Communication Avoiding Gaussian Elimination

Laura GRIGORI* James W. DEMMEL† Hua XIANG ‡

Abstract

We present CALU, a Communication Avoiding algorithm for the LU factorization of dense matrices distributed in a two-dimensional cyclic layout. The algorithm is based on a new pivoting strategy, which is stable in practice. The new algorithm is optimal (up to polylogarithmic factors) in the amount of communication it performs.

Our experiments show that CALU leads to a reduction in the parallel time, in particular when the latency time is an important factor of the overall time. The factorization of a block-column, a subroutine of CALU, outperforms the corresponding routine PDGETF2 from ScaLAPACK up to a factor of 4.37 *on an IBM POWER 5 system and up to a factor of* 5.58 *on a Cray XT4 system. On square matrices of order* 10^4*, CALU outperforms the corresponding routine PDGETRF from ScaLAPACK by a factor of* 1.24 *on IBM POWER 5 and by a factor of* 1.31 *on Cray XT4.*

1 Introduction

Solving linear systems of equations is one of the most common operation in scientific computing. These applications frequently lead to solving very large dense sets of linear equations, often with millions of rows and columns, and solving these problems is very time consuming.

In this paper we present a Communication-Avoiding LU factorization (CALU) algorithm for computing the LU factorization of a dense matrix A distributed in a two-dimensional (2D) layout. CALU is based on a new pivoting strategy, that we show it is numerically stable in practice. CALU has two main characteristics. First, it is latency avoiding, as the new pivoting strategy allows for a significant decrease in the number of messages exchanged during the factorization relative to conventional algorithms, though that comes at the cost of a small number of redundant computations. We refer to the new pivoting strategy as ca-pivoting. This approach is thus particularly beneficial on parallel architectures and for matrix sizes for which the overhead of sending a message between two processors is an expensive factor in the algorithm. Moreover, today's technology trends predict that arithmetic will continue to improve exponentially faster than bandwidth, and bandwidth exponentially faster than latency (see e.g. [8]). So CALU is well suited for future parallel architectures, in which conventional algorithms will spend more and more of their time communicating and less and less doing arithmetic. Second, unlike conventional algorithms, CALU allows the usage of the best available sequential algorithm for computing the LU factorization of a block-column, as for example the recursive algorithms [9, 13].

CALU uses a block right-looking approach in which a dense matrix A in a 2D layout is factorized by traversing iteratively blocks of columns. At each iteration, a block-column of width b is factored first. Then the trailing matrix is updated, and the decomposition continues on the trailing matrix. The main difference with respect to other block right-looking algorithms lies in the factorization of a block-column, which is performed very efficiently in CALU by using the new ca-pivoting strategy as follows. Unlike conventional partial pivoting, the LU decomposition of the block-column is performed in two steps. The first step, a preprocessing step, identifies efficiently in parallel b pivot rows, that provide good pivots for the LU factorization of the entire block-column. We describe in detail later in the paper how these rows are identified. The pivot rows are permuted to be in the first b positions of the block-column. In the second step the LU factorization with no pivoting of the block-column is performed. We refer to this approach for performing the LU factorization of a block-column as TSLU (Tall Skinny LU), since a block-column can be considered to be a matrix with a 1D layout for which the vertical dimension (number of rows) is much larger than the horizontal dimension (number of columns).

The new algorithm CALU is optimal (up to polylogarithmic factors) in the amount of communication it performs. This is shown in [5] where known lower bounds on communication bandwidth for parallel ma-

*INRIA Saclay-Ile de France, Bat 490, Universite Paris-Sud 11, 91405 Orsay France (laura.grigori@inria.fr).

†Computer Science Division and Mathematics Department, UC Berkeley, CA 94720-1776, USA (demmel@cs.berkeley.edu).

‡INRIA Saclay-Ile de France, Bat 490, Universite Paris-Sud 11, 91405 Orsay France (hua.xiang@inria.fr).

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

trix multiplication are extended to determine lower bounds on communication latency. These bounds are used to provide lower bounds on both bandwidth and latency for dense LU factorization. With an optimal choice of matrix layout, CALU attains both the bandwidth and the latency lower bounds, while the classic LU factorization as implemented in ScaLAPACK PDGETRF routine [4] attains only the bandwidth lower bounds. In addition, [5] presents also a communication avoiding QR (CAQR) factorization algorithm that attains the same communication lower bounds. CALU bears some similarities to CAQR algorithm discussed in [5]. Both algorithms use a reduction-like computation for the panel factorization, thus decreasing the communication cost. However the numerical stability issues related to the LU factorization lead to a number of significant differences. For instance, CALU performs the panel factorization twice, but the update of the trailing matrix is the same as in the classic LU factorization. In CAQR, the panel factorization is performed once, but there is some redundant computation in the update of the trailing matrix.

In this paper we present performance models for general matrix layouts that show that CALU overcomes the latency bottleneck of the LU factorization as implemented in the PDGETRF routine. In PDGETRF, the LU decomposition of an $m \times n$ matrix is performed in parallel using a block cyclic distribution of the matrix over a P_r by P_c grid of processors, where $P_r \cdot P_c = P$ and P is the number of processors. The latency bottleneck in ScaLAPACK lies in the LU factorization of a block-column that is spread over P_r processors, that leads to $2n \log_2 P_r$ messages communicated during the factorization. All the other terms in the number of messages are of the form $O(n/b) \log_2 P_r + O(n/b) \log_2 P_c$, where b is the size of the block used in the 2D distribution. CALU sends only $3(n/b) \log_2 P_r + 3(n/b) \log_2 P_c$ messages, i.e. smaller by a factor of b. The price for fewer messages is $b(mn - n^2/2)/P_r$ more floating point work, which is a small fraction of the overall $(mn^2 - n^3/3)/P$ work.

This paper focuses on comparing CALU with the approach used in ScaLAPACK PDGETRF, and the parallel implementation we present for CALU follows the main steps used in ScaLAPACK. However, the ca-pivoting scheme can be used in other parallel algorithms implementing the LU factorization, leading to the same reduction in communication. It could potentially be used for example in the highly optimized High Performance Linpack (HPL) benchmark, used in determining the Top500 list [1].

The new ca-pivoting scheme used in CALU may lead to a different row permutation than the classic LU factorization. In this paper we present numerical results that show that ca-pivoting scheme is stable in practice. We observe that it behaves as a threshold pivoting, where the minimum threshold value in practical experiments is 0.33. In other words, $|L|$ is bounded by 3, while in LU factorization with partial pivoting, $|L|$ is bounded by 1,where $|L|$ denotes the matrix of absolute values of the entries of L. Extensive testing on many different matrices always resulted in residuals $||Ax - b||$ comparable to those from conventional partial pivoting.

