Tall and Skinny QR Matrix Factorization Using Tile Algorithms on Multicore Architectures LAPACK Working Note - 222

Bilel Hadri[1], Hatem Ltaief[1], Emmanuel Agullo[1], and Jack Dongarra[1,2,3⋆]

[1] Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee
[3] School of Mathematics & School of Computer Science, University of Manchester
{hadri, ltaief, eagullo, dongarra}@eecs.utk.edu

Abstract. To exploit the potential of multicore architectures, recent dense linear algebra libraries have used tile algorithms, which consist in scheduling a Directed Acyclic Graph (DAG) of tasks of fine granularity where nodes represent tasks, either panel factorization or update of a block-column, and edges represent dependencies among them. Although past approaches already achieve high performance on moderate and large square matrices, their way of processing a panel in sequence leads to limited performance when factorizing tall and skinny matrices or small square matrices. We present a fully asynchronous method for computing a QR factorization on shared-memory multicore architectures that overcomes this bottleneck. Our contribution is to adapt an existing algorithm that performs a panel factorization in parallel (named Communication-Avoiding QR and initially designed for distributed-memory machines), to the context of tile algorithms using asynchronous computations. An experimental study shows significant improvement (up to almost 10 times faster) compared to state-of-the-art approaches. We aim to eventually incorporate this work into the Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA) library.

1 Introduction and Motivations

QR factorization is one of the major one-sided factorizations in dense linear algebra. Based on orthogonal transformations, this method is well known to be numerically stable and is a first step toward the resolution of least square systems [11]. We have recently developed a parallel tile QR factorization [7] as part of the Parallel Linear Algebra Software for Multi-core Architectures (PLASMA) project [8]. Tile algorithms in general provide fine granularity parallelism and

⋆ Research reported here was partially supported by the National Science Foundation and Microsoft Research.

standard linear algebra algorithms can then be represented as a Directed Acyclic Graph (DAG) where nodes represent tasks, either panel factorization or update of a block-column, and edges represent dependencies among them.

PLASMA Tile QR has been benchmarked on two architectures [4], a quad-socket quad-core machine based on an Intel Xeon processor and a SMP node composed of 16 dual-core Power6 processors. Table 1 and 2 report the parallel efficiency (the quotient of the division of the time spent in serial by the product of the time spent in parallel and the number of cores used) achieved with different matrix sizes on each architecture. PLASMA Tile QR scales fairly well for large

Table 1. Parallel efficiency on Intel

	Number of cores			
Matrix order	2	4	8	16
500	69%	55%	**39%**	**24%**
1000	88%	73%	60%	**45%**
2000	97%	91%	81%	69%
4000	98%	97%	94%	84%

Table 2. Parallel efficiency on Power6

	Number of cores				
Matrix order	2	4	8	16	32
500	72%	**43%**	**25%**	**12%**	**6%**
1000	80%	67%	**46%**	**24%**	**12%**
2000	92%	80%	65%	**46%**	**25%**
4000	95%	90%	79%	71%	51%

square matrices and up to the maximum number of cores available on those shared-memory machines, 16 and 32 cores on Intel and Power6, respectively. However, for small matrices, the parallel efficiency significantly decreases when the number of cores increases. For example, for matrix sizes lower than 1000, the efficiency is roughly at most 50% on Intel and Power6 with 16 cores. And this declines on Power6 with only a 6% parallel efficiency achieved on 32 cores with a matrix size of 500. This low efficiency is mainly due to the sequential factorization of panels and is expected to be even lower when dealing with tall and skinny (TS) matrices (of size $m-$by$-n$ with $m >> n$) where a large proportion of the elapsed time is spent in those sequential panel factorizations.

The purpose of this paper is to present a fully asynchronous method to compute a QR factorization of tall and skinny matrices on shared-memory multicore architectures. This new technique finds its root in combining the core concepts from the Tile QR factorization implemented in the PLASMA library and the Communication-Avoiding QR (CAQR) [9] algorithm introduced by Demmel et al. Initially designed for distributed-memory machines, CAQR factors general rectangular distributed matrices with a parallel panel factorization. Even if the present paper discusses algorithms for shared-memory machines where communications are not explicit, multicore platforms often symbolize, at a smaller scale, a distributed-memory environment with a design of a memory and/or cache hierarchy to take advantage of memory locality in computer programs. Hence the relevance of using algorithms that limit the amount of communication in our context too.

This paper is organized as follows. Section 2 presents the background work. Section 3 describes two new approaches that combine algorithmic ideas from tile algorithms and the communication avoiding approach. Section 4 explains how

the tasks from the resulting DAG are scheduled in parallel. In Section 5, we present a performance analysis of the method and a comparative performance evaluation against existing numerical libraries. Finally, in Section 6, we conclude and present future work directions.

2 Background

TS matrices are present in a variety of applications in linear algebra, e.g., in solving linear systems with multiple right-hand sides using block iterative methods by computing the QR factorization of a TS matrix [10, 16]. But above all, TS matrices show up at each panel factorization step while performing one-sided factorization algorithms (QR, LU and Cholesky). The implementation of efficient algorithms to handle such matrix shapes is paramount. In this section, we describe different algorithms for the QR factorization of TS matrices implemented in the state of the art numerical linear algebra libraries.

2.1 LAPACK/ScaLAPACK QR factorization

QR factorization of an $m \times n$ real matrix A has the form $A = QR$, where Q is generally an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix. QR factorization applies a series of elementary Householder matrices of the general form $H = I - \tau v v^T$ where v is a column reflector and τ is a scaling factor.

Regarding the block or block-partitioned algorithms in LAPACK [5] or ScaLAPACK [6], nb elementary Householder matrices are accumulated within each panel and the product is represented as $H_1 H_2 ... H_{nb} = I - VTV^T$. Here V is a $n \times nb$ matrix in which columns are the vectors v, T is a $nb \times nb$ upper triangular matrix and nb is the block size.

Although the panel factorization is characterized by the presence of a sequential operation that represents a small fraction of the total number of FLOPS performed ($\theta(n^2)$) FLOPS for a total of $\theta(n^3)$) FLOPS), the scalability of block factorizations is limited on a multicore system when parallelism is only exploited at the level of the BLAS or PBLAS routines, for LAPACK and ScaLAPACK respectively. This approach will be referred to as the fork-join approach since the execution flow of a block factorization would show a sequence of sequential operations (i.e., the panel factorizations) interleaved to parallel ones (i.e., the trailing submatrix updates).

