Optimization of an
LU Factorization Routine
Using Communication/Computation Overlap

F. Desprez, S. Domas and
B. Tourancheau

October 31, 1996

Abstract

This paper presents some works on the LU factorization from the ScaLAPACK library. First, a complexity analysis is given. It allows to compute the optimal block size for the block scattered distribution used in ScaLAPACK LU. It also gives the communication phases that are interesting to overlap. Second, two optimizations based on computations/communications overlap are given with experimental results on Intel Paragon system and IBM SP2 system.

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 of applications dealing with large matrices. Its efficient parallel implementation can bring real improvements in the execution speed of the whole application. The speed-up depends greatly on the kind of supercomputer chosen. Vector machines have very high performances, 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 [4, 5, 10] 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] is the parallel version of a subset of LAPACK. ScaLAPACK 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 [9]. Communications between processors on a virtual grid are done using the BLACS package [7].

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

The first section presents the parallel block LU decomposition. The second section presents a recent bibliography on the parallel LU factorization. It contains a selection of papers that reflects the different solutions for this problem. The third section presents a complexity study of the ScaLAPACK LU routines, and two possible optimizations based on communication / computation overlap. The main goal of the complexity analysis is to provide the optimal block size for the block scattered decomposition, but it also gives information about the communication phases that may be overlapped.

2 Parallel Block LU Decomposition

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 $\left\lceil \frac{M}{P \times r} \right\rceil \times \left\lceil \frac{N}{Q \times r} \right\rceil$ blocks. This distribution is really suitable for the block LU decomposition and it provides a good load balancing between processors.

0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5

Figure 1: Block scattered decomposition of a 12r $\times$ 12r matrix on a 2×3 grid.

Figure 1 shows how to distribute the $r \times r$ blocks of a $12r \times 12r$ matrix on a 2×3 grid in a block scattered way. Squares marked with a number represent a single block of the global matrix. For example, all grey blocks belong to processor 0.

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.

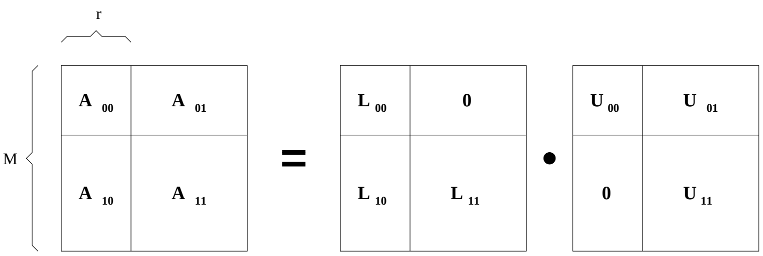

Figure 2: First step of the block LU factorization.

According to Figure 2, the three steps are the followings:

- in order to obtain (L_{00}, L_{10}) and U_{00}, a simple Gaussian elimination is computed on $\binom{A_{00}}{A_{10}}$.

$$L_{00}.U_{00} = A_{00} \tag{1}$$

$$L_{10}.U_{00} = A_{10} \tag{2}$$

- the U_{01} block is obtained by a triangular solve (see equation 3).

$$L_{00}.U_{01} = A_{01} \tag{3}$$

- $L_{11}.U_{11}$ is given by equation 4. A matrix product is needed $(A_{11} - L_{10}.U_{01})$.

```
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
       if (my_col = pcol) then
          find pivot and its position
       end if
       broadcast the two values to all processors
       exchange pivot rows                                         phase 1
       if (my_col = pcol) then
         divide under-diag.  elts.  of col.  i by pivot
         update col.  i+1 to r-1 /* _GER */
       end if
    end for

    if (my_row = prow) then
       broadcast L_00 to all processors of the prow row
       solve L_00.U_01 = A_01 /* _TRSM */                          phase 2
    end if

    broadcast L_10 on processors rows
    broadcast U_01 on processor columns                            phase 3
    update A_11 <- A_11 - L_10.U_01 /* _GEMM */

    pcol = (pcol + 1) mod Q
    prow = (prow + 1) mod P
end for
```

Figure 3: Parallel block LU factorization using a block scattered data distribution.

$$L_{10}.U_{01} + L_{11}.U_{11} = A_{11} \tag{4}$$

The three steps are recursively computed on $L_{11}.U_{11}$ to obtain L_{11} and U_{11}.

The general parallel algorithm is given in Figure 3. `_GER` is a level 2 BLAS, `_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. Figures 4 and 5 represent the execution schemes of these two routines:

- `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 3).
- `PDGETRF` calls `PDGETF2`, then updates the remaining blocks of the matrix to compute $L_{11}.U_{11}$, U_{01} (phase 2 and 3 of the general algorithm in Figure 3).

3 Related Work

Various methods have been proposed to improve the parallel LU factorization. The corresponding papers present experiments which are sometimes corroborated by complexity studies.

