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 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^TA. Rank-detection is performed within each frontal matrix using Heath's method, which does not require column pivoting. 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 (or a permuted matrix AP), resulting in the factorization $A = QR$ or $AP = QR$. The resulting factors can be used to solve a least-squares problem, to find the basic solution of an under-determined system $Ax = b$, or to find a minimum 2-norm solution of an under-determined system $A^Tx = b$.

The earliest sparse direct methods operated on A one row or column at a time ([Björck 1996; George et al. 1988; George and Heath 1980; Heath 1982; Heath and Sorensen 1986]; see also [Davis 2006]). These methods are unable to reach a substantial fraction of the theoretical peak performance of modern computers

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.

because of their irregular access of memory, although they are very competitive when R remains 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 small dense frontal matrices; the idea was first used for symmetric indefinite matrices [Duff and Reid 1983] and later extended to sparse LU [Duff and Reid 1984; Davis and Duff 1997; Davis 2004] and sparse Cholesky factorization [Liu 1989]. [Puglisi 1993] first extended the multifrontal approach to sparse QR factorization; other prior implementations of multifrontal sparse QR factorization include [Matstoms 1994; 1995; Amestoy et al. 1996; Lu and Barlow 1996; Sun 1996; Pierce and Lewis 1997; 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. Its two techniques for exploiting multicore architectures are discussed in Section 4. Comparisons with existing sparse QR factorization methods are given in Section 5. Throughout the paper, fixed width font (`x=A\b`, for example) denotes MATLAB notation, except in the figures.

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 A, (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 A with different numerical values but identical nonzero pattern. The sparse QR factorization is no exception to this rule, although one of the features for reducing work and fill-in presented here (exploiting singletons) can only be done when the symbolic phase has access to the numerical values of A.

2.1 Exploiting Singletons

The *structural rank* of A is based solely on its nonzero pattern, and is given by `sprank(A)` in MATLAB [Davis 2006]. It is the largest numeric rank of A that can be obtained via any selection of numerical values for these entries. If `sprank(A)` is less than $\min(m, n)$, then A is *structurally rank-deficient*.

A *column singleton* is a column j of the matrix A with a single nonzero entry a_{ij} whose magnitude is larger than a given threshold τ; the corresponding row i is called a *row singleton*. If the matrix A has a completely zero column then that column is still a "singleton," but it has no corresponding row singleton. The column singleton is permuted to the top left corner and removed from A. This is repeated until no more singletons exist, giving the block matrix

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

If A has full structural rank, R_{11} will be square and upper triangular and all its

diagonal entries will larger in magnitude than τ. If the matrix is structurally rank-deficient then there may be column singletons with no corresponding row singletons. These can be permuted to the end of R_{11}, so that R_{11} is a rectangular and upper trapezoidal. For example, suppose A has four column singletons, but the third one has no corresponding row singleton:

$$A = \begin{bmatrix} r_{11} & x & x & x & x & x \\ & r_{22} & x & x & x & x \\ & & & r_{34} & x & x \\ & & & & x & x \\ & & & & x & x \end{bmatrix}.$$

The matrix R_{11} can be permuted into the upper trapezoidal form

$$R_{11} = \begin{bmatrix} r_{11} & x & x & x \\ & r_{22} & x & x \\ & & r_{34} & \end{bmatrix}.$$

The sparse QR factorization for the singleton rows and columns of A requires no numerical work at all, and no fill-in is incurred in R_{11} or R_{12}. Once singletons are removed, the remaining matrix A_{22} is ordered, analyzed, and factorized by the multifrontal sparse QR algorithm discussed below.

Singletons frequently arise in real applications. University of Florida Sparse Matrix Collection [Davis 2008b] contains 353 linear programming problems (with $m < n$). For 215 of them, every column in the matrix is a column singleton, A_{22} is empty, and the QR factorization for a basic solution takes no floating-point work. For many of the rest (53 problems), A_{22} has less than half the columns of A. Only 29 have no column singletons. A minimum 2-norm solution of an under-determined system requires the factorization of A^T; in this case, 169 of them have at least one column singleton. The corresponding rows for these column singletons can be very dense, so that removing them from A prior to the QR factorization of A_{22} can greatly reduce work and fill-in (a single dense row of A causes A^TA and R to become completely nonzero). Of the 30 least-squares problems in the collection, 15 have at least one column singleton.

The algorithm is much like a breadth-first search of the graph of A, except that it can stop early when no more candidate singletons exist. It requires access to the rows of A, and thus a transpose of A is required since SuiteSparseQR takes as input the matrix A in compressed column form. However, the transpose of A is needed later to find the fill-reducing ordering, so this work is not wasted. Once the singletons are removed, their rows and columns are pruned (in-place) from the row-form copy of A, and the resulting matrix is then passed to the fill-reducing ordering. Thus, excluding the transpose (which must be done anyway), the time taken for finding singletons is linear in the number of nonzeros in R_{11} and R_{12}, plus $O(n)$ time for the initial scan. Only if singletons exist is the matrix pruned, taking $O(|A|)$ time, where $|A|$ denotes the number of nonzeros in the matrix A. If the symbolic analysis is to be reused by subsequent matrices with different numerical values, singletons are not exploited because they conflict with how rank-deficient matrices are handled.

