Parallel Computing 39 (2013) 212–232

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Hierarchical QR factorization algorithms for multi-core clusters ☆

Jack Dongarra [a,b,c], Mathieu Faverge [a], Thomas Hérault [a], Mathias Jacquelin [f], Julien Langou [e], Yves Robert [a,d,*]

[a] University of Tennessee Knoxville 1122 Volunteer Blvd, Knoxville, TN 37996, USA
[b] Oak Ridge National Laboratory 1 Bethel Valley Rd, Oak Ridge, TN 37831, USA
[c] Manchester University, UK School of Computer Science, Manchester, M13 9PL, United Kingdom.
[d] Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France
[e] University of Colorado Denver PO Box 173364. Denver, CO 80217-3364, USA
[f] INRIA Saclay Campus de l'École Polytechnique, 91120 Palaiseau, France

ARTICLE INFO

Article history:
Received 11 January 2012
Received in revised form 13 September 2012
Accepted 21 January 2013
Available online 30 January 2013

Keywords:
QR factorization
Numerical linear algebra
Hierarchical architecture
Distributed memory
Cluster
Multi-core

ABSTRACT

This paper describes a new QR factorization algorithm which is especially designed for massively parallel platforms combining parallel distributed nodes, where a node is a multi-core processor. These platforms represent the present and the foreseeable future of high-performance computing. Our new QR factorization algorithm falls in the category of the tile algorithms which naturally enables good data locality for the sequential kernels executed by the cores (high sequential performance), low number of messages in a parallel distributed setting (small latency term), and fine granularity (high parallelism). Each tile algorithm is uniquely characterized by its sequence of reduction trees. In the context of a cluster of nodes, in order to minimize the number of inter-processor communications (aka, "communication-avoiding"), it is natural to consider hierarchical trees composed of an "inter-node" tree which acts on top of "intra-node" trees. At the intra-node level, we propose a hierarchical tree made of three levels: (0) "*TS* level" for cache-friendliness, (1) "low-level" for decoupled highly parallel inter-node reductions, (2) "domino level" to efficiently resolve interactions between local reductions and global reductions. Our hierarchical algorithm and its implementation are flexible and modular, and can accommodate several kernel types, different distribution layouts, and a variety of reduction trees at all levels, both inter-node and intra-node. Numerical experiments on a cluster of multi-core nodes (i) confirm that each of the four levels of our hierarchical tree contributes to build up performance and (ii) build insights on how these levels influence performance and interact within each other. Our implementation of the new algorithm with the DAGuE scheduling tool significantly outperforms currently available QR factorization software for all matrix shapes, thereby bringing a new advance in numerical linear algebra for petascale and exascale platforms.

1. Introduction

Future exascale machines will likely be massively parallel architectures, with 10^5 to 10^6 nodes, each node itself being equipped with 10^3 to 10^4 cores. At the node level, the architecture is a shared-memory machine, running many

☆ A preliminary version of part of the results presented in this paper appears in IPDPS'2012.

* Corresponding author. Address: Laboratoire LIP, ENS Lyon, 69364 Lyon Cedex 07. Tel.: +33 472728586; fax: +33 472728080.
E-mail address: Yves.Robert@ens-lyon.fr (Y. Robert).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.01.003

parallel threads on the cores. At the machine level, the architecture is a distributed-memory machine. This additional level of hierarchy, together with the interplay between the cores and the accelerators, dramatically complicates the design of new versions of the standard factorization algorithms that are central to many scientific applications. In particular, the performance of numerical linear algebra kernels is at the heart of many grand challenge applications, and it is of key importance to provide highly-efficient implementations of these kernels to leverage the impact of exascale platforms.

This paper investigates the impact of this hierarchical hardware organization on the design of numerical linear algebra algorithms. We deal with the QR factorization algorithm which is ubiquitous in high-performance computing applications, and which is representative of many numerical linear algebra kernels. In recent years, the quest of efficient, yet portable, implementations of the QR factorization algorithm has never stopped [1–8]. In a nutshell, state-of-the-art software has evolved from block-column panels to tile-based versions, and then to multi-eliminator algorithms. We briefly review this evolution in the following paragraphs.

First the LAPACK library [9] has provided Level 3 BLAS kernels to boost performance on a single CPU. The ScaLAPACK library [10] builds upon LAPACK and provides a coarse-grain parallel version, where processors operate on large block-column panels, i.e. blocks of b columns of the original matrix. Here b is the block size, typically $b = 200$ or more, for Level 3 BLAS performance. Inter-processor communications occur through highly tuned MPI send and receive primitives. The factorization progresses panel by panel. Once the current panel is factored, updates are applied on all the following panels (remember that the matrix operated upon shrinks as the factorization progresses). Sophisticated *lookahead* versions of the algorithms factor the next panel while the current updates are still being applied to the trailing matrix.

Then, the advent of multi-core processors has led to a major modification in the programmings [1,4,5,7]. To avoid any confusion, we define a *node* as a processor equipped with several cores. Now each node should run several threads in parallel to keep all cores within that node busy. Tiled versions of the algorithms have thus been designed: dividing large block-column panels into several tiles allows for a decrease in the granularity down to a level where many smaller-size tasks are spawned. In the current panel, the diagonal tile, or eliminator tile, is used to eliminate all the tiles below it in the panel. Because the factorization of the whole panel is now broken into the elimination of several tiles, the update operations can also be partitioned at the tile level, which generates many tasks to feed all cores. However, the dependencies between these tasks must be enforced, and the algorithms have become much more complicated.

A technical difficulty arises with the elimination operations within the panel: these are sequential because the diagonal tile is used for each of them, hence it is modified at each elimination operation. This observation applies to the updates as well: in any trailing column, the update of a tile must wait until the update of its predecessor tile is completed. To further increase the degree of parallelism of the algorithms, it is possible to use several eliminator tiles inside a panel. The only condition is that concurrent elimination operations must involve disjoint tile pairs. Of course, in the end there must remain only one non-zero tile on the panel diagonal, so that all eliminators but the diagonal tile must be eliminated themselves later on, using a reduction tree of arbitrary shape (e.g. serial, fully binary, ...). The extra source for parallelism resides in the fact that the whole matrix can be partitioned into domains, with one eliminator per domain responsible for eliminating the tiles local to the domain. In each domain, all these operations, and the corresponding updates, are independent and can run concurrently. Such multi-eliminator algorithms represent the state-of-the-art for a single node, but they are still being refined, because the impact of the reduction trees which are chosen is not fully understood, and also because using many eliminators implies the use of different tile kernels, called *TT* kernels, which are less-efficient than the *TS* kernels used with a single eliminator per panel.

The goal of this paper is to move a step forward and to introduce a flexible and modular algorithm for *clusters*, where a cluster is a multi-node machine (each node being a multi-core processor). Tackling such hierarchical architectures is a difficult challenge for two reasons. The first challenge arises from the algorithmic perspective. Brand new avenues must be explored to accommodate the hierarchical nature of multi-core clusters. Concurrent eliminators allow for more parallelism, but the reduction tree that follows breaks the smooth pipelining of operations from one panel to the next. With one domain per node, we may not have enough parallel operations to feed all the many-cores, so we may need to have several domains per node. The reduction operations involve inter-node communications, which are much slower than intra-node shared memory accesses. Limiting their number could be achieved with a block row-distribution, but this would severely imbalance node loads. This small list is not exhaustive: good load-balance, efficient pipelining, and memory locality are all conflicting objectives. The main contribution of this paper is to provide a novel algorithm that is fully aware of the hierarchical nature of the target platform and squeezes the most out of its resources.

The second challenge is at the programing level. Within a node, the architecture is a shared-memory machine, running many parallel threads on the cores. But the global architecture is a distributed-memory machine, and requires MPI communication primitives for inter-node communications. A slight change in the algorithm, or in the matrix layout across the nodes, might call for a time-consuming and error-prone process of code adaptation. For each version, one must identify, and adequately implement, inter-node versus intra-node kernels. This dramatically complicates the task of the programmer if he relies on a manual approach. We solve this problem by relying on the DAGuE software [8,11], so that we can concentrate on the algorithm and forget about MPI sends and thread synchronizations. Once we have specified the algorithm at a task level, the DAGuE software will recognize which operations are local to a node (and hence correspond to shared-memory accesses), and which are not (and hence must be converted into MPI communications). Our experiments show that this

approach is very powerful, and that the use of a higher-level framework does not prevent our algorithms from outperforming all existing solutions.

In this paragraph, we briefly highlight our contribution with respect to existing work (see Section 3 for a full overview). Two recent papers [2,8] have discussed tiled algorithms for clusters of multi-core nodes. In [2], the authors use a two-level hierarchical tree made of an inter-node binary tree on top of an intra-node *TS* flat tree and use a 1D block data layout. A flat tree is a reduction tree with a single eliminator (the same tile here is sued to eliminate all other tiles). The limitations are: (1) the use of a flat tree at the node level is not adapted when the local matrices are tall and skinny; (2) the use of the 1D block data layout results in serious load imbalance for square matrices. In [8], the authors use a plain flat tree on top of a 2D block data layout. The limitations are: (1) the use of a flat tree is not adapted for tall and skinny matrices; (2) the flat tree with natural ordering is not aware of the 2D data block cyclic distribution and therefore performs many more communications than needed. Our algorithm addresses all these issues while keeping the positive features. At the intra-node level, we propose a hierarchical tree made of three levels: (0) "*TS* level" for cache-friendliness, (1) "low-level" for decoupled highly parallel inter-node reductions, (2) "domino level" to efficiently resolve interactions between local reductions and global reductions. Finally (3) a "high-level" tree is used for the inter-node reduction. The use of the "high-level" tree enables a small number of inter-processor communications, thereby making our algorithm "communication-avoiding". For the levels (1), (2) and (3) of the hierarchical algorithm, the reduction can accommodate any tree. Our implementation is flexible and modular, and proposes several reduction trees per level. This allows us to use those reduction trees which are efficient for a given matrix shape. Finally the "domino level" – which operates within a node, and fits in between the intra- and inter-node reductions – resolves all interactions between the low-level and high-level trees, in such a way that the low-level tree (acting within a node) becomes decoupled from the influence of the other nodes. To summarize, our new algorithm is a tile QR factorization which is (a) designed especially for massively parallel platforms combining parallel distributed multi-core nodes; (b) features a hierarchical four-level tree reduction; (c) incorporates a novel domino level; (d) is 2D block cyclic aware; and (e) implements a variety of trees at each level. The resulting properties of the algorithm are (i) cache-efficiency at the core level, (ii) high granularity at the node level, (iii) communication avoiding at the distributed level, (iv) excellent load balancing overall, (v) nice coupling between the inter-node and intra-node interactions, and (vi) ability to efficiently handle any matrix shape.

The rest of the paper is organized as follows. We start with a quick review of tiled QR algorithms (Section 2). Then we detail the key principles underlying the design of state-of-the-art algorithms from the literature (Section 3). The core contributions of the paper reside in the next three sections. The new algorithm is presented in full details, first with a general description (Section 4), and then with analytical formulas for the number of operations at each kernel that compose the algorithm (Section 5). Then in Section 6, we report experiments showing that we outperform current state-of-the-art implementations. We observe that the execution time of each algorithmic variant is in direct accordance with its critical path length, thereby showing a very good match between predicted and actual performance. Finally, we provide some concluding remarks and future research directions in Section 7.

