Adapting communication-avoiding LU and QR factorizations to multicore architectures

Simplice DONFACK
INRIA Saclay-Ile de France
Bat 490,Universite Paris-Sud 11
91405 Orsay France
simplice.donfack@lri.fr

Laura GRIGORI
INRIA Saclay-Ile de France
Bat 490,Universite Paris-Sud 11
91405 Orsay France
Laura.Grigori@inria.fr

Alok KUMAR GUPTA
BCCS, Bergen Norway-5075
alok.gupta@bccs.uib.no

***Abstract*—In this paper we study algorithms for performing the LU and QR factorizations of dense matrices. Recently, two communication optimal algorithms have been introduced for distributed memory architectures, referred to as communication avoiding CALU and CAQR. In this paper we discuss two algorithms based on CAQR and CALU that are adapted to multicore architectures. They combine ideas to reduce communication from communication avoiding algorithms with asynchronism and dynamic task scheduling. For matrices that are tall and skinny, that is, they have many more rows than columns, the two algorithms outperform the corresponding algorithms from Intel MKL vendor library on a dual-socket, quad-core machine based on Intel Xeon EMT64 processor and on a four-socket, quad-core machine based on AMD Opteron processor. For these matrices, multithreaded CALU outperforms the corresponding routine dgetrf from Intel MKL library up to a factor of 2.3 and the corresponding routine dgetrf from ACML library up to a factor of 5, while multithreaded CAQR outperforms by a factor of 5.3 the corresponding dgeqrf routine from MKL library.**

***Keywords*-LU and QR factorizations, communication avoiding algorithms, multicore architectures**

I. Introduction

The evolution of multicore processors has lead to an important development in the recent years of new numerical algorithms and libraries. In this paper we present multithreaded algorithms for LU and QR factorizations. These algorithms are based on a family of algorithms, referred to as communication avoiding, that were introduced recently for LU [12] and QR [7], [8], [9] factorizations. These algorithms have two important properties. First, they minimize the amount of communication they perform, where communication refers to both volume of data and number of messages exchanged. Second, they are stable, as classic algorithms for example implemented in LAPACK [2] and ScaLAPACK [4]. CAQR is as stable as QR using Householder transformations. CALU uses a new pivoting strategy, referred to as ca-pivoting. Based on extensive numerical experiments, it is shown in [12] that CALU is as stable as Gaussian elimination with partial pivoting in practice. Important speedups over ScaLAPACK have been observed for CALU [12] and predicted for CAQR [9] on distributed memory systems.

CALU and CAQR are block algorithms that factorize the input matrix by traversing iteratively blocks of columns. At each iteration, a block column is factored and then the trailing matrix is updated. The main insight to reduce communication in these algorithms comes from a different approach of performing the factorization of a block column. This is obtained by a formulation in which the LU and QR decompositions are reorganized into a reduce-like computation. With a flat reduction tree, the algorithms are optimal in the amount of communication they perform in sequential, that is the amount of data transferred between different levels of memory [8], [12]. With a binary reduction tree, they are optimal in the amount of communication they perform in parallel [8], [12], that is the data exchanged between processors. A block column can be seen as a dense matrix for which the number of rows is much larger than the number of columns. As in previous publications on communication-avoiding algorithms, we call these matrices tall and skinny and their LU and QR factorizations as TSLU and TSQR. The idea of performing the QR factorization of tall and skinny matrices using a binary reduction tree was previously introduced in [11], [16], the generalization to any reduction tree and to rectangular matrices is presented in [8], [9].

The multithreaded communication avoiding algorithms combine the main ideas to reduce communication in CALU and CAQR with appropriate blocking, task identification, and dynamic scheduling. The panel factorization is performed using a reduction tree which is binary or a tree of height 1. Hence it can be performed in parallel by T_r threads by using only $O(\log_2 T_r)$ number of synchronizations for a binary tree, at the cost of some redundant computation (one synchronization for a tree of height 1). This reduction compared to classic algorithms is important for multicore processors, since it requires less synchronization between cores, helps avoid bus contention, and hence can lead to better performance. In addition, with this approach most of

This work was supported by X-Scale-NL grant from Digiteo and Région Ile-de-France. The work of the third author was performed as a postdoctoral researcher at INRIA Saclay-Ile de France.

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

the floating-point operations are performed independently by each core, and hence the best available sequential algorithm can be used, as observed in [8], [9], [12].

Recent approaches in the literature for multicore processors [5] focused on exploiting more parallelism by removing the panel factorization from the critical path of the algorithm. In our approach, the panel factorization remains on the critical path. In particular for LU factorization, the stable ca-pivoting strategy requires that the panel factorization lies on the critical path. However with the usage of a communication optimal highly efficient algorithm for this step, the runtime of panel factorization decreases significantly with respect to classic approaches. Hence its presence on the critical path is less important, and it can be hidden by updates of the trailing matrix from previous iterations.

We evaluate the performance of the algorithms on a dual-socket, quad-core machine based on Intel Xeon EMT64 processor and on a four-socket, quad-core machine based on AMD Opteron processor. Our experiments focus mainly on tall and skinny matrices. The QR factorization of tall and skinny matrices is used in several important applications, for example when a set of vectors needs to be orthogonalized as in block iterative methods. A more detailed description is provided in [9]. In particular the LU and QR factorization of these matrices is important since it is a building block for factoring general rectangular or square matrices. We compare the performance of our algorithms with MKL [15] and ACML vendor libraries, and the so called tiled algorithms [5] as implemented in PLASMA [1]. In practice, the multithreaded communication avoiding algorithms lead to significant speedups for tall and skinny matrices.

- Multithreaded CALU is up to 2.3 times faster than the corresponding routine dgetrf from MKL for a $10^6 \times 500$ matrix, and up to 10 times faster than the BLAS2 routine dgetf2 for a $10^6 \times 100$ matrix. For square matrices, MKL tends to outperform CALU on the 8-core Intel machine, while CALU tends to outperform ACML on the 16-core AMD machine.
- Multithreaded TSQR is up to 5.3 times faster than the corresponding routine dgeqrf from MKL for a $10^5 \times 200$ matrix and up to 6.7 times faster than PLASMA for a $10^5 \times 10$ matrix. For larger number of columns, PLASMA tends to become more efficient than CAQR and MKL. We note that we are still working on improving the performance of CAQR.

