

# Fine-grained multithreading for the multifrontal $QR$ factorization of sparse matrices

ALFREDO BUTTARI \*

August 30, 2011

## 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 approach to the parallelization of the multifrontal method for the  $QR$  factorization of sparse matrices specifically designed for multicore based systems. High efficiency is achieved through a fine-grained partitioning of data and a dynamic scheduling of computational tasks relying on a dataflow parallel programming model. Experimental results show that an implementation of the proposed approach achieves higher performance and better scalability than existing equivalent software.

## 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 [21, 4] 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 [22]), 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 [12], 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 factorization of sparse, indefinite, symmetric matrices [10] and then extended to the  $QR$  factorization [16, 11], 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 [19, 3, 9]; all of them are based on the same approach to parallelization which suffers scalability limits on modern, multicore systems (see Section 3).

---

\*CNRS-IRIT, ENSEEIHT, 2 rue Charles Camichel, 31071 Toulouse, France

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. The proposed method leverages a fine-grained partitioning of computational tasks and a dataflow execution model [23] 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 [10] 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 [2] and UMF-PACK [8]. At the heart of this method is the concept of an *elimination tree*, extensively studied and formalized later by Liu [18]. 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 fact that the  $R$  factor of a matrix  $A$  and the Cholesky factor of the normal equation matrix  $A^T A$  share the same structure under the hypothesis that the matrix  $A$  is *Strong Hall* (for a definition of this property see, for example, [4]). 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 the case where the Strong Hall property does not hold, the elimination tree related to the Cholesky factorization of  $A^T A$  can still be used although the resulting  $QR$  factorization will perform more computations and consume more memory than what is really needed; alternatively, the matrix  $A$  can be permuted to a Block Triangular Form (BTF) where all the diagonal blocks are Strong Hall.

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 dense 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 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; the matrix that stores the coefficients of the computed Householder vectors, will be referred to as the  $H$  matrix from now on.

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 in the  $R$  factor. The elimination of amalgamated pivots and the consequent update of the trailing frontal submatrix can thus be performed by means of efficient Level-3 BLAS routines. Moreover, amalgamation reduces the number of assembly operations increasing the computations-to-communications ratio which results in better performance. The amalgamated elimination tree is also commonly referred to as *assembly tree*.

Figure 1 shows the  $R$  factor and the elimination/assembly tree, in the top-right and bottom-right parts, respectively, corresponding to the  $A$  matrix in the left part<sup>1</sup>; the dots in the  $A$  matrix represent the fill-in generated during the multifrontal factorization while the a coefficients represent the original data. The supernodes of the assembly tree are obtained by amalgamating the pivots

---

<sup>1</sup>The rows of  $A$  are sorted in order of increasing index of the leftmost nonzero in order to show more clearly the computational pattern of the method on the input data.

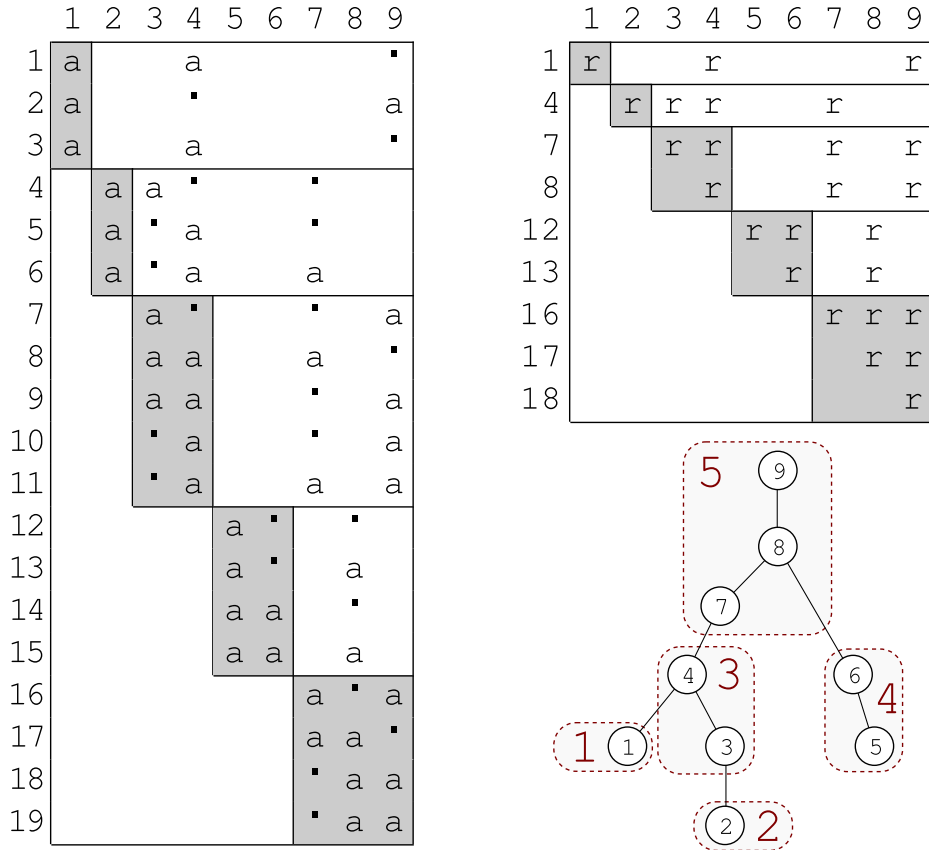


Figure 1: Example of multifrontal  $QR$  factorization. The dots denote the fill-in coefficients.

within the same shaded box. It has to be noted that in practical implementations the amalgamation procedure is based only on information related to the  $R$  factor and, as such, it does not take into account fill-in that may eventually appear in the  $H$  matrix; for example, the amalgamation of the pivots 3 and 4 generates the  $h_{10,3}$  and  $h_{11,3}$  fill-in coefficients although no extra fill-in appears in the  $R$  factor.

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. The number of operations can thus be reduced, as well as the fill-in in the  $H$  matrix, by ignoring the zeroes in the bottom-left part of the frontal matrix;
2. the frontal matrix is completely factorized. Despite the fact that more Householder vectors have to be computed for each frontal matrix, 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.

Fig. 2 shows the assembly and factorization operations on the supernode 3 in Fig. 1 when these optimization techniques (referred to as *Strategy 3* in [3]) are applied. The contribution blocks resulting from the factorization of supernodes 1 and 2 are appended to the rows of the input  $A$  matrix associated with supernode 3 in such a way that the resulting, assembled, frontal matrix has the *staircase* structure shown in Fig. 2 (*middle*). Once the front is assembled, it is factorized as shown in Fig. 2 (*right*).