No prior multifrontal sparse QR method exploits singletons, although MA49

[Amestoy et al. 1996; Puglisi 1993] and Edlund's method [Edlund 2002] both have an option for using a permutation to block triangular form (BTF). Finding singletons in A is much faster than a Dulmage-Mendelsohn decomposition such as `dmperm` in MATLAB [Davis 2006] or a permutation to block triangular form (BTF) [Duff 1977; 1981; Pothen and Fan 1990]. Finding singletons or the BTF takes far less time than the numerical factorization itself, however. Each singleton is a 1-by-1 block in BTF of A, if A is not rank-deficient. MA49 uses the BTF of [Pothen and Fan 1990], which also splits the under- and overdetermined blocks into the connected components of its bipartite graph, giving a block diagonal form for these two submatrices with rectangular diagonal blocks.

Using a BTF ordering is suitable only for full-rank matrices. For rank-deficient matrices where `rank(A)` $<$ `sprank(A)`, the BTF ordering may be numerically unsuitable. SuiteSparseQR avoids this problem by only allowing singletons whose magnitude exceeds τ. If numerical rank deficiency were not considered, singletons could be exploited in a purely symbolic analysis phase, but SuiteSparseQR does not provide this option.

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 A (or A_{22}, more precisely).

If the matrix A is *strong Hall*, the nonzero pattern of the factor R is identical to the Cholesky factorization of A^TA [Coleman et al. 1986; George and Heath 1980]. A strong-Hall matrix is a matrix that cannot be permuted into block upper triangular form. Structurally rank-deficient matrices are never strong-Hall, but non-strong-Hall matrices can have full structural and numeric rank (consider an n-by-n diagonal matrix, for example, which has n diagonal blocks in its block triangular form).

The symbolic analysis is thus modelled after the Cholesky factorization of A^TA. However, SuiteSparseQR never forms A^TA unless a particular ordering method requires it. It can order the matrix A (or the pruned matrix after removing singletons) with COLAMD [Davis et al. 2004a; 2004b]. COLAMD finds a fill-reducing ordering P that attempts to reduce the number of nonzeros in the QR factorization of AP or the Cholesky factorization of $(AP)^T(AP)$. It does not form A^TA to perform this ordering, but uses the graph of A itself; each row of A forms a clique (or hyperedge, equivalently) in the graph of A^TA. As the elimination proceeds, new cliques are formed, and these are also held as a list of the nodes in the cliques, just as each row of A represents a clique in the graph.

SuiteSparseQR uses CHOLMOD [Chen et al. 2009; Davis and Hager 2009] for its ordering and analysis phase, and thus can use any ordering method available to CHOLMOD: COLAMD on A, AMD[1] on the explicit pattern of A^TA [Amestoy et al. 1996; 2004], METIS [Karypis and Kumar 1998] applied to A^TA, CHOLMOD's nested dissection ordering based on METIS, or any combination of the above, where multiple methods can be tried and the ordering with the least fill in the Cholesky factorization of A^TA used. The asymptotic run times of these ordering methods do not have tight bounds. However, experimental results have shown that COLAMD and AMD, with occasional exceptions, take time roughly proportional to the num-

[1]AMD: an acronym for Approximate Minimum Degree, not to be confused with AMD, Inc.

ber of nonzeros in A and A^TA, respectively [Davis 2008b]. This is not an upper bound, but an average based on experiments with nearly 2000 matrices in the UF Sparse Matrix Collection, nearly all of which come from real applications.

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 and the column-count algorithm of [Gilbert et al. 2001] were developed. Edlund uses optionally uses COLAMD [Edlund 2002], but does not use the column-count method in [Gilbert et al. 2001].

2.3 Symbolic factorization

CHOLMOD then performs the symbolic supernodal Cholesky factorization of A^TA (or $A_{22}^TA_{22}$ if singletons have been removed). Each supernode in the Cholesky factor L represents a group of adjacent columns with identical or nearly identical nonzero pattern, which is the same as a set of rows of the R factor for a QR factorization. Each supernode from CHOLMOD becomes a single frontal matrix for the multifrontal QR factorization.

The supernodal Cholesky analysis starts by computing the elimination tree of A^TA (also called the column elimination tree of A) and the column counts for the Cholesky factorization of A^TA, without forming A^TA itself [Gilbert et al. 2001]. The column counts are the number of nonzeros $|L_{*j}|$ in each column of L. The time taken by this step is almost $O(|A|)$, where "almost" is theoretically at worst $O(|A| \log n)$. If CHOLMOD were to use a theoretically optimal implementation of the disjoint set-union-find [Tarjan 1975], this time would be $O(|A|\alpha(|A|, n))$ where α is the inverse Ackermann function, a very slowly growing function. CHOLMOD uses a method that is faster in practice but theoretically bounded by $O(|A| \log n)$. In practice, this step takes essentially $O(|A|)$ time; the super-linear growth is not observed.

The column counts and column elimination tree then determine the supernodes. Two columns j and $j+1$ are in the same supernode if $\text{parent}(j) = j+1$ and $|L_{*j}| = |L_{*,j+1}| + 1$, which implies that the two columns have the same nonzero pattern (except for j itself, which appears as a row index only in L_{*j}). Note that this test does not require the nonzero pattern of L itself, which takes time $O(|L|)$ to find. For these two columns to reside in the same *fundamental* supernode, j must also be the only child of $j+1$; SuiteSparseQR does not use this restriction.

