Fine-Grained Multithreading for the Multifrontal QR Factorization of Sparse Matrices

Alfredo Buttari

▶ To cite this version:

Alfredo Buttari. Fine-Grained Multithreading for the Multifrontal QR Factorization of Sparse Matrices. SIAM Journal on Scientific Computing, 2013, vol. 35 (n° 4), pp. 323-345. 10.1137/110846427 . hal-01122471

HAL Id: hal-01122471
https://hal.science/hal-01122471

Submitted on 4 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Open Archive TOULOUSE Archive Ouverte (OATAO)
OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in : http://oatao.univ-toulouse.fr/
Eprints ID : 12707

To link to this article : DOI :10.1137/110846427
URL : http://dx.doi.org/10.1137/110846427

To cite this version : Buttari, Alfredo *Fine-Grained Multithreading for the Multifrontal QR Factorization of Sparse Matrices*. (2013) SIAM Journal on Scientific Computing, vol. 35 (n° 4). pp. 323-345. ISSN 1064-8275

Any correspondance concerning this service should be sent to the repository administrator: staff-oatao@listes-diff.inp-toulouse.fr

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

ALFREDO BUTTARI [†]

March 3, 2015

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 [27, 3].

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 transformations (see Schreiber and Van Loan [29]), 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 [15], where the fill-in is minimized by using Givens

*This paper is a significantly revised and extended version of the one appearing in the PARA 2010 conference proceedings [5].

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

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 [13] and then extended to the QR factorization [21, 14], 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 better exploits the memory subsystems and the possibility of using Householder reflectors 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 [24, 2, 8]; all of them are based on the same approach to parallelization which suffers scalability limits on modern, multicore systems (see Section 3).

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 [30] 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 [13] 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 [7]. At the heart of this method is the concept of an *elimination tree*, introduced by Schreiber [28] and extensively studied and formalized later by Liu [23]. 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^TA share the same structure under the hypothesis that the matrix A is *Strong Hall* (for a definition of this property see, for example, Bjork's book [3, page 234]). Based on this equivalence, the elimination tree for the QR factorization of A is the same as that for the Cholesky factorization of A^TA. In the case where the Strong Hall property does not hold, the elimination tree related to the Cholesky factorization of A^TA can still be used although the resulting QR factorization will perform more com-

putations 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 topological order (i.e., 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. This operation simply consists in stacking these data one on top of the other: the rows of these data sets can be stacked in any order but the order of elements within rows has to be the same. The set of column indices associated with a node is, in general, a superset of those associated with its children;
- **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 through the WY representation [29]. 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 some details of a sparse QR factorization. The factorized matrix[1] is shown on the left part of the figure where `a` letters are used for representing the original matrix coefficients and dots for representing the fill-in introduced by the factorization. On the top-right part of the figure, the structure of the resulting R factor is shown. The elimination/assembly tree is, instead

[1]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.

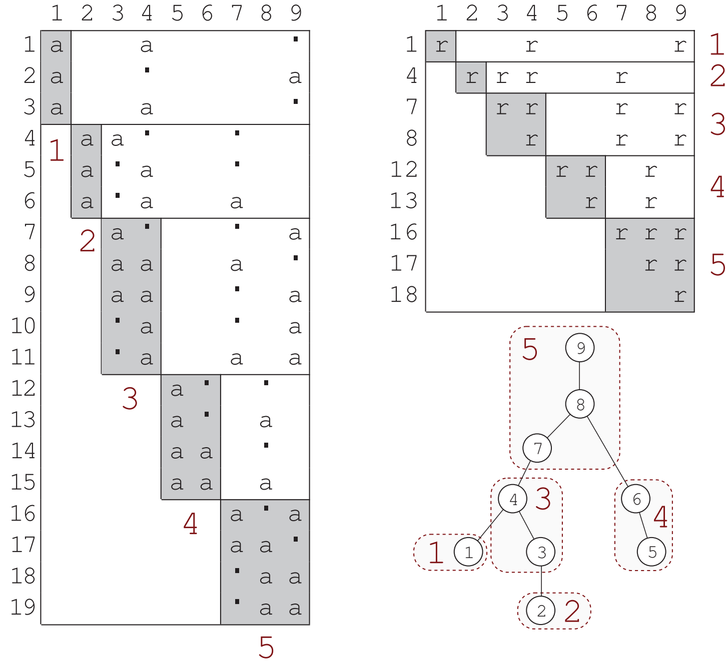

Figure 1: Example of multifrontal QR factorization. On the left side the matrix with the original coefficient represented as a and the fill-in coefficients introduced by the factorization as dots. On the upper-right part, the structure of the resulting R factor. On the right-bottom part the elimination tree; the dashed boxes show how the nodes are amalgamated into supernodes.

reported in the bottom-right part: dashed boxes show how the nodes can be amalgamated into supernodes with the corresponding indices denoted by bigger size numbers. The amalgamated nodes have the same row structure in R modulo a full, diagonal block. 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. The supernode indices are reported on the left part of the figure in order to show the pivotal columns eliminated within the corresponding supernode and on the top-right part to show the rows of R produced by the corresponding supernode factorization.

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.

