Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems

Fengguang Song*, Hatem Ltaief*, Bilel Hadri† and Jack Dongarra‡
*EECS, University of Tennessee, Knoxville, TN, USA
{song, ltaief}@eecs.utk.edu
†NICS, Oak Ridge National Laboratory, Oak Ridge, TN, USA
bhadri@utk.edu
‡University of Tennessee, Knoxville, TN, USA
Oak Ridge National Laboratory, Oak Ridge, TN, USA
University of Manchester, Manchester, UK
dongarra@eecs.utk.edu

***Abstract*—As tile linear algebra algorithms continue achieving high performance on shared-memory multicore architectures, it is a challenging task to make them scalable on distributed-memory multicore cluster machines. The main contribution of this paper is the extension to the distributed-memory environment of the previous work done by Hadri et al. on Communication-Avoiding QR (CA-QR) factorizations using tile algorithms for tall and skinny matrices (initially done on shared-memory multicore systems). The fine granularity of tile algorithms associated with communication-avoiding techniques for the QR factorization presents a high degree of parallelism where multiple tasks can be concurrently executed and computation steps fully pipelined. A decentralized dynamic scheduler has then been integrated as a runtime system to efficiently schedule tasks across the distributed resources. Our experimental results performed on two Beowulf clusters (with dual-core and 8-core nodes, respectively) and a Cray XT5 system with 12-core nodes show that the tile CA-QR factorization is able to outperform the de facto ScaLAPACK library by up to 4 times for tall and skinny matrices, and has good scalability on up to 3,072 cores.**

I. Introduction

The method of least squares has been used in many scientific fields such as mathematics, physics, statistics, and economics where applications of data fitting, regression analysis, and production function modeling happen frequently. The problem is to find the solution of an overdetermined system of linear equations $Ax = b$ with more equations than unknowns. The shape of the matrix A is tall and skinny. The modern classical method to solve such a system is based upon QR factorization by first computing $A = QR$ followed by solving the upper-triangular system $Rx = Q^*b$ for x.

Various numerical libraries have supplied the QR factorization subroutine. LAPACK [1] provides a collection of linear algebra software for shared-memory systems. ScaLAPACK [2], [3] includes a subset of LAPACK subroutines that is redesigned for distributed-memory message-passing systems. In addition, a number of vendors provide libraries optimized for their own hardware such as Intel MKL, AMD ACML, IBM ESSL (PESSL), and Cray XT LibSci. All the vendor libraries include the subroutines of LAPACK and ScaLAPACK.

However, with the increment of the number of cores on each chip, these existing libraries start to see degrading performance on multicore (or manycore) architectures. One important reason is that the libraries use the fork-join approach for parallelism to implement their routines. The join operation works as a barrier and increases the task graph's critical path length substantially. Assuming a fixed number of tasks, increasing the length of the critical path can seriously affect the program performance. For instance, the subroutine for QR factorization in LAPACK uses a block algorithm. Given an $m \times n$ matrix A that is partitioned as follows:

$$A = \begin{pmatrix} A_{1:b,1:b} & A_{1:b,b+1,n} \\ A_{b+1:m,1:b} & A_{b+1:m,b+1:n} \end{pmatrix},$$

where b is the block size, the block algorithm 1) first factorizes the left column panel $A_{1:m,1:b}$; 2) applies the panel factorization result to the top row panel $A_{1:b,b+1,n}$; 3) then to the trailing submatrix of $A_{b+1:m,b+1:n}$. All the three steps are executed in a fork-join manner for which the length of the critical path is increased. The same set of steps will be applied recursively to the submatrix of $A_{b+1:m,b+1:n}$ until the submatrix merely consists of a single column panel. The ScaLAPACK QR factorization subroutine uses the same block algorithm as LAPACK. In this paper, we use the term "block QR factorization" to refer to this algorithm.

During the last several years we have been working on designing new parallel linear algebra software for multicore architectures. We believe that the new software for multicore architectures should have the following characteristics: fine-grain tasks for a higher degree of parallelism, asynchronous execution to eliminate synchronization points, and good locality to improve data reuse. The tile algorithms designed in our Parallel Linear Algebra Software for Multicore Architectures (PLASMA) project [4] exhibit the three desirable characteristics. The subroutine for QR factorization in PLASMA adopts

This material is based upon work supported by the NSF grant CCF-0811642, and by Microsoft Research. This work also used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-ASC05-00OR22725.

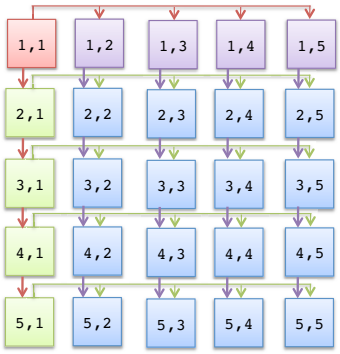


Fig. 1. Tile QR factorization on a square matrix with 5×5 tiles. Each tile is of size $b \times b$ and corresponds to a fine grain task. The arcs show the data dependencies between the tasks.

an updating-based algorithm that operates on matrices stored in a tile data layout [5]. A tile is a $b \times b$ square submatrix and is stored in memory contiguously. In this paper, we use the term "tile QR factorization" to refer to the updating-based QR factorization.

Unlike the block QR factorization that operates on panels, the tile QR factorization operates on much smaller tiles (hence more fine-grained). Given a matrix A consisting of $m_b \times n_b$ tiles, matrix A can be expressed as follows:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,n_b} \\ A_{2,1} & A_{2,2} & \dots & A_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m_b,1} & A_{m_b,2} & \dots & A_{m_b,n_b} \end{pmatrix},$$

where $A_{i,j}$ is a square tile of size $b \times b$. In the first iteration, the tile QR factorization computes the QR factorization for tile $A_{1,1}$. The factorization output of $A_{1,1}$ is then used to update the set of tiles on $A_{1,1}$'s right hand side in an embarrassingly parallel fashion, that is, $\{A_{1,2}, \dots, A_{1,n_b}\}$. As soon as the update on any tile $A_{1,j}$ is finished, the update on tile $A_{2,j}$ can read the modified $A_{1,j}$ and proceed. In other words, whenever a tile-update on the i-th row completes, its below tile on the $(i+1)$-th row can start if $A_{i+1,1}$ also completes. After updating the tiles on the last m_b-th row, tile QR applies the same steps to the trailing submatrix $A_{2:m_b,2:n_b}$ recursively. Figure 1 illustrates the data dependency relationships between tasks during the first iteration given a 5×5 tiled matrix. Each tile located at [i, j] corresponds to a task that reads a couple of inputs and modifies A[i, j]. For instance, tile A[2, 4] corresponds to a task that reads the output of two tasks located at [1, 4] and [2, 1] and then modifies A[2, 4]. In the tile QR factorization, the tasks within each row can be executed in embarrassingly parallel. However, the sequential dependency between tasks along a column clearly makes the algorithm inefficient, especially for tall and skinny matrices.