The paper is organized as follows. Section II presents briefly the main idea behind communication avoiding algorithms. Section III introduces the multithreaded CALU and CAQR algorithms and discusses various related aspects as blocking and dynamic scheduling. Section IV evaluates the performance of multithreaded CALU and CAQR on a two-socket, quad-core machine based on Intel Xeon EMT64 processor and on a four-socket quad-core machine based on AMD Opteron processor. And finally section V concludes the paper.

II. Communication avoiding LU and QR factorizations

In this section we briefly describe the communication avoiding LU and QR factorizations introduced in [12], [7], referred to as CALU and CAQR. We note a matrix A partitioned in T_r block rows as $A = [A_1; A_2; \cdots ; A_{T_r}]$.

We first describe CALU [12], which represents a family of algorithms for computing the factorization $\Pi A = LU$, where A is an $m \times n$ matrix, L is a unit lower triangular matrix, U is an upper triangular matrix, and Π is a permutation matrix. CALU factorizes the input matrix A by traversing iteratively its blocks of columns. At the first iteration, the matrix A is partitioned as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad (1)$$

where A_{11} is of size $b \times b$, A_{21} is of size $(m-b) \times b$, A_{12} is of size $b \times (n-b)$ and A_{22} is of size $(m-b) \times (n-b)$. The LU factorization of the first block column $\Pi_1[A_{11}; A_{21}] = [L_{11}; L_{21}]U_{11}$ is first performed. The first block row U_{12} is computed, and then the trailing matrix A_{22} is updated. This can be written as:

$$\Pi_1 A = \begin{bmatrix} L_{11} & \\ L_{21} & I_{n-b} \end{bmatrix} \cdot \begin{bmatrix} I_b & \\ & \bar{A}_{22} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix}$$

The factorization then continues on the trailing matrix $\bar{A}_{22}$. The factorization of each block column is referred to as TSLU (also as panel factorization).

The main difference between CALU and other classic LU factorization algorithms lies in the panel factorization. In classic LU factorization using partial pivoting, the computation of each column of L is performed by first permuting the element of maximum magnitude in this column to the diagonal position (called pivot and the corresponding row is called a pivot row). The permutations are reflected in the matrix Π_1. In a distributed memory machine, when the block column is distributed over several processors, the panel factorization is computed by more than one processor. Then each column computation involves finding the element of maximum magnitude, and hence requires communication among processors. In a shared memory machine, when the panel factorization is performed by more than one thread, each column computation requires a synchronization between threads.

CALU decreases the communication/synchronization required for this step by dividing the factorization into two steps. In a first preprocessing step, CALU identifies b rows which can be used as pivots for the factorization of the whole panel. In a second step, these pivots are moved in the diagonal positions, and then the factorization of the entire block column is performed.

The preprocessing step is performed as a reduction operation, where each step of the reduction consists of performing Gaussian elimination with partial pivoting (GEPP) to choose b pivot rows. The panel is partitioned in T_r block-rows $[A_1; A_2; \ldots; A_{T_r}]$. At the leaves of the reduction tree, b rows are chosen from each block A_i by computing GEPP independently for each block A_i.

Different reduction trees can be used during CALU. It is shown in [12] that CALU based on a binary tree is optimal in the amount of communication it performs in parallel. We illustrate in the following picture a binary tree of height two, when the panel is partitioned in 4 block rows. We use an arrow notation similar to [8], [9], with the following meaning. The function $f(A)$ computes Gaussian elimination with partial pivoting of A and returns the b rows used as pivots. In other words, it returns $\Pi A(1:b, 1:b)$, where $\Pi A = LU$ is the factorization obtained by Gaussian elimination with partial pivoting. The matrix A is formed by stacking atop one another the b rows obtained by the operation performed at the left side of the arrow.

$$\begin{array}{llll} A_1 \rightarrow f(A_1) & \searrow & & \\ & & f(A_5) & \searrow \\ A_2 \rightarrow f(A_2) & \nearrow & & \\ & & & f(A_7) \\ A_3 \rightarrow f(A_3) & \searrow & & \nearrow \\ & & f(A_6) & \\ A_4 \rightarrow f(A_4) & \nearrow & & \end{array}$$

CALU has several important characteristics. First, when $b = 1$ or $T_r = 1$, CALU is equivalent to partial pivoting. Second, the elimination of each column of A leads to a rank-1 update of the trailing matrix. This property is believed to be important for the numerical stability of Gaussian elimination. Third, extensive numerical tests presented in [12] show that it is as stable as Gaussian elimination with partial pivoting in practice.

We now present CAQR, a family of algorithms to compute the QR factorization of an $m \times n$ matrix A. Similar to CALU, CAQR considers that the matrix is partitioned as in Equation (1) and uses an algorithm (referred to as TSQR) based on a reduction tree to perform the panel factorization. The panel factorization is obtained by performing a sequence of QR factorizations until the lower trapezoidal part of A is annihilated and the final R factor is obtained. We present here TSQR for which the reduction tree is a binary tree:

$$\begin{array}{llll} A_1 \rightarrow R_1 & \searrow & & \\ & & R_5 & \searrow \\ A_2 \rightarrow R_2 & \nearrow & & \\ & & & R_7 \\ A_3 \rightarrow R_3 & \searrow & & \nearrow \\ & & R_6 & \\ A_4 \rightarrow R_4 & \nearrow & & \end{array}$$

At the leaves of the binary tree, the QR factorizations are performed on block rows of A. At the following levels of the binary tree, the QR factorizations are performed on previously obtained R factors. The new R factor is obtained by stacking atop one another the R factors at the left side of the arrow. The next picture presents TSQR based on a tree of height 1, that we find in our experiments can be an efficient alternative to the binary tree.

$$\begin{array}{lll} A_1 \rightarrow R_1 & \searrow & \\ A_2 \rightarrow R_2 & \rightarrow & R_5 \\ A_3 \rightarrow R_3 & \nearrow & \\ A_4 \rightarrow R_4 & \nearrow & \end{array}$$

CAQR implements the right-looking QR factorization using parallel TSQR as the panel factorization. The detailed algorithm is presented in [9]. We note a difference with CALU algorithms. In CALU the panel factorization is performed twice, and the update of the trailing matrix is similar to classic GEPP algorithms. In CAQR, the panel factorization is performed only once, but the update of the trailing matrix is triggered by the reduction tree, and it is different from classic QR factorization algorithms based on Householder transformations as in LAPACK or ScaLAPACK.