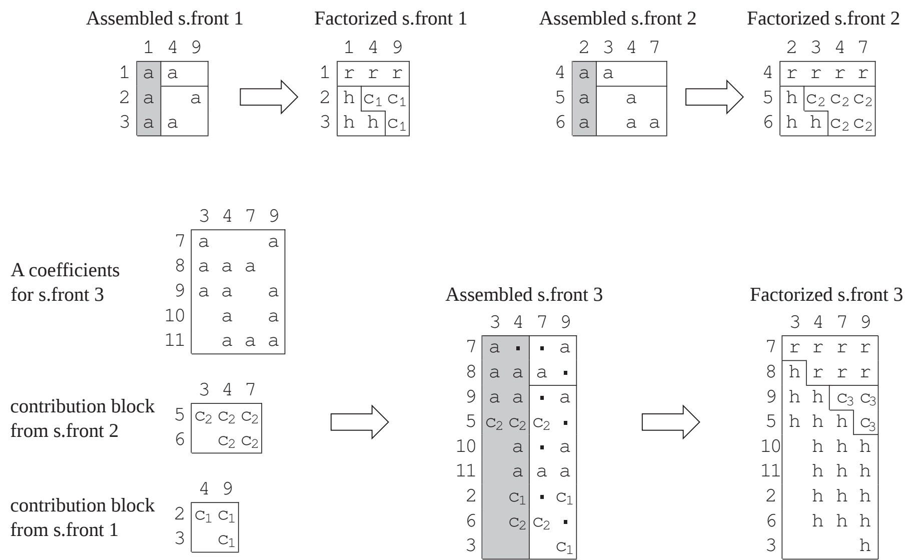

Figure 2: Assembly and factorization of the frontal matrices associated with supernodes 1, 2 and 3 in Figure 1. Coefficients from the original matrix are represented as `a`, those in the resulting R and H matrices as `r` and `h`, respectively, the fill-in coefficients as dots and the coefficients in the contribution blocks as `c` with a subscript to specify the supernode they belong to. Pivotal columns are shaded in gray.

Figure 2 shows the assembly and factorization operations for the supernodes 1, 2 and 3 in Figure 1 when these optimization techniques (referred to as *Strategy 3* in Amestoy *et al.* [2]) are applied. Note that, because supernodes 1 and 2 are leaves of the assembly tree, the corresponding assembled frontal matrices only include coefficients from the matrix A. 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 Figure 2 (*bottom-middle*). Once the front is assembled, it is factorized as shown in Figure 2 (*bottom-right*).

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 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 and the determination of the elimination tree or the symbolic factorization to estimate the amount of memory needed during the factorization phase. The cost of the analysis phase is $\mathcal{O}(|A| + |R|)$ and can be considered negligible with respect to the matrix factorization.

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 favorable ratio between memory accesses and computations can be achieved which reduces the utilization of the memory system and opens opportunities for multithreaded, parallel execution. This property, resulting from the use of Level-3 BLAS operations (i.e., matrix-matrix operations), is commonly known as the *surface-to-volume* [12] property because, for a problem of size n (the size of a frontal matrix, for the multifrontal algorithm), n^3 operations are performed on n^2 data.

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 Liu [23, Proposition 10.1]);
- 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 [24, 2, 8] 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 completed. 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 other hand, 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 (for example, Intel MKL) 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 the 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 factorized.

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 by Buttari *et al.* [6] and extended to the supernodal Cholesky factorization of sparse matrices by Hogg *et al.* [18].

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 Figure 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; Figure 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 reflectors resulting from the panel reduction; Figure 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;

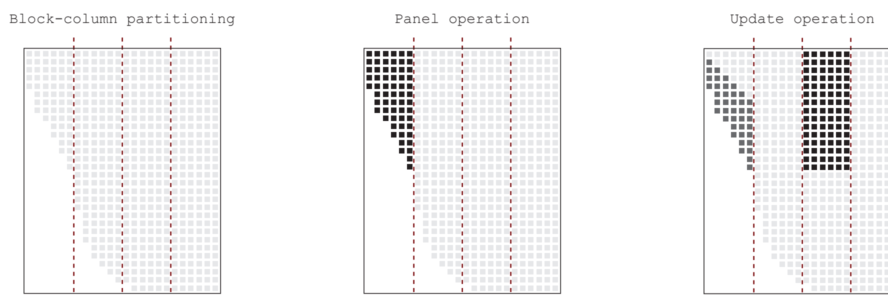

Figure 3: Block-column partitioning of a frontal matrix (*left*) and panel and update operations pattern (*middle* and *right*, respectively); dark gray coefficients represent data read by an operation while black coefficients represent written data.

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

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 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;

[2]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.

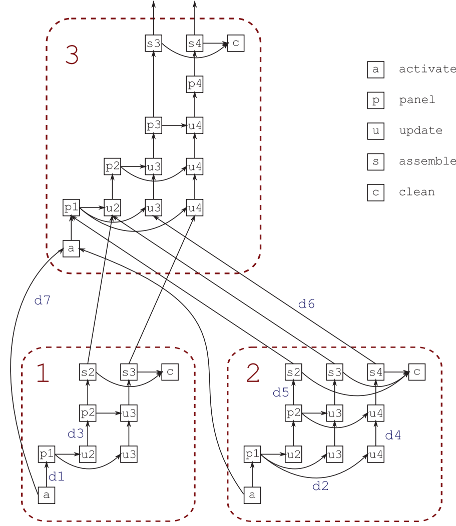

Figure 4: The DAG associated with supernodes 1, 2 and 3 of the problem in Figure 1; for the `panel`, `update` and `assemble` operations, the corresponding block-column index is specified. For this example, the block-column size is chosen to be one.

