LU Factorization: Analysis of the Broadcast-Based Parallel Algorithm for Clusters

Fernando G. Tinetti[*]
Investigador Asistente CICPBA, III-LIDI
Facultad de Informática, UNLP
50 y 115, 1900, La Plata
Argentina

Technical Report PLA-001-2005[†]
March 2005

Abstract

This report is specifically focused on the analysis of the broadcast-based parallel LU matrix factorization in order to have a prediction performance measure for the algorithm. Also, this prediction performance measure can be used for comparison with other parallel algorithms proposed for the same task -LU matrix factorization. Some experiments are presented to verify the accuracy of the performance prediction as well as to compare with the ScaLAPACK performance of the parallel LU matrix factorization. This report should be understood in the context of two previous technical reports: "LU Factorization: Number of Floating Point Operations and Parallel Processing in Clusters" and "Guidelines for Parallel Linear Algebra on Ethernet-Based Clusters: Matrix Multiplication and LU Factorization Results".

[*]fernando@info.unlp.edu.ar
[†]PLA stands for Parallel Linear Algebra

1 Introduction

The LU factorization algorithm as well as the number of operations required has been defined and/or analyzed in [13]. The number of floating point operations of the *traditional* method, $LU flops$, is exactly given by

$$LU flops = 2\left(\frac{(n-1)^3}{3} + \frac{(n-1)^2}{2} + \frac{(n-1)}{6}\right) + \frac{n^2-n}{2} \tag{1}$$

which is usually referred to as $2n^3/3$ [9]. Sequential as well as parallel approaches are based in the block-processing right-looking algorithm [9] [2] [8], defined in terms of a partition such as that shown in Fig. 1, where bs is the block size selected to optimize the sequential and/or parallel performance. From

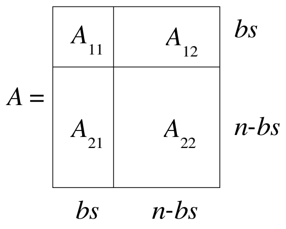

Figure 1: Block Partition of a Matrix.

a matrix partition such as that shown in Fig. 1, computing is made by LU factorizing A_{11} and updating the *trailing matrix*, i.e. A_{12}, A_{21}, and A_{22}. In terms of the partition shown in Fig. 1, the LU factorization to be computed is such that

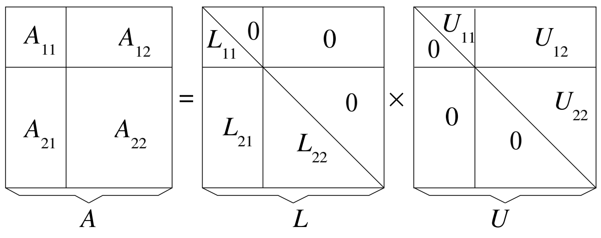

Figure 2: Partition of Matrices A, L, and U.

i.e.,

$$A_{11} = L_{11} \times U_{11} \tag{2}$$
$$A_{12} = L_{11} \times U_{12} \tag{3}$$
$$A_{21} = L_{21} \times U_{11} \tag{4}$$
$$A_{22} = L_{21} \times U_{12} + L_{22} \times U_{22} \tag{5}$$

And, as explained above, the A_{11} block is factorized and the rest of the blocks are updated in order to compute matrices L and U. Having factorized the A_{11} block implies having L_{11} and U_{11}, i.e., the whole Eq. (2) above is completely computed. It is necessary to obtain L_{12}, U_{12}, L_{22}, and U_{22}. To obtain U_{12} and L_{21} Ec. (6) and Ec. (7) are applied respectively, which are derived from Ec. (3) and Ec. (4) above respectively:

$$U_{12} = L_{11}^{-1} \times A_{12} \tag{6}$$
$$L_{21} = A_{21} \times U_{11}^{-1} \tag{7}$$

It is important to note that L_{11}^{-1} and U_{11}^{-1} are very easy to calculate from L_{11} and U_{11} since these matrices are triangular (lower and upper tiangular respectively). In terms of BLAS (Basic Linear Algebra Subroutines) [7] [11], U_{12} and L_{21} are obtained by using *directly* the functions to solve triangular systems of equations with multiple right hand sides, TRSM. The remaining problem is to find out L_{22} and U_{22}, and for this task, Ec. (5) is used according to Ec. (8)

$$L_{22} \times U_{22} = A_{22} - L_{21} \times U_{12} \tag{8}$$

Thus, finding out L_{22} and U_{22} from A_{22} is the same as finding out L_{22} and U_{22} from $A_{22} - L_{21} \times U_{12}$, i.e. A_{22} is *updated* to have $A_{22} - L_{21} \times U_{12}$ and the same blocking procedure can be directly applied to the *updated* submatrix A_{22}. The blocking procedure is intensively and successfully used in sequential as well as parallel approaches.

The broadcast-based parallel LU matrix factorization has been defined in [14] using the ideas presented in [15] [12] [13]. The matrix partitioning is chosen following the so-called row block cyclic matrix distribution [10]; the matrix is divided in many more blocks than computers and assigned cyclically as shown in Fig. 3 for four processors $P_1, \ldots, P_4$, where a block size (bs) has to be defined. Each row block has bs rows, i.e., $bs \times n$ elements. This matrix distribution among computers can be defined in general

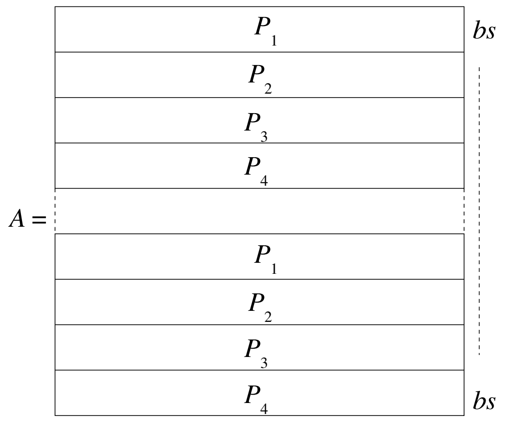

Figure 3: Row Block Cyclic Partitioning.

as follows. If a matrix of order n is divided into row blocks of size bs, there will be $nb = n/bs$ row blocks $rb_1, ..., rb_{nb}$ to be assigned among the available computers. In general, processor P_i will have blocks rb_j such that $i = (j-1) \ mod \ p + 1$, $1 \leq j \leq nb$, where p is the number of computers $(P_1, \ldots, P_p)$. In this context, when factorizing a matrix block of $bs \times n$ elements, three of the submatrices of Fig. 2 are found: L_{11}, U_{11}, and U_{12} as shown in Fig. 4. The three other submatrices (L_{21}, L_{22}, and U_{22}) are found

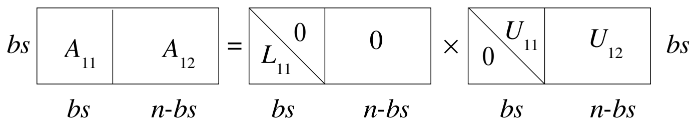

Figure 4: LU on a Row Block.