Several different pivoting strategies were recently considered by other authors. For multicore/multithreaded architectures, an extension of pairwise pivoting for block algorithms is used in [3, 10]. For grid computing, a technique reffered to as batched pivoting is proposed in [7]. Since batched pivoting has some similarities with ca-pivoting, we will discuss it more in detail later in the paper. In [2] the authors consider an approach in which the input matrix is transformed by a random matrix such that with high probability there is no need to pivot during Gaussian elimination. This is motivated by GPUs, where the cost of pivoting can be substantial; for a solution to this problem for GPUs see [15]. Our pivoting strategy is different in two aspects. The elimination of each column of A leads to a rank-one update. This property, present in LU with partial pivoting and in batched pivoting, but not in pairwise pivoting, is shown experimentally to be important for the stability of Gaussian elimination [14]. Our experimental results on sparse matrices (not presented in this paper) from various real applications show that ca-pivoting strategy is stable for sparse matrices as well. In contrast, batched pivoting can fail for sparse matrices.

The rest of the paper is organized as follows. Section 2 introduces CALU and the new ca-pivoting scheme. Section 3 describes the parallel LU factorization of a tall-skinny matrix using ca-pivoting, and discusses its performance in terms of computation and communication cost. Section 4 presents the parallel CALU algorithm of a matrix distributed in a 2D layout and discusses its computation and communication cost. Section 5 compares the classic LU factorization algorithms implemented in ScaLAPACK and the new proposed CALU algorithm. Section 6 describes experimental results that first discuss the stability of ca-pivoting scheme, and second evaluate the performance of CALU on two computational systems, an IBM POWER 5 system and a Cray XT4 system, located at National Energy Research Scientific Computing Center (NERSC). And Section 7 presents the conclusions and our future work.

2 Description of CALU

In this section we describe the main steps of CALU algorithm for computing the LU factorization of a matrix A of size $m \times n$. We use several notations. We

refer to the submatrix of A formed by elements of row indices from i to j and column indices from d to e as $A(i\colon j, d\colon e)$. When A is partitioned in P block rows, we use the notation $A = [A_0; A_1; \cdots; A_{P-1}]$ where the A_i are stacked atop one another. If A is the result of the multiplication of two matrices B and C, we refer to the submatrix of A as $(BC)(i\colon j, d\colon e)$.

CALU is a block algorithm that factorizes the input matrix by traversing iteratively blocks of columns. At the first iteration, the matrix A is partitioned as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

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 $(m-b) \times (n-b)$. As other classic right looking algorithms, CALU first computes the LU factorization of the first block-column, then determines the block U_{12}, and updates the trailing matrix A_{22}.

The main difference with respect to other existing algorithms lies in the factorization of the first block-column. CALU uses the new ca-pivoting strategy, consisting in performing first a preprocessing step in which a good set of pivot rows is identified. Second, the pivot rows are permuted in the first b positions of matrix A and the LU factorization with no pivoting of the first block-column is performed. CALU considers that the first block-column is partitioned in P block-rows $[A_0; A_1; \ldots; A_{P-1}]$. We consider $P = 4$ and we suppose that m divides 4.

The preprocessing step starts by performing the LU factorization with partial pivoting of each block-row A_i, as displayed in Equation 1. This leads to a decomposition in which the first factor, $\bar{\Pi}_0$, is an $m \times m$ block diagonal matrix, where each diagonal block $\bar{\Pi}_{i0}$ is a permutation matrix. The second factor, $\bar{L}_0$, is an $m \times Pb$ block diagonal matrix, where each diagonal block $\bar{L}_{i0}$ is an $m/P \times b$ lower unit trapezoidal matrix. The third factor, $\bar{U}_0$, is a $Pb \times b$ matrix, where each block $\bar{U}_{i0}$ is a $b \times b$ upper triangular factor. Note that this step aims at identifying in each block-row a set of b linearly independent rows, which correspond to the first b rows of $\bar{\Pi}_{i0}^T A_i$, with $i = 0 \ldots 3$.

From the P sets of local pivot rows, we perform a binary tree (of depth $\log_2 P = 2$ in our example) of LU factorizations of matrices of size $2b \times b$ to identify b global pivot rows. The 2 LU factorizations at the leaves of our depth-2 binary tree are shown in Equation 2, combined in one matrix. This decomposition leads to a $Pb \times Pb$ permutation matrix $\bar{\Pi}_1$, a $Pb \times 2b$ factor $\bar{L}_1$ and a $2b \times b$ factor $\bar{U}_1$.

The global pivot rows are obtained after applying one more LU decomposition (at the root of our depth-2 binary tree) on the pivot rows identified previously. This is displayed in Equation 3. Note that if one or more zero pivots are encountered in one of these LU factorizations, the factorization is still completed (the correspoding U factor will be singular), and a number of rows smaller than b are passed to the next step.

The permutations identified in the preprocessing step are applied on the original matrix A. Then the LU factorization with no pivoting of the first block-column is performed, the block-row of U is computed and the trailing matrix is updated. Note that $U_{11} = \bar{U}_2$. The factorization continues on the trailing matrix $\tilde{A}_{22}$. The permutation matrices $\bar{\Pi}_0, \bar{\Pi}_1, \bar{\Pi}_2$ do not have the same dimensions. By abuse of notation, we consider that $\bar{\Pi}_1, \bar{\Pi}_2$ are extended by the appropriate identity matrices to the dimension of $\bar{\Pi}_0$.

$$\bar{\Pi}_2^T \bar{\Pi}_1^T \bar{\Pi}_0^T A = \begin{bmatrix} L_{11} & \\ L_{21} & I_{n-b} \end{bmatrix} \cdot \begin{bmatrix} I_b & \\ & \tilde{A}_{22} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ & I_{n-b} \end{bmatrix}$$

The ca-pivoting strategy has several important characteristics. First, when $b = 1$ or $P = 1$, ca-pivoting is equivalent to partial pivoting. Second, the elimination of each column of A leads to a rank-1 update of the trailing matrix. The rank-1 update property is shown experimentally to be very important for the stability of LU factorization [14]. A large rank update might lead to an unstable LU factorization, as for example in another strategy suitable for parallel computing called parallel pivoting [14]. Third, the numerical tests presented in Section 6 show that it can be regarded as a threshold pivoting strategy.

Batched pivoting [7] has some similarities with our approach. To factor a block column partitioned as $[A_0; A_1; \ldots; A_{P-1}]$, batched pivoting identifies first b rows, that are then used as pivots for the entire block column. For this, each block A_i is factored using Gaussian elimination with partial pivoting, which corresponds to the first computation in our preprocessing step in Equation 1. From the P sets of b rows, the set considered by some criterion as the best will be used to factor the entire block column. Our approach is different because it uses a binary tree of LU factorizations to identify from the P sets of rows the final b rows that will be used for pivoting. Also batched pivoting fails when each block row A_i is singular, while the block-column is nonsingular. This can happen for example in the case of sparse matrices.

3 TSLU algorithm

In this section we present TSLU, a parallel algorithm for computing the LU factorization of an $m \times b$ matrix A, with $m \gg b$, which is distributed over P processors using a 1D layout. We also discuss its performance in terms of flops and number of messages exchanged during the factorization. This algorithm will be used in CALU for performing the factorization of a block-column.