Algorithm 1: Generic QR algorithm

for k $= 0$ *to* $\min(m,n)-1$ **do**
  for i $= k+1$ to $m-1$ **do**
    $elim(i, eliminator(i,k), k)$

Algorithm 2: $elim(i, eliminator(i,k), k)$

(a) *With TS (Triangle on top of square)* kernels
$GEQRT(eliminator(i,k), k)$
$TSQRT(i, eliminator(i,k), k)$
for j $= k+1$ to $n-1$**do**
  $UNMQR(eliminator(i,k), k, j)$
  $TSMQR(i, eliminator(i,k), k, j)$

(b) With *TT (Triangle on top of triangle)* kernels
$GEQRT(eliminator(i,k), k)$
$GEQRT(i,k)$
forj $= k+1$ to $n-1$**do**
  $UNMQR(eliminator(i,k), k, j)$
  $UNMQR(i,k,j)$
$TTQRT(i, eliminator(i,k), k)$
for j $= k+1$ to $n-1$**do**
  $TTMQR(i, eliminator(i,k), k, j)$

2. Tiled QR algorithms

The general shape of a tiled QR algorithm for a tiled matrix of $m \times n$ tiles, whose rows and columns are indexed from 0, is given in Algorithm 1. Here i and k are tile indices, and we have square $b \times b$ tiles, where b is the block size. Thus the actual size of the matrix is $M \times N$, with $M = m \times b$ and $N = n \times b$. The first loop index k is the panel index, and $elim(i, eliminator(i,k),k)$ is an orthogonal transformation that combines rows i and $eliminator(i,k)$ to zero out the tile in position (i,k). Each $elim(i, eliminator(i,k),k)$ consists of two sub-steps: (i) first in column k, tile (i,k) is zeroed out (or eliminated) by tile $(eliminator(i,k),k)$; and (ii) in each following column $j > k$, tiles (i,j) and $(eliminator(i,k),j)$ are updated; all these updates are independent and can be triggered as soon as the elimination is completed. The algorithm is entirely characterized by its *elimination list*, which is the ordered list of all the eliminations $elim(i, eliminator(i,k),k)$ that are executed.

To implement an orthogonal transformation $elim(i, eliminator(i,k),k)$, we can use either *TT* kernels or *TS* kernels, as shown in Algorithm 2. In a nutshell, a tile can have three states: square, triangle, and zero. Initially, all tiles are square. An eliminator must be a triangle, and we transform a square into a triangle using the *GEQRT* kernel. With a single eliminator, we start by transforming it into a triangle (kernel *GEQRT*) before eliminating square tiles. To eliminate a square with a triangle, we use the *TSQRT* kernel. With several eliminators, we have several triangles, hence the need for an additional kernel to eliminate a triangle (rather than a square): this is the *TTQRT* kernel. The number of arithmetic operations performed by a *TSQRT* kernel (to eliminate a square) is the same as that of a *GEQRT* (transform the square into a triangle) followed by a *TTQRT* (eliminate a triangle). The same observation basically applies for the corresponding updates, which can be decomposed in a similar way (see Algorithm 2): *UNMQR* is the update after a *GEQRT*, *TSMQR* is the update after a *TSQRT*, and *TTMQR* is the update after a *TTQRT*. The *TS* kernels can only be used within a flat tree at the first tree level (so that tiles are square). On the one hand, *TT* kernels offer more parallelism than *TS* kernels. On the other hand, the sequential performance of the *TS* kernels is higher (e.g., by 10% in our experimental section) than the one of the *TT* kernels. We refer to [1] for more information on the various kernels.

Any tiled QR algorithm used to factor a tiled matrix of $m \times n$ tiles is characterized by its elimination list. Obviously, the algorithm must zero out all tiles below the diagonal: for each tile $(i,k), i > k$, $0 \leqslant k < \min(m,n)$, the list must contain exactly one entry $elim(i, \star, k)$, where $\star$ denotes some row index $eliminator(i,k)$. There are two conditions for a transformation $elim(i, eliminator(i,k),k)$ to be valid:

- both rows i and $eliminator(i,k)$ must be ready, meaning that all their tiles left of the panel (of indices (i,k') and $(eliminator(i,k),k')$ for $0 \leqslant k' < k$) must have already been zeroed out: all transformations $elim(i, eliminator(i,k'),k')$ and $elim(eliminator(i,k), eliminator(eliminator(i,k),k'),k')$ must precede $elim(i, eliminator(i,k),k)$ in the elimination list;
- row $eliminator(i,k)$ must be a potential eliminator, meaning that tile $(eliminator(i,k),k)$ has not been zeroed out yet: the transformation $elim(eliminator(i,k), eliminator(eliminator(i,k),k),k)$ must follow $elim(i, eliminator(i,k),k)$ in the elimination list.

Assuming square $b \times b$ tiles and using a $b^3/3$ floating point operation unit, the weight of *GEQRT* is 4, *UNMQR* 6, *TSQRT* 6, *TSMQR* 12, *TTQRT* 2, and *TTMQR* 6. A critical result is that no matter what elimination list is used, or which kernels are called, the total weight of the tasks for performing a tiled QR factorization algorithm is constant and equal to $6mn^2 - 2n^3$. Using $M = m \times b$, and $N = n \times b$, we retrieve $2MN^2 - 2/3N^3$ floating point operations, the exact same number as for a standard Householder reflection algorithm, as found in LAPACK (e.g., [9]). In essence, the execution of a tiled QR algorithm is fully determined by its elimination list. Each transformation involves several kernels, whose execution can start as soon as they are ready, i.e., as soon as all dependencies have been enforced.

3. Related work

In this section, we survey tiled QR algorithms from the literature, and we outline their main characteristics. We start with several examples to help the reader better understand the combinatorial space that can be explored to design such algorithms.

3.1. Factoring the first panel

In this section we discuss several strategies for factoring the first panel, of index 0, of a tiled matrix of $m \times n$ tiles. When designing an efficient algorithm, individual panel factorization should not be considered separately from the rest of the factorization, but concentrating on a single panel is enough to illustrate several important points.

Consider a panel with $m = 12$. All tiles except the diagonal, tile 0, must be zeroed out. We also know that in all algorithms, tile 0 will be used as the eliminator in the last elimination. The simplest solution is to use a single eliminator for the whole panel. If we do so, this single eliminator has to be the diagonal tile. The eliminations will be all sequentialized (because the eliminator tile is modified during each elimination), but they can be performed in any order. In Table 1, we use an ordering from top to bottom. For each tile, we give the index of its eliminator. We also give the step at which it is zeroed out, assuming

Table 1
Flat tree reduction of panel 0.

Row index	Eliminator	Step
0	★	
1	0	1
2	0	2
3	0	3
4	0	4
5	0	5
6	0	6
7	0	7
8	0	8
9	0	9
10	0	10
11	0	11

that each elimination can be executed within one time unit. The elimination list is then $elim(1, 0, 0), elim(2, 0, 0), \ldots, elim(m-1, 0, 0)$. The corresponding *reduction tree* for panel 0 is a tree with m leaves and $m-1$ internal nodes, one per elimination. Each internal node can also be viewed as the value of the eliminator tile just after the elimination. Each internal node has two predecessors, namely the two tiles used to perform the elimination. In the example, internal nodes are arranged along a chain, with original tiles being sequentially input, see Fig. 1. The tree of Fig. 1 is called the *flat tree*. A tile eliminated at step i in Table 1 is at distance $S - i + 1$ of the tree root, where S is the last time-step ($S = 11$ in the example). Note that the reduction tree fully characterizes the elimination list for the panel, since it provides both the eliminator and the time-step for each elimination.

With a single eliminator, all eliminations in the panel must be executed one after the other. The only source of parallelism resides in the possibility to execute the updates of some previous eliminations while zeroing out the next tile. However, parallel eliminations are possible if we conduct these on disjoint pairs of rows. In the beginning, we can have as many eliminators as half the number of rows. During the next step, half of the remaining non-zero rows can be eliminated. Iterating, we reduce the panel with a *binary tree* instead of a flat tree, as illustrated in Fig. 2. The elimination list is $elim(2i+1, 2i, 0), i = 0, \ldots, \lceil \frac{m-2}{2} \rceil$, followed by $elim(4i+4, 4i, 0), i = 0, \ldots, \lceil \frac{m-5}{4} \rceil$, and so on. The last elimination is $elim(2^{\lceil \log_2 m \rceil - 1}, 0, 0)$.

With several eliminators, we have to use *TT* elimination kernels, which are less efficient than *TS* kernels. This relative inefficiency of the *TT* kernels is a first price to pay for parallelism. A second price to pay arises from locality issues. In a shared-memory environment, re-using the same eliminator several times allows for better cache reuse. This is even more true in distributed-memory environments, where the cost of communications can be much higher than local memory accesses. In such environments, we have to account for the data distribution layout. Assume that we have $p = 3$ nodes P_0, P_1 and P_2. There are two classical ways to distribute rows to nodes, by blocks, or cyclically. (In the general case one would use a 2D grid, but we use a 1D grid for simplicity in this example). These two distributions are outlined below:

Clusters	Matrix rows (block)	Matrix rows (cyclic)
P_0	0, 1, 2, 3	0, 3, 6, 9
P_1	4, 5, 6, 7	1, 4, 7, 10
P_2	8, 9, 10, 11	2, 5, 8, 11

In our example, the block distribution nicely fits with the flat tree reduction. With this combination of block/flat, the ordering of the eliminations is such that the diagonal tile is communicated only once from one node to the next one. Adding a last communication to store the tile back in P_0 gives a count of p communications. On the contrary, the cyclic distribution is communication-intensive for the flat tree reduction, since we obtain as many as m communications, one per elimination and one for the final storage operation. However, there are two important observations to make:

1. With any data layout, one can always re-order the eliminations so as to perform only p communications with a flat tree. The eliminator can perform all local eliminations before being sent to the next node. With the cyclic/flat combination in the example, we eliminate rows $3, 6, 9$, then rows $1, 4, 7, 10$, and finally rows $2, 5, 8, 11$.

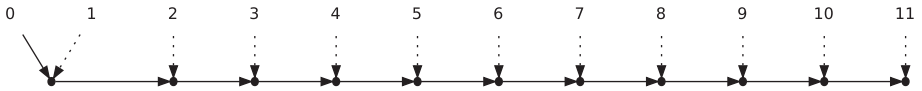


Fig. 1. Flat tree for panel 0.

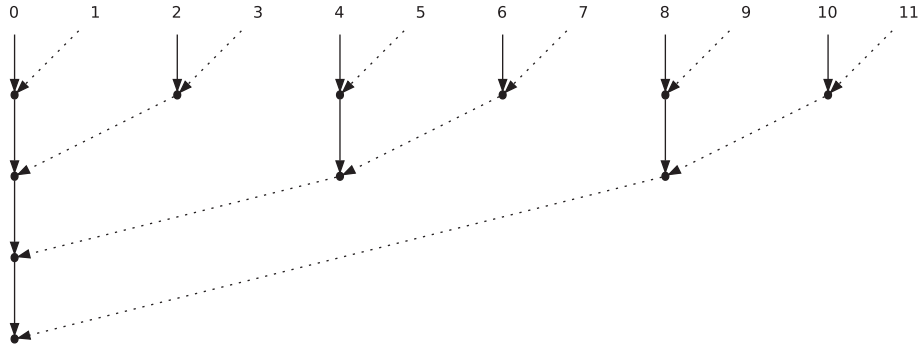


Fig. 2. Binary tree for panel 0.

2. There is a downside to fewer communications, namely higher start-up times. The cyclic/flat combination enables each node to become active much earlier (starting the updates) while the re-ordering dramatically increases waiting times. Note that waiting times are also high for the block/flat combination.

We make similar observations for the binary tree. In the example, the cyclic distribution requires many more inter-node eliminations than the block one, which requires only two, namely the last two eliminations. But this is an artifact of the example (take $p = 4$ instead of $p = 3$ to see this). In fact, for both distributions, a better solution may be to use *local* flat trees: within each node, a single tile acts as the eliminator for all the local tiles. These flat trees are independent from one node to another, and the eliminations proceed in parallel. Then a binary tree of size p is used to eliminate $p - 1$ out of the p remaining tiles (one per node). Moreover, since the local trees operate in parallel, there are no more high waiting times at start-up. Therefore, it alleviates the main drawback of the case with a single global eliminator giving priority to tiles local to the first process before unlocking other processes. This flat/binary reduction is illustrated in Fig. 3. In this example, the local eliminators are rows $0, 1$ and 2, and the binary tree has only 3 leaves, one per node. Note that the tree is designed with a cyclic distribution in mind: with a block distribution, the local eliminators would be rows $0, 4$ and 8.

