Parallel out-of-core matrix inversion

Eddy Caron, Gil Utard

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

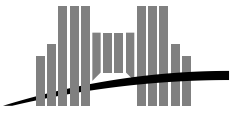
CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Parallel Out-of-Core Matrix Inversion

Eddy Caron
Gil Utard

Janvier 2002

Research Report N° 2002-04



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

INRIA

Parallel Out-of-Core Matrix Inversion

Eddy Caron
Gil Utard

Janvier 2002

Abstract

This paper presents a parallel out-of-core algorithm to invert huge matrices, that is when size of matrices is larger than the available physical memory by one or more orders of magnitude. Preliminary performance results are shown for a commodity cluster. An accurate prediction performance model of the algorithm is given. Thanks to the prediction model, optimizations which avoid the overhead of the out-of-core algorithm are derived. Performances of the optimized algorithm using a $O(N)$ memory size are similar to the performances of the best known parallel in-core algorithm using a $O(N^2)$ memory size (where N is the matrix order). There is no memory restriction for inversion of huge matrices!

Keywords: Data-parallelism, out-of-core, matrix factorization

Résumé

Ce papier présente un algorithme out-of-core pour l'inversion de matrices de taille supérieure à la capacité mémoire physique disponible. Une modélisation de l'algorithme est proposée dans ce rapport. Cette modélisation est ensuite validée par des expérimentations menées sur une architecture de type grappe. Outre, le fait que ce modèle nous permet de prédire les temps d'exécution, nous pouvons également extraire les surcoûts *out-of-core* et proposers des optimisations pour en éviter les effets. L'algorithme ainsi optimisé utilise une taille mémoire en $O(N)$ pour des performances similaires au cas *in-core* qui utilise une taille mémoire en $O(N^2)$ (où N est l'ordre de la matrice). Il n'y a plus de limite mémoire pour l'inversion matricielle !

Mots-clés: Parallélisme de donnée, out-of-core, factorization de matrices

Parallel Out-of-Core Matrix Inversion*

Eddy Caron† and Gil Utard†

ReMaP/LIP
UMR CNRS - ENS Lyon - INRIA 5668
Ecole Normale Supérieure de Lyon
69364 Lyon Cedex 07
(Eddy.Caron,Gil.Utard)@ens-lyon.fr

17th January 2002

1 Introduction

Many of important computational applications involve solving problems with very large data sets [9]. For example astronomical simulation [10], crash test simulation [5], global climate modeling, and many other scientific and engineering problems can involve data sets that are too large to fit in main memory. Using parallelism can reduce the computation time and increase the available memory size, but for challenging applications the memory is always insufficient in size: for instance in a mesh decomposition of a mechanical problem, a scientist would like to increase accuracy by an increase of the mesh size. Those applications are referred as "parallel *out-of-core*" applications.

Matrix inversion is used by many applications as a direct method to solve linear systems. Thus, the importance of optimizing this routine has not to be proved because of the increasing demand of applications dealing with large matrices. To increase the available memory size, a trivial solution is to use the *virtual memory* mechanism present in modern operating system. Unfortunately, in [2] we shown this solution is inefficient if standard *paging policy* is employed. To get the best performances, the algorithm must be generally *restructured* with explicit I/O calls.

This paper presents a new algorithm for parallel *out-of-core* matrix inversion. This algorithm is derived of our works on the parallel out-of-core matrix factorization [3], where we revisited previous parallel out-of-core factorization algorithms [1, 11, 8]. We implemented our algorithm with the ScaLAPACK library. Preliminary performance results are shown for a commodity cluster. An accurate prediction performance model of the algorithm is given. Thanks to the prediction model, optimizations which avoid the overhead of the out-of-core algorithm are derived.

In Section 2 we describe the mathematical basis of the matrix inversion from LU factorization, and its parallelization. In Section 3, we present the parallel *out-of-core* algorithm. In Section 4 we derive a performance prediction model of the algorithm for commodity cluster. We show the accuracy of this model by experimentation. In Section 5 we analyze the overhead of the algorithm and show how to avoid it.

*This work is supported by a grant of the "Pôle de Modélisation de la Région Picardie".

†This work has been done while the author was at LaRIA, Amiens, France.

2 Matrix Inversion from LU factorization

The computation of the inverse of a matrix $A = (a_{ij})_{1 \leq i,j \leq N}$ can be derived from its LU factorization. The LU factorization of a matrix $A = (a_{ij})_{1 \leq i,j \leq N}$ is the decomposition of A as a product of two matrices $L = (l_{ij})_{1 \leq i,j \leq N}$ and $U = (u_{ij})_{1 \leq i,j \leq N}$, such that $A = LU$ where L is *lower triangular* (i.e. $l_{ij} = 0$ for $1 \leq j < i \leq N$) and U is *upper triangular* (i.e. $u_{ij} = 0$ for $1 \leq i < j \leq N$).

