Workshop 07

Parallel Numerical Algorithms

Optimization of the ScaLAPACK *LU* Factorization Routine Using Communication/Computation Overlap*

F. Desprez[1], S. Domas[1] and B. Tourancheau[1]

LIP, ENS Lyon, CNRS URA 1398, INRIA Rhône-Alpes
46, Allée d'Italie, 69364 LYON cedex 07, France

Abstract. This paper presents some optimizations based on communications/computations overlap for the ScaLAPACK *LU* factorization. First a theoretical computation of the optimal block size is given for the block scattered decomposition of the matrix. Two optimizations of this routine are presented that use asynchronous communications to hide the communication overhead and to obtain optimal speed-ups.

1 Introduction

The *LU* factorization is the kernel of many applications. Thus, the importance of optimizing this routine has not to be proven because of the increasing demand for solving large dense system. Its efficient parallel implementation can bring real improvements in the execution speed of the whole application. High performances are obtained on vector machines, but a prohibitive cost. Distributed memory machines seem to be a good balance between performances and cost.

Portability is one of the key issue of computer programming. Many libraries have been designed to ensure portability and performances across multiple architectures. The BLAS [5, 7] and LAPACK [6] are available on many platforms, provided by computers vendors. *LU* factorization was released in the LAPACK package, using levels 1, 2 and 3 BLAS. ScaLAPACK [1] contains the parallel version of subsets of the BLAS and LAPACK and has been designed to ensure portability, performances and ease of use across many parallel machines. Matrices are distributed in a block scattered way. Parallelism is hidden in a parallel version of the BLAS called PBLAS [1]. Communications between processors on a virtual grid are done using the BLACS package.

Various methods have been proposed to improve the parallel *LU* factorization. The corresponding papers present experiments which are sometimes confirmed by complexity studies [1, 2, 4, 8]. They are presented in our technical report [3].

The aim of this paper is to show that improvements can be obtained in the existing ScaLAPACK *LU* factorization routine by the use of communication/computation overlap.

* This work has been supported by the INRIA Rhône-Alpes and the EUREKA-EUROTOPS project.

2 Parallel block *LU* decomposition

The block LU decomposition consists in three phases, repeated as many times as there are block columns to be factorized in the global matrix: instead of working on a single column of the global matrix A at a time, r columns are factored at each step. And for convenience, the local L and U matrices are stored in place of the matrix to be decomposed.

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} * \begin{pmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{pmatrix}$$

Three steps are thus necessary to compute the LU factorization of a matrix. In order to obtain (L_{00}, L_{10}) and U_{00}, a simple Gaussian elimination is computed on $\binom{A_{00}}{A_{10}}$ ($L_{00}U_{00} = A_{00}$ and $L_{10}U_{00} = A_{10}$). The U_{01} block is obtained by a triangular solve ($L_{00}U_{01} = A_{01}$). $L_{11}U_{11}$ is obtained using equation $L_{10}U_{01} + L_{11}U_{11} = A_{11}$. A matrix product is needed ($A_{11} - L_{10}U_{01}$). The three steps are recursively computed on $L_{11}U_{11}$ to obtain L_{11} and U_{11}.

In ScaLAPACK, the parallel LU factorization uses a block scattered decomposition of matrix A on a $P \times Q$ processors grid. The $M \times N$ matrix is divided in square blocks ($r \times r$). Thus, each processor owns a local matrix with $\lceil \frac{M}{P \times r} \rceil \times \lceil \frac{N}{Q \times r} \rceil$ blocks. The general parallel algorithm is given in Figure 1. `_TRSM` is the level 3 BLAS triangular solve routine and `_GEMM` is the general matrix product routine. The ScaLAPACK routines `PDGETRF` and `PDGETF2` execute the block LU factorization on a matrix distributed in a block scattered way. `PDGETF2` performs the factorization of a block column to compute L_{00},L_{10}, and U_{00} (phase 1 of the general algorithm in Figure 1). `PDGETRF` calls `PDGETF2`, then updates the remaining blocks of the matrix by computing U_{01} and $L_{11}.U_{11}$ (phase 2 and 3 of the general algorithm in Figure 1).

3 Analysis and optimization of the ScaLAPACK *LU* factorization

3.1 Complexity analysis

A prediction of the execution time is important to confirm the experimental results. In the ScaLAPACK LU factorization, performance depends greatly on the block size used for the block scattered decomposition. Thus, it is interesting to compute the optimal block size to avoid a number of tests. It is also interesting to compute automatically the best data distribution (for parallel compilers, for example).

