# Algorithm 9xx, SuiteSparseQR: multifrontal multithreaded rank-revealing sparse QR factorization

TIMOTHY A. DAVIS
University of Florida

SuiteSparseQR is a sparse QR factorization package based on the multifrontal method. Within each frontal matrix, LAPACK and the multithreaded BLAS enable the method to obtain high performance on multicore architectures. Parallelism across different frontal matrices is handled with Intel's Threading Building Blocks library. The symbolic analysis and ordering phase pre-eliminates singletons by permuting the input matrix $A$ into the form $[R_{11}\, R_{12}\, ;\, 0\, A_{22}]$ where $R_{11}$ is upper triangular with diagonal entries above a given tolerance. Next, the fill-reducing ordering, column elimination tree, and frontal matrix structures are found without requiring the formation of the pattern of $A^T A$. Approximate rank-detection is performed within each frontal matrix using Heath's method. While Heath's method is not always exact, it has the advantage of not requiring column pivoting and thus does not interfere with the fill-reducing ordering. For sufficiently large problems, the resulting sparse QR factorization obtains a substantial fraction of the theoretical peak performance of a multicore computer.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*linear systems (direct methods), sparse and very large systems*; G.4 [**Mathematics of Computing**]: Mathematical Software—*algorithm analysis, efficiency*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: QR factorization, least-square problems, sparse matrices

## 1. INTRODUCTION

Sparse QR factorization is one of the key direct methods for solving large sparse linear systems and least-squares problems. Typically, orthogonal transformations such as Givens rotations [Givens 1958] or Householder reflections [Householder 1958] are applied to a sparse matrix $A$ to obtain the factorization $A = QR$, where $Q$ is orthogonal and $R$ is upper triangular. The columns of $A$ are often permuted to reduce fill-in, resulting in the factorization $AP = QR$. The resulting factors can be used to solve a least-squares problem ($\min_x ||b - Ax||_2$), to find the basic solution

Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, USA. email: davis@cise.ufl.edu. http://www.cise.ufl.edu/∼davis. Portions of this work were supported by the National Science Foundation, under grants 0203270, 0620286, and 0619080.

of an under-determined system $Ax = b$, or to find a minimum 2-norm solution of an under-determined system $A^T x = b$.

The earliest sparse QR factorization methods operated on $A$ one row or column at a time ([George et al. 1988; George and Heath 1980; Heath 1982; Heath and Sorensen 1986]; see also [Björck 1996; Davis 2006]). This includes the prior sparse QR factorization in MATLAB [Gilbert et al. 1992], which SuiteSparseQR has replaced as of MATLAB R2009a. These prior methods are unable to reach a substantial fraction of the theoretical peak performance of modern computers because of their irregular access of memory, although they are very competitive when $R$ is very sparse. The row-merging method [Liu 1986] introduced the idea that groups of rows could be handled all at once. This idea was fully realized in the sparse *multifrontal* QR factorization method. In the multifrontal method, the factorization of a large sparse matrix is performed in a sequence of dense frontal matrices; the idea was first used for symmetric indefinite matrices [Duff and Reid 1983; Duff 2004] and later extended to sparse LU [Amestoy and Duff 1989; Amestoy and Puglisi 2002; Duff and Reid 1984; Davis and Duff 1997; Davis 2004] and sparse Cholesky factorization [Liu 1989]. Prior multifrontal sparse QR methods include those of [Matstoms 1994; 1995], [Amestoy et al. 1996], [Lu and Barlow 1996], [Sun 1996], [Pierce and Lewis 1997], and [Edlund 2002].

SuiteSparseQR is a multithreaded multifrontal sparse QR factorization method that is an extension of these prior methods, and its unique features are discussed here. Additional background, definitions, and examples are given to make this discussion self-contained. Sections 2 and 3 present the symbolic analysis and numeric factorization methods used in SuiteSparseQR; in both of these sections, the methods used in SuiteSparseQR are first described, and then compared with prior sparse QR methods. The two techniques used in SuiteSparseQR for exploiting multicore architectures are discussed in Section 5. Comparisons with existing sparse QR factorization methods are given in Section 6. Finally, Section 7 discusses the use of the SuiteSparseQR software package. Throughout the paper, fixed width font (`x=A\b`, for example) denotes MATLAB notation, except in the figures. The notation $||x||$ refers to the 2-norm of $x$, while $|x|$ refers to the number of nonzero entries in $x$.

## 2. ORDERING AND SYMBOLIC ANALYSIS

Direct methods for sparse linear systems typically split into three distinct phases: (1) ordering and symbolic analysis, typically based solely on the nonzero pattern of the matrix, (2) numeric factorization, and (3) a solve phase, which uses the factorization to solve a linear system or least-squares problem. This framework allows phase (1) to be performed once and reused multiple times when dealing with multiple matrices with different numerical values but identical nonzero pattern.

### 2.1 Exploiting Singletons

The first step in the symbolic analysis is to remove singletons. A *singleton* is a column $j$ with a single nonzero entry $a_{ij}$ whose magnitude is larger than a given threshold $\tau$, or a column $j$ with no entries at all. The latter can only occur when the matrix is *structurally* rank-deficient. A matrix has full structural rank if a permutation exists to make the diagonal zero-free. A *numerically* rank-deficient matrix is either mathematically rank-deficient or small changes would make it so.

If a singleton exists, its column $j$ is permuted to the left. If column $j$ has a single nonzero $a_{ij}$, row $i$ is permuted to the top. Column $j$ and row $i$ (if the row exists) are removed from $A$. The process repeats until no more singletons exist, leading to

$$\begin{bmatrix} R_{11} & R_{12} \\ 0 & A_{22} \end{bmatrix},$$

where $R_{11}$ is upper triangular or upper trapezoidal (the latter can occur if $A$ is structurally rank-deficient). The diagonal entries of $R_{11}$ are greater than $\tau$, and thus $R_{11}$ is *estimated* to have full numerical rank. This is of course an approximation; $R_{11}$ may be rank deficient but this will not be detected.

The sparse `qr` in MATLAB R2008b uses $\tau = 20(m+n)\epsilon \max_j ||A_{*j}||_2$, where $\epsilon$ is the machine roundoff (about $10^{-16}$ since only a double-precision version is provided), $\max_j ||A_{*j}||_2$ is the largest 2-norm of any column of $A$, and where $A$ is $m$-by-$n$. This value of $\tau$ and has worked well in practice. SuiteSparseQR is the new sparse `qr` in MATLAB, so it uses the same default $\tau$.

Singletons frequently arise in real applications. However, the determination of a singleton depends on the numerical values of $A$ the threshold $\tau$. Thus, singletons are not exploited if the symbolic analysis of one matrix is reused for the numeric factorization of another matrix with the same pattern but different values.

The sparse QR factorization for $R_{11}$ requires no numerical work at all, and no fill-in is incurred in $R_{11}$ or $R_{12}$. The time taken to find and remove all singletons is only $O(n + |R_{11}| + |R_{12}|)$. Once singletons are removed, the remaining matrix $A_{22}$ is ordered, analyzed, and factorized with a multifrontal sparse QR method.

No prior multifrontal sparse QR method exploits singletons, although MA49 [Amestoy et al. 1996] and Edlund's method [Edlund 2002] both have an option for using a permutation to block triangular form (BTF) [Duff 1977; 1981; Pothen and Fan 1990]. Computing the BTF takes much more time than finding singletons, but it finds not only all singletons, but larger diagonal blocks as well. Employing the BTF can reduce the numerical work required to compute the QR factorization, and would enable SuiteSparseQR to exactly predict the fill-in when the block upper triangular form of $A$ consists of more than one block [Coleman et al. 1986].

The MATLAB function `dmperm` computes the BTF and was written by this author [Davis 2006]. It is thus readily available for incorporation into SuiteSparseQR. However, SuiteSparseQR does not exploit the BTF ordering since it is suitable only for full-rank matrices. With BTF, the permuted matrix $A$ would be factorized into $QR + B$ where $B$ contains the unfactorized off-diagonal blocks. SuiteSparseQR uses Heath's method to handle rank-deficiency by "squeezing out" the linearly-dependent part of $B$. This cannot be done if $B$ is not factorized. See Section 3.2 for details of how rank-deficiency is handled.

### 2.2 Fill-reducing ordering

After singletons are found (if any), the remainder of the ordering and symbolic analysis phase depends solely on the nonzero pattern of the remaining matrix. Henceforth, let $A$ refer to the matrix after singletons have been removed.

If the BTF form of the matrix $A$ has a single block, the nonzero pattern of the factor $R$ is identical to the nonzero pattern of the Cholesky factorization $L^T$ of $A^T A$ [Coleman et al. 1986; George and Heath 1980]. Otherwise, the pattern of $L^T$ is an

upper bound on the pattern of $R$. The symbolic analysis is thus modeled after the Cholesky factorization of $A^TA$. SuiteSparseQR uses the CHOLMOD sparse Cholesky package [Chen et al. 2008; Davis and Hager 2009] for its ordering and analysis phase, which has the following ordering methods:

—COLAMD on $A$, which orders $A^TA$ without forming it [Davis et al. 2004a; 2004b]. With this option, SuiteSparseQR performs its entire symbolic analysis and numerical factorization without forming $A^TA$. This is the default.
—AMD on the explicit pattern of $A^TA$ [Amestoy et al. 1996; 2004]. AMD is an acronym for Approximate Minimum Degree, not to be confused with AMD, Inc.
—METIS [Karypis and Kumar 1998] applied to the explicit pattern of $A^TA$.
—CHOLMOD's nested dissection ordering based on METIS.
—More than one of the above, where multiple methods can be tried and the ordering with the least fill in the Cholesky factorization of $(AP)^TAP$ used.

SuiteSparseQR now operates solely on the permuted matrix $AP$, and thus henceforth in this paper, $P$ will be dropped and $A$ will refer to the permuted matrix after the fill-reducing ordering is applied.