- `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. Moreover, it implicitly represents both tree and node parallelism which allows to exploit both of them in a consistent way. Finally, it removes unnecessary dependencies making it possible, for example, 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 factorized.

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 [18];
- 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. Although the multifrontal method is rich in Level-3 BLAS operations (mostly in the `update` tasks), there is still a considerable amount of Level-2 BLAS operations (within the `panel` tasks) and symbolic or memory ones (the `activate`, `assembly` and `clean` tasks) whose efficiency is limited by the speed of the memory system. As the number of threads participating to the factorization increases, the relative cost of these memory-bound operations grows too high and there are fewer opportunities to hide this cost by overlapping these slow operations with faster ones. In addition, some frontal matrices, especially at the bottom of the tree, may be too small to achieve the surface-to-volume effect in Level-3 BLAS operations. Therefore, in order to improve the scalability, it is important to perform these memory-bound operations as efficiently as possible and this can be achieved by executing each of them on the core which is closest to the data it manipulates. The proposed 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 [4], 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.

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 (i.e., tasks whose dependencies are already satisfied) 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` [4].

3. executes the selected task if the `pick_task` routine has succeeded.

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; the search for an activable front follows a postorder traversal of the assembly tree, which provides a good memory consumption and temporal locality of data. Because the ownership of such a 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. An efficient use of tree-level parallelism makes it hard, if not impossible, to follow a postorder traversal of the tree which results in an increased memory consumption with respect to the sequential case [22, 16]. It is important to note that the proposed scheduling method tries to exploit node-level parallelism as much as possible and dynamically resorts to tree-level parallelism by activating a new node only when no more tasks are found on already active fronts. This keeps the tree traversal as close as possible to the one followed in the sequential execution and avoids the memory consumption to grow out of control.

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

Main loop

```
mainloop: do
   if(n_ready_tasks < ntmin) then
      ! if the number of threads falls
      ! below a certain value, fill-up
      ! the queues
      call fill_queues()
   end if

  tsk = pick_task()
  select case(tsk%id)
  case(panel)
     call execute_panel()
  case(update)
     call execute_update()
  case(assemble)
     call execute_assemble()
  case(activate)
     call execute_activate()
  case(clean)
     call execute_clean()
  case(finish)
     exit mainloop
end do
```

fill_queues()

```
found = .false.

forall (front in active fronts)
   ! for each active front try to schedule
   ! ready tasks
   found = found .or. push_panels(front)
   found = found .or. push_updates(front)
   found = found .or. push_assembles(front)
   found = found .or. push_clean(front)
end forall

if (found) then
   ! if tasks were pushed in the previous
   ! loop return
   return
else
   ! otherwise schedule the activation of
   ! the next ready front
   call push_activate(next ready front)
end if

if (factorization over) call push_finish()
```

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

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 what is really needed. A logical pruning can thus be 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.

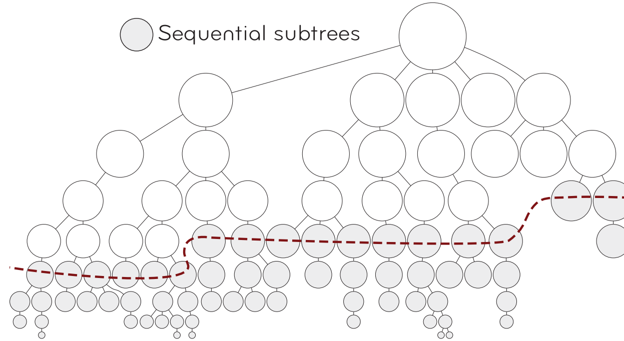

Figure 6: A graphical representation of how the logical amalgamation and logical pruning may be applied to an assembly tree.

- **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 processed. 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 [22]. 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, $P_{19} = 10$ (the nodes active at the moment when the peak is reached are highlighted with a thick border); instead, if the tree is 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 Liu and Guermouche *et al.* [22, 16]).

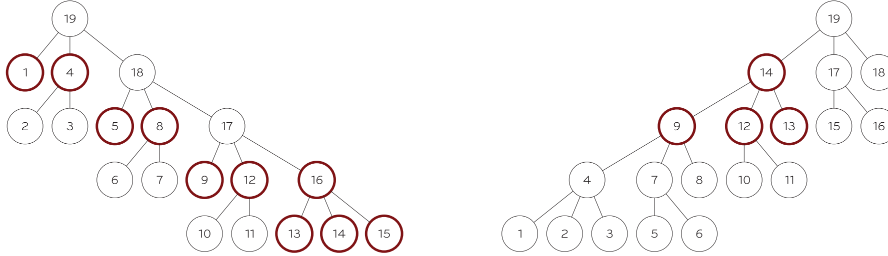

Figure 7: The effect of leaves reordering on the number of active nodes.

Both the tree pruning and reordering are executed during the analysis phase.