A detailed presentation of the multifrontal  $QR$  method, including the optimization techniques described above, can be found in [3].

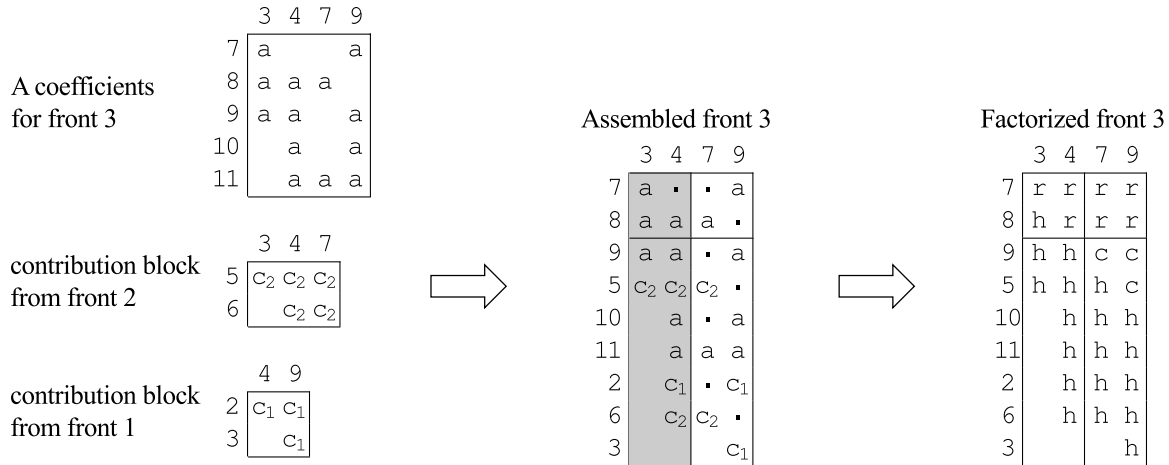


Figure 2: Assembly and factorization of the frontal matrix associated with supernode 3 in Fig. 1.

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 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.

The rest of this paper is based on the assumption that the analysis phase is already performed, and thus it only focuses on the factorization; specifically, it is assumed that a fill-reducing permutation of the input matrix and the corresponding assembly tree have been computed.

### 3 Fine-grained, asynchronous multithreading

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 (see Proposition 10.1 in [18]);
- node-level parallelism: if the size of a frontal matrix is big enough, its factorization can be performed in parallel by multiple threads.

The classical approach to shared-memory parallelization of  $QR$  multifrontal solvers (see [19, 3, 9]) 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. Although some recent parallel BLAS libraries allow to change the numbers of threads dynamically at run-time, it would require an accurate performance modeling and a rigid thread-to-front mapping in order to keep all the cores working at any time. Relying on some specific BLAS library could, moreover, limit the portability of a code.
- **synchronizations:** the assembly of a front is an atomic operation. This inevitably introduces synchronizations that limit the concurrency level in the multifrontal factorization; most importantly, it is not possible to start working on a front until all of its children have been fully treated.

The limitations of the classical approach discussed above can be overcome by employing a different parallelization technique based on fine granularity partitioning of data and operations combined with a dataflow model for the scheduling of tasks, as described in the following subsections. This approach was already applied to dense matrix factorizations [6] and extended to the supernodal Cholesky factorization of sparse matrices [15].

### 3.1 Fine-grained computational tasks definition

In order to handle both tree and node parallelism in the same framework, a block-column partitioning of the fronts is applied as shown in Fig. 3 (*left*) and five elementary operations defined:

1. **activate:** the activation of a frontal matrix corresponds to computing its structure (row/column indices, staircase structure, etc.) and allocating the memory needed for it;
2. **panel:** this operation amounts to computing the  $QR$  factorization of a block-column; Fig. 3 (*middle*) shows the data modified when the **panel** operation is executed on the first block-column;
3. **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; Fig. 3 (*right*) shows the coefficients read and modified when the third block-column is **update**'d with respect to the first panel;
4. **assemble:** for a block-column, assembles the corresponding part of the contribution block into the parent node (if it exists);
5. **clean:** stores the coefficients of the  $R$  and  $H$  factors aside and deallocates the memory needed for the frontal matrix storage;

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 or a front. The tasks are arranged in a Directed Acyclic Graph (DAG) such that the edges of the DAG define the dependencies among tasks and thus the relative order in which they have to be executed. Figure 4 shows the DAG associated with the subtree defined by supernodes

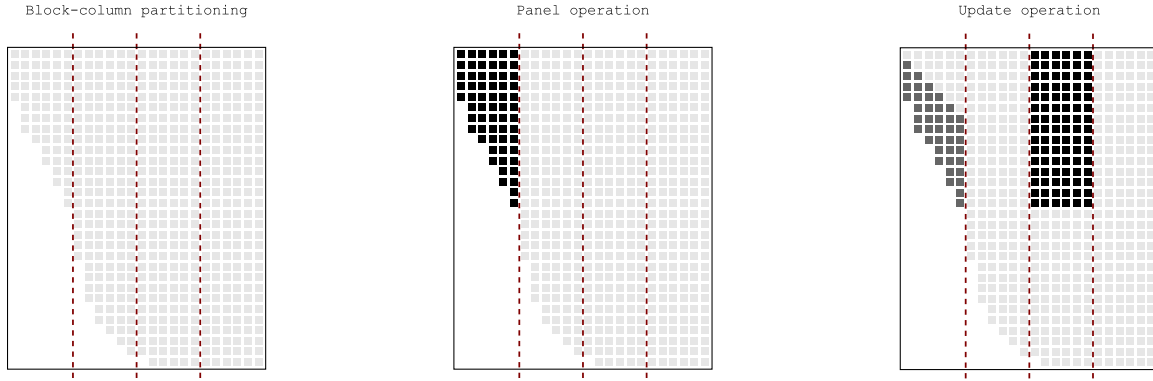


Figure 3: Block-column partitioning of a frontal matrix (*left*) and panel and update operations pattern (*middle* and *right*, respectively); dark gray coefficients represent data read by an operation while black coefficients represent written data.

one, two and three for the problem in Figure 1 for the case where the block-columns have size one<sup>2</sup>; the dashed boxes surround all the tasks that are related to a single front.