The asymptotic run times of these ordering methods have no tight known bounds in terms of quantities that can be readily calculated beforehand. However, experimental results have shown that COLAMD and AMD, with occasional exceptions, take time roughly proportional to the number of nonzeros in $A$ and $A^TA$, respectively. This is not an upper bound, but an average based on experiments with nearly 2000 matrices from real applications in the Univ. of Florida Sparse Matrix Collection [Davis and Hu 2011].

It is normally very straightforward for any multifrontal sparse QR to adapt to any fill-reducing ordering. MA49 uses AMD on the explicit matrix $A^TA$. [Lu and Barlow 1996] use nested dissection [George 1973; Lipton et al. 1979], which also requires $A^TA$. [Pierce and Lewis 1997] and [Matstoms 1994] do not discuss the fill-reducing ordering, but their methods were developed before $A^TA$-free methods such as COLAMD were available. Edlund optionally uses COLAMD [Edlund 2002].

### 2.3 Symbolic factorization

After finding a fill-reducing ordering, CHOLMOD performs the symbolic Cholesky factorization $LL^T$ of the permuted form of $A^TA$, by analyzing $A$ instead of forming $A^TA$ explicitly. The analysis proceeds with the Cholesky factor $L$, but to keep the discussion clear this paper will henceforth only discuss $R$.

2.3.1 *The column elimination tree.* The analysis starts by computing the *elimination tree* of $A^TA$ and the number of nonzeros in each row of $R$ (the *row counts*), using the methods of [Gilbert et al. 2001]. The elimination tree of $A^TA$ has one node per column of $A$. The parent of $i$ is $k$ if $k$ is the smallest row index $k > i$ such that $r_{ik} \neq 0$. This tree is required for the symbolic analysis and it also describes the data dependencies for the numeric factorization. The elimination tree of $A^TA$ is also called the column elimination tree of $A$, since each node corresponds to a column of $A$. Henceforth, this paper will use the latter term. The time taken by this step is roughly $O(|A|)$, where "roughly" is theoretically at worst $O(|A| \log n)$. In practice, this step takes essentially $O(|A|)$ time.

A more concise description of the pattern of $R$ when the BTF of $A$ has more than one block could be obtained from the row-merge tree [Liu 1986; George and Ng 1987; Oliveira 2001; Grigori et al. 2007], in place of the column elimination tree. However, SuiteSparseQR handles rank-deficient matrices using the method of [Heath 1982], which requires $R$ to accommodate any nonzero entry in the Cholesky factorization of $A^T A$. This requires the column elimination tree and thus SuiteSparseQR cannot exploit the row-merge tree.

2.3.2 *Finding supernodes.* A *supernode* in $R$ represents a group of adjacent $t$ rows with identical or nearly identical nonzero pattern. Each supernode in $R$ gives rise to a single corresponding *frontal matrix* for the numeric QR factorization. A frontal matrix is a dense submatrix of $A$ in which portions of the QR factorization of $A$ are performed. It contains a subset of the rows of $A$. Once factorized, its first $t$ rows contain the rows of the corresponding supernode of $R$. Its first $t$ columns are called the *pivotal columns* of the frontal matrix. A frontal matrix is normally much smaller than the matrix $A$ being factorized.

The row counts and column elimination tree determine the *exact* supernodes. Two rows $i$ and $i+1$ are in the same *exact* supernode if the parent of $i$ is $i+1$ and if $|R_{i*}| = |R_{i+1,*}| + 1$. These facts imply that the two rows in $R$ have identical nonzero pattern (except for $i$, which appears only in $R_{i*}$). This test does not require the nonzero pattern of $R$ itself (which can take time $O(|R|)$ to find), but just the row counts and column elimination tree (which take essentially $O(|A|)$ time to find).

Time and memory usage are typically decreased by exploiting *relaxed* supernodes. Two adjacent rows are combined into a relaxed supernode if the nonzero patterns of $R_{i,i+1:n}$ and $R_{i+1,*}$ are identical or similar. Relaxed supernodes are found solely from the counts of nonzeros in each row of $R$. They are always exploited, so the terms *exact* and *relaxed* are henceforth dropped.

Once the rows are grouped into supernodes, the nonzero pattern of each supernode is computed, taking time proportional to the number of nonzeros in the topmost row of each supernode. This is also the amount of integer memory space required to represent the supernodal nonzero pattern of $R$. If $R$ is dense it consists of a single supernode, so this step can take as little as $\Omega(n)$ time. In any case, $O(|R|)$ is a loose upper bound on the time; it is typically much less than this because of supernodes. The result is a supernodal representation of the nonzero pattern of $R$.

2.3.3 *The frontal matrix tree.* The column elimination tree has $n$ nodes, one per column of $A$. Grouping together columns in the same supernode gives the *frontal matrix tree*, with one node in the frontal matrix tree per supernode/frontal matrix. For those readers familiar with the supernodal method, the frontal matrix tree for sparse multifrontal QR of $A$ is the same as a *supernodal elimination tree* for a supernodal Cholesky factorization of $A^T A$ [Ashcraft and Grimes 1989].

2.3.4 *Sorting the rows of A.* After CHOLMOD's supernodal analysis, SuiteSparseQR sorts the rows of $A$ according to the column index of the leftmost nonzero in each row of $A$. This results in the permuted matrix $P_2 A$. A row $i$ of $P_2 A$ is assembled in the frontal matrix whose pivotal columns contain the leftmost column index $j$ of row $i$. The time for this step is $O(|A|)$.

2.3.5 *Finding the size of each frontal matrix.* The number of columns in each frontal matrix is given by the number of nonzeros in the leading row of the corresponding supernode in $R$. To find the number of rows in each frontal matrix, SuiteSparseQR simulates the sparse multifrontal QR factorization. The amount of work required to perform the Householder QR factorization of each frontal matrix is determined. The *staircase* for each frontal matrix is found, which is the row index of the last nonzero entry in each column. The memory usage and work may be less than this estimate if the matrix $A$ is rank-deficient. The time taken by this step is proportional to the number of integers required to represent the supernodal pattern of $R$, which is typically much less than $O(|R|)$.

An estimate of the floating-point work in each front is computed (this estimate is exact if the rank-detecting tolerance $\tau$ is disabled). This pass also determines the size of each frontal matrix and the stack size needed for the contribution blocks for a sequential numeric factorization.

2.3.6 *Total time and memory usage for the symbolic phase.* The total time and memory usage for the analysis is roughly $O(|A| + |R|)$, excluding the ordering time (which is quite often $O(|A|)$ in practice), and where $|R|$ is the number of integers required to represent the supernodal structure of the matrix $R$. This can be much less than the time and memory required to form $A^T A$, particularly when $m << n$. It is also much less than the time taken by the numeric factorization step.

2.3.7 *An example.* Figure 1 gives an example of the multifrontal sparse QR factorization of a small sparse matrix $A$. The rows of $A$ have been sorted according to the column index of their leftmost nonzero entry. The factor $R$ is shown to the right, with each supernode of $R$ consisting of an adjacent set of rows whose diagonal block is a full upper triangular matrix and whose rows otherwise have identical nonzero patterns. The horizontal lines in $A$ subdivide the rows according to the frontal matrix into which they are first assembled. A dot (.) is shown in $A$ for an entry that becomes structurally nonzero in the corresponding frontal matrix. The column elimination tree is shown in the bottom right of the figure, with supernodes shown as rounded boxes. The parent of $i$ in this tree is given by the smallest $k > i$ for which $r_{ik} \neq 0$. This example is continued in Section 3, which shows the assembly and numeric factorization of two of the five frontal matrices.

2.3.8 *Comparison with prior methods.* None of the prior multifrontal sparse QR methods cited in this paper use the results of [Gilbert et al. 2001] (namely, the ability to analyze $A^T A$ without forming it explicitly). All but one use the elimination tree of $A^T A$ and must form the nonzero pattern of $A^T A$ explicitly. [Edlund 2002] does not form $A^T A$, but finds a tree from the pattern of $R$ itself during an ordering method much like COLAMD; the resulting tree and pattern of $R$ would not be able to accommodate subsequent rank-detection, since it is not identical to the column elimination tree of $A$.

[Amestoy et al. 1996] point out that the exact amount of memory needed to hold the Householder vectors is difficult to predict with this strategy. However, the symbolic analysis phase of SuiteSparseQR precisely simulates the assembly of each frontal matrix, in $O(|A| + |R|)$ time, to find this exact amount if the rank-detecting threshold is disabled ($\tau < 0$). This corresponds to the case for MA49. When $\tau \geq 0$,

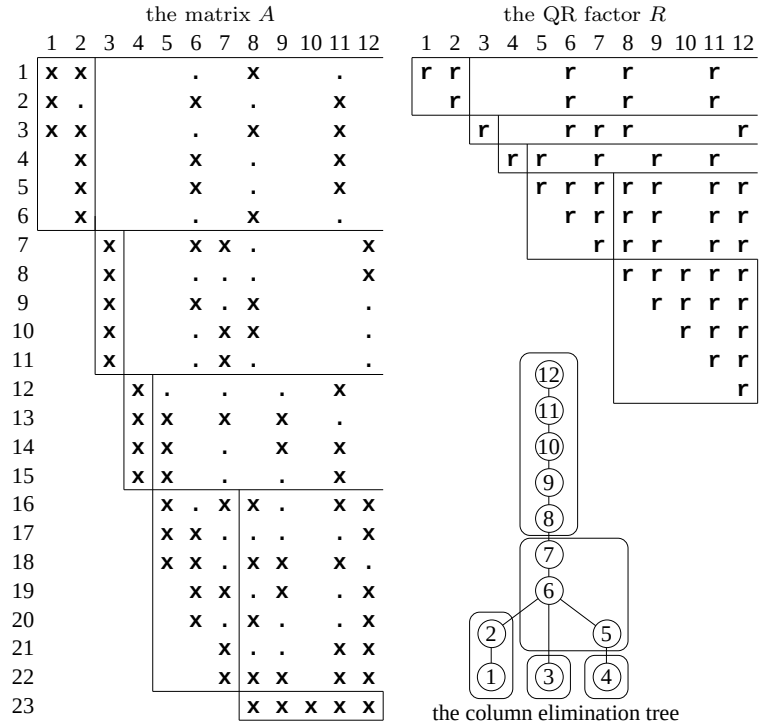


Fig. 1. A sparse matrix $A$, its factor $R$, and its column elimination tree

SuiteSparseQR's symbolic analysis finds an upper bound instead, which is the best that can be done since rank-detection cannot be done during symbolic analysis.