3.3 Blocking of dense matrix operations

It is obviously desirable to use blocked operations that rely on Level-3 BLAS routines in order to achieve a better use 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. 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.

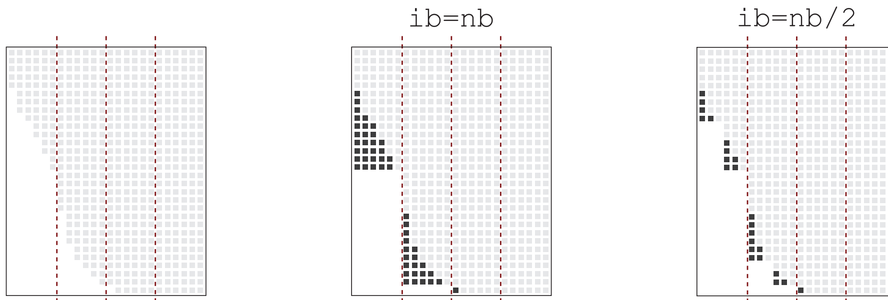

Figure 8: The effect of internal blocking on the generated fill-in. The light gray dots show the frontal matrix structure if no blocking of operations is applied whereas the dark gray dots show the additional fill-in introduced by blocked operations.

Figure 8 shows as dark gray dots the extra fill-in introduced by the blocking of operations (denoted as `ib` for internal blocking) with respect to the partition-

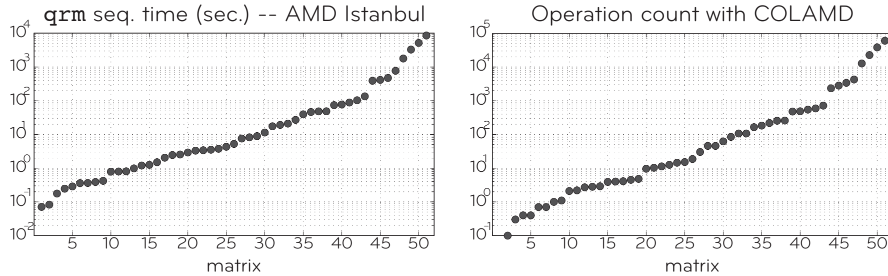

Figure 9: Some details of the 51 matrices in the test set. On the left side, the operation count obtained with the COLAMD ordering and on the right side the qrm sequential factorization time (in seconds) on the AMD Istanbul system. In each graph, matrices are sorted in increasing order of the reported measure.

ing size nb on an example frontal matrix. Details on how to choose these bock sizes will be provided in Section 4.1.

4 Experimental results

The method discussed in Section 3 was implemented in a software package called qr_mumps[3] (or qrm for the sake of brevity). The code is written in Fortran2003 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 on a set of fifty matrices from the UF Sparse Matrix Collection [10]; they were chosen from the complete set of over and under-determined problems by excluding those that are rank-deficient (because qr_mumps cannot currently handle them), those that are too small to evaluate the scalability or that are too big for being factorized on the computer platforms described below. To these, a large matrix from a meteorology application from the HIRLAM[4] research program was added for a total of 51 test matrices. In the case of under-determined systems, the transposed matrix is factorized, as it is commonly done to find the minimum-norm solution of a problem. All of the results presented below were produced without storing the H matrix in order to extend the test set to very large matrices that couldn't otherwise be factorized on the available computers. Figure 9 shows, for all these matrices, the operation count obtained when a COLAMD [9] fill-reducing column permutation is applied and the corresponding qrm sequential factorization time on the AMD system on the left and the right side, respectively; data are sorted in increasing order in each subfigure.

[3] http://buttari.perso.enseeiht.fr/qr_mumps

[4] http://hirlam.org

#	Mat. name	m	n	nz	op. count (Gflops)
1	image_interp	240000	120000	711683	30.2
2	LargeRegFile	2111154	801374	4944201	84.7
3	cont11_l	1468599	1961395	5382999	184.5
4	EternityII_E	11077	262144	1572792	544.0
5	degme	185,501	659415	8127528	591.9
6	cat_ears_4_4	19020	44448	132888	716.1
7	Hirlam	1385270	452200	2713200	2339.9
8	e18	24617	38602	156466	3399.5
9	flower_7_4	27693	67,593	202218	4261.2
10	Rucci1	1977885	109900	7791168	12768.5
11	sls	1748122	62729	6804304	22716.2
12	TF17	38132	48630	586218	38203.1

Table 1: A subset of the test matrices. The operation count is related to the matrix factorization with COLAMD column permutation.

Out of these 51 problems, a subset of 12 matrices, described in Table 1, was selected and analyzed in details in order to highlight the effectiveness of the proposed algorithms and methods.

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 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.

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.

4.1 Choosing the block size

As explained in Section 3.3, the execution time of the factorization operation depends on the `nb` and `ib` parameters which define the granularity of tasks (and thus the amount of concurrence) and the blocking of dense linear algebra operations, respectively. The optimal values for these two parameters depend mostly on the input matrix structure, the architectural features of the underlying computer and the number of working threads. Clearly, small `ib` values yield lower fill-in but inefficient BLAS operations and vice versa; on the other hand, small `nb` values deliver more parallelism but increase the complexity and the cost of the scheduling and set an upper bound to the `ib` parameter which may not be optimal.