The theoretical optimal block size S_b is obtained by an interpolation, based on experimental measurements (for a given supercomputer) of the LU subroutines. For all subroutines function of S_b (see below), the execution time is expressed literally as a sum of polynomials (with a_i, b_i... the coefficients given by the interpolation), and then derived to find the optimal S_b. The complexities are given below with S_b as the block size, B_c the number of block columns (or

```
pcol = 0, prow = 0
for k = 0 to min(M_b, N_b) − 1 by step r do
   for i = 0 to r − 1 do                                          } phase 1
     if (my_col = pcol) then find pivot and its position
     broadcast the two values to all processors
     exchange pivot rows
     if (my_col = pcol) then
       div. under-diag. elts. of col. i by piv. /* _GER */
   end for
   if (my_row = prow) then                                        } phase 2
     broadcast L_00 to all processors of the prow row
     solve L_00.U_01 = A_01 /* _TRSM */
   end if
   broadcast L_10 on proc. rows and U_01 on proc. columns         } phase 3
   update A_11 ← A_11 − L_10.U_01 /* _GEMM */
   pcol = (pcol + 1)modQ, prow = (prow + 1)modP
end for
```

Fig. 1. Parallel block LU factorization using a block scattered data distribution.

rows) in a single processor (i.e. $\frac{M}{P_c \times S_b}$), P_c the number of processor columns (or rows) on the grid (the grid is assumed to be squared (P_c^2 processors) in order to simplify the calculus below) and M the matrix size. The subroutines names are the BLAS or LAPACK names. We only give the complexity results of the three most important subroutines but the complete study can be found in [3].

DGER: is executed S_b times for each **PDGETF2** call, with a decreasing data size. The execution time of a single **DGER** call is quadratic (level 2 BLAS), but a pseudo-linear time, function of S_b, can replace it: $t^{S_b}_{dger}(i) = a_{dger}.(i \times S_b) + b_{dger}$ with $a_{dger} = \alpha_{dger} \times S_b$ and $b_{dger} = \gamma_{dger} \times S_b + \lambda_{dger}$.

$$T_{dger} = \sum_{i=1}^{B_c} P_c . \sum_{j=1}^{S_b - 1} [(\alpha_{dger} \times j)(S_b \times i) + (\gamma_{dger} \times j + \lambda_{dger})] \qquad (1)$$

DTRSM: is executed $B_c \times P_c$ time, with a variable data size, multiple of S_b. The execution time of a single **DTRSM** call is cubic, but a pseudo-linear time, function of S_b can replace it: $t^{S_b}_{dtrsm}(i) = a_{dtrsm}.(i \times S_b) + b_{dtrsm}$, with $a_{dtrsm} = \alpha_{dtrsm} \times S_b^2 + \beta_{dtrsm} \times S_b + \delta_{dtrsm}$, and $b_{dtrsm} = \gamma_{dtrsm} \times S_b + \lambda_{dtrsm}$.

$$T_{DTRSM} = \sum_{i=1}^{B_c} P_c.[a_{dtrsm}.(i \times S_b) + b_{dtrsm}] - [a_{dtrsm}.(B_c \times S_b) + b_{dtrsm}]^2 \qquad (2)$$

DGEMM: executed $B_c \times P_c$ time, with a variable data size, multiple of S_b. The execution time of a single **DGEMM** call is cubic, but a quadratic time, function of

[2] **DTRSM** is executed one less time at maximal size because of block scattered distribution and LU algorithm.

S_b, can replace it: $t^{S_b}_{dgemm}(i) = a_{dgemm}.(i^2 \times S_b^2) + b_{dgemm}.(i \times S_b) + c_{dgemm}$, with $a_{dgemm} = \alpha_{dgemm} \times S_b$, $b_{dgemm} = \beta_{dgemm} \times S_b + \delta_{dgemm}$ and $c_{dgemm} = \gamma_{dgemm} \times S_b + \lambda_{dgemm}$.

$$T_{DGEMM} = \sum_{i=1} P_c.[a_{dgemm}.(i \times S_b)^2 + b_{dgemm}.(i \times S_b) + c_{dgemm}] - [a_{dgemm}.(B_c \times S_b)^2 + b_{dgemm}.(B_c \times S_b) + c_{dgemm}]^3 \quad (3)$$