following the same guidelines as in the case of the two-dimensional matrix distribution. Matrix L_{21} is found by computing $L_{21} = A_{21} \times U_{11}^{-1}$, Eq. (7), using BLAS TRSM, and matrices L_{22} and U_{22} are obtained applying the same procedure to the updated matrix $A_{22} - L_{21} \times U_{12}$. Fig. 5 shows the pseudocode of the process for computer P_i with which the whole broadcast-based parallel LU factorization algorithm can be defined, where

- `p` is the total number of computers used in parallel for the LU factorization, identified from 0 to `p-1`.
- `nb` is the total number of blocks, which depends on the blocking size bs. Blocks are *globally numbered* across computers from 0 to `nb`, i.e., computer P_i ($0 \leq i \leq$ `p-1`) has blocks $i \ mod$ `p`.
- On the line `Factorize block j` matrices L and U are obtained from block j with partial pivoting, which is used for numerical stability.

```
if (i == 0)
  Factorize and send_broadcast block 0
for (j = 0; j < nb; j++)
{
   if (i == (j mod p))  /* Current block is local */
      Update local blocks
   else if (i == ((j+1) mod p))  /* Next block is local */
   {
      recv_broadcast_b (factorized block j)
      Update and Factorize block j+1
      send_broadcast_b (factorized block j+1)
      Update local blocks (block j+1 already updated)
   }
   else /* P_i does not hold block j nor block j+1 */
   {
      recv_broadcast_b (factorized block j)
      Update local blocks (block j+1 already updated)
   }
}
```

Figure 5: Parallel LU Matrix Factorization Algorithm with Overlapped Communications.

- The `trailing matrix` is, in fact, "the" A_{22} defined above, which is updated via a matrix multiplication.
- `send_broadcast_b` and `recv_broadcast_b` are the functions to send and receive broadcast messages in background respectively.

The algorithm defined in Fig. 5 contains the idea of *next block* to the classical algorithm. Given that the LU factorization and its corresponding broadcast message impose waiting (possible for a long time in Ethernet-based clusters) on $p-1$ computers, the next block is factorized and sent ahead (in background) allowing overlapped communication with local computing. Basically, the broadcast operation is intended to complete while every computer is updating the trailing matrix.

2 Performance Analysis of the Parallel Algorithm

The idealized case in which computing and communication are overlapped and there is no overhead due to broadcast messages (except for the communication of the first block) is shown in Fig. 6. Most of the

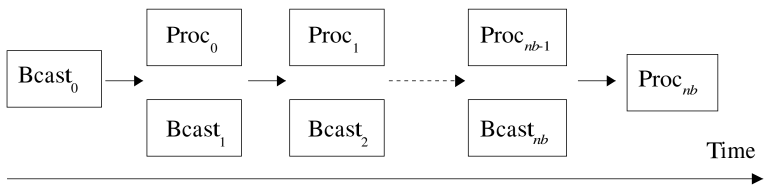

Figure 6: Overlapped Computing and Communication.

local computing time in each computer (shown as Proc$_i$ on Fig. 6) is mainly due to the trailing matrix update which includes a matrix multiplication. The following performance analysis is made taking into account:

1. The distribution of the trailing matrix update.
2. That Communication is carried out concurrently with computing (*in background*).

From the point of view of the program running on each computer, whose pseudocode is shown in Fig. 5, on each iteration j, computers execute one of the three alternatives defined in the pseudocode, as shown in Fig. 7. On iteration j, processor holding the current block, $P_{j \, mod \, p}$, just updates the rest of the local blocks of the trailing matrix, i.e. rb_k with $k > j$. This is shown in Fig. 7 with the sequence

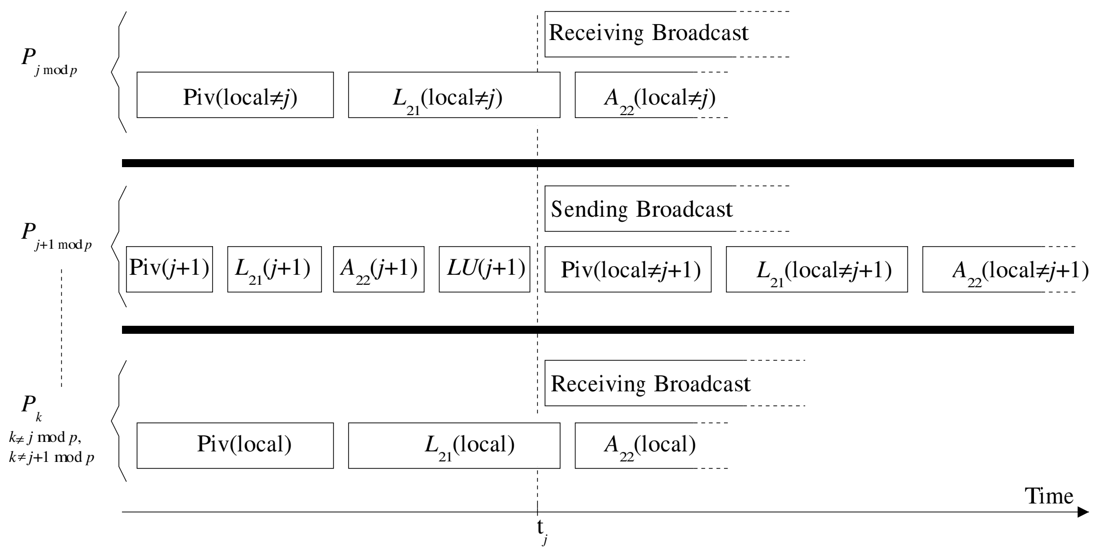

Figure 7: Computing on Each Processor.

- Piv(local $\neq$ j): apply pivots on every local block excluding the current block,
- L_{21}(local $\neq j$): compute L_{21} on every local block excluding the current block,
- A_{22}(local $\neq j$): compute/update A_{22} on every local block excluding the current block,

because the current block has been updated/computed in the previous iteration. Also, computer $P_{j\ mod\ p}$ receives in background the next current block which will be necessary in the next iteration: rb_{j+1}. The processor holding the next current block on iteration j, $P_{j+1\ mod\ p}$, has to update and LU factorize the next current block. This is shown in Fig. 7 with the sequence:

- Piv(j+1): apply pivots only to block $j+1$.
- $L_{21}(j+1)$: compute the submatrix of L_{21} corresponding to block $j+1$.
- $A_{22}(j+1)$: update the submatrix of A_{22} corresponding to block $j+1$.
- $LU(j+1)$: LU factorize the block $j+1$.

Once the block $j+1$ is factorized it is sent in background to all the other computers (from time t_j via a broadcast routine). The rest of the processing in processor $P_{j+1\ mod\ p}$ is straightforward. Most of the computers, P_k with $k \neq (j\ mod\ p)$ and $k \neq (j+1\ mod\ p)$, have simple computing and communication tasks: update local blocks and receive in background the next current block once the broadcast sending begins in processor $P_{j+1\ mod\ p}$.

