dimensional array containing the nonzero values, an array of nonzero row indices, and an array of nonzero column indices. For example, in the factorization of a 5-by-5 matrix, the contribution block

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 6 & 0 & 0 & 0 \end{pmatrix}$$

```
**[L, U, p] = umf_lu(A)** $\triangleright$ ignores sparsity
$A^{(0)} \longleftarrow A$
for $j \longleftarrow 1 \colon n$
  $\triangleright$ assemble column $j$ of $A^{(j-1)}$
  $\triangleright$ recall that $F^{(k)} = L_{\bar{p}(1\colon k),k} U_{k,k+1\colon n}$
  $A^{(j-1)}_{\bar{p}(1\colon j-1),j} \longleftarrow A^{(0)}_{\bar{p}(1\colon j-1),j} - \sum_{k=1}^{j-1} F^{(k)}_{\bar{p}(1\colon j-1),j}$
  $\triangleright$ now that column $j$ is assembled, find the pivot
  $p(j) \longleftarrow \arg\max_{\bar{p}(1\colon j-1)} \left| A^{(j-1)}_{\bar{p}(1\colon j-1),j} \right|$
  $\triangleright$ having determined $p(j)$, we assemble row $p(j)$
  $A^{(j-1)}_{p(j),j\colon n} \longleftarrow A^{(0)}_{p(j),j\colon n} - \sum_{k=1}^{j-1} F^{(k)}_{p(j),j:n}$
  $\triangleright$ now factor column $j$ and row $p(j)$
  $L_{\bar{p}(1\colon j-1),j} \longleftarrow A^{(j-1)}_{\bar{p}(1\colon j-1),j} / A^{(j-1)}_{p(j),j}$
  $U_{j,j\colon n} \longleftarrow A^{(j-1)}_{p(j),j\colon n}$
  $\triangleright$ compute the contribution block
  $F^{(j)} \longleftarrow L_{\bar{p}(1\colon j),j} U_{j,j+1\colon n}$
end
```

FIGURE 3.1. An unsymmetric-pattern multifrontal algorithm. This pseudo-code, while mathematically correct, leaves out many details that are essential for an efficient implementation.

is represented by

$$\left\{ \begin{array}{cc} & \begin{array}{cc} 1 & 2 \end{array} \\ \begin{array}{c} 2 \\ 5 \end{array} & \begin{pmatrix} 2 & 3 \\ 4 & 6 \end{pmatrix} \end{array} \right\} .$$

We will denote the nonzero structure of $F^{(j)}$'s columns by the ordered set Ξ_j. The nonzero structure of $F^{(j)}$'s rows will be denoted by the ordered set Ψ_j. In the example above, $\Xi_j = \{2, 5\}$ and $\Psi_j = \{1, 2\}$. During the factorization, a column/row of an contribution block may be used to assemble the current pivotal column/row. In that case this column/row is no longer really a member of the contribution block since it has already been used, so we trim it out of the contribution block. In the above example if we use the contribution block to assemble column 2 then after doing so we will have to trim Ψ_j to $\{1\}$.

Efficient assembly of rows and columns poses two main challenges. First, most of the terms in the summation contribute nothing. It is essential to efficiently identify the contribution blocks that do contribute to a particular assembly. Second, assembly operations sum multiple sparse vectors from rectangular contribution blocks into a single vector. These operations must be carried out efficiently.

Let us first determine the nonzero terms in the summation

$$\sum_{k=1}^{j-1} F^{(k)}_{\bar{p}(1:\,j-1),j} = \sum_{k=1}^{j-1} L_{\bar{p}(1:\,k),k} U_{k,j} \; .$$

The kth term is nonzero if and only if $U_{k,j} \neq 0$. We ignore numerical cancellation, which means here that we will explicitly add a zero term if $A^{(k-1)}_{p(k),j}$ is a structural nonzero. Therefore, to determine the set of terms that must be explicitly summed, we search for $k \in 1\colon j-1$ such that $A^{(k-1)}_{p(k),j}$ is a structural nonzero. We denote this set of contribution blocks by $\mathrm{lc}_j = \{k : j \in \Psi_k\}$. Similarly, the set of contribution blocks that contribute to the assembly of row $p(j)$ is denoted by $\mathrm{uc}_j = \{k : p(j) \in \Xi_k\}$. Multifrontal algorithms differ in how they identify these sets; we will explain how our algorithm performs this task in Section 4.

Once these sets are determined, the algorithm knows the sparse structure (in the reduced matrix $A^{(j-1)}$) of a column and of its pivot row. The element $A^{(j-1)}_{i,j}$ is nonzero if either $A_{i,j} \neq 0$ or if i is in the row set of one the contribution blocks that contribute to column j, that is, $i \in \Xi_k$ for some $k \in \mathrm{lc}_j$. This means that the nonzero structure of column j, denoted by Γ_j, is given by

$$\Gamma_j = \left(\mathrm{struct}(A_{:\,,j}) \cup \bigcup_{k \in \mathrm{lc}_j} \Xi_k \right) \cap \overline{p}(1:j-1) \; .$$

By the same logic, the nonzero structure of row $p(j)$, denoted by Δ_j, is given by

$$\Delta_j = \left(\mathrm{struct}(A_{p(j),:}) \cup \bigcup_{k \in \mathrm{uc}_j} \Psi_k \right) \cap j:n \; .$$

Once these non-zero structures are determined, it is easy to create a static data structure that will allow the assemblies to be performed efficiently. The assembly operations are carried out in a series of so-called *extend-add* operations, that each add one column/row from a contribution block to the currently assembled row or column. Again, multifrontal algorithms differ in the data structures that they use, so we defer the details until later in the paper.

Figure 3.2 presents the detailed management of the sparse nonzero structures in the form of pseudo-code. This essentially concludes the description of the basic unsymmetric-pattern multifrontal method, with one exception. This exception is the merging of contribution blocks. This is an optimization that prevents a storage explosion, and we describe it next.

```
**[L, U, p] = sparse_umf_lu(A)** $\triangleright$ sparse
$A^{(0)} \longleftarrow A$
for $j \longleftarrow 1 \colon n$
  $\triangleright$ assemble column $j$ of $A^{(j-1)}$
  $\mathrm{lc}_j \longleftarrow \{k : j \in \Psi_k\}$
  $\Gamma_j \longleftarrow (\mathrm{struct}(A^{(0)}_{:,j}) \cup \bigcup_{k \in \mathrm{lc}_j} \Xi_k) \cap \overline{p}(1 : j-1)$
  $A^{(j-1)}_{\Gamma_j,j} \longleftarrow A^{(0)}_{\Gamma_j,j}$
  foreach $k \in \mathrm{lc}_j$
    extend-add $A^{(j-1)}_{\Gamma_j,j} \longleftarrow A^{(j-1)}_{\Gamma_j,j} - F^{(k)}_{\Xi_k,j}$
    remove column $j$ from $F^{(k)}$: $\Psi_k \longleftarrow \Psi_k \setminus \{j\}$
  end
  $\triangleright$ now that column $j$ is assembled, find the pivot
  $p(j) \longleftarrow \arg\max_{\Gamma_j} \left| A^{(j-1)}_{\Gamma_j,j} \right|$
  $\triangleright$ having determined $p(j)$, we assemble row $p(j)$ except
  $\triangleright$ for the pivot element $A_{p(j),j}$, which is already assembled
  $\mathrm{uc}_j \longleftarrow \{k : p(j) \in \Xi_k\}$
  $\Delta_j \longleftarrow (\mathrm{struct}(A^{(0)}_{p(j),:}) \cup \bigcup_{k \in \mathrm{uc}_j} \Psi_k) \cap j : n$
  $A^{(j-1)}_{p(j),\Delta_j \setminus \{j\}} \longleftarrow A^{(0)}_{p(j),\Delta_j \setminus \{j\}}$
  foreach $k \in \mathrm{uc}_j$
    extend-add $A^{(j-1)}_{p(j),\Delta_j \setminus \{j\}} \longleftarrow A^{(j-1)}_{p(j),\Delta_j \setminus \{j\}} - F^{(k)}_{p(j),\Psi_k}$
    remove pivotal row from $F^{(k)}$: $\Xi_k \longleftarrow \Xi_k \setminus \{p(j)\}$
  end
  $\triangleright$ now factor column $j$ and row $p(j)$
  $L_{\Gamma_j,j} \longleftarrow A^{(j-1)}_{\Gamma_j,j} / A^{(j-1)}_{p(j),j}$
  $U_{j,\Delta_j} \longleftarrow A^{(j-1)}_{p(j),\Delta_j}$
  $\triangleright$ compute the contribution block
  $\Xi_j \longleftarrow \Gamma_j \setminus \{p(j)\}$
  $\Psi_j \longleftarrow \Delta_j \setminus \{j\}$
  $F^{(j)}_{\Xi_j,\Psi_j} \longleftarrow L_{\Xi_j,j} U_{j,\Psi_j}$
end
```

FIGURE 3.2. An unsymmetric-pattern multifrontal algorithm. This pseudo-code is more detailed than the code in Figure 3.1, but still leaves out details.

3.3. **Merging Contribution Blocks.** Each factorization step consumes a row and/or a column from some of the existing contribution blocks, and produces a new contribution block. When all the rows and columns of a contribution block have been consumed, it no longer exists, and memory is no longer allocated to it. However, this natural consumption of contribution blocks is often not fast enough, and space allocated to contribution blocks may cause the algorithm to run out of space. Fortunately, space can often be conserved by merging contribution blocks.

To appreciate the magnitude of the problem, consider the factorization of a dense matrix. After exactly $n/2$ rows and columns have been eliminated, $n/2$ contribution blocks have been produced, and each of them still contains $n/2$ unconsumed rows and $n/2$ unconsumed columns. Therefore, at this point the algorithm requires $\Theta(n^3)$ storage, far greater than the $\Theta(n^2)$ required to store the factors. A simple left-looking or right-looking algorithm can factor a dense matrix in place, so clearly the space that is used to store contribution blocks is not required, at least in this case.