2.2 Tile QR factorization (PLASMA-like factorization)

PLASMA Tile QR factorization [7, 8] is a derivative of the block algorithms that produces high performance implementations for multicore architectures. The algorithm is based on the idea of annihilating matrix elements by square tiles instead of rectangular panels as in LAPACK. PLASMA Tile QR algorithm relies on four basic operations implemented by four computational kernels:

– **CORE_DGEQRT**: this kernel performs the QR factorization of a diagonal tile A_{kk} of size $nb \times nb$ of the input matrix. It produces an upper triangular matrix R_{kk}, a unit lower triangular matrix V_{kk} containing the Householder reflectors and an upper triangular matrix T_{kk} as defined by the WY technique [18] for accumulating the transformations. R_{kk} and V_{kk} are written on the memory area used for A_{kk} while an extra work space is needed to store T_{kk}. The upper triangular matrix R_{kk}, called *reference tile*, is eventually used to annihilate the subsequent tiles located below, on the same panel.
– **CORE_DTSQRT**: this kernel performs the QR factorization of a matrix built by coupling the reference tile R_{kk} that is produced by CORE_DGEQRT with a tile below the diagonal A_{ik}. It produces an updated R_{kk} factor, V_{ik} matrix containing the Householder reflectors and the matrix T_{ik} resulting from accumulating the reflectors V_{ik}.
– **CORE_DORMQR**: this kernel is used to apply the transformations computed by CORE_DGEQRT (V_{kk}, T_{kk}) to a tile A_{kj} located on the right side of the diagonal tile.
– **CORE_DTSSSMQR**: this kernel applies the reflectors V_{ik} and the matrix T_{ik} computed by CORE_DTSQRT to two tiles A_{kj} and A_{ij}.

Since the tile QR is also based on Householder reflectors that are orthogonal transformations, this factorization is stable. Figure 1 shows the first panel reduction applied on a 3-by-3 tile matrix. The triangular shapes located on the left side of the matrices correspond to the extra data structure needed to store the T. The striped tiles represent the input dependencies for the trailing submatrix updates. The algorithm for general matrices, with MT tiles in row and NT tiles in column, is formulated in Algorithm 1 [8]. As of today, PLASMA

Algorithm 1 Tile QR factorization (PLASMA-like factorization)

for $k = 1$ to $min(MT, NT)$ **do**
  $R_{k,k}, V_{k,k}, T_{k,k} \leftarrow$ CORE_DGEQRT($A_{k,k}$)
  for $j = k + 1$ to NT **do**
    $A_{k,j} \leftarrow$ CORE_DORMQR($V_{k,k}, T_{k,k}, A_{k,j}$)
  end for
  for $i = k + 1$ to MT **do**
    $R_{k,k}, V_{i,k}, T_{i,k} \leftarrow$ CORE_DTSQRT($R_{k,k}, A_{i,k}$)
    for $j = k + 1$ to NT **do**
      $A_{k,j}, A_{i,j} \leftarrow$ CORE_DTSSSMQR($V_{i,k}, T_{i,k}, A_{k,j}, A_{i,j}$)
    end for
  end for
end for

implements Algorithm 1 through a given framework based on a static scheduling and discussed later in Section 4.1. In the rest of the paper, we will use the term *PLASMA-like factorization* to refer to any factorization based on Algorithm 1,

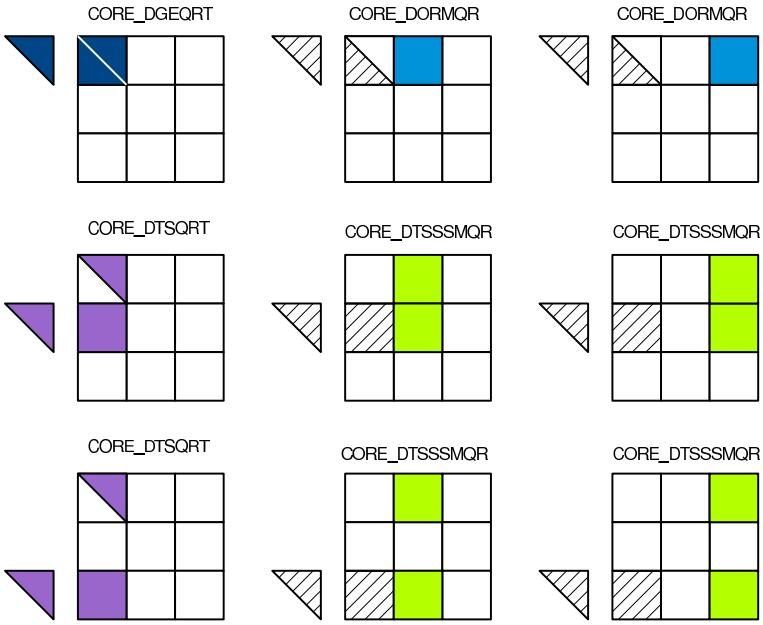

Fig. 1. Reduction of the first tile column.

without regard to the framework implementing it nor the scheduling mechanism used.

Although PLASMA achieves high performance on most types of matrices by implementing Algorithm 1 [4], each panel factorization is still performed in sequence, which limits the performance when processing small or TS matrices (see results reported in Section 1).

2.3 Parallel Panel Factorizations

The idea of parallelizing the factorization of a panel was first presented by Pothen and Raghavan, to the best of our knowledge, in 1988 [17]. The authors implemented distributed orthogonal factorizations using Householder and Givens algorithms. Each panel is actually composed of one single column in their case. Their idea is to split the column into P pieces or subcolumns (if P is the number of processors) and to perform local factorizations from which they merge the resulting triangular factors, as explained in Algorithm 2.

Demmel et al. [9] extended this work and proposed a class of QR algorithms that can perform the factorization of a panel (block-columns) in parallel, named Communication-Avoiding QR (CAQR). Compared to Algorithm 2, steps 1 and 2 are performed on panels of several columns thanks to a new kernel, called TSQR (since a panel is actually a TS matrix). CAQR successively performs a TSQR factorization (local factorizations and merging procedures) over the panels of

Algorithm 2 Pothen and Raghavan's algorithm.

