Avoiding communication through a multilevel LU factorization

Donfack, Simplice and Grigori, Laura and Khabou, Amal

2012

MIMS EPrint: **2013.13**

ISSN 1749-9097

Avoiding communication through a multilevel LU factorization

Simplice Donfack (simplice.donfack@lri.fr), Laura Grigori (laura.grigori@inria.fr), and Amal Khabou (amal.khabou@inria.fr).

INRIA Saclay-Ile de France, Laboratoire de Recherche en Informatique, Université Paris-Sud

Abstract. Due to the evolution of massively parallel computers towards deeper levels of parallelism and memory hierarchy, and due to the exponentially increasing ratio of the time required to transfer data, either through the memory hierarchy or between different compute units, to the time required to compute floating point operations, the algorithms are confronted with two challenges. They need not only to be able to exploit multiple levels of parallelism, but also to reduce the communication between the compute units at each level of the hierarchy of parallelism and between the different levels of the memory hierarchy.
In this paper we present an algorithm for performing the LU factorization of dense matrices that is suitable for computer systems with two levels of parallelism. This algorithm is able to minimize both the volume of communication and the number of messages transferred at every level of the two-level hierarchy of parallelism. We present its implementation for a cluster of multicore processors based on MPI and Pthreads. We show that this implementation leads to a better performance than routines implementing the LU factorization in well-known numerical libraries. For matrices that are tall and skinny, that is they have many more rows than columns, our algorithm outperforms the corresponding algorithm from ScaLAPACK by a factor of 4.5 on a cluster of 32 nodes, each node having two quad-core Intel Xeon EMT64 processors.

Keywords: LU factorization, communication avoiding algorithms, multiple levels of parallelism

1 Introduction

Due to the evolution of massively parallel computers towards deeper levels of parallelism and memory hierarchy, and due to an exponentially increasing ratio of the time necessary to transfer data, either through the memory hierarchy or between different compute units, to the time required to perform floating point operations, the algorithms are confronted with two challenges. They need to be able to exploit multiple levels of parallelism, and they also need to minimize communication and synchronization at each level of the hierarchy of parallelism and memory. The particularity of a computer system with multiple levels of parallelism is that a compute unit at a given level can be formed by several smaller compute units connected together. A machine formed by nodes of multicore processors can be seen as an example of a machine with two levels of parallelism. An approach to exploit such an architecture for an existing algorithm consists of identifying functions which are executed sequentially on a node, and then replacing them by a call to their multithreaded version. This approach, based mainly on combining MPI and threads, is easy to implement and has been used in many applications, but can have several drawbacks. It can lead to more communication and synchronization between MPI processes or between threads, load imbalance, and in general a simple adaptation of an existing algorithm can cause an important degradation of the overall performance on this type of architectures. For such a computer system, there are two levels of communication: inter-node communication, that is communication between two or more nodes, and intra-node communication, that is communication performed inside each node. For both types, an algorithm that takes into account at the design level the two levels of parallelism can be able to reduce communication at every level.

Motivated by the increased cost of communication with respect to the cost of computation [11], a new class of algorithms has been introduced in the recent years for dense linear algebra, referred to as communication avoiding algorithms. These algorithms, first proposed for dense LU factorization (CALU) [12] and QR factorization (CAQR) [6], allow to minimize communication on a computer system with one level of parallelism (or between two levels of fast and slow memory) and are as stable as classic algorithms as for example implemented in LAPACK [2] and ScaLAPACK [3]. They were shown to lead to good performance on distributed memory machines [6, 12], on multicore processors [7], and on grids [1]. In the distributed version of CALU and CAQR, blocks of the input matrix are distributed among processors, and data is communicated via MPI messages during the factorization. In the approach used for multicore processors [7], operations on a block are performed as tasks, which are scheduled statically or dynamically to the available cores. However, none of these algorithms has addressed the more realistic model of today's hierarchical parallel computers.