In the symmetric-positive-definite case, it is possible to show that a multifrontal algorithm requires a $\Theta(|L| \log n)$ memory for contribution blocks [38]. It is likely that in the unsymmetric case the situation is similar, in that the algorithm might need much more memory for contribution blocks than the size of the factors. Still, techniques that reduce the storage requirements in practice are crucial for preventing storage explosion.

The key to reducing the storage requirements is to merge existing contribution blocks, or parts thereof, into the new contribution block. This process, which is called *merging* or *absorption* is illustrated in Figure 3.3. If an existing block $F^{(k)}$ contributes to the assembly of column j, then any column in $F^{(k)}$ which is also in $F^{(j)}$ can be added to $F^{(j)}$ and trimmed from $F^{(k)}$. Similarly, if $F^{(k)}$ contributes to the assembly of row $p(j)$, then any row in $F^{(k)}$ which is also in $F^{(j)}$ can be trimmed and added to $F^{(j)}$. The best case occurs when $F^{(k)}$ contributes to both column j and row $p(j)$. In this case, all the rows and columns of $F^{(k)}$ can be absorbed into $F^{(j)}$.

We must prove formally that these merging rules are correct, in the sense that merging does not expand the nonzero structure of $F^{(j)}$. To prove that the merging rules are correct, we need a notation for the nonzero structure of an existing contribution block that contributes to a new one. Suppose that $F^{(k)}$ contributes to the assembly of column j or to the assembly of row $p(j)$. We denote by $\Xi_k^{(j)}$ the row structure of $F^{(k)}$ just prior to the assembly of column j, and by $\Psi_k^{(j)}$ the column structure of $F^{(k)}$ just prior to the assembly of row $p(j)$. After the factoring of column j and row $p(j)$, and just before factoring column $j+1$, the structure of $F^{(k)}$ is $\Xi_k^{(j+1)}$ and $\Psi_k^{(j+1)}$. We need these notations because these nonzero structures evolve over time as rows and columns are trimmed from $F^{(k)}$.

Figure 3.4 shows the algorithm with these merging rules, and using the new notation. To prove that the merging rules are true we have to show

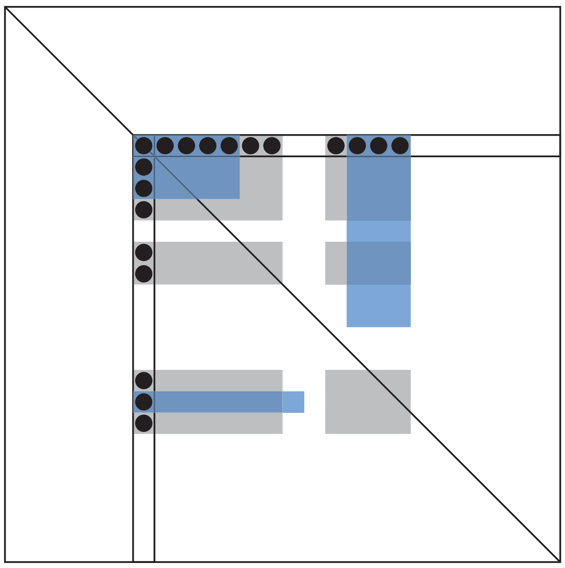

FIGURE 3.3. Merging an existing contribution block into a new contribution block. The figure shows the nonzeros in the current row and column, number 7. The contribution block $F^{(7)}$ is shown in gray. Three existing contribution blocks contribute to row and/or column 7. One of them contributes to both the row and the column, so it is completely absorbed. Another contribute only to the row, so some of its rows are absorbed but others do not, and similarly for the block that contributes to column 7 but not to row 7.

that the structures of the matrices that we use in the extend-add operation of the merging are indeed consumed inside the new frontal matrices. For example, in the case that $k \in \mathrm{lc}_j$ we have to show that $\Xi_k^{(j)} \setminus \{p(j)\} \subseteq \Xi_j^{(j+1)}$. This is shown in the next lemma.

Lemma 3.1. *For every $k < j$ such that $k \in lc_j$ ($k \in uc_j$) we have $\Xi_k^{(j+1)} \setminus \{p(j)\} \subseteq \Xi_j^{(j+1)}$ ($\Psi_k^{(j+1)} \setminus \{j\} \subseteq \Psi_j^{(j+1)}$)*

Proof. We will show the $k \in \mathrm{lc}_j$ case. The other case is symmetric. Notice that in the algorithm we have

$$\begin{aligned}\Gamma_j &= (\mathrm{struct}(A^{(0)}_{:,j}) \cup \bigcup_{k \in \mathrm{lc}_j} \Xi_k) \cap \overline{p}(1:j-1) \\ &= (\mathrm{struct}(A^{(0)}_{:,j} \cap \overline{p}(1:j-1)) \cup \bigcup_{k \in \mathrm{lc}_j} (\Xi_k^{(j)} \cap \overline{p}(1:j-1))\,.\end{aligned}$$

```
**[L, U, p] = sparse_umf_lu(A)**
$A^{(0)} \longleftarrow A$
for $j \longleftarrow 1\colon n$
    $\triangleright$ assemble column $j$ of $A^{(j-1)}$
    $\mathrm{lc}_j \longleftarrow \{k : j \in \Psi_k^{(j)}\}$
    $\Gamma_j \longleftarrow (\mathrm{struct}(A_{:,j}^{(0)}) \cup \bigcup_{k\in \mathrm{lc}_j} \Xi_k^{(j)}) \cap \overline{p}(1:j-1)$
    $A_{\Gamma_j,j}^{(j-1)} \longleftarrow A_{\Gamma_j,j}^{(0)}$
    foreach $k \in \mathrm{lc}_j$ extend-add $A_{\Gamma_j,j}^{(j-1)} \longleftarrow A_{\Gamma_j,j}^{(j-1)} - F_{\Xi_k^{(j)},j}^{(k)}$
    $\triangleright$ now that column $j$ is assembled, find the pivot
    $p(j) \longleftarrow \arg\max_{\Gamma_j} \left|A_{\Gamma_j,j}^{(j-1)}\right|$
    $\triangleright$ having determined $p(j)$, we assemble row $p(j)$ except
    $\triangleright$ for the pivot element $A_{p(j),j}$, which is already assembled
    $\mathrm{uc}_j \longleftarrow \{k : p(j) \in \Xi_k^{(j)}\}$
    $\Delta_j \longleftarrow (\mathrm{struct}(A_{p(j),:}^{(0)}) \cup \bigcup_{k\in \mathrm{uc}_j} \Psi_k^{(j)}) \cap j:n$
    $A_{p(j),\Delta_j\setminus\{j\}}^{(j-1)} \longleftarrow A_{p(j),\Delta_j\setminus\{j\}}^{(0)}$
    foreach $k \in \mathrm{uc}_j$ extend-add $A_{p(j),\Delta_j\setminus\{j\}}^{(j-1)} \longleftarrow A_{p(j),\Delta_j\setminus\{j\}}^{(j-1)} - F_{p(j),\Psi_k^{(j)}\setminus\{j\}}^{(k)}$
    $\triangleright$ now eliminate column $j$ and row $p(j)$
    $L_{\Gamma_j,j} \longleftarrow A_{\Gamma_j,j}^{(j-1)}/A_{p(j),j}^{(j-1)}$
    $U_{j,\Delta_j} \longleftarrow A_{p(j),\Delta_j}^{(j-1)}$
    $\triangleright$ compute the contribution block
    $\Xi_j^{(j+1)} \longleftarrow \Gamma_j \setminus \{p(j)\}$
    $\Psi_j^{(j+1)} \longleftarrow \Delta_j \setminus \{j\}$
    $F_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}}^{(j)} \longleftarrow L_{\Xi_j^{(j+1)},j} U_{j,\Psi_j^{(j+1)}}$
    $\triangleright$ continued in Figure 3.5 ...
```

FIGURE 3.4. The unsymmetric-pattern multifrontal method with the contribution-merging rules. Merging follows the elimination of every column. The elimination itself is nearly identical to the one showed in 3.2, except that superscript is added to every Ξ and Ψ. The pseudo code uses the notation that we need for the proof. Since $\Xi_k^{(j)}$ is never needed once $\Xi_k^{(j+1)}$ is constructed, there is no need to keep $\Xi_k^{(j)}$; in an actual code, $\Xi_k^{(j+1)}$ simply overwrites $\Xi_k^{(j)}$. Therefore, all the rules that keep $\Xi_k^{(j+1)}$ identical to $\Xi_k^{(j)}$ simply translate into no-ops, and similarly for $\Psi_k^{(j+1)}$. The code continues in Figure 3.5.

```
  $\triangleright$ ... contribution unification (continued from Figure 3.4)
  for each $k \in \mathrm{lc}_j \cap \mathrm{uc}_j$
    extend-add $F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} \longleftarrow F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} + F^{(k)}_{\Xi_k^{(j)}\setminus\{p(j)\},\Psi_k^{(j)}\setminus\{j\}}$
    discard $F^{(k)}$: $\Xi_k^{(j+1)} \longleftarrow \emptyset$, $\Psi_k^{(j+1)} \longleftarrow \emptyset$
  end
  for each $k \in \mathrm{lc}_j \setminus \mathrm{uc}_j$
    extend-add $F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} \longleftarrow F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} + F^{(k)}_{\Xi_k^{(j)}\setminus\{p(j)\},\Psi_k^{(j)}\cap\Psi_j^{(j+1)}}$
    $\Psi_k^{(j+1)} \longleftarrow \Psi_k^{(j)} \setminus \Delta_j$
    $\Xi_k^{(j+1)} \longleftarrow \Xi_k^{(j)} \setminus \{p(j)\}$
  end
  for each $k \in \mathrm{uc}_j \setminus \mathrm{lc}_j$
    extend-add $F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} \longleftarrow F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} + F^{(k)}_{\Xi_k^{(j)}\cap\Xi_j^{(j+1)},\Psi_k^{(j)}\setminus\{j\}}$
    $\Xi_k^{(j+1)} \longleftarrow \Xi_k^{(j)} \setminus \Gamma_j$
    $\Psi_k^{(j+1)} \longleftarrow \Psi_k^{(j)} \setminus \{j\}$
  end
  for all other $k < j$
    $\Xi_k^{(j+1)} \longleftarrow \Xi_k^{(j)}$
    $\Psi_k^{(j+1)} \longleftarrow \Psi_k^{(j)}$
  end
end
```

FIGURE 3.5. Continuation of the code from Figure 3.4
.