Successively apply the three following steps over each column of the matrix:

1. **Local factorization.** Split the current column into P pieces (if P is the number of processors) and let each processor independently zeroes its subcolumn leading to a single non zero element per subcolumn.
2. **Merge.** Annihilate those nonzeros thanks to what they call a *recursive elimination phase* and that we name *merging step* for consistency with upcoming algorithms. This merging step is itself composed of $\log_2(P)$ stages. At each stage, processors cooperate pairwise to complete the transformation. After its element has been zeroed, a processor takes no further part in the merging procedure. The processor whose element is updated continues with the next stage. After $\log_2(P)$ such stages, the only remaining nonzero is the diagonal element. All in all, the merging step can be represented as a binary tree where each node corresponds to a pairwise transformation.
3. **Update.** Update the trailing submatrix.

the matrix, applying the subsequent updates on the trailing submatrix after each panel factorization, as illustrated in Figure 3. The panels are themselves split in block-rows, called *domains*, that are factorized independently (step 1) and then merged (step 2) using a binary tree strategy similar to the one of Pothen and Raghavan. Figure 2 illustrates TSQR's merging procedure (step 2). Initially, at stage $k = 0$, a QR factorization is performed on each domain. Then, at each stage $k > 0$ of the binary tree, the R factors are merged into pairs $R_{i,k}$ and $R_{i+1,k}$ and each pair formed that way is factorized. This is repeated until the final R is obtained. If the matrix is initially split in P domains, again, $\log_2(P)$

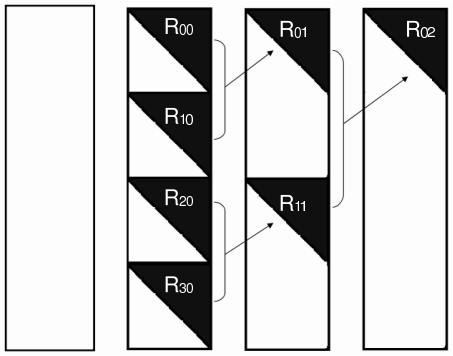

Fig. 2. TSQR factorization on four domains. The intermediate and final R factors are represented in black.

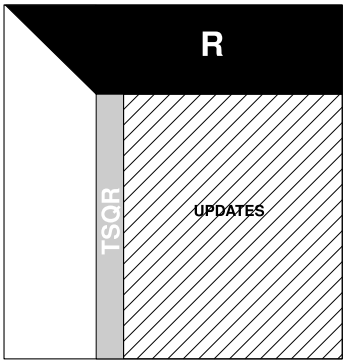

Fig. 3. CAQR: the panel (gray area) is factorized using TSQR. The trailing matrix (dashed area) is updated.

(the depth of the binary tree) stages are performed during the merge procedure. Demmel et al. proved that TSQR and CAQR algorithms a minimum amount of communication (under certain conditions, see Section 17 of [9] for more details) and are numerically as stable as the Householder QR factorization.

Both Pothen and Raghavan's and Demmel et al.'s approaches have a synchronization point between each panel factorization (TSQR kernel in Demmel et al.'s case) and the subsequent update of the trailing submatrix, leading to a suboptimal usage of the parallel processing power.

Synchronization 1 *Processors (or cores) that are no longer active in the merging step still have to wait the end of that merging step before initiating the computation related to the next panel.*

In the next section, we present an asynchronous algorithm that overcomes these bottlenecks and allows look-ahead in the scheduling.

3 Tile CAQR (SP-CAQR and FP-CAQR)

In this section, we present two new algorithms that extend the tile QR algorithm (as implemented in PLASMA and described in Section 2.2) by performing the factorization of a panel in parallel (based on the CAQR approach described in Section 2.3). Furthermore, we adapt previous parallel panel factorization approaches [9, 17] in order to enable a fully asynchronous factorization, which is critical to achieve high performance on multicore architectures. The names of our algorithms below come from the degree of parallelism of their panel factorization.

3.1 Semi-Parallel Tile CAQR (SP-CAQR)

As CAQR, our Semi-Parallel Tile CAQR algorithm (SP-CAQR) decomposes the matrix in domains. Within a domain, a PLASMA-like factorization (tile algorithm given in Algorithm 1) is performed. The domains are almost processed in an embarrassingly parallel fashion, from one to another.

First, a QR factorization is independently performed in each domain on the current panel (of a tile width), similary to step 1 of Algorithm 2. Second, the corresponding updates are applied to the trailing submatrix in each domain, similary to step 1 of Algorithm 2. For example, Figure 4 illustrates the factorization of the first panel and the corresponding updates for two domains of 3-by-3 tiles (MT=6 and NT=3). Compared to CAQR Demmel et al.'s approach, our algorithm has the flexibility to interleave steps 1 and 3 of the initial Algorithm 2. Third and last, the final local R factors from each domain are merged based on the TSQR algorithm described in Section 2.3 and the corresponding block-row is again updated. This is the only time where a particular domain needs another one to advance in the computation. The merging procedure can also be performed as the factorization and update processes go (first and second steps). Moreover, cores that no longer participate in the merging procedure can proceed

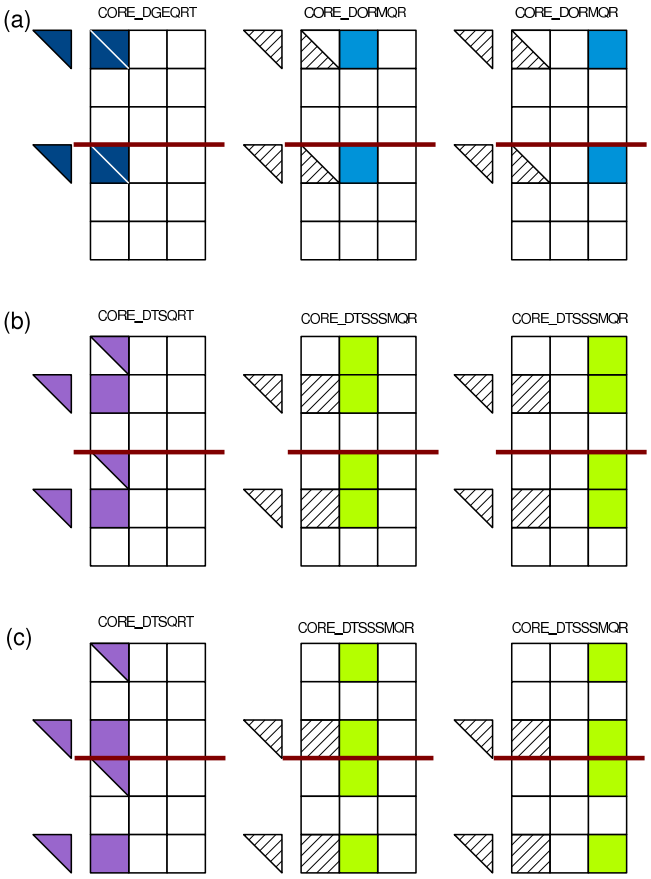

Fig. 4. Unrolling the operations related to the first panel in SP-CAQR. Two domains are used, separated by the red line. Two steps are illustrated. First, the factorization of the first tile in each domain and the corresponding updates are shown in (a). Second, the factorization of the second and third tiles in each domain using the reference tile and the corresponding updates are presented in (b) and (c) respectively.

right away with the computation of the next panel. This can potentially enable look-ahead in the scheduling.