In this paper we introduce an algorithm for performing the LU factorization of a dense matrix that is suitable for computer systems with two levels of parallelism, and that can be further extended to multiple levels of parallelism. We refer to this algorithm as multilevel CALU. It can be seen as a generalization of CALU [13, 12]. At each iteration of the initial 1-level CALU algorithm, a block column, referred to as a panel, is factored and then the trailing matrix is updated. A classic algorithm as Gaussian elimination with partial pivoting (GEPP) is not able to minimize the number of messages exchanged during the factorization. This is because of partial pivoting, which requires to permute the element of maximum magnitude to the diagonal position at each step of the panel factorization. To minimize communication, CALU uses tournament pivoting, a different strategy shown to be very stable in practice [12]. With this strategy, the panel factorization is performed in two steps. In the first step, tournament pivoting uses a reduction operation to select a set of pivot rows from different blocks of the panel distributed among different processors or different cores, where GEPP is the operator used at each node of the reduction tree. In the second step, these pivot rows are permuted to the top of the panel, and then the LU factorization without pivoting of the panel is performed.

Multilevel CALU uses the same approach as CALU, and it is based on tournament pivoting and an optimal distribution of the input matrix among compute units to reduce communication at the first level of the hierarchy. However, each building block of CALU is itself a recursive function that allows to be optimal at the next level of the hierarchy of parallelism. For the panel factorization, at each node of the reduction tree of tournament pivoting, CALU is used instead of GEPP to select pivot rows, based on an optimal layout adapted to the current level of parallelism. We present this algorithm in section 2. We also model the performance of our approach by computing the number of floating-point operations, the volume of communication, and the number of messages exchanged on a computer system with two levels of parallelism. We show that our approach is optimal at every level of the hierarchy and attains the lower bounds on communication of the LU factorization (modulo polylogarithmic factors). The lower bounds on communication for the multiplication of two dense matrices were introduced in [14, 15] and were shown to apply to LU factorization in [6]. We discuss how these bounds can be used in the case of two levels of parallelism. Due to the multiple calls to CALU, multilevel CALU performs additional flops compared to 1-level CALU. It is known in the literature that in some cases, these extra flops can degrade the performance of a recursive algorithm (see for example in [8]). However, for two levels of parallelism, the choice of the optimal layout at each level of the hierarchy allows to keep the extra flops as a lower order term. Furthermore, multilevel CALU may also change the stability of the 1-level algorithm. We argue in section 3 through numerical experiments that 2-level CALU specifically studied here is stable in practice. We also show that multilevel CALU is up to 4.5 times faster than the corresponding routine PDGETRF from ScaLAPACK tested in multithreaded mode on a cluster of multicore processors.

2 CALU for multiple levels of parallelism

In this section we introduce a multilevel communication avoiding LU factorization, presented in Algorithm 1, that is suitable for a hierarchical computer system with L levels of parallelism. Each compute unit at a given level i is formed by P_{i+1} compute units of level $i+1$. Correspondingly, the memory associated with a compute unit at level i is formed by the sum of the memories associated with the P_{i+1} compute units of level $i+1$. Level 1 is the first level and level L is the last level of the hierarchy of parallelism. Later in this section we model the communication cost of the algorithm for a computer system with two levels of parallelism.

The goal of multilevel CALU is to minimize communication at every level of a hierarchical system. It is based on a recursive approach, where at every level of the recursion optimal parameters are chosen, such as optimal layout and distribution of the matrix over compute units, optimal reduction tree for tournament pivoting. Algorithm 1 receives as input the matrix A of size $m \times n$, the number of levels of parallelism in the hierarchy L, and the number of compute units P_1 that will be used at the first level of the hierarchy and that are organized as a two-dimensional grid of compute units of size $P_1 = P_{r_1} \times P_{c_1}$. The input matrix A is partitioned into blocks of size $b_1 \times b_1$,