Hence $\Xi_k^{(j)} \cap \overline{p}(1:j-1) \subseteq \Gamma_j$. We claim that $\Xi_k^{(j)} \subseteq \overline{p}(1:j-1)$, so in fact $\Xi_k^{(j)} = \Xi_k^{(j)} \cap \overline{p}(1:j-1) \subseteq \Gamma_j$.

We will show that $\Xi_k^{(j)} \subseteq \overline{p}(1:j-1)$ using induction on j. First we note that $\Xi_k^{(k+1)} = \Gamma_k \setminus \{p(k)\}$. By definition $\Gamma_k \subseteq \overline{p}(1:k-1)$ we have $\Xi_k^{(k+1)} \subseteq \overline{p}(1:k-1) \setminus \{p(k)\}$ and we have $\Xi_k^{(k+1)} \subseteq \overline{p}(1:k)$. Suppose that $\Xi_k^{(j)} \subseteq \overline{p}(1:j-1)$ we now have to prove $\Xi_k^{(j+1)} \subseteq \overline{p}(1:j)$. Since $\Xi_k^{(j+1)} \subseteq \Xi_k^{(j)} \subseteq \overline{p}(1:j-1)$ we have only to prove for the case that $p(j) \in \Xi_k^{(j)}$. If indeed $p(j) \in \Xi_k^{(j)}$ we have $k \in \mathrm{uc}_j$. We now have two options: either $k \in \mathrm{lc}_j$ or $k \notin \mathrm{lc}_j$. If $k \in \mathrm{lc}_j$ we have the assignment $\Xi_k^{(j+1)} \longleftarrow \emptyset$ and we have $\Xi_k^{(j+1)} \subseteq \overline{p}(1:j)$. If $k \notin \mathrm{lc}_j$ we have the assignment $\Xi_k^{(j+1)} \longleftarrow \Xi_k^{(j)} \backslash \Gamma_j$. Since the pivot at column j is chosen only from Γ_j we must have $p(j) \in \Gamma_j$ and therefore $p(j) \notin \Xi_k^{(j)}$. We now have $\Xi_k^{(j+1)} \subseteq \overline{p}(1:j)$, and we have finished to prove that $\Xi_k^{(j)} \subseteq \overline{p}(1:j-1)$.

From $\Xi_k^{(j)} \subseteq \Gamma_j$ we conclude that $\Xi_k^{(j)} \setminus \{p(j)\} \subseteq \Gamma_j \setminus \{p(j)\}$. We now have two options: either $k \in \mathrm{uc}_j$ or $k \notin \mathrm{uc}_j$. If $k \in \mathrm{uc}_j$ we have $\Xi_k^{(j+1)} = \emptyset$ we are done. If $k \notin \mathrm{uc}_j$ then $\Xi_k^{(j+1)} = \Xi_k^{(j)} \setminus \{p(j)\}$ and by definition $\Xi_j^{(j+1)} = \Gamma_j \setminus \{p(j)\}$ so we also have $\Xi_k^{(j+1)} \subseteq \Xi_j^{(j+1)}$. □

These merging rules do not reduce the number of contribution blocks to a minimum, and there are also cases where the minimum number of contribution blocks is high. Even when a contribution block $F^{(k)}$ is completely covered by a new one $F^{(j)}$, our absorption rules may fail to absorb it if it does not contribute to column j or to row $p(j)$. There are also more complex cases where no absorption rules can reduce overlaps without increasing the number of contribution blocks.

4. The New Algorithm

4.1. **Finding Contributing Blocks.** The first task during the elimination of column j is the assembly of the column, which requires identifying the set $\mathrm{lc}_j = \{k \colon j \in \Psi_k^{(j)}\}$. Without absorption, lc_j is exactly the structure of column j of U, except for the diagonal element $U_{p(j),j}$. Because of absorption, lc_j may be a proper subset of the column structure. To show this, we first note that contribution blocks only shrinks during the factorization, that is $\Psi_k^{(j)} \subseteq \Psi_k^{(k+1)}$, so $\mathrm{lc}_j \subseteq \{k : j \in \Psi_k^{(k+1)}\}$. In the algorithm, $\Psi_k^{(k+1)} \longleftarrow \Delta_k \setminus \{k\}$, where Δ_k is the structure of the kth row of U. Therefore, $j > k$ is in $\Psi_k^{(k+1)}$ if and only if $U_{kj} \neq 0$.

The simplest way to determine lc_j is to determine the column structure in U, and to examine each candidate contribution block, to check whether j is still in $\Psi_k^{(j)}$. There are at least three ways to determine the structure of a column in U. The Gilbert-Peierls approach [27], which is also used in SuperLU [14], determines the column structure using a depth-first search (DFS) in the graph of L. Gilbert and Peierls proved that the total amount of work that all of these searches require is $O(\mathrm{flops}(LU) + m)$, where m is the number of non-zeros in A and $\mathrm{flops}(LU)$ is the number of non-zero multiplications required when doing the multiplication LU. A heuristic called symmetric pruning can often accelerate the searches by pruning edges from the graph of L [19].

A second approach is to maintain linked lists for the structure of each column of U. Pointers to the linked lists are stored in an array of size n. After forming row $p(k)$ of U, we insert the index k to the linked lists representing columns $\Psi_k^{(k+1)}$. This can be done in time proportional to

$\left|\Psi_k^{(k+1)}\right|$. The total time it takes to build the linked lists is proportional to the number of non-zeros in U. When we get to the elimination of column j, the structure of column j in U is explicitly represented by the corresponding linked list. UMFPACK used the linked-list approach, and it appears that it actually removes elements from these lists during absorption, to make the search more precise. There is no running-time analysis of that technique.

Although the linked-lists approach is simple and more efficient than the DFS approach, it is inappropriate for a parallel algorithm, due to the need to lock the lists.

We use a third approach, which computes a superset of the structure of column j of U using the column elimination tree. If $U_{kj} \neq 0$, then k must be a descendant of j in the column elimination tree. Therefore, the descendants of j in the column elimination tree form a superset of the actual non-zero structure of column j of U. We enumerate this superset and check each contribution block, to determine whether it contributes to lc_j.

We acknowledge that our approach may be less efficient than the DFS and linked-list approaches, but it is simple and require no locking. Our numerical experiments indicate that on real-world matrices, our approach is efficient. It may be the case that a more sophisticated approach, such as the DFS approach, will yield an algorithm with better theoretical running-time bounds, and perhaps even somewhat faster in practice.

Due to contribution-block merging, all the approaches only find a superset of lc_j. The algorithm still needs to find the actual contributors. We do this together with finding the actual location of column j inside the contributing contribution block, and is discussed in the next section.

Constructing uc_j is completely analogous and we perform that task in exactly the same way.

4.2. **Performing Extend-Add Operations.**

Recall that the contribution blocks are kept in a dense format, where each column/row corresponds to column/row of the sparse matrix. The algorithm has to find out whether a column is a member of the contribution block, and if so where it is located inside the dense matrix.

There are several ways which this can be done. The first method, used by UMFPACK, is suitable when linked lists represent the sets lc_j and uc_j (or supersets thereof). The elements of the list store not only the row or column index, but also its location in the contribution block. This data structure requires careful management when row/column locations within contribution blocks change due to merging.

Another method is to keep the column and row indices of the contribution block in a dictionary data structure, such as a sorted array, a balanced tree, or a hash table. Again, due to merging, the structure must support deletions.

Our code uses a simpler solution that simply stores the indices in an unsorted array. Our numerical experiments indicate that on real-world matrices, this simple approach is efficient and does not represent a bottleneck in the overall algorithm.

Once the contributing columns and rows are identified, we need to sum them up. The terms in these summations are sparse, so an appropriate data structure is required. The data structure that is used is called a *sparse accumulator* (SPA). There are several ways to implement a SPA. We describe here an implementation that is particularly effective in supernodal algorithms. Our SPA consists of an integer array `map` of size n, whose elements are initialized to in invalid value (-1 in our code), an integer initialized to 1, an array of numerical values (real or complex), and an array of integer indices. The size of the last two arrays must be large enough to store all the nonzeros in the sum their indices. The integer and these two arrays form a stack of value-index pairs, which is initially empty. The SPA maintains a vector, which is initially zero. To add a nonzero value to position i of the vector stored in the SPA, the algorithm first checks `map[`i`]`. If `map[`i`]=-1`, the algorithm pushes the nonzero and the index i onto the stack, and records their position in the stack in `map[`i`]`. If `map[`i`]` is valid, the nonzero value is simply added to the numerical value stored in position `map[`i`]` of the stack.

This sparse accumulator structure can be adapted easily to summing supernodal contributions, which we describe next.

4.3. **Supernodes in the New Algorithm.**

When the contribution blocks of several columns have similar nonzero structures, it is best to merge them. Consider columns i and $j > i$, such that $\Gamma_j = \Gamma_i \setminus \{p(i)\}$ and $\Delta_j = \Delta_i \setminus \{i\}$. The contribution blocks of the two columns are almost identical in structure. In fact, the contribution block of i will be merged into that of j. We can reorder the factorization process so that the two columns are first factored using a partial-pivoting dense LU factorization kernel, then the two rows of U are computed using a dense triangular solver, and then the two columns and the two rows are multiplied to produce a single contribution block. When this is done, we say that the two columns form a *supernode*.