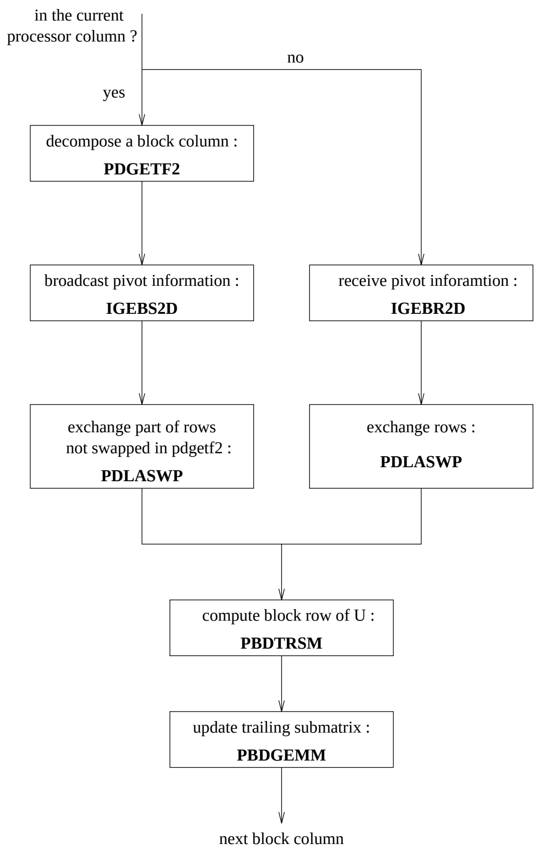

Figure 4: Execution scheme of the PDGETRF routine.

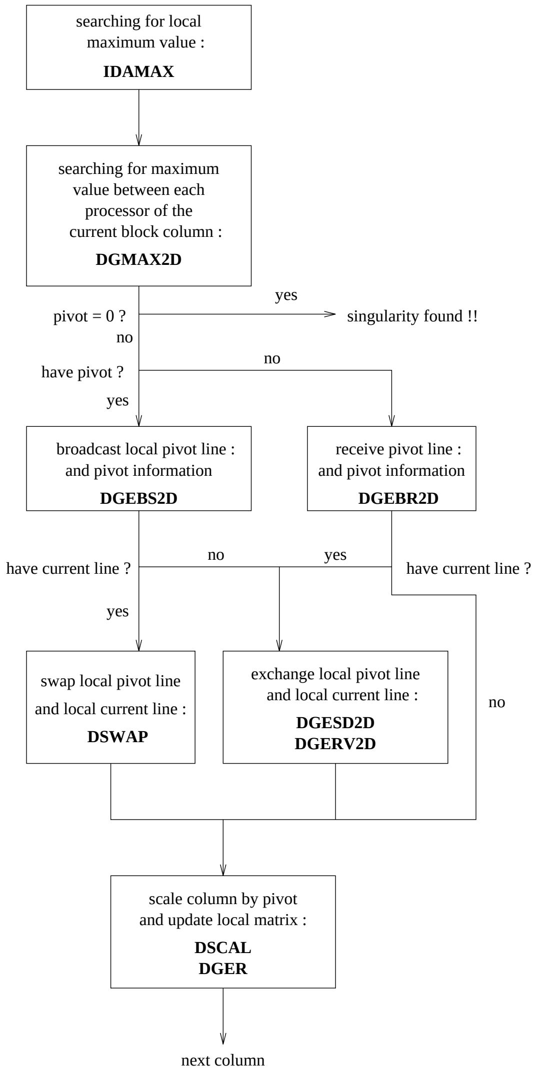

Figure 5: Execution scheme of the PDGETF2 routine.

In [8], a row oriented method is presented with an efficient pivot selection. It uses a reverse spanning tree broadcast to choose the real pivot among all the local pivots, owned by different processors. In ScaLAPACK, it is now achieved by the BLACS operation _GMAX2D. Furthermore, in [2], a load balancing strategy for the choice of the pivot is added to the row oriented method with a low cost row interchange.

Meanwhile, a row distribution supposes that the real pivot is often chosen on a different processor than the owner of the current row. Thus, there are some communications to exchange and to broadcast the pivot row.

In [11], a column-scattered distribution is chosen with the well-known pipelined ring algorithm. A complete performance estimation is provided for distributed memory parallel machines, and compared to experimental results.

In [3], a column-scattered distribution is chosen and asynchronous communications are used to overlap computations (columns updates). A theoretical model is given for the complexity on a complete network and a ring topology.

The two last methods are quite efficient because choosing the pivot on a single processor (instead of all, like in [2, 8]) is very fast and exchanging rows occurs in local memory. Therefore, it uses a fine grain parallelism.

ScaLAPACK LU is a block LU decomposition with a block scattered distribution of the data. In [9], a complexity study and different assumptions on the choice of the block size and the grid size are proposed. It appears that a rectangular grid with few rows of processors is optimal (for the reasons exposed upper). Thus, it can use the best of the precedent methods but with coarse grain computations.