Further refinements can be proposed. The flat/binary strategy may suffer from not exhibiting enough parallelism at the node level: local trees do execute in parallel, but each with a single eliminator. Parallelizing local eliminations may be needed when the node is equipped with many cores. The idea is then to partition the rows assigned to each node into smaller-size *domains*. Each domain is reduced using a flat tree, but there are more domains than nodes. This domain tree reduction is illustrated in Fig. 4 with two domains per node. In the example each domain is of size 2, hence the corresponding flat tree is reduced to a single elimination, but there are two domains, hence two eliminators, per node. The next question is: how to reduce these six eliminators? We can use a binary tree, as shown in Fig. 4. But there is a lot of flexibility here. For instance we may want to give priority to local eliminations, hence to reduce locally before going inter-node. This amounts to using a local reduction tree to eliminate all domain eliminators but one within a node, and then a global reduction tree to eliminate all remaining eliminators but one within the panel. Let $m = p \times d \times a$, where a is the domain size and p the number of nodes. There are d domains per node, hence each local reduction tree is of size d, while the global reduction tree is of size p. Note that these two trees may well be of different nature, all combinations are allowed! In the example, there are only $d = 2$

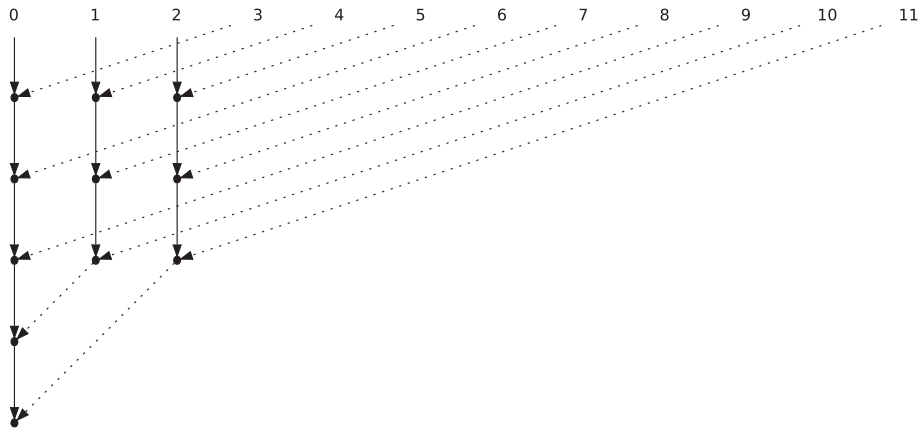


Fig. 3. Flat/binary tree for panel 0.

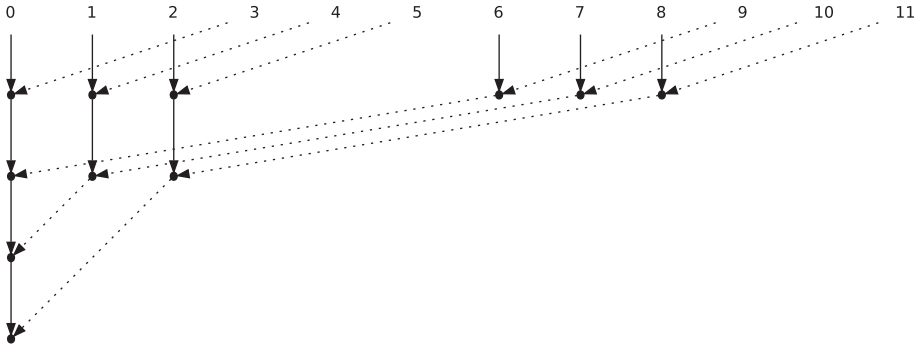


Fig. 4. Domain tree for panel 0, with two domains per node.

domains per node, so the local tree is unique, and using a binary tree for the global tree leads to the same elimination scheme as using a single binary tree for the six eliminators, as in Fig. 4.

3.2. Factoring several panels

We have reviewed several strategies to factor the first panel of the $m \times n$ tile matrix. But the whole game amounts to factoring $\min(m, n)$ panels, and efficiently pipelining these factorizations is critical to the performance of the QR algorithm. This section aims at illustrating several trade-offs that can be made.

A striking observation is that using a flat tree reduction in each panel provides a perfect pipelining, while using a binary tree reduction in each panel provokes "bumps" in the schedule, as illustrated with 3 panels in Tables 2 and 3. This explains that flat trees have been predominantly used in the literature, until the advent of machines equipped with several cores. Such architectures called for using several eliminators in a given panel, hence for binary trees, and later domain trees.

Table 2
Flat tree reduction for the first 3 panels.

Row index	Panel 0		Panel 1		Panel 2	
	Eliminator	Step	Eliminator	Step	Eliminator	Step
0	★		★		★	
1	0	1	★		★	
2	0	2	1	3	★	
3	0	3	1	4	2	5
4	0	4	1	5	2	6
5	0	5	1	6	2	7
6	0	6	1	7	2	8
7	0	7	1	8	2	9
8	0	8	1	9	2	10
9	0	9	1	10	2	11
10	0	10	1	11	2	12
11	0	11	1	12	2	13

Table 3
Binary tree reduction for the first 3 panels.

Row index	Panel 0		Panel 1		Panel 2	
	Eliminator	Step	Eliminator	Step	Eliminator	Step
0	★		★		★	
1	0	1	★		★	
2	0	2	1	3	★	
3	2	1	1	4	2	5
4	0	3	3	4	2	7
5	4	1	1	5	4	6
6	4	2	5	3	2	9
7	6	1	5	4	6	5
8	0	4	7	5	6	8
9	8	1	1	6	8	7
10	8	2	9	3	2	10
11	10	1	9	4	10	5

Table 4
Greedy reduction for the first 3 panels.

Row index	Panel 0		Panel 1		Panel 2	
	Eliminator	Step	Eliminator	Step	Eliminator	Step
0	★		★		★	
1	0	4	★		★	
2	1	3	1	6	★	
3	0	2	2	5	2	8
4	1	2	2	4	3	7
5	2	2	3	4	4	6
6	0	1	3	3	5	6
7	1	1	4	3	5	5
8	2	1	5	3	6	5
9	3	1	6	2	7	4
10	4	1	7	2	8	4
11	5	1	8	2	10	3

The inefficient pipelining of binary trees has only been identified recently. To remedy this problem while keeping several eliminators inside a panel, one can use the Greedy reduction outlined in Table 4. The Greedy algorithm nicely combines intra-panel parallelism and inter-panel pipelining. In fact, under the simplifying assumption of unit-time eliminations (hence regardless of their number of updates), it has been shown [12,13] that no algorithm can proceed faster! At each step, the Greedy algorithm eliminates as many tiles as possible in each column, starting with bottom rows. The pairing for the eliminations is done as follows: to eliminate a bunch of z consecutive tiles at the same time-step, the algorithm uses the z rows above them as eliminators, pairing them in the natural order. For instance in Table 4, the bottom six tiles in column 1 are simultaneously eliminated during the first step, using the six tiles above them as eliminators.

Recall from the study with a single panel that locality issues are very important in a distributed-memory environment, i.e. with several nodes. The previous Greedy algorithm is not suited to a matrix whose rows have been distributed across nodes, and two levels of reduction, local then global, are still highly desirable. But in addition to locality, a new issue arises when factoring a full matrix instead of a single panel: because the number of active rows decreases from one panel to the next, block distributions are no longer equivalent to cyclic distributions: the former induces a severe load imbalance (nodes become inactive as the execution progresses) while the latter guarantees that each node receives a fair share of the work until the very end of the factorization.

Finally, we point out that dealing with a coarse-grain model where each elimination requires one time unit, as in all previous tables and figures, is a drastic simplification. Tiled algorithms work at the tile level: after each zero-ing out, as many update tasks are generated as there are trailing columns after the current panel. The total number of tasks that are created during the algorithm is proportional to the cube of the number of tiles, and schedulers must typically set priorities to decide which tasks to execute among those ready for execution. Still, the coarse-grain model allows us to understand the main principles that guide the design of tiled QR algorithms.

3.3. Existing tiled QR algorithms

While the advent of multi-core machines is somewhat recent, there is a long line of papers related to tiled QR factorization. Tiled QR algorithms have first been introduced in Buttari et al. [4,14] and Quintana-Ortí et al. [5] for shared-memory (multi-core) environments, with an initial focus on square matrices. The sequence of eliminations presented in these papers is analogous to Sameh-Kuck [15], and corresponds to reducing each panel with a flat tree: in each column, there is a unique eliminator, namely the diagonal tile.

The introduction of several eliminators in a given column dates back to [15–17], although in the context of traditional element-wise (non-blocked) algorithms.

In the context of a single tile column, the first use of a binary tree algorithm (working on tiles) is due to da Cunha et al. [18]. Demmel et al. [6] present a general tile algorithm where any tree can be used for each panel, while Langou [19] explains the tile panel factorization as a reduction operation.

For shared-memory (multi-core) environments, recent work advocates the use of domain trees [7] to expose more parallelism with several eliminators while enforcing some locality within domains. Another recent paper [1] introduces tiled versions of the Greedy algorithm [12,13] and of the Fibonacci scheme [16], and shows that these algorithms are asymptotically optimal. In addition, they experimentally turn out to outperform all previous algorithms for tall and skinny matrices.

Preliminary hierarchical two-level trees have been presented by Agullo et al. in the context of grid computing environment [3] (binary on top of binary, for tall and skinny matrices), Agullo et al. in the context of multi-core platform [7] (binary on top of flat, for any matrix shapes), and Demmel et al. in the context of multi-core platform [20] (binary on top of flat, for tall and skinny matrices).

In this paper, we further investigate the impact of the Greedy and Fibonacci schemes, but for distributed-memory environments. There are two recent works for such environments. The approach of [3] uses a hierarchical approach: for each ma-

trix panel, it combines two levels of reduction trees: first several local binary trees are applied in parallel, one within each node, and then a global binary tree is applied for the final reduction across nodes. Because [3] focuses on tall and skinny matrices, it uses a 1D block distribution for the matrix layout (hence a 1D node grid). The approach of [2] also uses a hierarchical approach, and also uses a 1D block distribution. The main difference is that the first level of reduction is conducted with a flat tree within each node. We point out that the block distribution is suited only for tall and skinny matrices, not for general matrices. Indeed, with an $m \times n$ matrix and p nodes, the cyclic distribution is perfectly balanced (neglecting lower order terms), while the speedup attainable by the block distribution is bounded by $p(1 - \frac{n}{3m})$: this is acceptable if $n \ll m$ but a high price to pay if, say, $m = n$. However, it is quite possible to modify the algorithm in [2] so as to use a cyclic distribution, at the condition of re-ordering the eliminations to give priority to local ones over those that require inter-node communications. In fact, the hierarchical algorithm introduced in this paper can be parametrized to implement either version, the original algorithm in [2] as well as the latter variant with cyclic layout.

4. Hierarchical algorithm

This section is devoted to the new hierarchical algorithm that we introduce for clusters of multi-core nodes. We outline the general principles (Section 4.1) before working out the technical details through an example (Section 4.2). Then we briefly discuss the implementation within the DAGuE framework in Section 4.3.

4.1. General description

Here is a high-level description of the features of the hierarchical algorithm, HQR:

1. Use domains of a tiles, and use *TS* kernels within domains. Thus, within each node, every a-th tile sequentially eliminates the $a - 1$ tiles below it. The idea is to benefit from the arithmetic efficiency of *TS* kernels. Note that if $a = 1$, the algorithm will use only *TT* kernels.
2. Use intra-node reduction trees within nodes. Here, the idea is to locally eliminate as many tiles as possible, without inter-node communication. These intra-node trees depend upon the internal degree of parallelism of the nodes: we can use a binary tree or a greedy reduction for nodes with many cores, or a flat tree reduction if more locality and CPU efficiency is searched for. Note that these reductions are necessarily based upon *TT* kernels, because they involve eliminator tiles from the domains. *1* and *2* allow the algorithm to create a trade-off inside each node between parallelism (*2*) and efficiency (*1*) according to the number of resources available and the shape of the matrix.
3. Use inter-node reduction trees across nodes (again, necessarily based upon *TT* kernels). The inter-node reduction trees are of size p, because for each panel they involve a single tile per node. Here also, the trees can be freely chosen (FlatTree, BinaryTree, Greedy or Fibonacci).
4. Finally, use a 2D cyclic distribution of tiles along a virtual $p \times q$ node grid. The 2D-cyclic distribution is the one that best balances the load across resources.