TSLU essentially does an all-reduction (with a but-

$$A(:,1:b) = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{00}\bar{L}_{00}\bar{U}_{00} \\ \bar{\Pi}_{10}\bar{L}_{10}\bar{U}_{10} \\ \bar{\Pi}_{20}\bar{L}_{20}\bar{U}_{20} \\ \bar{\Pi}_{30}\bar{L}_{30}\bar{U}_{30} \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{00} & & & \\ & \bar{\Pi}_{10} & & \\ & & \bar{\Pi}_{20} & \\ & & & \bar{\Pi}_{30} \end{bmatrix} \cdot \begin{bmatrix} \bar{L}_{00} & & & \\ & \bar{L}_{10} & & \\ & & \bar{L}_{20} & \\ & & & \bar{L}_{30} \end{bmatrix} \cdot \begin{bmatrix} \bar{U}_{00} \\ \bar{U}_{10} \\ \bar{U}_{20} \\ \bar{U}_{30} \end{bmatrix} \quad (1)$$
$$\equiv \bar{\Pi}_0\bar{L}_0\bar{U}_0$$

$$\begin{bmatrix} \left(\bar{\Pi}_0^T A\right)(1:b,1:b) \\ \left(\bar{\Pi}_0^T A\right)(m/P+1:m/P+b,1:b) \\ \left(\bar{\Pi}_0^T A\right)(2m/P+1:2m/P+b,1:b) \\ \left(\bar{\Pi}_0^T A\right)(3m/P+1:3m/P+b,1:b) \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{01}\bar{L}_{01}\bar{U}_{01} \\ \bar{\Pi}_{11}\bar{L}_{11}\bar{U}_{11} \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{01} & \\ & \bar{\Pi}_{11} \end{bmatrix} \cdot \begin{bmatrix} \bar{L}_{01} & \\ & \bar{L}_{11} \end{bmatrix} \cdot \begin{bmatrix} \bar{U}_{01} \\ \bar{U}_{11} \end{bmatrix} \quad (2)$$
$$\equiv \bar{\Pi}_1\bar{L}_1\bar{U}_1$$

$$\begin{bmatrix} \left(\bar{\Pi}_1^T\bar{\Pi}_0^T A\right)(1:b,1:b) \\ \left(\bar{\Pi}_1^T\bar{\Pi}_0^T A\right)(2m/P+1:2m/P+b,1:b) \end{bmatrix} = \bar{\Pi}_{02}\bar{L}_{02}\bar{U}_{02} \equiv \bar{\Pi}_2\bar{L}_2\bar{U}_2 \quad (3)$$

terfly communication pattern) where the reduction operation is Gaussian elimination on a pair of matrices of size $b \times b$ stacked atop one another. For completeness, we describe this all-reduction operation in more detail as follows, and then show an example. The butterfly method uses a tree-like computation as described in the previous section, and takes place in $(\log_2 P + 1)$ steps, starting from the bottom level $k = 0$ of a binary tree. Each node of the binary tree is associated with a set of processors. For the sake of simplicity, we suppose that the processors are a power of two, numbered from 0 to $P-1$, and that m divides P. We use notations similar to [5]: $fstP(i,k)$ denotes the first processor affected to the node of the binary tree at level k to which processor i belongs; $target(i,k)$ refers to the processor with which processor i exchanges data at level k of the tree in a butterfly pattern; $tgtfstP(i,k)$ denotes the processor with which $fstP(i,k)$ exchanges data at level k; $level(i,k)$ denotes the node at level k of the binary tree which is assigned to a set of processors that includes processor i, and is computed as:

$$\begin{array}{lcl} level(i,k) & = & \left\lfloor \frac{i}{2^k} \right\rfloor \\ fstP(i,k) & = & 2^k level(i,k) \\ target(i,k) & = & fstP(i,k) + (i + 2^{k-1}) \ mod \ 2^k \\ tgtfstP(i,k) & = & fstP(i,k) + 2^{k-1} \end{array}$$

The algorithm starts with a local LU factorization on each processor of the $m/P \times b$ block-rows that it owns. Then at each level k of the binary tree and for each node at this level, pairs of processors perform redundantly an LU factorization. Consider for example a processor i and the node at level k which is mapped on processor i, and identified as $level(i,k)$. The factors L and U computed at this node are denoted as $\bar{L}_{level(i,k),k}$, $\bar{U}_{level(i,k),k}$. Processor i and its target processor exchange data and perform redundantly the LU factorization of two matrices of size $b \times b$.

Note that the sequence of local LU factorizations performed in the first three steps of TSLU are not performed in place (the input matrix is not overwritten). Hence TSLU needs an extra storage of size $m \times b$ to store the resulting L and U factors of these factorizations and a vector of size b to store the permutation vector.

TSLU algorithm

1. Let i be my processor number.
2. Compute the LU factorization of my $m/P \times b$ group of rows $A_i = \bar{\Pi}_{i0}\bar{L}_{i0}\bar{U}_{i0}$. Let B_i be formed by the b pivot rows, $B_i = \left(\bar{\Pi}_{i0}^T A_i\right)(1:b,1:b)$.
3. **for** $k = 1$ to $\log_2 P$ **do**

   if $i \geq tgtfstP(i,k)$ **then** $\phi = target(i,k)$, $\tau = i$
   else $\phi = i$, $\tau = target(i,k)$
   endif

   Let $l = level(i,k)$.

   (a) Processor ϕ exchanges its B_i with Processor τ.

   (b) Compute the LU factorization of the two matrices B_ϕ and B_τ of size $b \times b$ stacked one on top of another:

   $$\begin{bmatrix} B_\phi \\ B_\tau \end{bmatrix} = \bar{\Pi}_{l,k}\bar{L}_{l,k}\bar{U}_{l,k}$$

   (c) Let B_i be formed by the first b pivot rows of $\bar{\Pi}_{l,k}^T \begin{bmatrix} B_\phi \\ B_\tau \end{bmatrix}$

   end for
4. Let the final permutation be $\bar{\Pi} = \bar{\Pi}_0\bar{\Pi}_1 \ldots \bar{\Pi}_{\log_2 p}$ and permute the local $A = \bar{\Pi}^T A$.
5. Let $U = \bar{U}_{0\log_2 P}$, where U is the upper triangular factor of A.
6. Compute the local L factor, $L_i = A_i U^{-1}$.

We illustrate the execution of this algorithm on a small example in Figure 1, where we suppose that the matrix A of size 16×2 is distributed following a 1D block cyclic distribution, with blocks of size 2×2 on 4 processors. Let

$$A = \begin{bmatrix} 2\,0\,2\,0\,0\,1\,2\,0\,2\,1\,4\,1\,0\,0\,1\,4 \\ 4\,1\,0\,0\,1\,4\,1\,2\,0\,2\,1\,0\,0\,2\,0\,2 \end{bmatrix}^T$$

In this example, the 1st, 2nd, 9th, 10th rows are distributed on processor 0. First a local LU factorization is performed by each processor. For processor 0, the 1st and 9th rows are used as pivots. Second the processors 0 and 1 exchange the 2 rows used as pivots in the local LU factorization. Then they perform redundantly the LU factorization of the 4×2 matrix formed by these rows stacked one on top of another. Similarly, processors 2 and 3 exchange their rows and perform redundantly the LU factorization of the matrix formed by these rows. In the third step, processors 0 and 2 exchange the 2 pivot rows identified in the second step, and perform an LU factorization on the 4×2 matrix formed by these rows. The same computation is performed by the processors 1 and 3. The rows identified in the third step represent the pivots that will be used to factorize the entire matrix A. In this simple example, the pivot rows used by TSLU happen to be the same as those used by Gaussian elimination with partial pivoting.