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & & \vdots \\ A_{M1} & A_{M2} & \dots & A_{MN} \end{pmatrix},$$

where $M = m/b_1$ and $N = n/b_1$. The block size b_1 and the dimension of the grid $P_{r_1} \times P_{c_1}$ are chosen such that the communication at the top level of the hierarchy is minimized, by following the same approach as for the 1-level CALU algorithm [12]. That is, the blocks of the matrix are distributed among the P_1 compute units using a two-dimensional block cyclic distribution over the two-dimensional grid of compute units $P_1 = P_{r_1} \times P_{c_1}$ (we will discuss later in this section the values of P_{r_1} and P_{c_1}) and tournament pivoting uses a binary reduction tree. At each step of the factorization, a block of b_1 columns (panel) of L is factored, a block of b_1 rows of U is computed, and then the trailing submatrix is updated. Each of these steps is performed by calling recursively functions that will be able to minimize communication at the next levels of the hierarchy of parallelism. Note that Algorithms 1 and 2 do not detail the communication performed during the factorization, which is triggered by the distribution of the data. By abuse of notation, the permutation matrices need to be considered as extended by identity matrices to the desired dimensions. For simplicity, we also consider that the number of processors are powers of 2.

We describe in more detail now the panel factorization (line 8 of Algorithm 1) computed by using mTSLU, described in Algorithm 2. It is a multilevel version of TSLU, the panel factorization used in the 1-level CALU algorithm [13]. Let B denote the first panel of size $m \times b_1$, which is partitioned into P_{r_i} blocks. As in TSLU, mTSLU is performed in two steps. In the first step, a set of b_1 pivot rows are selected by using tournament pivoting. In the second step, these rows are permuted into the diagonal positions of the panel, and then the LU factorization with no pivoting of the panel is computed. Tournament pivoting uses a reduction operation, where at the leaves of the tree b_1 candidate pivot rows are selected from each block B_I of B. Then a tournament is performed among the P_{r_i} sets of candidate pivot rows to select the final pivot rows that will be used for the factorization of the panel. At each node of the reduction tree, a new set of candidate pivot rows is selected from the sets of candidate pivot rows of the children nodes in the reduction tree. The initial 1-level TSLU uses GEPP to select a set of candidate pivot rows. However this means that at the second level of parallelism, the compute units involved in one GEPP factorization will need to exchange $O(b_1)$ messages for each call to GEPP due to partial pivoting, and hence the number of messages will not be minimized at the second level of the hierarchy of parallelism. Differently from 1-level TSLU, multilevel TSLU selects a set of rows by calling multilevel CALU, hence being able to minimize communication at the next levels of parallelism. That is, at each phase of the reduction operation every compute unit from the first level calls multilevel CALU on its blocks with adapted parameters and data layout. At the last level of the recursion, 1-level CALU is called (referred to in the algorithms as CALU).

Once the panel factorization is performed, the trailing submatrix is updated using a multilevel solve for a triangular system of equations (referred to as dtrsm) and a multilevel algorithm for multiplying two matrices (referred to as dgemm). We do not detail here these algorithms, but one should use recursive versions of Cannon [4], or SUMMA [10], or a cache oblivious approach [9] if the transfer of data across different levels of the memory hierarchy is to be minimized as well (with appropriate data storages).

Algorithm 1 mCALU: multilevel communication avoiding LU factorization

```
1: **Input:** $m \times n$ matrix $A$, level of parallelism $i$ in the hierarchy, block size $b_i$, number
   of compute units $P_i = P_{r_i} \times P_{c_i}$
2: **if** $i == L$ **then**
3:     $[\Pi_i, L_i, U_i] = CALU(A, b_i, P_i)$
4: **else**
5:     $M = m/b_i, N = n/b_i$
6:     **for** $K = 1$ to $N$ **do**
7:         $[\Pi_{KK}, L_{K:M,K}, U_{KK}] = mTSLU(A_{K:M,K}, i, b_i, P_{r_i})$
8:         /* Apply permutation and compute block row of $U$ */
9:         $A_{K:M,:} = \Pi_{KK} A_{K:M,:}$
10:        **for each** compute unit at level $i$ owning a block $A_{K,J}$, $J = K+1$ to $N$ **do
           in parallel**
11:            $U_{K,J} = L_{KK}^{-1} A_{K,J}$   /* call multilevel dtrsm on $P_{i+1}$ compute units */
12:        **end for**
13:        /* Update the trailing submatrix */
14:        **for each** compute unit at level $i$ owning a block $A_{I,J}$ of the trailing submatrix,
           $I, J = K+1$ to $M, N$ **do in parallel**
15:            $A_{I,J} = A_{I,J} - L_{I,K} U_{K,J}$   /* call multilevel dgemm on $P_{i+1}$ compute units
           */
16:        **end for**
17:    **end for**
18: **end if**
```