Hadri et al. recently presented a strategy to compute the QR factorization on shared-memory multicore machines for tall and skinny matrices [6]. Their approach considerably increases the number of parallel tasks located in the same column. Their work was inspired by the tile QR factorization (available in PLASMA) and a communication-avoiding technique (known as CAQR) that was introduced by Demmel [7]. However, this algorithm has never been explored on distributed-memory systems. We investigate and extend the algorithm to modern large-scale distribute-memory machines and demonstrate its high efficiency and scalability. We call the distributed-version algorithm "distributed tile communication-avoiding QR factorization". In short, we refer to it as "distributed tile CA-QR factorization."

In this paper, we also analyze the tile CA-QR factorization in terms of operation count, number of messages, and communication volume. We then compare the algorithm to previous work such as LAPACK, ScaLAPACK, tile QR (PLASMA), TSQR [7], as well as CAQR [7]. Tile CA-QR has an operation count that is between tile QR and TSQR and could be comparable to that of LAPACK/ScaLAPACK by choosing appropriate parameters. Same as TSQR, tile CA-QR's communication volume is also optimal. Furthermore, its number of messages is much less than that of tile QR.

The distributed tile CA-QR factorization partitions a matrix's rows into D blocks of rows (i.e., D domains). Then on a distributed-memory system with P compute nodes, it continues to partition the D domains into P subsets (one subset per node) using a 1D block distribution, where $D \geq P$. Each node runs a single MPI process and is responsible for computing a number $\frac{D}{P}$ of domains. For each column panel (of a tile width), the factorization algorithm performs independent QR factorizations in each domain by different processes in parallel. Then, each domains updates its trailing submatrix concurrently. Third and last step, the local R factors from each domain are reduced by different processes to the final R factor and the corresponding block-rows are again updated.

The reduction operation among the domains adopts a binary-tree to attain the final R factor. Due to the complex binary-tree reduction residing on the critical path of the computation's task graph, we extended our dynamic scheduling runtime system to support distributed tile CA-QR more efficiently. We added new features such as look-ahead depth and three levels of task priority to the runtime system. A collection of trace analysis show that the new scheduling runtime system has been improved significantly.

This paper evaluates the efficiency of distributed tile CA-QR by comparing it to vendor optimized ScaLAPACK libraries. We conducted both strong-scalability and weak-scalability experiments on two Beowulf clusters and a Cray XT5 system consisting of hundreds of thousands of 12-core nodes. The experimental results show that our program is able to outperform ScaLAPACK by up to 4 times, and exhibits good scalability from 1 to 3,072 cores (3,072 cores is the largest experiment we have attempted).

This paper includes the following new and original work: (1) A major extension and improvement from shared-memory systems to distributed-memory systems. (2) First to analyze the algorithm with respect to operation count, number of messages, and communication volume. (3) An extended runtime

system to enable an efficient implementation of the distributed algorithm. (4) First to demonstrate good scalability of the algorithm on modern large-scale distributed-memory systems using up to 3,072 cores.

The rest of the paper is organized as follows. Section II introduces the related work. Sections III and IV describe the tile CA-QR factorization algorithm and the analysis of the algorithm, respectively. Section V provides an overview of the dynamic scheduling runtime system and explains our extensions. Section VI presents the performance evaluation on three distributed-memory systems. Section VII summarizes our work.

II. Related Work

In the mid 70s, Morven Gentleman introduced for sparse matrices [8] the approach of splitting a matrix into submatrices allowing the reduction to be done independently and recursively for the submatrices. Then, Pothen and Raghavan [9] developed the idea of parallelizing the factorization of a panel by implementing distributed orthogonal factorizations using Householder and Givens algorithms. Their approach divides the columns into P subcolumns (where P is the number of processors) and performs factorizations locally from which the final triangular factors are merged.

Based on Pothen and Raghavan's work, Demmel et al. [7] proposed a class of QR factorizations with the parallel panel factorization, called Communication-Avoiding QR (CAQR). The approach consists of performing the panel factorization on several columns thanks to a new algorithm called TSQR (Tall Skinny QR). The panels are divided into block-rows, and they are factorized independently and then merged using a binary reduction tree, which is optimal in the parallel case [7]. An estimate of the performance for CAQR has been provided by the authors.

Assuming that the QR factorization of a tall and skinny matrix can be represented as a reduction, Langou [10] implemented a methodology to perform the reduction by using user-defined MPI operation and MPI_Reduce. Moreover, in the context of grid computing, by identifying bottlenecks in ScaLAPACK, Agullo et al. [11] developed an approach to computing the QR factorization by articulating the CAQR factorization with a topology-aware middleware in order to confine intensive communications. Contrary to all the previous work on QR, they have used more original trees instead of the binary tree.

III. Tile CA-QR Factorization

Essentially the tile CA-QR factorization is an integration (or mixed version) of the CAQR factorization and the tile QR factorization. The basic idea is to store a matrix in a tile data layout and divide the matrix into a number of domains (i.e., blocks of rows). Each domain performs a local QR factorization independently. After finishing the local QR factorization, each domain participates in a global reduction to compute the final R factor.

Suppose an $m \times n$ matrix A consists of $m_b \times n_b$ tiles ($m > n$), and b is the tile size for which $m_b = \frac{m}{b}$ and $n_b = \frac{n}{b}$. Tile CA-QR partitions the matrix's m rows into D blocks: $A = [A_1; A_2; \ldots; A_D]$, where A_i is of dimension $\frac{m}{D} \times n$ and is called "Domain i." Note that the matrix A is stored in $b \times b$ tiles. The tiled matrix A that is divided into D horizontal domains can be expressed as follows:

$$A = \left(\begin{array}{cccc} A_{1,1} & A_{1,2} & \ldots & A_{1,n_b} \\ \ldots & \ldots & \ddots & \ldots \\ A_{\frac{m_b}{D},1} & A_{\frac{m_b}{D},2} & \ldots & A_{\frac{m_b}{D},n_b} \\ \hline A_{\frac{m_b}{D}+1,1} & A_{\frac{m_b}{D}+1,2} & \ldots & A_{\frac{m_b}{D}+1,n_b} \\ \ldots & \ldots & \ddots & \ldots \\ A_{\frac{2m_b}{D},1} & A_{\frac{2m_b}{D},2} & \ldots & A_{\frac{2m_b}{D},n_b} \\ \hline \vdots & \vdots & \ddots & \vdots \\ A_{m_b,1} & A_{m_b,2} & \ldots & A_{m_b,n_b} \end{array}\right),$$

