Fine granularity sparse QR factorization for multicore based systems

Alfredo Buttari

CNRS-IRIT,
118 route de Narbonne, F-31062 Toulouse, France
alfredo.buttari@irit.fr

Abstract. The advent of multicore processors represents a disruptive event in the history of computer science as conventional parallel programming paradigms are proving incapable of fully exploiting their potential for concurrent computations. The need for different or new programming models clearly arises from recent studies which identify fine-granularity and dynamic execution as the keys to achieve high efficiency on multicore systems. This work presents an implementation of the sparse, multifrontal QR factorization capable of achieving high efficiency on multicore systems through using a fine-grained, dataflow parallel programming model.

Keywords: Multifrontal, Sparse, QR, Least-Squares, Multicore

1 Introduction

The QR factorization is the method of choice for the solution of least-squares problems arising from a vast field of applications including, for example, geodesy, photogrammetry and tomography (see [15, 3] for an extensive list).

The cost of the QR factorization of a sparse matrix, as well as other factorizations such as Cholesky or LU, is strongly dependent on the fill-in generated, i.e., the number of nonzero coefficients introduced by the factorization. Although the QR factorization of a dense matrix can attain very high efficiency because of the use of Householder reflections (see [16]), early methods for the QR factorization of sparse matrices were based on Givens rotations with the objective of reducing the fill-in. One such method was proposed by Heath and George [10], where the fill-in is minimized by using Givens rotations with a row-sequential access of the input matrix. In order to exploit the sparsity of the matrix, such methods suffered a considerable lack of efficiency due to the poor utilization of the memory subsystem imposed by the data structures that are commonly employed to represent sparse matrices.

The *multifrontal method*, first developed for the Cholesky factorization of sparse matrices [8] and then extended to the QR factorization [12, 9], quickly gained popularity over these approaches thanks to its capacity to achieve high performance on memory-hierarchy computers. In the multifrontal method, the

factorization of a sparse matrix is cast in terms of operations on relatively smaller dense matrices (commonly referred to as *frontal matrices* or, simply, *fronts*) which gives a good exploitation of the memory subsystems and the possibility of using Householder reflections instead of Givens rotations while keeping the amount of fill-in under control. Moreover, the multifrontal method lends itself very naturally to parallelization because dependencies between computational tasks are captured by a tree-structured graph which can be used to identify independent operations that can be performed in parallel.

Several parallel implementations of the QR multifrontal method have been proposed for shared-memory computers [14, 2, 7]; all of them are based on the same approach to parallelization which suffers scalability limits on modern, multicore systems (see Section 3.1).

This work describes a new parallelization strategy for the multifrontal QR factorization that is capable of achieving very high efficiency and speedup on modern multicore computers. This method leverages a fine-grained partitioning of computational tasks and a dataflow execution model [17] which delivers a high degree of concurrency while keeping the number of thread synchronizations limited.

2 The Multifrontal QR Factorization

The multifrontal method was first introduced by Duff and Reid [8] as a method for the factorization of sparse, symmetric linear systems and, since then, has been the object of numerous studies and the method of choice for several, high-performance, software packages such as MUMPS [1] and UMFPACK [6].

At the heart of this method is the concept of an *elimination tree*, extensively studied and formalized later by Liu [13]. This tree graph describes the dependencies among computational tasks in the multifrontal factorization. The multifrontal method can be adapted to the QR factorization of a sparse matrix thanks to the equivalence of the R factor of a matrix A and the Cholesky factor of the normal equation matrix $A^T A$. Based on this equivalence, the elimination tree for the QR factorization of A is the same as that for the Cholesky factorization of $A^T A$.