The block scattered distribution used in the ScaLAPACK LU factorization seems to be the most interesting solution for distributed memory computers, as far as the block and the grid sizes are well chosen.

Consequently, this paper presents an optimized version of LU factorization with a block scattered distribution and an overlap between communications and computations. It also gives a method to compute an optimal block size without an "enquiry routine" (as previewed in [9]).

4 Analysis and Optimization of the ScaLAPACK LU Factorization

4.1 Complexity Analysis

A prediction of the execution time is important to confirm the experimental results. The value of the parameters which influence the execution time can be deduced from the model. Furthermore, it gives the execution time of the communication phases. Such predictions are very interesting to choose what communications to overlap.

In the ScaLAPACK LU factorization, performances depend 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).

Figure 6 shows the impact of the block size on the performance of the ScaLAPACK LU factorization. On the Intel Paragon at the University of Lyon, a size of 64 is less efficient than 8, which is the optimal value.

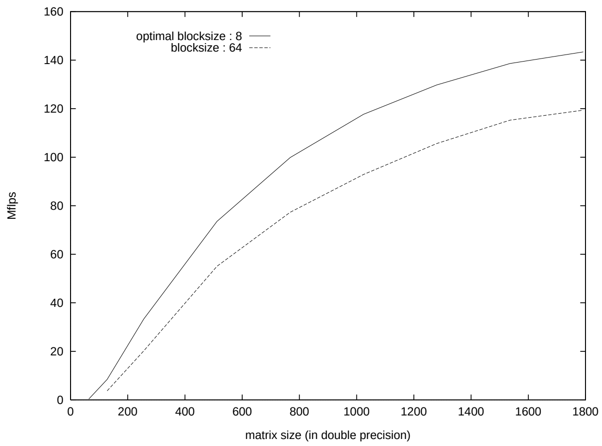

Figure 6: Performance (In Mflops) of LU factorization on 4 processors with two different block sizes on Intel Paragon.

4.1.1 The Model

In this section, 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, and then derived to find the optimal S_b.

The complexities are given below with S_b as the block size, P_r and P_c as the number of processor rows and columns, and M as the matrix size. The subroutines names are the BLAS or LAPACK names expressed in Figures 4 and 5. We assume that the global communications (`DGEBS2D, DGMAX2D, ...`) are proportional to the basic communication time $t(L) = \beta + L\tau$. For a given global communication op, $t_{op}(L) = f(\beta + L\tau)$, where f is determined by the communication scheme that achieve the global communication. For example, a tree broadcast on a row of processors is achieved in $t_{tree} = \lceil log_2 P_c \rceil.[\beta + L\tau]$, and a ring broadcast in $t_{oring} = (P_c - 1).[\beta + L\tau]$. The f function used to compute the total execution time is given in the expressions below.

4.1.2 Subroutines used in PDGETF2

- `IDAMAX`: is executed S_b times for each `PDGETF2` call. There are $\lceil \frac{M}{S_b} \rceil$ `PDGETF2` calls during the factorization (Figures 1 and 4). The execution time of a single `IDAMAX` call is linear.

$$T_{idamax} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil} S_b.[a_{idamax}(S_b \times \lceil \frac{i}{P_r} \rceil) + b_{idamax}]$$

- `DGMAX2D`: is executed S_b times for each `PDGETF2` call. The execution time of a single `DGMAX2D` call is linear but proportional to the number of processors

$$T_{dgmax} = \lceil \frac{M}{S_b} \rceil.S_b.f[a_{sendd} + b_{sendd}] \text{ [1] with } f = \lceil log_2 P_r \rceil$$

- `DCOPY`: is executed $2 \times S_b$ times for each `PDGETF2` call, with a fixed data size S_b. It is first used to copy the current pivot segment to broadcast it, and secondly to copy the real pivot segment into local matrix. The execution time of a single `DCOPY` call is linear.

$$T_{dcopy} = 2.\lceil \frac{M}{S_b} \rceil.S_b.[a_{dcopy} S_b + b_{dcopy}]$$

- `DGEBS2D,DGEBR2D`: is executed S_b times for each `PDGETF2` call, with a fixed data size S_b.

$$T_{broadcastd} = \lceil \frac{M}{S_b} \rceil.S_b.f[a_{sendd} S_b + b_{sendd}] \text{ with } f = \lceil log_2 P_r \rceil$$

- `DGESD2D,DGERV2D`: is executed S_b times for each `PDGETF2` call, with a fixed data size S_b.

$$T_{send} = \lceil \frac{M}{S_b} \rceil.S_b.f[a_{sendd} S_b + b_{sendd}]$$

- `DSCAL`: is executed S_b times for each `PDGETF2` call. The execution time of a single `DSCAL` call is linear.