Supernodes have been quickly recognized as a key element in efficient multifrontal algorithms [18, 3], as well as in other factorization algorithms [37, 36, 14, 30]. Supernodes reduce memory usage, cache misses, indexing overhead, and they help exploit fine-grained parallelism. The last issue is particularly important for our algorithm.

Amalgamating columns with similar but not identical nonzero structure often improves performance even though the amalgamation introduces explicit zeros into the sparse factors. In our example, if $\Gamma_j \neq \Gamma_i \setminus \{p(i)\}$ and/or $\Delta_j \neq \Delta_i \setminus \{i\}$, then column i (and/or row $p(i)$) in the supernodal data structure will include explicit zeros. These explicit zeros increase memory usage, data movement in the memory system, and instruction counts.

When the nonzero structures are similar enough or when the separate supernodes would otherwise be thin, these costs, however, are often smaller than the performance benefits that amalgamation brings. Like exact supernodes, amalgamated supernodes (sometimes called relaxed supernodes) were also identified useful early [18, 3].

Supernodes are easiest to exploit during the numerical factorization if they can be identified ahead of the numerical factorization phase. Supernodes are relatively easy to detect in symmetric factorizations and when pivoting is not necessary [35]. In our case, the situation is more complex because of the unsymmetry and because of pivoting. Our algorithm partitions the columns into supernodes prior to the numerical factorization. Due to pivoting, the partitioning is not exact: it may miss cases where the actual choice of pivoting leads to identical or almost identical row and column structures, if under another choice the structures differ considerably. It may also coalesce columns with different structures into supernodes. We describe our partitioning strategy later; for now, it suffices to say that a supernode in our algorithm always consists of a chain of vertices in the column elimination tree or of a leaf subtree (a subtree whose leaves are all leaves of the entire tree).

We now describe the supernodal numerical factorization. A supernode is ready to be factored when all the supernodes below it in the column elimination tree have been factored. When a supernode is ready to be factored, the algorithm determines the column structure of the supernode, which is the union of the column structures of the constituent columns. Next, the algorithm assembles all the columns together, using a rectangular compressed sparse matrix. This sparse matrix might have explicit zeros. The assembly operation consumes columns in the supernode from any existing contribution block that contributes to $A_{i,j}$, even if the (i, j) element in the supernode is an explicit zero (because it might fill due to the factorization of a column $j' < j$ in the supernode). Once the columns have been assembled, a partial-pivoting dense LU factorization kernel is applied to the supernode. This determines all the pivot rows, which are now assembled and factored. Next, the subdiagonal block column is multiplied by the block row to form the new contribution block. Finally, rows and columns from existing contribution blocks are merged into the new contribution block, and the factorization continues with the next supernode. The pseudocode for the algorithm is given in Figures 4.1 and 4.2.

We coalesce columns into supernodes using the following strategy. The algorithm traverses the column elimination tree bottom up. Near the leaves, we merge entire leaf subtrees into supernodes. The amalgamation criterion here is simple: a leaf supernode must have more than a certain number of columns, 20 in our implementation. If a leaf subtree is too small, the tree rooted at the subtree's root is examined, and so on. This criterion completely ignores the nonzero structure of columns.

```
**[L, U, p] = sparse_umf_lu(A)** $\triangleright$ contribution unification and supernodal
split $A$ into a set of $s$ supercolumns $\{\Omega_1, \Omega_2, ..., \Omega_s\}$
$A^{(0)} \longleftarrow A$
for $j \longleftarrow 1\colon s$
  $\triangleright$ assemble supercolumn $j$ of $A^{(j-1)}$
  $\mathrm{lc}_j \longleftarrow \{k : \Omega_j \cap \Psi_k^{(j)} \neq \emptyset\}$
  $\Gamma_j \longleftarrow (\mathrm{struct}(A^{(0)}_{:,\Omega_j}) \cup \bigcup_{k \in \mathrm{lc}_j} \Xi_k^{(j)}) \cap \overline{p}(\bigcup_{k \in \mathrm{lc}_j} \Omega_k)$
  $A^{(j-1)}_{\Gamma_j,\Omega_j} \longleftarrow A^{(0)}_{\Gamma_j,\Omega_j}$
  foreach $k \in \mathrm{lc}_j$ extend-add $A^{(j-1)}_{\Gamma_j,\Omega_j} \longleftarrow A^{(j-1)}_{\Gamma_j,\Omega_j} - F^{(k)}_{\Xi_k^{(j)},\Omega_j \cap \Psi_k^{(j)}}$
  $\triangleright$ factor the supercolumn itself
  solve $L_{\Gamma_j,\Omega_j} U_{\Omega_j,\Omega_j} = A^{(j-1)}_{\Gamma_j,\Omega_j}$ with pivots at $p(\Omega_j)$
  $\triangleright$ having determined $p(\Omega_j)$, we can assemble the rows $p(\Omega_j)$
  $\mathbf{uc}_j \longleftarrow \{k : p(\Omega_j) \cap \Xi_k^{(j)} \neq \emptyset\}$
  $\Delta_j \longleftarrow (\mathrm{struct}(A^{(0)}_{p(\Omega_j),:}) \cup \bigcup_{k \in \mathrm{uc}_j} \Psi_k^{(j)}) \cap (\bigcup_{k=j+1}^{s} \Omega_k)$
  $A^{(j-1)}_{p(\Omega_j),\Delta_j \setminus \Omega_j} \longleftarrow A^{(0)}_{p(\Omega_j),\Delta_j \setminus \Omega_j}$
  foreach $k \in uc_j$
    extend-add $A^{(j-1)}_{p(\Omega_j),\Delta_j \setminus \Omega_j} \longleftarrow A^{(j-1)}_{p(\Omega_j),\Delta_j \setminus \Omega_j} - F^{(k)}_{p(\Omega_j) \cap \Xi_k^{(j)},\Psi_k^{(j)} \setminus \Omega_j}$
  end
  $\triangleright$ now complete the factorization of the rest of the pivotal rows
  solve $L_{p(\Omega_j),\Omega_j} U_{\Omega_j,\Delta_j \setminus \Omega_j} = A^{(j-1)}_{p(\Omega_j),\Delta_j \setminus \Omega_j}$
  $\triangleright$ compute the contribution block
  $\Xi_j^{(j+1)} \longleftarrow \Gamma_j \setminus p(\Omega_j)$
  $\Psi_j^{(j+1)} \longleftarrow \Delta_j \setminus \Omega_j$
  $F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} \longleftarrow L_{\Xi_j^{(j+1)},\Omega_j} U_{\Omega_j,\Psi_j^{(j+1)}}$
  $\triangleright$ continued in Figure 4.2 ...
```

FIGURE 4.1. The supernodal version of the unsymmetric multifrontal algorithm. This pseudo-code leaves out the details on how to implement some of the operations. Continued in Figure 4.2.

```
  $\triangleright$ … continued from Figure 4.1.
  $\triangleright$ contribution unification
  for each $k \in \mathrm{lc}_j \cap \mathrm{uc}_j$
    extend-add $F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} \longleftarrow F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} + F^{(k)}_{\Xi_k^{(j)}\setminus p(\Omega_j),\Psi_k^{(j)}\setminus\Omega_j}$
    discard $F^{(k)}$: $\Xi_k^{(j+1)} \longleftarrow \emptyset$, $\Psi_k^{(j+1)} \longleftarrow \emptyset$
  end
  for each $k \in \mathrm{lc}_j \setminus \mathrm{uc}_j$
    extend-add $F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} \longleftarrow F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} + F^{(k)}_{\Xi_k^{(j)}\setminus p(\Omega_j),\Psi_k^{(j)}\cap\Psi_j^{(j)}}$
    $\Psi_k^{(j+1)} \longleftarrow \Psi_k^{(j)} \setminus \Delta_j$
    $\Xi_k^{(j+1)} \longleftarrow \Xi_k^{(j)} \setminus p(\Omega_j)$
  end
  for each $k \in \mathrm{uc}_j \setminus \mathrm{lc}_j$
    extend-add $F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} \longleftarrow F^{(j)}_{\Xi_j^{(j+1)},\Psi_j^{(j+1)}} + F^{(k)}_{\Xi_k^{(j)}\cap\Xi_j^{(j+1)},\Psi_k^{(j)}\setminus\Omega_j}$
    $\Xi_k^{(j+1)} \longleftarrow \Xi_k^{(j)} \setminus \Gamma_j$
    $\Psi_k^{(j+1)} \longleftarrow \Psi_k^{(j)} \setminus \Omega_j$
  end
  for all other $k < j$
    $\Xi_k^{(j+1)} \longleftarrow \Xi_k^{(j)}$
    $\Psi_k^{(j+1)} \longleftarrow \Psi_k^{(j)}$
  end
end
```

FIGURE 4.2. Continuation of Figure 4.1.

Above the leaf subtrees, our algorithm is more conservative. The algorithm uses a-priori *nonzero-count bounds* for the columns of the L and on the rows of U. We compute these bounds by constructing a bi-partite clique-cover representation of the row-merge graph [21]. We denote by μ_j the upper bound on the nonzero count of $L_{:,j}$ and by ν_j the upper bound on the nonzero count of $U_{j,:}$. If a vertex has more than one child, it will start a new supernode. If a vertex has only one child, the algorithm may include it in the supernode that contains the child. Consider a column j whose only child in the col-etree is $j-1$, whose only child is $j-2$, and so on, down to $j-q$, such that $j-1, \ldots j-q$ have already been coalesced into a supernode, and such that the children of $j-q$ are part of other supernodes. Should the algorithm add column j to the supernode starting at $j-q$? Adding j to the supernode may add explicit zeros to $L_{j-q:j-1,:}$ and to $U_{:,j-q:j-1}$. If we add column j to the supernode, the a-priori nonzero-count bound for columns $j-q$ through $j-1$ in L will rise to μ_j (minus superdiagonal elements), and the bound for the corresponding rows in U will rise to ν_j, again minus subdiagonal elements. The algorithm is designed not to disallow the addition of too many explicit zeros in the *predicted nonzero structure*. More specifically, we add column j to the supernode only if

$$(\mu_j + (q-1))\, q \leq \alpha \sum_{k=j-q}^{j} \mu_k \text{ and } (\nu_j + (q-1))\, q \leq \alpha \sum_{k=j-q}^{j} \nu_k \ ,$$

where α is an implementation parameter (we use $\alpha = 2$). Note that this formula does count superdiagonal elements that will be represented in the representation of L and subdiagonal elements in U. The actual increase in the nonzero counts may be larger than α, because the expressions on the left side of the two inequalities are a-priori upper bounds, not actual nonzero counts.