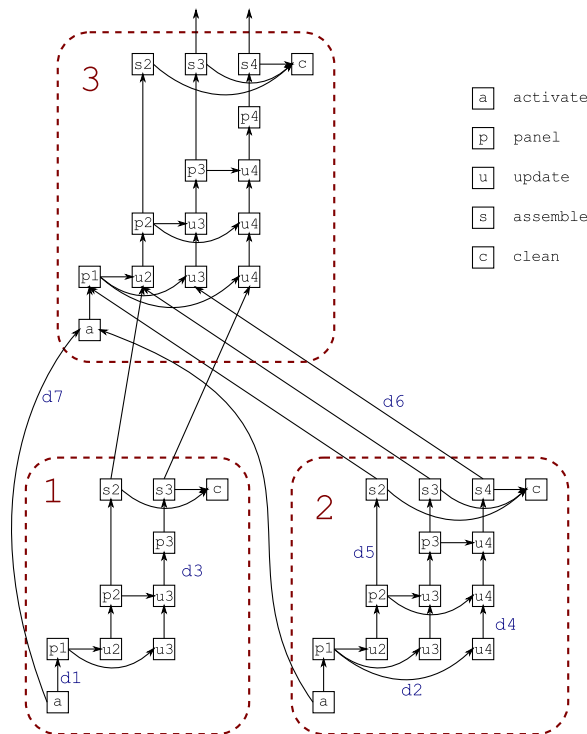


Figure 4: The DAG associated with supernodes 1, 2 and 3 the problem in Figure 1; for the **panel**, **update** and **assemble** operations, the corresponding block-column index is specified. For this example, the block-column size is chosen to be one.

The dependencies in the DAG are defined according to the following rules (an example of each of these rules is presented in Figure 4 with labels on the edges):

- **d1**: no other elementary operation can be executed on a front or on one of its block-columns

<sup>2</sup>Figure 4 actually shows the transitive reduction of the DAG, i.e., the direct dependency between two nodes is not shown in the case where it can be represented implicitly by a path of length greater than one connecting them.

until the front is not activated;

- **d2:** a block column can be updated with respect to a panel only if the corresponding panel factorization is completed;
- **d3:** the `panel` operation can be executed on block-column  $i$  only if it is up-to-date with respect to panel  $i - 1$ ;
- **d4:** a block-column can be updated with respect to a panel  $i$  in its front only if it is up-to-date with respect to the previous panel  $i - 1$  in the same front;
- **d5:** 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 (in this case it is assumed that block-column  $i$  is up-to-date with respect to panel  $i$  when the corresponding `panel` operation is executed);
- **d6:** no other elementary operation can be executed on a block-column until all the corresponding portions of the contribution blocks from the child nodes have been assembled into it, in which case the block-column is said to be *assembled*;
- **d7:** since the structure of a frontal matrix depends on the structure of its children, a front can be activated only if all of its children are already active;

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 makes it possible to handle both tree and node parallelism in a consistent way. For example, it is possible to start working on the assembled block-columns of a front even if the rest of the front is not yet assembled and, most importantly, even if the children of the front have not yet been completely treated.

### 3.2 Scheduling and execution of tasks

The execution of the tasks in the DAG is controlled by a dataflow 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,

- **data locality:** in order to maximize the reuse of data into the different level of the memory hierarchy, tasks may be assigned to threads based on a locality policy (see [15]);
- **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 priority since their execution generates more concurrency. In the case of the *QR* method described above, panel factorizations are regarded as higher priority operations over the updates and assemblies.

A scheduling technique was developed aiming at optimizing the reuse of local data in a NUMA system while still prioritizing tasks that have a high fan-out. The method is based on a concept of ownership of a front: the thread that performs the `activate` operation on a front becomes its owner and, therefore, becomes the privileged thread to perform all the subsequent tasks related to that front. By using methods like the “first touch rule” (memory is placed on the NUMA node which generates the first reference) or allocation routines which are specific for NUMA architectures [5], the memory needed for a front can be allocated in the NUMA node which is closest to its owner thread.

At the moment no front-to-thread mapping is performed, and thus the ownership of a front is dynamically set at the moment when the front is activated; experimental results in Section 4 show that such a mapping could provide a better exploitation of the memory system and will be the object of future work.

The scheduling and execution of tasks is implemented through a system of task queues: each thread is associated with a task queue containing all the executable tasks related to the fronts it owns.

The pseudo-code in Figure 5 (*left*) illustrates the main loop executed by all threads; at each iteration of this loop a thread:

1. checks whether the number of tasks globally available for execution has fallen below a certain value (which depends, e.g., on the number of threads) and, if it is the case, it calls the `fill_queues` routine, described below, which searches for ready tasks and pushes them into the corresponding local queues;
2. picks a task. This operation consists in popping a task from the thread's local queue. In the case where no task is available on the local queue, an architecture aware work-stealing technique is employed, i.e., the thread will try to steal a task from queues associated with threads with which it shares some level of memory (caches or DRAM module on a NUMA machine) and if still no task is found it will attempt to steal a task from any other queue. The computer's architecture can be detected using tools such as `hwloc` [5].
3. executes the selected task if the `pick_task` routine has succeeded.

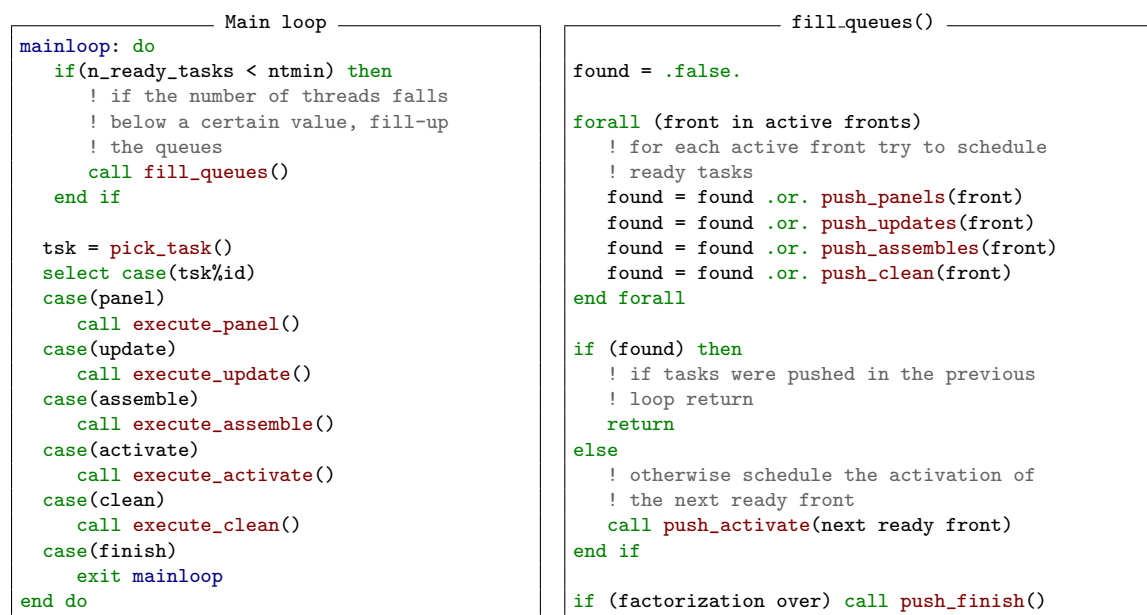


Figure 5: Pseudo-code showing, on the left, the main execution loop and, on the right, the instructions used to fill the task queues.