$$T_{dscal} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil} S_b.[a_{dscal}(S_b \times \lceil \frac{i}{P_r} \rceil) + b_{dscal}]$$

- `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. It is a level 2 BLAS which computes $A \leftarrow \alpha x.y^t + A$ with $A_{(m \times n)}$:

$$t_{dger}(m, n) = (a_{dger} m + b_{dger}).n + (c_{dger} m + d_{dger})$$

$$T_{dger} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil} \sum_{j=1}^{S_b - 1} [(a_{dger}(S_b \times \lceil \frac{i}{P_r} \rceil) + b_{dger}).j + (c_{dger}(S_b \times \lceil \frac{i}{P_r} \rceil) + d_{dger})]$$

[1] $a_{sendd} = \tau$ and $b_{sendd} = \beta$. These parameters are for double precision buffer send. For integers, the a_{sendi} notation is used.

4.1.3 Subroutines used in PDGETRF

- `IGEBS2D,IGEBR2D`: is executed $\lceil \frac{M}{S_b} \rceil$ time, with a fixed data size S_b.

  $T_{broadcasti} = \lceil \frac{M}{S_b} \rceil .f[a_{sendi} S_b + b_{sendi}]$ with $f = \lceil log_2 P_c \rceil$

- `PDLASWP`: is executed $\lceil \frac{M}{S_b} \rceil$ time, but exchanges S_b rows at one time. We assume that there is a neglectful number of times for which the pivot is at the right place. Furthermore, the probability to find the pivot on the same processor that owns the current row is considered uniform. In the expression below, N_{mem} is the number of times where the rows are swapped within local memory, and N_{com} when the rows are swapped by communications. These parameters can be computed precisely for a given M, S_b, and P_r.

  $T_{swap} = N_{mem}.[a_{dswap}(S_b \times \left\lceil \frac{\lceil \frac{M}{S_b} \rceil}{P_r} \right\rceil) + b_{swap}] + N_{com}.[(2.a_{dcopy} + a_{sendd})(S_b \times \left\lceil \frac{\lceil \frac{M}{S_b} \rceil}{P_r} \right\rceil) + (2.b_{dcopy} + b_{sendd})]$

- `PBDTRSM`: is executed $\lceil \frac{M}{S_b} \rceil - 1$ times, with a variable data size, multiple of S_b. It must be decomposed in two parts:

  1. **broadcasting:** the factored column can be broadcast in one chunk in order to distribute the current L_{00} block (see figure 2) for `DTRSM` computation, and L_{10} for later `DGEMM` computation. Before broadcasting, the whole column is copied in a working buffer.

     $T_{broadcastd} = \sum_{i=2}^{\lceil \frac{M}{S_b} \rceil} [(f(a_{sendd}) + a_{dcopy})(S_b^2 \times \lceil \frac{i}{P_r} \rceil) + (f(b_{sendd}) + b_{dcopy})]$ with $f = \lceil log_2 P_c \rceil$

  2. `DTRSM`: The execution time of a single `DTRSM` call is cubic. It is a level 3 BLAS which computes $B \leftarrow \alpha.A^{-1}.B$ with $A_{(m \times m)}$ and $B_{(m \times n)}$:
     $t_{dtrsm}(m,n) = (a_{dtrsm} n + b_{dtrsm}).m^2 + (c_{dtrsm} n + d_{dtrsm}).m + (e_{dtrsm} n + f_{dtrsm})$

$$\begin{aligned} T_{DTRSM} &= \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil - 1} [(a_{dtrsm}.(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + b_{dtrsm}).S_b^2 + (c_{dtrsm}.(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + d_{dtrsm}).S_b + \\ & (e_{dtrsm}.(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + f_{dtrsm})] \end{aligned}$$

- `PBDGEMM`: executed $\lceil \frac{M}{S_b} \rceil - 1$ time, with a variable data size, multiple of S_b. It must be decomposed in two parts:

  1. **broadcasting:** the last factored row (see `DTRSM`) must be broadcast in order to achieve `DGEMM` computation. Before broadcasting, the whole row is copied in a working buffer.

     $T_{broadcastd} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil - 1} [(f(a_{sendd}) + a_{dcopy})(S_b^2 \times \lceil \frac{i}{P_c} \rceil) + (f(b_{sendd}) + b_{dcopy})]$ with $f = \lceil log_2 P_r \rceil$

  2. `DGEMM`: The execution time of a single `DGEMM` call is cubic. It is a level 3 BLAS which computes $C \leftarrow \alpha.A.B + \beta.C$ with $A_{(m \times k)}$, $B_{(k \times n)}$ and $C_{(m \times n)}$:

$$\begin{aligned} t_{dgemm}(m,n,k) &= (a_{dgemm} m.n + b_{dgemm} m + c_{dgemm} n + d).k + \\ & (e_{dgemm} m.n + f_{dgemm} m + g_{dgemm} n + h) \end{aligned}$$

$$T_{DGEMM} = \sum_{i=1}^{\lceil \frac{M}{S_b} \rceil - 1} [(a_{dgemm}.S_b + b_{dgemm})(S_b \times \left\lceil \frac{i}{P_r} \right\rceil).(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + (c_{dgemm}.S_b + d_{dgemm})(S_b \times \left\lceil \frac{i}{P_r} \right\rceil) + (e_{dgemm}.S_b + f_{dgemm})(S_b \times \left\lceil \frac{i}{P_c} \right\rceil) + (g_{dgemm}.S_b + h_{dgemm})]$$