Figure 10 shows how the execution time is affected by different choices of the two blocking parameters on the Power6 machine for two different matrices. It can be seen that, for both matrices, smaller `nb` values are required in order to achieve the shortest execution time on 32 cores compared to the 16 cores case.

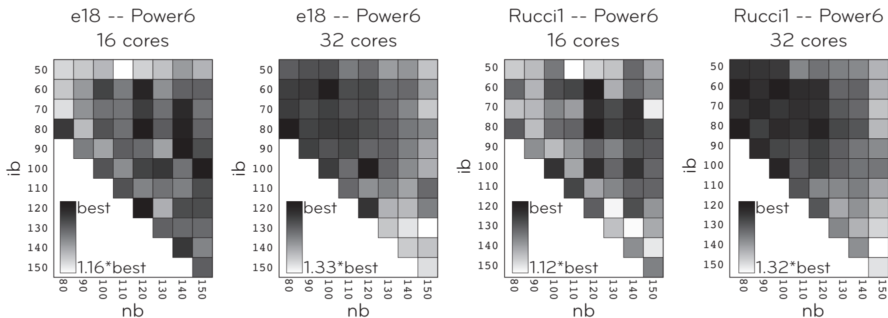

Figure 10: The effect of the blocking sizes on the factorization time of the e18 and Rucci1 matrices on the Power6 computer.

Although it is very expensive to determine experimentally the optimal values for `nb` and `ib`, Figure 10 as well as 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.

For all the 51 matrices in the test set and for the two architectures in Table 2, the blocking values were chosen among three different combinations, i.e., `(nb,ib)`=(120, 120), (120, 60) or (60, 60) (this last one is not shown in Figure 10) as those that delivered the shortest factorization time using all the cores available on the system. All the experimental results presented in the rest of this section are related to that choice.

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 [25, 31] 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[5];
- the number of conflicts on the DRAM controllers[6]

Experiments were run on all the matrices in the test set and with three different settings for the memory policy:

- no locality: this is a code variant where the multiple task queues are replaced with a single one shared by all threads. Tasks are, thus, pushed and popped from this queue regardless of their affinity with the data placement in the memory system;
- 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;
- round robin: this setting uses the same code variant of the "locality" one but, in this case, allocated memory pages are interleaved in a round robin fashion over all the DRAM modules. This is achieved using the `numactl` [20] 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.

[5] 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.

[6] 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.

Matrix	4	7	8	9	10	11
	Time (sec.)					
no. loc.	8.97	28.48	53.38	53.07	157.20	371.78
loc.	7.57	25.77	48.43	48.25	143.48	345.75
r. r.	6.54	22.38	28.90	42.50	113.70	328.40
	Dwords on HT ($\times 10^9$)					
no. loc.	23.59	65.61	81.40	109.45	371.73	803.27
loc.	11.92	47.30	76.94	90.52	259.89	656.50
r. r.	25.00	72.60	86.13	125.66	378.09	869.87
	Conflicts on DCT ($\times 10^9$)					
no. loc.	0.26	0.68	0.98	1.07	3.29	7.36
loc.	0.29	0.70	0.92	1.21	4.07	8.14
r. r.	0.24	0.52	0.62	0.79	3.17	6.31

Table 3: Analysis of the memory usage for the matrix factorization on the AMD system with 24 threads.

Table 3 shows the results of these experiments for a subset of the matrices in Table 1. It can be seen that the locality aware scheduling strategy described in Section 3.2 provides better execution times with respect to a naive dynamic scheduling policy: the improvement may be as high as almost 20% (for the EternityII_E matrix) and quite often around 10%. This can be explained by the reduced amount of data transferred over the HyperTransport links as shown in the middle of the table. 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 completely scattered over the NUMA nodes; nonetheless, this more even distribution of data reduces the number of conflicts on the DRAM controllers (see Table 3 (*bottom*)) and provides a better use of the memory bandwidth. This behavior is coherent to what was observed on the `HSL_MA87` [18] code which uses a similar approach for the Cholesky factorization of sparse linear systems.

The considerable performance improvement resulting from the interleaved memory allocation suggests that reducing the memory conflicts may be more important than minimizing the amount of data transferred over the HyperTransport links. A closer look to the behavior of matrices e18 and flower_7_4 (columns three and four in Table 3) shows, instead, that both these objectives are very important for the efficiency of the code. The factorization of these two matrices takes roughly the same time in the “locality” and “no locality” cases but the memory interleaving provides only a small improvement to the second matrix due to a bigger increase of the data traffic on the HyperTransport links.

Maximizing the data locality and minimizing the memory conflicts are not conflicting objectives although it may be rather complicated to achieve both of them at the same time on a very heterogeneous workload such as a sparse

Matrix	1	2	3	5	6	11
none	3.52	385.10	151.40	7.12	9.53	9239.00
reord.	2.89	113.80	140.90	7.01	8.75	1651.00
prune	0.94	17.27	5.52	6.99	9.38	328.50
both	0.84	14.44	5.16	6.83	9.20	326.40

Table 4: The effect of tree pruning and tree reordering on the matrix factorization time on the AMD system with 24 threads.

factorization. It has to be noted that in the proposed locality aware scheduling policy, the placement of data and the ownership of the associated tasks is defined on a front basis. Because the number of fronts becomes smaller than the number of working threads when the factorization approaches the root front, a lot of work stealing and memory contention take place on the top part of the assembly tree where most of the work is done. Further improvements can be obtained by defining the data placement and the tasks affinity on a block-column basis; despite this would make some operations such as the front assembly much more complex, it is reasonable to expect that it will allow to more evenly and efficiently distribute the data and thus improve the locality of reference and reduce the memory contention at the same time. This is the object of ongoing work.