The detailed view of processing given in Fig. 7 is accurate but may lead to confussion on relative requirements of each individual task. Fig. 8 shows schematically that most of the time needed for each iteration is due to communication and A_{22} update. There are two issues related to relative times shown in Fig. 8:

1. The reason/s for which A_{22} update requirements are greater than those of applying pivots plus L_{21} computing and also greater than updating and LU factorizing a whole block.
2. The relationship between the time required by a broadcast communication and the time required by an A_{22} update, more specifically, which one of both tasks requires more time than the other. This relationship between the relative times is necessary to be able to predict the time of an iteration.

Once each iteration time can be predicted, the time prediction for the whole algorithm is straightforward.

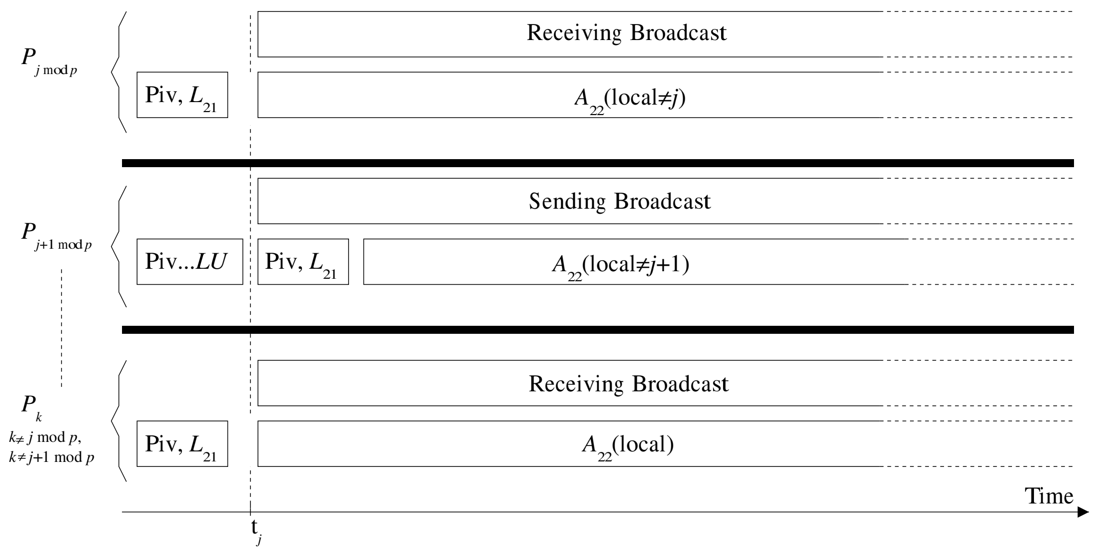

Figure 8: Relative Times of Computing on Each Processor.

2.1 Brief Performance Analysis of Computing Subtasks

The time required by subtasks of the algorithm should be detailed at least at the level of having a comparison in orders of magnitude. More specifically, the processing requirements of each task shown in Fig. 7 should be analyzed, i.e.: Piv(local $\neq j$), L_{21}(local $\neq j$), A_{22}(local $\neq j$), Piv(j+1), $L_{21}(j+1)$, $A_{22}(j+1)$, $LU(j+1)$, Piv(local), L_{21}(local), and A_{22}(local). Before analyzing every subtask in detail, basically: Piv, L_{21}, A_{22}, and LU computing, it is useful to show in more detail how the algorithm of Fig. 5 works on the original matrix to be factorized.

Taking into account the pseudocode of Fig. 5 and the matrix partition into row blocks shown in Fig. 3, a matrix of $n \times n$ elements is processed at the beginning of the algorithm as shown in Fig. 9, where the first block of $bs \times n$ elements is completely contained in processor P_0. Before the first iteration the first

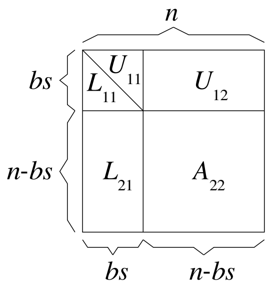

Figure 9: Processing Before the First Iteration.

block of $bs \times n$ elements is processed as a matrix and LU factization is computed, obtaining submatrices L_{11}, U_{11}, and U_{12}, and the associated factorization pivots. Given that the matrix is partitioned into row blocks, the first processor P_0 computes locally (without needing extra data/communication) submatrices L_{11}, U_{11}, and U_{12}, and associated pivots. From the point of view of libraries such as LAPACK [2] or ATLAS (Automatically Tuned Linear Algebra Software) [16] this is done via a single call to `GETRF`. On the first iteration, i.e. $i = 1$, with $i = j + 1$, where j is the pseudo-code iteration index, several tasks have to be done:

- Computing matrix L_{21}.
- Updating matrix A_{22}.
- Computing and sending broadcast the LU factorization on the next current block.

On this first iteration, the matrix is processed as shown in Fig. 10, where

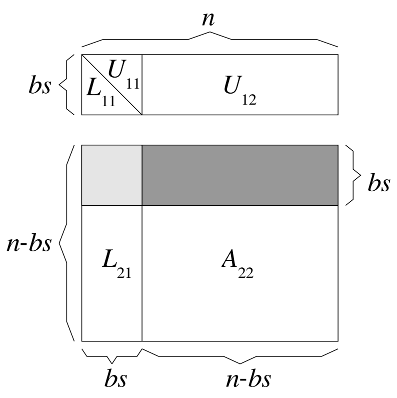

Figure 10: Processing on the First Iteration.

- The shaded submatrix of $bs \times n$ elements is to be computed at processor P_1.
- The dark submatrix of $bs \times n - bs$ elements is the next current block to be sent in background while computing/updating matrices L_{21} and A_{22} in all computers, including P_1.

More specifically, let tL_1 the submatrix of L_{21} to be computed on the first iteration, tU_1 the matrix U_{12} to be computed before the first iteration, and tA_1 the submatrix of A_{22} to be computed on the first iteration,

$$\begin{aligned} tL_1 &\in \mathbb{R}^{(n-bs)\times bs} \\ tU_1 &\in \mathbb{R}^{bs\times(n-bs)} \\ tA_1 &\in \mathbb{R}^{(n-bs)\times(n-bs)} \end{aligned}$$

And computing on the first iteration involves

- Applying pivots to the whole trailing matrix, including tL_1 and tA_1
- Computing $tL_1 = A_{21} \times U_{11}^{-1} = tL_1 \times U_{11}^{-1}$.
- Computing $tA_1 = A_{22} - tL_1 \times tU_1 = tA_1 - tL_1 \times tU_1$.

However, the next current block (in the first bs rows of tA_1) is processed in a different way, in order to be LU factorized and available on every computer in the next iteration of the algorithm. The dark submatrix of tA_1 is updated and LU factorized in P_1 before updating the rest of tA_1 assigned to processor P_1.

