HAL
open science

Tile QR Factorization with Parallel Panel Processing for Multicore Architectures

Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, Jack Dongarra

To cite this version:

Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, Jack Dongarra. Tile QR Factorization with Parallel Panel Processing for Multicore Architectures. 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Apr 2010, Atlanta, United States. inria-00548899

HAL Id: inria-00548899

https://inria.hal.science/inria-00548899v1

Submitted on 20 Dec 2010

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

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

Tile QR Factorization with Parallel Panel Processing for Multicore Architectures *

Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, Jack Dongarra

Department of Electrical Engineering and Computer Science,

University of Tennessee, Knoxville

{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 new 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 Multi-core 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 [3]. Tile algorithms in general provide fine granularity parallelism and 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 factorization 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 factorization scales fairly well for large 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

*Research reported here was partially supported by the NSF and Microsoft Research.

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	4	8	16	32
500	**43%**	**25%**	**12%**	**6%**
1000	67%	**46%**	**24%**	**12%**
2000	80%	65%	**46%**	**25%**
4000	90%	79%	71%	51%

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 for a matrix of size 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 TS 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 memory and/or cache hierarchy to benefit from 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 a new approach that combines algorithmic ideas from tile algorithms and the communication avoiding approaches. Section 4 explains how the tasks from the resulting DAG are scheduled in parallel. In Section 5, an experimental study shows the behavior of our algorithm on multicore architectures and compares it 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, 17]. 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 handling 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

Generally, a QR factorization of an $m \times n$ real matrix A is the decomposition of A as $A = QR$, where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper

triangular matrix. QR factorization uses a series of elementary Householder matrices of the general form $H = I - \tau vv^T$ where v is a column reflector and τ is a scaling factor.

Regarding the block or block-partitioned algorithms as performed in LAPACK [5] or ScaLAPACK [6] respectively, nb elementary Householder matrices are accumulated within each panel and the product is represented as $H_1H_2...H_{nb} = I - VTV^T$. 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 can be identified as a sequential execution 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 of block factorizations is limited on a multicore system. The parallelism is only exploited at the level of the BLAS routines for LAPACK or PBLAS routines for ScaLAPACK. This methodology complies a fork-join model since the execution flow of a block factorization represents a sequence of sequential operations (panel factorizations) interleaved with parallel ones (updates of the trailing submatrices).

2.2 Tile QR factorization (PLASMA-like factorization)

PLASMA Tile QR factorization [7, 8] evolves from the block algorithms that provides high performance implementations for multicore system architectures. The algorithm is based on annihilating matrix elements by square tiles instead of rectangular panels as in LAPACK. PLASMA Tile QR algorithm relies on four primary operations developed by four computational kernels:

- **CORE_DGEQRT**: this routine 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} and a unit lower triangular matrix V_{kk} containing the Householder reflectors. An upper triangular matrix T_{kk} is also computed as defined by the WY technique [19] 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 the structure 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 routine 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, a matrix V_{ik} containing the Householder reflectors and a matrix T_{ik} resulting from accumulating the reflectors V_{ik}.

- **CORE_DORMQR**: this routine applies 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 routine 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 factorization 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 different T_{ij} triangular matrices. 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. As of today, PLASMA implements

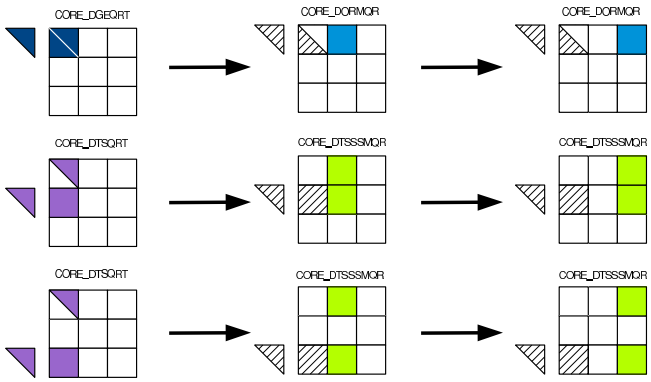

Figure 1: Reduction of the first tile column.

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

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, 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 the late 1980s [18]. 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

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 step and remains idle until the end of that step. 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.