In order to compute the optimal block size, all the precedent equations have been derivated over S_b. The total complexity optimum is obtain 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. Only one solution was positive each time. Results are given in Tables 1 and 3. It gives the optimal block size as a function of different grid sizes and matrix sizes (1000 signifies a 1000×1000 matrix in double precision).

4.1.4 Results on an Intel Paragon

The Intel Paragon is a distributed memory machine based on i860 processors connected via a 2D grid. Each processor reaches about 50 Mflops at peak performance.

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
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 a Paragon system.

type	**matrix size**			
	1000	2000	3000	4000
experimental	8 - 9	8	8	10
theoretical	8.31	9.28	9.64	9.83

Table 2: Comparison between theoretical and experimental block sizes on a Paragon system.

Five important points have to be noticed:

- Theoretical optimal block sizes are close to the experimental ones. They range between 7.2 and 10.1 and actual tests give an optimal block size of 8 or 10 on a 4×4 grid[2].
- The theoretical optimal block size is a function of the matrix size, and it raises up to a top value around 10. But this size is a real. A "good" block size is the nearest integer value from the theoretical one.
- The optimal block size hardly depends on the number of processors. Asymptotic values are 9 for 256 processors and 10 for 16. Again, experimental results confirm this point.

[2] 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.

- According to experimental results, the grid shape has no real influence on the optimal block size, though it greatly influences the execution time [9]. 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).

4.1.5 Results on an IBM SP2

The IBM SP2 is also a distributed memory machine based on RS6000 processors connected via a multistage network. Each processor reaches 265 Mflops at peak performance.

grid size	**matrix size**							
	256	512	768	1024	1280	1536	1792	2048
2×2	25.5	29.8	32.8	35.5	37.9	40.2	42.3	44.4
3×3	24.1	27.5	29.7	31.6	33.3	35.0	36.6	38.0
4×4	23.5	26.4	28.2	29.7	31.0	32.3	33.5	34.7

Table 3: Optimal block size on an IBM SP2.

type	**matrix size**							
	256	512	768	1024	1280	1536	1792	2048
3×3 exp.	27	33	30	35	36	35	38	38
3×3 theo.	24.1	27.5	29.7	31.6	33.3	35.0	36.6	38.0
1×9 exp.	27	25	24	25	26	27	26	27
1×9 theo.	20.3	21.7	22.7	23.7	24.6	25.4	26.3	27.1

Table 4: Comparison between theoretical and experimental block sizes on an IBM SP2.

Three important points have to be noticed :

- The experimental block sizes are quite difficult to obtain because the execution time can vary between two executions. The results in Table 4 are the average optimal block sizes given by the executions.
- Some experimental values are quite far from the theoretical optimum. But in most cases, this optimum gives a performance very close from the best experimental result. This is particularly true for grids smaller than 2×4. For example, on a 2×2 grid with a matrix size of 1024, the theoretical value is 35 and the experimental is 20. The performances are respectively 280 and 270 Mflops which represents a gap of 3.7%.
- On the SP2, the optimal block size is very dependent from the grid size and the matrix size. This justifies the interest of the complexity analysis.

4.2 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 wide range 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.

4.2.1 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 broadcast 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 we gain at each step i of factorization the broadcast 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). Moreover, this deceiving result can be predicted by the complexity analysis. The sum of broadcasting time for all steps never exceeds 2 percents of total execution time.

4.2.2 Rows Pivoting Overlap

It appears that a good speed-up cannot be obtained unless 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 (`PDGETF2`), 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 7 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 `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