Figure 5 illustrates the merge procedure related to the first panel factorization. Two new kernels are used in this step to reduce a triangular tile on top

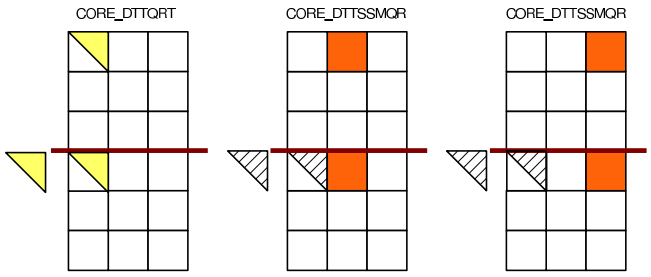

Fig. 5. Unrolling the merging procedure related to the first panel factorization in SP-CAQR.

of another triangular tile as well as the related updates. From that point on, we consider the matrices locally to their domain and we note them with three subscripts. For instance $A_{p,i,j}$ is the tile (or block-matrix) at (local) block-row i and (local) block-column j in domain p. And we want to merge two domains, let us say $p1$ and $p2$. With these notations, here are the two new kernels:

- **CORE_DTTQRT**: this kernel performs the QR factorization of a matrix built by coupling factor $R_{p1,k,k}$ from the domain $p1$ with the $R_{p2,1,k}$ from the domain $p2$. It produces an updated $R_{p1,k,k}$ factor, an upper triangular matrix $V_{p2,1,k}$ containing the Householder reflectors and an upper triangular matrix $T^r_{p2,1,k}$ resulting from accumulating the reflectors $V_{p2,1,k}$. The reflectors are stored in the upper annihilated part of the matrix. Another extra storage is needed for $T^r_{p2,1,k}$.
- **CORE_DTTSSMQR**: this kernel applies the reflectors $V_{p2,1,k}$ and the matrix $T^r_{p2,1,k}$ computed by CORE_DTTQRT to two tiles $A_{p1,k,j}$ and $A_{p2,1,j}$.

Finally, Figure 6 unrolls the third and last panel factorization. A QR factorization is performed on the last tile of the first domain as well as on the entire panel of the second domain. The local R factors are then merged to produce the final R factor.

We call the overall algorithm Semi-Parallel because the degree of parallelism of the panel factorization depends on the number of domains used. For instance, on a 32 core machine, let us assume that a matrix split in 8 domains. Even if each domain is itself performed in parallel (with a PLASMA-like factorization), then 8 cores (maximum) may simultaneously factorize a given panel (one per domain). The main difference against Algorithm 1 is that Algorithm 1 is optimized for cache reuse [4] (data is loaded into cache a limited number of times) whereas

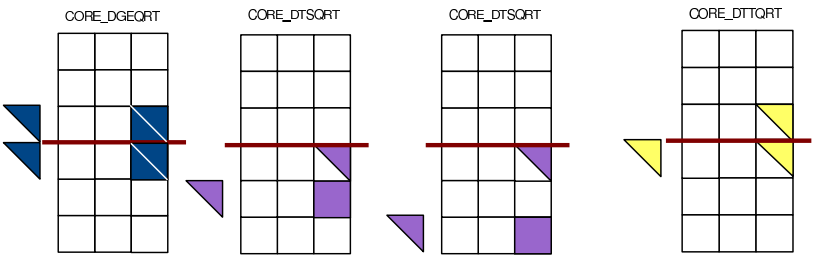

Fig. 6. Factorization of the last panel (including its merging) in SP-CAQR.

our new algorithm (SP-CAQR) provides more parallelism by processing a panel in parallel. The expected gain will thus be a trade off between increased degree of parallelism and efficient cache usage.

Assuming that a matrix A is composed of MT tiles in row and NT tiles in column, SP-CAQR corresponds to Algorithm 3. The PLASMA-like factorization occurring within each domain p is interleaved with the merge operations for each panel k. We note MT_{loc} the number of tiles per column within a domain (assumed constant) and *proot* the index of the domain containing the diagonal block of the current panel k. The PLASMA-like factorization occurring in a domain is similar to Algorithm 1 except that the reference tile in domain p is not always the diagonal block of the domain (as already noticed in Figure 6). Indeed, if the diagonal block of the current panel k is part of domain *proot* ($p == proot$), then the reference tile is the diagonal one ($ibeg = k - proot \times MT_{loc}$). Otherwise (i.e., $p \neq proot$), the tile of the first block-row of the panel is systematically used as a reference ($ibeg = 0$) to annihilate the subsequent tiles located below, within the same domain. The index of the block-row merged is then affected accordingly ($i1 = k - proot \times MT_{loc}$ when $p1 == proot$).

3.2 Fully-Parallel Tile CAQR (FP-CAQR)

One may proceed further in the parallelization procedure by aggressively and independently factorizing each tile located on the local panel of each domain. The idea is to process the remaining part of a panel within a domain in parallel too, with a local merging procedure. Figure 7 describes this Fully-Parallel QR factorization (FP-CAQR), and the corresponding algorithm is given in Algorithm 4.

Actually, FP-CAQR does not depend on the number of domains used, provided that the number of tiles per column is a power of two (otherwise the pairs used for the merge operations might not match, from one instance to another). Furthermore, a given instance of FP-CAQR can be obtained with an instance of SP-CAQR by choosing the instance of SP-CAQR with a number of domains P equal to the number of tiles per row MT. This approach has been mainly mentioned for pedagogic and completeness purposes. Therefore, we will focus on SP-CAQR in the remainder of the paper. In the following section, we will discuss frameworks for exploiting this exposed parallelism.