We have also experimented with detecting supernodes on the fly during the factorization. Although in principle one can coalesce columns based on the actual number of explicit zeros that must be represented, doing so prevents the algorithm from utilizing a dense LU factorization kernel. The dense kernel cannot be used because we can only decide whether to include column j in the supernode after the elimination of column $j-1$. To utilize a dense kernel, we must decide which columns it will factor before we invoke it. We used an on-the-fly strategy that does allow us to use a dense kernel. We assemble columns one by one into a supernodal block column. When the number of explicit zeros in this yet-unfactored block column of the trailing submatrix exceeds a threshold, we stop adding columns to the supernode. We then call a dense kernel to factor the supernode as in Figure 4.1. This strategy is conservative relative to the fully dynamic one, because some of the explicit zeros that we count in the yet-unfactored block may fill in L. In preliminary experiments method did not prove significantly superior to the static bounds-based decomposition, so we did not experiment with it any further.

4.4. **Exposing and Exploiting Parallelism.** Our algorithm exposes and exploits parallelism at several levels.

4.4.1. *Parallel Factorization of Siblings.* In factorization algorithms that are based on an elimination-tree, columns that are not in an ancestor-descendant relationship can be eliminated concurrently. In particular, this is true for LU factorizations with partial pivoting [25]. Virtually all the column-elimination-tree partial pivoting factorization codes today exploit this form of parallelism.

In our algorithm, whenever a node in the supernodal column elimination tree has more than one child, it spawns concurrent recursive factorizations of all its children.

This source of parallelism is not the only one in sparse LU with partial pivoting. Demmel, Gilbert, and Li found that LU factorization codes do not scale well unless more parallelism is exploited [15].

4.4.2. *Overlapping Factorizations with Column Assemblies.* Before a supernode can be factored, the contributions from its descendants must be assembled into the supernode. The assembly of the contributions is a summation operation, so it can be performed in any order. A contribution can only be summed after it has been computed, but it can be summed before other contributions have been computed.

Our algorithm partially exploits this source of parallelism. Once the factorization of a child subtree is completed, the parent supernode assembles the contributions from that subtree. This allows this summation to overlap the factorization of the other children. However, at any given time a supernode sums contributions from only one of its children's subtrees, to avoid data races on the supernode itself (multiple children can contribute to the same element of a supernode). The summation of the contributions from a child's subtree is also performed sequentially, contribution block by contribution block, to avoid data races. The serialization of the children's contribution is achieved using the inlet mechanism of Cilk.

We note that the data-flow constraints allow for more parallelism than we exploit. A contribution block from a distant descendant can be summed as soon as the block is computed. Our algorithm waits until the child is factored, and only then sums the contributions from that entire subtree. However, exploiting this form of parallelism is difficult, for two reasons. First, it is difficult to keep track of the exact data-flow constraints. More importantly, if a contribution block is assembled early into supernode j, it cannot be later merged into the contribution block of another descendant of j, since that might lead to summing the same contribution twice.

4.4.3. *Splitting the Computation of a Contribution Block.* Different columns of a contribution blocks are assembled into different supernodes. By splitting the computation of a contribution block into groups of column, we can assemble an already-computed block column into a near ancestor concurrently with the computation of another block column.

Our algorithm does exploit this source of parallelism, but in a limited way. First, we only split the column set of a contribution block into two sets, the set of columns that contribute to the parent of the supernode and the set of all the other columns. Second, we only split a contribution block if it is an only child.

When a supernode has two or more children we do not exploit this form of parallelism. This is because it is impossible to express this form of parallelism in Cilk without sacrificing the parallelism gained by computing contribution blocks in parallel. We note that in most cases, when a supernode has two or more children, there is at least some elimination-tree parallelism, so the loss of concurrency due to this restriction has a limited impact on scalability.

4.4.4. *Parallel Merging of Contribution Blocks.* After the contribution block of a supernode j has been computed, our algorithm attempts to merge existing contribution blocks into the contribution block of j. Contribution blocks of supernodes that are not an in an ancestor-descendant relation in the elimination tree can be merged concurrently, because their row structure is disjoint. Lemma 4.2 proves this claim. To prove the lemma we will need a theorem which is not new; it is due to Gilbert and appears in [25], but we prove it here because the technical report is difficult to obtain.

Theorem 4.1. *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 etree of A, and let $M = L + U$. If i and j do not have an ancestor-descendant relation in T then columns i and j in M are disjoint. That is struct$(M_{:,i}) \cap$ struct$(M_{:,j}) = \emptyset$.*

Proof. Suppose, by contradiction, that there are such an i and j, and let us assume that $i < j$. Let $k \in \text{struct}(M_{:,i}) \cap \text{struct}(M_{:,j})$. Since $k \in \text{struct}(M_{:,i})$ then either $L_{ki} \neq 0$ or $U_{ki} \neq 0$, depending if $k > i$. Since $i < j$ there are three case: (a) $U_{ki} \neq 0$ and $U_{kj} \neq 0$, (b) $L_{ki} \neq 0$ and $U_{kj} \neq 0$, and (c) $L_{ki} \neq 0$ and $L_{kj} \neq 0$.

In case (a) the column etree theorem dictates that k is a descendant of both i and j, which cannot be unless i is a descendant of j.

In case (b) the column etree dictates that k is a descendant of j, and i is a descendant of k, so i is a descendant of j.

We now consider case (c). Let us look on L_{ki}. Either it is a filled-in element or it is a non-zero in PA. If it is a non-zero in PA then let us define $i' = i$. If it is a filled-in element then there must exist an i' such that the element at ki' is a non-zero in PA. We will denote by i' the minimum such element. By the column etree theorem i' is a descendant of i. We define j' in a symmetric way, and it too is a descendant of j. We will assume that $i' \leq j'$, the other case is symmetric. Let us denote by k' the row in A that corresponds to row k in PA. Let P' be any permutation such that the pivot in column i' is k', and there exists a factorization $P'A = L'U'$ (not necessarily numerical stable). Such a permutation exists since $A_{k'i'} \neq 0$ and A is nonsingular. Let us now look at $U_{i'j'}$. Since $A_{k'i'} \neq 0$ we have that index $k'i'$ is non-zero in $P'A$, so we must have $U_{i'j'} \neq 0$. By the column etree theorem we conclude that i' is a descendant of j'. Since i' is a descendant of i and j' is a descendant of j, then i' is a descendant of both i and j, which can only be true if i is a descendant of j □

Lemma 4.2. *If supernodes i and j do not have an ancestor-descendant relation in the supercolumn elimination tree of A then for every k we have*

$$\Xi_i^{(k)} \cap \Xi_j^{(k)} = \emptyset\,.$$

Proof. Since $\Xi_i^{(k)} \subseteq \Xi_i^{(i+1)} \subseteq \Gamma_i$ and $\Xi_j^{(k)} \subseteq \Xi_j^{(j+1)} \subseteq \Gamma_j$ it is enough to prove that $\Gamma_i \cap \Gamma_j = \emptyset$. Recall that Γ_i (Γ_j) is the structure in L of the first column in supernode i (j). Therefore Γ_i and Γ_j are the structure of two distinct columns in L, columns that are members of supernodes that do not have an ancestor-descendant relationship in the supercolumn elimination tree. Recall that all supernodes are connected-subsets in the column elimination tree. Therefore the columns that are the structure of Γ_i and Γ_j do not have an ancestor-descendant relation in the column elimination tree of A. Using theorem 4.1 we conclude that the structure of the columns are disjoint, and therefore $\Gamma_i \cap \Gamma_j = \emptyset$. □

We exploit this source of parallelism as follows. The algorithm spawns concurrent procedures that merge contributions from all the children of a supernode j. Each of these procedures recursively invokes parallel contribution merging from the child's children, and so on. After the contributions from a subtree rooted at supernode i have been merged, the merging procedure tries to merge the contribution block of i; this is not done concurrently with the merging of other descendants of j.

4.4.5. *Parallel Dense Operations.* Another source of parallelism comes from operations on dense submatrices: factorization of supernodes, triangular solves to compute a supernodal row block of U, and matrix-matrix multiplication to compute a contribution block.

We have parallelized all of these operations using recursion in Cilk. At the bottom of the recursion, our code calls the level-3 sequential Basic Linear Algebra Subroutines (BLAS) [17] or LAPACK [2].

These parallel dense algorithms are standard, so we do not describe the details. We only mention that the parallel dense LU factorization algorithm that we implemented utilizes some of the techniques that the sparse algorithm uses. For example, we split the computation of an update to the trailing submatrix, to allow the factorization of the next block column to start as quickly as possible.

4.4.6. *Miscellaneous.* Our algorithm exploits two more sources of parallelism.

Once a supercolumn has been factored and the pivot rows have been assembled, we know the row and column structure of its contribution block. At this point, we cannot yet compute the contribution block, because we first need to compute the pivot rows using a dense triangular solve. But we can already merge contribution blocks from descendants. Therefore, our algorithm concurrently computes the pivot rows and merges contribution blocks. When both operations terminate, we multiply the block column with the block row and add the result to the contribution block.

The numerical operations during the merging of a contribution block i into another j are independent additions that can all be performed in parallel. Our algorithm partitions the merged contribution block i into blocks that are merged into j concurrently.