Both CALU and CAQR perform some amount of extra floating point operations compared to classic algorithms, which depends on the size of the input matrix, the block size, and the number of threads. However these computations stay of a lower order term compared to the overall cost of the LU and QR factorizations. Detailed flop counts can be found in [8], [12].

III. Multithreaded algorithms

In this section we present the multithreaded algorithms for CALU and CAQR factorizations. They are block algorithms that traverse iteratively block columns of the input matrix. Each iteration requires two steps: factorize the current panel and then use it to update the trailing matrix. The input $m \times n$ matrix A is partitioned in blocks as

$$A = \begin{pmatrix} A_{11} & A_{21} & \ldots & A_{1N} \\ A_{21} & A_{22} & \ldots & A_{2N} \\ \vdots & \vdots & & \vdots \\ A_{M1} & A_{M2} & \ldots & A_{MN} \end{pmatrix},$$

where each block A_{ij} is of size $b \times b$. The block matrix is of size $M \times N$, with $M = m/b$ and $N = n/b$. We refer to the submatrix of A formed by blocks of row indices from I to J and column indices from D to E as $A_{I:\,J,D:\,E}$.

Communication avoiding algorithms allow to perform efficiently the panel factorization. For this, at each iteration of the algorithm, the computation on the active part of the input matrix A is performed as follows. The first panel is divided into T_r blocks, where Tr represents the number of threads participating in the operation. With the same approach, every column block of the trailing matrix is divided into Tr block-rows. The computation performed on each block is associated with a task and can be performed in parallel. The partitioning of the matrix into blocks depending on b and T_r is used to separate the algorithms into multiple

tasks. We describe in the following more in detail the tasks identified in multithreaded CALU and then in multithreaded CAQR. A task dependency graph is associated with the computation of CAQR and CALU, which is then mapped on the available processors using a scheduling algorithm. The values of the two parameters T_r and b depend on the underlying architecture, they determine the granularity of the tasks, the parallelism available in the algorithm, as well as the usage of the memory hierarchy.

Multithreaded CALU: Algorithm 1 presents multithreaded CALU. Without loss of generality, we consider that m and n divide b and T_r. The keyword **task** is used to separate the algorithm into multiple tasks. For each matrix operation, the algorithm also displays the associated LAPACK routines used.

The panel factorization, performed by TSLU, is broken into several tasks. As described in section II, TSLU is formed of a preprocessing step that uses a reduction tree to find b pivot rows, which are then used to factor the entire panel. The selection of b pivot rows at each node of the reduction tree is performed by computing Gaussian elimination with partial pivoting (GEPP) of the rows selected at lower levels of the reduction tree. At the leaves, GEPP is performed on one block of the current panel. The temporary matrix B_i stores the result of the preprocessing step in TSLU.

In the initial version of CALU [12] developed for distributed memory, at every level of the reduction tree, half of the processors participating in the previous level become inactive until the final b pivot rows of the panel are computed. Then they use these final pivot rows to compute the L factor of their local blocks of the panel. This idle time is not considered important in the distributed memory environment. However in the shared memory environment we noticed that it is inefficient to leave a large number of threads inactive. Hence in the preprocessing step, the computation at each node of the reduction tree is associated with a task. This corresponds to the tasks at lines 8 and 14 in Algorithm 1. Once the final b pivot rows are computed, the computation of each block of the current block column of L is associated with a task. This is done at line 24 in Algorithm 1. With this approach, the threads that participate at the lower levels of the tree and are not involved in the upper levels can be reallocated and execute other available tasks as updating the trailing matrix.

After the panel factorization, the rows permutations need to be applied to the left and to the right parts of the panel. The permutations to the right need to be applied before continuing the factorization, while the permutations to the left can be applied at the end. In the algorithm, the permutations to the left are applied right before computing the current block row of U. Then, the block of size $b \times b$ representing the first block of the panel factorization is used to compute the upper triangular parts of the matrix and to update the trailing submatrix.

In summary, the different tasks at each iteration of the algorithm are as following (we consider the trailing matrix is of size $m_r \times n_r$):

- Task P (line 8 and 14): implements the preprocessing step in TSLU to determine b pivot rows to factor the panel. It corresponds to performing Gaussian elimination with partial pivoting (GEPP) at each node of the TSLU reduction tree.

- Task L (line 24): computes a block of the current block column of L. The block is of size $m_r/T_r \times b$.

- Task U (line 28): permute a block column of the trailing matrix and compute a block of size $b \times b$ of the current block row of U. By abuse of notation, we consider that the permutation matrices are extended by identity to the correct dimensions.

- Task S (line 36): update a block of size $m_r/T_r \times b$ of the trailing matrix.

Multithreaded CAQR: The algorithm 2 for performing multithreaded CAQR is very similar to multithreaded CALU. The main difference is that for CAQR, the panel factorization is performed only once as a reduction operation. The Householder transformations performed during the reduction need to be applied on the trailing matrix as well. The reduction tree used for panel factorization is also used for updating the trailing matrix.

At each iteration of the algorithm, the panel is partitioned into T_r block rows A_i. The QR factorization of the panel using a binary tree requires $log(T_r)$ steps. At the first step, the QR factorization of each block row $A_i = Q_i R_i$ is computed. Then, from the second to the last step, the R_i factors previously computed are stacked atop one another, and their QR factorization is performed. Updates of the trailing matrix are also performed in two steps. First, the trailing matrices corresponding to QR factorization of A_i are updated. Second, the trailings blocks corresponding to R_i computed during the tree are updated.

In summary, the different tasks at each iteration of the algorithm are as following (we consider the trailing matrix is of size $m_r \times n_r$):

- Task P (line 8 and 19): implements the step in TSQR to compute Q and R factor of block panel

- Task S (line 11): update a block of size $m_r/T_r \times b$ of the trailing matrix corresponding to the QR factorization of the leaves of the binary tree.