In general, in the *i_th* iteration, the processing can be schematically described as shown in Fig. 11. And defining tL_i, tU_i, and tA_i as matrices tL_1, tU_1, and tA_1 but on *i_th* iteration, i.e. tL_i the submatrix

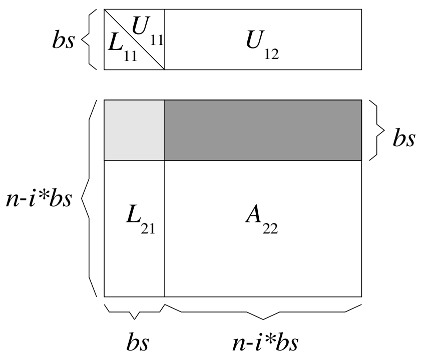

Figure 11: Processing on the *i_th* Iteration.

of L_{21} to be computed on the i_th iteration, tU_i the matrix U_{12} computed before the i_th iteration, and tA_i the submatrix of A_{22} to be computed on the i_th iteration,

$$\begin{array}{rcl} tL_i & \in & \mathbb{R}^{(n-i*bs)\times bs} \\ tU_i & \in & \mathbb{R}^{bs\times(n-i*bs)} \\ tA_i & \in & \mathbb{R}^{(n-i*bs)\times(n-i*bs)} \end{array}$$

And computing on the i_th iteration involves

- Applying pivots to the whole trailing matrix, including tL_i and tA_i
- Computing $tL_i = A_{21} \times U_{11}^{-1} = tL_i \times U_{11}^{-1}$.
- Computing $tA_i = A_{22} - tL_i \times tU_i = tA_i - tL_i \times tU_i$.

And the next current block (in the first bs rows of tA_i) is processed in a different way, in order to be LU factorized and available on every computer in the next iteration of the algorithm.

2.1.1 Applying Pivots on the i_th Iteration

There are several alternatives in order to apply pivots and maintain data consistency as well as numerical stability. The minimum task to be done is trailing matrix update/pivoting with the pivots of the current LU factorized block. This implies updating tL_i and tA_i. However, it is worth noting that:

- pivoting do not imply any floating point operation, just data movement.
- there are at most bs column interchanges, since pivots are produced/necessary on LU factorization of a matrix with $bs \times (n - i * bs)$ elements on i_th iteration.

and these are the reasons for which pivoting is not taken into account from the point of view of computing time analysis.

2.1.2 Computing tL_i

Computing tL_i implies

$$tL_i = A_{21} \times U_{11}^{-1} = tL_i \times U_{11}^{-1}$$

with

$$\begin{array}{rcl} tL_i & \in & \mathbb{R}^{(n-i*bs)\times bs} \\ U_{11} & \in & \mathbb{R}^{bs\times bs} \end{array}$$

Thus, a triangular system with $(n-i*bs)$ equations and bs unknowns is solved bs times to obtain/update matrix tL_i. In terms of the BLAS library [7] [11], tL_i is obtained by using *directly* the function to solve triangular systems of equations with multiple right hand sides, `TRSM`. Roughly speaking, solving a triangular system of equations with the coefficient matrix of $bs \times bs$ elements implies $O(bs^2)$ operations, and given that the triangular system of equations is solved with $n - i * bs$ right hand sides, the number of floating point operations is $O((n - i * bs) * bs^2)$. Thus, in general, computing tL_i requires $O(n)$ operations, and taking into account that bs is relatively small compared to n the hidden constant/s in the $O()$ notation cannot be very large. Sumarizing,

$$ReqFlops(tL_i) = O(n) \tag{9}$$

where $ReqFlops(tL_i)$ is the number of floating point operations required to compute tL_i and, in fact, is a measure of the computing time needed for tL_i.

2.1.3 Computing LU on the i_th Iteration: LU_i

The LU factorization method on the i_th iteration is applied to a submatrix of $bs \times (n - i * bs)$ elements. Following the description made in [13] to calculate the number of flops for LU factorization, the number of floating points required for the factorization of a $bs \times (n - i * bs)$ elements matrix is given by

$$\begin{aligned} ReqFlops(LU_i) &= \sum_{i=1}^{bs} 2 \times (bs - i) \times (n - i) \\ &= \sum_{i=1}^{bs} 2(bs * n - bs * i - i * n + i^2) \\ &= \sum_{i=1}^{bs} 2 * bs * n - 2 * (bs + n) * i + 2 * i^2 \end{aligned}$$

which is $O(n)$ but with more terms than $ReqFlops(LU_i)$, and some of them being bs^2 and bs^3. However, it is still remarkable that

$$ReqFlops(LU_i) = O(n) \tag{10}$$

Furthermore, analyzing the evolution of computing as i increases (more iterations are made), the number of flops decreases, so the number of flops is very small in the last iterations taking into account the $(bs-i)$ and $(n - i)$ factors in the equations above.

2.1.4 Updating tA_i

Taking into account Fig. 8, this is the task with the greatest floating point requirements on each iteration. Furthermore, the whole blocking method has been designed to have most of the operations in a matrix multiplication and this is made in the updating of tA_i. in order to update the trailing matrix, a matrix multiplication should be done. More specifically, the operation to be carried out is

$$tA_i = tA_i - tL_i \times tU_i$$

where

$$\begin{aligned} tA_i &\in \mathbb{R}^{(n-i*bs)\times(n-i*bs)} \\ tL_i &\in \mathbb{R}^{(n-i*bs)\times bs} \\ tU_i &\in \mathbb{R}^{bs\times(n-i*bs)} \end{aligned}$$

Each element of the matrix multiplication $tL_i \times tU_i$ requires $2*bs-1$ operations, and the resulting matrix is of $(n - i * bs) \times (n - i * bs)$ elements, thus

$$ReqFlops(tA_i) = (2 * bs - 1) * (n - i * bs) * (n - i * bs) + (n - i * bs) * (n - i * bs) \tag{11}$$

Summarizing,

$$ReqFlops(tA_i) = O(n^2) \tag{12}$$

2.1.5 Comparison of Subtasks Requirements

Summarizing, applying pivots requires a negligible running time, and from the previous subsections:

$$\begin{aligned} ReqFlops(tL_i) &= O(n) \\ ReqFlops(LU_i) &= O(n) \\ ReqFlops(tA_i) &= O(n^2) \end{aligned}$$

and these equations justify a quite more formally the running times shown in Fig. 8. It is worth noting that even though not every block of tA_i is updated *at the same time* or *concurrently*, the required number of flops is $O(n^2)$ because there is just one block updated and LU factorized ahead the rest: the *next current block*.

2.2 Brief Performance Analysis of Broadcast

Unfortunately, modeling the time required for a broadcast operation is not as simple as using the well-known model for a point-to-point communication:

$$t(m) = \alpha_{ptp} + \beta_{ptp} * m \tag{13}$$

where m is the amount of data to transfer, α is the communication latency or startup cost, and $1/\beta_{ptp}$ is the network communication bandwidth. Timing models for broadcast communication usually depend on the implementation selected for the broadcast routine in a specific implementation. Many implementations construct spanning trees and this implies logarithmic coefficients on the timing models. In the specific case of the research being made for parallel computing in clusters, the broadcast message implementation is such that:

- Data is physically broadcasted using UDP, and there is not retransmission unless some data have been lost.
- Acknowledgements are received at the broadcast root from all the receivers to provide a reliable broadcast message to the parallel application.