Time and memory usage can be decreased by exploiting relaxed supernodes, where two adjacent columns may be combined into a supernode where the nonzero patterns of $L_{*j} \setminus \{j\}$ and $L_{*,j+1}$ are similar but not identical. CHOLMOD does this relaxed amalgamation based solely on the nonzero counts of L. Once the relaxed supernodes are found, the nonzero pattern of each supernode is computed, taking time proportional to the number of nonzeros in the leftmost columns of each supernode. This is also the amount of integer memory space required to represent the supernodal nonzero pattern of L. A dense matrix consists of a single supernode, so this step takes as little as $\Omega(n)$ time. In any case, $O(|L|)$ is a loose upper bound on the time; it is typically much less than this because of relaxed amalgamation.

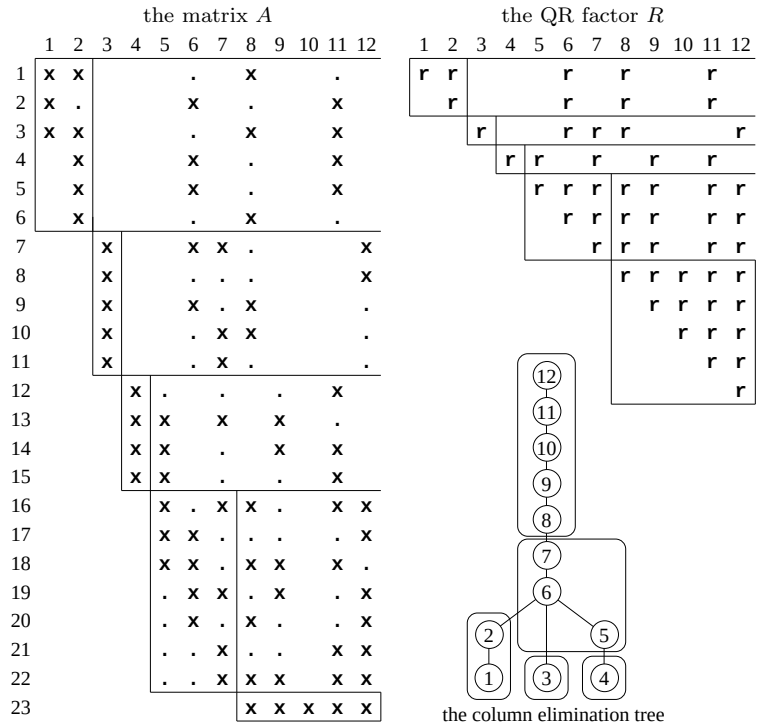

Fig. 1. A sparse matrix A, its factor R, and its column elimination tree

The result of this step is the supernodal elimination tree and the supernodal representation of L for the Cholesky factorization of $A^T A$, but recall that singletons are typically removed from A prior to this analysis. If A is strong Hall, then it will have no singletons at all, and also this analysis will be exact. Otherwise, it is possible that the analysis gives an upper bound on the nonzero pattern of R.

A more concise description of the pattern of R for the non-strong-Hall case could be obtained from the row-merge tree [Oliveira 2001]. However, SuiteSparseQR handles rank deficient matrices using the method of [Heath 1982]. For a multifrontal sparse QR factorization, Heath's method requires R to accommodate any nonzero entry in the Cholesky factorization of $A^T A$. SuiteSparseQR thus does not use the row-merge tree.

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 with identical nonzero pattern. The horizontal lines in A subdivide the rows according to the frontal matrix of R 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. The parent of i is given by the smallest $j > i$ for which $r_{ij} \neq 0$. The supernodes are shown in the column elimination tree as rounded boxes. This example is continued in Section 3, which

shows the assembly and numeric factorization of two of the five frontal matrices.

After CHOLMOD's supernodal analysis, SuiteSparseQR continues its analysis by sorting the rows of AP according to the column index of the leftmost nonzero in each row of AP. This results in the permuted matrix P_2AP. A row i of P_2AP is assembled in the frontal matrix whose pivotal columns contain the leftmost column index j of row i. The time taken by this step is $O(|A|)$.

Finally, SuiteSparseQR simulates the sparse multifrontal QR factorization by counting how many rows may appear in each frontal matrix, and determining the amount work required to perform the Householder QR factorization of each frontal matrix. It also finds the *staircase* for each frontal matrix, which is the row index of where the zero entries start in each column. This analysis provides an upper bound in case A is rank-deficient, and is exact if rank-detection is disabled. The time taken by this step is proportional to the number of integers required to represent the supernodal pattern of L.

The total time and memory usage for the analysis is thus 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 multifrontal structure of the matrix R. This can be much less than the time and memory required to form A^TA, particularly when $m << n$. In practice, this time is so low that the total QR factorization time of a dense matrix stored in sparse format, including the ordering, analysis, and numeric factorization, is not much higher, and sometimes even lower, than the time taken by LAPACK [Anderson et al. 1999] to factorize the same matrix in a dense storage format.

None of the six prior multifrontal sparse QR methods use the results of [Gilbert et al. 2001]. Five of them use the elimination tree of A^TA and must form the nonzero pattern of A^TA explicitly. [Edlund 2002] does not form A^TA, 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.