where $A_{i,j}$ is a tile of size $b \times b$. In the first step, all the domains start to execute the tile QR factorization of the first panel and the associated updates concurrently as shown in Fig. 1. There is no data dependency or communication between different domains. That is, each domain is independent of the other domains. After the QR factorization of the first panel within each domain is finished, each domain i gets a $b \times b$ upper triangular factor $\hat{R}_i$ located at $A_{(i-1)\times\frac{m_b}{D}+1,1}$. For instance, $\hat{R}_1$ is located at $A_{1,1}$ and $\hat{R}_2$ is located at $A_{\frac{m_b}{D}+1,1}$. Note that all the $\hat{R}_i$'s belong to the first block-column for the first iteration. Next, the tile CA-QR factorization performs a reduction among all the $\hat{R}_i$'s, where $i \in \{1, \ldots, D\}$. The output of the reduction is the final factor of $R_{1,1}$ assuming $A = QR$ and R is stored in tiles. Then the final $R_{1,1}$ will be applied to the top block-row $\{A_{1,2}, \ldots, A_{1,n_b}\}$ to compute $\{R_{1,2}, \ldots, R_{1,n_b}\}$. The next step of the factorization can be initiated on $A_{2:m_b,2:n_b}$ while the previous step is still in process as long as the dependencies are satisfied. The factorization steps are therefore pipelined which can potentially hide the light points of synchronizations required during the merging procedure.

Before describing the distributed tile CA-QR factorization, we briefly overview the six kernel subroutines used by the factorization. For more details of these kernels, please refer to Section 3 of the Hadri et al. paper [6].

The first four kernel subroutines are called locally within each domain.

- `dgeqrt`: R[k,k], V[k,k], T[k,k] ← dgeqrt(A[k,k])
  dgeqrt computes the QR factorization of a tile A[k,k] and generates three outputs: an upper triangular tile R[k,k], a unit lower triangular tile V[k,k] containing the Householder reflectors, and an upper triangular tile T[k,k] for storing the accumulated transformations.
- `dtsqrt`: R[k,k], V[i,k], T[i,k] ← dtsqrt(R[k,k], A[i,k])
  After dgeqrt is called, dtsqrt stacks tile R[k,k] on top of tile A[i,k] and computes an updated QR factorization.

The subroutine updates the tile R[k,k] and generates a tile V[i,k] and an upper triangular tile T[i,k]. V[i,k] and T[i,k] also store the Householder reflectors and accumulated transformations, respectively.

- `dormqr`: R[k,j] ← dormqr(V[k,k], T[k,k], A[k,j])
  dormqr applies dgeqrt's output (i.e. V[k,k], T[k,k]) to tile A[k,j] located on the right hand side of A[k,k] and computes the R factor R[k,j].
- `dtsssmqr`: R[k,j], A[i,j] ← dtsssmqr(V[i,k], T[i,k], R[k,j], A[i,j])
  dtsssmqr applies dtsqrt's output (i.e, V[i,k], T[i,k]) to a stacked R[k,j] and A[i,j], and then updates the R factor R[k,j] and A[i,j], respectively.

The algorithm `Domain_Tile_QR` applies the four kernel subroutines to factorize a domain of size nrows × ncols tiles starting from the position A[I, J]. For instance, Fig. 1 can be viewed as a single domain that applies this algorithm. Note that here I, J are indexed from 0.

Algorithm 1 Domain_Tile_QR Algorithm

```
Domain_Tile_QR(A, I, J, nrows, ncols)
R[I,J], V[I,J], T[I,J] ← dgeqrt(A[I,J])
for j ← J+1 to J+ncols-1 /*I-th row*/ do
    A[I,j] ← dormqr(V[I,J], T[I,J], A[I,j])
end for
for i ← I+1 to I+nrows-1 /*J-th column*/ do
    R[I,J], V[i,J], T[i,J] ← dtsqrt(A[I,J], A[i,J])
end for
for i ← I+1 to I+nrows-1 /*trailing submatrix update*/ do
    for j ← J+1 to J+ncols-1 do
        R[I,j],A[i,j]←dtsssmqr(V[i,J],T[i,J],R[I,j],A[i,j]
    end for
end for
```

The remaining two kernel subroutines are used in the reduction step that involves merging a collection of domains.

- `dttqrt`: R[i_1,k],V[i_2,k],T[i_2,k]←dttqrt(R[i_1,k],R[i_2,k])
  This is the "merge" operation. dttqrt stacks one domain's factor R[i_1,k] on top of another domain's factor R[i_2,k] and computes an updated factor R[i_1,k]. It also generates an upper triangular tile V[i_2,k] and an upper triangular tile T[i_2,k].
- `dttssmqr`: A[i_1,j], A[i_2,j] ← dttssmqr(V[i_2,k], T[i_2,k], A[i_1,j], A[i_2,j])
  After dttqrt is called, dttssmqr applies the output of dttqrt to update A[i_1,j] and A[i_2,j] (j ∈ $[k+1, n_b]$) that are located on the right hand side of R[i_1,k] and R[i_2,k], respectively.

The algorithm `Merge_Domains` merges two R factors from a pair of domains.

Algorithm 2 Merge_Domains Algorithm

```
Merge_Domains(R, A, i1, i2, k, ncols)
/*merge two R factors from two domains*/
R[i1,k], V[i2,k], T[i2,k] ← dttqrt(R[i1,k], R[i2,k])
/*update the i1-th and i2-th rows*/
for j ← k+1 to k+ncols-1 do
    A[i1,j], A[i2,j] ← dttssmqr(V[i2,k], T[i2,k], A[i1,j], A[i2,j])
end for
```

Distributed Tile CA-QR Factorization: Given P processes on a distributed-memory system, we distribute the matrix's D domains across different processes by 1-D block distribution. Each process P_i owns a number $\frac{D}{P}$ of domains from $D_{\frac{D}{P}i}$ to $D_{\frac{D}{P}(i+1)-1}$. Although D is a parameter used at the algorithm level, we assume $D \geq P$ so that a process owns at least one domain. A process may consist of one or more threads running on multiple cores. The algorithm of the distributed tile CA-QR factorization is shown as follows:

Algorithm 3 Distributed_Tile_CAQR Algorithm

```
Distributed_Tile_CAQR(A, m_b, n_b, D, P)
nr ← m_b/P /*number of rows per process*/
nd ← D/P /*number of domains per process*/
ds ← m_b/D /*domain size*/
for each tile column k ← 0 to n_b − 1 do
    root ← ⌊k/ds⌋ /*index of the root domain*/
    /*process P_my could factorizes its nd domains in parallel*/
    for each domain i ← 0 to nd − 1 do
        if (d = my × nd + i) < root then
            if d = root then
                I ← k
            else
                I ← my × nr + i × ds
            end if
        end if
        /*[I,k] is the top left corner of domain d*/
        Domain_Tile_QR(A, I, k, (my+1)×nr-I, n_b-k)
    end for
    /*binary-tree merge*/
    LB ← my × nd, UB ← LB+nd-1
    for m ← 1 to ⌈log_2(D − root)⌉ do
        d1 ← root, d2 ← d1 + 2^(m−1)
        while d2 < D do
            if both d1, d2 ∉ [LB, UB] then
                P1 ← d1/nd, P2 ← d2/nd
                if d1 = root then
                    i1 ← k
                else
                    i1 ← d1 × ds
                end if
            end if
            i2 ← d2 × ds
            processes P1 and P2 exchange R[i1,k], R[i2,k]
            Merge_Domains(R, A, i1, i2, k, n_b-k)
            d1 += 2^m, d2 += 2^m
        end while
    end for
end for
```

Figure 2 illustrates the operations of `Distributed_Tile_CAQR`. It shows a matrix of 12 × 3 tiles that is distributed across four domains. Each domain is stored and computed by one process and has a submatrix of 3 × 3 tiles. The figure shows the corresponding operations in the first iteration. That is, each domain invokes `Domain_Tile_QR` in parallel followed by a binary-tree merge between the first panels of each domain. The second iteration would be the same as the first iteration except for working on a trailing submatrix of size 11 × 2 tiles.

IV. Algorithm Analysis

In this section, we present the total number of operations for the sequential tile CA-QR factorization and the number of messages and the communication volume for the distributed

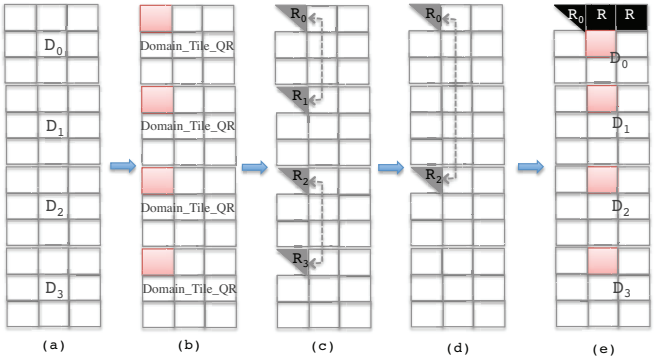


Fig. 2. The operations of distributed tile CA-QR. (a) Matrix A is divided into four domains horizontally. (b) We now apply `Domain_Tile_QR` to each domain in parallel. (c) Each domain computed a R factor located in the first column. We merge R_0 and R_1, R_2 and R_3. (d) We merge R_0 and R_2 and get the final factor R_0. (e) At the beginning of the second iteration, domain D_0 has 2×2 tiles and the other three domains have 3×2 tiles. Similarly, we continue to apply (b), (c), (d) to the four new domains in the trailing submatrix.

tile CA-QR factorization. We also compare the metrics with a number of related QR factorization algorithms.

A. Operation Count

We use aggregate analysis to calculate the number of operations for each kernel. Note that each kernel takes as input tiles of size $b \times b$.

dgeqrt does $2b^3$ floating point operations. For each iteration k, dgeqrt is invoked $D - \frac{k-1}{m_b/D}$ times. There are n_b iterations, thus

$$\begin{aligned} T_{dgeqrt} &= \sum_{k=1}^{n_b}(D - \frac{k-1}{m_b/D}) \times 2b^3 \\ &= 2b^3 n_b D - b^3(\frac{n_b-1}{m_b/D})(2n_b - \frac{m_b}{D} - \frac{m_b}{D}\frac{n_b-1}{m_b/D}) \end{aligned}$$

dormqr does $3b^3$ floating point operations. At iteration k, there exist $D - \frac{k-1}{m_b/D}$ domains each of which has a number $n_b - k + 1$ of tile columns. Every domain applies dormqr to all the tiles on its top row except for the first tile.

$$T_{dormqr} = \sum_{k=1}^{n_b}(D - \frac{k-1}{m_b/D})(n_b - k) \times 3b^3 \simeq \frac{3}{2}b^3 D n_b^2$$

dtsqrt does $\frac{10}{3}b^3$ floating point operations. At iteration k, there exist $D - \frac{k-1}{m_b/D}$ domains and one of them is the root domain. The root domain has $\frac{m_b}{D} - (k \bmod \frac{m_b}{D}) + 1$ tile rows and the other domains has $\frac{m_b}{D}$ tile rows.

$$\begin{aligned} T_{dtsqrt} &= \sum_{k=1}^{n_b}(\frac{m_b}{D} - k \bmod \frac{m_b}{D} + (D - \frac{k-1}{\frac{m_b}{D}} - 1)\frac{m_b}{D})\frac{10b^3}{3} \\ &\simeq (2m_b n_b - n_b(n_b + \min(n_b, \frac{m_b}{D})))\frac{5}{3}b^3 \end{aligned}$$

dtsssmqr does $4b^3 + sb^2$ floating point operations, where s is a parameter used to implement dtsssmqr. s is the inner blocking size which divides the tile size. At iteration k, there exist $D - \frac{k-1}{m_b/D}$ domains. The root domain consists of $\frac{m_b}{D} - (k \bmod \frac{m_b}{D}) + 1$ tile rows and $n_b - k + 1$ tile columns. The remaining domains consist of $\frac{m_b}{D} \times (n_b - k + 1)$ tiles.

$$\begin{aligned} T_{dtsssmqr} &= (4b^3 + sb^2)\sum_{k=1}^{n_b}[(\frac{m_b}{D} - k \bmod \frac{m_b}{D})(n_b - k) + \\ &\quad (D - \frac{k-1}{m_b/D} - 1)\frac{m_b}{D}(n_b - k)] \\ &= 2b^3 n_b^2(1 + \frac{s}{4b})(m_b - \frac{n_b}{3}) \end{aligned}$$

dttqrt is the merge operation and does $\frac{5}{3}b^3$ floating operations. At iteration k, the binary tree has $D - k + 1$ leaf nodes and $D - k$ internal nodes.

$$T_{dttqrt} = \sum_{k=1}^{n_b}(D - k)\frac{5}{3}b^3 = \frac{5}{6}b^3 n_b(2D - n_b)$$

dttssmqr does $\frac{1}{2}(4b^3 + sb^2)$ floating point operations. Every merge operation dttqrt is followed by a number $n_b - k$ of dttssmqr operations.