In a basic multifrontal method, the elimination tree has n nodes, where n is the number of columns in the input matrix A, each node representing one pivotal step of the QR factorization of A. Every node of the tree is associated with a frontal matrix that contains all the coefficients affected by the elimination of the corresponding pivot. The whole QR factorization consists in a bottom-up traversal of the tree where, at each node, two operations are performed:

- **assembly**: a set of rows from the original matrix is assembled together with data produced by the processing of child nodes to form the frontal matrix;
- **factorization**: one Householder reflector is computed and applied to the whole frontal matrix in order to annihilate all the subdiagonal elements in the first column. This step produces one row of the R factor of the original

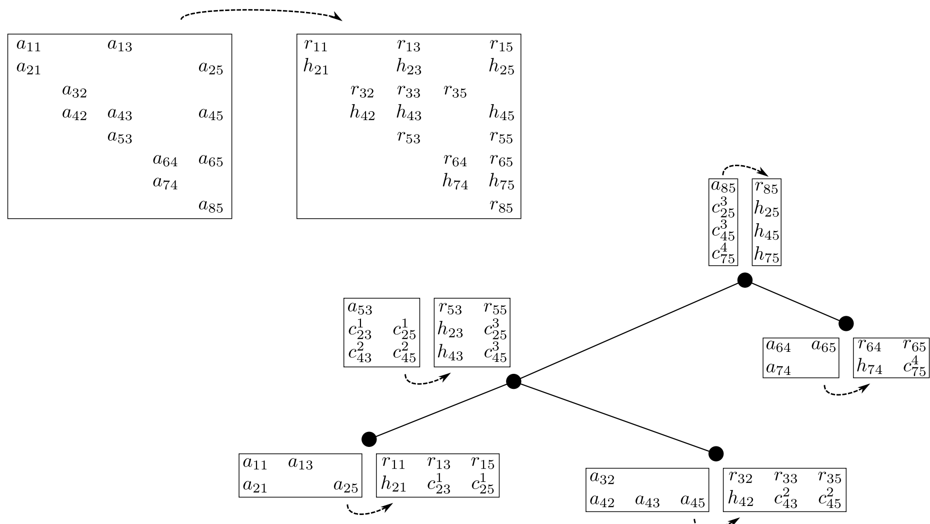

Fig. 1. Example of multifrontal QR factorization. The c_{ij} coefficients denote the contribution blocks.

matrix and a complement which corresponds to the data that will be later assembled into the parent node (commonly referred to as a *contribution block*). The Q factor is defined implicitly by means of the Householder vectors computed on each front.

Figure 1 shows how the QR factorization of the small 8×5 matrix in the top-left part can be achieved through the multifrontal method. The related elimination tree is depicted in the bottom-right part of the figure. Beside each node of the tree, the corresponding frontal matrix is shown after the assembly and after the factorization operations (the transition between these two states is illustrated by the dashed arrows).

In practical implementations of the multifrontal QR factorization, nodes of the elimination tree are amalgamated to form *supernodes*. The amalgamated pivots correspond to rows of R that have the same structure and can be eliminated at once within the same frontal matrix without producing any additional fill-in. This operation can be performed by means of efficient Level-3 BLAS routines. The amalgamated elimination tree is also commonly referred to as *assembly tree*.

In order to reduce the operation count of the multifrontal QR factorization, two optimizations are commonly applied:

1. once a frontal matrix is assembled, its rows are sorted in order of increasing index of the leftmost nonzero (Figure 2 (*middle*)). The number of operations can thus be reduced by ignoring the zeroes in the bottom-left part of the frontal matrix;
2. the frontal matrix is completely factorized (Figure 2 (*right*)). Despite the fact that more Householder vectors have to be computed for each frontal matrix,

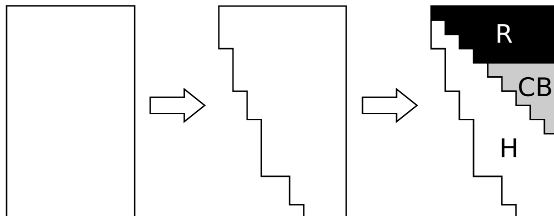

Fig. 2. Techniques to reduce the complexity of the multifrontal QR factorization.

the overall number of floating point operations is lower since frontal matrices are smaller. This is due to the fact that contribution blocks resulting from the complete factorization of frontal matrices are smaller.