- Task S (line 26): update a block of size $b \times b$ of the trailing matrix corresponding to the QR factorization of two R stacked atop one another at the the upper levels of the binary tree.

Task scheduling: The tasks identified in multithreaded CAQR and CALU are dynamically scheduled to the available cores. The task dependency graph used by the scheduler is constructed on the fly, since its size can be important. At each iteration of the algorithms, there are on the order of $Tr \times n/b$ tasks. By partitioning the matrix into blocks of

Algorithm 1 Multithreaded CALU

Input: $m \times n$ matrix A, block size b, number of panel tasks T_r
$M = m/b, N = n/b$
for K = 1 to $N = n/b$ **do**
  /* TSLU preprocessing step*/
  for I = 1 to Tr **do in parallel**
    $I_1 = (K-1) + (I-1)\lceil (M-K+1)/T_r \rceil$
    $I_2 = MIN(M, K-1+I\lceil (M-K+1)/T_r \rceil)$
    task P $[\Pi_I, L_I, U_I]$ = $GEPP(A_{I_1:I_2,K})$ /* rgetf2 */
    Let B_I be formed by the pivot rows, $B_I = (\Pi_I A_{I_1:I_2,K})(1:b,:)$
  end for
  for $level$ = 1 to $log_2(Tr)$ **do**
    for I = 1 to Tr **do in parallel**
      if $((I-1) \mod 2^{level-1} == 0)$ **then**
        task P $[\Pi_I, L_I, U_I]$ = GEPP $([B_I; B_{I+2^{level-1}}])$
        Let B_I be formed by the pivot rows, $B_I = (\Pi_I[B_I; B_{I+2^{level-1}}])(1:b,:)$
      end if
    end for
  end for
  Let $L_{KK} = L_K(1:b,:), U_{KK} = U_K(1:b,:)$, Π_{KK} is the permutation performed for this panel
  /* Compute block column of L */
  for I = 1 to Tr **do in parallel**
    $I_1 = (K-1) + (I-1)\lceil (M-K+1)/T_r \rceil$
    $I_2 = MIN(M, K-1+I\lceil (M-K+1)/T_r \rceil)$
    task L $L_{I_1:I_2,K} = A_{I_1:I_2,K} U_{KK}^{-1}$ /* dtrsm */
  end for
  /* Apply permutation to the right and compute block row of U */
  for J = $K+1$ to N **do in parallel**
    task U $A_{K:M,J} = \Pi_{KK} A_{K:M,J}$
    $U_{K,J} = L_{KK}^{-1} A_{K,J}$ /* dtrsm */
  end for
  /* Update the trailing submatrix */
  for J = $K+1$ to N **do in parallel**
    for I = 1 to Tr **do in parallel**
      $I_1 = (K-1) + (I-1)\lceil (M-K+1)/T_r \rceil$
      $I_2 = MIN(M, K-1+I\lceil (M-K+1)/T_r \rceil)$
      task S $A_{I_1:I_2,J}$ = $A_{I_1:I_2,J} - L_{I_1:I_2,K} U_{K,J}$ /* dgemm */
    end for
  end for
end for
/* Apply permutations to the left */
$L_{1:M,1:N} = \Pi_{NN} \ldots \Pi_{11} L_{1:M,1:N}$ /* dlaswap */

Algorithm 2 Multithreaded CAQR

Input: $m \times n$ matrix A, block size b, number of panel tasks T_r
$M = m/b, N = n/b$
for K = 1 to $N = n/b$ **do**
  for I = 1 to Tr **do in parallel**
    $I_1 = (K-1) + (I-1)\lceil (M-K+1)/T_r \rceil$
    $I_2 = MIN(M, K-1+I\lceil (M-K+1)/T_r \rceil)$
    /* Apply QR on the first block*/
    task P $[V_I, T_I, R_I] = QR(A_{I_1:I_2,K})$ /* dgeqr3 */
    /* Update the trailing submatrix */
    for J = $K+1$ to N **do in parallel**
      task S $A_{I_1:I_2,J}$ = $(I - V_I T_I V_I^T) A_{I_1:I_2,J}$ /* dlarfb */
    end for
  end for
  /* Compute R using TSQR tree*/
  for $level$ = 1 to $log_2(Tr)$ **do**
    for I = 1 to Tr **do in parallel**
      if $((I-1) \mod 2^{level-1} == 0)$
        $target = I + 2^{level-1}$
        task P $[V_I, T_I, R_I] = QR([R_I; R_{target}])$
        $I_1 = K-1+(I-1)\lceil (M-K+1)/T_r \rceil$
        $I_2 =$ $MIN(M, K-1+I\lceil (M-K+1)/T_r \rceil)$
        $I_3 = K-1+(target-1)\lceil (M-K+1)/T_r \rceil$
        $I_4 =$ $MIN(M, K-1+target\lceil (M-K+1)/T_r \rceil)$
        /* Update the trailing submatrices*/
        for J = $K+1$ to N **do in parallel**
          task S $[A_{I_1:I_2,J}; A_{I_3:I_4,J}]$ = $(I - V_I T_I V_I^T)[A_{I_1:I_2,J}; A_{I_3:I_4,J}]$ /* dlarfb */
        end for
      end if
    end for
  end for
end for

smaller size increases the parallelism available. However for a too large number of tasks, the time spent in the scheduling can become significant and can lead to a loss of performance. We will discuss in the experiments section IV choices of T_r and b and their influence on the performance.

Figure 1 displays the task dependency graph of CALU for a matrix partitioned into 4×4 blocks. Figure 2 shows an example of task scheduling corresponding to this task dependency graph.