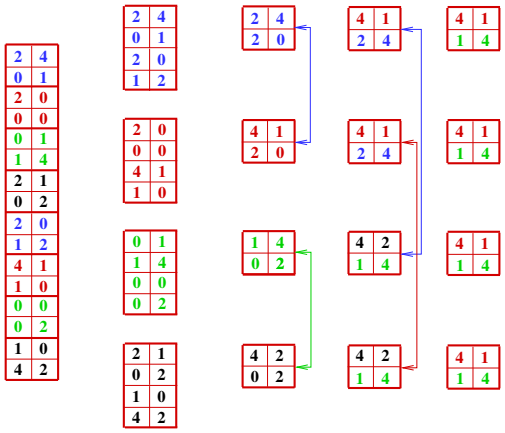

Figure 1: Example of execution of TSLU on 4 processors.

To study the performance of TSLU, we use a classical model to describe a machine architecture in terms of processor speed, network latency and bandwidth. Here and in the rest of the paper, we use one parameter to describe the time per flop (add and multiply), denoted γ, and one parameter to count the time per divide, denoted γ_d. We estimate the time for sending a message of m words between two processors as $\alpha + m\beta$, where α denotes the latency and β the inverse of the bandwidth. We approximate the time of broadcasts and combines that involve P processors by assuming $\log_2 P$ identical steps of communication and/or computation are needed. With these notations, the runtime of TSLU is estimated to be (we omit low order terms):

$$\begin{aligned} T_{TSLU}(m,b,P) &= \left[\frac{2mb^2}{P} + \frac{2b^3}{3}(\log_2 P - 1)\right]\gamma + \\ &+ b(\log_2 P + 1)\gamma_d + \\ &+ \log_2 P\alpha + b^2 \log_2 P\beta \end{aligned} \tag{4}$$

4 Parallel CALU algorithm for matrices distributed in a 2D layout

In this section we present a parallel algorithm that implements the CALU method presented in Section 2. We consider an $m \times n$ matrix block cyclically distributed over a bi-dimensional grid of processors $P = P_r \times P_c$, using square blocks of dimension $b \times b$. The parallel algorithm uses a block right-looking approach, as used for example in PDGETRF routine in ScaLAPACK or in HPL benchmark. That is, it iterates over block-columns of A, and at each step first a block-column of width b is factored. Then the trailing matrix is permuted and updated, and the decomposition continues on the trailing matrix. The main difference with the other algorithms is that CALU factors a block-column using the TSLU factorization presented in Section 3, which leads to an important reduction in the number of messages exchanged during the factorization.

Consider that the first $j-1$ iterations of the LU factorization were performed. That is, the first $j-1$ block columns were factored and the trailing matrix was permuted and updated. The active matrix at step j is of dimension $(m-(j-1)b) \times (n-(j-1)b) = m_j \times n_j$. For the clarity of presentation, we suppose that m and n divide b. We describe here the main steps involved in the j-th iteration of CALU:

1. The column of the grid that holds matrix block-column j computes its LU factorization using TSLU (Algorithm in Section 3).

2. Every processor in the processor column holding the matrix block-column j broadcasts along its processor row the locally stored subblock of L. It also broadcasts an array of size b that stores the permutation vector Π_j associated with the LU factorization of block-column j.

3. The matrix A is permuted according to Π_j.

4. Every processor in the processor row holding the matrix block-row j of U computes its local block.

5. Every processor in the processor row holding matrix block-row j of U broadcasts its local block down its column.

6. All processors update the trailing matrix.

In our current implementation, we use routines from ScaLAPACK for several steps of CALU. Step 3 is performed by a call to PDLASWP, step 4 is done by PDTRSM, and steps 5 and 6 correspond to a call to PDGEMM. However, CALU can be implemented differently, and can incorporate techniques which allow some overlap between computation and communication as the so-called look-ahead technique used in HPL benchmark.

To estimate the performance of CALU, we assume that the network bandwidth and latency is not necessarily the same everywhere, e.g. it can be different along columns of the grid than along rows of the grid. We use a different bandwidth and latency for communication between processors in different rows and the same column (α_c and β_c) versus different columns and the same rows (α_r and β_r). This is a first step towards understanding certain hierarchical parallel machines, where there is high bandwidth among processors on the same chip (or node or module) and lower between processors on different chips (or nodes or modules).

The total computation time over a rectangular grid of processors is given in Equation 5 (we omit some lower order terms and the time of pivoting rows locally). In this estimation we consider that a total of $(2n/b) \cdot \log_2 P_r$ messages are exchanged for swapping rows of matrix A in step 3. This is because the swapping of b rows occurs after each block-column factorization. Hence, this operation can be implemented in two steps, using $2\log_2 P_r$ messages. First each processor sends at most b rows that need to be swapped to the root processor as a reduce operation. Second the root processor broadcasts the necessary rows to all the processors in its processor column. However in our current implementation we use PDLASWP, and this routine performs one message exchange for each row swap, which leads to a total of $n\log_2 P_r$ messages exchanged for step 3. In our current work, we are replacing this routine by a routine that is implemented as explained above, and we include the number of messages associated with the future routine instead of PDLASWP in our time estimation.

5 Comparison with the ScaLAPACK's LU factorization

Consider that we decompose an $m \times n$ matrix which is distributed block cyclically over a P_r by P_c grid of processors, where $P_r \cdot P_c = P$ and $m \geq n$. The two-dimensional block cyclic distribution uses square blocks of dimension $b \times b$. The algorithm loops over n/b block-columns. At the j-th step, the first $j-1$ block-columns of L and block-rows of U are already computed. At this step, the block-column j of L is factored (call to $PDGETF2$) using pivoting. The pivoting information is applied to the rest of the matrix (call to $PDLASWP$). The block-row j of U is computed using triangular solves (call to PDTRSM), and then the trailing matrix is updated (call to $PDGEMM$).

Equation 6 represents the runtime estimation of PDGETRF routine in ScaLAPACK LU. To be consistent with the runtime estimation of CALU, we consider for PDGETRF as well that the swapping of b rows performed by a call to PDLASWP leads to $2 \cdot \log_2 P_r$ messages exchanged.

To better understand the differences between CALU and the LU factorization implemented in ScaLAPACK, we will compare the runtime estimation of the two factorizations as given by Equation 5 and Equation 6. Comparing the additions, multiplications flop counts, CALU adds a lower order term of about $b(mn - n^2/2)/P_r$. This term comes from TSLU, which performs twice the factorization of a block-column, first to get the pivot rows, and second to actually compute the factors. Comparing the division flop counts, CALU adds a lower order term of $n\log_2 P_r$, all from the TSLUs of block-columns (the factorizations of two $b \times b$ matrices). Comparing communication costs within processor columns (α_c and β_c terms), for bandwidth, both algorithms have the same communication volume. For latency, CALU is lower by a factor of $b(1+1/\log_2 P_r)$. The reduction in the number of messages within processor columns comes from the reduction in the factorization of a block-column performed by TSLU versus PDGETF2. Comparing communications costs within processor rows (α_r and β_r terms), in PDGETRF, the number of broadcasts within processor rows is already of the order of n/b, and hence both algorithms have the same costs.