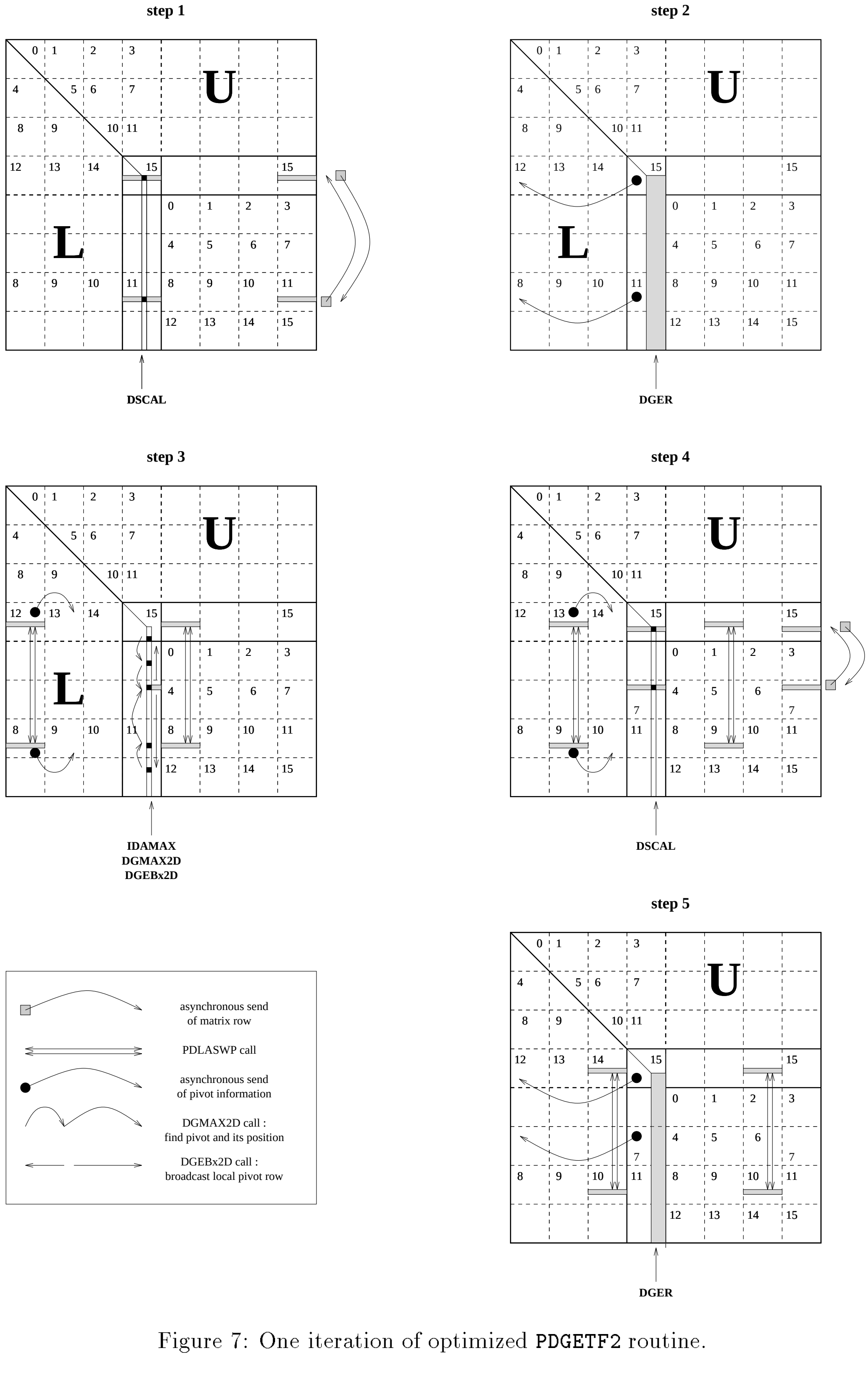

Figure 7: One iteration of optimized PDGETF2 routine.

from the processors row. In the figure 7, 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.

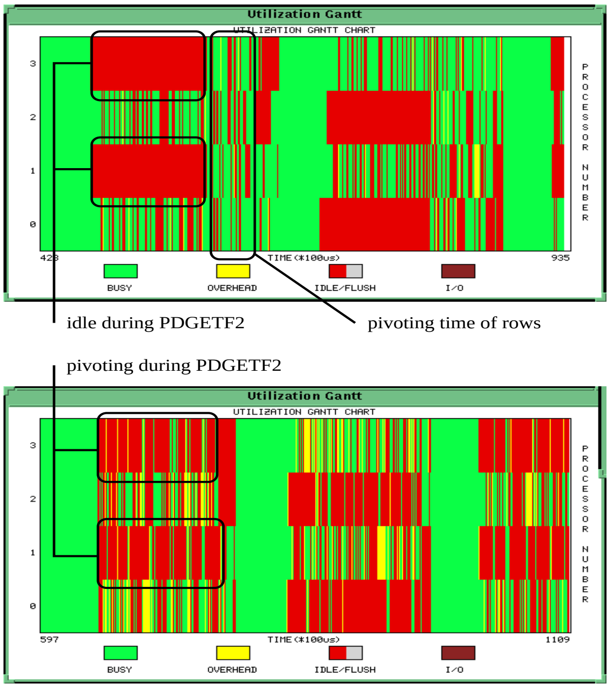

Figure 8: Comparison of the Gantt diagram for optimized and non-optimized versions.

The impact of this optimization is clearly shown on Figure 8. On the top of the figure, we can see that processors 1 and 3 are completely idle during the non-optimized `PDGETF2` execution. After, there is a communication phase to exchange the pivot row. On the bottom of the figure, processors 1 and 3 are no more idle : they are receiving the pivot information and exchanging rows during the optimized `PDGETF2` execution. Thus, there are no more communication post-phase. They are overlapped by `PDGETF2`.