$$\begin{aligned} T_{dttssmqr} &= \sum_{k=1}^{n_b}(D - k)(n_b - k)\frac{1}{2}(4b^3 + sb^2) \\ &= 2b^3 n_b^2(1 + \frac{s}{4b})(\frac{D}{2} - \frac{n_b}{6}) \end{aligned}$$

The total number of operations of tile CA-QR is the sum of the above six equations:

$$\begin{aligned} T_{tile-caqr} &= T_{dgeqrt} + T_{dormqr} + T_{dtsqrt} \\ &\quad + T_{dtsssmqr} + T_{dttqrt} + T_{dttssmqr} \\ &\simeq 2n^2(1 + \frac{s}{4b})(m - \frac{n}{2} + \frac{Db}{2}) \end{aligned}$$

Compared to the operation count of the tile QR factorization [12], that is, $T_{tile-qr} = 2n^2(1 + \frac{s}{4b})(m - \frac{n}{3})$, then

$$\frac{T_{tile-caqr}}{T_{tile-qr}} = \frac{2n^2(1 + \frac{s}{4b})(m - \frac{n}{2} + \frac{Db}{2})}{2n^2(1 + \frac{s}{4b})(m - \frac{n}{3})} = 1 + \frac{3Db - n}{6m - 2n}.$$

Based upon the above equation, we can make the following observations:

- if $m \gg n$, $\frac{T_{tile-caqr}}{T_{tile-qr}} \simeq 1 + \frac{1}{2\frac{m_b}{D}}$. Note that $\frac{m_b}{D}$ is the domain size in terms of tiles and is often not small.
- if $D = m_b$, $\frac{T_{tile-caqr}}{T_{tile-qr}} = 1 + \frac{3m-n}{6m-2n} = 1.5$.

B. Number of Messages

We compute for the process that has the maximum number of messages. We know that communication only occurs during the binary tree merge where the dttqrt and dttssmqr operations are called. Given P processes, for each iteration k, a process is involved in at most $\log_2 P$ merge stages, thus,

$$\begin{aligned} Message_{tile-caqr} &= \sum_{k=1}^{n_b}\log_2(P)(n_b - k + 1) \\ &\simeq \log_2(P)n_b^2 = \log_2(P)\frac{n^2}{b^2}. \end{aligned}$$

C. Communication Volume

Similar to computing the number of messages, we compute for the process that has the maximum number of words communicated with other processes. Since each message contains an upper triangular tile of $\frac{b^2}{2}$ words,

$$Word_{tile-caqr} = \frac{1}{2}\log_2(P)n^2.$$

D. Comparison with Other Algorithms

We compare tile CA-QR with LAPACK, ScaLAPACK, tile QR, TSQR, and CAQR factorizations for tall and skinny matrices. The numbers for TSQR and CAQR are provided by Demmel's paper [7]. As for ScaLAPACK and CAQR, we let $P_r \gg P_c$ assuming a very tall and skinny matrix input.

We have implemented the tile QR factorization on distributed-memory systems in our previous work [13]. We briefly introduce it here. The distributed tile QR factorization maps tiles to a $P_r \times P_c$ process grid using the 2-D block cyclic data distribution. $P = P_r \times P_c$ is the total number of processes. A tile indexed by [i, j] will be allocated to the process $P[i \mod P_r, j \mod P_c]$ so that each process stores a set of tiles and computes the tasks that modify the tiles. We skip the calculation of the number of messages and words for tile QR and just give the result in Table I.

As shown in Table I, from the least to the most operations are LAPACK, ScaLAPACK, CAQR, tile QR, tile CA-QR, and TSQR. LAPACK is a library used for share-memory systems and thus does not have any communication. Although TSQR has the minimum number of messages, it uses a much larger tile size such that $b = n$ given an $m \times n$ matrix. CAQR also has a smaller number of messages than tile CA-QR, but the algorithm typically uses the fork-join approach and is not suited for dynamic scheduling (e.g., the whole step of panel factorization must be completed before the step of trailing matrix update can start). Differently, tile CA-QR provides more fine grain tasks operating on tiles and can be executed in a fully asynchronous manner. Furthermore, the communication volume $\frac{n^2}{2}\log P$ for the QR factorization on tall and skinny matrices has been proven to be optimal [7].

TABLE I
ALGORITHM COMPARISON

	Seq. operation count	#Messages	#Words
LAPACK	$2n^2(m-\frac{n}{3})$	–	–
ScaLAPACK	$2n^2(m-\frac{n}{3})$	$3n\log P$	$(n^2+bn)\log P$
TSQR	$2n^2(m+(\frac{D-1}{2}-\frac{1}{3})n)$	$\log P$	$\frac{n^2}{2}\log P$
CAQR	$2n^2(m-\frac{n}{3})$	$\frac{3n}{b}\log P$	$(n^2+\frac{bn}{2})\log P$
Tile QR	$2n^2(m-\frac{n}{3})(1+\frac{s}{4b})$	$(\frac{n}{b})^2\frac{P_r}{P_c}\frac{m}{b}$	$n^2\frac{P_r}{P_c}\frac{m}{b}$
Tile CA-QR	$2n^2(m-\frac{n}{2}+\frac{Db}{2})\times(1+\frac{s}{4b})$	$(\frac{n}{b})^2\log P$	$\frac{n^2}{2}\log P$

V. THE DISTRIBUTED FRAMEWORK

We build upon our previous work of Task-based Basic Linear Algebra Subroutines (TBLAS) dynamic runtime system [13] to realize tile CA-QR on distributed-memory systems. This section first overviews the TBLAS runtime system, then describes how we extend TBLAS to support tile CA-QR efficiently.

Given a matrix A of $m_b \times n_b$ tiles and a multicore cluster consisting of N nodes each with T cores, we launch on each node N_i a process P_i, respectively. The rows of matrix A are preallocated to N nodes by 1D block distribution. That is, P_i (on $nodeN_i$) stores a submatrix of A from $(\frac{m_b}{N}i)$-th to $(\frac{m_b}{N}(i+1)-1)$-th tile-rows. Note that by default TBLAS uses a general 2D block cyclic data distribution. But the 1D data distribution which is a special case of 2D data distribution is more suitable for tall and skinny matrices.

A. TBLAS Runtime System

Every process runs an instance of the TBLAS runtime system in parallel, which are started by `mpirun`. As shown in Fig. 3, the TBLAS runtime system includes three types of threads: task-generation thread, computing thread, and communication thread. Given a node with T cores, we launch T computing threads on T different cores, as well as a task-generation thread and a communication thread on two arbitrary cores. The task-generation thread executes a tile CA-QR program and generates tasks to fill in its node's local task queues. Also, whenever becoming idle, a computing thread picks up a ready task from the ready task queue and computes it. After finishing a task, the computing thread scans the task queues to resolve data dependency and finds the finished task's children and starts them. The communication thread is responsible for sending and receiving data between a parent task and its children to meet the data dependency demands. An advantage of the tile CA-QR factorization is that we do not need a dedicated core to perform MPI communications because of the high parallelism degree and the minimized communication of the algorithm.