6 Experimental results

In this section, we evaluate the performance of CALU algorithm, and the goal of our experiments is three-fold. First, we study the numerical stability of the new ca-pivoting strategy. Second, we evaluate the performance improvement obtained in the panel factorization by TSLU compared to the corresponding routine in ScaLAPACK. And third, we evaluate the performance of CALU and compare it to PDGETRF routine in ScaLAPACK.

The experiments are performed on two computational systems at the National Energy Research Scientific Computing Center (NERSC). The first system is an IBM p575 POWER 5 system, which has 888 compute processors distributed among 111 compute nodes. Each processor is clocked at 1.9 GHz and has a theo-

$$
\begin{aligned}
T_{CALU}(m,n,P_r,P_c) &= \left[\frac{1}{P}\left(mn^2-\frac{n^3}{3}\right)+\frac{1}{P_r}\left(mn-\frac{n^2}{2}\right)2b+\frac{n^2b}{2P_c}+\frac{2nb^2}{3}(\log_2 P_r-1)\right]\gamma+ \\
&+n(\log_2 P_r+1)\gamma_d+ \\
&+\log_2 P_r\left[\frac{3n}{b}\alpha_c+\left(\frac{nb}{2}+\frac{3n^2}{2P_c}\right)\beta_c\right]+ \\
&+\log_2 P_c\left[\frac{3n}{b}\alpha_r+\left(\frac{1}{P_r}\left(mn-\frac{n^2}{2}\right)\right)\beta_r\right] \qquad (5)\\
T_{PDGETRF}(m,n,P_r,P_c) &= \left[\frac{1}{P}\left(mn^2-\frac{n^3}{3}\right)+\frac{1}{P_r}\left(mn-\frac{n^2}{2}\right)b+\frac{n^2b}{2P_c}\right]\gamma+ \\
&+n\gamma_d+ \\
&+\left[2n\left(1+\frac{2}{b}\right)\log_2 P_r+n\right]\alpha_c+\left(\frac{nb}{2}+\frac{3n^2}{2P_c}\right)\log_2 P_r\beta_c+ \\
&+\log_2 P_c\left[\frac{3n}{b}\alpha_r+\frac{1}{P_r}\left(mn-\frac{n^2}{2}\right)\beta_r\right] \qquad (6)
\end{aligned}
$$

retical peak performance of 7.6 GFLOPs/s. Each node of 8 processors has 32 Gbytes of memory. The compute nodes are connected to each other with a high-bandwidth, low-latency switching network. The peak bandwidth is 3100 MB/s and the MPI Point to Point internode latency is 4.5 usec [11]. On IBM POWER 5 we use the BLAS routines from the ESSL library (Engineering and Scientific Subroutine library). For all the runs we used the maximum number of processors available per node.

The second system is a Cray XT4 system with 9660 compute nodes. Each compute node has a 2.6 GHz dual-core AMD Opteron processor with a theoretical peak performance of 5.2 GFLOPs/s. Each compute node has 4 GBytes of memory. In our comparisons we use the routines PDGETRF and PDGETF2 from the Cray Scientific Libraries package, LibSci. However these routines have no significant optimization with respect to the routines from ScaLAPACK [12]. In our tests we use ScaLAPACK in mixed mode, that is MPI is used in between compute nodes, and threaded BLAS level parallelism on cores within a node. The threaded BLAS used is libGoto library.

6.1 Stability of ca-pivoting strategy

In this section we show that CALU is as stable as Gaussian elimination with partial pivoting. For this we summarize results that express the stability of Gaussian elimination in terms of the pivot growth and the normwise backward stability attained. We perform our tests in Matlab, using matrices from a normal distribution with varying size from 1024 to 8192. We have performed experiments on different matrices, as matrices following different random distributions, dense Toeplitz matrices, and we have obtained similar results to those presented here.

The growth factor is computed using the values of the elements of A during the elimination process. We use the growth factor as defined by Trefethen and Schreiber [14], $g_T = \frac{\max_{i,j,k}|a_{ij}^{(k)}|}{\sigma_A}$, where $a_{ij}^{(k)}$ denotes the absolute value of the element of A at row i and column j at the k-th step of elimination, and σ_A is the standard deviation of the initial element distribution. It is shown experimentally in [14] that in practice $g_T \approx n^{2/3}$ for partial pivoting, and $g_T \approx n^{1/2}$ for complete pivoting (at least for $n \leqslant 1024$).

In Figure 2 we display the value of the growth factor g_T obtained for different block sizes and different number of processors. Here two samples are used for each test. From the point of view of stability, only the number of rows in the process grid P_r plays a role. Hence we vary only P_r, presented as P in Figure 2. We observe that the growth factor of ca-pivoting grows as $c \cdot n^{2/3}$ (c being a small constant around 1.5), and has the same behavior as partial pivoting.

The new ca-pivoting strategy does not ensure that the element of maximum magnitude is used as pivot at each step of factorization. Hence $|L|$ is not bounded by 1 as in Gaussian elimination with partial pivoting. However, in practice the pivots used by ca-pivoting are very close to the elements of maximum magnitude in the respective columns. In our tests we also computed the value of the minimum threshold in CALU, where the threshold is computed at each step of factorization i as the quotient of the pivot used at step i divided by the maximum value in column i. We observed that this value is always larger than 0.33, meaning that in our tests $|L|$ is bounded by 3. The average value of the threshold is larger than 0.84. To evaluate the stability of ca-pivoting in terms of normwise backward stability, we compute three accuracy tests as performed in the HPL benchmark, and denoted as HPL1, HPL2 and HPL3. For stability, the expected values are of the order of $O(1)$, that is a slowly growing function of n. In HPL, the accuracy tests are passed if the values of the three quantities are smaller than 16.

$$
\begin{aligned}
\text{HPL1} &= ||Ax-b||_\infty/(\epsilon||A||_1 * N), \\
\text{HPL2} &= ||Ax-b||_\infty/(\epsilon||A||_1||x||_1), \\
\text{HPL3} &= ||Ax-b||_\infty/(\epsilon||A||_\infty||x||_\infty * N).
\end{aligned}
$$

We present in Table 1 the results obtained for the three tests for CALU, when varying the matrix size,

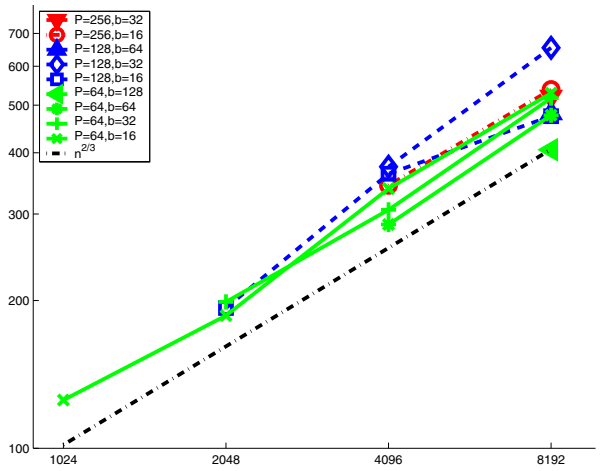

Figure 2: The growth factor for matrices following a normal distribution

the number of processors and the block size. For the matrix of size $m = n = 2^k$ in Table 1, the sample size is $S = \max\{10 * 2^{10-k}, 3\}$. We display in Table 2 the results obtained by LU factorization with partial pivoting for the same matrix sizes, where S is the sample size. All the three tests, as performed in HPL, are passed by CALU. Moreover, for all the test cases, CALU leads to results of the same order of magnitude (10^{-2}, 10^{-3}) as LU factorization with partial pivoting.