Algorithm 3 Semi-Parallel Tile CAQR (SP-CAQR)

$nextMT = MT_{loc}$
$proot = 0$
for $k = 1$ to $min(MT, NT)$ **do**
  if $k > nextMT$ **then**
    $proot++$
    $nextMT += MT_{loc}$
  end if
  /* PLASMA-like factorization in each domain */
  for $p = proot$ to $P - 1$ **do**
    $ibeg = 0$
    if $p == proot$ **then**
      $ibeg = k - proot \times MT_{loc}$
    end if
    $R_{p,ibeg,k}, V_{p,ibeg,k}, T_{p,ibeg,k} \leftarrow$ CORE_DGEQRT$(A_{p,ibeg,k})$
    for $j = k + 1$ to NT **do**
      $A_{p,ibeg,j} \leftarrow$ CORE_DORMQR$(V_{p,ibeg,k}, T_{p,ibeg,k}, A_{p,ibeg,j})$
    end for
    for $i = ibeg + 1$ to MT_{loc} **do**
      $R_{p,ibeg,k}, V_{p,i,k}, T_{p,i,k} \leftarrow$ CORE_DTSQRT$(R_{p,ibeg,k}, A_{p,i,k})$
      for $j = k + 1$ to NT **do**
        $A_{p,ibeg,j}, A_{p,i,j} \leftarrow$ CORE_DTSSSMQR$(V_{p,i,k}, T_{p,i,k}, A_{p,ibeg,j}, A_{p,i,j})$
      end for
    end for
  end for
  /* Merge */
  for $m = 1$ to $ceil(\log_2(P - proot))$ **do**
    $p1 = proot$; $p2 = p1 + 2^{m-1}$
    while $p2 < P$ **do**
      $i1 = 0$; $i2 = 0$
      if p1==proot **then**
        $i1 = k - proot \times MT_{loc}$
      end if
      $R_{p1,i1,k}, V_{p2,i2,k}, T^r_{p2,i2,k} \leftarrow$ CORE_DTTQRT$(R_{p1,i1,k}, R_{p2,i2,k})$
      for $j = k + 1$ to NT **do**
        $A_{p1,i1,j}, A_{p2,i2,j} \leftarrow$ CORE_DTTSSMQR$(V_{p2,i2,k}, T^r_{p2,i2,k}, A_{p1,i1,j}, A_{p2,i2,j})$
      end for
      $p1 += 2^m$; $p2 += 2^m$
    end while
  end for
end for

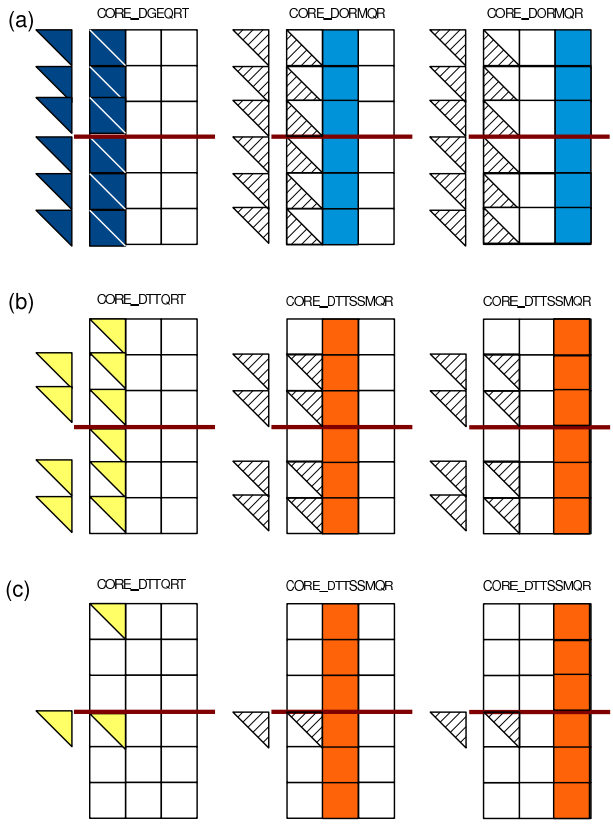

Fig. 7. Unrolling the operations related to the first panel in FP-CAQR. Two domains are used, separated by the red line. The tree steps are illustrated, the factorization(a), the local merge (b) and the global merge(c).

Algorithm 4 Fully-Parallel Tile CAQR (FP-CAQR)

$nextMT = MT_{loc}$
$proot = 0$
for $k = 1$ to $min(MT, NT)$ **do**
  if $k > nextMT$ **then**
    $proot++$
    $nextMT += MT_{loc}$
  end if
  /* PLASMA-like factorization in each domain */
  for $p = proot$ to $P - 1$ **do**
    $ibeg = 0$
    if $p == proot$ **then**
      $ibeg = k - proot \times MT_{loc}$
    end if
    for $i = ibeg$ to $MT_{loc} - 1$ **do**
      $R_{p,i,k}, V_{p,i,k}, T_{p,i,k} \leftarrow$ CORE_DGEQRT$(A_{p,i,k})$
      for $j = k + 1$ to NT **do**
        $A_{p,i,j} \leftarrow$ CORE_DORMQR$(V_{p,i,k}, T_{p,i,k}, A_{p,i,j})$
      end for
    end for
    /* Local Merge */
    for $m = 1$ to $ceil(\log_2(MT_{loc} - ibeg)$ **do**
      $i1 = ibeg; i2 = i1 + 2^{k-1}$
      while $i2 < MT_{loc}$ **do**
        $R_{p,i1,k}, V_{p,i2,k}, T^r_{p,i2,k} \leftarrow$ CORE_DTTQRT$(R_{p,i1,k}, R_{p,i2,k})$
        for $j = k + 1$ to NT **do**
          $A_{p,i1,j}, A_{p,i2,j} \leftarrow$ CORE_DTTSSMQR$(V_{p,i2,k}, T^r_{p,i2,k}, A_{p,i1,j}, A_{p,i2,j})$
        end for
        $i1 += 2^k$; $i2 += 2^k$
      end while
    end for
  end for
  /* Global Merge */
  for $m = 1$ to $ceil(\log_2(P - proot))$ **do**
    $p1 = proot$; $p2 = p1 + 2^{m-1}$
    while $p2 < P$ **do**
      $i1 = 0$; $i2 = 0$
      if p1==proot **then**
        $i1 = k - proot \times MT_{loc}$
      end if
      $R_{p1,i1,k}, V_{p2,i2,k}, T^r_{p2,i2,k} \leftarrow$ CORE_DTTQRT$(R_{p1,i1,k}, R_{p2,i2,k})$
      for $j = k + 1$ to NT **do**
        $A_{p1,i1,j}, A_{p2,i2,j} \leftarrow$ CORE_DTTSSMQR$(V_{p2,i2,k}, T^r_{p2,i2,k}, A_{p1,i1,j}, A_{p2,i2,j})$
      end for
      $p1 += 2^m$; $p2 += 2^m$
    end while
  end for
end for

4 Parallel Scheduling

This section explains how the DAG induced by SP-CAQR can be efficiently scheduled on a multicore machine. Two schedulers approaches are discussed: a static approach where the scheduling is predetermined (exactly the one implemented in PLASMA) and a dynamic approach where decisions are made at runtime.