## 3. NUMERIC FACTORIZATION

For its numeric factorization, SuiteSparseQR uses much of the theory and algorithms from the prior implementations of multifrontal sparse QR factorization [Matstoms 1994; Amestoy et al. 1996; Lu and Barlow 1996; Pierce and Lewis 1997; Edlund 2002]. Described here are the key features that differ in SuiteSparseQR, plus enough background to make this discussion self-contained.

### 3.1 Frontal matrix assembly

For the assembly and factorization of its frontal matrices, SuiteSparseQR uses a modified version of Strategy 3 from MA49 [Amestoy et al. 1996] (refer to Figure 7 on page 284 of their paper).

This method is illustrated in Figures 2 and 3. Figure 2 shows the assembly of a leaf node in the frontal matrix tree corresponding to nodes 1 and 2 of the column elimination tree in Figure 1. In this example, rows 1, 2, and 3 of $A$ have a leftmost nonzero in column 1, and rows 4, 5, and 6 have a leftmost nonzero in column 2. The first two rows of $R$ have identical nonzero pattern and thus can be handled in a single frontal matrix. The corresponding columns, 1 and 2, are the pivotal columns for this front (Amestoy, Duff, and Puglisi refer to them as *fully-summed variables*;

rows of $A$ for front 1

| | 1 | 2 | 6 | 8 | 11 |
|---|---|---|---|---|---|
| 1 | x | x | . | x | . |
| 2 | x | . | x | . | x |
| 3 | x | x | . | x | x |
| 4 | | x | x | . | x |
| 5 | | x | x | . | x |
| 6 | | x | . | x | . |

factorized front 1

| | 1 | 2 | 6 | 8 | 11 |
|---|---|---|---|---|---|
| 1 | r | r | r | r | r |
| 2 | h | r | r | r | r |
| 3 | h | h | c | c | c |
| 4 | | h | h | c | c |
| 5 | | h | h | h | c |
| 6 | | h | h | h | h |

Fig. 2. Assembly and factorization of a leaf frontal matrix

child front 1

pivot cols 1 and 2
pivot rows 1 and 2

| | 6 | 8 | 11 |
|---|---|---|---|
| 3 | $c_1$ | $c_1$ | $c_1$ |
| 4 | | $c_1$ | $c_1$ |
| 5 | | | $c_1$ |

child 2

pivot col 3
pivot row 7

| | 6 | 7 | 8 | 12 |
|---|---|---|---|---|
| 8 | $c_2$ | $c_2$ | $c_2$ | $c_2$ |
| 9 | | $c_2$ | $c_2$ | $c_2$ |
| 10 | | | $c_2$ | $c_2$ |
| 11 | | | | $c_2$ |

child 3

pivot col 4
pivot row 12

| | 5 | 7 | 9 | 11 |
|---|---|---|---|---|
| 13 | $c_3$ | $c_3$ | $c_3$ | $c_3$ |
| 14 | | $c_3$ | $c_3$ | $c_3$ |
| 15 | | | $c_3$ | $c_3$ |

rows of $A$ for front 4

| | 5 | 6 | 7 | 8 | 9 | 11 | 12 |
|---|---|---|---|---|---|---|---|
| 16 | x | . | x | x | . | x | x |
| 17 | x | x | . | . | . | . | x |
| 18 | x | x | . | x | x | x | . |
| 19 | | x | x | . | x | . | x |
| 20 | | x | . | x | . | . | x |
| 21 | | | x | . | . | x | x |
| 22 | | | x | x | x | x | x |

assembled front 4

| | 5 | 6 | 7 | 8 | 9 | 11 | 12 |
|---|---|---|---|---|---|---|---|
| 16 | x | . | x | x | . | x | x |
| 17 | x | x | . | . | . | . | x |
| 18 | x | x | . | x | x | x | . |
| 13 | $c_3$ | . | $c_3$ | . | $c_3$ | $c_3$ | . |
| 19 | | x | x | . | x | . | x |
| 20 | | x | . | x | . | . | x |
| 3 | | $c_1$ | . | $c_1$ | . | $c_1$ | . |
| 8 | | $c_2$ | $c_2$ | $c_2$ | . | . | $c_2$ |
| 21 | | | x | . | . | x | x |
| 22 | | | x | x | x | x | x |
| 9 | | | $c_2$ | $c_2$ | . | . | $c_2$ |
| 14 | | | $c_3$ | . | $c_3$ | $c_3$ | . |
| 4 | | | | $c_1$ | . | $c_1$ | . |
| 10 | | | | $c_2$ | . | . | $c_2$ |
| 15 | | | | | $c_3$ | $c_3$ | . |
| 5 | | | | | | $c_1$ | . |
| 11 | | | | | | | $c_2$ |

factorized front 4, full−rank case

| | 5 | 6 | 7 | 8 | 9 | 11 | 12 |
|---|---|---|---|---|---|---|---|
| 16 | r | r | r | r | r | r | r |
| 17 | h | r | r | r | r | r | r |
| 18 | h | h | r | r | r | r | r |
| 13 | h | h | h | c | c | c | c |
| 19 | | h | h | h | c | c | c |
| 20 | | h | h | h | c | c | c |
| 3 | | h | h | h | h | h | c |
| 8 | | h | h | h | h | h | h |
| 21 | | | h | h | h | h | h |
| 22 | | | h | h | h | h | h |
| 9 | | | h | h | h | h | h |
| 14 | | | h | h | h | h | h |
| 4 | | | | h | h | h | h |
| 10 | | | | h | h | h | h |
| 15 | | | | | h | h | h |
| 5 | | | | | | h | h |
| 11 | | | | | | | h |

factorized front 4, column 6 linearly dependent

| | 5 | 6 | 7 | 8 | 9 | 11 | 12 |
|---|---|---|---|---|---|---|---|
| 16 | r | r | r | r | r | r | r |
| 17 | h | | r | r | r | r | r |
| 18 | h | | h | c | c | c | c |
| 13 | h | | h | h | c | c | c |
| 19 | | | h | h | h | c | c |
| 20 | | | h | h | h | h | c |
| 3 | | | h | h | h | h | h |
| 8 | | | h | h | h | h | h |
| 21 | | | h | h | h | h | h |
| 22 | | | h | h | h | h | h |
| 9 | | | h | h | h | h | h |
| 14 | | | h | h | h | h | h |
| 4 | | | | h | h | h | h |
| 10 | | | | h | h | h | h |
| 15 | | | | | h | h | h |
| 5 | | | | | | h | h |
| 11 | | | | | | | h |

Fig. 3. Assembly and factorization of a frontal matrix with three children

Pierce and Lewis call them *internal columns*). The other columns (6, 8, and 11) are *exterior* or *non-pivotal*.

Entries in $A$ are shown as an `x`. Zeros to the right of the leftmost nonzero that will become nonzero in a frontal matrix are shown as a dot (`.`). Assuming columns 1 and 2 are linearly independent, the QR factorization of this frontal matrix gives the result shown in the right half of Figure 2, with two rows of $R$ (the entries marked

`r`), a 3-by-3 upper triangular contribution block (the entries marked `c`) that must be assembled into the parent, and 5 Householder vectors (the entries marked `h`). In general, the contribution block $C$ can be upper trapezoidal.

Consider the parent of front 1. The parent (front 4) is the front which contains the first non-pivotal column of front 1 (in this case, column 6) as one of its pivotal columns. In Figure 3, the parent has three children, one of which is front 1 given in Figure 2. Only the $C$ blocks of these three children are shown in Figure 3; note that one of them is upper trapezoidal. The entries in the $C$ blocks are given subscripts according to the block in which they reside. The pivotal columns of the parent, front 4, are columns 5, 6, and 7. Rows 16 through 22 of $A$ have a leftmost nonzero entry in the pivotal columns of front 4. The staircase of the assembled front 4 is $(4, 8, 12, 14, 15, 16, 17)$, which gives the (0-based) row index of where the structural zeros start in each column of the frontal matrix.

The contribution blocks of the three children and rows 16 through 22 of $A$ are assembled into the front; this is shown in the bottom left of Figure 3. The Householder QR factorization is shown to its right. A 4-by-4 upper triangular contribution block remains to be assembled by the parent of front 4. The rank-deficient case (the rightmost frontal matrix in Figure 3) is discussed in the next section.

### 3.2 Frontal matrix factorization

Once the frontal matrix is assembled, the dense Householder QR factorization of the frontal matrix is found. If the pivotal columns of the frontal matrix are all linearly independent, this factorization is nearly the same as DGEQRF in LAPACK [Anderson et al. 1999]. SuiteSparseQR does not call DGEQRF; instead, it calls the functions that DGEQRF relies upon, namely:

(1) DLARFG, which constructs a single Householder vector,
(2) DLARF, which applies a single Householder vector,
(3) DLARFT, which constructs the $T$ matrix for a block of Householder vectors [Bischof and Van Loan 1987; Schreiber and Van Loan 1989], and
(4) DLARFB, which applies a block of Householder vectors.

DGEQRF uses a workspace of size $n$-by-$b$ where typically $b = 32$. It then splits this into two arrays, $T$ of size $b$-by-$b$, and another array of size $(n - b)$-by-$b$, for the compact $WY$ update of DLARFT and DLARFB. However, both arrays are given a leading dimension of $n$. When $m << n$, the small $T$ array is spread over a vast region of memory, with a detrimental impact on performance of DGEQRF. By contrast, the dense frontal factorization algorithm in SuiteSparseQR gives $T$ a leading dimension of $b$. As a result, its dense QR factorization is faster than DGEQRF when $m << n$, and achieves the same performance otherwise.