3.2 Results on Intel Paragon

In order to compute the optimal block size for each code, the derivative of each complexity over S_b is computed. The total complexity optimum is obtained when the sum of the derivatives equals zero. Hence, the optimal block size is given by the resolution of a third degree equation. We computed the three roots for different grid sizes and matrix sizes, from the coefficients (a_i, b_i ...) found on an Intel Paragon at Lyon. Only one solution was positive each time. Results are given in table 1. It gives the theoretical optimal block size as a function of different grid sizes and matrix sizes (1000 signifies a 1000×1000 matrix in double precision). For a 4×4 grid and up to matrix size 4000, experimental results are also given.

grid size	matrix size							
	1000	2000	3000	4000	5000	6000	7000	8000
4×4	8.31	9.28	9.64	9.83	9.94	10.0	10.1	10.1
4×4 exp.	8 - 9	8	8	10	/	/	/	/
8×8	7.38	8.38	8.79	9.03	9.17	9.27	9.35	9.41
16×16	7.26	8.0	8.38	8.61	8.77	8.88	8.96	9.03

Table 1. Optimal block size on Lyon Intel Paragon.

Five important points have to be noticed. Theoretical optimal block sizes are not exactly the same as experimental ones but they range between 7.2 and 10.1. Actual tests give an optimal block size of 8 or 10 on a 4×4 grid.[4] The theoretical optimal block size is a function of the matrix size, and raises up to a top value around 10. Thus, a "good" theoretical block size is given by the asymptotic value. The optimal block size does not very depend on the number of processors. Asymptotic values are 9 for 256 processors and 10 for 16. Again, experimental results confirm this point. The grid is assumed to be squared for convenience. According to experimental results, the grid shape has no real influence on the optimal block size, though it greatly influences the execution time [1]. A rectangular grid with few rows of processors works faster than a square grid : there are less communications since pivoting is achieved more often in local memory. Results are identical if only the subroutines `DGER`, `DTRSM`, and `DGEMM` are used for block size computation (they represent 95% of total computation time).

[4] A size of 16 was found on previous tests. It has been done on the same machine but with an older version of the operating system. Results depend greatly on the machine and its system.

3.3 Optimizations

The ScaLAPACK version of LU has been implemented in order to be scalable: each subroutine call is a BLAS or BLACS call. These two libraries are already fully optimized for a lot of computers. Furthermore, the minimal number of operations to achieve a LU factorization is well-known ($\frac{2}{3}n^3 + 2n^2$). Thus, only communication phases can be optimized since the computation time is fixed. Asynchronous messages are used to overlap communications with computations.

Broadcast overlap By looking closely at the algorithm, we can see that processors are often waiting results from other processors. During the block column decomposition, only a processor column is working. And only one processor row is working during the triangular solve. This brings us to the first optimization solution: **"Instead of broadcasting (L_{00}, L_{10}) panel before the triangular solve (_TRSM), do it at the same time."** This means that general synchronous BLACS broadcasting routines (_GEBS2D and _GEBR2D) are used on processors that do not compute _TRSM, and asynchronous communications are used to send (L_{00}, L_{10}) during _TRSM on processors that compute it. Therefore, the single block L_{00} must be broadcast on the current processor row to perform _TRSM. But it takes less time than broadcasting (L_{00}, L_{10}). This solution seems interesting since, at each step i of the factorization, we gain the broadcasting time of $P_b - i - 1$ blocks, compared to the original version. But, unfortunately, that gain represents only 1 to 2 percents of total factorization time (on the Paragon system). This is due to the number of times this broadcast is done (only $P_b - 1$ times).

Rows pivoting overlap It appears that an optimal speed-up cannot be obtained until a communication phase is overlapped most of the time. Thus, it is interesting to overlap the pivoting time since it is executed for each row of the matrix:**"Instead of broadcasting pivot informations and then exchanging rows after the (A_{00}, A_{10}) decomposition, do it at the same time."** This means that we can use the DSCAL time of the processors column which decompose (A_{00}, A_{10}), to exchange current and real pivot rows using asynchronous communications. Then we use the DGER time to send asynchronously the pivot informations to the next processor in the processors row. Thus, as soon as this processor receives pivot informations, it can exchange the rows for pivoting and send the information to the next processor in the row. And so on, until the last processor on the pseudo-ring receives its data. During this step, the block column decomposition continues on the processor column.