There are many parameters to explore: the arithmetic performance parameter a, the shape $p \times q$ of the virtual grid if we are given C_1 physical nodes with C_2 cores each, and the shape of the intra- and inter- node reduction trees. In fact, there are two additional complications:

- Consider a given node: ideally, we would like to eliminate all tiles but one in each panel, i.e., we would like to reduce each node sub-matrix to a diagonal, and then proceed with inter-node communications to finish up the elimination. Unfortunately, because of the updates, it is not possible to locally eliminate "in advance" so many tiles, and one needs to wait for the inter-node reduction to progress significantly to be able to perform the last local eliminations. This problem is illustrated in Fig. 5(a). We propose an optimization, that we call *domino optimization* or *domino level tree*, to relax this constraint. Each process will perform the local eliminations on the local matrix as if it was the complete operation. Then, the tiles between the local diagonal and the global are reduced progressively with a domino effect from bottom to top. One can well imagine that other kinds of tree could have been used for this domino level tree. The domino technique greatly improves on the pipelining between the intra-node and inter-node trees, as illustrated in Fig. 5(b). However, the domino reductions may have a negative effect: they faster reduce the area of the local matrix that can be eliminated with combinations of the first two tree levels, *TS* domains and local trees. Therefore they improve parallelism and pipelining on tall and skinny matrices, but can deteriorate the efficiency on square matrices.
- The actual (physical) distribution of tiles to nodes needs not obey the virtual $p \times q$ node grid. In fact, we can always use another grid to map tiles to nodes. This additional flexibility allows us to execute all previously published algorithms simply by tuning the actual distribution parameters. For instance, to run the algorithm of [2] on a $m \times n$ tiled matrix, using a block distribution on r nodes, we take a virtual grid value $p = 1$ with domains of size $a = m/r$, and we let the actual data distribution be $CYCLIC(r)$.

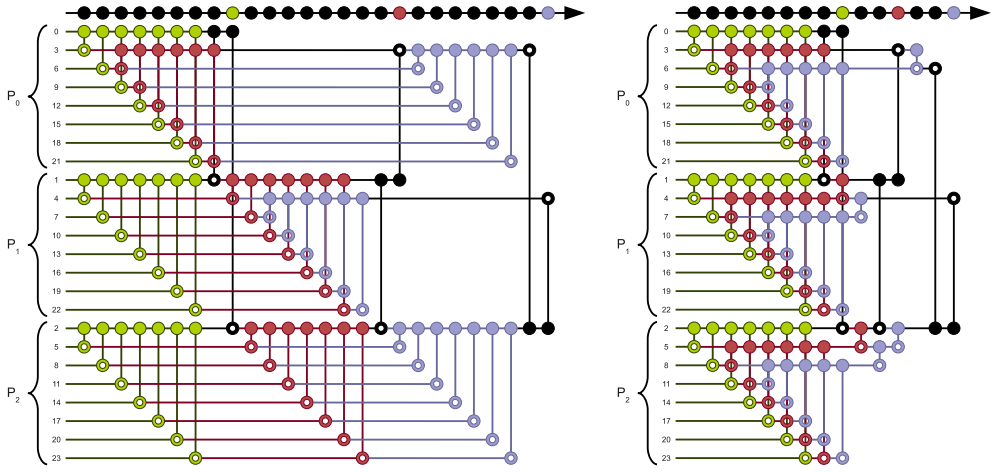


Fig. 5. Effect of the domino optimization on the pipelining of the low-level and high-level trees. Each color represents the elimination of one panel. Black nodes correspond to eliminations done in distributed mode (high-level tree) to finish up the factorization. Both trees used in this example are FlatTree.

4.2. Working out an example

Consider a $m \times n$ tiled matrix, with $m = 24$ and $n = 10$. We use a $p \times q$ virtual grid with $p = 3$ and $q = 1$, and an arithmetic parameter $a = 2$. Thus we have a unidimensional grid with $p = 3$ nodes. Global and local views of the matrix are shown in Fig. 6, where each tile is labeled with its reduction level. These reduction levels are explained below. In addition, the time-step at which each tile gets eliminated is given in Fig. 7, with domino optimization either disabled or enabled. In both figures, tiles are colored according to their assigned node (red[1] for P_0, yellow for P_1 and green for P_2).

4.2.1. Inter-node eliminations: Level 3 tiles

Consider a given panel of index k. The first p tiles on or below the diagonal of that panel, in position (k, k) to $(k + p - 1, k)$, are called Level 3 tiles. Hence for panel k, there is a single Level 3 tile per node. Unless it is the diagonal tile, the Level 3 tile of a node is the last tile eliminated in that node; it is also the only tile in that node that is involved in inter-node eliminations. Reducing the p Level 3 tiles for a given panel is achieved by a reduction tree of size p which we call *high-level*, because it is the only inter-node tree. For each panel, the high-level tree can be freely chosen as FlatTree, BinaryTree, Greedy or Fibonacci.

4.2.2. Intra-node eliminations: Level 0, Level 1 and Level 2 tiles

Level 0 tiles. Level 0 tiles are tiles eliminated by a *TS* kernel. In the example, domains are of size $a = 2$, so that in essence every second tile is eliminated by a *TS* kernel, and the eliminator is always the tile above it in the local view of the Fig. 6(b). However, as shown in Fig. 6(b), this holds true only for even-numbered tiles that are below the local diagonal. This local diagonal is a line of slope 1 in the local view, hence of slope p in the global view. If the matrix is tall and skinny, the proportion of Level 0 tiles tends to be one half, but it is much less for square matrices. This is why the domino optimization can be disabled for square matrices, in order to keep the ratio of *TS* kernels. In this case, the parallelism is naturally created by the large number of columns.

Level 1 tiles. Level-1 tiles are the local eliminators of Level 0 tiles that lie strictly below the local diagonal. Such tiles can be eliminated locally, without any inter-node communication. In other words, it is possible to eliminate all Level 0 and Level 1 tiles locally, in parallel on each node, before needing any inter-node communication. At the end of this local elimination, all tiles lying in the lower triangle below the local diagonal have been eliminated, and the last eliminator on each panel is the tile on the local diagonal (e.g., tile $(6, 2)$ for panel 2 in node P_0). The elimination of the lower triangle can be conducted using various types of reduction trees, FlatTree, BinaryTree, Greedy or Fibonacci.

Level 2 tiles. Level 2 are the "domino" tiles. In each panel, using the local view within a node, they are located between the top tile (not included) and the local diagonal tile (included). Their number increases together with the panel index, since Level 2 tiles lie between a line of slope $1/p$ and one of slope 1 in the local view. While Level 0 and Level 1 tiles are eliminated independently within each node, Level 2 tiles can only be eliminated after some inter-node communication has taken place. The goal of the domino level tree is to efficiently resolve interactions between local reductions and global reductions by removing the dependencies between each local tree, so that they can be fully performed in parallel, and eliminating all Level 2 tiles as soon as possible to release next communications. To see the domino level tree in action, consider the first Level 2

[1] (For interpretation of the colours in figs. 5,6,7,8,9,10,11,12 the reader is referred to the web version of this article.)

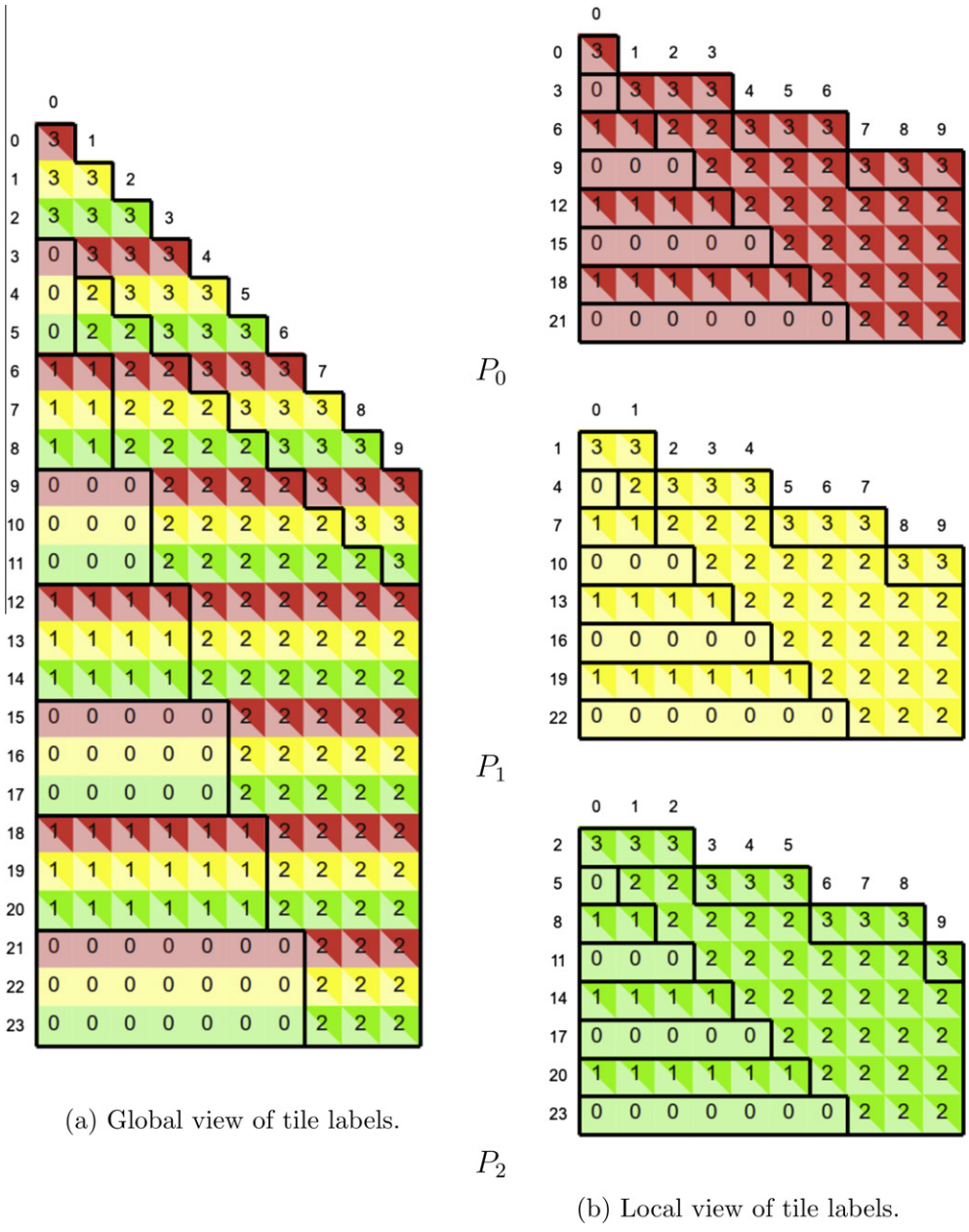


Fig. 6. Tile levels.

tile, in position $(4,1)$ and assigned to P_1. Tile $(4,1)$ is eliminated by tile $(1,1)$, the top tile of P_1 for panel 1: this corresponds to the elimination $elim(4,1,1)$, which is intra-node (within P_1). But tile $(1,1)$ is not ready to eliminate tile $(4,1)$ until it has been updated for the elimination $elim(1,0,0)$, which is inter-node: Level 3 tile $(0,0)$ eliminates Level 3 tile $(1,0)$, and tile $(1,1)$ is updated during this elimination. As soon as the update ends, $elim(4,1,1)$ is triggered, and tile $(4,1)$ is eliminated. A similar sequence takes place on to P_2, where the update of tile $(2,1)$ during $elim(2,0,0)$ (inter-node) must precede the elimination of Level 2 tile $(5,1)$ (during $elim(5,2,1)$, intra-node). This is illustrated by the Figs. 5((a) and (b)). In fact, we see that inter-node eliminations in the high-level tree successively trigger eliminations in the domino tree, like a domino that ripples in the area of Level 2 tiles.

4.3. Implementation with DAGuE

With an infinite number of resources, the execution would progress as fast as possible. The elimination list of the algorithm is the composition of the reduction trees at all the different levels. All eliminators are known before the execution.