To illustrate the difference between the row-merge tree and the column elimination tree, consider the following matrix,

$$A = \begin{bmatrix} a_{11} & a_{13} & a_{13} \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix}.$$

The row-merge tree is a forest of three nodes and no edges, and r_{23} is structurally zero. The column elimination tree is a chain, $1 \rightarrow 2 \rightarrow 3$, A^TA is all nonzero, and r_{23} is structurally nonzero. If $|a_{11}| < \tau$ it is treated as zero, and in Heath's method row 1 must be completely zeroed via Givens rotations. Row 2 fills in from row 1, because column 2 is the parent of column 1 in the column elimination tree, and r_{23} becomes nonzero. If all the diagonal entries of A for this particular matrix are larger than τ, the row-merge tree is sufficient. The matrix A would also be permuted into R via the singleton preordering described in Section 2.1.

3. NUMERIC FACTORIZATION

For its numerical factorization, SuiteSparseQR uses much of the theory and algorithms from the prior implementations of multifrontal sparse QR factorization

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
	r	r	r	r	r
	h	r	r	r	r
	h	h	c	c	c
		h	h	c	c
		h	h	h	c
		h	h	h	h

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

[Puglisi 1993; 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 article). MA49 does not have the option of discarding the Householder vectors except when solving problems via the seminormal equations (which do not require Q, but which can be less accurate). In contrast, SuiteSparseQR can be given the right-hand-side b of a least-squares problem in which case it factorizes the concatenated matrix $[A\,b]$ and can discard the Householder vectors once each frontal matrix is factorized. SuiteSparseQR's numerical factorization handles rank-deficient frontal matrices; MA49 does not.

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 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 referred to as the *pivotal* columns for this front (Amestoy, Duff, and Puglisi refer to them as *fully-summed variables*; 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 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 this front. The parent (front 4) is the front which contains the first non-pivotal column (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. Just 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,

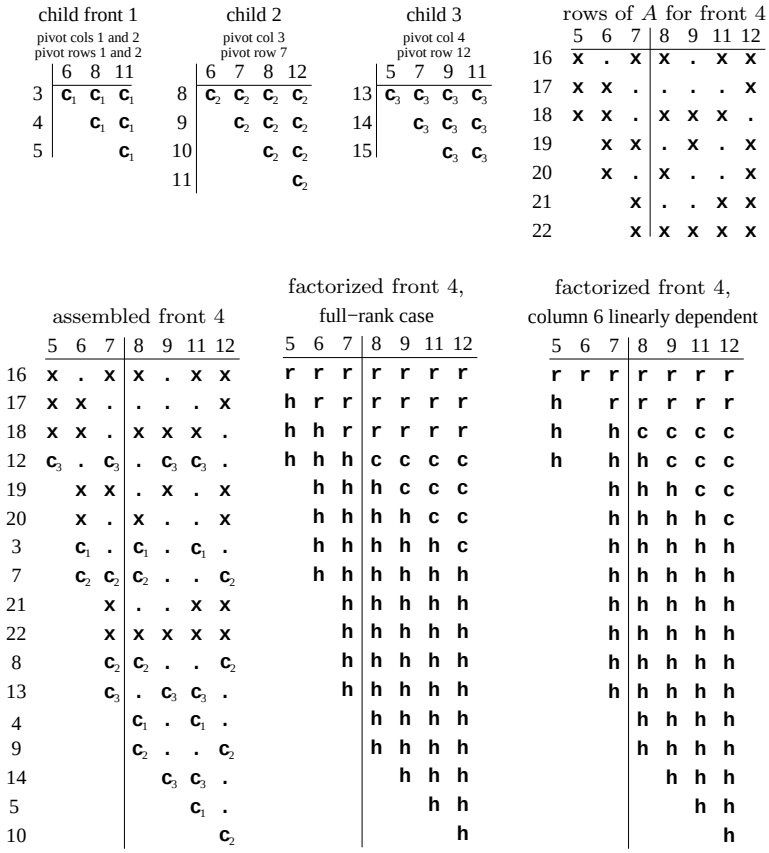

Fig. 3. Assembly and factorization of a frontal matrix with three children

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 is discussed in the next section.

[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 $\tau \geq 0$, 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.

3.2 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. The dense frontal factorization algorithm in SuiteSparseQR gives T a leading dimension of b, instead. As a result, its dense QR factorization is faster than DGEQRF when $m << n$, and achieves the same performance as DGEQRF 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.

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. 2009]. The column that needs to be removed may 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 $(AP)^TAP$ 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. The MATLAB sparse `qr` also uses Heath's strategy for rank-deficient matrices, but (in MATLAB version R2008b) does not use a multifrontal method. Instead, the MATLAB sparse `qr` in R2008b and earlier uses Gilbert's implementation of [George and Heath 1980] based on Givens rotations.

As an example of how rank-deficient matrices are handled in SuiteSparseQR, consider Figure 3. After the Householder reflection of column 5 is applied to column 6, suppose the 2-norm of column 6 drops below the threshold τ. The result is shown in the bottom right of Figure 3. The Householder reflection for column 6 is skipped, and this front holds one less row of R than expected. This gives 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. 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 τ. 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; 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 R is identical.

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 dis-

carded). 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.

3.3 Using the QR factorization

If P is the fill-reducing column permutation, then the QR factorization is $AP = QR$. The solution to a least-squares problem ($\min_x ||b - Ax||_2$) is given by solving $\begin{bmatrix} R \\ 0 \end{bmatrix} P^T x = Q^T b$, or `x=P*(R\(Q'*b))` in MATLAB notation [Golub 1965]. If given the right-hand-side b when A is factorized, SuiteSparseQR applies the Householder reflections to b while factorizing A, giving $c = Q^T b$ when the frontal matrix factorization is complete. Thus, the Householder reflections can be discarded after each frontal matrix is factorized.