These details are hidden to the user (pertain to the broadcast implementation). Physical broadcast plus the low rate at which data are lost in a local area network produce a very good scalability. However, latency cannot be modeled as in a point-to-point message, since the time required for broadcast acknowledgement/s and other tasks related to synchronization are found to be proportional to the number of computers involved in the communication. Given that data is sent as in a point-to-point operation and there is a very low rate of message loss, the time required for data transmission through the network can be modeled as in the point-to-point messages, i.e. $(\beta_{bcast} * m)$ in Eq. (13), with $\beta_{bcast} \cong \beta_{ptp}$ because the data bandwidth of the broadcast routine is not necessarily the same as that of point-to-point messages, and $m = bs * (n - i * bs)$ in the *i_th* iteration. More specifically, one block has to be broadcasted on each iteration, whose size is $m = bs * (n - i * bs)$. However, ackowledgements sent from receivers to the root attempt against scalability, because these messages cannot be received simultaneously at the broadcast root. Even when there are multiple ways of avoiding such performance drawback, it is still possible to analyze the time required by broadcast messages (and the parallel program using this broadcast *version*) from the point of view of the time required for communication. Summarizing, the timing model for the broadcast operation in the *i_th* iteration is

$$t(bcast(i,p)) = \alpha_b + lpp * (p - 1) + \beta_{bcast} * bs * (n - i * bs) \tag{14}$$

where α_b is the latency of the broadcast implementation independently of the number of computers involved, lpp is the latency per processor of the broadcast implementation, $p-1$ is the number of receivers in a broadcast operation, and Eq. (14) is the timing model of the broadcast implementation used in the current version of the parallel LU factorization algorithm.

2.3 Computing and Communication Time Requirements

Taking into account the pseudocode in Fig. 5 and the algorithm behavior described in Fig. 8, computing tA_i and broadcast communication of the next current block in each iteration should be compared. The broadcast communication time in the *i_th* iteration is given in Eq. (14), but the computing time in each iteration has been only partially given above.

The number of floating point operations for updating the trailing matrix is given in Eq. (11). Two important factors should be taken into account for modeling the computing time:

- The processing workload is evenly distributed amongst p computers, and each one of them has to carry out $1/p$ of the total workload. Thus, computing is made simultaneously in p computers and the required time is reduced by a factor of p.
- The time required for a single floating point operation, which has to be used to *translate* from processing workload in terms of number of floating point operations to computing time.

Rewriting Eq. (11)

$$\begin{aligned} ReqFlops(tA_i) &= (2 * bs - 1) * (n - i * bs) * (n - i * bs) + (n - i * bs) * (n - i * bs) \\ &= (2 * bs - 1) * (n - i * bs)^2 + (n - i * bs)^2 \\ &= (n - i * bs)^2 * ((2 * bs - 1) + 1) \\ &= 2 * bs * (n - i * bs)^2 \end{aligned} \tag{15}$$

Taking into account that the processing workload is evenly distributed amongst p computers, the expected time required for floating point operations on each computer in iteration i is modeled by

$$\begin{aligned} et(tA_i, p) &= \frac{tf * ReqFlops(tA_i)}{p} \\ &= \frac{tf * 2 * bs * (n - i * bs)^2}{p} \end{aligned} \tag{16}$$

where tf is the time required for a single floating point operation.

Taking into account Eq. (15), Eq. (14), and Fig. 8, the time required to complete the parallel algorithm on p processors is given by

$$et(parLU, p) = \sum_{i=1}^{n/bs} max(et(tA_i, p), t(bcast(i, p))) \tag{17}$$

It is expected that the numerical computing time in the first iterations is greater than the time required by broadcast communications. Also, given that

- the trailing matrix is made smaller as more iterations are completed, and
- broadcast message latency is constant from the point of view of trailing matrix size,

there is some iteration value k, for which computing time is lower than communication time, i.e.,

$$\begin{aligned} et(tA_i, p) &\geq t(bcast(i, p)); \quad i \leq k \\ et(tA_i, p) &< t(bcast(i, p)); \quad i > k \end{aligned}$$

and Eq. (17) *becomes*

$$et(parLU, p) = \sum_{i=1}^{k} et(tA_i, p) + \sum_{i=k+1}^{n/bs} t(bcast(i, p)) \tag{18}$$

The specific value of k is dependent on many factors, and the analysis would be highly simplified if it were known *a priori*. More specifically, the analysis can be made by finding out the value at which computing time is equal to communication time, i.e.

$$\frac{tf * 2 * bs * (n - i * bs)^2}{p} = \alpha_b + lpp * (p - 1) + \beta_{bcast} * bs * (n - i * bs) \tag{19}$$

For a fixed value of p $\alpha_b + lpp * (p - 1)$ is constant, and given that $(n - i * bs)$ changes with i:

$$\alpha_p = \alpha_b + lpp * (p - 1) \tag{20}$$

$$is = n - i * bs \tag{21}$$

Thus, Eq. (19) can be rewritten as

$$((tf * 2 * bs)/p) * is^2 = \alpha_p + \beta_{bcast} * bs * is \tag{22}$$

which can be rewritten as the quadratic equation

$$\begin{aligned} ((tf * 2 * bs)/p) * is^2 - \beta_{bcast} * bs * is - \alpha_p = 0 \\ \underbrace{((tf * 2 * bs)/p)}_{a} * is^2 + \underbrace{-\beta_{bcast} * bs}_{b} * is + \underbrace{-\alpha_p}_{c} = 0 \end{aligned}$$

i.e.

$$a * is^2 + b * is + c = 0 \tag{23}$$

with

$$a = (tf * 2 * bs)/p \tag{24}$$

$$b = -\beta_{bcast} * bs \tag{25}$$

$$c = -\alpha_p \tag{26}$$

and the values of is which satisfy Eq. (23) are given by the quadratic formula

$$is = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a} \tag{27}$$

A quite deeper analysis is now needed to find out at least if it can be assured that the square root is applied to a number greater than 0. Taking into acount Eq. (24) and Eq. (25), the term

$$-4 * a * c = -4 * ((tf * 2 * bs)/p) * (-\alpha_p) = 4 * \alpha_p * (tf * 2 * bs)/p > 0 \tag{28}$$

and, thus,

$$b^2 - 4 * a * c > 0$$

and there will be two values $\in \mathbb{R}$ from the results of the square root to such expression. Now, it is necessary to know which value of is will be used, given that Eq. (27) provides two. Defining

$$is_+ = \frac{-b + \left|\sqrt{b^2 - 4 * a * c}\right|}{2 * a} \tag{29}$$

and

$$is_- = \frac{-b - \left|\sqrt{b^2 - 4 * a * c}\right|}{2 * a} \tag{30}$$

it is necessary to know whether is_+ or is_- will be used. Taking into acount Eq. (28),

$$\left|\sqrt{b^2 - 4 * a * c}\right| > \left|\sqrt{b^2}\right| \tag{31}$$

thus,

$$+\left|\sqrt{b^2 - 4 * a * c}\right| > |b| \tag{32}$$

and, given that $b < 0$,

$$+\left|\sqrt{b^2 - 4 * a * c}\right| > -b \tag{33}$$

and this implies

$$-b - \left|\sqrt{b^2 - 4 * a * c}\right| < 0 \Longrightarrow is_- < 0 \tag{34}$$