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 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 et al. 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 ($R_{0,2}$ in Figure 2) is obtained. If the matrix is initially split in P domains, $\log_2(P)$ (the depth of the

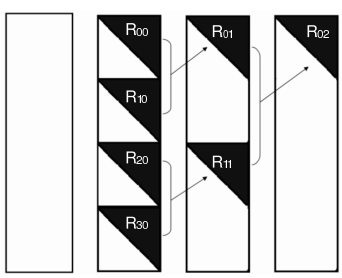

Figure 2: TSQR factorization on four domains. The intermediate and final R factors are represented in black.

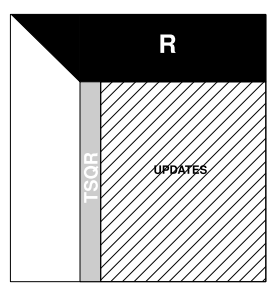

Figure 3: CAQR: the panel (gray area) is factorized using TSQR. The trailing matrix (dashed area) is updated.

binary tree) stages are performed during the merge procedure. Demmel proved that TSQR and CAQR algorithms induce 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 enables look-ahead in the scheduling.

3 Tile CAQR (SP-CAQR)

In this section, we present a new algorithm that extends the Tile QR factorization (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, 18] in order to enable a fully asynchronous factorization, which is critical to achieve high performance on multicore architectures. The name of our algorithm, Semi-Parallel Tile CAQR (SP-CAQR), comes from the degree of parallelism of its panel factorization, higher than PLASMA (that has a serial panel factorization) [1].

As CAQR, SP-CAQR decomposes the matrix in domains (block-rows). 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), similarly to step 1 of Algorithm 2. Second, the corresponding updates are applied to the trailing submatrix in each domain, similarly to step 3 of Algorithm 2. For example, Figure 4 (a,b,c) illustrates the factorization of the first panel and the corresponding updates for two domains of 3-by-3 tiles (MT=6 and NT=3). The update procedure is triggered while the panel is still being factorized. Indeed, 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 (steps 1 and 2). Moreover, cores that no longer participate in the merging procedure can proceed *right away* with the computation of the next panel. Synchronization 1 is now released in our SP-CAQR approach which can potentially enable look-ahead in the scheduling. Figure 4(d) illustrates the merging procedure related to the first panel factorization. The factorization of the second panel can be initiated while the merging procedure of the first panel has not yet terminated.

Two new kernels are used in this step for reducing a triangular tile on top of another triangular tile as well as applying 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-

[1] In a technical report [13], we also proposed a variant of this algorithm benefiting from an even higher degree of parallelism within the panel factorization. We called this variant Fully-Parallel Tile CAQR (FP-CAQR). For a matter of conciseness, we will not discuss it here.

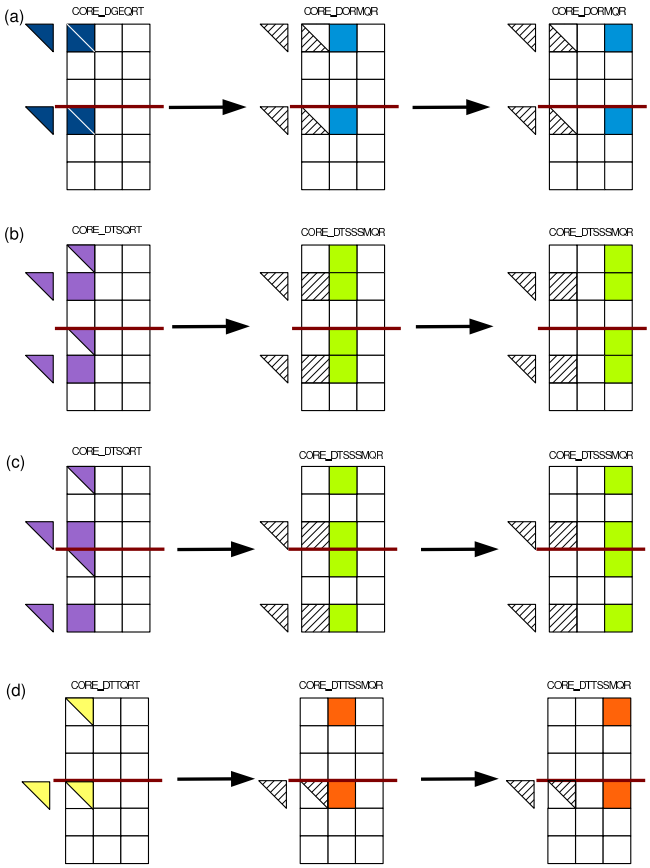

Figure 4: Unrolling the operations related to the first panel in SP-CAQR. Two domains are used, separated by the red line. First step, the factorization of the first tile in each domain and the corresponding updates are shown in (a). Second step, the factorization of the second and third tiles in each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