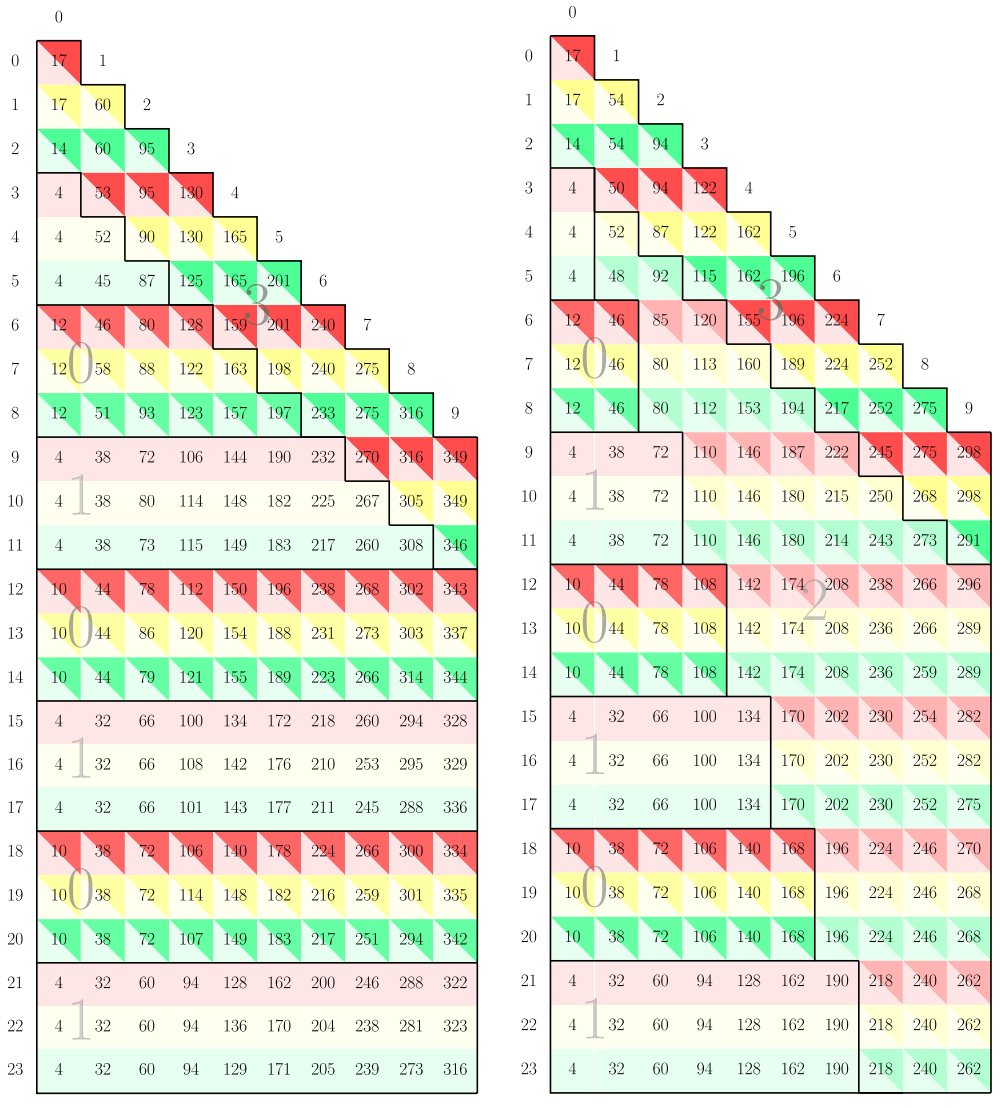


Fig. 7. Steps at which tiles are eliminated.

Each component of an elimination is triggered as soon as possible, i.e. as soon as all dependencies are satisfied: first we have the elimination of the tile, and then the updates in the trailing panels. Note that the overall elimination scheme is complex, and mixes the elimination of tiles at all levels. With a fixed number of resources, it is necessary to decide an order of execution of the tasks, hence to schedule them: this is achieved through the DAGuE environment.

DAGuE is a high-performance fully-distributed scheduling environment for systems of micro-tasks. It takes as input a problem-size-independent, symbolic representation of a Direct Acyclic Graph of tasks, and schedules them at runtime on a distributed parallel machine of multi-cores. Data movements are expressed implicitly by the data flow between the tasks in the DAG representation. The runtime engine is then responsible for actually moving the data from one machine (node) to another, using an underlying communication mechanism, like MPI. A full description of DAGuE, and the implementation of classical linear algebra factorizations in this environment, can be found in [11,8].

To implement the generic QR algorithm in DAGuE, it is sufficient to give an abstract representation of all the tasks (eliminations and updates) that constitute the QR factorization, and how data flows from one task to the other. Since a tiled QR algorithm is fully determined by its elimination list, this basically consists only into providing a function that the runtime engine is capable of evaluating, and that computes this elimination list. The DAGuE object obtained this way is generic: when instantiating a DAGuE QR factorization, the user sets all parameters that define this elimination list (p, q, a, the shape of the local and high-level trees), defining a new DAG at each instantiation. Note that this DAG is not fully generated: DAGuE keeps

only a parametric representation of the DAG in memory, and interprets symbolic expressions at runtime to explicitly represent only the ready tasks at any given time. The technique used is similar to the Parametrized Tasks Graphs [11].

At runtime, task executions trigger data movements, and create new ready tasks, following the dependencies defined by the elimination list. Tasks that are ready to compute are scheduled according to a data-reuse heuristic: each core will try to execute close successors of the last task it ran, under the assumption that these tasks require data that was just touched by the completed one. This policy is tuned by the user through a priority function: among the tasks of a given core, the choice is done following this function. To balance load between the cores, tasks of a same node in the algorithm (reside on a same shared memory machine) are shared between the computing cores, and a NUMA-aware job stealing policy is implemented. The user is responsible for defining the affinity between data and tasks, and to distribute the data between the computing nodes. Thus, he defines which task execute on which node, and remains responsible for this level of load balancing. In our case, the data distribution is a $p \times q$ grid of $b \times b$ tiles, with a cyclic distribution $CYCLIC(1)$ of tiles across both grid dimensions. Since all kernel operations modify a single tile (or a tile and its reflectors, which are distributed the same way), we chose the strategy "owner computes" for the tasks: tasks affinity is set to the node that owns the data that is going to be modified, and the data that must be read might be transferred.

5. Kernel ratios

The HQR algorithm is based on six different kernels referred to as *TT* kernels or *TS* kernels as shown in Algorithm 2. On the one hand, the *TT* kernels provide more parallelism than the *TS* kernels, on the other hand, the sequential performance of the *TT* kernels is lower than the sequential performance of the *TS* kernels.

In this section, we provide formulas to compute the number of these kernels during a factorization. In conjunction with critical path lengths, these formulas enable us to better understand the performance of the HQR algorithm. The formulas depend upon the following parameters: m and n, the matrix size in tiles, p, the number of nodes in the cluster, a, the *TS* domain size, and d, whether the domino optimization is on or off. It is noteworthy to remark that these formula are independent of the elimination tree used at the low-level and at the high-level.

From a qualitative point of view, we expect that: (i) the more a increases, the more *TS* kernels there are; (ii) when the domino optimization is on, there are fewer *TS* kernels than when it is off; and (iii) a higher p leads to fewer *TS* kernels. The goal of this section is to quantify these expectations.

Tile and kernel count in the general case We explained the labeling of the levels in Fig. 6. For each elimination step k (ranging from 0 to $min(m,n)-1$), the number of tiles of each level (0,1,2 or 3) is given by the following formulas:

1. Let $nb_3(k)$ be the number of Level 3 tiles:

$$nb_3(k) = min(p, m-k) \tag{1}$$

2. Let $nb_2(k)$ be the number of Level 2 tiles:

$$nb_2(k) = \begin{cases} min(k(p-1), max(0, m-k-p)) & \text{if domino enabled} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

3. Let $nb_1(k)$ be the number of Level 1 tiles:

$$nb_1(k) = \frac{1}{a} max(0, nb_{12} - nb_{11}(k)) + nb_{1d}(k) + min(p, m - nb_{12}) \tag{3}$$

where

$$\begin{cases} nb_{11}(k) = \begin{cases} \lceil \frac{(k+1)p}{pa} \rceil \times pa & \text{if domino enabled} \\ \lceil \frac{(k+p)}{pa} \rceil \times pa & \text{otherwise} \end{cases} \\ nb_{12} = \lfloor \frac{m}{pa} \rfloor \times pa \\ nb_{1d} = \begin{cases} pa - (k \bmod pa) & \text{if } (k \bmod pa) > (pa - p) \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

To help understand this formula, we have used the following notations:

- $nb_{11}(k)$ is the position of the first Level 1 tile under the diagonal of slope p if the domino optimization is enabled, and under the diagonal of slope 1 if it is disabled
- nb_{12} is the largest multiple of $p \times a$ that does not exceed m
- $nb_{1d}(k)$ is the number of extra Level 1 tiles that lie between the Level 3 tile, and tile $nb_{11}(k)$, when the domino optimization is disabled.

4. Finally, let $nb_0(k)$ be the number of Level 0 tiles:

$$nb_0(k) = m - k - nb_1(k) - nb_2(k) - nb_3(k) \quad (4)$$

In order to obtain the actual number of kernels of each type, the following rules apply:

- Level 3 tiles are first factored, then eliminated by a *TT* kernel by the high-level tree, except the last one, on the diagonal.
- Level 2 tiles are first factored, then eliminated by a *TT* kernel.
- Level 1 tiles are first factored, then eliminated by a *TT* kernel.
- Level 0 tiles are directly eliminated by a *TS* kernel.

Each of these kernels triggers $(n - k - 1)$ corresponding updates on the trailing matrix. These formulas are summarized in Tables 5 and 6. In order to get the total number of tiles or kernels, it is necessary to sum table values over all elimination steps (k ranges from 0 to $min(m, n) - 1$). This final count is provided by Algorithm 3.

Algorithm 3: Kernel count algorithm

Input: m the number of rows in tiles, n the number of columns in tiles, p the number of cluster nodes, a the *TS* domain size, and d the state of the domino optimization.
Output: $GEQRT, TTQRT, TSQRT, UNMQR, TTMQR, TSMQR$

$GEQRT = 0$
$TTQRT = 0$
$TSQRT = 0$
$UNMQR = 0$
$TTMQR = 0$
$TSMQR = 0$
for step $k = 0$ to $min(m, n) - 1$ **do**
  /*First compute the number of tile of each level */
  $nb_3 = min(p, m - k)$
  $nb_1 = \frac{1}{a} max\left(0, \lfloor \frac{m}{pa} \rfloor \times pa - \lceil \frac{(k+1)p}{pa} \rceil \times pa\right) + min\left(p, m - \lfloor \frac{m}{pa} \rfloor \times pa\right)$
  /*Handle the domino optimization */
  if $d == 1$ **then**
    $nb_2 = min(k(p-1), max(0, m - k - p))$
  else
    $nb_2 = 0$
    if $(k \bmod pa) > (pa - p)$ bf then
      $nb_1 = nb_1 + pa - (k \bmod pa)$
  $nb_0 = m - k - nb_1 - nb_2 - nb_3$
  /*Then increment the number of kernels accordingly /*
  $GEQRT = GEQRT + nb_3 + nb_1 + nb_2$
  $UNMQR = UNMQR + (n - k - 1) \times (nb_3 + nb_1 + nb_2)$
  $TSQRT = TSQRT + nb_0$
  $TTQRT = TTQRT + nb_3 - 1 + nb_2 + nb_1$
  $TSMQR = TSMQR + (n - k - 1) \times nb_0$
  $TTMQR = TTMQR + (n - k - 1) \times (nb_3 - 1 + nb_2 + nb_1)$
return $GEQRT, TTQRT, TSQRT, UNMQR, TTMQR, TSMQR$

Exact formulas As an example, we provide closed-form formulas for a tall and skinny matrix, assuming that $m - p > n, m$ divisible by $p, a = 1$ and domino optimization enabled. Results are given in Table 7.