m=n	P	b	HPL1	HPL2	HPL3
2^{13}	256	32	5.06e-02	2.26e-02	4.54e-03
		16	2.24e-02	2.15e-02	4.34e-03
	128	64	4.78e-02	2.21e-02	4.22e-03
		32	3.90e-02	2.12e-02	4.30e-03
		16	6.67e-02	1.97e-02	3.89e-03
	64	128	2.09e-02	2.07e-02	3.84e-03
		64	3.45e-02	2.05e-02	4.26e-03
		32	6.64e-02	2.31e-02	4.85e-03
		16	1.32e-02	2.08e-02	4.24e-03
2^{12}	256	16	1.38e-02	1.94e-02	4.36e-03
	128	32	2.35e-02	2.22e-02	4.99e-03
		16	5.58e-01	2.11e-02	3.95e-03
	64	64	1.22e-02	2.13e-02	4.55e-03
		32	2.39e-02	2.13e-02	4.56e-03
		16	2.76e-02	2.10e-02	3.98e-03
2^{11}	128	16	3.74e-02	2.01e-02	4.36e-03
	64	32	5.52e-02	2.32e-02	5.16e-03
		16	2.83e-02	2.06e-02	4.49e-03
2^{10}	64	16	2.24e-02	2.11e-02	5.18e-03

Table 1: HPL accuracy tests for ca-pivoting strategy

6.2 Performance of TSLU

We evaluate the performance of TSLU using matrices of a size $m \times n$, a block size $b = n$, and varying

m=n	S	HPL1	HPL2	HPL3
2^{13}	5	1.41e-01	1.40e-02	2.75e-03
2^{12}	5	1.22e-02	1.40e-02	3.02e-03
2^{11}	5	1.86e-02	1.38e-02	3.01e-03
2^{10}	10	2.41e-02	1.57e-02	3.63e-03

Table 2: HPL accuracy tests for LU with partial pivoting

both m and n ($m \in \{10^3, 5 \cdot 10^3, 10^4, 10^5, 10^6\}$ and $n \in \{50, 100, 150\}$). Our goal is to study the performance improvement of TSLU compared to the ScaLAPACK PDGETF2 routine. The time ratio between PDGETF2 and TSLU obtained on the IBM POWER 5 system and the Cray XT4 system is displayed in Table 3.

The improvement is expected in part due to using a better LU factorization algorithm in the local sequential LU factorization (step 2 of Algorithm TSLU), and in part due to reducing the latency cost. In our algorithm we use the recursive LU factorization, the RGETF2 routine as given in Appendix B of [9]. Recall that TSLU performs twice the number of flops of PDGETF2. To better understand these issues, we compare two different configurations of TSLU. In the first one the local LU factorization performed by each processor on its group of rows is done using the classic LU factorization. We use the LAPACK DGETF2 routine, and the results for this configuration are displayed in the columns denoted *Cl* in Table 3. In the second one, displayed in the columns *Rec*, we use the recursive LU factorization, the RGETF2 routine as given in Appendix B of [9].

In each table we show results for fixed m and different values of n and number of processors. Several results are missing in the plots, and this is because either there was not enough memory to perform the factorization or the input matrix is too small and some processors are not involved in the operation.

On the IBM POWER 5 system, the best improvement is obtained for the largest matrix in our test set $m = 10^6$ and $n = b = 150$, where TSLU outperforms PDGETF2 by a factor of 4.37 on 16 processors. The improvement due to latency reduction is almost a factor 2. This shows that reducing the latency cost is an important part of the overall improvement.

The best performance of TSLU on the IBM POWER 5 system is 215 GFLOPs/s, and it is obtained for $m = 10^6$ and $n = 150$ on 64 processors (we count here the total number of flops performed by TSLU). This represents 44% of the theoretical peak performance. This performance corresponds to an improvement of 1.22 over PDGETF2.

For small matrices, we can notice that the improvement comes mainly from reducing the latency cost. For intermediate size matrices and a small number of processors (up to 4, 8 processors), the improvement comes

IBM POWER 5											
		No of processors $P = P_r \times P_c$									
m	$n = b$	4		8		16		32		64	
		2×2		2×4		4×4		4×8		8×8	
		Rec	Cl	Rec	Cl	Rec	Cl	Rec	Cl	Rec	Cl
10^3	50	**1.66**	1.59	1.96	**2.06**	**2.24**	2.09	-	-	-	-
10^3	100	1.27	1.17	1.44	1.37	-	-	-	-	-	-
10^3	150	1.06	0.97	-	-	-	-	-	-	-	-
$5 \cdot 10^3$	50	**1.62**	1.08	**1.48**	1.44	**1.97**	1.94	1.78	**2.05**	**2.09**	1.71
$5 \cdot 10^3$	100	0.98	0.85	1.13	1.04	1.36	1.29	1.39	1.47	-	-
$5 \cdot 10^3$	150	1.08	0.81	1.00	1.01	1.06	0.96	0.99	0.97	-	-
10^4	50	1.24	0.87	**1.71**	0.88	**1.78**	1.68	1.66	**1.94**	**2.18**	1.76
10^4	100	1.34	0.81	1.01	0.80	1.26	1.15	1.30	1.28	1.51	1.21
10^4	150	**3.07**	0.88	1.03	0.78	1.01	0.89	1.00	0.97	1.06	0.80
10^5	50	1.07	0.70	1.09	0.72	1.18	0.85	1.15	**1.32**	**1.50**	1.23
10^5	100	1.00	0.70	1.04	0.67	1.09	0.73	1.21	1.03	1.19	1.01
10^5	150	**1.13**	0.68	**1.13**	0.69	**1.36**	0.77	1.08	0.84	1.03	0.75
10^6	50	1.36	0.71	1.27	0.71	1.25	0.70	1.12	0.69	**2.01**	0.82
10^6	100	1.84	0.75	1.95	0.87	1.62	0.73	2.90	0.84	1.08	0.70
10^6	150	**2.32**	0.81	**2.34**	0.89	**4.37**	0.90	**3.42**	0.85	1.22	0.70
Cray XT4											
		No of processors $P = P_r \times P_c$									
m	$n = b$	4		8		16		32		64	
		2×2		2×4		4×4		4×8		8×8	
		Rec	Cl	Rec	Cl	Rec	Cl	Rec	Cl	Rec	Cl
10^3	50	1.42	**2.23**	1.85	**2.71**	2.09	**3.09**	-	-	-	-
10^3	100	1.14	1.39	1.29	1.56	-	-	-	-	-	-
10^3	150	1.12	0.91	-	-	-	-	-	-	-	-
$5 \cdot 10^3$	50	1.22	1.42	1.65	**2.15**	1.97	**2.72**	2.10	**3.06**	1.04	**2.59**
$5 \cdot 10^3$	100	1.27	1.24	1.25	1.32	1.35	1.53	1.38	1.65	-	-
$5 \cdot 10^3$	150	**1.67**	1.22	0.97	0.90	0.88	0.91	0.85	0.90	-	-
10^4	50	1.20	1.14	1.37	1.19	1.85	**2.42**	1.03	**2.88**	2.03	**3.10**
10^4	100	2.19	1.34	1.56	1.44	1.30	1.41	0.94	1.56	1.36	1.94
10^4	150	**2.61**	1.30	**2.03**	1.44	0.92	0.88	0.81	0.90	0.87	0.93
10^5	50	2.12	1.13	2.23	1.37	2.29	1.50	1.20	1.47	1.76	1.88
10^5	100	3.14	1.25	3.13	1.45	2.97	1.43	1.92	1.39	**2.38**	1.39
10^5	150	**3.78**	1.30	**3.57**	1.47	**3.14**	1.30	**2.12**	1.26	2.34	1.15
10^6	50	2.99	1.49	3.02	1.51	2.86	1.28	2.09	1.03	2.14	1.31
10^6	100	4.51	1.65	4.55	1.71	4.04	1.36	3.04	1.13	3.07	1.27
10^6	150	**5.58**	1.61	**5.52**	1.76	**4.80**	1.39	**3.60**	1.12	**3.67**	1.20