4.3 The effect of tree pruning and reordering

The tree reordering and pruning techniques have been evaluated on the test matrices. For the tree pruning, the initial threshold was set to 0.01 which means that all the subtrees whose weight is smaller than 1% of the total factorization workload are pruned off. If the remaining tree does not provide enough tree-level parallelism, the threshold is divided by 2 and a new pruning is done on the original assembly tree. More precisely this procedure is iterated until the number of leaves in the pruned tree is bigger than twice the number of working threads. Clearly, the optimal values for both the starting threshold and the stopping criterion depend on the structure of the tree and, therefore, on the specific input matrix; the values described above were determined experimentally and were found to work well on the large set of test matrices previously described.

Table 4 shows the experimental results related to a subset of matrices for which these techniques have proved to be particularly effective. The experiments show that, when applied separately, these two methods provide considerable benefits in some specific cases. The tree reordering yields good improvements on very unbalanced and irregular trees: this is the case of the LargeRegFile and sls matrices. The tree pruning, proved to be very effective on all the problems but particularly on those with extremely large trees and extremely small frontal matrices such as the cont11_l and the sls matrices.

It can be observed that the pruning clearly reduces the need for sorting as

well as its effectiveness. Nonetheless on some matrices (see, particularly, the first three columns in Table 4) the best execution time is achieved when both techniques are applied. It is important to note that the size of the pruned tree increases with the number of working threads and, therefore, the improvements provided by the sorting, when both techniques are applied, are likely to be more important for higher degrees of parallelism.

4.4 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 [8] (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 nodes amalgamation methods were used so that the differences between the produced assembly trees can be considered negligible. Note that other ordering tools based on nested dissection (such as METIS [19] or SCOTCH [26]) will produce more balanced and wider assembly trees which means better tree parallelism. Using such orderings will, thus, change the behavior of both packages as well as the outcome of the comparison; however, it is reasonable to expect that differences will not be substantial. 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).

Figure 11 shows time and memory profiles [11] for both codes using 24 threads. For any given code, a data point (x, y) on the profile means that the code is no worse than x times the best of the two codes for y problems. This means that the $(1, y)$ point gives the number y of problems on which the associated code was found to be the best. The time profile shows that `qrm` is faster than `spqr` on 43 problems out of 51; the remaining eight problems are very small, i.e., their factorization takes less than one second. On basically half the problems in the test set `qrm` was more than twice as fast as `spqr`. The memory profile is more difficult to interpret. Because `qrm` is based on a much more eager execution model it is reasonable to expect that it consumes more memory. Indeed, as explained in Section 3.2, the scheduling method exploits as much as possible the node parallelism and resorts on tree parallelism, by activating new nodes, only when needed; this keeps the tree traversal close to the one followed in the sequential case and limits the memory consumption growth in parallel. Moreover, the tree reordering technique, as explained in the same section, helps reducing the memory footprint. As a result, `qrm` achieves, in general, a smaller memory consumption than `spqr` on the 51 test matrices as shown in the memory

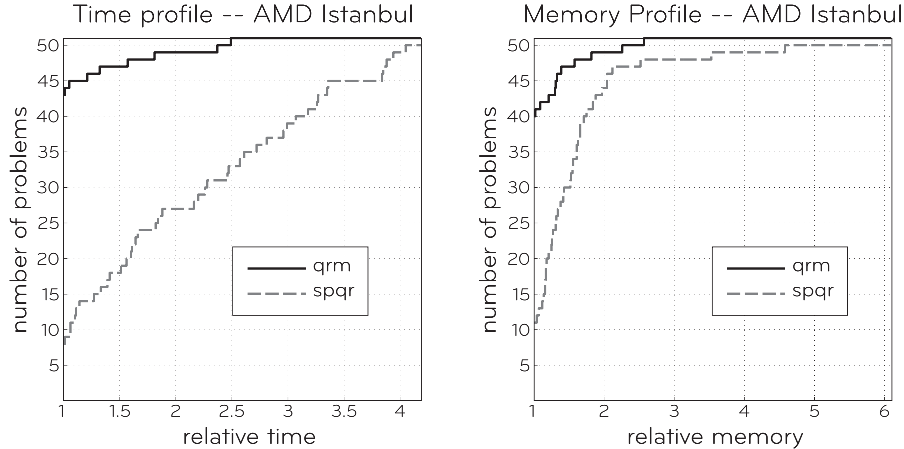

Figure 11: Time and memory profiles comparing the qrm and spqr factorizations for the 51 test matrices on the AMD Istanbul system.

profile of Figure 11. Experimental data show that most of the matrices where qrm has a better memory consumption are relatively small, which may probably be explained with some overhead in spqr that becomes negligible for bigger size problems. Note that, in both codes, the memory footprint is defined as the peak memory consumption reached during the factorization, including the input matrix and all the allocated memory areas of any type. On bigger problems, the two codes have a similar memory consumption and no clear winner can be identified. If the H matrix was kept in memory, which is not the case for the results reported in Figure 11, the difference between the two codes would be even smaller because the relative weight of the contribution blocks, responsible for the increased memory consumption in parallel, would be lower.