Computing the minimum 2-norm solution to an underdetermined system $Ax = b$ can be done with the factorization of A^T (namely, $A^T P = QR$). The solution is found by solving $\begin{bmatrix} R^T 0 \end{bmatrix} Q^T x = P^T b$, or `x=Q*(R'\(P'*b))` in MATLAB notation. The Householder reflections cannot be applied and discarded during factorization. SuiteSparseQR does not form Q explicitly, but instead saves the set of Householder vectors. After the forward solve (`c=R'\(P'*b)`), a separate SuiteSparseQR function then applies Q to c, giving `x=Q*c`.

Although it is not required to solve a least-squares problem or a linear system, if needed SuiteSparseQR can construct the matrix form of Q in two different ways. It can keep the Householder reflections and then apply them when done to a sparse identity matrix I. Alternatively, it can apply them to $b = I$ during factorization and discard them, resulting in the matrix $C = Q^T$ which can then be simply transposed to obtain Q. In practice, the latter method is always faster.

4. PARALLELISM

At least two opportunities for parallelism exist within a multifrontal sparse QR factorization. The first opportunity arises in the column elimination 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 5.

In its analysis phase, SuiteSparseQR determines the column elimination tree (with n nodes) and the frontal-matrix tree (with n_f nodes; an amalgamated version in which each node is a frontal matrix consisting of one or more nodes in the column elimination tree). Next, an estimate of the floating-point work in each front is computed (this estimate is exact if the rank-detecting tolerance τ is disabled). This pass also determines the size of each frontal matrix and the stack size needed for the contribution blocks for a sequential factorization (parallelism can still occur within the BLAS if the matrix is factorized with a single TBB task).

The goal of the parallel analysis phase 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. 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 ω is the total flop count for the entire QR factorization, and α and β are user-definable parameters that control the task tree granularity. Typically α 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 α and β.

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. This completes the analysis required for parallel factorization.

In the numerical 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

	AMD Opteron 875		Intel T2500 laptop	
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

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. Both sequential and parallel performance results are discussed in the next section.

5. EXPERIMENTAL RESULTS

In this section, the different ordering methods for SuiteSparseQR are compared, and a default ordering strategy for SuiteSparseQR is defined. The performance of SuiteSparseQR is compared with other solvers. When different factorization methods are compared, they are always compared using the same fill-reducing preordering. To test the methods, all rectangular matrices in the University of Florida Sparse Matrix Collection [Davis 2008b] are used. As of September 2008, the collection contains 30 least-squares problems and 353 underdetermined systems (most of which come from linear programming problems).

Most results are from a Rackable Systems shared-memory computer with eight dual-core AMD Opteron 875 processors. Single-core results in Sections 5.2 to 5.5 are in MATLAB R2007a with the AMD ACML BLAS; parallel results in Section 5.6 and the statistics in Table I are in MATLAB R2008a. Additional results are presented on a Dell Latitude D620 dual-core laptop. Table I lists the statistics and performance of these two computers. All timings presented in this paper are in MATLAB with `tic` and `toc` (wall-clock time), using the version of the 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 method, since it computes the QR factorization of many small dense frontal matrices.

5.1 The methods

The following methods are compared in this section:

—**SuiteSparseQR**: When given an underdetermined system ($m < n$), SuiteSparseQR can factorize A to find a basic solution or it can factorize A^T to find a minimum 2-norm solution. All other solvers presented in this section can perform only one or the other (or can do both but only one can be done efficiently).
  (1) If $m \geq n$, `A*P=Q*R` is factorized, `c=Q'b` is computed during factorization, and `Q` is discarded. The least-squares solution is `x=P*(R\c)`.
  (2) If $m < n$, the factorization is `A'*P=Q*R`. The matrix `Q` is kept in Householder form. The minimum 2-norm solution is `x=Q*(R'\(P'*b))`.
  (3) If $m < n$, `A*P=Q*R` is factorized, `c=Q'b` is computed during factorization, and `Q` is discarded. The basic solution is `x=P*[R1\c;0]` where `R1` is the leading square submatrix of `R`. If `R` is "squeezed" then a column post-ordering is applied so that `R1` consists of all columns found to be linearly independent, where `R1` is square with diagonal entries greater than τ.

—**MA49** [Amestoy et al. 1996; Puglisi 1993]: MA49 provides a non-default option to use the Level-3 BLAS, but for large matrices it was found that using the BLAS was always faster, so only results with the BLAS are presented.
  (1) If $m \geq n$, `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) This 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.
  (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]. By default, `P` is found via COLMMD, but this is replaced in these experiments with AMD or COLAMD.
  (1) If $m \geq n$, `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) If $m < n$, `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.

5.2 Comparing solvers for least-squares problems

In these comparisons the AMD ordering on $A^T A$ 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,

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

Table II. Large least-squares problems

the 11 largest (with $m \geq 10,000$) are shown in Table II (excluding one matrix too large for this 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. Since the collection has so few full-rank least-squares problems, the four rank-deficient matrices were also solved via Tikhonov regularization (appending γI 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 γI ensures that the matrix has no column singletons, unless A itself has columns with no nonzeros in them at all. 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).