A detailed presentation of the multifrontal QR method, including the optimization techniques described above, can be found in Amestoy et al. [2].

The multifrontal method can achieve very high efficiency on modern computing systems because all the computations are arranged as operations on dense matrices; this reduces the use of indirect addressing and allows the use of efficient Level-3 BLAS routines which can achieve a considerable fraction of the peak performance of modern computing systems.

The factorization of a sparse matrix is commonly preceded by a preprocessing phase, commonly referred to as the *analysis phase*, where a number of (mostly symbolic) operations are performed on the matrix such as row and column permutations to reduce the amount of fill-in, the determination of the elimination tree or the symbolic factorization to estimate the amount of memory needed during the factorization phase.

In the reminder of this paper, we will assume that the analysis phase is already performed, and thus we will only focus on the factorization; specifically, we will assume that a fill-reducing permutation of the input matrix and the corresponding assembly tree have been computed.

3 Thread-Level Parallelism

Sparse computations are well known for being hard to parallelize on shared-memory, multicore systems. This is due to the fact that the efficiency of many sparse operations, such as the sparse matrix-vector product, is limited by the speed of the memory system. This is not the case for the multifrontal method; since computations are performed as operations on dense matrices, a *surface-to-volume* ratio between memory accesses and computations can be achieved which reduces the utilization of the memory system and opens opportunities for multithreaded, parallel execution.

In a multifrontal factorization, parallelism is exploited at two levels:

– tree-level parallelism: computations related to separate branches of the assembly tree are independent and can be executed in parallel;

– node-level parallelism: if the size of a frontal matrix is big enough, its partial factorization can be performed in parallel by multiple threads.

3.1 The classical approach

The classical approach to shared-memory parallelization of QR multifrontal solvers (see [14, 2, 7]) is based on a complete separation of the two sources of concurrency described above. The node parallelism is delegated to multithreaded BLAS libraries and only the tree parallelism is handled at the level of the multifrontal factorization. This is commonly achieved by means of a task queue where a task corresponds to the assembly and factorization of a front. A new task is pushed into the queue as soon as it is ready to be executed, i.e., as soon as all the tasks associated with its children have been treated. Threads keep polling the queue for tasks to perform until all the nodes of the tree have been processed.

Although this approach works reasonably well for a limited number of cores or processors, it suffers scalability problems mostly due to two factors:

– separation of tree and node parallelism: the degree of concurrency in both types of parallelism changes during the bottom-up traversal of the tree; fronts are relatively small at leaf nodes of the assembly tree and grow bigger towards the root node. On the contrary, tree parallelism provides a high level of concurrency at the bottom of the tree and only a little at the top part where the tree shrinks towards the root node. Since the node parallelism is delegated to an external multithreaded BLAS library, the number of threads dedicated to node parallelism and to tree parallelism has to be fixed before the execution of the factorization. Thus, a thread configuration that may be optimal for the bottom part of the tree will result in a poor parallelization of the top part and vice-versa.
– synchronizations: the assembly of a front is an atomic operation. This inevitably introduces synchronizations that limit the concurrency level in the multifrontal factorization.

3.2 A new, fine-grained approach

The limitations of the classical approach discussed above can be overcome by employing a different parallelization technique based on fine granularity partitioning of operations combined with a data-flow model for the scheduling of tasks. This approach was already applied to dense matrix factorizations [4] and extended to the supernodal Cholesky factorization of sparse matrices [11].