In our approach the panel factorization lies on the critical path. However, due to the efficient CALU algorithm, its time is significantly reduced compared to classic approaches, and it is often hidden by the tasks performing updates on the trailing matrix. In addition, the scheduler uses a so called look-ahead of 1 technique. It consists in giving a

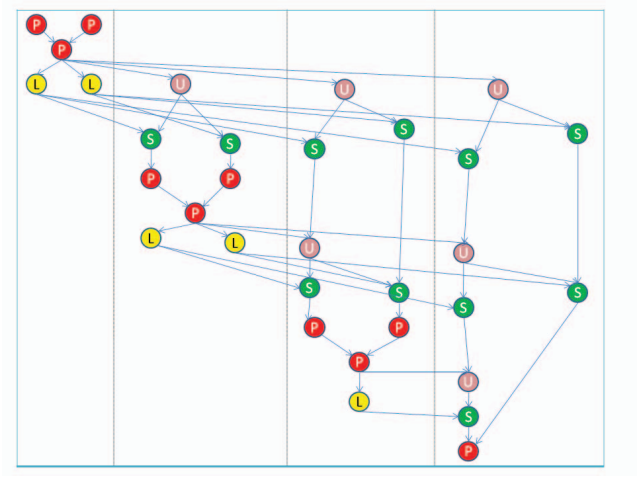

Figure 1. Task dependency graph for a matrix partitioned into 4×4 blocks. Tasks are presented as circles. P denotes the preprocessing step to factor a panel, L represents the computation of a block column of the L factor, U represents the computation of a block row of U, and S represents the update of the trailing matrix.

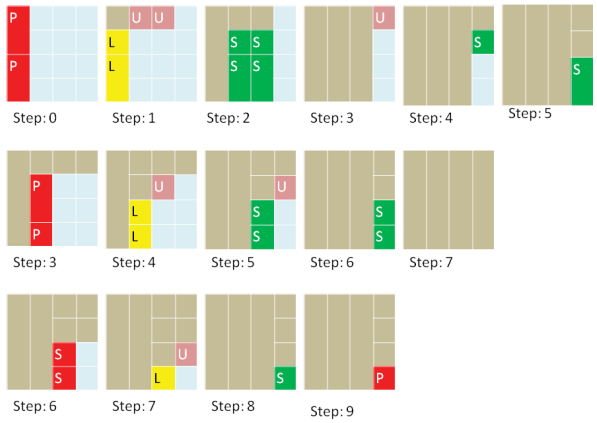

Figure 2. Example of execution of the task dependency graph displayed in Figure 1 using 4 threads. The input matrix is partitioned into 4×4 blocks and $T_r = 2$. Each step describes the tasks executed concurrently.

high priority to the tasks that are on the critical path. That is, at step K of factorization, after factoring the panel, the update of block column $K + 1$ has the highest priority and is scheduled next. Then the factorization of panel $K+1$ will be performed.

For large matrices (relative to the number of cores), the time to perform the panel factorization is often hidden by the time to update the trailing matrix, which dominates the overall execution. However, for small matrices or for tall and skinny matrices, the computation performed in updating the trailing matrix is not sufficient to keep the cores busy, and hence the panel factorization can create important idle times in the execution. This is illustrated in 3 where CALU is executed on a tall and skinny matrix of size $10^5 \times 1000$, using $b = 100$ and $T_r = 1$, on a 8-core machine. In classic factorizations, the panel factorization is parallelized, but not very efficiently (as it will be shown in the experimental section). Hence the idle times due to the panel factorization can still be observed. Figure 4 displays the execution of CALU on the same matrix and the same machine, by using $T_r = 8$. That is, the panel factorization is executed as fast as possible using all the available cores. We note that except the very beginning and the very end of the algorithm, there is no idle time and all the cores are kept busy. Hence a more efficient algorithm is obtained, as it will be shown in the experiments section IV.

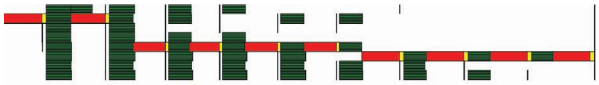

Figure 3. Example of execution of CALU for a $10^5 \times 1000$ tall skinny matrix, using $b = 100$ and $T_r = 1$, on the 8-core Intel machine. The red bar represents the panel factorization, the yellow bar corresponds to the computation of L, and the green bar is the update of the trailing matrix. The time to compute U is very small, and hence it is not displayed.

Figure 4. Example of execution of CALU for a $10^5 \times 1000$ tall skinny matrix, using $b = 100$ and $T_r = 8$, on the 8-core Intel machine. The red bar represents the panel factorization, the yellow bar corresponds to the computation of L, and the green bar is the update of the trailing matrix. The time to compute U is very small, and hence it is not displayed.

IV. Experimental results

In this section we evaluate the performance of multi-threaded CALU and CAQR on a two-socket, quad-core machine based on Intel Xeon EMT64 processor and on a four-socket, quad-core machine based on AMD Opteron processor running on Linux. Each core has a frequency of 2.50GHz for the Intel processor and 2.194GHz for the AMD processor. We compare the performance of our algorithms with the corresponding routines from MKL 10.0.4.23 [15] vendor library, ACML vendor library, and PLASMA 2.0 [1].

For the LU factorization of general rectangular matrices, we refer to the corresponding routines from MKL, ACML, and PLASMA as MKL_dgetrf, ACML_dgetrf and PLASMA_dgetrf. In our experiments we focus in particular on tall and skinny matrices. Hence we also compare the performance of TSLU and CALU with the BLAS2 routines from MKL (that we refer to as MKL_dgetf2) that is used to perform the factorization of block of columns in dgetrf. The corresponding routines for computing the QR factorization of a matrix are dgeqrf for general rectangular matrices. We

refer to these routines as MKL_dgeqrf, ACML_dgeqrf for the vendor libraries and PLASMA_dgeqrf for PLASMA. MKL_dgeqr2 is the BLAS2 algorithm for computing the QR factorization of a block of columns in dgeqrf. We use in our experiments Intel BLAS from MKL-10.0.4.23.

The performance of CALU and CAQR depends on several parameters. As described in section III, CALU and CAQR iterate over panels, where the size of a panel is given by the parameter b. At each iteration, the matrix is considered to be partitioned using a two-dimensional block layout into $T_r \times n_r/b$ blocks, where n_r is the number of columns in the trailing matrix. The number of tasks available at each step of our algorithms is given by this two-dimensional partitioning of the matrix. That is, T_r threads can contribute to the panel factorization by using multithreaded TSLU or TSQR algorithm, while as many as $T_r \times n_r/b$ threads can perform updates on the trailing submatrix.