Table 3: Time ratio of PDGETF2 to TSLU obtained on IBM POWER 5 and Cray XT4 systems, using DGETF2 for the local LU factorization (Cl), and using RGETF2 for the local LU factorization (Rec). The matrix factorized is $m \times n$, with a block of size $b = n$.

mainly from using recursion. For the same matrices and a large number of processors, the improvement comes from decreasing the latency cost. On small number of processors the recursion leads to important improvements for large matrices ($m = 10^5, 10^6$ and varying n), for instance a factor of 2.3 for $m = 10^6$ and $n = 150$ on 4 processors. However, with increasing number of processors, even for large matrices, reducing the latency plays an important role in the overall improvement. The best results are obtained on 16 and 32 processors for the biggest matrices $m = 10^6$ and $n = 150$, showing the overall improvement factors of 4.37 and 3.42 respectively.

On the Cray XT4 system, the best improvement is seen for $m = 10^6$ and $n = 150$: a factor of 5.58 on 4 processors and a factor of 5.52 on 8 processors. The best performance of the TSLU algorithm is 240 GFLOPs/s, obtained by TSLU on 64 processors for $m = 10^6$ and $n = 150$. This represents 36% of the theoretical peak performance and it corresponds to an improvement of 3.67 over PDGETF2.

For the small matrices ($m = 10^3$ and n varying from 50 to 150 or $m = 5 \cdot 10^3$ and $n = 50$ or 100) the best results are obtained using classic LU, hence solely due to reduction of the latency cost. For all the large matrices, using recursive LU shows a better performance.

In summary, for all the cases tested, at least one of the new TSLU algorithms outperforms the ScaLAPACK routine PDGETF2. For small matrices the usage of classic LU leads to better performance than the recursive LU. This is in accordance with the results in [9, 13] which show that the recursive algorithm do not fare better than the classic algorithm for small matrices. For larger matrices, recursive LU performs better than classic LU. The improvements obtained by TSLU are due to both reducing the latency and the local LU factorization costs.

6.3 Performance of CALU

In this section we study the performance improvement of CALU compared to the ScaLAPACK PDGETRF routine. We use matrices of a size $m \times m$, and varying both m and b ($m \in \{10^3, 5 \cdot 10^3, 10^4\}$ and $b \in \{50, 100, 150\}$). The time ratio between PDGETRF and CALU obtained on the IBM POWER 5 system and the Cray XT4 system are presented in Table 4.

We have observed in Table 3 that for a small number of processors, the best performance for TSLU is obtained when recursive LU is used for the local LU factorization. The number of processors used for TSLU corresponds to the number of rows P_r in the 2D grid of processors used for CALU. Since in our tests for CALU P_r is relatively small (with values going from 2 to 8), in all our tests we use for the panel factorization TSLU with recursive LU.

For IBM POWER 5 system, the first matrix ($m = 10^3$ and varying b) is relatively small, and thus we do not expect any important speedup with increasing number of processors. In fact, for $m = 10^3$ we see no speedup for a number of processors larger than 16. Still, this matrix is helpful in showing the improvement due to the reduction of the latency cost. The best improvement is obtained for $m = 10^3$ and $b = 50$ (factors of 2.23 on 16 processors and 2.29 respectively on 64 processors), mainly as a result of reducing the latency cost. An important improvement is obtained also for $m = 5 \cdot 10^3$, a factor of 1.67 on 32 processors and a factor of 1.69 on 64 processors. For $m = 10^4$, the best improvement is a factor of 1.59 on 32 processors.

For Cray XT4 system, we notice that the improvements are smaller than on the IBM POWER 5 system. The best improvement obtained is a factor of 1.81 for $m = 10^3$ and $b = 100$ on 64 processors. For $m = 5 \cdot 10^3$, the best improvements obtained are a factor of 1.38 on 8 processors and a factor of 1.36 on 64 processors, both for $b = 150$. For $m = 10^4$, the best improvements are a factor of 1.38 on 32 processors and a factor of 1.33 on 64 processors.

The improvements presented in Table 4 are obtained for a fixed number of processors and a fixed block size. However the best improvements do not correspond always to the best performance of CALU or PDGETRF. CALU can have a better performance for a different block size or grid shape than PDGETRF. Hence, an interesting question to answer is: for a given problem size m and a given maximum number of processors, what is the improvement obtained by the best CALU with respect to the best PDGETRF? To answer this question, we present in Table 5 the improvement obtained by taking the best performance independently for CALU and PDGETRF, when varying the number of processors (from 8 to 64) and the block size (values of 50, 100 and 150). For a given number of processors, we use one grid shape, as in our previous experiments. We also display the best performance for CALU and PDGETRF in GFLOPs/s (GFlops columns), the block size (b) and the number of processors for which the best performance was obtained, and the percentage of theoretical peak performance obtained by CALU (columns Prcnt). The speedup is computed as follows:

$$speedup(m, m, P_{max}) = \frac{\min_{P \leq P_{max}, b} T_{PDGETRF}(m, m, P, b)}{\min_{P \leq P_{max}, b} T_{CALU}(m, m, P, b)}$$

CALU leads to improvements up to 1.69 on IBM POWER 5 and up to 1.53 on Cray XT4. Note that the improvements are obtained when the performance of the algorithm is a small percentage of the theoretical peak performance. The smallest percentage is obtained on Cray XT4, for $m = 10^3$ and $P = 32$. This