In contrast to DGEQRF, the dense frontal matrix factorization exploits the zero entries in the lower left corner of the frontal matrix. As an example, consider the frontal matrix in Figure 3. The blocksize parameter $b$ (default 32) defines the number of columns in each panel of the frontal matrix. For this small example, suppose the panel size is two instead. The first Householder vector is computed with DLARFG and applied to the current panel (the first two columns of the frontal matrix). Only the first four rows are operated on. Next, the Householder

reflection that zeros out the entries marked `h` in column 6 is found. Since the panel is now complete, it is applied in its entirety to the rest of the frontal matrix, using DLARFT and DLARFB. This block of Householder updates is applied only to the first 8 rows of the frontal matrix, thus exploiting most of the zeros in the lower left corner. In contrast, DGEQRF would operate on all 17 rows with each panel.

The frontal matrices are factorized in post-ordered fashion, so that the contribution blocks can be easily placed on a stack. To assemble and factorize a front, space for a rectangular frontal matrix is first allocated at the top of a workspace that will hold $R$ and the Householder vectors $H$ (assuming the latter is not discarded). Next, the children are popped from the stack (held at the bottom of this same block of memory) and assembled into the front. The frontal is factorized, and then its contribution block $C$ is copied out of the frontal matrix and pushed on the stack. Finally, the $R$ and $H$ components are compressed, in place, to hold just the entries `r` and `h` entries shown in Figure 3 (reclaiming the space used by $C$ block and the explicit zeros in the bottom left corner of the frontal matrix). The components of the Householder vectors $H$ need not be preserved if the matrix $Q$ is not needed.

SuiteSparseQR also handles rank deficient matrices, in contrast to all but one prior sparse multifrontal QR factorization method ([Pierce and Lewis 1997]). Their method uses a sparse incremental condition estimator [Bischof et al. 1990], and restricted column pivoting. The method is very accurate, but it requires a dynamic updating/downdating of the factor $R$ computed so far, if a column is found to be linearly dependent. The structure of the update/downdate is identical to a sparse update/downdate of the Cholesky factorization of $A^TA$ [Davis and Hager 1999; 2001; 2009; Chen et al. 2008]. The column that needs to be removed need not be the current column being eliminated; thus the need for a dynamic restructuring of the existing $R$. The nonzero pattern of the resulting matrix can arbitrarily differ from the Cholesky factorization of $(AP)^T(AP)$ with the original fill-reducing ordering $P$. Since it requires a mixture of Householder reflections and Givens rotations, their method is not well-suited to preserving the factor $Q$, in either matrix or Householder-vector form; they do not provide the option of keeping $Q$.

SuiteSparseQR, in contrast, uses a simpler method by [Heath 1982] for handling rank-deficiency, extended in the present work to the sparse multifrontal QR and allowing $Q$ to be kept as a set of Householder vectors. In Heath's method, if the diagonal of $R$ drops below a given threshold, the entire row is zeroed out via Givens rotations, and the row is deleted from $R$. The result is a "squeezed" factor $R$ that is no longer upper triangular.

[Foster 2009] reports that SuiteSparseQR correctly finds the rank of $A$ for about two-thirds of the rank-deficient matrices in the Univ. of Florida Sparse Matrix Collection. However, for many of those matrices, the numerical rank is ill-defined. If the numerical rank is very well-defined, then SuiteSparseQR frequently finds the correct rank. In particular, if the rank of $A$ is $k$, and $\sigma_k/\sigma_{k+1} > 10^3$ where $\sigma_k$ is the $k$th largest singular value of $A$, the correct rank is found 95% of the time.

As an example of how rank-deficient matrices are handled in SuiteSparseQR, consider Figure 3. The first Householder reflection (column 5 of the $A$ matrix) is applied to the second column (6). The second Householder reflection would annihilate the entries below the diagonal in the second column, replacing the diagonal

with the norm of the vector of entries on or below the diagonal. If this norm is below the threshold $\tau$ (the same $\tau$ used in the symbolic analysis), then the Householder reflection for this column is skipped. The entire column (entries on or below the diagonal) is treated as zero. The Householder reflection of the third column annihilates everything below the second entry in that column.

The result is shown on the right of Figure 3. The front holds one less row of $R$ than expected, resulting in Heath's "squeezed" $R$. The contribution block is still 4-by-4 in this case, but in general its size can increase. The number of columns in $C$ remains the same, namely, the number of non-pivotal columns in the front. If $C$ would have started out as upper trapezoidal (as in the case for the third front), the loss of one column would cause $C$ to grow in size by one row. There is no net growth in workspace, since $R$ is smaller by one row. These observations lead to the following theorem.

**Theorem 1:** *Heath's method of handling rank-deficient matrices in a sparse QR factorization requires the use of the column elimination tree, and causes no fill-in in $R$ if the nonzero pattern of $R$ is taken as the Cholesky factor of $A^T A$.*

**Proof:** The proof is by induction on the path from node $i$ to the root, where $|r_{ii}| < \tau$ for the given threshold $\tau$. This entry is treated as numerically zero, and all of row $i$ must be zeroed out in the factor $R$. Let $j$ be the column index of the next nonzero entry in row $i$, to the right of column $i$. To zero this entry, Heath's method performs a Givens rotation between row $i$ and row $j$ of $R$. Node $j$ is the parent of $i$ in the column elimination tree, and prior to the rotation, the nonzero pattern of $R_{i*}$ is a subset of $R_{j*}$ (excluding $i$ itself), assuming $R$ has the same pattern as the Cholesky factorization of $A^T A$ [George and Liu 1981]. Thus, this rotation causes no fill-in in $R_{j*}$, and causes the updated row $i$ to take on the same nonzero pattern of $R_{j*}$ (excluding the entries $r_{ii}$ and $r_{ij}$, which have been set to zero). Let $k$ be the column index of the next nonzero entry in row $i$; then $k$ is the parent of $j$.

If the updated row $i$ has been zeroed out from columns $i$ to $k-1$, it must take part in a Givens rotation with row $k$, taking on the same nonzero pattern as row $k$ (except for $r_{ik}$, which is set to zero) and causing no fill-in in $R_{k*}$ because the pattern of row $i$ is a subset of the pattern of row $k$. If $p$ is the least column index of the nonzero entries in the updated row $i$, node $p$ is the parent of $k$ and the pattern of row $i$ is a subset of row $p$. The process stops at the root of the tree, at which point row $i$ is completely zero. □

SuiteSparseQR uses Householder reflections applied to frontal matrices instead of Givens rotations applied to rows, but structurally the effect on a rank-deficient $R$ is the same as in the above theorem.

Exploiting the BTF is not compatible with Heath's rank-detection method. For example, suppose $A$ is a 3-by-3 upper triangular matrix. In the BTF form, each scalar diagonal would be "factorized" into $QR$ (with $Q = I$). No numerical work is done. Now suppose $a_{22}$ is numerically zero, or very tiny. In the BTF,

$$R = \begin{bmatrix} x & x & x \\ & 0 & x \\ & & x \end{bmatrix},$$

the $a_{23}$ entry is not in the QR factorization, but in an off-diagonal block (in $B$ for the factorization $QR + B$). The entry cannot be taken into account during

factorization without operating on entries outside the diagonal blocks. These operations and their potential fill-in were not predicted during the symbolic analysis, and consequently Heath's form of the "squeezed" $R$, below, cannot be found:

$$R = \begin{bmatrix} x & x & x \\ & & x \\ & & \end{bmatrix}.$$

Heath's squeezed form of $R$ can be used to find a basic solution to a rank-deficient problem. For example, the $R$ matrix shown above can be implicitly or explicitly permuted into the following, placing the "live" columns first:

$$R = \begin{bmatrix} x & x & x \\ & x & \\ & & \end{bmatrix}.$$

Now the matrix is upper trapezoidal, and a basic solution can be found by setting to zero the components of the solution corresponding to "dead" columns and doing a regular backsolve on the remaining 2-by-2 upper triangular system.

## 4. USING THE QR FACTORIZATION

The QR factorization computed by SuiteSparseQR can be used to solve least squares problems and under-determined systems (both basic and minimum 2-norm solutions). Each of these three solve methods are described below. MATLAB notation is used, but these features are available both in MATLAB and in the C/C++ API (except where otherwise noted). In each method, `P` is the fill-reducing permutation, represented as a permutation vector rather than as a matrix (see Section 2.2).

(1) If `m` $\geq$ `n`, the system is a least-squares problem ($\min_x ||b - Ax||_2$), and `A*P=Q*R` is factorized. If the right-hand side `b` is provided prior to factorization, `c=Q'*b` is computed during factorization, and `Q` is not constructed. Rather, each Householder reflection is discarded once it is applied to `A` and `b`. The least-squares solution is `x=P*(R\c)` [Golub 1965]. This is the default for least-square problems in MATLAB (`x=A\b`). If `b` is not provided during the factorization of `A`, or if the factors are to be reused with a new `b`, then the Householder reflections are not applied to `b`. Two alternatives exist for computing `x`:
   —The factor `Q` can be discarded. The least-squares solution is computed via the seminormal equations and a single step of iterative refinement [Stewart 1998]. In MATLAB, this process requires a few lines of additional MATLAB code that are not part of SuiteSparseQR (`x=R\R'\(A'*b)); r=b-A*x; e=R\R'\(A'*r)); x=x+e`). For the C/C++ interface, these operations can be performed by a handful of calls to SuiteSparseQR or CHOLMOD functions.
   —The factor `Q` can be kept in one of two formats: as a set of Householder reflections, or as a matrix. The first option is faster and takes far less memory, but the latter is simpler for the MATLAB end user. The MATLAB interface to SuiteSparseQR provides both options, but the built-in SuiteSparseQR in MATLAB R2009a only provides the matrix representation of `Q`,

(2) If `m` $<$ `n`, a minimum 2-norm solution to the under-determined system starts with the factorization `A'*P=Q*R`. The Householder reflections cannot be applied and discarded during factorization, so SuiteSparseQR saves the set of

Householder vectors representing `Q`. After the forward solve (`c=R'\(P'*b)`), a separate SuiteSparseQR function then applies the Householder-vector representation of `Q` to `c`, giving `x=Q*c`.