The optimal choice of parameters b and T_r depends on the size of the input matrix and on the architecture of the multicore machine. These parameters determine the parallelism available in the algorithm as well as the granularity of matrix operations. In general there is a trade-off between the number of available tasks and their granularity. There should be enough tasks to keep the cores busy. However, the tasks, in particular those performing updates on the trailing matrix, should not operate on too small matrices, otherwise there is a loss of performance in the use of BLAS3 routines.

As noted in [9], [12], TSLU and TSQR outperform ScaLAPACK on a distributed memory computer due to two main reasons. First, the number of messages exchanged is minimized. Second, the algorithms compute most of their floating-point operations using the most efficient sequential algorithm available. In our experiments the best results are obtained by using recursive LU [13], [17] and QR [10] algorithms, that we refer to as rgetf2 and dgeqr3 respectively.

Performance of multithreaded CALU: We focus first on the performance of CALU for tall and skinny matrices. On the 8-core Intel machine, the best results were often obtained for a block of size around $b = 100$. Hence, for matrices of size $m \times n$, we use a block size of $b = \min(100, n)$. We vary the number of tasks for performing the panel factorization from 1 to 8.

Figures 5 and 6 display the performance results of CALU relative to MKL_dgetf2, MKL_dgetrf and PLASMA_dgetrf routines on the 8-core Intel machine. The tall and skinny matrices considered are of size $m \times n$, where $m = 10^5$ (in figure 5) and $m = 10^6$ (in figure 6) and n varies as 10, 25, 50, 100, 150, 200, 500, 1000.

The best performance is obtained for CALU with $T_r = 8$. In general CALU is between 1.5 and 2 times faster than MKL_dgetrf, with the best speedup of 2.3 obtained for matrices of size $10^6 \times 500$.

Important speedups are obtained for CALU compared to MKL_dgetf2. For example for matrices of size $10^6 \times 100$, speedups of 10 and 8.3 are obtained for CALU with $T_r = 8$ and $T_r = 4$ respectively. However even for matrices with a smaller number of columns, as for example $n = 25$, CALU leads to an important speedup of 4 with respect to MKL_dgetf2, and a speedup of 2 with respect to MKL_dgetrf.

These results show that MKL_dgetf2 has poor performance. However, MKL_dgetrf leads to better performance. It is possible that MKL_dgetrf uses a very small block size for the panel factorization, and it performs very efficiently the other operations as updates on the trailing matrix. However for tall and skinny matrices, or for small square matrices, the panel factorization can lead to an important loss of performance, and hence it is important to compute it as efficiently as possible. Our results suggest that the usage of a fast panel factorization using TSLU, combined with the optimized MKL_dgetrf for the other operations can lead to an efficient algorithm.

For matrices for which the number of columns is $n \leq 300$, CALU outperforms PLASMA. The best speedup of 9.4 is obtained for a $10^5 \times 10$ matrix, while a speedup of 3.2 is observed for a $10^5 \times 200$ matrix. Although we observe an almost constant speedup with respect to MKL, we note a decrease of the speedup with respect to PLASMA when n increases. For example, when $m = 10^5$, the speedup decreases from 3.2 (for $n = 200$) to 1.6 (for $n = 500$) and to 1.1 (for $n = 1000$). This shows that PLASMA becomes more efficient when the number of columns increases. In figure 6 it becomes more efficient than CALU when $n = 1000$.

Figure 7 displays the performance of CALU, MKL_dgetrf, and PLASMA_dgetrf routines on the 16-cores AMD machine. The matrix considered in the tests has $m = 10^5$ and n varies from 10 to 1000. CALU($Tr = 16$) is on average 5 times faster than the corresponding routine in ACML library and on average 1.5 times faster than PLASMA.

Table I presents performance results for square matrices for which $m = n$ on the 8-core Intel machine. CALU is slower than MKL_dgetrf when $m = n < 5000$, while for $m = n = 10^4$ CALU ($T_r = 2$) is slightly faster than MKL_dgetrf. A speedup of 1.5 is obtained for matrices of size 5000×5000 with respect to MKL_dgetrf. For $n > 3000$, CALU is faster than PLASMA_dgetrf, for which we have used the default parameters.

We conclude that CALU reduces the time to perform the panel factorization. However its performance is affected by a slower update of the trailing matrix, which can be highly optimized in MKL_dgetrf. Furthermore, our algorithm uses dynamic scheduling, and hence the time spent in the scheduling itself can lead to a loss of performance, in particular when the number of tasks is very large. These results suggest that by combining a fast panel factorization as in CALU with a highly optimized update of the trailing matrix as in MKL_dgetrf can lead to a more efficient algorithm for

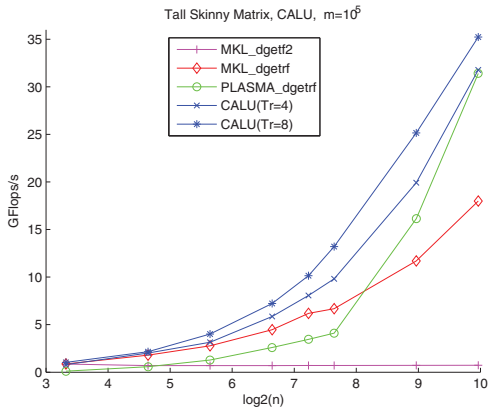

Figure 5. Performance of CALU, MKL_dgetrf, PLASMA_dgetrf on 8-core Intel machine for matrices with m=10^5 and varying n from 10 to 1000. For CALU, the block size is $b = MIN(n, 100)$ and T_r is varied as $4, 8$.

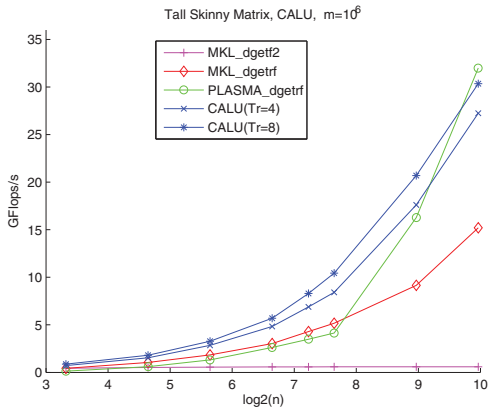

Figure 6. Performance of CALU, MKL_dgetrf, PLASMA_dgetrf on 8-core Intel machine for matrices with m=10^6 and varying n from 10 to 1000. For CALU, the block size is $b = MIN(n, 100)$ and T_r is varied as $4, 8$.