5.3 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 underdetermined 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. The relative timing results in Table IV are typical for larger matrices as well, although unlike nearly all other matrices, the "nug" matrices in the collection experience extreme fill-in in R.

Mtx	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**	-	-	-	-	-	-	17318.2
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**	-	-	-	-	-	-	67928.6
11	**5568.5**	**5.86**	12615.4	41.8	14741.7	41.83	15956.9	7.38	>3 days
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

	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

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

The primary difference between the two methods is that SuiteSparseQR finds a more precise estimate of the space needed for Q. This estimate is exact if the rank-deficiency check is disabled and an upper bound by default (the default tolerance was used for these experiments). The analysis of MA49 always finds an upper bound, even though it only factorizes full-rank matrices. The second difference

name	$m/10^3$	$n/10^3$	$\|A\|/10^3$	SuiteSparseQR		MA49	
				time	mem	time	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	1054.0	**1.4**	0.08	3.6	**0.07**
Meszaros/fxm3_16	41.3	85.6	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	1055.1	**4.1**	**0.23**	140.9	0.48
Mittelmann/watson_2	352.0	677.2	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

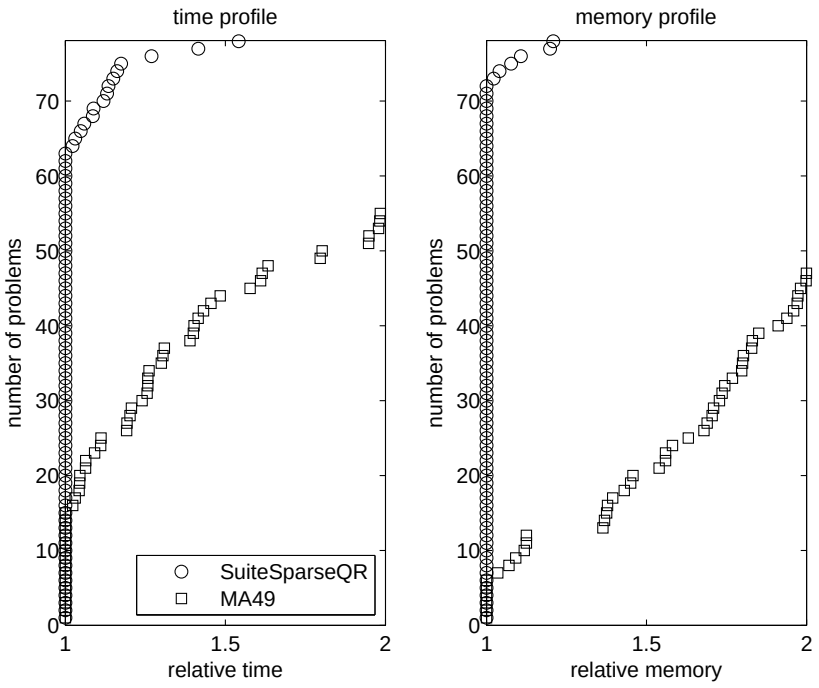

Fig. 4. Performance profiles for minimum 2-norm solutions with the AMD ordering

between the methods is that SuiteSparseQR exploits singletons, whereas the BTF pre-ordering in MA49 is only available for least-squares problems.

Mtx	AMD		COLAMD		METIS		Default		Best	
	time	mem	time	mem	time	mem	time	ord	time	ord
1	0.1	0.00	0.1	0.00	0.2	0.00	0.1	COL	0.2	AMD
2	1222.0	1.62	922.8	1.63	1218.1	1.62	1221.1	AMD	941.7	COL
3	0.1	0.01	0.1	0.01	0.3	0.01	0.1	AMD	0.4	AMD
4	0.8	0.02	0.7	0.01	0.8	0.01	0.8	AMD	0.9	MET
5	0.1	0.01	0.2	0.01	0.3	0.01	0.1	AMD	0.4	AMD
6	0.1	0.01	0.2	0.01	0.5	0.01	0.2	AMD	0.5	AMD
7	6654.6	5.38	1617.9	2.26	2163.5	2.41	1614.1	COL	1801.4	COL
8	187.8	0.45	516.9	1.01	95.3	0.34	96.2	MET	96.6	MET
9	1.2	0.04	46.9	0.20	1.5	0.04	1.3	AMD	2.7	AMD
10	874.4	2.86	844.8	2.43	386.9	0.62	408.2	MET	414.4	MET
11	5568.5	5.86	5203.2	5.56	2696.1	3.20	2687.6	MET	2698.1	MET

Table VI. SuiteSparseQR results for large least-squares problems with different orderings

5.4 Selecting an ordering for SuiteSparseQR

The fill-reducing ordering has a huge impact on the performance of any sparse matrix factorization. All methods discussed here can use any ordering, but a single ordering method was selected in the results above to remove one source of variability when comparing between methods. This section considers the effect of different orderings on the performance of SuiteSparseQR.

Ideally, a method would be able to select the right ordering for each matrix automatically without trying them all. SuiteSparseQR relies on CHOLMOD's ordering/analysis phase which has a simple interface that allows the user to specify a list of orderings; CHOLMOD tries them all and picks the one with the least fill-in in R. Trying all three (AMD, COLAMD, and METIS) increases the ordering/analysis time, however.