Influence of the domino optimization on the TS/TT ratio Another use of the formulas is shown in Fig. 8(a), where we depict the number of kernels of each level used by HQR for various matrix sizes, with or without the domino optimization. The matrix size varies from a 64×16 tiles matrix to a tall and skinny matrix of $1,024 \times 16$ tiles. The parameter a is set to $a = 4$ and $p = 15$, thus reflecting the values used in Section 6.3. While the domino optimization increases parallelism, see 10(b), this happens to the price of a higher ratio of *TT* kernels vs *TS* kernels, see Fig. 8(b). There is therefore a trade-off between the amount of parallelism needed and the *TS*/*TT* kernel ratio controlled in part by the domino optimization.

In Fig. 8(b), we see that the number of *TT* kernels (blue) is about the same as the number of $GEQRT + UNMQR$ kernels (green). This is always true, no matter what the input parameters of the algorithm $(M, N, a, p,$ domino) are, because, except for the N diagonal tiles (negligible), a *GEQRT* always triggers a *TTQRT*.

Another expected result from Fig. 8(b) is the fact that, when the domino optimization is not activated, for tall and skinny matrices ($M = 286,720$), the number of *TS* kernels (red) is about 60%, while the number of *TT* kernels (green) is about 20%. There is therefore 1 *TT* kernel for 3 *TS* kernels. This makes a lots of sense since $a = 4$.

Table 5
Number of tiles of each level at elimination step k.

Tile count	
Tile level	# tiles
If domino optimization is enabled	
Level 3	$nb_3(k) = min(p, m-k)$
Level 2	$nb_2(k) = min(k(p-1), max(0, m-k-p))$
Level 1	$nb_1(k) = \frac{1}{a} max\left(0, \lfloor \frac{m}{pa} \rfloor \times pa - \lceil \frac{(k+1)p}{pa} \rceil \times pa\right) + min(p, m - \lfloor \frac{m}{pa} \rfloor \times pa)$
Level 0	$nb_0(k) = m - k - nb_1(k) - nb_2(k) - nb_3(k)$
If domino optimization is disabled	
Level 3	$nb_3(k) = min(p, m-k)$
Level 2	$nb_2(k) = 0$
Level 1	$nb_1(k) = \begin{cases} \frac{1}{a} max\left(0, \lfloor \frac{m}{pa} \rfloor \times pa - \lceil \frac{(k+p)}{pa} \rceil \times pa\right) & \\ \quad + pa - (k \bmod pa) & \text{if } (k \bmod pa) > (pa - p) \\ \quad + min\left(p, m - \lfloor \frac{m}{pa} \rfloor \times pa\right) & \\ \frac{1}{a} max\left(0, \lfloor \frac{m}{pa} \rfloor \times pa - \lceil \frac{(k+p)}{pa} \rceil \times pa\right) & \\ \quad + min\left(p, m - \lfloor \frac{m}{pa} \rfloor \times pa\right) & \text{otherwise} \end{cases}$
Level 0	$nb_0(k) = m - k - nb_1(k) - nb_2(k) - nb_3(k)$

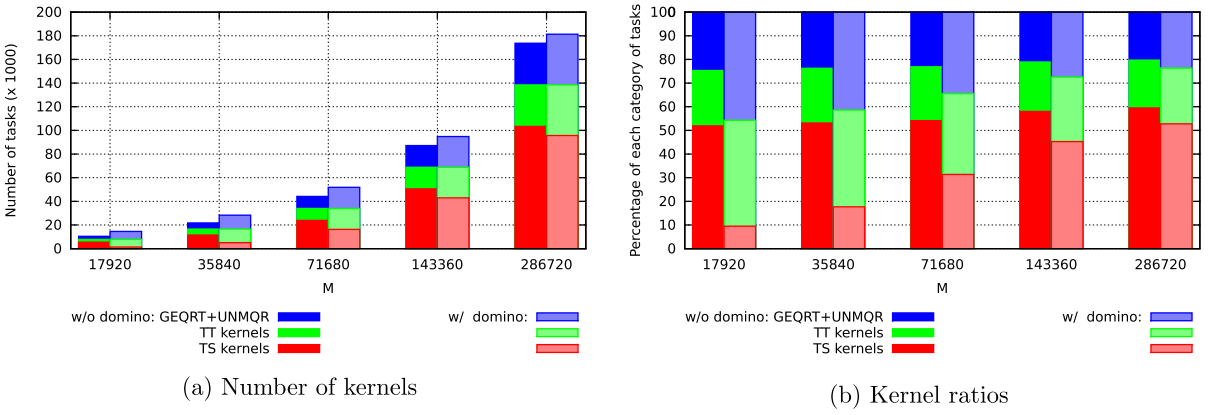
Table 6
Number of kernels of each type at elimination step k.

Kernel count	
Kernel type	# kernels
GEQRT	$nb_3(k) + nb_1(k) + nb_2(k)$
UNMQR	$(n-k-1) \times (nb_3(k) + nb_1(k) + nb_2(k))$
TSQRT	$nb_0(k)$
TSMQR	$(n-k-1) \times nb_0(k)$
TTQRT	$nb_3(k) - 1 + nb_2(k) + nb_1(k)$
TTMQR	$(n-k-1) \times (nb_3(k) - 1 + nb_2(k) + nb_1(k))$

Table 7
Tile and kernel count with $m - p > n, m$ divisible by $p, a = 1$ and domino enabled.

Tile count	
Tile level	# tiles
Level 3	$nb_3 = np$
Level 2	$nb_2 = \sum_{k=0}^{\frac{m}{p}-1} k(p-1) + \sum_{k=\frac{m}{p}}^{n-1} m - p - k = \frac{m+n}{2} + mn - \frac{m^2}{2p} - \frac{n^2}{2} - np$
Level 1	$nb_1 = \frac{m^2}{2p} - \frac{m}{2}$
Level 0	$nb_0 = 0$
Kernel count	
Kernel type	# kernels
GEQRT	$mn - \frac{n^2-n}{2}$
UNMQR	$\frac{(m+1)n^2}{2} - \frac{n^3+3mn+2n}{6}$
TTQRT	$mn - \frac{n^2+n}{2}$
TTMQR	$\frac{mn^2-mn}{2} - \frac{n^3-n}{6}$
TSQRT	0
TSMQR	0

The domino optimization considerably decreases the number of *TS* kernels in the square case. The effect is negligible in the tall and skinny case. In the square case ($M = 17,920$), we see in Fig. 8(b) that there is about 10% of *TS* kernels when the domino optimization is on. This contrasts quite significantly with the 50% obtained when the domino optimization is off. We have performed numerical simulation. Increasing the a value does increase the number of *TS* kernels (as expected) but the improvement is negligible. Indeed, for $p = 15$ (our setup), square matrices, and domino on, it turns out that no matter how large $M = N = a$ are, there will be at most 10% of *TS* kernels during the whole factorization. If the domino is off, for $p = 15$, the ratio of *TS* kernels tends to 100% when $M = N = a$ goes to infinity. For $p = 30$, there is at most 5% of *TS* kernels when the domino is on for square matrices. For $p = 60$, there is at most 2.5% of *TS* kernels when the domino is on for square matrices.



(a) Number of kernels

(b) Kernel ratios

Fig. 8. Number of kernels of each type used by HQR.

6. Experiments

6.1. Experimental conditions

The purpose of this performance evaluation is to highlight the features of HQR, and to compare its efficiency with state-of-the-art QR factorization implementations. We use *edel*, a parallel machine hosted by the Grid'5000 experimental platform [21], to support the experiments. These experiments feature 60 multi-core machines, each equipped with 8 cores, and an Infiniband 20 G interconnection network. The machines feature two NUMA Nehalem Xeon E5520 at 2.27 GHz (hyperthreading is disabled), with 12 GB of memory (24 GB per machine). The system is running the Linux 64 bit operating system, version 2.6.32-5-amd64 (Debian 2.6.32-35). The software is compiled with Gcc version 4.4.5, and GFortran 4.4.5 when applicable. The sequential version of the BLAS kernels were provided by the MKL library from the Intel compiler suite 11.1. The DAGuE software from the mercurial repository revision 3130 uses Open MPI version 1.4.3 as network backend. It uses the pthread library to schedule each kernel on all the cores of each nodes. All experiments have been run at least 5 times, and the average value is presented, together with the standard deviation. We use *whiskers* to represent standard deviation on all of our figures. For each experiment, we compute the Q factor of the QR factorization (by applying the reverse trees to the identity) and check (a) that Q has orthonormal columns and (b) that A is equal to $Q \times R$. All checks were satisfactory up to machine precision.

The theoretical peak performance of this machine for double-precision is 9.08 GFlop/s per core, 72.64 GFlop/s per node, and 4.358 TFlop/s for the whole machine. The best performance for running the dTSMQR operation in a single core, has been measured at 7.21 GFlop/s (79.4% of the theoretical peak), and the dTTMQR operation has been measured at 6.28 GFlop/s (69.2% of the theoretical peak). Depending on the a value chosen, these numbers can be seen as practical peaks. For example, if $a = 1$, most of the flops are in dTTMQR (69.2% of the theoretical peak). As a gets larger, more flops shift to dTSMQR (79.4% of the theoretical peak).

Our implementation of HQR operates on a virtual grid $p \times q$ set to 15×4, it features a *TS* level with parameter a (set a to 1 for no *TS*, and $a = m/p$ for full *TS* on the node), a choice of four different *TT* trees for the low-level (Greedy, BinaryTree, FlatTree, Fibonacci), the domino optimization can be activated or not. When it is activated, the domino *TT* tree is used by default, and there is a choice of four different *TT* trees for the high-level (Greedy, BinaryTree, FlatTree, Fibonacci). Tiles of size $b \times b$ are used. The DAGuE engine offers several data distributions and automatically handles the data transfers when needed. As a consequence, our DAGuE implementation would operate on any DAGuE-supported data distribution. For HQR, we focus on 2D block cyclic distribution using a $p \times q$ process grid mapping the algorithm virtual grid.

We compare our algorithm to [BBD + 10] [8], [SLHD10] [2], and ScaLAPACK [10]. Since [SLHD10] is a sub-case of the HQR algorithm (see Section 4.1), we use our DAGuE-based implementation of HQR to execute it. [SLHD10] for a $m \times n$ tiled matrix, using a block distribution on p nodes, corresponds to the HQR algorithm with the following parameters: virtual grid value $p = 1$, domains of size $a = m/p$, data distribution $CYCLIC(a)$, low-level binary tree. (Since $p = 1$, neither the domino level nor the high-level are relevant.) [BBD + 10] corresponds to the QR operation currently available in DAGuE, which implements the Tile QR factorization described in [8]. ScaLAPACK experiments use the ScaLAPACK implementation of the QR factorization found in the MKL libraries. The MKL number of threads was set to 8, and one MPI process was launched per node. For all other setups (that are DAGuE based), the binary was linked with the sequential version of the MKL library, and DAGuE was launched with 8 computing threads and an additional communication thread per node. All threads are bound to a different core, except the communication thread that is allowed to run on any core. In ScaLAPACK, the algorithmic block size used is the same as the data distribution block size. We set it to $b = 280$ since it consistently provide good performance.

In all experiments, we used 60 nodes (480 cores), and the data was distributed along a 15×4 process grid for HQR, [BDD + 10], and ScaLAPACK, and a 60×1 1D block distribution for [SLHD10].

All HQR runs use a *virtual* node grid exactly mapping the *process* grid used for data distribution. The domino optimization, whenever activated, is implemented with the domino scheme. We fix the tile size parameter b in our experiments as being the block size which renders the best sequential performance for the sequential *TS* update kernel. As for ScaLAPACK, the algorithmic block size used by HQR is the same as the data distribution block size. In the context of HQR, we called it the tile size. More tuning could be done for HQR with respect to the tile size and to the process grid shape parameters. In particular, b directly influences at least two key performance metrics, namely the number of messages sent and the granularity of the algorithm. We have fixed these parameters for the whole experiment set. Choosing $b = 280$ and a process grid $p \times q$ of 15×4 leads to values that consistently provides good performance.

6.2. Evaluation of HQR

HQR is a highly modular algorithm. The design space offered by its parameters is large. The goal of this section is to confront our intuition of HQR with critical path lengths as well as experimental data in order to build up understanding on how these parameters influence the overall performance of HQR. In Section 6.3, we use this newly acquired understanding to set up the parameters for various fixed-parameters experiments. We note that, overall, HQR is an intrinsically better algorithm than what has been proposed in the past. Although we explain in this section that some significant critical path length improvements and experimental performance gains can be obtained by tuning the parameters, setting some default values is enough to outperform the current state of the art.