In order to handle both tree and node parallelism in the same framework, a block-column partitioning of the fronts is applied and three elementary operations defined:

1. `panel`: this operation amounts to computing the QR factorization of a block-column;

2. `update`: updating a block-column with respect to a panel corresponds to applying to the block-column the Householder reflections resulting from the panel reduction;
3. `assemble`: assembles a block-column into the parent node (if it exists);

The multifrontal factorization of a sparse matrix can thus be defined as a sequence of tasks, each task corresponding to the execution of an elementary operation of the type described above on a block-column. The tasks are arranged in a Direct Acyclic Graph (DAG); the edges of the DAG define the dependencies among tasks and thus the order in which they have to be executed. These dependencies are defined according to the following rules:

- a block-column is fully assembled when all the corresponding portions of the contribution blocks from its children have been assembled into it. Once a block-column is fully assembled, any elementary operation can be performed on it (according to the other dependencies) even if the rest of the front is not yet assembled or if the factorization of its children is not completed;
- a panel factorization can be executed on a fully assembled block-column if the block-column is up-to-date with respect to all the previous panel factorizations in the same front;
- a fully assembled block-column can be updated with respect to panel i in its front if it is up-to-date with respect to all the panels $1, ..., i-1$ in the same front and if the panel factorization on block-column i has completed;
- a block-column can be assembled into the parent (if it exists) when it is up-to-date with respect to the last panel factorization to be performed on the front it belongs to.

Figure 3 shows the DAG associated with the problem in Figure 1 for the case where the block-columns have size one. The dashed boxes surround all the tasks that are related to a single front and the horizontal displacement of a task identifies the index, within the front, of the column on which the task is executed. In the figure and in the above discussion, the assembly of the matrix nonzero entries into the frontal matrices has been ignored for the sake of readability.

This DAG globally retains the structure of the assembly tree but expresses a higher degree of concurrency because tasks are defined on a block-column basis instead of a front basis. This allows us to handle both tree and node parallelism in a consistent way.

The execution of the tasks in the DAG is controlled by a data-flow model; a task is dynamically scheduled for execution as soon as all the input operands are available to it, i.e., when all the tasks on which it depends have finished. The scheduling of tasks can be guided by a set of rules that prioritize the execution of a task based on, for example,

- cache awareness: in order to maximize the reuse of data into cache memories, tasks may be assigned to threads based on a locality policy (see [11]);
- fan-out: the fan-out of a task in the DAG defines the number of other tasks that depend on it. Thus, tasks with a higher fan-out should acquire higher

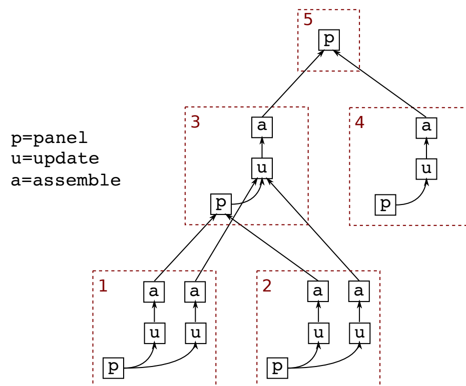

Fig. 3. The DAG associated to the problem in Figure 1.

priority since they generate more concurrency. In the case of the QR method described above, panel factorizations are regarded as higher priority operations over the updates and assemblies.

4 Experimental results

The method discussed in Section 3.2 was implemented in a software package referred to as `qrm` below. The code is written in Fortran95 and OpenMP is the technology chosen to implement the multithreading. Although there are many other technologies for multithreaded programming (e.g., pThreads, Intel TBB, Cilk or SMPSS), OpenMP offers the best portability since it is available on any relatively recent system. The current version of the code does not include cache-aware scheduling of tasks. The `qrm` code was compared to the SuiteSparseQR [7] (referred to as `spqr`) released by Tim Davis in 2009. For both packages, the COLAMD matrix permutation was applied in the analysis phase to reduce the fill-in and equivalent choices were made for other parameters related to matrix preprocessing (e.g., nodes amalgamation); as a result, the assembly trees produced by the two packages only present negligible differences. Both packages are based on the same variant of the multifrontal method (that includes the two optimization techniques discussed in Section 2) and, thus, the number of floating point operations done in the factorization and the number of entries in the resulting factors are comparable. The size of block-columns in `qrm` and the blocking size (in the classical LAPACK sense) in `spqr` were chosen to be the best for each matrix. The rank-revealing feature of `spqr` was disabled as it is not present in `qrm`.