SuiteSparseQR was tested with different orderings on all 30 least-squares problems and all 353 minimum 2-norm problems for underdetermined systems in the collection. The default ordering strategy of SuiteSparseQR was determined based on these results. In this default strategy, COLAMD is used if $m \leq 2n$ after removing singletons; otherwise, CHOLMOD's default ordering is used. CHOLMOD's default strategy tries AMD and then tries METIS only if the AMD ordering leads to high fill-in and flop count. Let f be the flop count for the Cholesky factorization of $(AP)^T AP$ using the ordering P from AMD. If $f/|R| \geq 500$ and $|R|/|A| \geq 5$ then METIS is attempted and the best ordering (AMD or METIS) is used. Otherwise, AMD is used without trying METIS. CHOLMOD uses this strategy because the ordering time for METIS can be many times higher than AMD, although the extra ordering time is worth it for large matrices. When $m < n$ the time and memory required by AMD and METIS can be high (even exceeding the space needed for the numerical factorization), although there are problems with $m < n$ where AMD and/or METIS give better results.

Table VI illustrates the results on the 11 large least-squares problems from Table II. The first three columns (AMD, COLAMD, and METIS) reflect the time and memory taken if just one ordering is used; the AMD column is the same as the SuiteSparseQR column in Table III. The column labeled "Default" is SuiteSparseQR's default ordering strategy. The last column, "Best," reflects the time

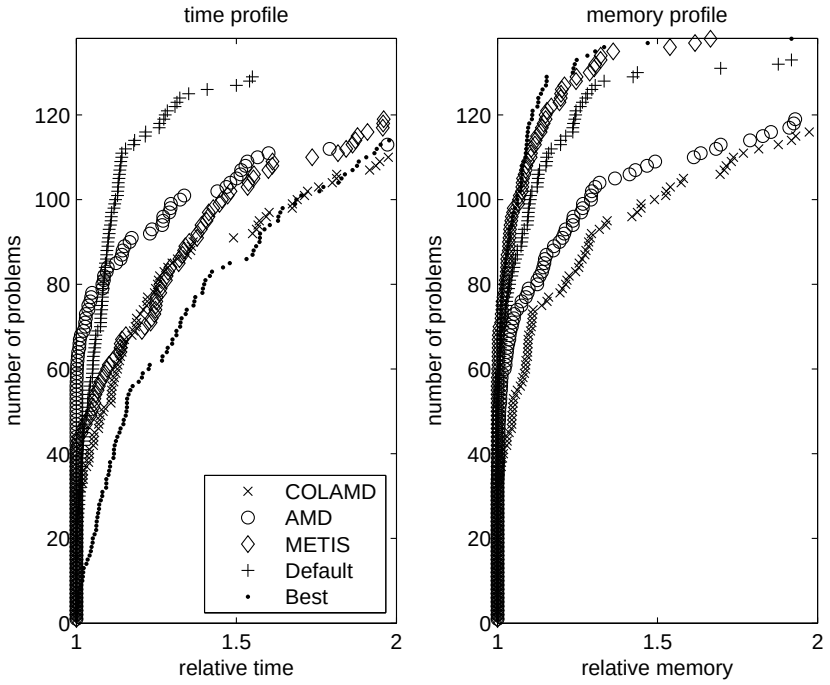

Fig. 5. SuiteSparseQR performance profiles with different orderings

taken by SuiteSparseQR to try all three orderings and use the one with the smallest $|R|$. The last two columns show the ordering actually used by these two strategies (with COLAMD and METIS abbreviated COL and MET, respectively).

Figure 5 presents time and memory profiles for these five strategies using the 11 least-squares problems used in Table VI and 138 minimum 2-norm problems with $n \geq 10,000$ (this excludes the 12 matrices that are too large for SuiteSparseQR).

The "Best" ordering strategy typically finds an ordering that minimizes memory usage but the total time is high because of increased ordering time, and because lower fill-in does not always lead to a lower factorization time. The default strategy has the best time profile over all, and a very good memory profile.

5.5 Comparison with LAPACK

[Chen et al. 2009] showed that the ratio of the number of floating-point operations over the number of nonzeros in L was a good predictor of the GFlop rate that a sparse matrix factorization algorithm can attain. Figure 6 plots a similar metric for SuiteSparseQR with the AMD ordering. The X axis is the total flop count divided by the memory usage in bytes. The Y axis is the GFlop performance, with the top of the figure equal to the theoretical peak of 4.42 GFlops.

The flop count is found after relaxed amalgamation, but the Householder reflections are counted as if they are applied one at a time. They actually are applied as a block, via DLARFT and DLARFB. That is, SuiteSparseQR is actually performing