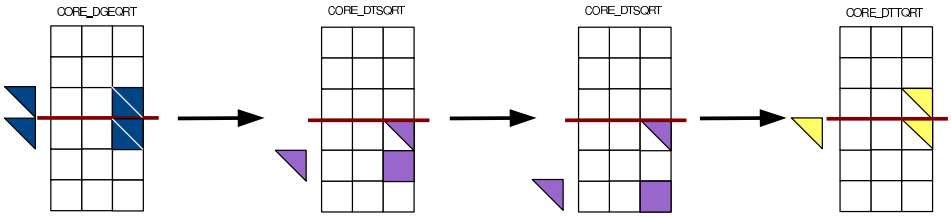

Figure 5: Factorization of the last panel and the merging step in SP-CAQR.

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 routine performs the QR factorization of a matrix built by coupling the factor $R_{p1,k,k}$ from the domain $p1$ with the factor $R_{p2,1,k}$ from the domain $p2$. It produces an updated factor $R_{p1,k,k}$, 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 storing $T^r_{p2,1,k}$.
- **CORE_DTTSSMQR**: this routine 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 5 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 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 5). 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 $p1 == proot$). In the following section, we will discuss frameworks for exploiting this exposed parallelism.

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.

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 × MT_loc
        end if
        R_{p,ibeg,k}, V_{p,ibeg,k}, T_{p,ibeg,k} ← CORE_DGEQRT(A_{p,ibeg,k})
        for j = k + 1 to NT do
            A_{p,ibeg,j} ← 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} ← 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} ← 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 × MT_loc
            end if
            R_{p1,i1,k}, V_{p2,i2,k}, T^r_{p2,i2,k} ← 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} ← 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
```

4.1 Static scheduling

Developed initially on the IBM Cell processor [15], the static scheduling implemented in PLASMA uses POSIX threads and naive synchronization mechanisms. Figure 6 shows the step-by-step scheduling execution with 8 threads on a square tile matrix ($MT = NT = 5$). In this particular figure, the work is distributed by columns of tiles and there are five panel factorization steps and each of those steps is performed sequentially. It implements a right-looking QR factorization and the steps of the factorization are pipelined. The cores are mapped on a one dimensional partitioning. The mapping to the tasks is executed before the actual numerical factorization based on a look-ahead of varying depth. The look-ahead strategy greedily maps the cores that might 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 [16]. This static scheduling could be extended to SP-CAQR algorithm since SP-

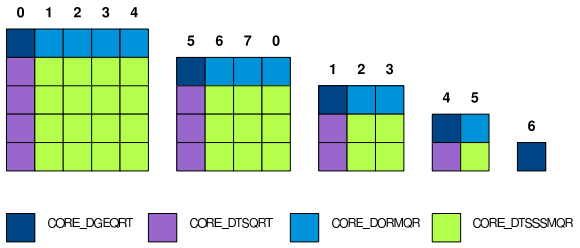

Figure 6: Work assignment in the static pipeline implementation of the tile QR factorization.

CAQR performs a PLASMA-like factorization on each domain. However, this would raise load balancing issues difficult to address with a hand-written code[2]. 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.

4.2 Dynamic scheduling

We decided to present experimental results obtained with a well established and robust dynamic scheduler, SMP Superscalar (SMPSs) [2]. SMPSs 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 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, a 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.

[2]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.

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 [16] 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. In the next section, experimental results of our SP-CAQR algorithm with SMPSs are presented.