Table I
PERFORMANCE OF CALU, MKL_DGETRF, AND PLASMA_DGETRF (IN GFLOPS/S) FOR SQUARE MATRICES ON 8-CORE INTEL MACHINE. THE PARAMETERS OF CALU ARE AS FOLLOWING: THE BLOCK SIZE $b = 100$ AND T_r VARIES AS $1, 2, 4, 8$.

m=n	MKL dgetrf	PLASMA dgetrf	CALU			
			Tr=1	Tr=2	Tr=4	Tr=8
1000	**38.4**	17.8	15.7	15.5	15.1	13.6
2000	**45.3**	32.6	26.5	31.2	32.9	30.3
3000	**48.8**	38.8	33.7	43.2	43.6	40.7
4000	**53.1**	42.5	38.9	50.5	49.9	47.5
5000	**55.6**	42.3	42.1	54.2	54.1	51.7
10000	61.39	48.3	52.3	**63.5**	62.7	61.4

square matrices.

Table II displays the performance of CALU, ACML_dgetrf, and PLASMA_dgetrf on the 16-core AMD machine for square matrices. For $m = n \leq 2000$,

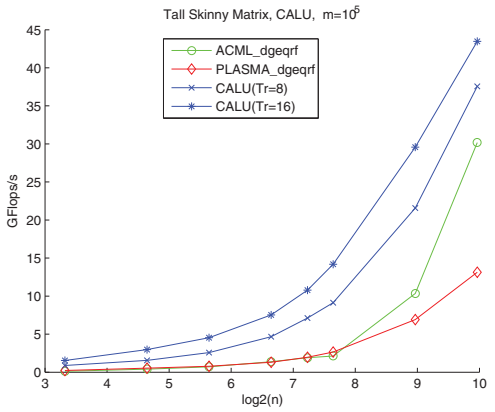

Figure 7. Performance of CALU, ACML_dgetrf, PLASMA_dgetrf on 16-core AMD machine for matrices with m=10^5 and varying n from 10 to 1000. For CALU, the block size is $b = MIN(n, 100)$ and T_r is varied as $8, 16$.

Table II
PERFORMANCE OF CALU, ACML_DGETRF, AND PLASMA_DGETRF (IN GFLOPS/S) FOR SQUARE MATRICES ON 16-CORE AMD MACHINE. THE PARAMETERS OF CALU ARE AS FOLLOWING: THE BLOCK SIZE $b = 100$ AND T_r VARIES AS $1, 2, 4, 8, 16$.

m=n	ACML dgetrf	PLASMA dgetrf	CALU				
			Tr=1	Tr=2	Tr=4	Tr=8	Tr=16
1000	**16.2**	10.0	10.8	10.4	10.2	11.5	11.8
2000	**29.6**	25.9	21.3	22.6	28.3	26.8	22.1
3000	31.0	32.2	27.8	30.5	**34.4**	34.3	28.9
4000	26.3	35.2	34.5	36.4	**37.9**	37.8	34.1
5000	26.8	38.0	38.6	39.5	**39.7**	39.2	38.9

ACML_dgetrf is faster than CALU, while for $m = n \geq 3000$ CALU outperforms ACML_dgetrf. For all the sizes, CALU outperforms PLASMA_dgetrf.

Performance of multithreaded CAQR: We now evaluate the performance of multithreaded CAQR, and compare it to MKL_dgeqrf and PLASMA_dgeqrf routines. The results presented here are for CAQR that uses a tree of height one. We also present results for TSQR, where a binary tree is used. We first present the performance results for tall and skinny matrices. Figure 8 displays performance results for CAQR on the 8-core Intel machine. The matrices are of size $m \times n$, where $m = 10^5$ and n varies as 10, 25, 50, 100, 150, 200, 500 and 1000. For CAQR we present the results obtained for $T_r = 4$ and $b = 100$, which were the best over other values of T_r and b.

We note that TSQR outperforms all the other codes for tall and skinny matrices. The best improvement is obtained for a $10^5 \times 200$ matrix, where TSQR is 5.3 times faster than MKL_dgeqrf and 3.6 times faster than PLASMA_dgeqrf. Even for a larger $10^5 \times 500$ matrix, TSQR is 3.4 times faster than MKL_dgeqrf and 1.2 times faster than PLASMA_dgeqrf. The best speedup with respect to PLASMA_dgeqrf is 6.7, obtained for a $10^5 \times 10$ matrix.

With increasing n, TSQR outperforms MKL_dgeqrf, but the improvement is less important and decreases gradually with respect to PLASMA_dgeqrf. PLASMA_dgeqrf becomes faster than TSQR for $n = 1000$.

TSQR is a very simple algorithm, and hence it is very interesting to see that it outperforms other more complicated algorithms for tall and skinny matrices. There are two reasons for this. First, TSQR uses less synchronization than the other algorithms. Second, it performs most of its computation by using the efficient recursive QR factorization [10]. However with increasing number of columns, TSQR performs more redundant computations, and hence it becomes less efficient.

CAQR is slower than TSQR, but can be faster than MKL_dgeqrf. For example for $n = 500$ and $n = 1000$ CAQR ($Tr = 4$) is approximately 1.6 times faster than MKL_dgeqrf and 20 times faster than MKL_dgeqr2. PLASMA_dgeqrf seems better optimized when the matrix becomes larger.