In fact, this value of is_- should be used to find out the value of k, and using Eq. (21),

$$k_- = \frac{n - is_-}{bs} \tag{35}$$

but, given that $is_- < 0$,

$$k_- = \frac{n - is_-}{bs} = \frac{n}{bs} + \frac{-is_-}{bs} > \frac{n}{bs} \tag{36}$$

and this value of k is not useful, since k should be such that $1 \leq k \leq n/bs$ acording to Eq. (18). On the other hand,

$$-b + \left|\sqrt{b^2 - 4 * a * c}\right| > 0 \Longrightarrow is_+ > 0 \tag{37}$$

and

$$k_+ = \frac{n - is_+}{bs} = \frac{n}{bs} - \frac{is_+}{bs} < \frac{n}{bs} \tag{38}$$

and this value, k_+, has to be used for k in Eq. (18) or, in fact, the integer number immediately greater than k_+ as defined in Eq. (38).

Summarizing, the expected time for the broadcast-based parallel LU factorization algorithm with the broadcast implemented such as described in subsection 2.2 for a fixed number of computers is

$$\begin{aligned} et(parLU, p) &= \sum_{i=1}^{k} et(tA_i, p) + \sum_{i=k+1}^{n/bs} t(bcast(i, p)) \\ et(parLU, p) &= \sum_{i=1}^{k} \frac{tf * 2 * bs * (n - i * bs)^2}{p} + \sum_{i=k+1}^{n/bs} \alpha_p + \beta * bs * (n - i * bs) \end{aligned} \tag{39}$$

with

$$
\begin{aligned}
k &= \frac{n - is_{+}}{bs} = \frac{n - \frac{-b+\left|\sqrt{b^2-4*a*c}\right|}{2*a}}{bs} && (40)\\
a &= \frac{tf*2*bs}{p} && (41)\\
b &= -\beta_{bcast}*bs && (42)\\
c &= -\alpha_p && (43)
\end{aligned}
$$

where

- bs is the block size used for the parallel algorithm.
- tf is the time required for a single floating point operation.
- $1/\beta_{bcast}$ is the network communication bandwidth achieved with the broadcast communication routine.
- α_p is the communication latency or startup cost of the broadcast communication routine with p computers.
- p is the fixed number of computers.

3 Comparison with ScaLAPACK: Performance Analysis

The expected time for the Scalapack LU factorization algorithm is well known [4] [5] [3]:

$$et(ScaLU,p) = \frac{2*n^3}{3*p}tf + \frac{(3+log_2(p)/4)*n^2}{\sqrt{p}}\beta_{ptp} + (6+log_2(p))*n*\alpha_{ptp} \qquad (44)$$

where α_{ptp} and β_{ptp} are the message latency and the inverse of the bandwidth for point-to-point messages respectively, and the rest of parameters/coefficients have already been explained and used. Some different points of view prevent a direct comparison between Eq. (39) and Eq. (44) above. The first different approach in modeling the required time is on the number of floating point operations. The first term of Eq. (44) reflects the number of floating point operations in ScaLAPACK's timing model: $2/3*n^3$. This is the *traditional* number of operations for the sequential LU factorization as given in the literature [9]. The timing model given for the proposed parallel algorithm takes into account that most of the computing time is needed for the trailing matrix update whose number of operations is given in Eq. (15) for the *i_th* iteration. However, both algorithms are directly based on the blocked LU factorization, so the number of floating point operations should be the same and a deeper comparison analysis is not necessary to determine which one -Eq. (39) or Eq. (44)- is more accurate.

The ScaLAPACK timing model for communication is reflected in the second and third terms of Eq. (44). The first strong difference with the approach proposed in this report is that ScaLAPACK's communication costs are taken into account for every block/element of the matrix. On the other hand, for the approach proposed in this paper, Eq. (17) and Eq. (18) directly reflect that a broadcast communication adds time to the total expected algorithm time only when it is greater than the corresponding trailing matrix update time. Even if the numerical computing time is greater than the broadcast time in only a few iterations -e.g. $k = 20$ or $k = 30$ in Eq. (18), the communication time (in those iterations) would not add time to the total processing time, since it is overlapped with numerical computing. However, the broadcast timing model of Eq. (14) is far from optimal and implies at least that the latency grows linearly with the number of processors. On the other hand, ScaLAPACK relies on spanning trees and, thus, the timing model implies a logarithmical growth depending on the number of processors.

The next subsection will show a small example of the expected time for each algorithm on a *current* cluster. The specific values of parameters needed for modeling the performance of each algorithm are given, along with their similarities/differences.

3.1 Expected Times: Specific Example

Having a real cluster, it is possible to calculate the expected time of each parallel algorithm using Eq. (44) and Eq. (39) for the ScaLAPACK and the broadcast-based LU factorization algorihtms respectively. The

Table 1: Cluster Characteristics.

Clock	Mem	Mflop/s (DGETRF)
2.4 GHz	1 GB	$\cong 2500$

specific cluster is composed of 20 identical computers, whose characteristics are summarized in Table 1 and the interconnection network is 100 Mb/s Ethernet with complete switching. The specific values for the parameters required by the ScaLAPACK time model are given in Table 2, where β_{ptp} and α_{ptp} were

Table 2: ScaLAPACK's Parameter Values.

Parameter	Value
p	20
tf	$(2500 * 10^6)^{-1}$ sec.
β_{ptp}	$(11.6/8 * 10^6)^{-1}$ sec.
α_{ptp}	$200 * 10^{-6}$ sec.

measured with the MPICH implementation of MPI, using the ping-pong program distributed along with the implementation. Given the memory available on each computer and the number of computers, the matrix size was set to $n = 45000$. The expected running time of the ScaLAPACK's LU on the cluster with 20 computers just described is computed using Eq. (44):

$$et(ScaLU, 20) \cong 2582 \ sec. \tag{45}$$

Taking into account that

- 20 computers of approximately 2.5 Gflop/s have a computing power of aproximately 50 Gflop/s,
- the total number of floating point operations required for a LU factorization of a matrix of 45000×45000 elements is 2/3 * 45000^3,

the theoretical (minimum) time for 20 computers is given by

$$tt(LU, 20) = 1215 \ sec. \tag{46}$$

And this implies that the expected (parallel) efficiency of the ScaLAPACK parallel LU factorization algorithm is

$$ee(ScaLU, 20) = \frac{tt(LU, 20)}{et(ScaLU, 20)} \cong \frac{1215}{2582} \cong 0.47 \tag{47}$$

Thus, it is expected that the ScaLAPACK parallel LU matrix factorization algorithm will make use of less than 50% of the available computing power of this specific cluster. LU matrix factorization is specially penalized in ScaLAPACK's two dimensional matrix distribution due to the partial pivoting needed for numerical stability. Partial pivoting implies a collective communication in a row or a column of processors (for pivot selection) which implies a group communication penalization in an algorithm defined mainly for point-to-point communications. Given that the proposed parallel LU matrix factorization distributes data by column block or row block, this penalization is not found.