4.3 Experimental Results

All experimental results have been produced on two Paragon systems and a SP2 system, with various grid size. The two Paragon (ORNL, Tennessee USA, and Lyon, France) have not the same 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). The 16 processors SP@ system is located at the LaBRI (Bordeaux, France).

All the results have been successfully run with the test driver included in the ScaLAPACK *LU* factorization package in order to test the correctness of the optimizations. This driver completes the *LU* factorization on a matrix A and then solves the system $LUx = B$. It also provides different possibilities to vary the execution parameters such as block size, number of processors, number of columns of the right-hand-side (RHS) matrix B, ...

The results on Paragon systems are shown on Figures 9, 10, 11 and 12. The IBM SP2 results are on Figures 14 and 13.

Figure 9 shows a comparison between optimized and non-optimized results for 16×16 grid. The optimized version grows faster (in Mflops) than the non-optimized, before becoming almost parallel. Indeed, the optimization works better for small matrix sizes ($< 375 \times P_c$ with P_c, the number of processor columns) because the pivoting communications are all executed during `PDGETF2`. After this limit, the pivoting time becomes more important and communications cannot be completely overlapped by the `PDGETF2` execution time. This is confirmed by the complexity analysis. The Figure 10 shows the predicted and the experimental gain for a 4×4 grid. For each matrix size, the predicted gain is given by the minimum of the total time of `PDGETF2` and the total communication time for pivoting.

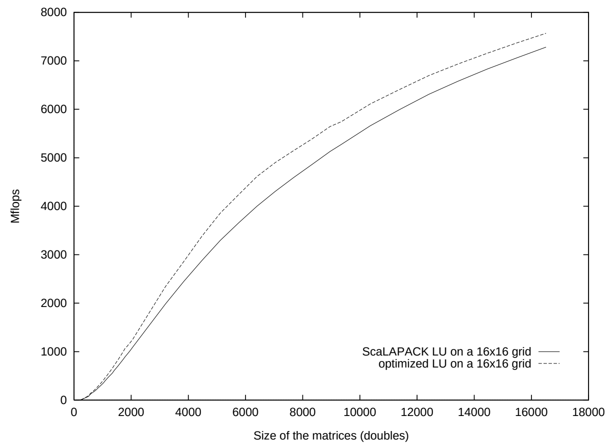

Figure 9: Experimental results on a 16×16 grid, on a Paragon.

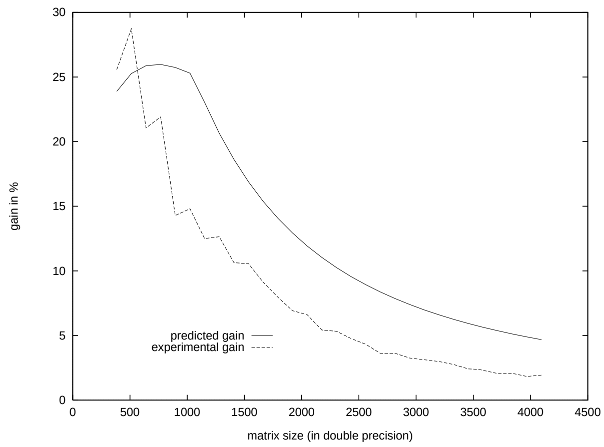

Figure 10: Predicted and experimental gain on a 4×4 grid, on a Paragon.

Figure 11 shows the gain in percents over 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 gain progressively decreases for a matrix size greater than $375 \times P_c$.

Figure 12 is the same as figure 11 but the x-coordinates are expressed as a function of the sub-matrix size for a single processor. It shows that the gain does not depend on the total matrix size but only on the size of the sub-matrix that one processor owns. This is due to the block scattered distribution which provides a good load balancing.

Figure 13 shows a comparison between the optimized and non-optimized results for a 3×3 grid on a IBM SP2. For small matrices, the optimized version works much faster than the basic version because the pivoting is completely overlapped by the `PDGETF2` execution. For large matrices, there is no gain since the pivoting is hardly overlapped. The gain can even be negative as shown on Figure 14. In fact, asynchronous communications are often slower than blocking communications, especially for large messages. Consequently, the asynchronous pivoting is often slower than in the basic version and it leads to worse performances when it is hardly overlapped.