Tables 5 and 6 show detailed performance results for a selected subset of problems. These where chosen to be the biggest ones not belonging to the same family in the complete test set and correspond to matrices 3-12 in Table 1. 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.

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 5. 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 5 shows, for spqr, the factorization times for the best combination of tree

	Matrix	3	4	5	6	7	8	9	10	11	12
	AMD Istanbul										
qrm	th.										
	1	48.8	88.5	103.2	134.9	392.0	474.5	774.7	1786	3301	5185
	2	33.8	49.2	52.2	65.5	209.9	250.3	357.4	961	1802	2770
	4	14.8	24.8	26.5	33.4	106.4	126.3	181.6	495	932	1372
	6	12.4	17.6	18.4	22.9	71.9	86.6	124.7	341	655	923
	12	6.2	9.8	10.4	12.8	37.8	46.2	65.8	181	421	477
	18	5.2	7.9	7.7	9.1	27.3	32.9	48.6	132	341	325
	24	4.8	6.5	6.8	8.4	22.4	28.9	42.5	114	327	260
spqr	th.										
	1	40.6	99.5	111.0	123.3	406.3	538.3	687.5	2081	4276	5361
	2	29.9	63.4	69.1	74.2	238.2	290.1	379.2	1155	2870	2959
	4	23.2	44.9	46.6	48.4	148.4	169.0	229.9	738	2001	1659
	6	19.1	36.0	38.2	39.6	116.4	128.5	178.7	599	1846	1203
	12	14.7	26.3	33.0	32.5	85.7	90.5	131.6	468	1644	770
	18	12.5	22.6	28.8	29.1	73.4	78.7	119.1	405	1603	637
	24	11.7	20.7	26.2	27.8	68.6	74.1	114.2	372	1389	589
	b.	11.7	20.7	26.1	27.4	68.6	74.1	113.9	372	1375	586

Table 5: Factorization times, in seconds, on the AMD Istanbul system for `qrm` (*top*) and `spqr` (*bottom*). The first row shows the matrix number.

	Matrix	3	4	5	6	7	8	9	10	11	12
	IBM Power6										
qrm	th.										
	1	36.2	51.9	60.5	75.2	227.3	290.2	426.1	1156	2142	3121
	2	24.5	27.6	32.0	38.6	117.2	151.7	192.9	600	1259	1656
	4	10.6	14.0	16.0	19.4	57.1	76.5	98.0	304	672	850
	8	5.9	7.3	8.2	9.9	28.6	39.3	49.5	155	348	458
	16	3.9	4.1	4.7	5.7	15.4	21.3	27.6	84	197	235
	24	4.5	3.1	3.7	4.3	11.3	15.5	22.8	62	183	174
	32	6.4	2.6	3.3	4.0	9.5	12.9	18.9	52	182	141

Table 6: Factorization times, in seconds, on the Power6 system for `qrm`. The first row shows the matrix number.

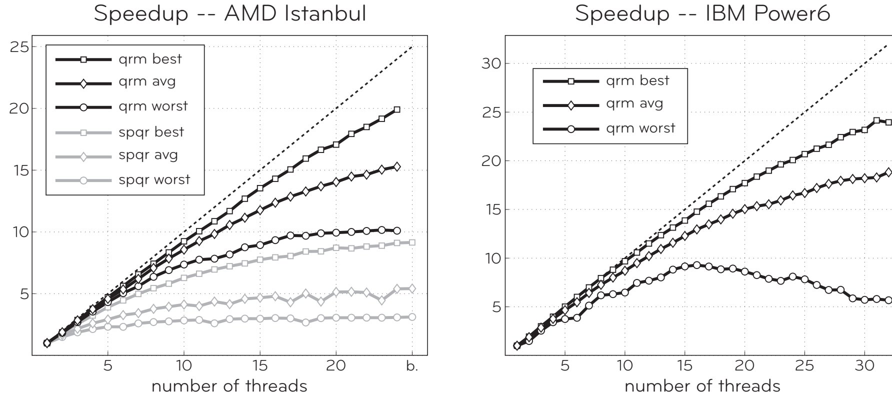

Figure 12: Best, worst and average speedup achieved on matrices 3-12 in Table 1, for both codes, on both systems for core numbers from one to the maximum available.

and node parallelism; for example, for the sls matrix (number 11) the shortest factorization time is achieved by allocating 22 threads to the tree parallelism and 7 to the BLAS parallelism for a total of 154 threads.

Table 5 shows that the proposed approach clearly outperforms the classical approach to parallelization of the QR multifrontal method on the AMD system as `qrm` achieves speedups of more than three over `spqr`.