5. Experimental results

We now describe experimental results that we have obtained with the new solver, as well as comparisons to two other solvers, the sequential solver UMFPACK 4.0 [5] and the multithreaded SuperLU_MT [15]. We describe the matrices that we used for the experiments, the hardware and software environment, and the results of the experiments.

The comparisons to the two other codes are meant to achieve specific goals. The comparison to UMFPACK is meant to show that on a single processor, our algorithm achieve a level of performance similar to that of a state-of-the-art unsymmetric code. We do not claim that our code is preferable to UMFPACK on a uniprocessor, and certainly not to more recent versions of UMFPACK. The comparison to SuperLU_MT is meant to show that our code scales well. Unsymmetric direct solvers are notoriously hard to parallelize, so it is essential to evaluate the speedups of a new code relative to the speedups that other codes achieve on the same matrices, not to the theoretical hardware speedup limits.

In general, the results that we present are designed to substantiate our claims regarding the performance and scalability of the algorithm. The comparisons that we present here are *not meant* to assist prospective users in selecting a code; The selection of a code should ideally be based on an unbiased and carefully designed study, such as [29].

The results below include only the time for the symbolic analysis and for the numerical factorization, but not the time for ordering and triangular solves. However, in UMFPACK, the ordering and symbolic analysis phases are integrated. Therefore, for UMFPACK we measured only the numerical factorization time; we do not count UMFPACK's symbolic analysis time. As a consequence, comparisons of our code to UMFPACK have a bias that favors UMFPACK. (It is possible to separate UMFPACK's ordering and symbolic analysis phases, but this causes noticeable performance deterioration; we preferred to use the best scenario for UMFPACK.)

We used COLAMD [12, 13] to order all the matrices. As mentioned above, UMFPACK comes with a built-in slightly modified version of COLAMD, which it uses.

In the results below, our new code is labeled TAUCS, since it is now part of the TAUCS suite of linear solver that our group has been developing and distributing.[2]

[2]Available from `http://www.tau.ac.il/~stoledo/taucs/`.

5.1. **The Hardware and Software Environment.** We performed all the experiments reported here on a SGI Origin 3000 series computer with 32 processors and 32 GB of memory running the IRIX 6.5 operating system. The processors are 500 MHz MIPS R14000 with a 8 MB level-2 cache and a 32 KB level-1 data cache. [3]

We linked all the codes with the vendor's Basic Linear Algebra Subroutines (BLAS), SCSL version 1.4. We used the sequential version of the library.

We used SuperLU_MT version 1.0, the latest version. SuperLU_MT can utilize either OpenMP directives or POSIX threads. On the Origin, SuperLU defaults to using OpenMP, using the SGI compiler, and using a relatively old sequential BLAS (`complib.sgimath`). We compiled SGI using these defaults, except that we switched to the newer and faster BLAS library SCSL 1.4. The documentation specifically requires sequential BLAS, so we did not use the OpenMP version of SCSL. The version of the SGI compiler that we used is MIPSPro 7.3 with the optimization flags specified by the SuperLU_MT makefile, except that we changed the compiler target to R14000.

We used UMFPACK version 4.0. This was the latest version when we started this research, but it is no longer the most recent; we expect that newer versions give better results, at least on some of the matrices. We compiled UMFPACK with the default compiler (gcc) and optimization flags specified by the UMFPACK makefile. We used version 2.95 of the gcc compiler.

By default, UMFPACK uses threshold pivoting with a threshold of 0.1. We have implemented only partial pivoting (the pivot must be at least as large in absolute value as the rest of the elements in its column). To factor out this issue from the comparisons, we also used partial pivoting in UMFPACK, not the default threshold pivoting. Therefore, the comparisons below reflect the same numerical strategy, but not necessarily the best-performance/reliability tradeoff for UMFPACK.

We compiled our code using gcc, since Cilk only supports the gcc compiler. We used version 2.95 of the compiler and the with the `-O3` optimization flag.

All the codes were compiled using 32-bit mode, since the version of Cilk that we used does not support 64-bit mode.

5.2. **The Matrices.** We used a suite of 77 matrices in the evaluation of our code. The test suite includes most of the matrices that were used in articles [1, 10, 6, 14, 8, 30], as well as 4 new matrices[4]. The only matrices from this set that we did not use were matrices that we could not find[5],

[3]A note to the editor and the referees: unfortunately, this machine has been recently decommissioned, so we will be unable to run additional experiments on it.

[4]cage8, cage9, cage10, cage11.

[5]inaccura, comp2c, invextr1, mil053, mixtank, tib, wang3old, olaf1, av4408.

	Name	Classification	Order	1000's of nonzeros	Symmetry
1	rim	symmetric	22560	1015	0.64
2	twotone	circuit	120750	1206	0.24
3	zhao2	symmetric	33861	167	0.92
4	psmigr_1	unsymmetric	3140	543	0.48
5	ex11	symmetric	16614	1097	1.00
6	raefsky3	symmetric	21200	1489	1.00
7	raefsky4	symmetric	19779	1317	1.00
8	fidap011	symmetric	16614	1091	1.00
9	fidapm11	symmetric	22294	61787	1.00
10	wang4	circuit	26068	177	1.00
11	cage10	symmetric	11397	151	1.00
12	bbmat	symmetric	38744	1772	0.53
13	av41092	symmetric	41092	1684	0.00
14	mark3jac140	unsymmetric	64089	376	0.07
15	xenon1	symmetric	48600	1181	1.00
16	g7jac200	unsymmetric	59310	718	0.03
17	li	symmetric	22695	1215	1.00
18	ecl32	circuit	51993	380	0.92

Table 1. The large matrices that we use to measure speedups.

matrices that our code could not read[6], and a few matrices that we omitted due to oversight[7].

Some of the graphs that present the results of our experiments partition the matrices into three sets: highly symmetric structure, highly unsymmetric structure, and circuit-simulation matrices. Matrices were classified as circuit-simulation matrices if they were clearly labeled as such. Matrices were classified as highly symmetrically structured if more than 50% of the entries are matched ($a_{ij} \neq 0$ and $a_{ji} \neq 0$), and as highly unsymmetric otherwise.

We present speedup results only for the largest matrices that our code was able to solve on a uniprocessor. The selection criterion for these matrices was a factorization time of 20 seconds or more (by our code). These matrices are listed in Table 1. We do not claim that our code scales well on matrices that can be factored in several seconds on a uniprocessor.

Several of these matrices were not successfully factored by all codes. We document these matrices and the reasons for the failures, where we could determine the reason.

[6]This includes mostly symmetric and rectangular matrices that our matrix-import code could not handle: nasarb, bcsstk08/28/16, plat1919, eris1176, bscpwr10, finan512.

[7]gemat11, wang3, west2021.

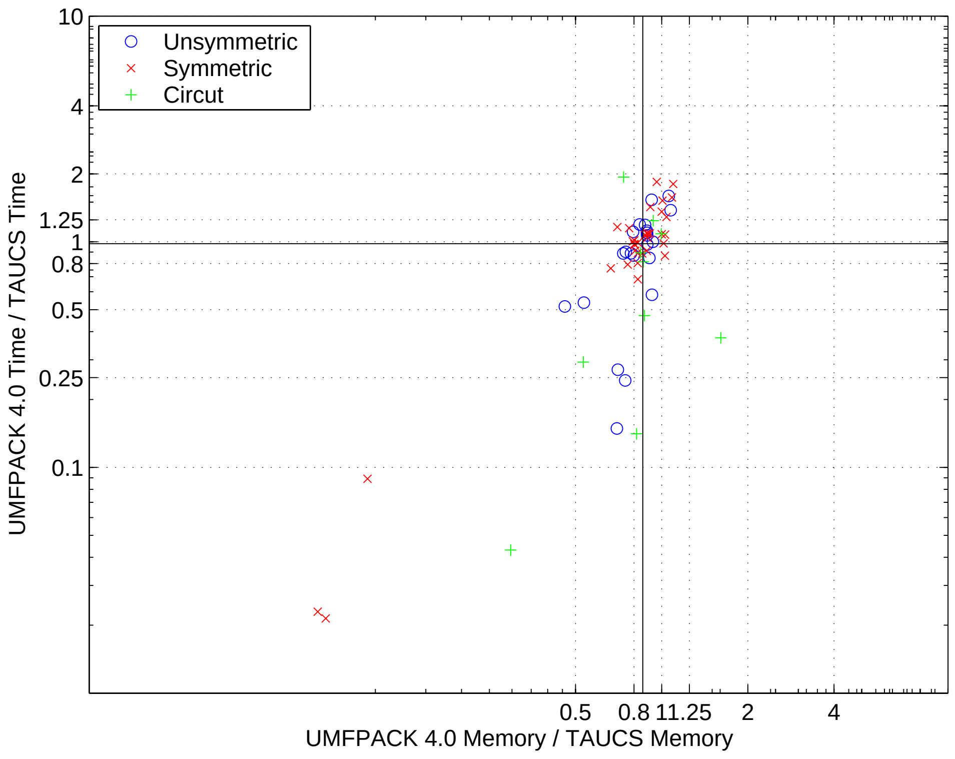

FIGURE 5.1. The performance of our new code relative to that of UMFPACK 4.0 on a single processor. The solid lines represent the medians of the data points. Each symbol on the plot represents one matrix. Data points higher than 1 represent better performance of our code.

Four of the matrices were too large to solve within the 32-bit address-space constraint: circuit_4, cage11, pre2, and xenon2. None of the codes was able to factor these matrices. On two matrices, e40r0000 and e40r5000, UMFPACK produces solution with a large residual; the two other codes exhibited no such problem on these matrices. On three matrices, shyy161, shyy41, and rw5151 all three codes produced solutions with poor residuals. Our code crashed on one matrix, mahindas (a small matrix, factored in less than 0.1 seconds by both UMFPACK and SuperLU_MT); we have not been able to determine the reason for this failure. SuperLU_MT always failed on two matrices, ecl32 and li, probably due to lack of memory.