B. Extensions

Our first implementation of tile CA-QR with the original TBLAS runtime system did not yield good performance automatically. By profiling the execution using the Intel trace analyzer and collector [14], we found that each core's computing time is only half of the wall-clock execution time, which implies there is a nearly 50% idle time on each core.

Figure 4 a) shows an example trace of the first version of tile CA-QR running on 16 dual-core nodes. The colored regions represent the computation time, and the gaps represent the idle time during the execution. By analyzing the trace, we found a few reasons for the poor performance. 1) In the program's corresponding task graph, between domains, tasks from two iterations (i.e., from i-th and i+1-th panels) are connected by tasks computing the global binary-tree reduction across domains. The merge tasks must be executed earlier in order to pull tasks from the next iteration to execute. 2) Within a

(a) The original version (b) An improved version (c) The final version

Fig. 4. Traces for tuning the TBLAS runtime system. The colored regions denote computation time and the empty gaps denote idle time. After applying a number of modifications, the final version of the TBLAS runtime system has much less idle time and is faster than the original version by 35%.

domain, the panel factorization tasks should also be executed as early as possible because many trailing-matrix update tasks are awaiting a single panel-factorization task. 3) Lookahead to the next d iterations can not only pull tasks from the next iteration but also from the next d iterations.

Essentially we want to make sure the TBLAS runtime system executes the tasks on the critical path as early as possible. We modified the runtime system in the following ways:

- We added the lookahead feature to the runtime system. The lookahead depth d is a parameter to the runtime system and has been tuned to provide the best performance.
- We assign priorities to different tasks. The binary-tree merge tasks have the highest priority. At iteration i, the tasks located between the i-th column and (i+d)-th column have the 2nd highest priority given a lookahead depth of d. The remaining tasks have a regular priority.
- We also added message priorities to the communication subsystem of the runtime system. The output of a high priority task will be assigned a high priority accordingly and sent out by the communication thread earlier than the other messages. Similarly, the receiver will process the high priority message earlier too.
- The task window size has been tuned to optimize the program performance. With a small window size, the runtime system is not able to see tasks in the other domains and the following iterations so that there is a lesser degree of parallelism. But a large window size will increase the runtime system overhead due to longer queues and length access time to search for and resolve data dependencies in the queues.

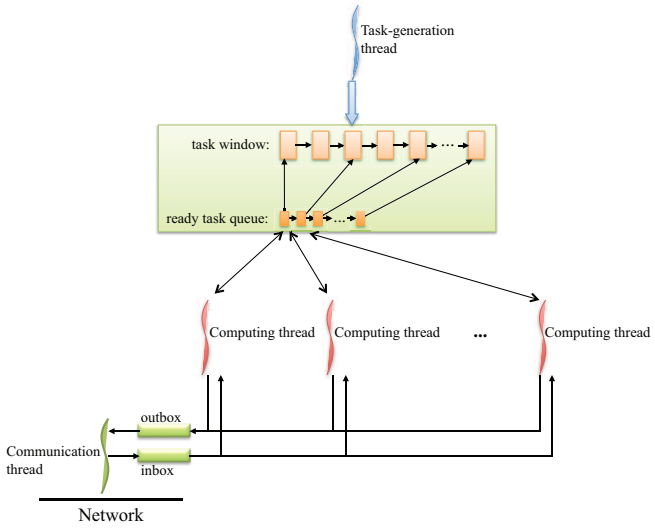


Fig. 3. TBLAS runtime system.

Figure 4 displays examples of traces for three different versions of the runtime system. Figure 4 a) shows the trace of the original version that has significant idle time. After setting appropriate task priorities, the performance is improved by 27% as shown in b). Figure 4 c) shows the trace of our final optimized version after applying all the above modifications and tunings. The final version is better than the original one by 35%. It is easy to see the significantly reduced empty gaps (i.e., idle time) in the figure.

VI. Performance Evaluation

In this section, we provide strong scalability and weak scalability performance results on three different distributed-memory machines. We also present the crossover point of the tile CA-QR implementation for matrices that are not tall and skinny.

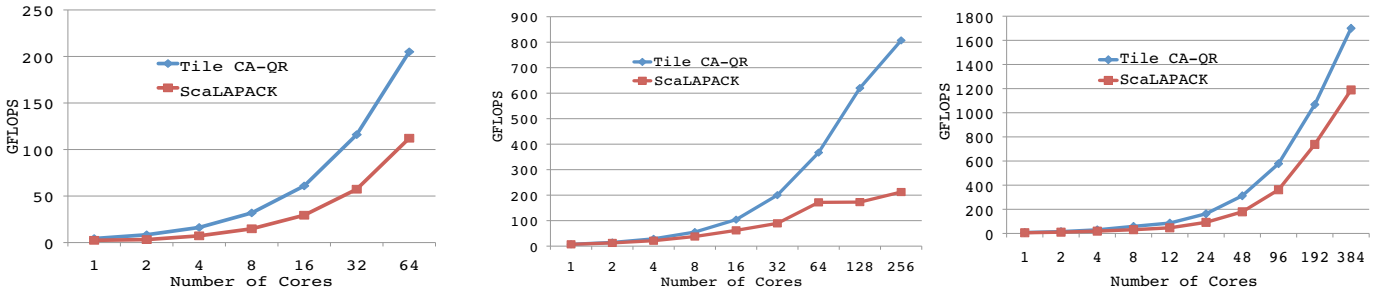
We conducted experiments on two Beowulf clusters (`Grig` and `Newton` at University of Tennessee) and a Cray XT5 system (`Jaguar` at Oak Ridge National Laboratory) to compare tile CA-QR with the ScaLAPACK library. Whenever possible, we use a vendor-optimized ScaLAPACK library. Table II lists the hardware and software resources we used to do our experiments. The Grig cluster has two cores per node, the Newton cluster has eight cores, and the Cray XT5 system has 12 cores per node. On Newton and the Cray XT5 system, we use Intel MKL and Cray XT LibSci libraries to conduct ScaLAPACK experiments, respectively.

A. Strong Scalability

For strong scalability experiments, we fix the matrix input size and increase the number of cores to solve the matrix.

TABLE II
EXPERIMENT RESOURCES.