5 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 big communication phases. Furthermore, it allows to

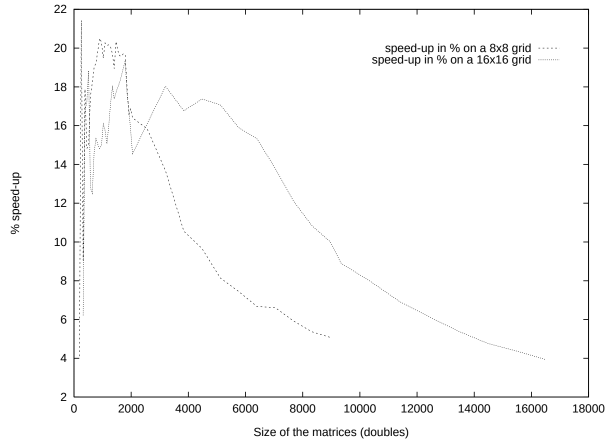

Figure 11: Gain on the 8×8 and 16×16 grids, on a Paragon

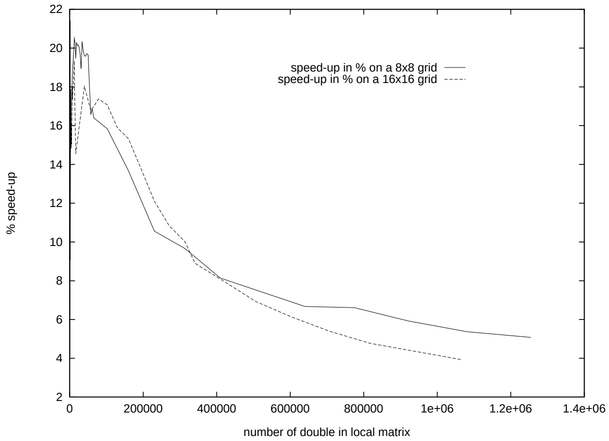

Figure 12: Gain on the 8×8 and 16×16 grids, as a function of the matrix size on a single processor, on a Paragon.

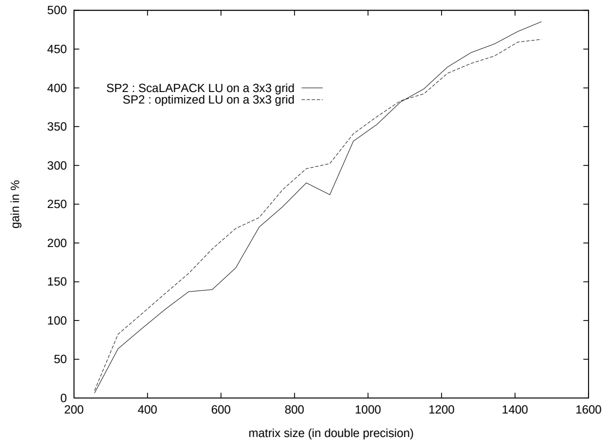

Figure 13: Experimental results on a 3×3 grid, on an IBM SP2.

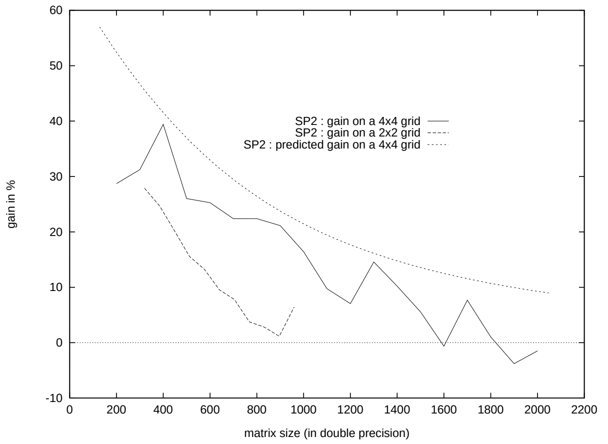

Figure 14: Gain on a 2×2 and 4×4 grid, on an IBM SP2.

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 as future work:

- All the updates have been done using Paragon system calls syntax because asynchronous BLACS do not exist. Consequently, our code is not fully portable. But the modifications are simple enough to be rewritten on any supercomputer.It will soon be ported on a CRAY T3D.
- The gain decreases as the matrix size grows: the `PDGETF2` time becomes not big enough to overlap the communications due to pivoting. Thus, another routine could be used to overlap more communications (like `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. It would be interesting to include such a computation in an HPF compiler.

References

[1] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. Van de geijn. Lapack for distributed memory architecture progress report. In *Fifth SIAM Conference on Parallel Processing for Scientific Computing,*, 1991.

[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, 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.

[4] J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transaction on Mathematical Software*, 16(1):1–17, 1990.

[5] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Transaction on Mathematical Software*, 14(1):1–17, March 1988.

[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 National Laboratory, July 1992.

[7] J.J. Dongarra, R.A. Van De Geijn, and R.C. Whaley. A User's Guide to the BLACS, March 1993.

[8] G.A. Geist and M.T. Heath. Matrix Factorization on a Hypercube Multiprocessor. Technical report, Oak Ridge National Laboratory, 1985.

[9] 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, Department of Computer Science - University of Tennessee, 1995.

[10] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transaction on Mathematical Software*, 5:308–323, 1979.

[11] 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.