5 Experimental Results

5.1 Experimental environment

The experiments were conducted 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 the Tile QR factorization strongly depends on two tunable parameters: the tile size (NB) and the inner blocking sizes (IB) [4]. The tile size trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size trades off memory load with extra-flops due to updating factorization techniques [11]. In the experiments, NB and IB were set to 200 and 40, respectively.

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 factorization (but SP-1 relies on SMPSs whereas PLASMA implements a static scheduler). As discussed in Section 4, our SP-CAQR algorithm is scheduled with SMPSs dynamic scheduler.

In this section, we essentially present experiments on TS 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. All the packages have been linked against the BLAS from Intel MKL.

5.2 Tall and Skinny matrices

Figure 7 shows the performance obtained on matrices of only two tiles per row, using 16 cores. 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. The overall limited performance (at best 12% of the theoretical peak of the machine) shows the difficulty to achieve 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, SP-CAQR (if the number of domains is large too) is up to more than 3 times faster than the (PLASMA-like) Tile QR algorithm (SP-1). With such TS matrices, the greater the number of domains, the higher the performance. In particular, For instance SP-32 is optimum on a 6400 by 400 matrix.

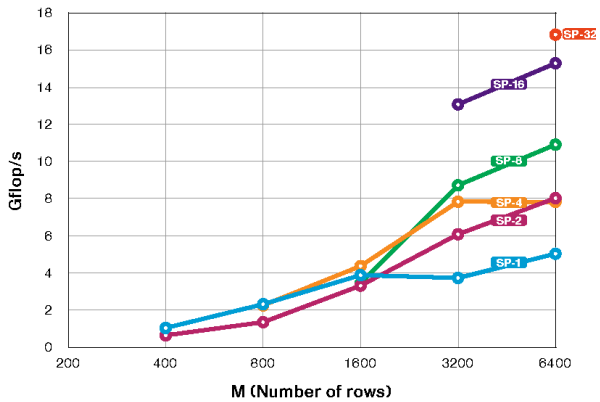

Figure 7: Performance of 16 core executions on TS matrices with 2 tiles per row ($N = 400$ is fixed).

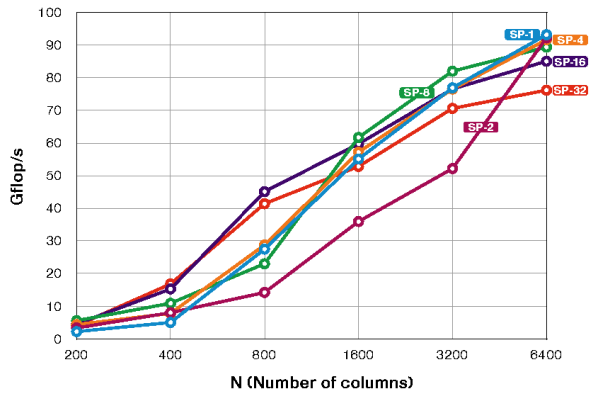

Figure 8: Performance of 16 core executions on TS matrices with 32 tiles per column ($M = 6400$ is fixed).

Figure 8 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 as fast as SP-1 when $N = 800$). However, when the shape of the matrix tends to be square (right part of the graph), 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 by 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 9 and 10 show the performance obtained on square matrices using 8 and 16 cores, respectively. They confirm that the lack of parallelism of PLASMA-like algorithm (SP-1) on small matrices leads to a limited performance and are outperformed by SP-CAQR (SP-P, $P > 1$). On the other hand, PLASMA-like factorization becomes 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.

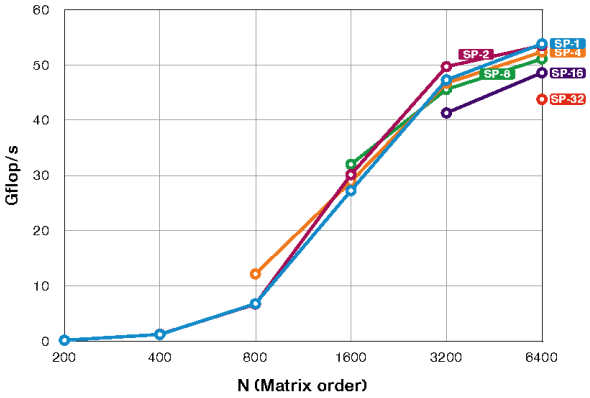

Figure 9: Performance on square matrices using 8 cores.

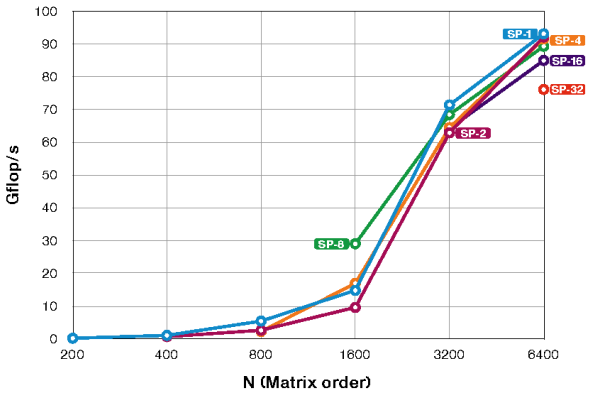

Figure 10: Performance on square matrices using 16 cores.

5.4 Comparison with state-of-the-art libraries

In Figure 11, we compare our new approach, SP-CAQR against PLASMA, ScaLAPACK, LAPACK and MKL for a TS matrix of size 51200×3200. SP-CAQR is 27% faster than 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.

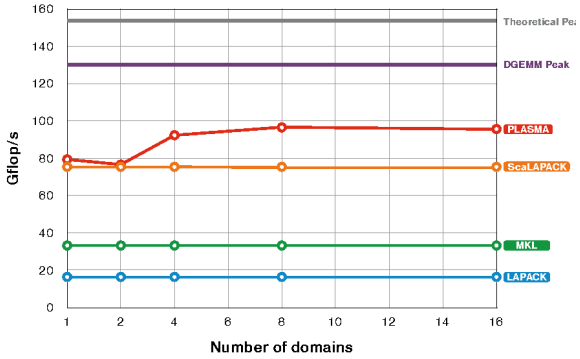

Figure 11: Performance Comparisons of SP-CAQR depending on the number of domains.

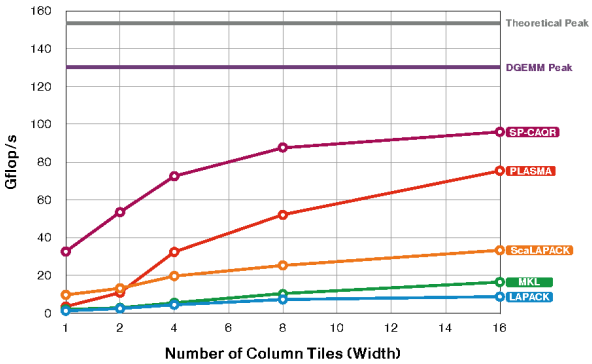