The tasks are pushed into the local queues by the `fill_queues` routine whose pseudo-code is shown in Figure 5 (*right*). At every moment, during the factorization there exists a list of active fronts; the `fill_queues` routine goes through this list looking for ready tasks on each front. Whenever one such task is found, it is pushed on the queue associated with the thread that owns the front. If no task is found related to any of the active fronts, a new ready front (if any) is scheduled for activation; because the owner of a ready front is not yet defined, the `activate` task is pushed on the queue attached to the thread that executes the `fill_queues` routine. Simultaneous access to the same front in the `fill_queues` routine is prevented through the use of locks. It is important to note that, as long as tasks are available on already active nodes, no other node is activated which avoids an excessive growth of the consumed memory and keeps the nodes traversal order as close as possible to a postorder.

Although tasks are always popped from the head of each queue, they can be pushed either on the head or on the tail which allows to prioritize certain tasks. In the implementation described in

Section 4, the `panel` operations are always pushed on the head because the corresponding nodes in the execution DAG have higher fan-out.

The size of the search space for the `fill_queues` routine is, thus, proportional to the number of fronts active, at a given moment, during the factorization. The size of this search space may become excessively large and, consequently, the relative cost of the `fill_queues` routine excessively high, in some cases like, for example, when the assembly tree is very large and/or when it has many nodes of small size. Two techniques are employed in order to keep the number of active nodes limited during the factorization:

- **Logical pruning.** As the target of this work are systems with only a limited number of cores, extremely large assembly trees provide much more tree-level parallelism than it is really needed. A logical pruning can be thus applied to simplify the tree: all the subtrees whose relative computational weight is smaller than a certain threshold are made invisible to the `fill_queues` routine. When one of the remaining nodes is activated, all the small subtrees attached to it are processed sequentially by the same thread that performs the activation. As shown in Figure 6, this corresponds to identifying a layer in the assembly tree such that all the subtrees below it will be processed sequentially. This layer has to be as high as possible in order to reduce the number of potentially active nodes but low enough to provide a sufficient amount of tree-level parallelism on the top part of the tree.

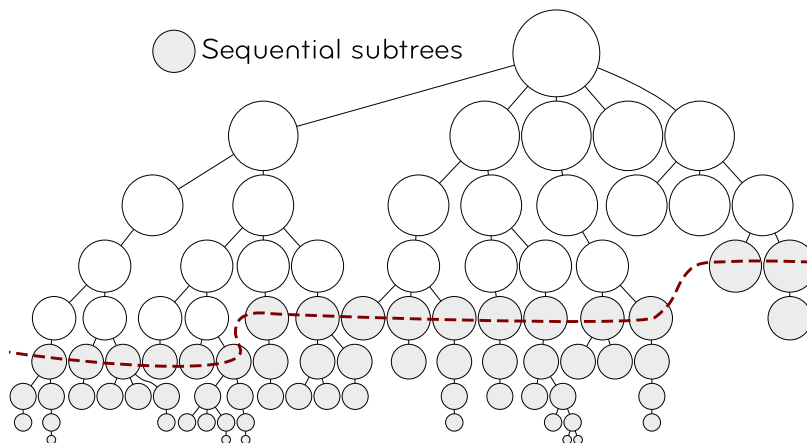


Figure 6: A graphical representation of how the logical amalgamation and logical pruning may be applied to an assembly tree.

- **Tree reordering.** Assuming that the assembly tree is processed sequentially following a postorder traversal, the maximum number of fronts active at any time in the subtree rooted at node  $i$ ,  $P_i$  is defined as the maximum of two quantities:
  1.  $nc_i + 1$ , where  $nc_i$  is the number of children of node  $i$ . This is the number of active nodes at the moment when  $i$  is activated;
  2.  $\max_{j=1, \dots, nc_i} (j - 1 + P_j)$ . This quantity captures the maximum number of active fronts when the children of node  $i$  are being treated. In fact, at the moment when the peak  $P_j$  is reached in the subtree rooted at the  $j$ -th child, all the previous  $j - 1$  children of  $i$  are still active.

In order to minimize the maximum number of active nodes during the traversal of the assembly tree it is, thus, necessary to minimize, for each node  $i$ , the second quantity, which is achieved by sorting all of its children  $j$  in decreasing order of  $P_j$  [17]. The effect of this reordering on an example tree is illustrated in Figure 7. If the tree is traversed in the order shown in the left part of the figure, the maximum number of active nodes  $P_{19} = 10$  (the nodes active at the moment when the peak is reached are highlighted with a thick border); instead, if the tree is

| Mat. name  | Mat. n. | m       | n      | nz      |
|------------|---------|---------|--------|---------|
| mk11-b4    | 1       | 10395   | 17325  | 51975   |
| scagr7-2r  | 2       | 32847   | 46679  | 120141  |
| route      | 3       | 20894   | 43019  | 206782  |
| deltaX     | 4       | 68600   | 21961  | 247424  |
| lp_nug20   | 5       | 15240   | 72600  | 304800  |
| ASIC_100ks | 6       | 99190   | 99190  | 578890  |
| TF17       | 7       | 38132   | 48630  | 586218  |
| sls        | 8       | 1748122 | 62729  | 6804304 |
| ohne2      | 9       | 181343  | 181343 | 6869939 |
| Rucci1     | 10      | 1977885 | 109900 | 7791168 |

Table 1: Test matrices.

reordered as in the right part of the figure following the method described above  $P_{19}$  is equal to four. Although no guarantee is given that a postorder is followed in a parallel factorization, this tree reordering technique still provides excellent results on every problem that has been tested so far. Besides, by reducing the number of active nodes, this reordering also helps in reducing the consumed memory although it will not be optimal in this sense as the actual size of the frontal matrices is not taken into account (the reordering technique for memory consumption minimization is described in [17, 13]).