5.3. **The Results of the Experiments.** Figures 5.1 and 5.2 show that on a uniprocessor, our new code performs well compared to both SuperLU_MT and to UMFPACK. The design of the plots shown in these figures is taken from [9]. The running-time-ratio median line for UMFPACK is very close to 1, which implies that our code is faster than UMFPACK on roughly the same

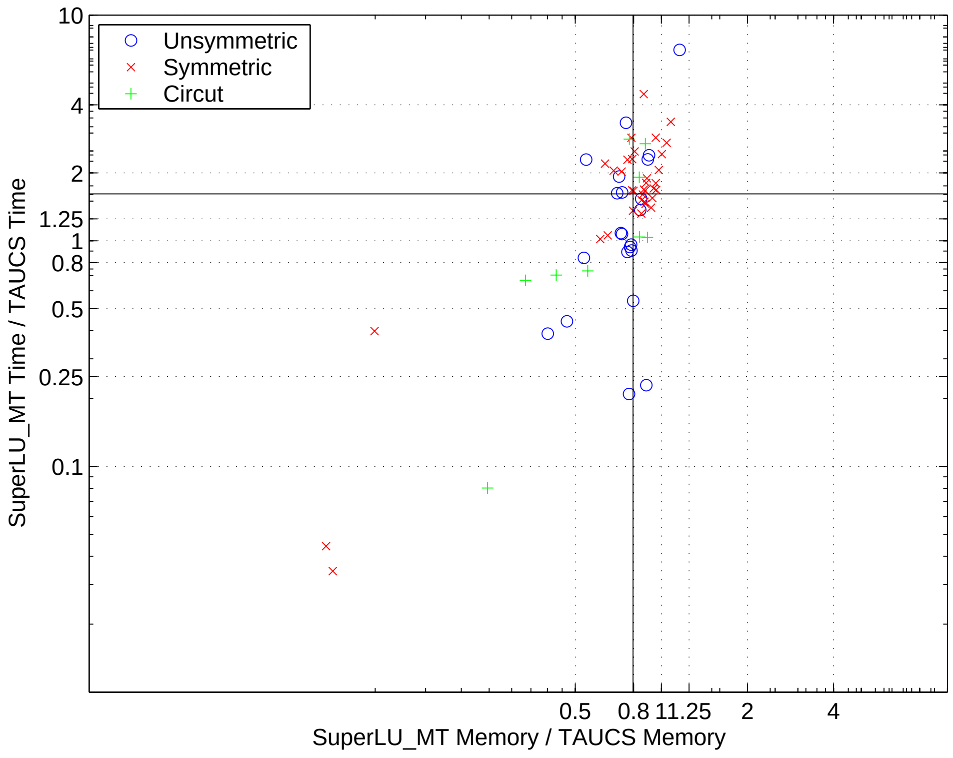

FIGURE 5.2. The performance of our new code relative to that of SuperLU_MT on a single processor.

number of matrices as those on which UMFPACK is faster. The running-time-ratio median for SuperLU_MT shows that our code is faster than SuperLU_MT on more matrices than the other way around. The memory-ratio median lines show that on many matrices our code uses more memory than the two other codes.

Our code is never more than twice as fast as UMFPACK, but on a few matrices, it is much slower. All of these are matrices that can be factored very quickly by all codes. Our code is sometimes more than 4 times faster than SuperLU_MT; as we show later, this happens even on large matrices.

The plots also show that when our code is slow, it also uses much more memory. There does not seem to be a correlation between they type of matrix, as defined in [9] (symmetric, unsymmetric, and circuit simulation) and the behavior of our code relative to other codes.

Figure 5.3 compares the uniprocessor performance of our code to that of UMFPACK and SuperLU_MT, but only on the 18 large matrices (factorization times larger than 20 seconds). On most of these matrices, our code is slightly faster than UMFPACK. Except for one of these matrices ecl32, our code is never much slower. On ecl32 our code is significantly slower than UMFPACK.

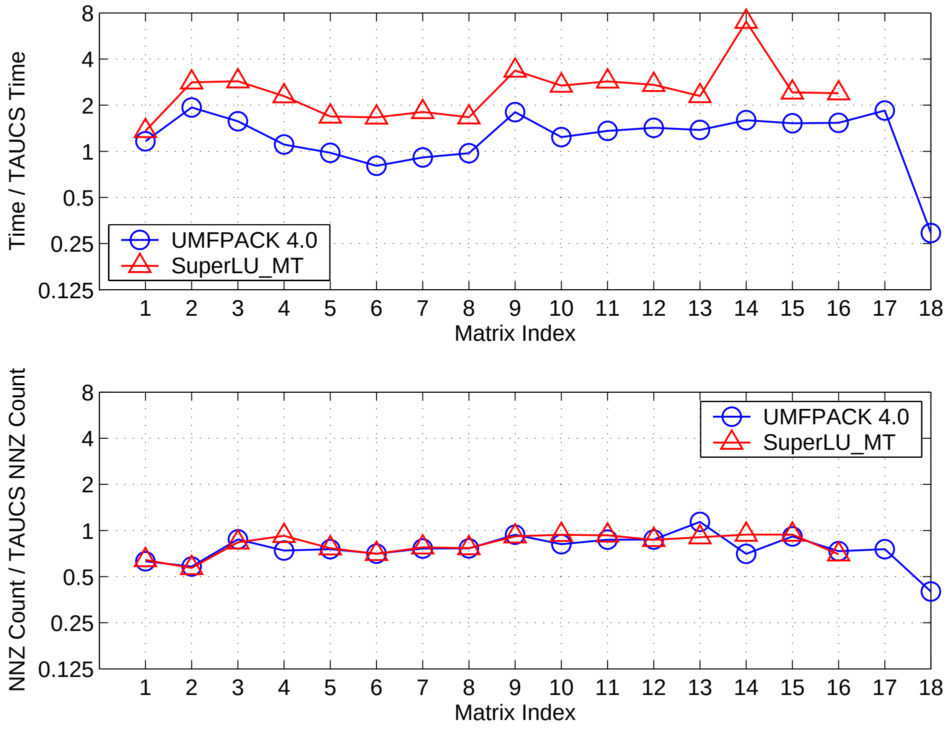

FIGURE 5.3. The uniprocessor performance of our new code relative to that of UMFPACK 4.0 and SuperLU_MT, on the 18 largest matrices in our test suite. The graph on top shows factorization-time ratios, the one on the bottom ratios of nonzero-counts in the computed factors.

The number-of-nonzeros ratios show that the poor performance of our code on this matrix is correlated with a higher nonzero count: ecl32 is the only matrix on which our code generates more than twice as many nonzeros as the other codes. Our code does generate more nonzeros than the other codes on many of the large matrices, but not by a large factor. The higher nonzero counts probably reflect our aggressive supernode amalgamation strategy.

On the large matrices and a single processor, our code is always faster than SuperLU_MT, often by more than a factor of 2 and once by an even larger factor.

Figure 5.4 presents the speedups that our code achieves on the large matrices. On 2 processors, the behavior is fairly uniform: the code speeds up by a factor of 1.5 to 1.8. On larger numbers of processors, the speedups are less uniform, and tend to improve with the cost of the factorization. On 4 processors, speedups often approach 3 (and sometimes slightly higher). Increasing the number of processors from 4 to 8 improves the running times significantly, with speedups around 4 for the largest matrices. Increasing the

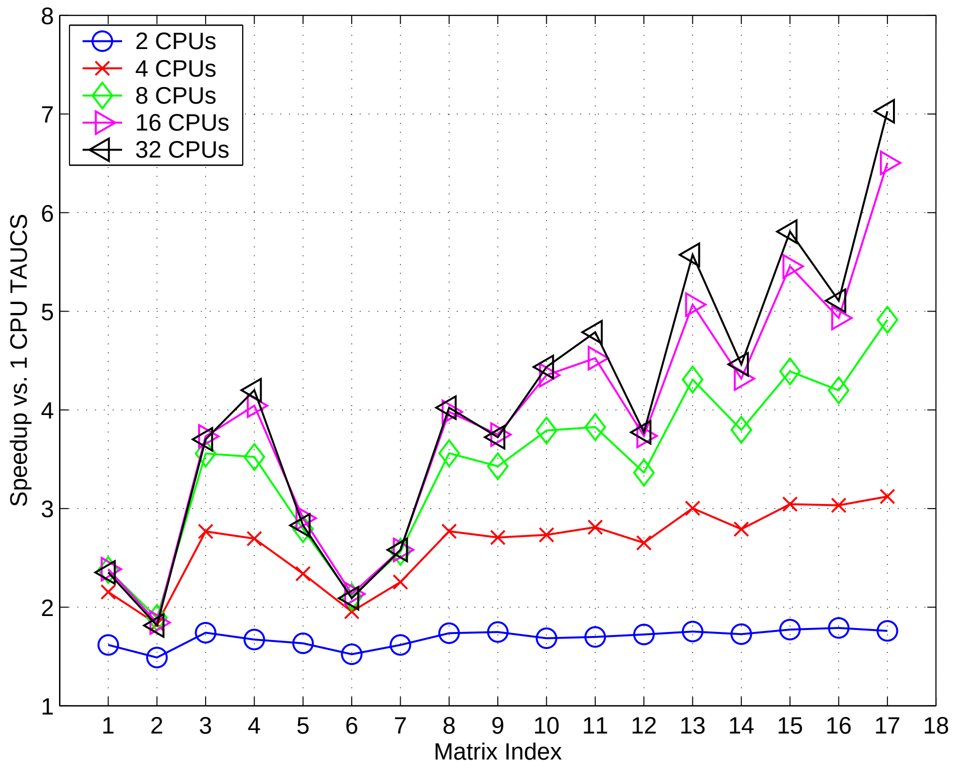

FIGURE 5.4. Speedups of our code relative to the uniprocessor factorization times. This plot focuses on the 18 largest matrices, except that our code failed to solve matrix 18 in parallel.

number of processors to 16 and then 32 improves the absolute performance, but not significantly. Performance never drops significantly with increasing numbers of processors.