2.1 Performance model

In this section we present a performance model of the 2-level CALU factorization of a matrix of size $n \times n$ in terms of the number of floating-point operations (#flops), the volume of communication (#words moved), and the number of messages exchanged (#messages) during the factorization. Let $P_1 = P_{r1} \times P_{c1}$ be the number of processors and b_1 be the block size at the first level of parallelism.

Algorithm 2 mTSLU: multilevel panel factorization

```
1: Input: matrix $B$, level of parallelism $i$ in the hierarchy, block size $b_i$, number of
   compute units $P_{r_i}$
2: Partition $B$ on $P_{r_i}$ blocks /* Here $B = (B_1^T, B_2^T, ..., B_{P_{r_i}}^T)^T$ */
3: /*Each compute unit owns a block $B_I$*/
4: for each block $B_I$ do in parallel
5:    $[\Pi_I, L_I, U_I] = mCALU(B_I, i+1, b_{i+1}, P_{i+1})$
6:    Let $B_I$ be formed by the pivot rows, $B_I = (\Pi_I B_I)(1 : b_i, :)$
7: end for
8: for $level = 1$ to $log_2(P_{r_i})$ do
9:    for each block $B_I$ do in parallel
10:      if $((I-1) \quad mod \quad 2^{level-1} == 0)$ then
11:         $[\Pi_I, L_I, U_I] = mCALU([B_I; B_{I+2^{level-1}}], i+1, b_{i+1}, P_{i+1})$
12:         Let $B_I$ be formed by the pivot rows, $B_I = (\Pi_I [B_I; B_{I+2^{level-1}}])(1 : b, :)$
13:      end if
14:   end for
15: end for
16:    Let $\Pi_{KK}$ be the permutation performed for this panel
17: /* Compute block column of $L$ */
18: for each block $B_I$ do in parallel
19:    $L_I = B_I U_1(1 : b_i, :)^{-1}$     /* using multilevel dtrsm */
20: end for
21: end if
```

Each compute unit at the first level is formed by $P_2 = P_{r2} \times P_{c2}$ compute units at the second level of parallelism. The total number of compute units at the second level is $P = P_1 \cdot P_2$. Let b_2 the block size at the second level of parallelism. We note CALU(m, n, P, b) the routine that performs 1-level CALU on a matrix of size $m \times n$ with P processors and a panel of size b.

We first consider the arithmetic cost of 2-level CALU. It is formed by the factorization of the panel, the computation of a block row of U, and the update of the trailing matrix, at each step k of the algorithm. To factorize the panel k of size b_1, we perform 1-level CALU on each block of the reduction tree, using a grid of P_2 smaller compute units and a panel of size b_2. The number of flops performed to factor the k-th panel is,

$$\#\text{flops}(CALU(\frac{n_k}{P_{r1}}, b_1, P_2, b_2)) + \log P_{r1} \cdot \#\text{flops}(CALU(2b_1, b_1, P_2, b_2)),$$

where n_k denotes the number of columns of the k-th panel. To perform the rank-b_1 update, first the input matrix of size $n \times n$ is divided into P_1 blocks of size $\frac{n}{P_{r1}} \times \frac{n}{P_{c1}}$. Then each block is further divided among P_2 compute units. Hence each processor from level two computes a rank-b_1 update on a block of size $\frac{n}{P_{r1} \times P_{r2}} \times \frac{n}{P_{c1} \times P_{c2}}$. It is then possible to estimate the flops count of this step as a rank-b_1 update of a matrix of size $n \times n$ distributed into $P_{r1}.P_{r2} \times P_{c1}.P_{c2}$ processors. The same reasoning holds for the arithmetic cost of the computation of block row of U.