The two packages were tested on a set of ten matrices with different characteristics from the UF Sparse Matrix Collection [5]; in this section, only results related to the matrices listed in Table 1 are presented as they are representative of the general behavior of the `qrm` and `spqr` codes measured on the whole test

Mat. name	m	n	nz	nz(R)	nz(H)	Gflops
Rucci1	1977885	109900	7791168	184115313	1967908664	12340
ASIC_100ks	99190	99190	578890	110952162	53306532	729
ohne2	181343	181343	6869939	574051931	296067596	3600
mk11-b4	10395	17325	51975	21534913	42887317	396
route	20894	43019	206782	3267449	7998419	2.4

Table 1. Test matrices. nz(R), nz(H) and Gflops result from the `qrm` factorization.

set. In the case of underdetermined systems, the transposed matrix is factorized, as it is commonly done to find the minimum-norm solution of a problem.

Experiments were run on two architectures whose features are listed in Table 2.

Type	# of cores	freq.	mem. type	compilers	BLAS/LAPACK
Intel Xeon	8 (4-cores × 2-sockets)	2.8 GHz	UMA	Intel 11.1	Intel MKL 10.2
AMD Opteron	24 (6-cores × 4-sockets)	2.4 GHz	NUMA	Intel 11.1	Intel MKL 10.2

Table 2. Test architectures.

Figure 4 shows the speedup achieved by the `qrm` code for the factorization of the Rucci1 matrix on both test architectures compared to the `spqr` code; the curves plot the results in Tables 3 and 4 normalized to the sequential execution time.

On the Intel Xeon platform (Figure 4, left), a remarkable 6.9 speedup is achieved on eight cores which is extremely close to the value obtained by the LAPACK `dgeqrf` dense factorization routine; the `spqr` code only achieves a 3.88 speedup using eight cores on the Intel Xeon system.

On the AMD Opteron system (Figure 4, right), the `qrm` code still shows a good speedup when compared to `spqr` and `dgeqrf` although it must be noted that all of them exhibit some scalability limits; this is most likely caused by poor data locality due to the NUMA architecture of the memory subsystem. An ongoing research activity aims at investigating cache-aware task scheduling policies that may mitigate this problem.

Figure 5 shows the fraction of the `dgemm` matrix multiply routine performance that is achieved by the `qrm` and `spqr` factorizations.

Tables 3 and 4 show the factorization times for the test matrices on the two reference architectures. Analysis times are also reported for `qrm` in parentheses.

The number of threads participating in the factorization in the `spqr` code is given by the product of the number of threads that exploit the tree parallelism times the number of threads in the BLAS routines. As discussed in Section 3.1,