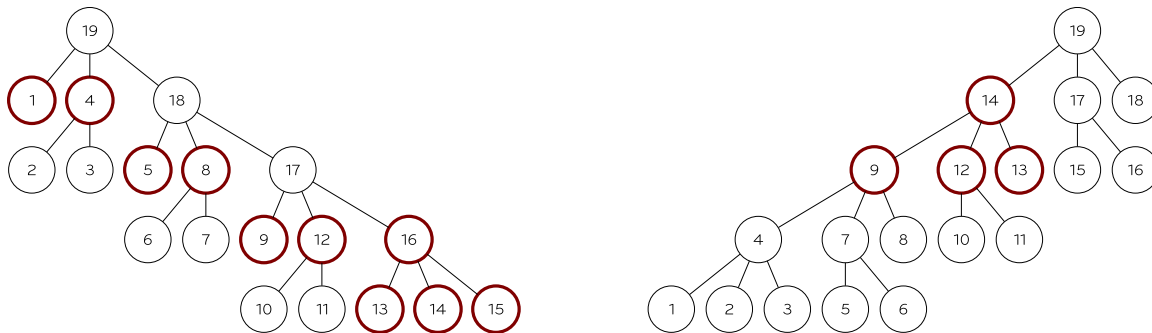


Figure 7: The effect of leaves reordering on the number of active nodes.

## 4 Experimental results

The method discussed in Section 3 was implemented in a software package called `qr_mumps` (or `qrm` for the sake of brevity). 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 experiments were run for a set of ten matrices from the UF Sparse Matrix Collection [7] with different characteristics shown in Table 1. 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:

- **AMD Istanbul:** this system is equipped with four hexa-core AMD Istanbul processors clocked at 2.4 GHz. Each of these CPUs has six cores and is attached to a DRAM module through two DRAM controllers and the CPUs are connected to each other through HyperTransport links in a ring layout. If, say, core 0 on processor 1 needs some data which is located in the memory attached to processor 0, a request has to be submitted to the DRAM controllers of

| System           | AMD Istanbul   | IBM Power6 p575 |
|------------------|----------------|-----------------|
| total # of cores | 24 (6×4)       | 32 (2×4×4)      |
| freq.            | 2.4 GHz        | 4.7 GHz         |
| mem. type        | NUMA           | NUMA            |
| compilers        | Intel 11.1     | IBM XL 13.1     |
| BLAS/LAPACK      | Intel MKL 10.2 | IBM ESSL 4.4    |

Table 2: Test architectures.

processor 0 and, when this request is processed, the corresponding data is sent through the HyperTransport link. Concurrent access to the same memory causes conflicts which slow-down the access to data.

- **IBM Power6 p575:** one node of the larger Vargas supercomputer installed at the IDRIS supercomputing institute is equipped with 16 dual-core Power6 processors clocked at 4.7 GHz. Processors are grouped in sets of four called MCMs (Multi Chip Module) and each node has four MCMs for a total of 32 cores. Processors in an MCM are fully connected as well as MCMs in a node.

All the tests were run with real data in double precision.

#### 4.1 Choosing the block size

It is obviously desirable to use blocked operations that rely on Level-3 BLAS routines in order to achieve a better exploitation of the memory hierarchy and, thus, better performance. The use of blocked operations, however, introduces additional fill-in in the Householder vectors due to the fact that the staircase structure of the frontal matrices cannot be fully exploited. It can be safely said that it is always worth paying the extra cost of this additional fill-in because the overall performance will be drastically improved by the high efficiency of Level-3 BLAS routines; nonetheless it is important to choose the blocking value that gives the best compromise between number of operations and efficiency of the BLAS. Moreover this blocking size, which defines the granularity of computations, has to be chosen with respect to the block-columns size used for partitioning the frontal matrices, which defines the granularity of parallel tasks. Figure 8 shows as dark gray dots the extra fill-in introduced by the blocking of operations (denoted as  $ib$  for internal blocking) wrt the partitioning size  $nb$  on an example frontal matrix.

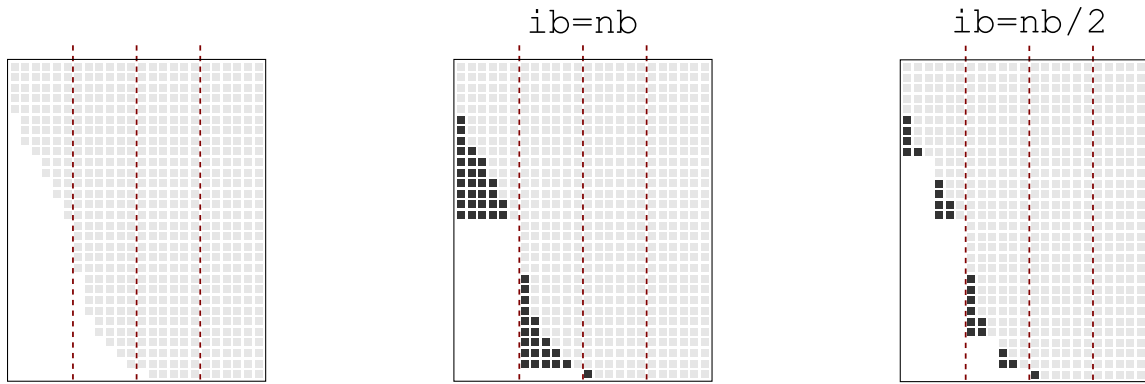


Figure 8: The effect of internal blocking on the generated fill-in. The light gray dots show the frontal matrix structure if no blocking of operations is applied whereas the dark gray dots show the additional fill-in introduced by blocked operations.

Figure 9 shows how the performance is affected by different choices of the two blocking parameters on the Power6 machine for two different matrices. Although the best values clearly depend on the problem and the computing system, experiments conducted on the other test matrices and architecture confirm that any reasonable (according to common knowledge on BLAS libraries) choice for the two parameters falls very close to the optimum. Choosing `nb` to be a multiple of `ib`, generally delivers better results.

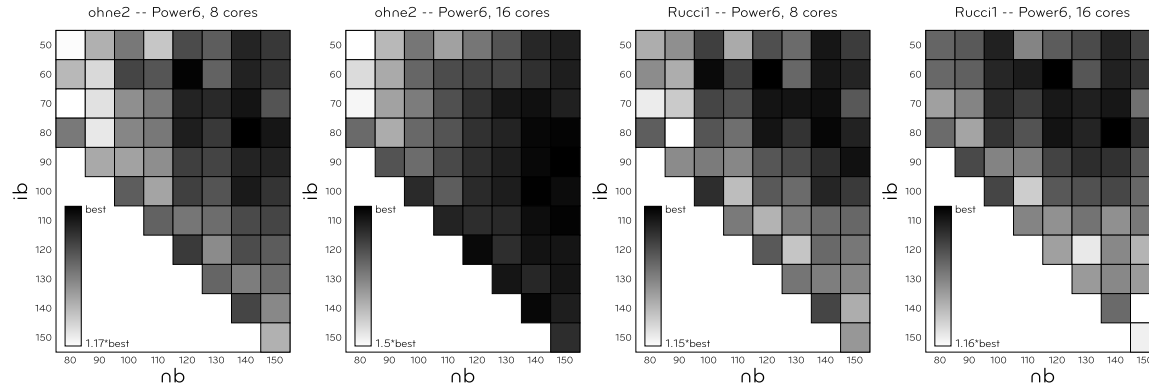


Figure 9: The effect of the blocking sizes on the ohne2 and Rucci1 matrices on the Power6 computer.