We estimate now the communication cost at each level of parallelism. At the first level we consider the communication between the P_1 compute units. This corresponds to the communication cost of the initial 1-level CALU algorithm, which is presented in detail in [12]. The size of the memory of one compute unit at the first level is formed by the sum of the sizes of the memories of the compute units at the second level. We consider here that this size is of $O(n^2/P_1)$, that is each node stores a part of the input and output matrices, and this is sufficient for determining a lower bound on the volume of communication that needs to be performed during our algorithm. However, the number of messages that are transferred at this level depends on the maximum size of data that can be transferred from one compute unit to another compute unit in one single message. We consider here the case when the size of one single message is of the order of n^2/P_1, which is realistic if shared memory is used at the second level of parallelism. However, if the size of one single message is smaller, and it can be as small as n^2/P when distributed memory is used at the second level of parallelism, the number of messages and the lower bounds presented in this section need to be adjusted for the given memory size.

At the second level we consider in addition the communication between the P_2 smaller compute units inside each compute unit of the first level. We note that we consider Cannon's matrix-matrix multiplication algorithm [4] in our model. Here we detail the communication cost of the factorization of a panel k at the second level of parallelism. Inside each node we first distribute the data on a grid of P_2 processors, then we apply 1-level CALU using P_2 processors and a panel of size b_2:

$$\begin{aligned}
\#\text{messages}_k &= \#\text{messages}(CALU(\tfrac{n_k}{P_{r1}}, b_1, P_2, b_2)) \\
&\quad + \log P_{r1} \cdot \#\text{messages}(CALU(2b_1, b_1, P_2, b_2)) + \log P_2 \times (1 + \log P_{r1}), \\
\#\text{words}_k &= \#\text{words}(CALU(\tfrac{n_k}{P_{r1}}, b_1, P_2, b_2)) \\
&\quad + \log P_{r1} \cdot \#\text{words}(CALU(2b_1, b_1, P_2, b_2)) + b_1^2 \log P_2 \times (1 + \log P_{r1}).
\end{aligned}$$

We estimate now the performance model for a square matrix using an optimal layout, that is we choose values of P_{ri}, P_{ci}, and b_i at each level of the hierarchy that allow to attain the lower bounds on communication. By following the same approach as in [12], for two levels of parallelism these parameters can be written as $P_{r1} = P_{c1} = \sqrt{P_1}$, $P_{r2} = P_{c2} = \sqrt{P_2}$, $b_1 = \frac{n}{\sqrt{P_1}} \log^{-2} P_1$, and $b_2 = \frac{b_1}{\sqrt{P_2}} log^{-2} P_2 = \frac{n}{\sqrt{P_1 P_2}} log^{-2} P_1 log^{-2} P_2$. We note that $P = P_1 \cdot P_2$. Table 1 presents the performance model of 2-level CALU. It shows that 2-level CALU attains the lower bounds on communication of dense LU, modulo polylogarithmic factors, at each level of parallelism.

2.2 Implementation on a cluster of multicore processors

In the following we describe the specific implementation of these algorithms on a cluster of nodes of multicore processors. At the top level, we have used a static distribution of the data on the nodes, more specifically the matrix is distributed using a two-dimensional block cyclic partitioning on a two-dimensional grid of

Table 1. Performance estimation of parallel (binary tree based) 2-level CALU with optimal layout. The matrix factored is of size $n \times n$. Some lower-order terms are omitted.