Experimental results are obtained on the platform described in 6.1 while critical paths are obtained through simulation on a dedicated simulator. This simulator is also based on the DAGuE scheduler and task weights are set according to the tiled model, as in [1]. Note that in all the following, critical path lengths are given with a unit of $\frac{b^3}{3}$, where b is the unit block size.

Fig. 9 presents the performance of HQR, for different matrix sizes, different trees and different values of the a parameter. The matrix size varies from a square matrix of 16×16 tiles to a tall and skinny matrix of $1,024 \times 16$ tiles. Since we are working on a 15×4 process grid, this means that local matrices range from 1×4 tiles to 68×4 tiles. In order to first focus only on the influence of the *TS* level, low-level and high-level trees, the domino optimization is not yet activated. Figs. 9((a) and (c)) present the critical path lengths and performance for all possible high-level trees with a low-level tree set to Greedy, while Figs. 9((b) and (d)) present the same with a low-level tree set to FlatTree. Figures with a low-level tree set to BinaryTree or Fibonacci are omitted due to lack of space; however they exhibit a behavior similar to Fig. 9(a) (Greedy). Figs. 10((a) and (b)) present the performance of the HQR algorithm, for the same set of matrices, with a fixed value $a = 4$, and a high-level tree set to Fibonacci. Measurements were done alternatively turning on or off the domino optimization.

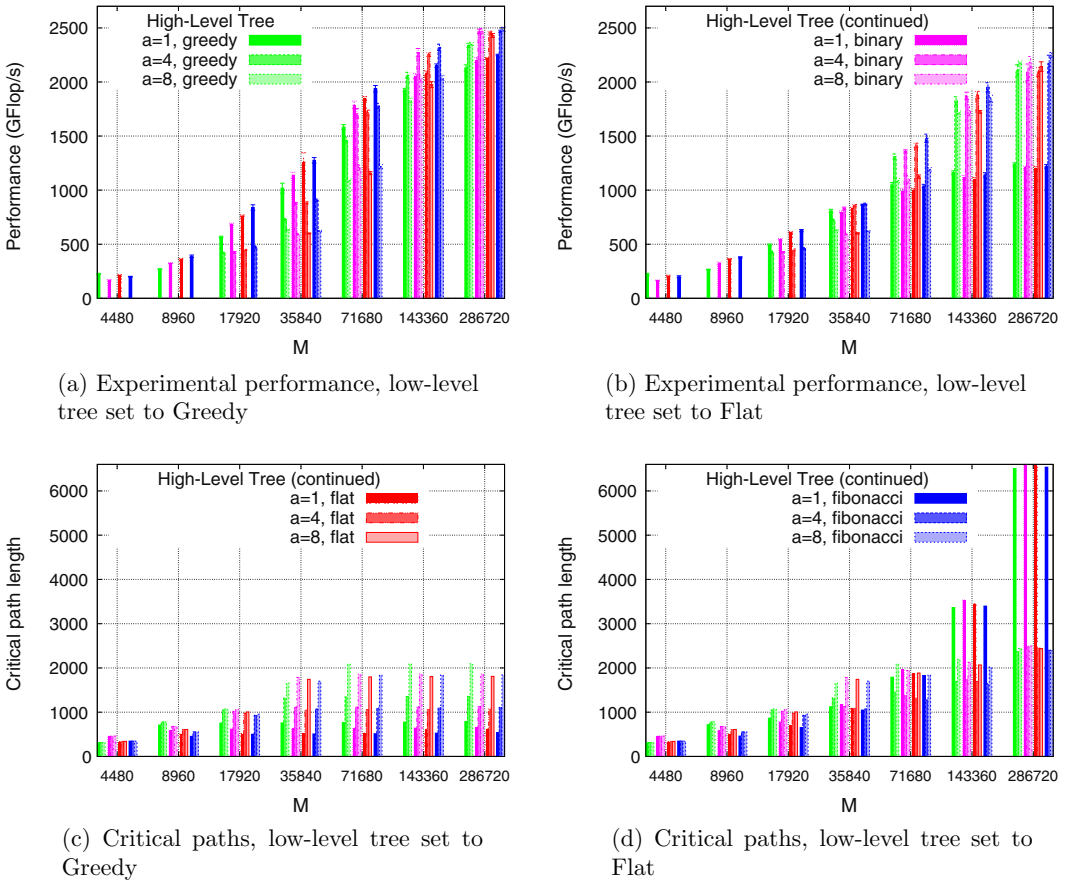
Influence of a. Looking at Fig. 9(a), we see that, for small values of M, the value $a = 1$ is best. Furthermore, this is confirmed by Fig. 9(c), while in the same conditions, the value $a = 1$ offers the shortest critical paths. This is because a higher value of a negatively impacts the degree of parallelism of the algorithm when we use the low-level Greedy tree, deeply impacting performance on small matrices. When M increases, the number of tasks increases, ending up with enough parallelism to keep the platform busy. Consequently, we can safely increase the value of a up to 4 or 8. While such values increase critical path lengths, the higher individual performance of *TS* update kernels alleviate this trend. Therefore, for large M, we see that the speedup between $a = 1$ and $a = 4, 8$ is about 10% which is the speedup between *TT* update kernels and *TS* update kernels. When the low-level tree is FlatTree, (Figs. 9((b) and (d)), we have a different story. Adding a flat tree (*TS* kernels) beneath a low-level flat tree in the tall and skinny case (large M) actually increases the parallelism. In effect, the *TS* flat trees divide the length of the pipeline created by the low-level flat tree by a factor a. So there are two benefits for tall and skinny matrices in adding a flat tree *TS* beneath a flat tree *TT*: (1) faster kernels; (2) better parallelism, thus shortest critical paths. This explains why the speedup for $a = 4$ or $a = 8$ with respect to $a = 1$ is way above 10% for large M. Altogether, we conclude that significant gain can be obtain by tuning the parameter a for various matrix shapes, number of nodes and *TT* vs *TS* ratio.

Influence of the low-level tree. For tall and skinny matrices, Greedy is better than FlatTree. In the $286,720 \times 4,480$ case, the low-level tree performs on a 68×16 matrix ($m/p \times n$), and in that case the critical path length of FlatTree is approximately 2.6x the one of Greedy ($((68 + 2 \times 16)/(\log_2(68) + 2 \times 16))$ [1]). Looking at Figs. 9((a) and (b)), we see a speedup of about 2x when the low-level tree changes from FlatTree to Greedy in the $a = 1$ case. When a increases, the low-level trees affect fewer tiles and, consequently, its influence on the overall algorithm is reduced. See also Fig. 10(a), where we have set $a = 4$, and we observe that all low-level trees perform more or less similarly.

Influence of the high-level tree. We observe similar performance for all variants, although Fibonacci is slightly better than its competitors. The same holds for critical path lengths, as they are not much impacted by the high-level tree.

Influence of domino optimization. In Fig. 10(b), we see the positive effect of the domino optimization in all cases. When activated, it significantly reduces the critical path whatever low-level tree is used. As for the experimental performance, Fig. 10(a) shows the positive effect of the domino optimization in the case of tall and skinny matrices. When activated, for a tall and skinny matrices, it never significantly deteriorates the performance and can have significant impact.

The domino optimization is all the more important when a good coupling between the local tree and the distributed tree is critical. This is illustrated best with the case of low-level FlatTree. In such a case, the domino optimization has a tremendous impact on critical path lengths since it introduces more parallelism by reducing the length of the flat trees. Moreover, this optimization enables look-ahead on the local panels as explained in Section 4.2, thereby further increasing the degree of parallelism.



(a) Experimental performance, low-level tree set to Greedy

(b) Experimental performance, low-level tree set to Flat

(c) Critical paths, low-level tree set to Greedy

(d) Critical paths, low-level tree set to Flat

Fig. 9. Experimental performance and critical path lengths of the HQR algorithm on a $M \times 4,480$ matrix. (Domino optimization not activated). Influence of the *TS* level (a value), low-level and high-level trees.

Although not reported in this manuscript, we note that domino optimization have a negative impact when the matrix becomes large and square. As a matter of a fact, in such cases and despite the fact that critical path lengths are actually shorter with the domino optimization, the number of *TS* update kernels is heavily reduced, thereby leading to lower experimental performance.

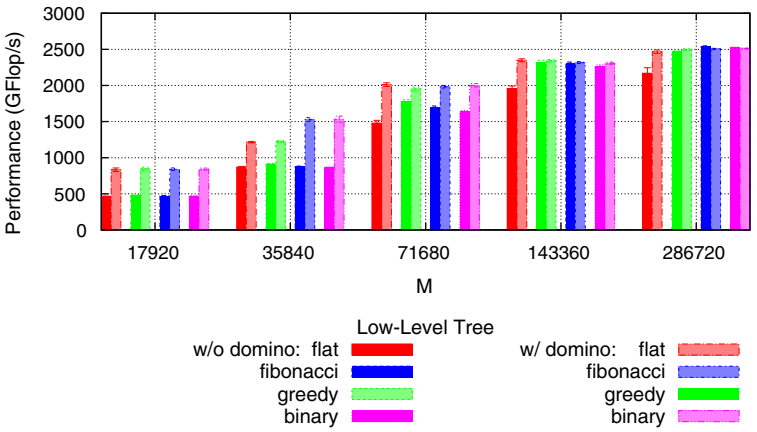
6.3. Comparison

Figs. 11 and 12 compare the performance of the DAGuE implementation of the HQR algorithm with the DAGuE implementation of [BBD + 10] and [SLHD10], and with the MKL implementation of the ScaLAPACK algorithm.

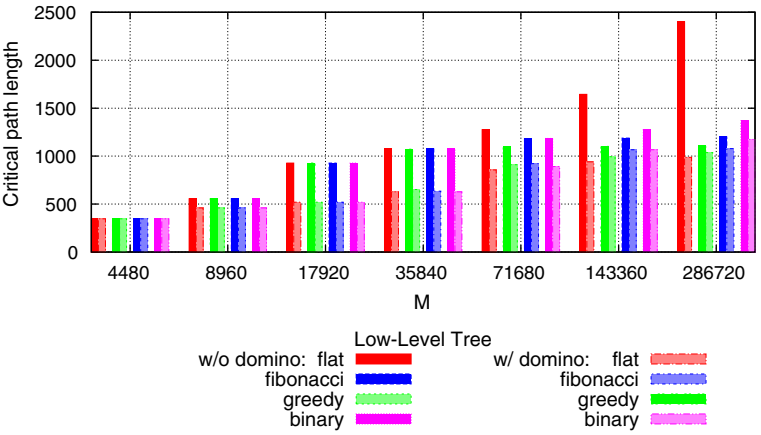
N fixed, M varies from square to tall and skinny. In Fig. 11, we evaluate the performance on various matrices, from a square 16×16 tiled matrix to a tall and skinny $1,024 \times 16$ tiled matrix. This is the same matrix set as in Figs. 9 and 10(a). We need low and high-level trees adapted to tall and skinny matrices, so we set both level trees to Fibonacci. The *TS* level trades off some parallelism in the intra-level reduction to enable the use of *TS* sequential kernels which are more efficient than the *TT* sequential kernels. Since, in this experiment, the local matrices have a large number of rows with respect to the number of cores on the node, there is enough intra-node parallelism within a column reduction to afford a *TS* level, so we set $a = 4$. Finally in the tall and skinny case, we really want to decouple the low-level tree from the inter-node communication, so we activate the domino optimization. HQR scores 2,505 GFlop/s (57.5% of peak).

The algorithm in ScaLAPACK is not "tiled", so it is not "communication avoiding". The algorithm performs one parallel distributed reduction per column, this contrasts with a tiled algorithm which performs one parallel distributed reduction per tile. As a consequence, there is a factor of b in the latency term between both algorithms. For a tall and skinny matrix, the algorithm in ScaLAPACK is indeed not compute-bounded but latency-bounded and obtains at best 277 GFlop/s (6.4% of peak).