All the results presented in the rest of this paper are related to the best choice of both of these parameters value.

## 4.2 Understanding the memory utilization

As described in Section 3.2, the scheduling of tasks in `qrm` is based on a method that aims at maximizing the locality of data in a NUMA environment. The purpose of this section is to provide an analysis of the effectiveness of this approach. This analysis was conducted on the AMD system always using all of the 24 cores available with the PAPI [20, 24] tool. Recalling the architectural characteristics described in Section 4, the efficiency of the scheduling technique has been evaluated based on three metrics:

- the completion time;
- the amount of data transferred on the HyperTransport links. This quantity was measured as the number of occurrences of the PAPI `HYPERTRANSPORT_LINKx:DATA_DWORD_SENT` event (where `x` is 0, 1, 2, 3 since each processor has 4 HyperTransport links) which counts the number of double-words transferred over the HyperTransport links;
- the number of conflicts on the DRAM controllers. This quantity was measured as the number of occurrences of the PAPI `DRAM_ACCESSES_PAGE:DCTx_PAGE_CONFLICT` event (where `x` is 0, 1 since each processor has two DRAM controllers) which counts the number of conflicts occurring on the DRAM controllers.

Experiments were run on three different matrices and with three different settings for the memory policy:

- no locality: in this case each task is pushed on the queue associated with a randomly chosen thread;
- locality: this corresponds to the scheduling strategy described in Section 3.2, i.e., each task is pushed on the queue attached to the thread which owns the related front;

| Matrix   | Strat.  | Time (s) | Dwords on HT<br>( $\times 10^9$ ) | Conflicts on DCT<br>( $\times 10^9$ ) |
|----------|---------|----------|-----------------------------------|---------------------------------------|
| Rucci1   | no loc. | 155      | 2000                              | 23.8                                  |
|          | loc.    | 144      | 1634                              | 25.4                                  |
|          | r. r.   | 117      | 2306                              | 19.7                                  |
| ohne2    | no loc. | 43       | 504                               | 6.63                                  |
|          | loc.    | 39       | 375                               | 6.71                                  |
|          | r. r.   | 38       | 665                               | 4.54                                  |
| lp_nug20 | no loc. | 89       | 879                               | 11.4                                  |
|          | loc.    | 84       | 778                               | 11.9                                  |
|          | r. r.   | 66       | 985                               | 6.81                                  |

Table 3: Memory access behavior.

- round robin: in this case allocated memory pages are distributed in a round robin fashion over all the DRAM modules. This is achieved using the `numactl` [1] tool with the “-i all” option. Since the code does not control the placement of data in the memory system, the ownership of fronts does not make sense anymore and, consequently, the data locality is completely destroyed.

Table 3 shows the results of these experiments. It can be seen that the scheduling strategy described in Section 3.2 provides better execution times with respect to a naive scheduling of tasks; as expected, this can be explained by the reduced amount of data transferred over the HyperTransport links. However, the best execution times are achieved with the round robin distribution of memory allocations. In this case the amount of data transfers is higher than the other two cases since the frontal matrices are randomly distributed over the NUMA nodes; nonetheless, this more even distribution of data reduces the number of conflicts on the DRAM controllers and provides a better exploitation of the memory bandwidth. The considerable performance improvement resulting from the interleaved memory allocation suggest that reducing the memory conflicts may be much more important than minimizing the amount of data transferred over the HyperTransport links. As they are not conflicting objectives, a fronts-to-threads mapping can be defined which aims at maximizing the data locality while reducing the number of memory conflicts; this is the object of ongoing research. This behavior is coherent to what observed on the `HSL_MA87` [15] code which uses a similar approach for the Cholesky factorization of sparse linear systems.

### 4.3 Absolute performance and Scaling

This section shows experimental results aiming at evaluating the efficiency and scaling of the proposed approach to the parallelization of the multifrontal QR method.

The `qrm` code was compared to the SuiteSparseQR [9] (referred to as `spqr`) released by Tim Davis in 2009; because the `spqr` is based on Intel TBB which is only available for x86 systems, this comparison is only made on the AMD Istanbul system. 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. For both codes, runs were executed with `numactl` memory interleaving where available (i.e., only on the AMD Istanbul machine).

Tables 4 and 5 show the factorization times for the test matrices on the two reference architectures. For each architecture, results were measured for number of threads equal to multiples of the number of cores per NUMA node, i.e., six and eight for the AMD system and IBM system, respectively.

| AMD Istanbul |      |      |      |      |       |       |       |      |      |       |      |
|--------------|------|------|------|------|-------|-------|-------|------|------|-------|------|
|              | #    | 1    | 2    | 3    | 4     | 5     | 6     | 7    | 8    | 9     | 10   |
| qrm          | th.  |      |      |      |       |       |       |      |      |       |      |
|              | 1    | 62.8 | 38.3 | 1.27 | 182.0 | 852.6 | 125.4 | 5115 | 4240 | 660.8 | 1878 |
|              | 2    | 33.7 | 19.7 | 0.70 | 96.5  | 450.3 | 65.0  | 2661 | 2175 | 341.9 | 978  |
|              | 4    | 17.2 | 10.9 | 0.42 | 49.0  | 228.3 | 33.1  | 1348 | 1251 | 172.8 | 495  |
|              | 6    | 12.8 | 8.0  | 0.33 | 34.1  | 155.5 | 22.6  | 903  | 909  | 119.1 | 334  |
|              | 12   | 7.7  | 5.2  | 0.23 | 19.1  | 89.6  | 13.4  | 470  | 749  | 62.3  | 179  |
|              | 18   | 6.8  | 4.6  | 0.20 | 14.9  | 71.6  | 9.3   | 329  | 744  | 45.0  | 134  |
|              | 24   | 6.7  | 4.6  | 0.19 | 13.3  | 64.0  | 7.7   | 260  | 746  | 36.5  | 117  |
| spqr         | 1    | 65.8 | 38.1 | 1.15 | 191.2 | 908.9 | 134.4 | 5361 | 4276 | 712.8 | 2081 |
|              | 2    | 38.8 | 22.6 | 0.64 | 115.6 | 522.3 | 83.1  | 2959 | 2978 | 428.2 | 1206 |
|              | 4    | 25.4 | 14.4 | 0.38 | 74.4  | 317.4 | 54.2  | 1659 | 2112 | 279.4 | 773  |
|              | 6    | 19.8 | 11.7 | 0.30 | 59.6  | 246.2 | 44.5  | 1203 | 1845 | 210.0 | 598  |
|              | 12   | 16.9 | 9.6  | 0.29 | 48.4  | 180.4 | 35.2  | 767  | 1644 | 148.1 | 469  |
|              | 18   | 16.8 | 8.6  | 0.24 | 39.6  | 163.4 | 27.7  | 643  | 1611 | 129.9 | 404  |
|              | 24   | 16.5 | 8.5  | 0.26 | 33.0  | 185.0 | 28.0  | 592  | 1539 | 108.4 | 390  |
|              | best | 16.5 | 8.5  | 0.24 | 33.0  | 172.8 | 26.5  | 592  | 1539 | 107.1 | 379  |