Figure 12: Scalability of SP-CAQR.

Figure 12 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 TS matrix of size 51200 by 200, our approach for computing the QR factorization is almost 10 times faster than the Tile QR factorization 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

Table 3: Improvement of SP-CAQR against other libraries (performance ratio).

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

is interesting to notice that the ratio is of 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 compared to ScaLAPACK is still at 3, while SP-CAQR is more than 4 times and 11 times faster than MKL and LAPACK respectively.

6 Conclusions and Future Work

By combining two existing algorithms (Tile QR factorization from PLASMA and CAQR approach), we have proposed a new 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). If we have experimentally assessed the impact of the number of domains on performance, we have considered however fixed values for the two other tunable parameters (a tile size NB of 200 and inner blocking size IB of 40). We expect to achieve an even better performance by tuning those parameters together with the number of domains. In particular, we plan to develop autotuning techniques to achieve an optimum performance. The experiments presented in this paper have been conducted with a well established dynamic scheduler, SMPSs. However, we have also plugged SP-CAQR with PLASMA's experimental dynamic scheduler [14], making it possible to release SP-CAQR as part of PLASMA library. We plan to do so when the dynamic scheduler will be in a more advanced stage of development.

SP-CAQR also represents a natural building block for extending PLASMA library to distributed-memory environments. We will indeed benefit from the low amount of communication induced by communication-avoiding algorithms. 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] SMP Superscalar. http://www.bsc.es/ → Computer Sciences → Programming Models → SMP Superscalar.

[3] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, and H. Ltaief. PLASMA Users Guide. Technical report, ICL, UTK, 2009.

[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. Van Loan. *Matrix Computation*. John Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

[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] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tall and Skinny QR Matrix Factorization Using Tile Algorithms on Multicore Architectures. LAPACK Working Note 222, ICL, UTK, September 2009.

[14] J. Kurzak and J. Dongarra. Fully Dynamic Scheduler for Numerical Computing on Multicore Processors. LAPACK Working Note 220, ICL, UTK, June 2009.

[15] J. Kurzak and J. Dongarra. QR factorization for the Cell Broadband Engine. *Sci. Program.*, 17(1-2):31–42, 2009.

[16] 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.

[17] D. P. O'Leary. The Block Conjugate Gradient Algorithm and Related Methods. *Linear Algebra and Its Applications*, 29:293–322, 1980.

[18] A. Pothen and P. Raghavan. Distributed orthogonal factorization: Givens and Householder algorithms. *SIAM Journal on Scientific and Statistical Computing*, 10:1113–1134, 1989.

[19] 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.