	Communication cost at the first level of parallelism	Lower bound	Memory size
# messages	$O(\sqrt{P_1} \log^3 P_1)$	$\Omega(\sqrt{P_1})$	$O(\frac{n^2}{P_1})$
# words	$O(\frac{n^2}{\sqrt{P_1}} \log P_1)$	$\Omega(\frac{n^2}{\sqrt{P_1}})$	$O(\frac{n^2}{P_1})$
	Communication cost at the second level of parallelism		
# messages	$O(\sqrt{P} \log^6 P_1 + \sqrt{P} \log^3 P_1 \log^3 P_2)$	$\Omega(\sqrt{P})$	$O(\frac{n^2}{P})$
# words	$O(\frac{n^2}{\sqrt{P}} \log^3 P_1 \log P_2)$	$\Omega(\frac{n^2}{\sqrt{P}})$	$O(\frac{n^2}{P})$
	Arithmetic cost of 2-level CALU		
# flops	$\frac{1}{P}\frac{2n^3}{3} + \frac{3n^3}{2P\log^2 P} + \frac{5n^3}{6P\log^3 P}$	$\frac{1}{P}\frac{2n^3}{3}$	

processors. This is similar to the distribution used in 1-level CALU [13], and hence the communication between nodes is performed as in the 1-level CALU factorization. For each node, the blocks are again decomposed using a two-dimensional layout, and the computation of each block is associated with a task. A dynamic scheduler is used to schedule the tasks to the available threads as described in [7].

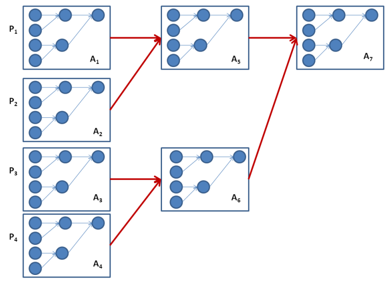

Fig. 1. Multilevel TSLU on a computer system with two levels of parallelism.

Figure 1 shows an example of execution of multilevel TSLU algorithm on a cluster of 4 nodes, each node being formed by a 4 cores processor. The white squares represent the nodes and the blue circles represent the cores inside a node. We consider that a binary tree is used at each level of the hierarchy of parallelism. The figure shows two levels of reduction tree. The red lines represent communication between different nodes during one panel factorization, and the blue lines represent synchronization between the different cores during one step of a smaller panel factorization performed at the second level of parallelism.

3 Experimental results

In this section we discuss first the stability of 2-level CALU, and then we evaluate its performance on a cluster of multicore processors.

3.1 Stability

It was shown in [12] that CALU is as stable as GEPP in practice. Since 2-level CALU is based on a recursive call to CALU, its stability can be different from 1-level CALU. We present here a set of experiments performed on random matrices and a set of special matrices that were also used in [12] for discussing the stability of 1-level CALU, where a detailed description of these matrices can be found. The size of the test matrices is varying from 1024 to 8192. We use different combinations of the number of processors P_1 and P_2 and of the panel sizes b_1 and b_2. We study both the stability of the LU decomposition and of the linear solver, in terms of growth factor and three different backward errors, the normwise backward error, the componentwise backward error, and $\|PA - LU\|/\|A\|$. For all the test matrices, the worst growth factor obtained is smaller than 10^2. Figure 2 shows the ratios of 2-level CALU's backward errors to those of GEPP. For almost all test matrices, 2-level CALU's normwise backward error is at most $3\times$ larger that GEPP's normwise backward error. However, for special matrices the other two backward errors of 2-level CALU can be larger by a factor of the order of 10^2 than the corresponding backward errors of GEPP. We note that for several test matrices, at most two steps of iterative refinement are used to attain the machine epsilon. These experiments show that 2-level CALU exhibits a good stability, however further investigation is required if more than two levels of parallelism are to be used.

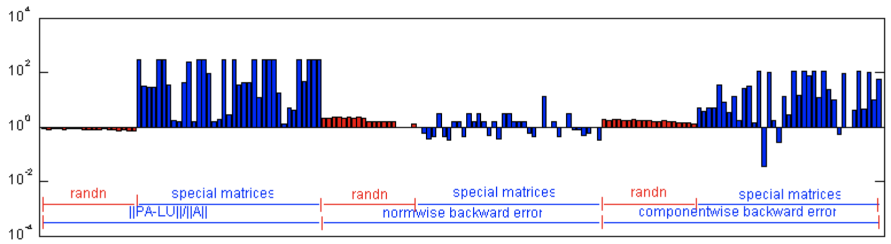

Fig. 2. The ratios of 2-level CALU's backward errors to GEPP's backward errors.