(3) If `m` < `n`, a basic solution to the under-determined system starts with the factorization `A*P=Q*R`. The Householder reflections are applied to `b` during factorization via `c=Q'*b`, and `Q` is discarded. The basic solution is `x=P*[R1\c;0]` where `R1` is the leading square submatrix of `R`. This is the default for under-determined problems, for historical reasons (MATLAB does this for `x=A\b`).

If `A` is rank-deficient, then the resulting matrix `R` is in "squeezed" form. This implicitly defines a column permutation that places `R` in upper-trapezoidal form, and ensures that the implicit leading square submatrix of `R` is upper triangular with dimension $r$-by-$r$ if $r$ is the estimated rank of `A`. All diagonal entries of this leading square submatrix have magnitude greater than the threshold parameter $\tau$. For the trailing columns to the right of this triangular part, all corresponding entries of the solution vector are set to zero.

## 5. PARALLELISM

At least two opportunities for parallelism exist within a multifrontal sparse QR factorization. The first opportunity arises in the frontal matrix tree. In the example given in Figure 1, the first three frontal matrices can be factorized in parallel (one front for computing the first two rows of $R$, and the next two which are used to compute rows 3 and 4 of $R$). Using this level of parallelism requires explicit thread-based software in SuiteSparseQR.

The second opportunity arises within each frontal matrix. The factorization of a frontal matrix relies on LAPACK, which in turn uses the Level-3 BLAS [Dongarra et al. 1990]. Most of the vendor-supplied BLAS, such as the Intel®Math Kernel Library (MKL) BLAS, the AMD ACML BLAS, the Sun Performance Library BLAS, as well as the Goto BLAS [Goto and van de Geijn 2008] can exploit a shared-memory multicore architecture with no additional programming effort on the part of developers of packages that use the BLAS.

SuiteSparseQR exploits tree-based parallelism by using Intel's new Threading Building Blocks (TBB) software for writing parallel applications in C++ on shared-memory multicore architectures [Reinders 2007]. Often, a TBB-based application need only determine the tasks; TBB itself takes care of the task scheduling and synchronization. This is the case for SuiteSparseQR. Although TBB does provide application interfaces for mutual exclusion, atomic operations, queues, and the like, SuiteSparseQR does not require these features.

Note that the Intel MKL uses OpenMP [Chapman et al. 2007] rather than Intel's TBB for exploiting parallelism in its BLAS. This design choice has performance implications in the current version of TBB which will be illustrated in Section 6.

Let $n_f$ be the number of frontal matrices. The goal of the analysis phase for the parallel factorization is to assign the $n_f$ frontal matrices to TBB tasks, where there are normally fewer than $n_f$ tasks. To simplify synchronization between tasks, the relationship between the tasks is kept as a tree. It would be possible to place each frontal matrix in its own task, but this could lead to synchronization overhead in TBB, particularly for the very small frontal matrices at the bottom of the tree.

Similar strategies have been used for sparse Cholesky factorization [Geist and Ng 1989]. The front-to-task assignment takes $O(n_f)$ time.

For each frontal matrix $f$, the work in the subtree rooted at $f$ is found (including $f$ itself). A *big* node is defined as a node for which the work in its subtree is greater than $\max(\omega/\alpha, \beta)$ where $\omega$ is the total flop count for the entire QR factorization, and $\alpha$ and $\beta$ are user-definable parameters that control the task tree granularity. Typically $\alpha$ should be at least twice the number of cores, and $\beta = 10^6$. All other nodes are *small*. To ensure the frontal matrix tree is truly a tree and not a forest, a placeholder node $n_f + 1$ is added which is the parent of any root nodes. This placeholder node is also marked as *big*, regardless of parameters $\alpha$ and $\beta$.

The first pass assigns all small nodes to tasks. Suppose front $f$ is a small node but its parent $p$ is a big node. If $f$ is the least numbered such child of $p$, all fronts in the subtree rooted at $f$ are placed in a new task. Additional children of $p$ which are also small nodes are added to this task, until the task has at least $\max(\omega/\alpha, \beta)$ work. After that, a new task is created for the subtrees of the children of $p$.

The second pass assigns all big nodes to tasks, in order 1 to $n_f + 1$ so that all children of a big node $f$ are assigned to their tasks before considering node $f$ itself. If all of the children of $f$ are assigned to the same task, then $f$ is also assigned to the same task. If $f$ has no children, or if it has children assigned to different tasks, then a new task is created to which $f$ is assigned.

Finally, the stack size for each task is found. Two tasks can share a stack if one is the ancestor of the other, so there are only as many stacks as there are leaves in the task tree. Allocating one stack per leaf of the tree ensures that there are no memory conflicts between different TBB tasks.

In the numeric factorization phase, all workspace used by the tasks (including the set of contribution block stacks) is allocated before TBB schedules the tasks. No dynamic memory allocation is needed during the TBB-parallel phase of the factorization. Each task factorizes all frontal matrices in a subtree of the frontal matrix tree, and the results are left on the stack used by that task. No synchronization is needed except that a task can start when all its children are finished; this is handled by specifying the set of tasks and their dependencies to TBB, and TBB handles all synchronization and scheduling. Additional parallelism within each task is exploited via a multicore implementation of the Level-3 BLAS.

## 6. EXPERIMENTAL RESULTS

In this section, the performance of SuiteSparseQR is compared with other solvers. When different factorization methods are compared, they are always compared using the same fill-reducing ordering. To test the methods, all rectangular matrices in the University of Florida Sparse Matrix Collection [Davis and Hu 2011] are used. As of Sept. 2008, the UF Collection contains 30 least-squares problems and 353 under-determined systems (most of which are linear programming problems).

Most results are from a Rackable Systems shared-memory computer with eight dual-core AMD Opteron 875 processors. Additional results are presented on a Dell Latitude D620 dual-core laptop with an Intel T2500 Core Duo processor. Table I lists the statistics and performance of these two computers. All timings presented are in MATLAB with `tic` and `toc` (wall-clock time), using the version of the

| | Rackable Systems | | Dell D620 laptop | |
|---|---|---|---|---|
| processor | AMD Opteron 875 | | Intel T2500 Core$^{TM}$Duo | |
| clock cycle | 2.21 GHz | | 2.00 GHz | |
| memory | 64 GB | | 2 GB | |
| operating system | Red Hat Linux | | Ubuntu Linux 8.04 | |
| MATLAB | R2008a | | R2008a | |
| BLAS | Intel MKL 9.1 | | Intel MKL 9.1 | |
| Intel TBB | Version 2.1 | | Version 2.0 | |
| performance | 1 core | 16 cores | 1 core | 2 cores |
| theoretical peak | 4.42 GFlops | 70.72 GFlops | 2.00 GFlops | 4.00 GFlops |
| `C=A*B` peak | 3.75 GFlops | 53.4 GFlops | 1.63 GFlops | 3.24 Gflops |
| `C=A*B` speedup | | 14.24 | | 1.99 |
| `C=A*B` half | $n = 32$ | $n = 2500$ | $n = 20$ | $n = 28$ |
| `X=qr(A)` peak | 2.72 GFlops | 12.1 GFlops | 1.49 GFlops | 2.80 Gflops |
| `X=qr(A)` speedup | | 4.65 | | 1.88 |
| `X=qr(A)` half | $n = 70$ | $n = 600$ | $n = 40$ | $n = 129$ |

Table I. Computers used for the experimental results

BLAS bundled with MATLAB. The `C=A*B` and `X=qr(A)` statements are light-weight interfaces to DGEMM in the BLAS and DGEQRF in LAPACK, respectively; these results are for $n$-by-$n$ dense matrices. The *half* metric is the value of $n$ needed to obtain half the peak performance; this is a critical metric for a multifrontal sparse QR, since it computes the QR factorization of many small dense frontal matrices.

### 6.1 The methods

Let `A` be an `m`-by-`n` matrix. The following methods are compared in this section:

—**SuiteSparseQR**: This method can be used in many different ways, depending on the matrix and what solution is desired:

(1) If `m` $\geq$ `n` then `A*P=Q*R` is factorized. SuiteSparseQR can apply the Householder vectors to the right-hand side `b` during factorization and discard them as they are computed. This option is used in these comparisons, and it is how SuiteSparseQR is used by `x=A\b` in MATLAB R2009a to solve least-squares problems. If `b` is not available during factorization then `Q` must be kept in Householder form, but this option is not considered for these experiments. SuiteSparseQR takes more time and memory in this latter case.

(2) If `m` $<$ `n` the system is under-determined. If a basic solution is requested, SuiteSparseQR factorizes `A*P=Q*R`. By default, `Q` is not kept. This is how SuiteSparseQR is used by `x=A\b` in MATLAB R2009a.

(3) If `m` $<$ `n` and a minimum 2-norm solution is requested, SuiteSparseQR factorizes `A'P=Q*R` and keeps `Q` in Householder form. MATLAB R2009a does not provide access to option in `x=A\b`.

For the experiments, singleton removal and rank-detection are enabled (the default for SuiteSparseQR). Section 4 discusses the SuiteSparseQR solve phase.

—**MA49** [Amestoy et al. 1996]: MA49 provides a non-default option to use the Level-3 BLAS, but during these experiments it was found that using the BLAS was always faster (except for small sparse matrices whose results not included here), so only results with the BLAS are presented here. Using the BLAS in MA49

can increase the space used, however. Parallelism is not enabled in the current version of MA49 provided in the HSL software library, so its parallel performance is not considered in this paper. The shared-memory version of MA49 is based on OpenMP with explicit synchronization and its own task scheduling. MA49 also performs the solve phase in parallel. MA49 was selected for this comparison since it is a good example of a prior multifrontal sparse QR method, and was readily available for comparison. The following methods are considered:

(1) If `m` $\geq$ `n` then `A*P=Q*R` is factorized without BTF, and `Q` is kept in Householder form. The least-squares solution is `x=P*(R\(Q'*b))`. It is referred to below as MA49:default.
(2) Option (2) is the same as option (1), but with BTF. It is referred to as MA49:BTF.
(3) If `m` $\geq$ `n`, `A*P=Q*R` is factorized without BTF, and `Q` is discarded. The least-squares solution uses the seminormal equations, `x=P*(R'\(R\(P'*A'*b)))`, and one step of iterative refinement. It is referred to as MA49:seminormal. This is the only case where MA49 allows the Householder vectors to be discarded during factorization.
(4) If `m` $<$ `n`, the factorization is `A'*P=Q*R` without BTF, and `Q` is kept in Householder form. The minimum 2-norm solution is `x=Q*(R'\(P'*b))`. It is referred to as MA49:default.