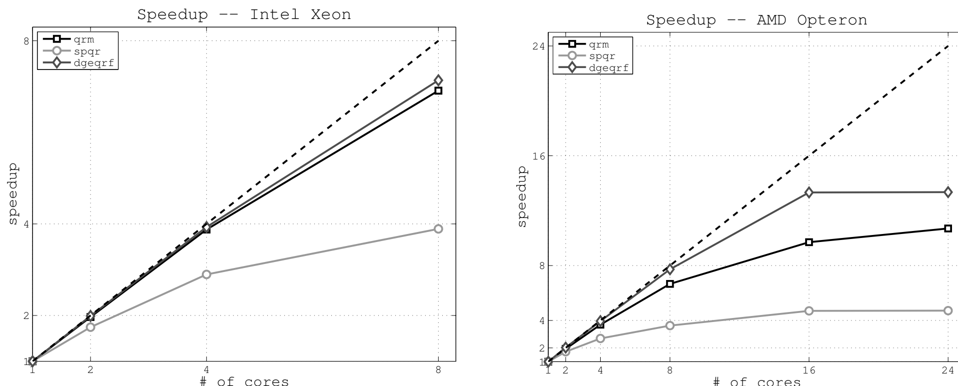

Fig. 4. Speedup for the **qrm** and **spqr** multifrontal factorization of the Rucci1 matrix compared to the LAPACK dense **dgeqrf** factorization routine. The dashed lines represent linear speedup.

this rigid partitioning of threads may result in suboptimal performance; choosing a total number of threads that is higher than the number of cores available on the system may yield a better compromise. This obviously does not provide any benefit to **qrm**. The last line in Tables 3 and 4 shows, for **spqr**, the factorization times for the best combination of tree and node parallelism; for example, for the ohne2 matrix, on the Intel Xeon system, the shortest factorization time is achieved by allocating five threads to the tree parallelism and three to the BLAS parallelism for a total of 15 threads.

The experimental results show that the proposed approach described in Section 3.2 achieves better scalability and better overall execution times on modern,

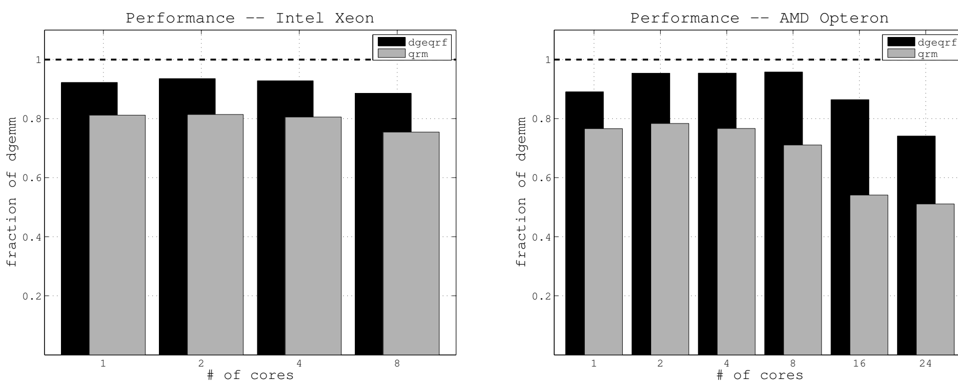

Fig. 5. Performance for the **qrm** multifrontal factorization of the Rucci1 matrix and the LAPACK **dgeqrf** dense matrix factorization routine compared to the BLAS **dgemm** dense matrix product routine.

Intel Xeon

	# th.	Rucci1	ASIC_100ks	ohne2	mk11-b4	route
qrm	1	1237.4 (3.0)	81.7 (0.5)	427.7 (6.6)	41.3 (0.1)	0.88 (0.1)
	2	629.5	41.9	218.3	21.2	0.54
	4	319.8	21.7	110.8	11.2	0.33
	8	179.4	12.4	60.8	6.6	0.21
spqr	# th.	Rucci1	ASIC_100ks	ohne2	mk11-b4	route
	1	1245.2	84.8	449.3	42.3	0.85
	2	714.9	50.3	271.3	24.4	0.53
	4	430.0	32.3	161.0	15.0	0.34
	8	320.7	25.0	111.9	10.8	0.31
	best	295.5	22.2	104.4	10.8	0.29

Table 3. Factorization times, in seconds, on the Intel Xeon system for `qrm` (*top*) and `spqr` (*bottom*). Analysis times are reported in parentheses for `qrm`.