On the other hand, the specific values for the parameters required by the time model of the parallel LU matrix factorization proposed in this report are given in Table 3. The values of p and tf are exactly the same as those given in Table 2, for the ScaLAPACK analysis. The data bandwidth obtained by the broadcast routine implemented on top of UDP, $1/\beta_{bcast}$ is similar to that obtained by MPICH point-to-point messages. More specifically, the data bandwidth of the implemented broadcast routine is about 13.8% worse than that obtained by MPICH point-to-point messages. The communication latency of the implemented broadcast routine for 20 computers, α_p, is much worse than that of the MPICH point-to-point messages. One of the reasons has been explained above: acknowledgements are sent from every receiver (19 in this specific case) to the sender. More specifically, the value of α_p is about three *orders of magnitude* worse than α_{ptp}. The parameter bs in Table 3 is *relatively new* because there is not a similar parameter for the ScaLAPACK analysis. However, ScaLAPACK routines *do need* such a value

Table 3: Broadcast-Based Parameter Values.

Parameter	Value
p	20
tf	$(2500 * 10^6)^{-1}$ sec.
β_{bcast}	$(10/8 * 10^6)^{-1}$ sec.
α_p	0.1 sec.
bs	64

for running on a given parallel platform (and, also, that of the processors grid configuration). There are not many possible values for bs, being 32, 64, and 128 the most used in the ScaLAPACK references. The same values (32, 64 and 128) could be used for the broadcast-based parallel LU factorization analysis, 64 was finally used in the experiments. The matrix size was set as for the ScaLAPACK analysis, i.e. $n = 45000$. The expected running time of the proposed parallel LU factorization routine on the cluster with 20 computers just described is computed using Eq. (39). The first value to be obtained is the specific iteration in which communication time is greater than computing time, k in Eq. (40):

$$k = 363 \tag{48}$$

and this means that the communication time of the first 363 iterations is expected to be hidden by the the computing time. Also, it is worth noting that the rest of the iterations

$$\frac{n}{bs} - k = \frac{45000}{64} - 363 \cong 703 - 363 = 340$$

computing is hidden by communication time, i.e. computing time is not *added to* communication time, since communications are carried out *concurrently*. Now, it is possible to obtain the expected time for the parallel LU factorization using Eq. (39):

$$et(parLU, 20) \cong 1298 \ sec. \tag{49}$$

which is very near the maximum theoretical time, $tt(LU, 20) = 1215$ seconds. The expected efficiency of the broadcast-based parallel LU factorization is

$$ee(parLU, 20) = \frac{tt(LU, 20)}{et(parLU, 20)} \cong \frac{1215}{1298} \cong 0.94 \tag{50}$$

Thus, it is expected that the proposed parallel LU matrix factorization algorithm will use about 94% of the available computing power of this specific cluster.

From the point of view of the analysis, the proposed parallel algorithm is far better than the ScaLAPACK algorithm for parallel LU factorization at least on this specific cluster. The ScaLAPACK algorithm is expected to use 47% of the available computing power while the proposed parallel algorithm is expected to use about 94% of the available computing power. The experimentation should make clear the accuracy of the time models as well as the real performance difference between both algorithms at least on the specific cluster described in this section.

4 Comparison with ScaLAPACK: Experimentation

Some simple experimentation will clarify the comparison on a real environment. Computers (PCs) used for experimentation have the characteristics summarized above in Table 1, which is copied here

Clock	Mem	Mflop/s (DGETRF)
2.4 GHz	1 GB	$\cong$ 2500

and the interconnection network is 100 Mb/s Ethernet with complete switching. Performance in the table above is given in Mflop/s and DGETRF, the sequential LU matrix factorization with double precision floating point number representation, was used to measure sequential performance.

The total number of available computers is 20, and experiments were made with 2, 4, 8, 16, and 20 computers. Matrix sizes are scaled up according to the number of computers and memory available.

Local/sequential computing is made by using fully optimized ATLAS BLAS (Automatically Tuned Linear Algebra Software BLAS) [16]. ScaLAPACK communication is made as usual: BLACS (Basic Linear Algebra Communication Subroutines) implemented on top of MPICH implementation of MPI. Every possible bidimensional processors grid $P \times Q$ was considered for ScaLAPACK routines, e.g. for 16 processors, the experimental grids were: 1×16, 16×1, 2×8, 8×2, and 4×4. Also, square block sizes were used for ScaLAPACK routines: 16, 32, 64, 100 and 128. The proposed algorithm does not need to define a bidimensional processors grid, and the block values used for experimentation are the same as those used for ScaLAPACK routines. Given the one-dimensional matrix partitioning and distribution described for the algorithm, the matrix is divided in row blocks instead of square blocks.

Figure 12 shows the parallel perfomance measured as efficiency for LU matrix factorization on different number of computers from 2 to 20. The matrix order (size) for each number of computers is shown between parenthesis on the x axis. Bars show the best efficiency value obtained by the algorithms for each number of computers. Light gray bars labeled as "Sca" correspond to values obtained by ScaLAPACK's PDGETRF. Dark gray bars labeled as "Prop" correspond to values obtained by the proposed parallel LU matrix factorization algorithm. It is worth to mention the similarity among the ScaLAPACK's results

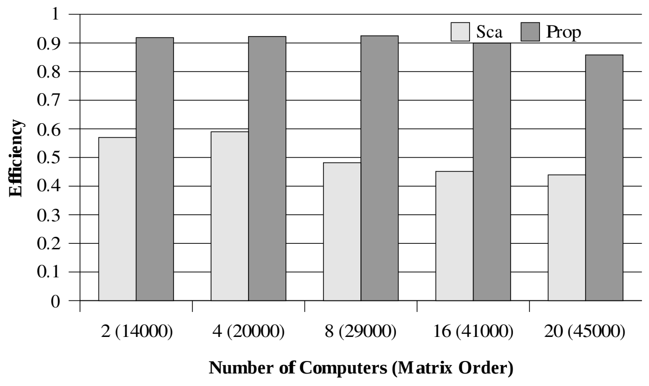

Figure 12: LU Matrix Factorization Efficiency.

shown in Fig. 12 with those in [5], where ScaLAPACK is used for LU matrix decomposition and linear equation system solving. In Fig. 12 as well as in [5] the efficiency is about 0.5 (or 50% of the total available computing power).