4.1 Static scheduling

Developed initially on the IBM Cell processor [14], the static implementation is a hand-written code using POSIX threads and primitive synchronization mechanisms. In the tile QR, the work is distributed by columns of tiles. Figure 8 shows the step-by-step scheduling execution with 8 threads on a square 5-by-5 tile matrix ($MT = NT = 5$). There are five panel factorization steps and each of those steps is performed in sequence. It implements a right-looking QR factorization and the steps of the factorization are pipelined. The mapping of the cores to the tasks is performed before the actual numerical factorization based on a one-dimensional partitioning of work and a lookahead of varying depth. The lookahead strategy greedily maps the cores that are expected to run out of work to the different block column operations. This static approach is well adapted to schedule Algorithm 1 and achieves high performance [4] thanks to an efficient cache reuse [15]. This static scheduling could be extended to SP-CAQR

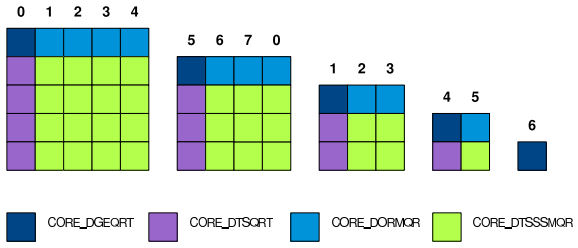

Fig. 8. Work assignment in the static pipeline implementation of the tile QR factorization.

algorithm since SP-CAQR performs a PLASMA-like factorization on each domain. However, this would raise load balance issues that are out-of-scope for this paper[4]. Another solution consists in using a dynamic scheduler where the tasks are scheduled as soon as their dependencies are satisfied and that prevents cores from stalling.

[4] One might think to map a constant number of cores per domain, but, after NT panels have been processed, the cores of the first domain would then run out-of-work.

4.2 Dynamic scheduling

We have developed an experimental dynamic scheduler well suited for linear algebra algorithms [13]. This scheduler is still at the level of a prototype especially because the current version has:

- a non negligible cost for scheduling (scheduling itself basically requires one dedicated core on our both platforms);
- a centralized scheduling (handled by one single thread), which would not scale above hundreds of cores.

Although we have coupled our algorithms to this scheduler, we preferred to present experimental results obtained with a well established dynamic scheduler, SMPSs.

SMP Superscalar (SMPSs) [3] is a parallel programming framework developed at the Barcelona Supercomputer Center (Centro Nacional de Supercomputación). SMPSs is a dynamic scheduler implementation that addresses the automatic exploitation of the functional parallelism of a sequential program in multicore and symmetric multiprocessor environments.

SMPSs allows the programmers to write sequential applications, and the framework is able to exploit the existing concurrency and to use the different processors by means of an automatic parallelization at execution time. As in OpenMP [2], the programmer is responsible for identifying parallel tasks, which have to be side-effect-free (atomic) functions. However, he is not responsible for exposing the structure of the task graph. The task graph is built automatically, based on the information of task parameters and their directionality.

Based on the annotations in the source code, a source to source compiler generates the necessary code and a runtime library exploits the existing parallelism by building at runtime a task dependency graph. The runtime takes care of scheduling the tasks and handling the associated data.

Regarding its implementation, it follows the same approach as described in [15] in order to get the best performance by drastically improving the scheduling. However, SMPSs is not able to recognize accesses to triangular regions of a tile. For example, if only the lower triangular region is accessed during a particular task, SMPSs will still create a dependency on the whole tile and therefore prevent the scheduling of any subsequent tasks that only use the strict upper triangular region of the same tile. To bypass this bottleneck, we force the scheduler to drop some dependencies by shifting the starting pointer address of the tile back and forth [5]. In the next section, experimental results of our SP-CAQR algorithm with SMPSs are presented.

[5] On the other hand, our dynamic scheduler prototype can identify such fake dependencies and eliminate them with the keyword NODEP, meaning there is no dependency required for this particular data.

5 Experimental Results

5.1 Experimental environment

The experiments were performed on a quad-socket, quad-core machine based on an Intel Xeon EMT64 E7340 processor operating at 2.39 GHz. The theoretical peak is equal to 9.6 Gflop/s/ per core or 153.2 Gflop/s for the whole node, composed of 16 cores. There are two levels of cache. The level-1 cache, local to the core, is divided into 32 kB of instruction cache and 32 kB of data cache. Each quad-core processor being actually composed of two dual-core Core2 architectures, the level-2 cache has 2×4 MB per socket (each dual-core shares 4 MB). The machine is running Linux 2.6.25 and provides Intel Compilers 11.0 together with the MKL 10.1 vendor library [1].

The performance of QR tile algorithms strongly depends on tunable execution parameters of the outer and the inner blocking sizes [4]. The outer block size (NB) trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size (IB) trades off memory load with extra-flops due to redundant calculations. In the experiment, NB and IB were set to 200 and 40.

We recall that SP-CAQR depends on the number P of domains used, and we note SP-P an instance of SP-CAQR with P domains. If $P = 1$, it corresponds to a PLASMA-like implementation (except that SP-1 uses a dynamic scheduler instead of a static scheduler). On the other side of the spectrum, we recall that $P = MT$ (MT is the number of tiles per row) corresponds to the FP-CAQR algorithm. As discussed in Section 4, we remained that our SP-CAQR algorithm is scheduled thanks to SMPSs.

In this section, we essentially present experiments on tall and skinny matrices (where the higher improvements are expected), but we also consider general and square matrices. A comparison against state of the art linear algebra packages (LAPACK, ScaLAPACK, PLASMA) and the vendor library MKL 10.1 concludes the section.

5.2 Tall and Skinny matrices

Figure 9 shows the performance obtained on matrices of only two tiles per row, using 16 cores. The overall limited performance (at best 12% of the theoretical peak of the machine) shows the difficulty achieving high performance on TS matrices. This is mainly due to the Level-2 BLAS operations which dominate the panel factorization kernels.

If the matrix is tall enough, FP-CAQR and SP-CAQR (if the number of domains is large too) are significantly faster than the original (PLASMA-like) Tile QR (up to more than 3 times faster). With such tall and skinny matrices, the more subdomains, the higher performance. In particular, FP-CAQR (that is SP-CAQR with $P = 32$ since $MT = 32$) is consistently optimum or close to the optimum.