IBM POWER 5											
		No of processors $P = P_r \times P_c$									
$m = n$	b	4		8		16		32		64	
		2×2		2×4		4×4		4×8		8×8	
		Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU
10^3	50	**1.57**	9.79	**1.59**	11.9	**2.23**	11.8	**2.07**	11.5	2.25	9.8
10^3	100	1.48	8.84	1.47	10.5	1.91	10.6	1.91	11.2	**2.29**	10.9
10^3	150	1.36	8.26	1.41	9.9	1.70	9.5	-	-	-	-
$5 \cdot 10^3$	50	1.05	21.5	1.09	39.4	**1.31**	61.2	1.51	95.5	**1.69**	118.6
$5 \cdot 10^3$	100	**1.06**	21.1	**1.13**	37.3	1.21	56.7	**1.67**	84.1	1.66	103.1
$5 \cdot 10^3$	150	1.04	20.6	1.08	35.1	1.18	52.3	1.26	74.8	1.45	89.1
10^4	50	1.00	23.27	1.00	45.1	1.08	80.3	1.17	143.2	**1.35**	213.9
10^4	100	1.00	23.62	1.00	44.4	1.10	78.0	1.19	133.2	1.24	197.6
10^4	150	**1.01**	23.47	**1.02**	42.3	**1.33**	74.1	**1.59**	122.1	1.17	173.8
CRAY XT4											
		No of processors $P = P_r \times P_c$									
$m = n$	b	4		8		16		32		64	
		2×2		2×4		4×4		4×8		8×8	
		Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU	Impvt	GFlops CALU
10^3	50	1.19	5.4	1.20	6.6	1.33	6.6	1.35	7.5	1.67	7.6
10^3	100	**1.28**	5.5	**1.39**	7.0	**1.52**	7.2	**1.60**	8.5	**1.81**	8.3
10^3	150	1.23	5.3	1.32	6.6	1.44	6.7	-	-	-	-
$5 \cdot 10^3$	50	1.03	19.2	1.09	33.3	1.12	44.3	1.16	69.2	1.11	67.2
$5 \cdot 10^3$	100	1.12	19.4	1.20	32.3	1.13	42.8	1.24	67.4	1.32	76.1
$5 \cdot 10^3$	150	**1.23**	19.1	**1.38**	31.0	**1.22**	40.8	**1.35**	61.9	**1.36**	70.5
10^4	50	1.01	24.4	1.05	45.7	1.04	69.5	1.08	121.3	1.31	154.9
10^4	100	1.09	25.3	1.18	46.5	1.13	69.8	1.22	118.2	**1.33**	153.3
10^4	150	**1.16**	25.2	**1.31**	45.4	**1.22**	67.3	**1.38**	111.4	1.30	140.3

Table 4: Time ratio of PDGETRF to CALU (Impvt columns) and performance for CALU in GFLOPs/s (GFlops columns) obtained on IBM POWER 5 system. The matrix factorized is $m \times m$, with a block of size b.

IBM Power 5								
m	speedup	CALU				PDGETRF		
		GFlops	P	b	Prcnt	GFlops	P	b
10^3	1.59	11.9	8	50	19.6	7.5	8	50
$5 \cdot 10^3$	1.69	118.6	64	50	24.4	70.0	64	50
10^4	1.34	213.9	64	50	40.6	159.8	64	100
Cray XT4								
m	speedup	CALU				PDGETRF		
		GFlops	P	b	Prcnt	GFlops	P	b
10^3	1.53	8.5	32	100	2.5	5.54	32	50
$5 \cdot 10^3$	1.26	76.1	64	100	11.4	60.2	64	50
10^4	1.31	154.9	64	50	23.2	118.1	64	50

Table 5: Speedup estimated as the ratio of best PDGETRF over best CALU for a given problem size, the best performance for CALU and PDGETRF in GFLOPs/s (GFlops columns), the block size (b columns) and the number of processors (P columns) for which the best performance was obtained. *Prcnt* denotes the percentage of theoretical peak performance obtained by CALU.

is somehow expected, since this is a small matrix executed on 32 dual-core processors. A better percentage is obtained on IBM POWER 5, for example 40.6 for $m = 10^4$ on 64 processors. However, even when the percentage of peak performance is small, the Table 5 shows that CALU will let a user more efficiently use the same resources than PDGETRF, and so it is always worth using it.

7 Conclusions and future work

In this paper we have introduced CALU, a new algorithm for computing the LU factorization of dense matrices. This algorithm uses a new pivoting strategy, the ca-pivoting, which is used to efficiently compute the LU factorization of a block-column, and leads to an important decrease in the number of messages of CALU with respect to classic algorithms.

We have compared CALU with the corresponding PDGETRF routine from ScaLAPACK. Our experiments have shown that depending on the size of the matrix and the characteristics of the underlying computer architecture, it is either latency reduction or recursion or both which are major factors in reducing the parallel time of the LU factorization. Interestingly, the gains due to latency reduction are not limited only to the small matrices, but affects also the large matrices. In these cases the recursion is very efficient in reducing the local LU factorization time and thus leaves the latency as the time consuming bottleneck, which needs to be alleviated. The factorization of a block-column, TSLU, outperforms the corresponding routine PDGETF2 from ScaLAPACK up to a factor of 4.37 on the IBM POWER5 system and up to a factor of 5.58 on the Cray XT4 system. CALU outperforms PDGETRF up to a factor of 2.29 on IBM POWER5 and up to a factor of 1.81 on Cray XT4.

The factorization of a block-column lies on the critical path of the parallel LU factorization, and hence we expect that the usage of ca-pivoting strategy by other parallel LU algorithms, as HPL, will lead to improvements of the overall time. This is the object of our current research.

As future work, it will be interesting to study the suitability of the new ca-pivoting strategy for parallel LU on multicore architectures. Another direction consists of using the ca-pivoting strategy for the LU factorization or the incomplete LU factorization of sparse matrices. This may pay off much more than the dense case, since there is a higher proportion of communication versus computation.

Acknowledgments

The authors thank M. Hoemmen, J. Langou, and J. Riedy for helpful discussions on this subject.

References

[1] Top 500 supercomputer sites. available at www.top500.org.

[2] 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.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600, University of Tennessee, 2007. LAPACK Working Note 191.

[4] 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.

[5] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. Technical Report UCB/EECS-2008-89, UC Berkeley, 2008. LAPACK Working Note 204.

[6] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency: Practice and Experience*, 15:803–820, 2003.

[7] T. Endo and K. Taura. Highly Latency Tolerant Gaussian Elimination. *Proceedings of 6th IEEE/ACM International Workshop on Grid Computing*, pages 91–98, 2005.

[8] S. L. Graham, M. Snir, and C. A. Patterson, editors. *Getting Up To Speed: The Future Of Supercomputing*. National Academies Press, Washington, D.C., USA, 2005.

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

[10] G. Quintana-Orti, E. S. Quintana-Orti, E. Chan, F. G. Van Zee, and R. van de Geijn. Programming algorithms-by-blocks for matrix computations on multithreaded architectures. Technical Report TR-08-04, University of Texas at Austin, 2008. FLAME Working Note 29.

[11] D. Skinner. IBM SP Parallel Scaling Overview. http://www.nersc.gov/news/reports/technical/seaborg_scaling.

[12] A. Tate. Personal communication.

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

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

[15] V. Volkov and J. W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. *Proceedings of the ACM/IEEE SC08 Conference*, 2008.