3.2 Performance of 2-level CALU

We perform our experiments on a cluster composed of 32 nodes based on Intel Xeon EMT64 processors running on Linux, each node is a two-socket, quad-core processor. Each core has a frequency of 2.50 GHz. The cluster is part of Grid 5000 [5]. Both ScaLAPACK and multilevel CALU are linked with BLAS from MKL 10.1. We compare the performance of our algorithm with the corresponding routine from ScaLAPACK. Our algorithm is implemented using MPI and Pthreads. In the experiments, P refers to the number of nodes (which corresponds to the number of compute units P_1 at the first level of parallelism described in section 2), T refers to the number of threads per node (which corresponds to the number of compute units P_2 at the second level of parallelism described in section 2), and b_1 and b_2 are the block sizes used respectively at the first and the second level of the hierarchy. The choice of the block size at each level depends on the architecture, the number of levels in the hierarchy, and the input matrix size. In our experiments, we empirically tune the block sizes b_1 and b_2. This tuning is simple because we only have two levels of parallelism, but it should be replaced by an automatic approach for multiple levels of parallelism.

At the second level of parallelism, 2-level CALU uses a grid $T = T_r \times T_c$ of threads, where T_r is the number of threads on the vertical dimension working on the panel, and T_c is the number of threads on the horizontal dimension. Here we evaluate the performance of three different parametric choices, $T = 8 \times 1$, $T = 4 \times 2$, and $T = 2 \times 4$. We note that ScaLAPACK is executed over P MPI processes, and in each node we call multithreaded BLAS routines with T threads.

Figure 3 shows the performance of 2-level CALU and GEPP as implemented in ScaLAPACK for tall and skinny matrices with varying number of rows. We observe that 2-level CALU is scalable and faster than ScaLAPACK. For a matrix of size $10^6 \times 1200$, 2-level CALU is twice faster than ScaLAPACK. Furthermore, an important loss of performance is observed for ScaLAPACK when $m = 10^4$, while all the variants of 2-level CALU lead to good performance.

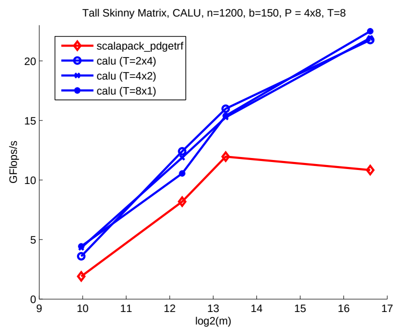

Fig. 3. Performance of 2-level CALU and ScaLAPACK on $P = 4 \times 8$ nodes, for matrices with n=1200, m varying from 10^3 to 10^6, $b_1 = 150$, and $b_2 = MIN(b_1, 100)$.

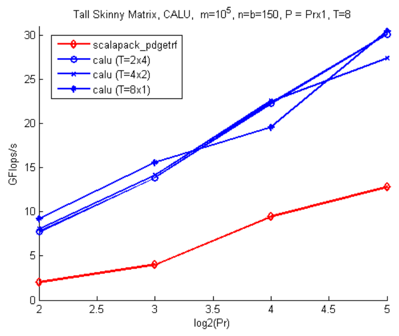

Fig. 4. Performance of 2-level CALU and ScaLAPACK on $P = P_r \times 1$ nodes, for matrices with $n = b_1 = 150$, $m = 10^5$, and $b_2 = MIN(b_1, 100)$.