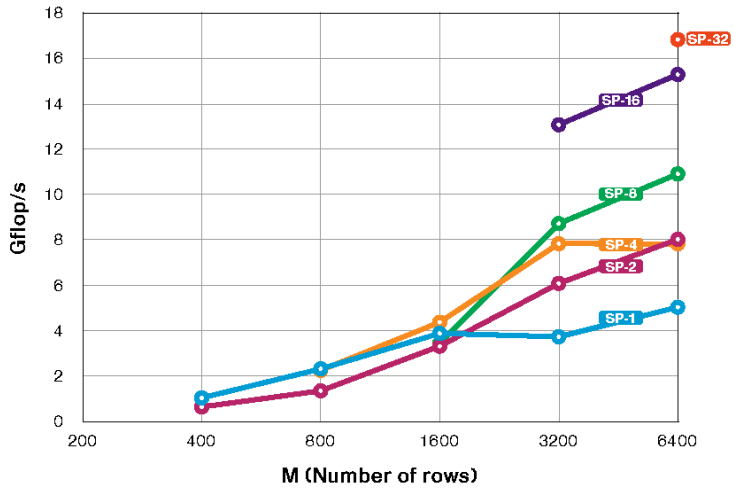

Fig. 9. Performance of 16 core executions on TS matrices with 2 tiles per row ($N = 400$ is fixed). The plot is under-scaled (the actual theoretical peak performance is 153.2 Gflop/s. The number of tiles per column MT has to be greater than or equal to the number of domains P; for instance, SP-16 can only be executed on matrices of at least $M = 16 * 200 = 3200$ rows, since a tile is itself of order 200.

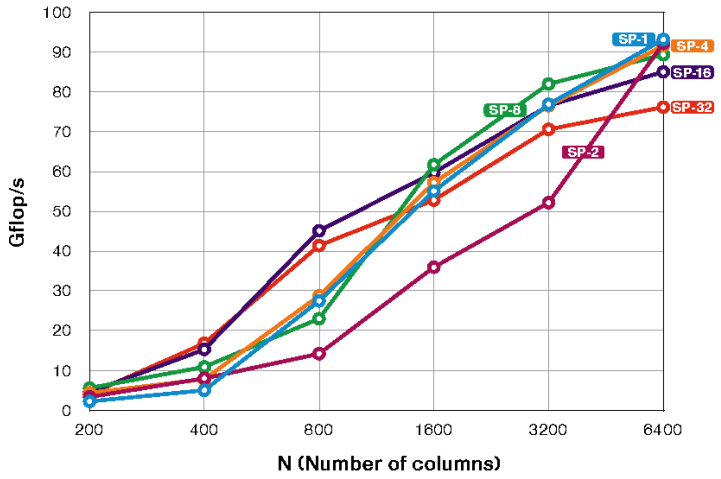

Fig. 10. Performance of 16 core executions on TS matrices with 32 tiles per column ($M = 6400$ is fixed).

Figure 10 shows the performance of matrices with 32 tiles per column on execution using 16 cores. The improvement brought by SP-CAQR is again strong for TS matrices (SP-16 is twice faster than SP-1 when $N = 800$). However, when the shape of the matrix tends to be square (right part of the graph), the PLASMA-like algorithm (SP-1) becomes relatively more and more efficient. It is the fastest execution in the case of the factorization of a square matrix ($6400 * 6400$). The reason is that, for such large square matrices, the lack of parallelism within the panels is mostly hidden by the other opportunities of parallelism (see Section 2.2) and is thus completely balanced by the very good cache usage of PLASMA-like factorizations.

5.3 Square matrices

Figures 11 and 12 show the performance obtained on square matrices using 8 and 16 cores, respectively. They confirm that the lack of parallelism of PLASMA-like algorithms (SP-1) on small matrices leads to a limited performance and are outperformed by semi-parallel (SP-P, $P > 1$) and fully-parallel algorithms (SP-MT). On the other hand, PLASMA-like factorizations become the most efficient approach for matrices of order greater than 3200. Note that the number of tiles per column MT has to be greater than or equal to the number of domains P; for instance, SP-16 can only be executed on matrices of order at least equal to $M = 16 * 200 = 3200$ rows, since a tile is itself of order 200.

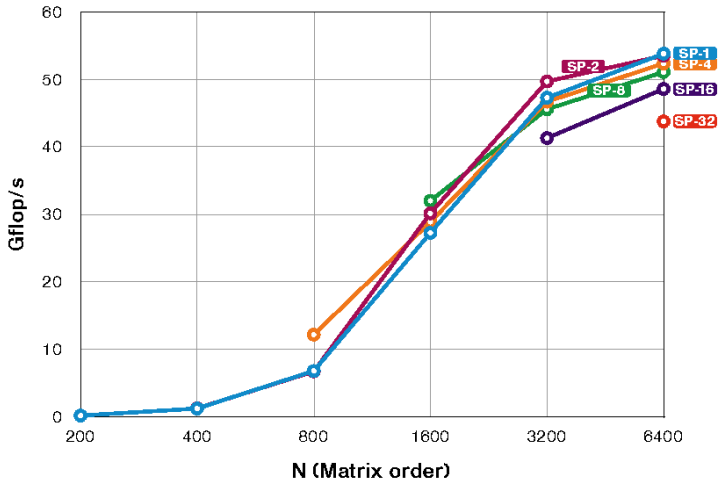

Fig. 11. Performance on square matrices using 8 cores.

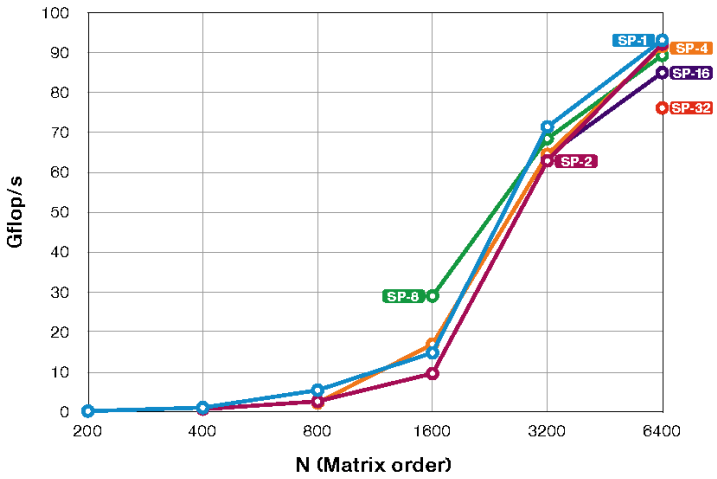

Fig. 12. Performance on square matrices using 16 cores.

5.4 Comparison with state-of-the-art libraries

In Figure 13, we compare our new approach, SP-CAQR against PLASMA, ScaLAPACK, LAPACK and MKL for a tall and skinny matrix of size 51200 × 3200. SP-CAQR is 27% faster than the original Tile QR as implemented in PLASMA, if the matrix is split in 16 domains (SP-16). Furthermore, for this matrix shape, SP-CAQR is slightly faster when scheduled dynamically (SP-1) than statically (PLASMA) with a ratio of 79 Gflop/s against 75 Gflop/s. The performance of SP-CAQR depends on the number of domains. In this case, the most significant performance variation (21%) is obtained between 2 and 4 domains.