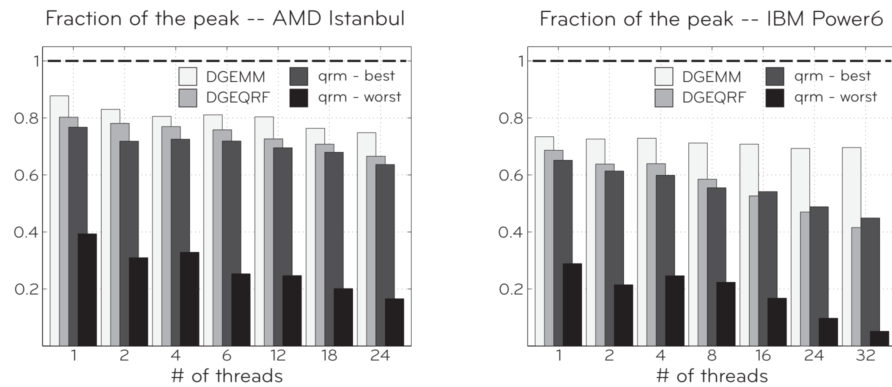

Figure 13: Fraction of the peak performance achieved by `qrm` compared to the QR factorization of a dense matrix.

Figure 12 shows the best, worst and average speedup achieved by the two codes on the AMD Istanbul system and by `qrm` only on the IBM Power6. The best cases are the TF17 and the Hirlam matrices on the AMD and IBM systems, respectively, whereas the worst are the sls and cont11_l ones, respectively; the best and worst case matrices are the same for `qrm` and `spqr`. Figure 13 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 cont11_l (number 12 and 3 in Table 1) matrices, respectively. The absolute performance and scalability of `qrm` is compared to that of the matrix multiply operation (`DGEMM`) and the QR factorization (`DGEQRF`) of square dense matrices of size 40,000 which is big enough to achieve asymptotic performance using any number of cores on the target machines.

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. 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 that `DGEQRF` performance in Figure 13 is related to the factorization of a square matrix; 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 Buttari *et al.* [6] and Hadri *et al.* [17], 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 proposed by Hadri *et al.* [17] 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.

  For example, the assembly tree in Figure 14 (*left*) could be rearranged into the tree in Figure 14 (*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.

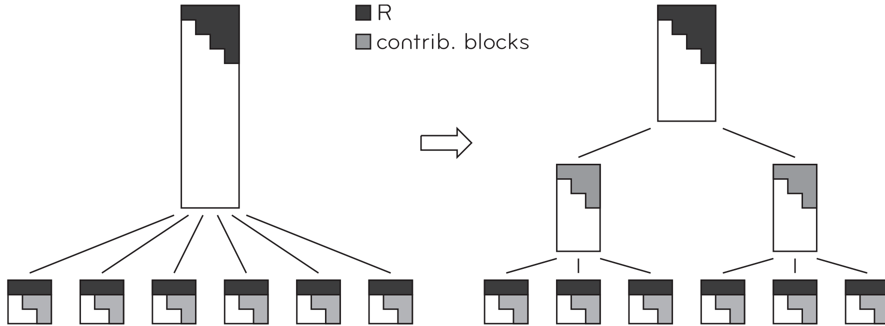

Figure 14: Fronts splitting in order to increase parallelism.

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 use 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 more important that maximizing the data locality although both play an important role in the scalability of a parallel, multithreaded code; a possible way of achieving both these objectives based on a finer data placement was suggested (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 51 test matrices to prove the efficiency of the proposed approach. These results show that an actual implementation of the proposed methods achieves good scalability and 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.

I also wish to thank Dan Terpstra for helping me with using the PAPI tool and interpreting the results it gave.

This work was performed using HPC resources from GENCI-IDRIS (Grant x2012065063).

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] 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 application. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference*, pages 180 –186, feb. 2010.

[5] A. Buttari. Fine granularity sparse qr factorization for multicore based systems. In *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*, PARA'10, pages 226–236, Berlin, Heidelberg, 2012. Springer-Verlag.

[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. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

[8] T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Math. Softw.*, 38(1):8:1–8:22, December 2011.

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

[10] T. A. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[11] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

[12] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM Press, Philadelphia, 1998.

[13] 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.

[14] 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.

[15] 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.

[16] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.

[17] B. Hadri, H. Ltaief, A. Agullo, and J. Dongarra. Tile QR factorization with parallel panel processing for multicore architectures. In *IPDPS*, pages 1–10. IEEE, 2010.

[18] 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.

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

[20] A. Kleen. An NUMA API for Linux. Technical report, SUSE Labs, 2004.

[21] J. W. H. Liu. On general row merging schemes for sparse Givens transformations. *SIAM J. Sci. Stat. Comput.*, 7:1190–1211, 1986.

[22] 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.

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

[24] P. Matstoms. Parallel sparse QR factorization on shared memory architectures. Technical Report LiTH-MAT-R-1993-18, Department of Mathematics, 1993.

[25] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters, 1999. Proceedings of Department of Defense HPCMP Users Group Conference.

[26] F. Pellegrini and Jean Roman. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of HPCN'96, Brussels,* LNCS 1067, pages 493–498, April 1996.

[27] J. R. Rice. PARVEC workshop on very large least squares problems and supercomputers. Technical Report CSD-TR 464, Purdue University, IN., 1983.

[28] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.

[29] 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.

[30] 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.

[31] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with PAPI-C. *Tools for High Performance Computing*, pages pp. 157–173, 2009.