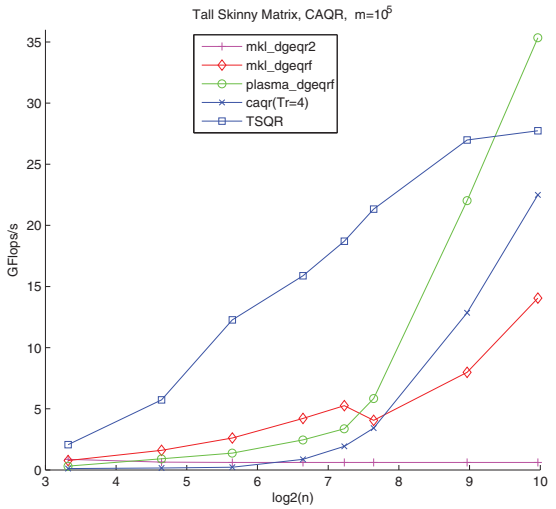

Figure 8. Performance of CAQR, MKL_dgeqrf, PLASMA_dgeqrf on 8-core Intel machine for matrices with m=10^5 and varying n from 100 to 1000. For CAQR, the block size is $b = MIN(n, 100)$ and $T_r = 4$.

Table III presents performance results for square matrices on the 8-core Intel machine. MKL is more efficient than PLASMA, which is at its turn more efficient than CAQR.

We observed the same behaviour on the 16-core AMD machine, when comparing with ACML vendor library.

We note that for PLASMA we present only the time spent in the dgetrf and dgeqrf factorization routines. We do not present the time spent in the initialization routine of the library that has to be called before any other routine. For all the other routines, as CAQR, CALU, MKL or ACML, there is no initialization routine.

Table III
PERFORMANCE OF CAQR, MKL_DGEQRF, AND PLASMA_DGEQRF (IN GFLOPS/S) FOR SQUARE MATRICES. THE PARAMETERS OF CAQR ARE AS FOLLOWING: THE BLOCK SIZE $b = 100$, T_r VARIES AS $1, 2, 4, 8$.

m=n	MKL dgeqrf	PLASMA dgeqrf	CAQR Tr=1	CAQR Tr=2	CAQR Tr=4	CAQR Tr=8
1000	**41.0**	27.3	4.3	11.8	22.6	17.6
2000	**52.1**	41.3	26.2	33.3	37.5	37.5
3000	**50.3**	46.5	22.1	40.2	43.1	40.9
4000	**49.4**	48.4	38.1	45.0	46.0	44.8
5000	**54.5**	49.5	40.9	46 .7	47.7	46.7

V. CONCLUSION

In this paper we have introduced multithreaded CALU and CAQR, two algorithms based on CALU and CAQR, initially developed for distributed memory machines. Our results for tall and skinny matrices show that CALU and TSQR (that is CAQR computed as one panel factorization) are in general 2 to 3 times faster than the corresponding routines in MKL and PLASMA. Furthermore, when the number of columns is very small, for example 10, CALU and TSQR lead to a speedup of 2 to 5 compared to corresponding routines in MKL and PLASMA. We note that a very recent paper [14] describes an algorithm that combines tiled algorithms with CAQR. It corresponds to CAQR that uses a reduction tree formed by a flat tree at the lower levels, followed by a binary tree at the upper levels. The authors report speedups for tall and skinny matrices on a 16-core Intel machine. Further experiments are necessary to compare with this new approach.

For square matrices, the improvements are not very important. On the 16-core AMD machine, CALU starts to outperform ACML with increasing matrix size. CAQR is less efficient than PLASMA. These results suggest in particular that CALU is an interesting approach for multicore architectures.

Our experiments show that the panel factorization is efficiently performed by TSLU and TSQR, while the update of the trailing matrix can be further optimized. As future work we plan to investigate several directions to improve the update of the trailing matrix. In particular we plan to introduce a parameter $B > b$ such that the update of the trailing matrix is performed on blocks of a larger size. The algorithms will use b as panel block size and B as trailing submatrix block size. Hence, we obtain a partitioning of the matrix in two parts of size $Tr \times b$ and $Tr \times (N - b)/B$. B being chosen independently of the panel, we can optimize trailing submatrix updating time by reducing the number of tasks and by better exploiting BLAS3. This will decrease the number of tasks to schedule, and hence as well the time spent in the dynamic scheduling.

REFERENCES

[1] Plasma. http://icl.cs.utk.edu/plasma/.

[2] 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.

[3] M. Baboulin, J. Dongarra, and S. Tomov. Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures. Technical Report UT-CS-08-615, University of Tennessee, 2008. LAPACK Working Note 200.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D. and̃ J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, May 1997.

[5] 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.

[6] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR and Cholesky Factorization Routines. *Scientific Programming*, 5(3):173–184, 1996. ISSN 1058-9244.

[7] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-avoiding parallel and sequential QR and LU factorizations. Technical Report UCB/EECS-2008-89, University of California Berkeley, EECS Department, 2008. LAWN #204.

[8] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report short version of UCB/EECS-2008-89, arxiv, 2008.

[9] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Implementing communication optimal parallel and sequential QR factorizations. Technical Report short version of UCB/EECS-2008-89, arxiv, 2008.

[10] E. Elmroth and F. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In B. K. et al., editor, *Applied Parallel Computing. Large Scale Scientific and Industrial Problems.*, volume 1541 of *Lecture Notes in Computer Science*, pages 120–128. Springer, 1998.

[11] G. H. Golub, R. J. Plemmons, and A. Sameh. Parallel block schemes for large-scale least-squares computations. In R. B. Wilhelmson, editor, *High-Speed Computing: Scientific Applications and Algorithm Design*, pages 171–179. University of Illinois Press, Urbana and Chicago, IL, USA, 1988.

[12] L. Grigori, J. W. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. *Proceedings of the ACM/IEEE SC08 Conference*, 2008.

[13] F. Gustavson. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.

[14] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tall and Skinny QR Matrix Factorization Using Tile Algorithms on Multicore Architectures. Technical report, University of Tennessee, 2009. LAPACK Working Note 200.

[15] Intel. Math kernel library (mkl). http://www.intel.com/software/products/mkl/.

[16] A. Pothen and P. Raghavan. Distributed orthogonal factorization: Givens and Householder algorithms. *SIAM J. Sci. Stat. Comput.*, 10(6):1113–1134, Nov. 1989.

[17] S. Toledo. Locality of reference in LU Decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4), 1997.

[18] L. N. Trefethen and R. S. Schreiber. Average-case stability of Gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 11(3):335–360, 1990.