	Grig cluster	Newton cluster	Cray XT5
Processor	Intel Xeon 3.2GHz	Intel Xeon E5530 2.4GHz	AMD Opteron 2.6GHz
Cores per processor	1	4	6
Processors per node	2	2	2
Nodes	60	170	18,688
Memory per nod	4 GB	16 GB	16 GB
Peak perf. per core	6.4 GFLOPS/s	9.6 GFLOPS/s	10.4 GFLOPS/s
Network	Myrinet	Infiniband	Cray SeaStar2+
OS	Linux 2.6	Scientific Linux 5.3	Compute Node Linux 2.2
Compilers	gcc 64bit 3.4.4	Intel compilers 11.0	PGI 9.0.4
MPI lib	mpich-mx 1.1	OpenMPI 1.2.8	Cray XT MPT 3.5.1
ScaLAPACK lib	Netlib scalapack 1.8	Intel MKL 10.1	Cray XT LibSci 10.3.6



(a) Grig cluster (b) Newton cluster (c) Cray XT5

Fig. 5. Strong scalability.

Then we compare the total number of GFLOPS between tile CA-QR and ScaLAPACK.

The matrix input to the Newton cluster and Cray XT5 system is of 512×32 tiles with a tuned tile size of $b = 200$. The matrix input to the Grig cluster is a bit smaller (due to its smaller memory), that is, 512×16 tiles with a tile size 200. Since the configuration of a process grid $P_r \times P_c$ can affect the performance of ScaLAPACK significantly, we tried all possible grid configurations and chose the best process grid for ScaLAPACK. Based on our experiments, we also found that running an MPI process on each core has a better performance than running an MPI process on each node with multithreaded computational kernels. Therefore in our ScaLAPACK experiments, each MPI process is single-threaded.

Figure 5 displays the overall performance of tile CA-QR and ScaLAPACK on three systems. On the Grig cluster, as we increase the number of cores from 1 to 64, the performance of tile CA-QR increases from 4.3 GFLOPS to 206 GFLOPS. By contrast ScaLAPACK increases from 2.4 GFLOPS to 112 GFLOPS.

On Newton, between 1 and 128 cores, the performance of tile CA-QR increases from 7.3 GFLOPS to 620 GFLOPS. Then from 128 cores to 256 cores, the increasing rate of tile CA-QR drops and its performance rises from 620 GFLOPS to 810 GFLOPS. The performance of ScaLAPACK is much worse than that of tile CA-QR. In the beginning it rises from 7.1 GFLOPS to 172 GFLOPS (1 to 64 cores), after which it nearly stops increasing.

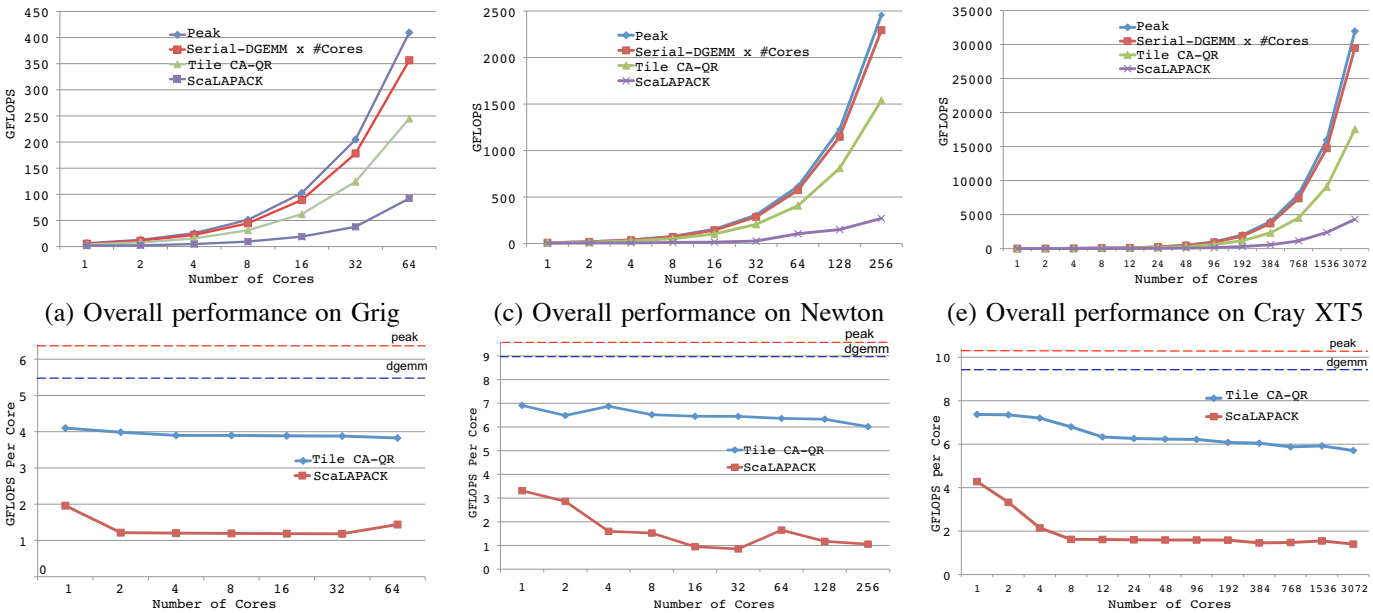
On the Cray XT5 system, with an increasing number of cores from 1 to 384, tile CA-QR improves from 7.5 GFLOPS to 1700 GFLOPS while ScaLAPACK improves from 5.8 to 1180 GFLOPS.

B. Weak Scalability

For weak scalability experiments, we fix the amount of computation on each core. When we double the number of cores, we also double the total amount of computation accordingly. Weak scalability demonstrates a program's ability to solve larger problems with more resources.

In our experiment, each matrix input has a fixed number of eight tile-columns but different number of tile-rows. When we double the number of cores, we double the number of tile-rows in the input. For instance, the input to the single-core experiment has 64×8 tiles. And the two-core experiment has a matrix input of 128×8 tiles.

Figure 6 shows the performance of the weak scalability experiments on three different systems. Besides tile CA-QR and ScaLAPACK, we also display the theoretical peak performance and the serial DGEMM performance times the number of cores for each system. The DGEMM performance serves as an upper bound for all of our experiments. Again for ScaLAPACK, we always choose the best process grid and use the vendor optimized ScaLAPACK library whenever possible.



(a) Overall performance on Grig

(c) Overall performance on Newton

(e) Overall performance on Cray XT5

(b) Per-core performance on Grig

(d) Per-core performance on Newton

(f) Per-core performance on Cray XT5

Fig. 6. Weak scalability.

There are two subfigures for each system. The top subfigure shows the total number of GFLOPS, and the bottom one shows the number of GFLOPS per core (i.e., the total number of GFLOPS divided by the total number of cores). Ideally the number of GFLOPS per core is a constant and does not change from 1 to n cores so that the per-core performance curve is flat.