Table 4: Factorization times, in seconds, on the AMD Istanbul system for **qrm** (*top*) and **spqr** (*bottom*). The first row shows the matrix number.

| IBM Power6 |     |      |      |      |       |       |      |      |      |       |      |
|------------|-----|------|------|------|-------|-------|------|------|------|-------|------|
|            | #   | 1    | 2    | 3    | 4     | 5     | 6    | 7    | 8    | 9     | 10   |
| qrm        | th. |      |      |      |       |       |      |      |      |       |      |
|            | 1   | 37.6 | 23.9 | 0.94 | 117.3 | 539.7 | 77.6 | 3299 | 3003 | 380.4 | 1152 |
|            | 2   | 18.8 | 11.8 | 0.49 | 58.7  | 276.3 | 38.4 | 1709 | 1547 | 191.2 | 589  |
|            | 4   | 10.0 | 6.5  | 0.33 | 30.7  | 144.7 | 20.1 | 877  | 909  | 100.6 | 307  |
|            | 8   | 6.1  | 3.9  | 0.42 | 17.1  | 77.3  | 10.8 | 451  | 698  | 52.7  | 161  |
|            | 16  | 4.9  | 3.0  | 0.69 | 11.1  | 51.4  | 6.9  | 268  | 773  | 32.3  | 93   |
|            | 24  | 4.9  | 2.9  | 0.93 | 9.8   | 45.8  | 6.1  | 204  | 820  | 32.6  | 71   |
|            | 32  | 5.1  | 2.9  | 1.39 | 9.3   | 44.0  | 6.1  | 175  | 889  | 47.7  | 67   |

Table 5: Factorization times, in seconds, on the Power6 system for **qrm**. The first row shows the matrix number.

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. For each fixed number of threads, the best results among all the possible combinations are reported in Table 4. As discussed in Section 3, 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 Table 4 shows, for `spqr`, the factorization times for the best combination of tree and node parallelism; for example, for the `lp_nug20` (matrix number 5) matrix the shortest factorization time is achieved by allocating 3 threads to the tree parallelism and 12 to the BLAS parallelism for a total of 36 threads.

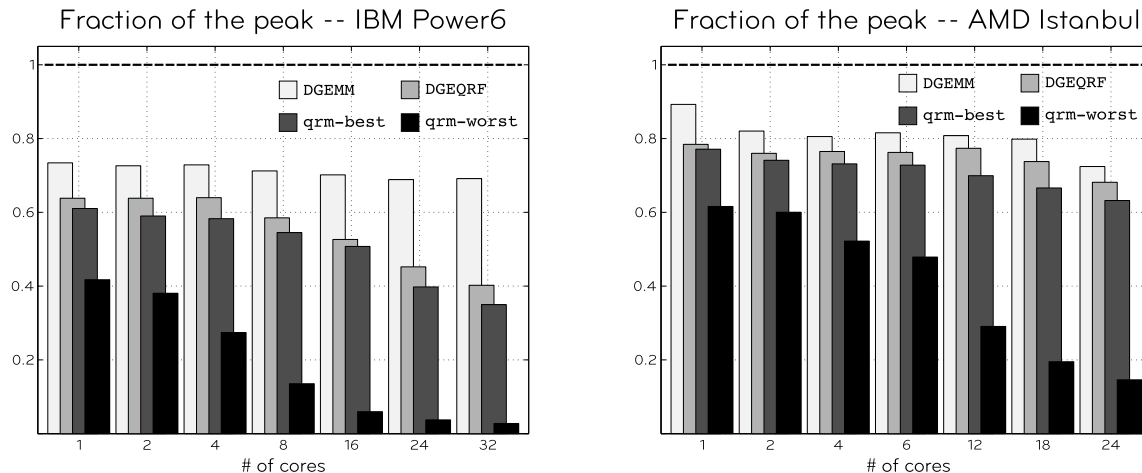


Figure 10: Fraction of the peak performance achieved by `qrm` compared to the QR factorization of a dense matrix.

Figure 10 shows the fraction of the peak performance achieved by `qrm` on the two systems in the best and worst cases which correspond to the TF17 and `sls` (number 7 and 8 in Table 1) matrices, respectively; the best and worst cases have chosen among the matrices that provide a volume of computations big enough to achieve asymptotic performance. The absolute performance and scalability of `qrm` is compared to that of the matrix multiply operation (DGEMM) and the QR factorization of a dense matrix (DGEQRF); square matrices of size big enough to achieve asymptotic performance were chosen for both these reference operations.

The experimental results reported above show that `qrm` achieves a good scalability up to 24 cores, the best speedup being more than 19 for the TF17 matrix on the AMD system.

Table 4 shows that the proposed approach clearly outperforms the classical approach to parallelization of the QR multifrontal method as `qrm` achieves speedups of more than three over `spqr` on the AMD system.

The scalability of the `qrm` code tends to level-off past 24 threads. A closer analysis of the execution profiles shows that this may be due to a lack of parallelism explained by the fact that `panel` operations, which lie on the DAG’s critical path, are extremely inefficient. This problem becomes more evident when frontal matrices tend to be “tall and skinny” (i.e., having many more rows than columns) and thus the relative cost of `panel` operations is even higher; this is, for example, the case of the `sls` matrix on which `qrm` achieves a poor scalability and absolute performance, although still doing better than `spqr`. It has to be noted, however, that DGEQRF performance in Figure 10 is related to the factorization of square matrices; when run on strongly overdetermined matrices the DGEQRF routine achieves a very poor absolute performance and scaling as well.

Two techniques may be used to overcome this limitation and are the object of ongoing research work:

- **2D node parallelism:** following the methods presented in [6, 14], a 2D partitioning could be applied to frontal matrices; this delivers finer granularity and thus better parallelism although

it requires more logic to correctly handle the assembly operations and to keep track of the status of the global factorization.