The main performance bottleneck for [BDD + 10] is the use of FlatTree. FlatTree has a long start-up time to initiate the first column, it operates sequentially on the tiles along the first tile column so that there are as many *TS* kernels pipelined the one after the other as there are tiles in a column (that is m,e.g., =1024 in the largest example considered here). This is not suitable



(a) Performance of the HQR algorithm



(b) Critical paths of the HQR algorithm

Fig. 10. Experimental performance and critical path lengths of the HQR algorithm, on a $M \times 4,480$ matrix. High-level set to Fibonacci and $a = 4$. Influence of the low-level tree and of the domino optimization.

when there are only $n = 16$ tile columns to amortize the pipeline startup cost. Another issue with [BDD + 10] is that the algorithm does not take into account the 2D block cyclic distribution of the data. This has a secondary negative impact on the performance. For a tall and skinny matrix, the algorithm in [BDD + 10] suffers from a long pipeline on the first tile column. The length of this pipeline is m, the number of row tiles, or the whole matrix. The algorithms scores at best 798 GFlop/s (18.3% of peak).[SLHD10] has been specially designed for tall and skinny matrices [2]. The negative load imbalance that occurs by using a 1D block data distribution instead of a 2D block cyclic distribution is not significant for tall and skinny matrices. At the inter-node level, the use of BinaryTree is a good solution. Yet, the use of *TS* FlatTree at the intra-node level is not appropriate when the local matrices have many rows. As in [BDD + 10], a long pipeline is instantiated. A better tree is needed at the intra-node level. For a tall and skinny matrix, the algorithm in [SLHD10] suffers from a long pipeline on the first tile column. The length of this pipeline is m/p, the number of row tiles held by a node (which is an improvement with respect to [BDD + 10] but yet too much). The algorithms scores at best 1,897 GFlop/s (43.5% of peak).

M fixed, N varies from tall-skinny to square. In Fig. 12, we evaluate the performance from a tall and skinny 240×4 tiles matrix to a square 240×240 tiles matrix. The high-level tree is set to FlatTree, while the low-level tree is set to Fibonacci. Depending on the value of N, we choose different values for $a : a = 1$ for small values of N, and $a = 4$ for larger values. Similarly, the domino optimization is de-activated once the parallelism due to the number of columns of tiles is sufficient enough to avoid starvation, and the efficiency of the kernels becomes more important. The choice of the FlatTree high-level tree is guided by the same reason: once the parallelism is high enough to avoid starvation, the FlatTree ensures a significantly smaller number of inter-node communications.

[BDD + 10] performs well on square matrices, however it suffers from its more demanding communication pattern than the HQR algorithm (since it does not take into account the 2D block cyclic distribution of the data). [SLHD10] performs better on tall and skinny matrices, however the 1D data distribution implies a load imbalance that becomes paramount when the

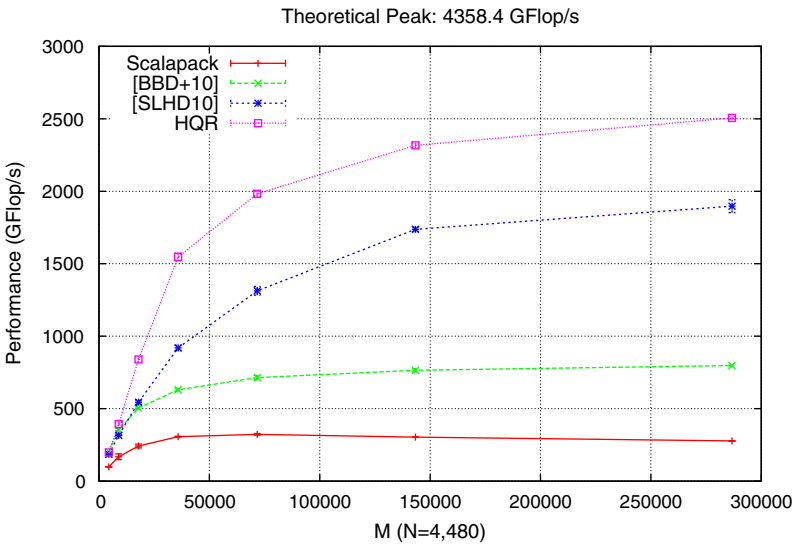


Fig. 11. Comparison of performance for different algorithms, on a $M \times 4,480$ matrix.

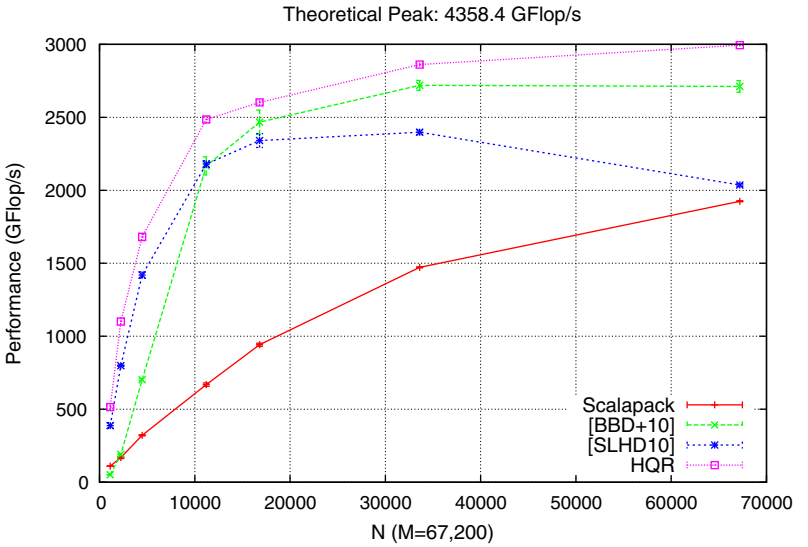


Fig. 12. Comparison of performance for different algorithms, on a $67,200 \times N$ matrix.

matrix becomes square. This is illustrated by the ratio of performance between HQR and [SLHD10]: on the square matrix, HQR reaches 3 TFlop/s, while [SLHD10] reaches 2 TFlop/s, thus 2/3 of the performance, as predicted in Section 3.3. Likewise, when $N = M/2$, [SLHD10] reaches 2.4 TFlop/s, and HQR 2.9 TFlop/s, and $2.4/2.9 \approx 5/6$, as predicted by the model. Although the performance of ScaLAPACK is lagging behind the performance of the other tile based algorithms, ScaLAPACK builds performance as M increases and score a respecTable 1,925 GFlop/s (44.2% of peak) on a square matrix.

7. Conclusion

We have presented HQR, a hierarchical QR factorization algorithm which introduces several innovative components to squeeze the most out of clusters of multi-cores. On the algorithmic side, we have designed a fully flexible algorithm, whose many levels of tree reduction each significantly contributes to improving state-of-the-art algorithms. A key feature is that the high-level specification of the algorithm makes it suitable to an automated implementation with the DAGuE framework. This greatly alleviates the burden of the programmer who faces the complex and concurrent programing environments required for massively parallel distributed-memory machines.

On the experimental side, our algorithm dramatically outperforms all competitors, which can be seen as a major achievement given (i) the ubiquity of QR factorization in many application domains; and (ii) the vast amount of efforts that have been recently devoted to numerical linear algebra kernels for petascale and exascale machines. Our implementation of the new algorithm with the DAGuE scheduling tool significantly outperforms currently available QR factorization software for all matrix shapes, thereby bringing a new advance in numerical linear algebra for petascale and exascale platforms. More specifically, our experiments on the Grid'5000 edel platform show the following gains at both ends of the matrix shape spectrum:

- On tall and skinny matrices, we reach 57.5% of theoretical computational peak performance, to be compared with 6.4% for ScaLAPACK (9.0x speedup), 18.3% for [BDD + 10] (3.1x), and 43.5% for [SLHD10] (1.3x)
- On square matrices, we reach 68.7% of theoretical computational peak performance, to be compared with 44.2% for ScaLAPACK (1.6x), 62.2% for [BDD + 10] (1.1x), and 46.7% for [SLHD10] (1.5x).

Future work includes several promising directions. From a theoretical perspective, we could use critical paths to assess priorities to the different elimination trees. This is a very promising but technically challenging direction, because it is not clear how to account for the different architectural costs, and because of the huge parameter space to explore. From a more practical perspective, we could perform further experiments on machines equipped with accelerators (such as GPUs): again, the flexibility of the DAGuE software will dramatically ease the design of HQR on such platforms, and will enable us to explore a wide combination of reduction trees and priority settings.

Acknowledgements

The authors thank the reviewers for their numerous comments and suggestions, which greatly improved the final version of the paper.

References

[1] H. Bouwmeester, M. Jacquelin, J. Langou, Y. Robert, Tiled QR factorization algorithms, The IEEE/ACM Conference on High Performance Computing Networking, Storage and Analysis SC'2011, ACM Press, 2011.
[2] F. Song, H. Ltaief, B. Hadri, J. Dongarra, Scalable tile communication-avoiding QR factorization on multicore cluster systems, The 2010 ACM/IEEE Conference on Supercomputing SC'10, IEEE Computer Society Press, 2010.
[3] E. Agullo, C. Coti, J. Dongarra, T. Herault, J. Langou, QR factorization of tall and skinny matrices in a grid computing environment, in: IPDPS'10, the 24st IEEE International Parallel and Distributed Processing Symposium, 2010.
[4] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, Parallel tiled QR factorization for multicore architectures, Concurrency: Practice and Experience 20 (13) (2008) 1573–1590.
[5] G. Quintana-Ortí, E.S. Quintana-Ortí, R.A. van de Geijn, F.G.V. Zee, E. Chan, Programming matrix algorithms-by-blocks for thread-level parallelism, ACM Transactions on Mathematical Software 36 (3) (2009).
[6] J.W. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-avoiding parallel and sequential QR and LU factorizations: theory and practice, LAPACK working note, Tech. Rep. 204, 2008. Available: <http://www.netlib.org/lapack/lawnspdf/lawn204.pdf>.
[7] B. Hadri, H. Ltaief, E. Agullo, J. Dongarra, Tile QR factorization with parallel panel processing for multicore architectures, in: IPDPS'10, the 24st IEEE International Parallel and Distributed Processing Symposium, 2010.
[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, J. Dongarra, Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA, in: 12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'11), 2011.
[9] S. Blackford, J.J. Dongarra, Installation guide for LAPACK, LAPACK Working Note, Tech. Rep. 41, Jun. 1999, originally released March 1992. Available: <http://www.netlib.org/lapack/lawnspdf/lawn41.pdf>.
[10] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, SIAM, 1997.
[11] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, DAGuE: A generic distributed DAG engine for high performance computing, in: 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11), 2011.
[12] M. Cosnard, J.-M. Muller, Y. Robert, Parallel QR decomposition of a rectangular matrix, Numerische Mathematik 48 (1986) 239–249.
[13] M. Cosnard, Y. Robert, Complexity of parallel QR factorization, Journal of the ACM 33 (4) (1986) 712–723.
[14] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Computing 35 (2009) 38–53.
[15] A. Sameh, D. Kuck, On stable parallel linear systems solvers, Journal of the ACM 25 (1978) 81–91.
[16] J. Modi, M. Clarke, An alternative Givens ordering, Numerische Mathematik 43 (1984) 83–90.
[17] A. Pothen, P. Raghavan, Distributed orthogonal factorization: givens and householder algorithms, SIAM Journal on Scientific Computing 10 (6) (1989) 1113–1134.
[18] R. da Cunha, D. Becker, J. Patterson, New parallel (rank-revealing) QR factorization algorithms, in: Euro-Par 2002, Parallel Processing: Eighth International Euro-Par Conference, Paderborn, Germany, August 27–30, 2002.
[19] J. Langou, Computing the R of the QR factorization of tall and skinny matrices using MPI_Reduce, Tech. Rep. <arXiv:1002.4250>, 2010.
[20] J. Demmel, M. Hoemmen, M. Mohiyuddin, K. Yelick, Minimizing communication in sparse matrix solvers, The ACM/IEEE Conference on Supercomputing SC'09, IEEE Computer Society Press, 2009, pp. 1–12.
[21] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jegou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, G. Mornet, Grid'5000: a large scale, reconfigurable, controlable and monitorable grid platform, Proc. 6th IEEE/ACM Int. Workshop on Grid Computing (Grid'2005), IEEE Computer Society Press, 2005.