—**MATLAB backslash**, or `x=A\b` in MATLAB (R2008b or earlier). It uses the implementation from [Gilbert et al. 1992] of the Givens-based method of [George and Heath 1980; Heath 1982], including Heath's strategy for rank-deficient matrices. By default, `P` is found via COLMMD, but this is replaced in these experiments with AMD or COLAMD. The MATLAB R2008a sparse QR was selected for this comparison since it is the method that was replaced by SuiteSparseQR as the built-in sparse QR for MATLAB R2009a. The following methods are considered:

(1) For `x=A\b` when `m` $\geq$ `n`, `A*P=Q*R` is factorized. `Q` is discarded, and `c=Q'*b` is computed during factorization. The least-squares solution is `x=P*(R\c)`.
(2) For `x=A\b` when `m` $<$ `n`, `A*P=Q*R` is factorized, `Q` is discarded, and `c=Q'*b` is computed during factorization. The basic solution is `x=P*[R1\c;0]`.
(3) MATLAB can compute the minimum 2-norm solution if `Q` is kept, with the statements `p=colamd(A') ; [Q,R]=qr(A(p,:)') ; x=Q*(R'\b(p))`. However, `Q` is returned in its matrix form which is infeasible for large problems.

### 6.2 Comparing solvers for least-squares problems

In these comparisons the AMD ordering on $A^TA$ is used, which is the default for MA49 and a more effective ordering than COLMMD used by backslash in MATLAB. SuiteSparseQR, the MATLAB backslash, and all three of MA49's methods for solving least-squares problems are compared. Of the 30 least-squares problems, the 11 largest (with `m` $\geq 10,000$) are shown in Table II (excluding one matrix too large for the Rackable Systems computer). Of these, four are rank-deficient.

Table III lists the total time and memory usage required by each of the 5 methods to solve the 11 problems, for the entire solution (symbolic analysis, numeric factorization, and solve phases).

| # | name | $m/10^3$ | $n/10^3$ | $s$ | $b$ | $\|A\|/10^3$ | description |
|---|---|---|---|---|---|---|---|
| 1 | YCheng/psse1 | 14.3 | 11.0 | 482 | 1722 | 57.4 | power system simulation |
| 2 | NYPA/Maragal_6 | 21.3 | 10.2 | 10 | 89 | 537.7 | from NY Power Authority |
| 3 | YCheng/psse0 | 26.7 | 11.0 | - | 1 | 102.4 | power system simulation |
| 4 | Kemelmacher | 28.5 | 9.7 | - | 1 | 100.9 | 3D computer vision |
| 5 | YCheng/psse2 | 28.6 | 11.0 | - | 1 | 115.3 | power system simulation |
| 6 | Sumner/graphics | 29.5 | 11.8 | - | 1 | 118.0 | computer graphics problem |
| 7 | NYPA/Maragal_7 | 46.8 | 26.6 | 46 | 1294 | 1200.5 | from NY Power Authority |
| 8 | Toledo/deltaX | 68.6 | 22.0 | - | 1 | 247.4 | computer graphics problem |
| 9 | Pereyra/landmark | 72.0 | 2.7 | 32 | 5 | 1146.8 | surveying problem |
| 10 | Springer/ESOC | 327.1 | 37.8 | 553 | 74 | 6019.9 | satellite orbits |
| 11 | Rucci/Rucci1 | 1977.9 | 109.9 | - | 1 | 7791.2 | an ill-conditioned problem |

Table II. Large least-squares problems, where $s$ is the number of singletons and $b$ is the number of blocks in the BTF ($s$ can exceed $b$ for rank-deficient matrices).

Since the UF Collection has so few full-rank least-squares problems, the four rank-deficient matrices were also solved via Tikhonov regularization (appending $\gamma I$ to $A$, where $\gamma = 10^{-12} \max_j ||A_{*j}||_2$) to get more results to compare with MA49. These are shown in the second part of Table III (MATLAB backslash is skipped for these regularized problems). Note that appending $\gamma I$ ensures that the matrix has no singletons in them at all (unless a matrix has a completely-zero column, which none of these matrices have). In each table in this paper, run times are in seconds and memory usage is in gigabytes. A dash is shown in the MA49 results for rank-deficient matrices; this is not a failure on the part MA49. Memory usage statistics are not available from the MATLAB backslash.

Except for two small matrices for which it is tied with MA49, and two regularized problems, SuiteSparseQR is the fastest method for these matrices and uses the least amount of memory. When MA49 uses the seminormal equations it can discard $Q$, and in this case it requires about the same memory as SuiteSparseQR (which also discards $Q$ for these problems).

### 6.3 Comparing solvers for minimum 2-norm solutions

The `qr` function in MATLAB (R2008b and earlier) is not well-suited to computing the minimum 2-norm solution to a sparse under-determined system since it returns $Q$ in matrix form rather than as a set of Householder vectors or Givens rotations (`qr` uses Givens rotations). Consider the results shown in Table IV for the lp_nug08 linear programming matrix obtained from M. Resende. It is 912-by-1632 with rank 742 and was selected for this example because it has no singletons. For these results, AMD is used for both SuiteSparseQR and the MATLAB `qr`. The matrix $H$ is the set of Householder vectors as computed by SuiteSparseQR. It is not uncommon for $H$ to have fewer nonzeros than $R$, while $Q$ can be almost a full matrix. Both methods find solutions with equally low residuals (about $10^{-14}$), in spite of the rank-deficiency of the matrix. In general, finding the minimum 2-norm solution takes more time and memory than finding the basic solution, for both MATLAB R2008b and SuiteSparseQR. Also, unlike nearly all other matrices, the "nug" matrices in the UF Collection experience extreme fill-in in $R$.

Attempting to use the MATLAB `qr` to find the minimum 2-norm solution is

| # | SuiteSparseQR | | MA49:default | | MA49:BTF | | MA49:semi. | | Backslash |
|---|---|---|---|---|---|---|---|---|---|
| | time | mem | time | mem | time | mem | time | mem | time |
| 1 | **0.1** | **0.00** | **0.1** | 0.01 | 0.2 | 0.01 | **0.1** | **0.00** | 0.2 |
| 2 | **1222.0** | **1.62** | - | - | - | - | - | - | 7298.5 |
| 3 | **0.1** | **0.01** | 0.2 | **0.01** | 0.2 | **0.01** | 0.2 | **0.01** | 0.4 |
| 4 | **0.8** | **0.02** | **0.8** | 0.10 | **0.8** | 0.10 | **0.8** | **0.02** | 9.4 |
| 5 | **0.1** | **0.01** | 0.2 | **0.01** | 0.2 | **0.01** | 0.2 | **0.01** | 0.6 |
| 6 | **0.1** | **0.01** | 0.4 | **0.01** | 0.4 | 0.02 | 0.4 | **0.01** | 0.8 |
| 7 | **6654.6** | **5.38** | - | - | - | - | - | - | 17,318.2 |
| 8 | **187.8** | **0.45** | 394.6 | 3.41 | 401.1 | 3.42 | 411.1 | 0.60 | 5708.2 |
| 9 | **1.2** | **0.04** | - | - | - | - | - | - | 23.9 |
| 10 | **874.4** | **2.86** | - | - | - | - | - | - | 67,928.6 |
| 11 | **5568.5** | **5.86** | 12,615.4 | 41.8 | 14,741.7 | 41.83 | 15,956.9 | 7.38 | 946,475.2 |
| with regularization for rank-deficient matrices: | | | | | | | | | |
| 2' | **1246.0** | **2.39** | 1393.9 | 5.13 | 1389.4 | 5.15 | 1395.5 | 2.51 | |
| 7' | 6764.9 | 8.67 | 3725.0 | 25.60 | 3733.9 | 25.61 | **3653.8** | **3.58** | |
| 9' | 1.3 | 0.04 | 1.1 | 0.08 | - | - | **1.0** | **0.03** | |
| 10' | **1015.0** | 3.42 | 2228.7 | 23.97 | 2422.7 | 24.02 | 2340.7 | **2.57** | |

Table III. Results for large least-squares problems; best results in bold. Time is seconds and memory usage in GB, for the analysis, factorization, and solve. A dash is shown for MA49 for rank-deficient matrices.

| | Basic solution | Min. 2-norm solution |
|---|---|---|
| $\|R\|$ | 452,924 | 362,496 |
| $\|H\|$ | 116,234 | 210,084 |
| $\|Q\|$ | 486,877 | 1,768,457 |
| SuiteSparseQR | 0.34 sec. ($Q$ discarded) | 0.31 sec. (using $H$) |
| MATLAB | 3.42 sec. ($Q$ discarded) | 32.3 sec. (using $Q$) |

Table IV. Basic and minimum 2-norm solutions for the Qaplib/lp_nug08 matrix.

infeasible for large problems, in both time and memory. In contrast, both MA49 and SuiteSparseQR can efficiently compute the minimum 2-norm solution by factorizing $A^T$ and keeping $Q$ in Householder form (the $H$ matrix) to compute the solution `x=Q*(R'\(P'*b))`. There are 150 rectangular matrices in the UF Collection with $m < n$ and $n \geq 10,000$; of those, 61 rank-deficient matrices and 11 matrices too large for SuiteSparseQR and MA49 to handle are excluded. The resulting test set contains 78 matrices. Table V lists the results for those matrices with $n \geq 70,000$.

Figure 4 presents the results for all 78 matrices in two performance profiles (one for the run-time and the other for the memory usage). For any given method, a data point $(x, y)$ on the time profile means that the method is no worse than $x$ times slower than the fastest time of any method for those $y$ problems. For example, the $y$-intercept gives the number of problems for which a method is the fastest (or is tied for the fastest). The $(2, y)$ point gives the number of problems ($y$) for which a method is either fastest or no worse than twice as slow as the fastest method.