Figure 4 shows the performance of 2-level CALU on tall and skinny matrices with varying number of processors working on the panel factorization. Since the matrix has only one panel, recursive TSLU is called at the top level of the hierarchy and the adapted multhithreaded CALU is called at the second level. We observe that for $P_r = 4$, 2-level CALU is 4.5 times faster than ScaLAPACK. When $P_r > 4$, ScaLAPACK's performance is at most 10 GFlops/s, while 2-level CALU's performance is up to 30 GFlops/s, that is twice faster. We note that for $P_r = 8$, the variant $T = 8 \times 1$ is slightly better than the others. This shows the importance of determining a good layout at runtime.

Figure 5 shows the performance of 2-level CALU on a square matrix when the number of processors varies. For each value of P, we use the same layout for ScaLAPACK and at the top level of parallelism of 2-level CALU. The layout

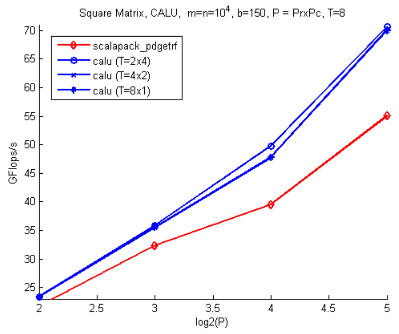

Fig. 5. Performance of 2-level CALU and ScaLAPACK on $P = P_r \times P_c$ nodes, for matrices with $m = n = 10^4$, $b_1 = 150$, and $b_2 = MIN(b_1, 100)$.

at the second level is one of the three variants discussed previously. We observe that all the variants of 2-level CALU are faster than ScaLAPACK. The variant $T = 2 \times 4$ is usually better than the others. This behavior has also been observed for multithreaded 1-level CALU [7]. We recall that the matrix is partitioned into $T_r \times N/b$ blocks. Thus increasing T_r increases the number of tasks and the scheduling overhead, and this impacts the performance of the entire algorithm.

4 Conclusion

In this paper we have introduced a communication avoiding LU factorization adapted for a computer system with two levels of parallelism, which minimizes communication at each level of the hierarchy of parallelism in terms of both volume of data and number of messages exchanged during the decomposition. On a cluster of multicore processors, that is a machine with two levels of parallelism based on a combination of both distributed and shared memories, our experiments show that our algorithm is faster than the corresponding routine from ScaLAPACK. On tall and skinny matrices, a loss of performance is observed for ScaLAPACK when the number of rows increases, while 2-level CALU shows an improving speedup. Our performance model shows that 2-level CALU increases the number of flops, words, and messages just by a polylogarithmic factor.

As future work, we plan to model and to evaluate the performance of our algorithm for multiple levels of parallelism, and to extend the same approach to other factorizations as QR. It will also be important to evaluate the usage of autotuning for computing the optimal layout and the optimal block size at each level of parallelism.

References

[1] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. Langem. QR factorization of tall and skinny matrices in a grid computing environment. In *Parallel Distributed Processing Symposium (IPDPS)*, pages 1–11. IEEE, 2010.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 1999.

[3] 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.C. Whaley. Scalapack: A linear algebra library for message-passing computers. In *In SIAM Conference on Parallel Processing*, 1997.

[4] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[5] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jegou, S. Lanteri, N. Melab, R. Namyst, P.V.B. Primet, O. Richard, et al. Grid5000: a nation wide experimental grid testbed. *International Journal on High Performance Computing Applications*, 20(4):481–494, 2006.

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

[7] S. Donfack, L. Grigori, and A. K. Gupta. Adapting communication-avoiding LU and QR factorizations to multicore architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2010.

[8] E. Elmroth and F. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, pages 120–128, 1998.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[10] R. A. Van De Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency Practice and Experience*, 9(4):255–274, 1997.

[11] S. L. Graham, M. Snir, and C. A. Patterson. *Getting up to speed: The future of supercomputing*. National Academies Press, 2005.

[12] L. Grigori, J. Demmel, and H. Xiang. CALU: A communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32:1317–1350, 2011.

[13] L. Grigori, J.W. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 29. IEEE Press, 2008.

[14] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. ACM, 1981.

[15] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.