- **tree preprocessing:** the method described in [14] for the QR factorization of tall and skinny dense matrices relies on the idea of parallelizing the computations according to a tree reduction pattern. This idea can be easily adapted to the multifrontal QR factorization of sparse matrices, where the concept of tree is already present.

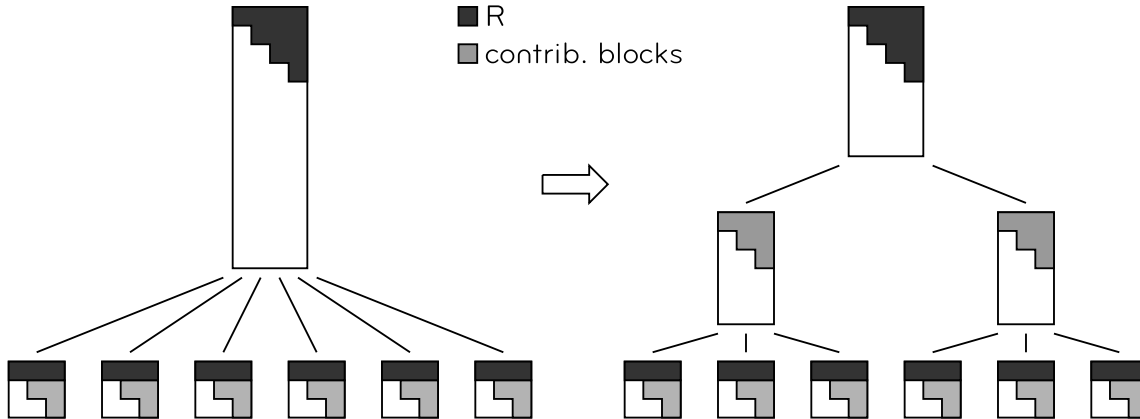


Figure 11: Fronts splitting in order to increase parallelism.

For example, the assembly tree in Figure 11 (*left*) could be rearranged into the tree in Figure 11 (*right*) which provides more tree parallelism and makes the root front less overdetermined. It has to be noted that the two intermediate nodes that appear in the assembly tree on the right, do not receive any coefficient from the original matrix in the assembly operation and do not provide any coefficient to the global  $R$  factor. Because the staircase structure of frontal matrices is exploited, no additional floating point operations are introduced by this modification which is completely transparent to the factorization phase and only involves some symbolic operations during the problem analysis.

## 5 Summary

A novel approach to the parallelization of the multifrontal QR factorization of a sparse matrix has been presented. A fine grained partitioning is applied to the frontal matrices and computational tasks are defined as the sequential execution of an elementary operation on a data partition. The tasks are arranged into a DAG where the edges define the dependencies among them and, thus, the order in which they have to be executed. This DAG globally retains the shape of the classical assembly tree in the multifrontal method but exposes the concurrency in both the tree and node types of parallelism allowing for a consistent exploitation of the two. Following a dataflow execution model, the tasks in the DAG can be scheduled dynamically depending on different policies. The scheduling of tasks is done according to a technique which aims at minimizing the transfer of data between NUMA nodes of large multicore systems and experimental results were presented which prove its effectiveness. Nonetheless, through a low-level performance profiling based on the PAPI tool, it has been shown that minimizing the conflicts on the memory system is much more important than maximizing the data locality and it has been speculated that a careful front-to-core mapping could help achieving both of these objectives; this is currently under investigation. Two techniques to reduce the size of the scheduling search space have been described. Finally, experimental results have been presented on two different architectures for ten test matrices to prove the efficiency of the proposed approach. These results show that an actual implementation of the proposed methods achieves a good scalability and a good absolute performance up to 32 cores outperforming by a factor of up to three the best equivalent software currently available. The causes of poor efficiency

on some of the test cases have been identified and possible solutions have been presented which are the object of ongoing research work.

## Acknowledgments

I wish to express my gratitude to the members of the MUMPS team for the countless advices they gave me on the implementation of multifrontal methods and to Chiara Puglisi for sharing with me her deep knowledge of the sparse QR factorization techniques.

## References

- [1] Kleen A. An NUMA API for Linux. Technical report, SUSE Labs, 2004.
- [2] Patrick R. Amestoy, Iain S. Duff, J. Koster, and Jean-Yves L'Excellent. MUMPS: a general purpose distributed memory sparse solver. In A. H. Gebremedhin, F. Manne, R. Moe, and T. Sorevik, 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.
- [3] Patrick R. Amestoy, Iain S. Duff, and Chiara Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Int. Journal of Num. Linear Alg. and Appl.*, 3(4):275–300, 1996.
- [4] Å. Björck. *Numerical methods for Least Squares Problems*. SIAM, Philadelphia, 1996.
- [5] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186, feb. 2010.
- [6] 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.
- [7] T. A. Davis. University of Florida sparse matrix collection, 2002. <http://www.cise.ufl.edu/research/sparse/matrices>.
- [8] T. A. Davis. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.
- [9] T. A. Davis. Multifrontal multithreaded rank-revealing sparse QR factorization. *Accepted for publication on ACM Transactions on Mathematical Software*, 2009.
- [10] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
- [11] 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.
- [12] J. A. George and Michael T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and its Applications*, 34:69–83, 1980.
- [13] Abdou Guermouche, Jean-Yves L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.
- [14] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Enhancing parallelism of tile qr factorization for multicore architectures. *Submitted to Transaction on Parallel and Distributed Systems*.
- [15] Jonathan Hogg, John K. Reid, and Jennifer A. Scott. A DAG-based sparse Cholesky solver for multicore architectures. Technical Report RAL-TR-2009-004, Rutherford Appleton Laboratory, 2009.

- [16] J. W. H. Liu. On general row merging schemes for sparse Givens transformations. *SIAM J. Sci. Stat. Comput.*, 7:1190–1211, 1986.
- [17] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12:127–148, 1986.
- [18] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.
- [19] P. Matstoms. Parallel sparse QR factorization on shared memory architectures. Technical Report LiTH-MAT-R-1993-18, Department of Mathematics, 1993.
- [20] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. 1999.
- [21] J. R. Rice. PARVEC workshop on very large least squares problems and supercomputers. Technical Report CSD-TR 464, Purdue University, IN., 1983.
- [22] 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.
- [23] J. Silc, B. Robic, and T. Ungerer. Asynchrony in parallel computing: From dataflow to multi-threading. *Journal of Parallel and Distributed Computing Practices*, 1:1–33, 1998.
- [24] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with PAPI-C. *Tools for High Performance Computing 2009*, pages pp. 157–173.