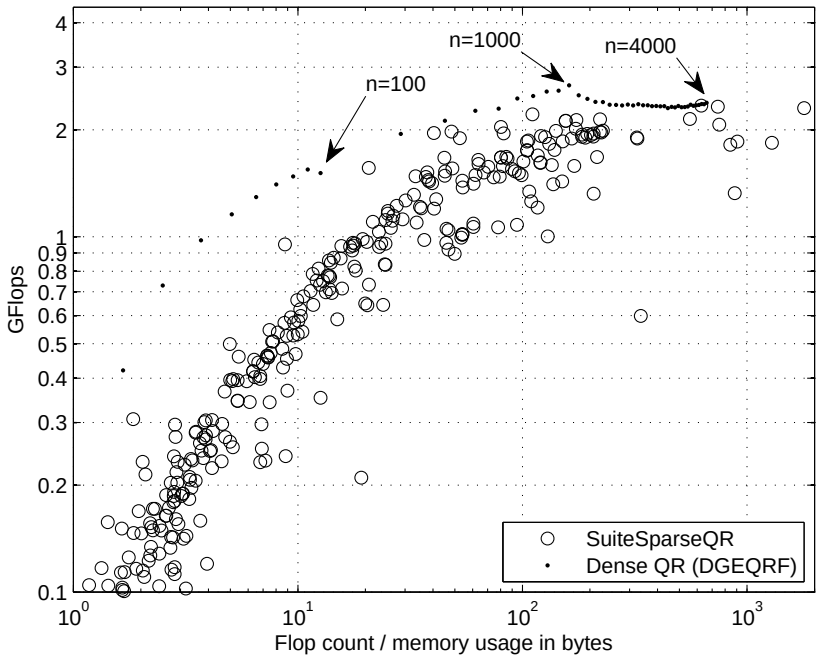

Fig. 6. SuiteSparseQR performance compared with LAPACK

more flops than is indicated in Figure 6, but these are useless flops on the zeros in the lower left part of each frontal matrix. Each circle is one of the 383 sparse matrices in the collection (all 30 least-squares problems, and all 353 minimum 2-norm problems), excluding smaller matrices that fall outside the plot and 12 matrices too large for SuiteSparseQR (requiring more than 64GB of memory to factorize).

The MATLAB statement `X=qr(A)` when `A` is dense is a light-weight interface to DGEQRF, except that it must make a copy of its input. For large matrices, this work is about 1% of the total time. The floating-point work is $2(mn^2 - n^3/3)$ for an m-by-n matrix, and if the copy of `X` is excluded, it uses $8mn$ bytes for the matrix and $8\min(32, m)n$ workspace. In Figure 6 the performance of `qr` for square dense matrices is shown. Its peak performance is 2.67 GFlops on a single core when $n = 1000$ (which does not include a forward/backsolve, but does include the copy of `X`). SuiteSparseQR obtains a peak of 2.49 GFlops on a single core, which includes the AMD ordering, the symbolic analysis, the numeric factorization, the application of Q to b, and the backsolve with R.

Like SuiteSparseQR, DGEQRF does most of its work in DLARF* routines, but unlike DGEQRF which allocates the 32-by-32 workspace matrix T with a leading dimension of n, SuiteSparseQR ensures this matrix has a leading dimension of 32. This can have a large impact on run time if $m << n$. For example, with the matrix `A=rand(100,20000)` and `S=sparse(A)`, the sparse QR (`R=spqr(S)`) is almost twice as fast as `R=qr(A)` on the dual-core laptop (0.64 vs 1.2 seconds).

5.6 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-based parallelism, but can be a rich source of BLAS-based parallelism. These two kinds of parallelism should be complimentary, 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 the TBB task tree, and the BLAS should use all the threads at 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 different multicore computers are shown in Tables VII and VIII.

Table VII 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 VIII 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. In Table VIII, k is the number of threads used in TBB, b is the number of threads used in the BLAS, and speedup is abbreviated as "sp." 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.

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

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 VII. Parallel performance on a dual-core Dell laptop

Matrix	TBB:1, BLAS:1		TBB:1, BLAS:b		TBB:k, BLAS:1		best (TBB:k, BLAS:b)			
	time	GFlops	speedup	b	speedup	k	sp.	k	b	GFlops
deltaX	97.5	2.11	2.99	10	4.13	16	5.70	15	6	12.01
landmark	1.6	0.76	1.07	5	1.93	14	1.93	14	1	1.47
ESOC	381.2	1.93	2.59	8	4.85	15	5.47	14	3	10.56
Rucci1	2525.8	2.45	3.70	12	2.44	16	5.76	14	12	14.10

Table VIII. Parallel performance on a 16-core AMD Opteron system

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. 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. Exploiting singletons and ordering/analyzing the matrix A without forming A^TA can lead to asymptotic reductions in time and memory usage, depending on the matrix. Its default ordering strategy is better than the underlying orderings it relies on by rapidly selecting a good ordering automatically without trying all three. Because it accurately predicts the memory required to hold Q as a set Householder vectors, its memory usage is lower than MA49. Unlike MA49, it can handle rank-deficient matrices, matrices with $m < n$, and complex matrices. Its MATLAB interface provides substantial improvements for the MATLAB user, such as the representation of Q as a collection of sparse Householder vectors which allow it to efficiently solve minimum 2-norm problems. The availability of the software and interfaces for MATLAB, C, and C++ programs are described in a companion paper, [Davis 2008a].

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

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.

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. 2009. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw. 35,* 3.

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. 2008a. Algorithm 8xx: SuiteSparseQR, a multifrontal multithreaded sparse QR factorization package. *ACM Trans. Math. Softw.*. under submission.

Davis, T. A. 2008b. University of Florida sparse matrix collection. Tech. rep., Univ. of Florida. under submission.

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.

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. 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.

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.

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.

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.

Puglisi, C. 1993. QR factorization of large sparse overdetermined and square matrices using a multifrontal method in a multiprocessor environment. Ph.D. thesis, Institut National Polytechnique de Toulouse. CERFACS report TH/PA/93/33.

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

Robison, A. 2008. Primary 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.

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

Tarjan, R. E. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 215–225.

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