Experiments made using different numbers of computers -2, 4, 8, 16 and 20- make it possible to evaluate the performance from the point of view of scalability. This scalability analysis can be made with the timing models given in the previous subsection. The proposed algorithm performance is better than that implemented in ScaLAPACK from the point of view of "raw" efficiency and performance degradation from 2 to 20 computers. The proposed parallel LU matrix factorization algorithm efficiency for 20 computers is about 7% worse than the efficiency for 2 computers, while SaLAPACK efficiency for 20 computers is about 23% worse than the efficiency for 2 computers. At least for this cluster, the scalability of the proposed algorithm is better than that of the ScaLAPACK algorithm.

5 Conclusions and Further Work

The timing model of the broadcast-based parallel LU matrix factorization has been introduced in this report. This timing model can be used for performance prediction as well as for performance comparison with other approaches for parallel LU matrix factorization.

From the analytical point of view, the parallel LU matrix factorization algorithm is expected to obtain very good performance (speedup and efficiency) values. More specifically, using the time model given for the algorithm the expected efficiency on a cluster with 20 computers is about 0.94, i.e. it is expected to use about 94% of the available computing performance on a specific cluster with 20 computers. Using the timing model of the ScaLAPACK parallel LU matrix algorithm for the same cluster, the expected efficiency is about 0.47, i.e., the ScaLAPACK parallel LU matrix factorization algorithm expects to use less than 50% of the available parallel performance.

Experiments have shown that the broadcat-based parallel LU matrix factorization algorithm obtains better performance values than the ScaLAPACK approach on a specific cluster. Also, experiments have shown that timing models for both parallel algorithms are very accurate at least for the total number of computers available for experimentation. Table 4 shows the summary of the timing models as well as the experimentation for 20 computers. Values in the column Expected Eff. are obtained using the timing

Table 4: Summary of Values for 20 Computers.

Algorithm	Expected Eff.	Experim. Eff.	Accuracy	% Better (Experim.)
ScaLAPACK	0.47	0.44	+6.8%	-
Broadcast-Based	0.94	0.86	+9.3%	95.45%

models for the algorihtms and values in the column Experim. Eff. are those obtained in the experiment on the *current* cluster with 20 computers. Timing models accuracy is shown in the column Accuracy of Table 4 and the last column show that the broadcast-based algorithm obtains more than 95% better performance than the ScaLAPACK algorithm on this specific cluster.

Table 5 shows the summary of the experimentation in the cluster. Values in the columns ScaLAPACK Eff. and Broadcast-Based Eff. were obtained in the experimentation on the clusters with different numbers of computers. Values in the column % Better (Broadcast-Based) show that the broadcast-

Table 5: Experimentation Summary.

Number of Computers	ScaLAPACK Eff.	Broadcast-Based Eff.	% Better (Broadcast-Based)
2	0.57	0.92	61.2%
4	0.59	0.92	56.49%
8	0.48	0.93	92.01%
16	0.45	0.9	99.36%
20	0.44	0.86	95.35%

based LU factorization algorithm obtains much better performance than the ScaLAPACK LU matrix factorization algorithm. Furthermore, as the number of computers is greater, the difference in performance between both algorithms is better for the broadcast-based one, thus showing that the broadcast-based algorithm has better scalability than the ScaLAPACK algorithm.

References

[1] Anderson E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-Performance Computers", Proceedings of Supercomputing '90, pages 1-10, IEEE Press, 1990.

[2] Anderson E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide (Second Edition), SIAM Philadelphia, 1995.

[3] Blackford L., J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. Whaley, ScaLAPACK Users' Guide, SIAM, Philadelphia, 1997.

[4] Chen Z., J. Dongarra, P. Luszczek, K. Roche, "Self adapting software for numerical linear algebra and LAPACK for clusters", Parallel Computing 29, pp. 1723-1743, Elsevier B.V., 2003.

[5] Chen Z., J. Dongarra, P. Luszczek, K. Roche, "The LAPACK for Clusters Project: an Example of Self Adapting Numerical Software", Proceedings of the 37th Hawaii International Conference on System Sciences, pp. 1-10, 0-7695-2056-1/04, IEEE, 2004.

[6] Choi J., J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, R. Whaley, "The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines", Report ORNL/TM-12470, Sep. 1994. Also available as a "lawn".

[7] Dongarra J., J. Du Croz, S. Hammarling, R. Hanson, "An extended Set of Fortran Basic Linear Subroutines", ACM Trans. Math. Soft., 14 (1), pp. 1-17, 1988.

[8] Dongarra J., D. Walker, "Libraries for Linear Algebra", in Sabot G. W. (Ed.), High Performance Computing: Problem Solving with Parallel and Vector Architectures, Addison-Wesley Publishing Company, Inc., pp. 93-134, 1995.

[9] Golub G., C. Van Loan, Matrix Computations, 2nd Edition, The John Hopkins University Press, 1989.

[10] Kumar V., A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing. Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, Inc., 1994.

[11] Lawson C., R. Hanson, D. Kincaid, F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage", ACM Transactions on Mathematical Software 5, pp. 308-323, 1979.

[12] Tinetti F. G., "Cómputo Paralelo en Redes de Estaciones de Trabajo para Aplicaciones Basadas en Algebra Lineal", Fernando G. Tinetti, Reporte Técnico PP004 - 01, Centro de Técnicas Analógico-Digitales (CeTAD), Fac. de Ingeniería, Laboratorio de Investigación y Desarrollo en Informática (LIDI), Facultad de Informática, Laboratorio de Química Teórica (LQT), CEQUINOR, Departamento de Química, Facultad de Ciencias Exactas, Universidad Nacional de La Plata, Julio de 2001.

[13] Tinetti F. G., "LU Factorization: Number of Floating Point Operations and Parallel Processing in Clusters", Laboratorio de Investigación y Desarrollo en Informática (LIDI), Facultad de Informática, Universidad Nacional de La Plata, Mayo de 2003, Reporte Técnico PLA-001-2003.

[14] Tinetti F. G., "Guidelines for Parallel Linear Algebra on Ethernet-Based Clusters: Matrix Multiplication and LU Factorization Results", Laboratorio de Investigación y Desarrollo en Informática (LIDI), Facultad de Informática, Universidad Nacional de La Plata, Junio de 2003, Reporte Técnico PLA-002-2003.

[15] Tinetti F. G., E.Luque, "Parallel Matrix Multiplication on Heterogeneous Networks of Workstations", Proceedings VIII Congreso Argentino de Ciencias de la Computación (CACIC), Fac. de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Buenos Aires, Argentina, p. 122, Oct. 2002. Available at http://lidi.info.unlp.edu.ar/ fernando/publis/pmm.pdf

[16] Whaley R. C., A. Petitet, J. J. Dongarra, Automated Empirical Optimization of Software and the ATLAS Project. Available at http://www.netlib.org/lapack/ lawns/lawn147.ps