Figure 5.5 compares the running times of our codes to that of SuperLU_MT on 1–16 processors, on the large matrices. We were unable to run SuperLU_MT on 32 processors, and it also sometimes failed on smaller numbers of processors. The usual behavior in these cases seemed to be an infinite loop. We are uncertain as to what exactly caused these failures. The data in the figure shows that on up to 4 processors, our code is almost always faster than SuperLU_MT, and never significantly slower. On 8 and 16 processors, SuperLU_MT is sometimes faster; on a few matrices by a factor of about 1.5, and on one, by a factor of 2. On the 10 largest matrices in this group, our code is almost always faster and never significantly slower. The data in this graph demonstrates that the parallel performance of our code achieves similar to that obtained by another state-of-the-art parallel factorization code.

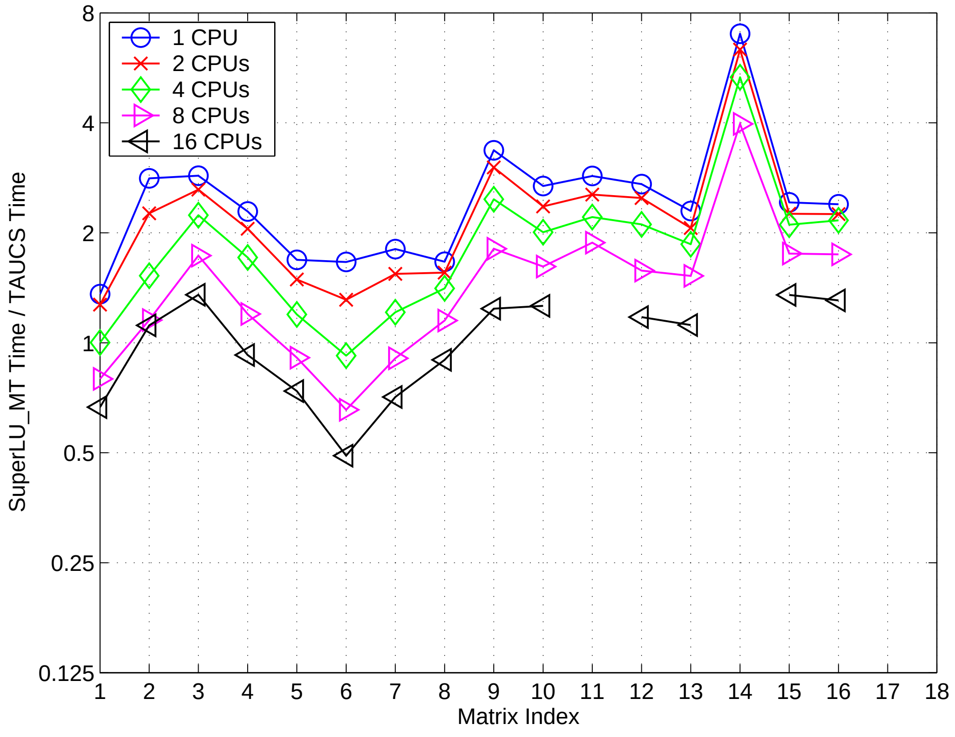

FIGURE 5.5. The performance of SuperLU_MT relative to that of our new code on 1, 2, 4, 8, and 16 processors, on the 18 largest matrices. Some of the data points for matrices 11, 14, and 17 are missing because SuperLU_MT failed to factor the matrices. Our code was able to factor matrix 18 on one processor but not on more; SuperLU_MT was not able to factor it at all.

6. Conclusions

The main question that our research aimed to resolve was whether the unsymmetric-pattern multifrontal partial-pivoting sparse LU factorization can be effectively parallelized. We believe that our results demonstrate that this class of algorithms can indeed be effectively parallelized.

Our methodology has been to produce a sequential code whose performance is on par with that of a state-of-the-art unsymmetric-pattern multifrontal partial-pivoting sparse LU code, UMFPACK 4.0, and to parallelize it. We then compared the parallel performance to that of another partial-pivoting sparse LU code, SuperLU_MT. In most cases, our code is faster than SuperLU_MT. These results establish our main conclusion, that the unsymmetric-pattern multifrontal partial-pivoting sparse LU factorization can be effectively parallelized.

Partial pivoting algorithms, and more generally partial pivoting algorithms using column preordering, have advantages over the two other forms of numerical pivoting that are used in sparse LU codes. First, algorithms that preorder the column and stick to that ordering (up to equivalent exchanges) guarantee an a-priori bound on fill and arithmetic operations. In contrast, the other common form of dynamic numerical pivoting, called delayed pivoting, does not provide any a-priori guarantees. Second, algorithms that incorporate dynamic numerical pivoting are more reliable than static-pivoting algorithms, like SuperLU_DIST [33], that preorder both the rows and the columns. We do not claim that partial pivoting is an absolute necessity: static-pivoting codes [33] and delayed-pivoting codes [30] have been shown to be effective in practice. But partial pivoting does have the advantages that we mentioned.

There are two algorithmic approaches to the sparse partial-pivoting LU factorization: the left-looking approach [14, 23] and the unsymmetric-pattern multifrontal approach [9]. The left-looking approach has a theoretical advantage over the multifrontal approach, in that the total number of operations performed by the algorithm is proportional to the number of arithmetic operations [23]. No such bound is known for the multifrontal approach. However, we have found that UMFPACK, the implementation of the multifrontal approach, is often faster than SuperLU, the best implementation of the left-looking approach. It is hard to determine whether the difference is inherent to the algorithms or due to the different implementations, but since SuperLU has already been parallelized [15], we decided to try to parallelize the multifrontal algorithm.

One interesting question remains open: can the unsymmetric-pattern multifrontal algorithm be implemented in space proportional to that of the resulting factors, and in total operation count proportional to the arithmetic operations? The left-looking approach has these properties, but they are not necessarily true for UMFPACK and not necessarily true for our code. We believe that such an algorithm is highly desirable, even if it will be a little slower in practice than delayed- and static-pivoting algorithms.

References

[1] Patrick R. Amestoy and Chiara Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 24(2):553–569, 2002.

[2] 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 User's Guide*. SIAM, Philadelphia, PA, 2nd edition, 1994. Also available online from `http://www.netlib.org`.

[3] Cleve Ashcraft and Roger Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[4] Igor Brainman and Sivan Toledo. Nested-dissection orderings for sparse LU with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 23:998–112, 2002.

[5] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. Technical Report TR-03-006, Department of Computer and Information Science and Engineering, University of Florida, 2003.

[6] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18:140–158, 1997.

[7] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Transactions on Mathematical Software*, 25:1–19, 1999.

[8] Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. Technical Report TR-03-006, Department of Computer and Information Science and Engineering, University of Florida, May 2003.

[9] Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):165–195, 2004.

[10] Timothy A. Davis and Iain S. Duff. Unsymmetric-pattern multifrontal methods for parallel sparse LU factorization. Technical Report TR-91-023, Department of Computer and Information Science and Engineering, University of Florida, January 1991.

[11] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. A column approximate minimum degree ordering algorithm. Technical Report TR-00-005, Department of Computer and Information Science and Engineering, University of Florida, 2000.

[12] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 30(3):377–380, September 2004.

[13] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. A column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 30(3):353–376, September 2004.

[14] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20:720–755, 1999.

[15] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20:915–952, 1999.

[16] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20:915–952, 1999.

[17] Jack J. Dongarra, Jeremy Du Cruz, Sven Hammarling, and Ian Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

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

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

[20] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, 1998.

[21] Alan George and Esmond Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM Journal on Scientific and Statistical Computing*, 8:877–898, 1987.

[22] J. R. Gilbert, X. S. Li, E. G. Ng, and B. W. Peyton. Computing row and column counts for sparse QR and LU factorization. *BIT Numerical Mathematics*, 41(4):693–710, 2001.

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

[24] John R. Gilbert. *Graph Separator Theorems and Sparse Gaussian Elimination.* PhD thesis, Stanford University, 1980.

[25] John R. Gilbert. An efficinet parallel sparse partial pivoting algorithm. Technical Report 88/45052-1, Christian Michelsen Institute, Bergen, Norway, 1988.

[26] John R. Gilbert and Esmond 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.

[27] John R. Gilbert and Tim Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9:862–874, 1988.

[28] John R. Gilbert and Robert Schreiber. Nested dissection with partial pivoting. In *Sparse Matrix Symposium 1982: Program and Abstracts*, page 61, Fairfield Glade, Tennessee, October 1982.

[29] Nicholas I. M. Gould and Jennifer A. Scott. A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations. *ACM Transactions on Mathematical Software*, 30(3):300–325, September 2004.

[30] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 24:529–552, 2002.

[31] Anshul Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, 28(3):301–324, September 2002.

[32] Dror Irony, Gil Shklarski, and Sivan Toledo. Parallel and fully recursive multifrontal supernodal sparse cholesky. *Future Generation Computer Systems*, 20(3):425–440, April 2004.

[33] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29:110–140, 2003.

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

[35] Joseph W. H. Liu, Esmond G. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14:242–252, 1993.

[36] Esmond G. Ng and Barry W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14(5):1034–1056, 1993.

[37] Edward Rothberg and Anoop Gupta. Efficient sparse matrix factorization on high-performance workstations—exploiting the memory hierarchy. *ACM Transactions on Mathematical Software*, 17(3):313–334, 1991.

[38] Elad Rozin and Sivan Toledo. Locality of reference in sparse Cholesky factorization methods. Submitted to the *Electronic Transactions on Numerical Analysis*, 2004.

[39] Supercomputing Technologies Group, MIT Laboratory for Computer Science, Cambridge, MA. *Cilk-5.3.2 Reference Manual*, November 2001. Available online at `http://supertech.lcs.mit.edu/cilk`.