Figure 2 represents the different steps of these operations for an 64×64 matrix distributed on a 4×4 grid using a 8×8 block size. We explain this example in the following:

step 1: This is the k^{th} iteration of PDGETF2. The real pivot row has been found on processor 11. Then, processor 15, that owns the current pivot row k, divides the current pivot by the real pivot and asynchronously send the whole k^{th} row to processor 11. After, it proceeds with DSCAL and after, waits for the completion of the asynchronous send and receives the real pivot row. Identically, processor 11 asynchronously sends the whole pivot row to processor 15 and computes the

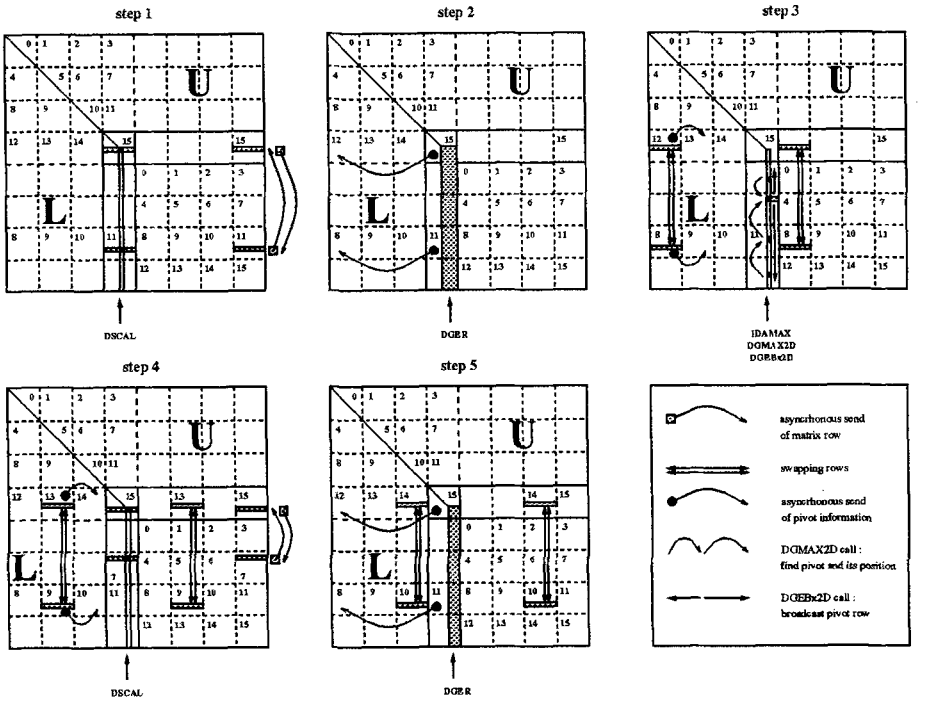

Fig. 2. One iteration of optimized PDGETF2 routine.

DSCAL at the same time. After completion, it receives the current pivot row. In the best case, processors 15 and 11 do not need to wait for send completion since the communication is already over when DSCAL ends. In fact, a wait state appears with a very large matrix, when their size reaches memory size limits. In this case, the DSCAL time does not completely overlap the communication time, since DSCAL domain size decreases each step. Other processors in the pivot processor column just execute the DSCAL routine.

step 2: DSCAL and pivoting is done. Now, the local sub-matrix must be updated with DGER. Processors 15 and 11 use this time to send the pivot information to their right neighbor on a pseudo-ring made from the processors row. In Figure 2, a pseudo-ring is $(12, 13, 14, 15)$, and the right neighbor of 15 is 12. Processors 12 and 8 are just waiting for pivot information using a blocking receive.

step 3: processors 12 and 8 have just received the pivot information. They can now exchange the k^{th} row and the pivot row, and send the pivot informations to their right neighbor. Meanwhile, processors in the pivot column continue the decomposition, finding the pivot and its position, broadcasting the local pivot row, ...

step 4: as in step 1, `DSCAL` is overlapped by an asynchronous exchange of current and real pivot row. But now, processors 13 and 9 are working, exchanging rows, instead of waiting for the completion of `PDGETF2` to work.

step 5: as in step 2, but with two more processors working.

3.4 Experimental results

All experimental results have been obtained on Intel Paragon systems, with various grid sizes, on two different machines (ORNL, Tennessee, and Lyon, France). These two machines have not the same version of the operating system and it appeared that it leads to different experimental results. There are four different grid sizes: 4×4, 6×5 (Lyon), 8×8, and 16×16 (ORNL).