### 6.4 Parallel results

The leaves of the TBB task tree can often contain small frontal matrices with little scope for parallelism in the BLAS. In contrast, the root of the tree provides no tree-

| name | $m/10^3$ | $n/10^3$ | $s/10^3$ | $\|A\|/10^3$ | SuiteSparseQR time | SuiteSparseQR mem | MA49:default time | MA49:default mem |
|---|---|---|---|---|---|---|---|---|
| Meszaros/pltexpa | 26.9 | 70.4 | - | 143.1 | **2.4** | **0.09** | 3.1 | 0.19 |
| Qaplib/lp_nug20 | 15.2 | 72.6 | - | 304.8 | **2032.0** | **6.17** | 4665.3 | 19.57 |
| Meszaros/rlfdual | 8.1 | 75.0 | - | 282.0 | **26.2** | **0.37** | 37.0 | 0.79 |
| Meszaros/nemsemm1 | 3.9 | 75.4 | 0.1 | 1054.0 | **1.4** | 0.08 | 3.6 | **0.07** |
| Meszaros/fxm3_16 | 41.3 | 85.6 | 1.1 | 392.3 | **0.8** | **0.04** | 1.7 | 0.06 |
| Mittelmann/fome13 | 48.6 | 97.8 | - | 285.1 | **37.7** | **0.52** | 75.8 | 3.06 |
| Meszaros/stat96v1 | 6.0 | 197.5 | - | 588.8 | 0.7 | **0.06** | **0.6** | 0.07 |
| Meszaros/dbic1 | 43.2 | 226.3 | - | 1081.8 | **11.1** | **0.28** | 31.1 | 0.95 |
| Mittelmann/sgpf5y6 | 246.1 | 312.5 | - | 832.0 | **7.6** | **0.48** | 95.1 | 1.46 |
| Mittelmann/watson_1 | 201.2 | 387.0 | 15.3 | 1055.1 | **4.1** | **0.23** | 140.9 | 0.48 |
| Mittelmann/watson_2 | 352.0 | 677.2 | 26.8 | 1846.4 | **7.6** | **0.43** | 431.9 | 0.77 |
| Meszaros/stat96v2 | 29.1 | 957.4 | - | 2852.2 | 3.6 | **0.29** | **3.5** | 0.32 |
| Meszaros/stat96v3 | 33.8 | 1113.8 | - | 3317.7 | 4.2 | **0.34** | **4.1** | 0.38 |

Table V. Minimum 2-norm solutions; best results in bold. Time is in seconds and memory usage is in GB. Four matrices have singletons ($s$ is the number singletons).

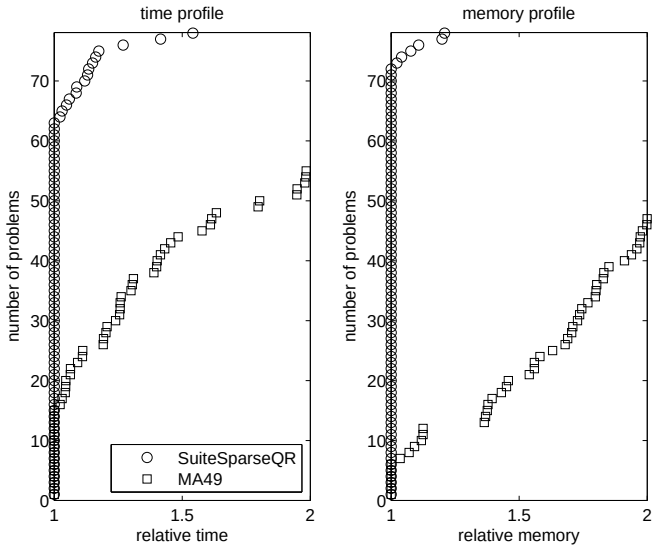


Fig. 4. Performance profiles for minimum 2-norm solutions with the AMD ordering

based parallelism, but can be a rich source of BLAS-based parallelism. These two kinds of parallelism should be complementary, but in the current implementation of TBB they compete with one another. A future implementation of TBB hopes to address this issue [Robison 2008]. The threads created by TBB and OpenMP are separate, and each TBB thread will create its own set of OpenMP threads (although for small matrices, the OpenMP-based BLAS uses just one thread). The maximum total number of threads used is thus the product of these two sets of threads. In a future implementation, TBB should use all the threads at the leaves and lower part of the TBB task tree, and the BLAS should use all the threads at

| Matrix | TBB:1, BLAS:1 | | TBB:1, BLAS:2 | | TBB:2, BLAS:1 | | TBB:2, BLAS:2 | |
|---|---|---|---|---|---|---|---|---|
| | time | GFlops | speedup | GFlops | speedup | GFlops | speedup | GFlops |
| deltaX | 156.6 | 1.31 | 1.76 | 2.30 | 1.74 | 2.29 | 1.41 | 1.85 |
| landmark | 1.5 | 0.81 | 1.22 | 0.99 | 1.48 | 1.19 | 1.26 | 1.02 |
| ESOC | 580.1 | 1.27 | 1.75 | 2.22 | 1.72 | 2.18 | 1.41 | 1.78 |

Table VI. SuiteSparseQR parallel performance with METIS on a dual-core Dell D620 laptop

| Matrix | TBB:1 BLAS:1 | | | TBB:1 BLAS:$b$ | | TBB:$k$ BLAS:1 | | best TBB:$k$, BLAS:$b$ | | | | TBB :16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | time | GF | mem | sp. | $b$ | sp. | $k$ | sp. | $k$ | $b$ | GF | mem |
| deltaX | 97.5 | 2.11 | 0.34 | 2.99 | 10 | 4.13 | 16 | 5.70 | 15 | 6 | 12.01 | 1.03 |
| landmark | 1.6 | 0.76 | 0.04 | 1.07 | 5 | 1.93 | 14 | 1.93 | 14 | 1 | 1.47 | 0.06 |
| ESOC | 381.2 | 1.93 | 0.62 | 2.59 | 8 | 4.85 | 15 | 5.47 | 14 | 3 | 10.56 | 3.57 |
| Rucci1 | 2525.8 | 2.45 | 3.20 | 3.70 | 12 | 2.44 | 16 | 5.76 | 14 | 12 | 14.10 | 7.35 |

Table VII. SuiteSparseQR parallel performance on a 16-core AMD Opteron system

the root node of the TBB task tree. In the middle a mixture should be used.

The four least-squares systems with largest $n$ from Table II are selected for this experiment. METIS is used since it gives the best orderings for parallel factorization. The results on two multicore computers are shown in Tables VI and VII.

Table VI reports the results for three of these matrices on the dual-core laptop (one matrix is too large). For the largest problems, SuiteSparseQR obtains a substantial fraction of the dense `qr` performance on this laptop (refer to Table I). The total single-threaded time (TBB:1, BLAS:1) is in seconds (including ordering, analysis, numeric factorization, and solve time). TBB:1 means that the entire matrix is factorized as a single task without the use of any calls to the TBB library. Columns to the right of the single-threaded column give the speedup relative to the total single-threaded time (note that only the numeric factorization is done in parallel). Each column also reports the GFlops obtained (total time / flops, but excluding any useless flops performed). The (TBB:2, BLAS:2) method creates up to four threads, resulting in a drop in performance compared with the 2-thread methods, (TBB:1, BLAS:2) and (TBB:2, BLAS:1). The 2-thread methods have comparable performance and show good speedup for large problems.

The same four matrices were tested with one to 16 TBB threads and with one to 16 BLAS threads (all 256 combinations) on the 16-core AMD Opteron system. Table VII lists the results with a single thread, the best speedup found with BLAS-only parallelism and with TBB-only parallelism, respectively, and the best speedup and GFlops found with any mixture of multithreading methods. The last column is the memory usage for 16 TBB threads. In Table VII, $k$ is the number of threads used in TBB, $b$ is the number of threads used in the BLAS, speedup is abbreviated as "sp.," GFlops is abbreviated GF, and memory usage is listed in gigabytes. On 16 cores, the peak of 14.1 GFlops obtained by SuiteSparseQR actually exceeds the peak of 12.1 GFlops obtained by the dense QR in MATLAB (refer to Table I), which is LAPACK plus the multithreaded BLAS. A true parallel dense QR factorization ([Blackford et al. 1997; van de Geijn 1997]) would likely obtain a higher performance.

| Method | ordering | cores | time |
|---|---|---|---|
| `x=A\b` | COLMMD | 1 | failed |
| `x=A\b` | AMD | 1 | 11 days |
| MA49 | AMD | 1 | 3.5 hours |
| SuiteSparseQR | AMD | 1 | 1.5 hours |
| SuiteSparseQR | METIS | 1 | 45 minutes |
| SuiteSparseQR | METIS | 16 | 7.3 minutes |

Table VIII. Performance gains for Rucci1, the largest least squares problem in Table II.

Memory usage increases when using multiple TBB tasks because each thread requires its own stack for frontal matrices and their contribution blocks. When using 16 cores, memory usage typically increases by a factor of 1.5 to 3, with a peak increase of 5.7x for the ESOC matrix.

These results show that the tree-based parallelism exploited with TBB is typically able to utilize all the cores more efficiently than when just exploiting BLAS-based parallelism within each frontal matrix. The BLAS-only speedup is lower, and drops slightly below the fastest speedup if too many cores are used. The best results are typically obtained by exploiting nearly all of the available cores for TBB, with each TBB thread allowed to use just a few threads in the BLAS. Ideally, when a future version of Intel's TBB is able to work better with OpenMP [Robison 2008], these settings would be chosen automatically.

### 6.5 Overall performance

Table VIII summarizes the overall performance gain when using SuiteSparseQR for the largest least squares problem (Rucci1). The algorithmic speedup over the prior sparse `qr` in MATLAB is 375x. Parallelism provides another 5.75x speedup for a total speedup of 2,155x. Note that Rucci1 has no singletons and its BTF form consists of a single block, so the singletons exploited by SuiteSparseQR and the BTF exploited by MA49 have no effect on the two methods, respectively.