AMD Opteron

	# th.	Rucci1	ASIC_100ks	ohne2	mk11-b4	route
qrm	1	1873.8 (2.5)	125.9 (0.3)	664.8 (4.1)	66.7 (0.1)	1.33 (0.1)
	2	969.0	64.7	338.8	34.8	0.76
	4	507.1	33.8	175.7	18.5	0.45
	8	281.7	18.3	92.2	11.4	0.31
	16	193.7	12.7	55.7	10.5	0.61
	24	175.4	12.2	46.0	9.9	0.97
spqr	# th.	Rucci1	ASIC_100ks	ohne2	mk11-b4	route
	1	2081.1	134.4	712.8	65.8	1.15
	2	1206.8	83.1	428.2	38.8	0.63
	4	773.2	54.2	279.4	25.4	0.37
	8	574.1	40.2	178.8	17.8	0.26
	16	443.4	31.1	138.0	17.0	0.21
	24	390.1	28.0	108.4	16.5	0.24
	best	379.5	26.5	107.1	16.5	0.21

Table 4. Factorization times, in seconds, on the AMD Opteron system for `qrm` (*top*) and `spqr` (*bottom*). Analysis times are reported in parentheses for `qrm`.

multicore-based systems when compared to the classical parallelization strategy implemented in the `spqr` software. On the AMD Opteron architecture, the `qrm` code has consistently higher factorization times than `spqr` and a poor scaling for the *route* matrix: this is exclusively due to flaws in the implementation of the tasks scheduler and are not related to the proposed parallelization approach. The `qrm` tasks scheduler is currently undergoing a complete rewriting that aims at improving its efficiency by reducing the search space in the tasks DAG.

Acknowledgments

I would like to thank the MUMPS team and, particularly, Chiara Puglisi and Patrick Amestoy for their precious help and support.

References

1. P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. MUMPS: a general purpose distributed memory sparse solver. In A. H. Gebremedhin, F. Manne, R. Moe, and T. Sørevik, editors, *Proceedings of PARA2000, the Fifth International Workshop on Applied Parallel Computing, Bergen, June 18-21*, pages 122–131. Springer-Verlag, 2000. Lecture Notes in Computer Science 1947.
2. P. R. Amestoy, I. S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Int. Journal of Num. Linear Alg. and Appl.*, 3(4):275–300, 1996.
3. Å. Björck. *Numerical methods for Least Squares Problems.* SIAM, Philadelphia, 1996.
4. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, 2009.
5. T. A. Davis. University of Florida sparse matrix collection, 2002. http://www.cise.ufl.edu/research/sparse/matrices.
6. T. A. Davis. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.
7. T. A. Davis. Multifrontal multithreaded rank-revealing sparse QR factorization. *Submitted to ACM Transactions on Mathematical Software*, 2009.
8. I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
9. A. George and J. W. H. Liu. Householder reflections versus Givens rotations in sparse orthogonal decomposition. *Linear Algebra and its Applications*, 88/89:223–238, 1987.
10. J. A. George and M. T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and its Applications*, 34:69–83, 1980.
11. J. Hogg, J. K. Reid, and J. A. Scott. A DAG-based sparse Cholesky solver for multicore architectures. Technical Report RAL-TR-2009-004, Rutherford Appleton Laboratory, 2009.
12. J. W. H. Liu. On general row merging schemes for sparse Givens transformations. *SIAM J. Sci. Stat. Comput.*, 7:1190–1211, 1986.
13. J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.
14. P. Matstoms. Parallel sparse QR factorization on shared memory architectures. Technical Report LiTH-MAT-R-1993-18, Department of Mathematics, 1993.
15. J. R. Rice. PARVEC workshop on very large least squares problems and supercomputers. Technical Report CSD-TR 464, Purdue University, IN., 1983.
16. R. Schreiber and C. Van Loan. A storage-efficient WY representation for products of householder transformations. *SIAM J. Sci. Stat. Comput.*, 10:52–57, 1989.
17. J. Silc, B. Robic, and T. Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. *Journal of Parallel and Distributed Computing Practices*, 1:1–33, 1998.