Figure 14 shows the performance on 16 cores of the QR factorization of a matrix where the number of rows is fixed to 51200 and the number of columns varies. For tall and skinny matrix of size 51200 by 200, our approach for computing the QR factorization is almost 10 times faster than the Tile QR of PLASMA and around 9 times than MKL (exactly 9.54 and 8.77 as reported in Table 3). This result is essentially due to the higher degree of parallelism brought by the parallelization of the panel factorization. It is interesting to notice that the ratio is of the order of magnitude of the number of cores, 16, which is clearly an upper bound. LAPACK is around 30 times slower than our approach, while ScaLAPACK is only 3 times slower. By increasing the number of tiles in a column of the matrix, the ratio is less important, however, SP-CAQR is still faster by far compared to state of the art linear algebra packages. PLASMA is performing better and tends to reach the performance of SP-CAQR when the number of tiles in the column are increased. For instance, PLASMA is only 1.27 times slower for matrix size of 51200 by 3200. Regarding the other libraries, the ratio

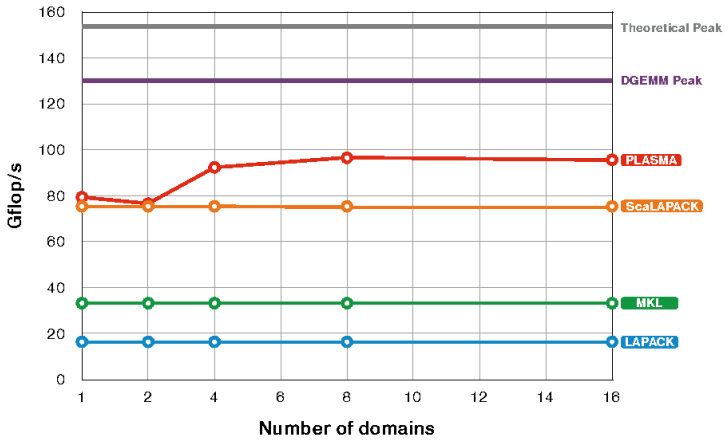

Fig. 13. Performance Comparisons of SP-CAQR depending on the number of domains.

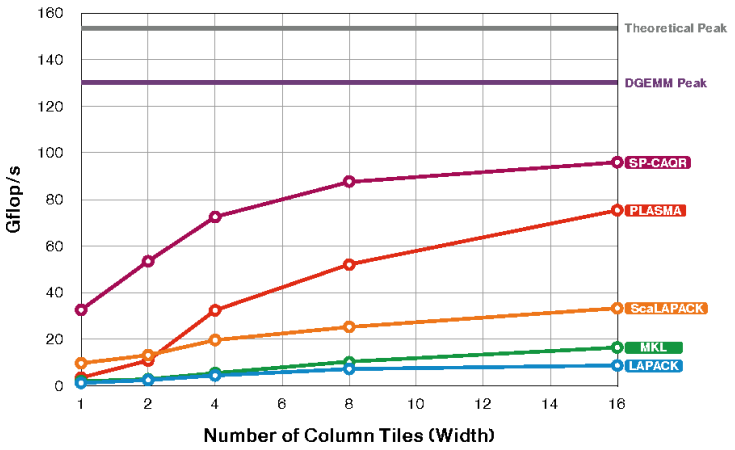

Fig. 14. Scalability of SP-CAQR.

compared to ScaLAPACK is still at 3, while SP-CAQR is more than 4 times and 11 times faster than MKL and LAPACK respectively.

Table 3. Ratio comparison of the performance of SP-CAQR.

Matrix sizes	**PLASMA**	**MKL**	**ScaLAPACK**	**LAPACK**
$51200 - 200$	9.54	8.77	3.38	28.63
$51200 - 3200$	1.27	4.10	2.88	11.05

6 Conclusions and Future Work

By combining two existing algorithms (Tile QR from PLASMA and CAQR), we have proposed a fully asynchronous and numerically stable QR factorization scheme for shared-memory multicore architectures. We have shown a significant performance improvement (up to almost 10 times faster against previous established linear algebra libraries). In this paper, we have considered a fixed tile size (200) and inner blocking size (40). It would be interesting to see the impact of those tunable parameters on performance, together with the choice of the number of domains. Autotuning techniques will certainly have to be considered.

We have inserted our algorithms into the PLASMA library through its interface for dynamic scheduling [15] and eventually plan to release them. These algorithms also represent a natural basis for extending the PLASMA library to distributed-memory environments. We will indeed benefit from the low amount of communication induced by communication-avoiding algorithms, which are actually minimum [9]. Furthermore, we plan to investigate the extension of this work to the LU factorization where numerical stability issues are more complex [12].

References

1. Intel, Math Kernel Library (MKL). http://www.intel.com/software/products/mkl/.
2. OpenMP. http://www.openmp.org/.
3. SMP Superscalar. http://www.bsc.es/ → Computer Sciences → Programming Models → SMP Superscalar.
4. E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. LAPACK Working Note 217, ICL, UTK, April 2009.
5. E. Anderson, Z. Bai, C. Bischof, L. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

6. 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, and R. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.
7. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.
8. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.
9. J. W. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou. Communication-optimal parallel and sequential qr and lu factorizations. LAPACK Working Note 204, UTK, August 2008.
10. R. W. Freund and M. Malhotra. A block qmr algorithm for non-hermitian linear systems with multiple right-hand sides. *Linear Algebra and its Applications*, 254(1–3):119–157, 1997.
11. G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, second edition, 1989.
12. L. Grigori, J. W. Demmel, and H. Xiang. Communication avoiding gaussian elimination. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, 2008.
13. J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. LAPACK Working Note 220, ICL, UTK, June 2009.
14. J. Kurzak and J. Dongarra. Qr factorization for the cell broadband engine. *Sci. Program.*, 17(1-2):31–42, 2009.
15. J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia. Scheduling linear algebra operations on multicore processors. LAPACK Working Note 213, ICL, UTK, February 2009.
16. D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra and Its Applications*, 29:293–322, 1980.
17. A. Pothen and P. Raghavan. Distributed orthogonal factorization: Givens and Householder algorithms. *SIAM Journal on Scientific and Statistical Computing*, 10:1113–1134, 1989.
18. R. Schreiber and C. Van Loan. A storage efficient WY representation for products of householder transformations. *SIAM J. Sci. Statist. Comput.*, 10:53–57, 1989.