## 7. THE SUITESPARSEQR SOFTWARE PACKAGE

Details on how to use the software package are provided in documents in the package itself. SuiteSparseQR requires four prior Collected Algorithms of the ACM: CHOLMOD [Chen et al. 2008; Davis and Hager 2009], AMD [Amestoy et al. 1996; 2004], COLAMD [Davis et al. 2004a; 2004b], and the BLAS [Dongarra et al. 1990]. LAPACK is also required [Anderson et al. 1999]. Using Intel's Threading Building Blocks is optional [Reinders 2007], but without it, only parallelism within the BLAS can be exploited. SuiteSparseQR can optionally use METIS 4.0.1 [Karypis and Kumar 1998], CCOLAMD, and CAMD [Chen et al. 2008] for its fill-reducing ordering options. In addition to appearing as a Collected Algorithm of the ACM, SuiteSparseQR is available at http://www.cise.ufl.edu/research/sparse.

## 8. SUMMARY

SuiteSparseQR is an efficient multithreaded multifrontal sparse QR factorization method whose peak performance matches and sometimes exceeds that of the dense QR in LAPACK on both single and multiple cores. It is the first multifrontal

implementation of Heath's rank-revealing sparse QR method [Heath 1982]. Unlike MA49, its symbolic analysis phase finds an exact bound on the memory space required for $Q$ and $R$. No prior multifrontal sparse QR method exploits the optimal column-count algorithm of [Gilbert et al. 2001]. Its MATLAB interface provides substantial improvements for the MATLAB user, such as the representation of $Q$ as a set of sparse Householder vectors which allow it to efficiently solve minimum 2-norm problems.

## 9. ACKNOWLEDGEMENTS

I would like to thank Pat Quillen for his help with TBB, C++ templates, and the incorporation of SuiteSparseQR into MATLAB R2009a. I would like to thank Chiara Puglisi and the other referees for their helpful and detailed comments.

## REFERENCES

Amestoy, P. R., Davis, T. A., and Duff, I. S. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl. 17,* 4, 886–905.

Amestoy, P. R., Davis, T. A., and Duff, I. S. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 381–388.

Amestoy, P. R. and Duff, I. S. 1989. Vectorization of a multiprocessor multifrontal code. *Intl. J. Supercomp. Appl. 3,* 3, 41–59.

Amestoy, P. R., Duff, I. S., and Puglisi, C. 1996. Multifrontal QR factorization in a multiprocessor environment. *Numer. Linear Algebra Appl. 3,* 4, 275–300.

Amestoy, P. R. and Puglisi, C. 2002. An unsymmetrized multifrontal LU factorization. *SIAM J. Matrix Anal. Appl. 24*, 553–569.

Anderson, E., Bai, Z., Bischof, C. H., Blackford, S., Demmel, J. W., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. C. 1999. *LAPACK Users' Guide*, 3rd ed. SIAM, Philadelphia, PA.

Ashcraft, C. C. and Grimes, R. 1989. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Softw. 15,* 4, 291–309.

Bischof, C. H., Lewis, J. G., and Pierce, D. J. 1990. Incremental condition estimation for sparse matrices. *SIAM J. Matrix Anal. Appl. 11,* 4, 644–659.

Bischof, C. H. and Van Loan, C. F. 1987. The WY representation for products of Householder vectors. *SIAM J. Sci. Statist. Comput. 8,* 1, s2–s13.

Björck, A. 1996. *Numerical methods for least squares problems.* SIAM, Philadelphia, PA.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. 1997. *ScaLAPACK User's Guide.* SIAM, Philadelphia, PA.

Chapman, B., Jost, G., and van der Pas, R. 2007. *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, Cambridge, MA.

Chen, Y., Davis, T. A., Hager, W. W., and Rajamanickam, S. 2008. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Trans. Math. Softw. 35,* 3, 1–14.

Coleman, T. F., Edenbrandt, A., and Gilbert, J. R. 1986. Predicting fill for sparse orthogonal factorization. *J. ACM 33*, 517–532.

Davis, T. A. 2004. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30,* 2, 165–195.

Davis, T. A. 2006. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA.

Davis, T. A. and Duff, I. S. 1997. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Anal. Appl. 18,* 1, 140–158.

Davis, T. A., Gilbert, J. R., Larimore, S. I., and Ng, E. G. 2004a. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 377–380.

Davis, T. A., Gilbert, J. R., Larimore, S. I., and Ng, E. G. 2004b. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 353–376.

Davis, T. A. and Hager, W. W. 1999. Modifying a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl. 20,* 3, 606–627.

Davis, T. A. and Hager, W. W. 2001. Multiple-rank modifications of a sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl. 22*, 997–1013.

Davis, T. A. and Hager, W. W. 2009. Dynamic supernodes in sparse cholesky update/downdate and triangular solves. *ACM Trans. Math. Softw. 35,* 4, 1–23.

Davis, T. A. and Hu, Y. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw. 28,* 1. To appear; see also http://www.cise.ufl.edu/sparse/matrices.

Dongarra, J. J., Du Croz, J. J., Duff, I. S., and Hammarling, S. 1990. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw. 16*, 1–17.

Duff, I. S. 1977. On permutations to block triangular form. *J. Inst. Math. Appl. 19*, 339–342.

Duff, I. S. 1981. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Softw. 7,* 1, 315–330.

Duff, I. S. 2004. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw. 30,* 2, 118–144.

Duff, I. S. and Reid, J. K. 1983. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw. 9*, 302–325.

Duff, I. S. and Reid, J. K. 1984. The multifrontal solution of unsymmetric sets of linear equations. *SIAM J. Sci. Statist. Comput. 5,* 3, 633–641.

Edlund, O. 2002. A software package for sparse orthogonal factorization and updating. *ACM Trans. Math. Softw. 28,* 4, 448–482.

Foster, L. V. 2009. Calculating ranks, null spaces, and pseudoinverse solutions for sparse matrices using SuiteSparseQR. In *LA09: SIAM Conference on Linear Algebra 2009.* http://www.math.sjsu.edu/∼foster/spqr_rank_LA09.pdf.

Geist, G. A. and Ng, E. G. 1989. Task scheduling for parallel sparse Cholesky factorization. *Intl. J. Parallel Programming 18,* 4, 291–314.

George, A. 1973. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal. 10,* 2, 345–363.

George, A. and Heath, M. T. 1980. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra Appl. 34*, 69–83.

George, A. and Liu, J. W. H. 1981. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ.

George, A., Liu, J. W. H., and Ng, E. G. 1988. A data structure for sparse QR and LU factorizations. *SIAM J. Sci. Statist. Comput. 9,* 1, 100–121.

George, A. and Ng, E. G. 1987. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Statist. Comput. 8,* 6, 877–898.

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

Gilbert, J. R., Moler, C., and Schreiber, R. 1992. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl. 13,* 1, 333–356.

Givens, W. 1958. Computation of plane unitary rotations transforming a general matrix to triangular form. *SIAM J. Appl. Math. 6*, 26–50.

Golub, G. H. 1965. Numerical methods for solving linear least squares problems. *Numer. Math. 7*, 206–216.

Goto, K. and van de Geijn, R. 2008. High performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw. 35,* 1 (July), 4. Article 4, 14 pages.

Grigori, L., Cosnard, M., and Ng, E. G. 2007. On the row merge tree for sparse LU factorization with partial pivoting. *Bit 47,* 1, 45–76.

Heath, M. T. 1982. Some extensions of an algorithm for sparse linear least squares problems. *SIAM J. Sci. Statist. Comput. 3,* 2, 223–237.

Heath, M. T. and Sorensen, D. C. 1986. A pipelined Givens method for computing the QR factorization of a sparse matrix. *Linear Algebra Appl. 77*, 189–203.

Householder, A. S. 1958. Unitary triangularization of a nonsymmetric matrix. *J. ACM 5*, 339–342.

Karypis, G. and Kumar, V. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 359–392.

Lipton, R. J., Rose, D. J., and Tarjan, R. E. 1979. Generalized nested dissection. *SIAM J. Numer. Anal. 16*, 346–358.

Liu, J. W. H. 1986. On general row merging schemes for sparse Givens transformations. *SIAM J. Sci. Statist. Comput. 7,* 4, 1190–1211.

Liu, J. W. H. 1989. The multifrontal method and paging in sparse Cholesky factorization. *ACM Trans. Math. Softw. 15,* 4, 310–325.

Lu, S. M. and Barlow, J. L. 1996. Multifrontal computation with the orthogonal factors of sparse matrices. *SIAM J. Matrix Anal. Appl. 17,* 3, 658–679.

Matstoms, P. 1994. Sparse QR factorization in MATLAB. *ACM Trans. Math. Softw. 20,* 1, 136–159.

Matstoms, P. 1995. Parallel sparse QR factorization on shared memory architectures. *Parallel Computing 21,* 3, 473–486.

Oliveira, S. 2001. Exact prediction of QR fill-in by row-merge trees. *SIAM J. Sci. Comput. 22,* 6, 1962–1973.

Pierce, D. J. and Lewis, J. G. 1997. Sparse multifrontal rank revealing QR factorization. *SIAM J. Matrix Anal. Appl. 18,* 1, 159–180.

Pothen, A. and Fan, C. 1990. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw. 16,* 4, 303–324.

Reinders, J. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly Media, Sebastopol, CA.

Robison, A. 2008. Architect of Intel's Threading Building Blocks, personal communication.

Schreiber, R. and Van Loan, C. F. 1989. A storage-efficient WY representation for products of Householder matrices. *SIAM J. Sci. Statist. Comput. 10,* 1, 53–57.

Stewart, G. W. 1998. *Matrix algorithms, Volume 1: Basic decompositions.* SIAM, Philadelphia.

Sun, C. 1996. Parallel sparse orthogonal factorization on distributed-memory multiprocessors. *SIAM J. Sci. Comput. 17,* 3, 666–685.

van de Geijn, R. A. 1997. *Using PLAPACK.* Scientific and Engineering Computation Series. MIT Press, Cambridge, MA.

Received Month Year; revised Month Year; accepted Month Year