Figure 6 a) and b) display the overall performance and per-core performance of tile CA-QR and ScaLAPACK on the Grig cluster, respectively. We set the tile size to $b = 200$. As shown in a), as the number of cores increase from 1 to 64, tile CA-QR increases from 4.1 GFLOPS to 244.9 GFLOPS while ScaLAPACK increases from 1.96 to 92.1 GFLOPS. In b), the per-core performance of tile CA-QR keeps at a rate of 4 GFLOPS that outperforms ScaLAPACK by nearly four times.

On the Newton cluster, the ScaLAPACK experiment calls the QR factorization subroutine provided by Intel MKL 10.1. Figure 6 c) shows that the performance of tile CA-QR rises from 6.9 GFLOPS to 1,540 GFLOPS while ScaLAPACK rises only from 3.3 GFLOPS to 270 GFLOPS. In d), the per-core performance of tile CA-QR decreases slightly from 6.9 to 6.4 GFLOPS between 1 and 128 cores, and then drops 0.4 GFLOPS from 128 to 256 cores. ScaLAPACK does not perform as well as tile CA-QR. For instance, the performance of ScaLAPACK on 256 cores is only 1/6 of that of tile CA-QR.

On the Cray XT5 system, we use the ScaLAPACK routine provided by Cray XT LibSci 10.3.6 and let tile size $b = 300$. In Fig. 6 e), with an increasing number of cores from 1 to 3,072, the performance of tile CA-QR increases from 7.4 GFLOPS to 17.5 TFLOPS while ScaLAPACK increases from 4.3 GFLOPS to 4.3 TFLOPS. In Fig. 6 f), the per-core performance of tile CA-QR decreases gradually from 7.4 GFLOPS (with 1 core) to 6.3 GFLOPS (with 12 cores). The reason for the performance drop is related to the NUMA architecture and requires an optimized memory-affinity setup. Afterward tile CA-QR scales well from 12 cores to 3,072 cores (i.e., from 1 node to 256 nodes). By contrast the performance of ScaLAPACK drops from 4.3 GFLOPS to 1.4 GFLOPS as we increase the number of cores, which is 1/4 of that of tile CA-QR.

C. Crossover Point

This section discusses how distributed tile CA-QR behaves if the matrix is not tall and skinny. In our experiment, a matrix has a fixed number of 512 tile-rows but an increasing number of tile-columns. The tile size is set to $b = 200$. Since we want to view the number of columns as a unique variable, we choose to use a fixed number of 192 cores. We conducted the experiment on the Cray XT5 system. Note that 192 cores correspond to 16 nodes.

Figure 7 shows the crossover point when a matrix becomes wider and wider until it is eventually square. We can see that the performance of tile CA-QR becomes worse than that of ScaLAPACK after the number of columns is greater than 1/4 of the number of rows. This is because the matrix's 512 tile-rows have been distributed to 16 processes by the 1D block distribution. Every process is allocated with 32 tile-rows and is only responsible for the computation on its own 32 tile-rows. As the algorithm visits and computes the matrix from top left to bottom right, more and more processes on the top become idle, which results in a load imbalance and poor performance.

Figure 8 shows an example of the tile CA-QR factorization that explains the cause of idle processes. The matrix input has 8×4 tiles and is partitioned across eight processes. We can see from the figure that when the algorithm is working on the third tile-column, processes P_0 and P_1 become idle until the end of

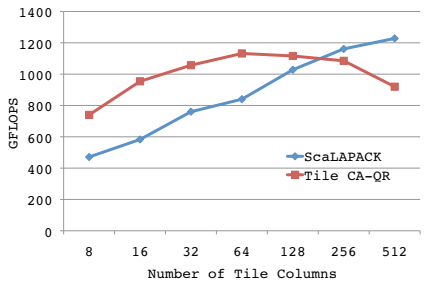


Fig. 7. Crossover point. The input has a fixed number of 512 tile rows.

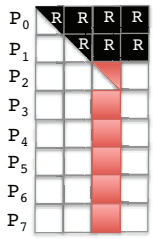


Fig. 8. Existent idle processes given a matrix of 8×4 tiles distributed across eight processes.

the factorization. A two-dimensional block cyclic distribution, similar to ScaLAPACK, would then be necessary to efficiently handle general matrix sizes and overcome this bottleneck. This would also require a revision of the algorithm correspondingly. This is out of the scope of this paper which focuses on how to factorize tall and skinny matrices in a more efficient way.

VII. Conclusion and Future Work

The QR factorization of tall and skinny matrices has been used in many scientific fields that require solving least square problems. This paper extends an existing algorithm for shared-memory architectures and enables it to work efficiently on modern large-scale distributed-memory systems. We have implemented the algorithm with an augmented TBLAS runtime system. The distributed tile CA-QR factorization has a high degree of parallelism and allows for a fully dynamic execution that can overlap computation and communication greatly. We have presented the algorithm, the analysis of the algorithm, the extension of the runtime system, and the performance evaluation. Our experiments on two multicore clusters and a Cray XT5 system demonstrate that the tile CA-QR factorization is scalable on up to 3,072 cores and can outperform the ScaLAPACK library by up to 4 times for tall and skinny matrices.

In summary, we make the following contributions: (1) An extension from shared-memory systems to distributed-memory systems; (2) A detailed analysis of the algorithm with respect to operation count, number of messages, and communication volume; (3) An extended TBLAS runtime system to support an efficient distributed implementation; (4) First demonstration of the scalability of the algorithm on large scale distributed-memory systems. Our future work includes looking for new methods to partition matrices to different processes to improve load balance for general matrix size, and applying this approach to solving other linear algebra problems on distributed-memory multicore systems.

References

[1] 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.

[2] 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.

[3] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker, "ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance," in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1996, p. 5.

[4] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, "PLASMA 2.0 Users' Guide," ICL, UTK, Tech. Rep., 2009.

[5] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 13, pp. 1573–1590, 2008.

[6] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra, "Tile QR factorization with parallel panel processing for multicore architectures," in *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, 2010.

[7] J. W. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," UTK, LAPACK Working Note 204, August 2008.

[8] W. M. Gentleman, "Row elimination for solving sparse linear systems and least squares problems. Numer. Anal.,," *Proc. Dundee Conf. 1975, LEct Notes Math*, vol. 506, pp. 122–133, 1976.

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

[10] L. Julien, "Computing the R of the QR factorization of tall and skinny matrices using MPI reduce," University of Colorado, Denver, Tech. Rep., 2010.

[11] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. Langou, "QR factorization of tall and skinny matrices in a grid computing environment," in *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, 2010.

[12] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, 2009.

[13] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.

[14] Intel, "Intel Trace Analyzer Reference Guide (Revision 8.0)," http://software.intel.com/en-us/intel-trace-analyzer, 2010.