A block size of 16 appears to be the optimal value for the LU factorization on the ORNL system, and 8 on the Lyon system. So, all Mflops are given for these block sizes and one RHS vector for the solve computation.

Figure 3 shows a comparison between optimized and non-optimized results for a 16×16 grid. The optimized version grows faster (in Mflops) than the non-optimized, before becoming almost parallel. Indeed, it works better for small matrix sizes ($< 375 \times P_c$ with P_c, the number of column of processors) because the `DSCAL` time can completely overlap the communication time of the pivoting rows. After this limit, processors have to wait for the completion of the communication.

Figure 4 shows the gain in percents over the non-optimized version, for 8×8 and 16×16 grids. It can reach 15% for small matrix sizes, and stay above 4% for the largest matrix size that can be allocated. This figure confirms that the speed-up progressively decreases for a matrix size greater than $375 \times P_c$.

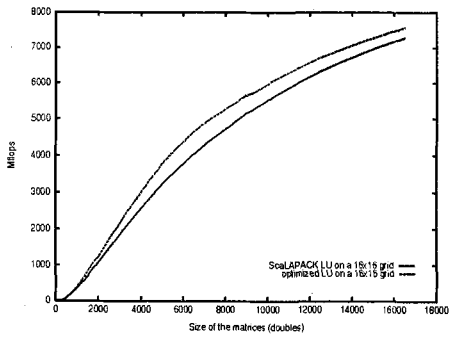

Fig. 3. Experimental results on a 16×16 grid.

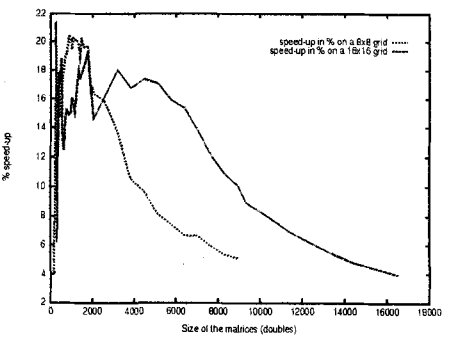

Fig. 4. Gain on the 8×8 and 16×16 grids.

4 Conclusion and future work

After a description of the *LU* algorithm in ScaLAPACK, a complete analysis of complexity has been presented. This theoretical model allows the computation of the optimal block size for the block scattered decomposition. Thus, it is possible to have the best performance with a simple pre-computation. The second part has presented two optimizations based on communication / computation overlap. In the two cases, our aim was to "hide" the time of some large communication phases. Furthermore, it allows to reduce the idle time of some processors that are waiting results from other to continue the execution.

All experimentations have been done on Paragon systems but the methods presented in this paper are general. Thus, they can be applied to any supercomputer. Meanwhile, some hints can be given. All the updates have been done using Paragon system calls syntax. Asynchronous BLACS do not exist. Consequently, our code is not portable. But the modifications are simple enough to be rewritten on any supercomputer. The speed-up decreases as the matrix size grows: the `DSCAL` time becomes not big enough to overlap the communications due to pivoting. Thus, another routine could be used to overlap such communications (like `DGER`, or `DGEMM`). The optimal block size computation is also interesting as input for parallel compilers because it gives the best data distribution for a given matrix size, and number of processors.

We are now working on the computation of optimal distributions and grain size when BLAS routines are chained.

References

1. J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. LAPACK Working Note: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performances. Technical Report 95, CS Dept - Univ. of Tennessee, 1995.
2. E. Chu and A. George. Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor. *Parallel Computing*, 5:65–74, 1987.
3. F. Desprez, S. Domas, and B. Tourancheau. Optimization of Parallel LU Factorization by Communication Overlap. Technical Report ???, LIP - ENS Lyon, 1996.
4. F. Desprez, J.J. Dongarra, and B. Tourancheau. Performance Complexity of LU Factorization with Efficient Pipelining and Overlap on a Multiprocessor. *Parallel Processing Letters*, 5-II, 1995.
5. J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, 1990.
6. J.J. Dongarra, R. Van De Geijn, and D.W. Walker. A Look at Dense Linear Algebra Libraries. Technical Report ORNL/TM-12126, Oak Ridge Nat. Lab., July 1992.
7. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.
8. B.V. Purushotham, A. Basu, P.S. Kumar, and L.M. Patnaik. Performance Estimation of LU Factorisation on Message Passing Multiprocessors. *Parallel Processing Letters*, 2(1):51–60, 1992.