From this decomposition, the inverse matrix A^{-1} can be obtained by two *triangular solve* (or *backward substitution*). Let I be the identity matrix : $AA^{-1} = I$ implies $LUA^{-1} = I$. Let $Y = UA^{-1}$, Y is determined by backward substitution such that $LY = I$. The inverse matrix A^{-1} is determined by a second backward substitution $UA^{-1} = Y$.

Generally, for numerical stability, the LU decomposition is computed using partial pivoting. The pivoting information is represented by a permutation matrix P, such that $PA = LU$. So A^{-1} is derived from the permutation of the identity matrix PI.

First, we present the LU factorization and the backward substitution by block, then their parallelization in ScaLAPACK.

2.1 Blocked LU factorization

A well known method for parallelization of the LU factorization is based on the *blocked right-looking* algorithm. This algorithm is based on a *block decomposition* of matrices A, L and U:

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

This block decomposition gives the following equations:

$$A_{00} = L_{00}U_{00} \quad (1) \qquad A_{10} = L_{10}U_{00} \quad (3)$$
$$A_{01} = L_{00}U_{01} \quad (2) \qquad A_{11} = L_{10}U_{01} + L_{11}U_{11} \quad (4)$$

These equations lead to the following recursive algorithm:

1. Compute the factorization $A_{00} = L_{00}U_{00}$ in equation (1) (may be by another method).
2. Compute L_{01} (resp. U_{10}) from equation (2) (resp. (3)). This computation can be done by triangular solve (L_{00} and U_{00} are triangular).
3. Compute L_{11} and U_{11} from equation (4):
   (a) Compute the new matrix $A' = A_{11} - L_{10}U_{01}$.
   (b) Recursively factorize $A' = L_{11}U_{11}$.

This algorithm is called *right-looking* because once new matrix A' is computed, the left part (L_{00} and L_{01}) of the matrix is not used in the recursive computation. It is also true for the upper part (U_{00} and U_{10}). Moreover, it is easy to show that this computation can be done *data in place*: only one array is necessary to hold initial matrix A and resulting matrices L and U.

For numerical stability, *partial pivoting* (generally row pivoting) is introduced in the computation. Then, the result of the factorization is matrices L and U plus the permutation matrix P such that $PA = LU$.

In right looking algorithm with partial pivoting, the factorization of A_{00} and the computation of L_{01} are merged in the first step. For the sake of presentation, we present an algorithm with partial pivoting (data in place) where row interchanges are applied in two stages.

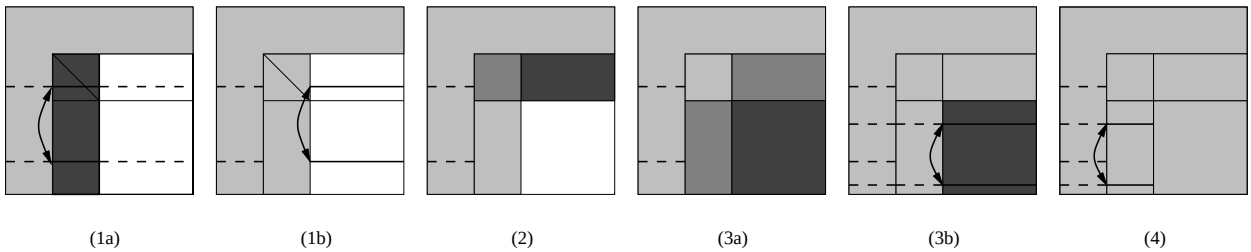


Figure 1: A recursive call to the right-looking algorithm. Horizontal lines represent pivoting. Dashed lines represent part of rows which are not yet pivoted.

1a. Compute factorization $P \begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix} = \begin{pmatrix} L_{00} \\ L_{10} \end{pmatrix} U_{00}$ where P is permutation matrix which represents partial pivoting: the left part of matrix A (i.e. $\begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix}$) is factorized.

1b. Apply pivot P to the right part of matrix A (i.e. $\begin{pmatrix} A_{01} \\ A_{11} \end{pmatrix}$)

2. Compute U_{01} from equation (2).

3a. Compute new matrix $A' = A_{11} - L_{10}U_{01}$.

3b. Compute L_{11}, U_{11} and P' by a recursive call of factorization $P'A' = L_{11}U_{11}$ (P' is the permutation matrix.)

4. Apply pivot P' to the lower left part of matrix A (i.e. the L_{10} computed in the first step). Finally, return the composition of P and P'.

Figure 1 shows the different steps for the second recursive call of the right-looking factorization:

2.2 Blocked triangular substitution

This approach is also based on block decomposition of matrices. Consider

$$\begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{pmatrix} = \begin{pmatrix} I_{00} & I_{01} \\ I_{10} & I_{11} \end{pmatrix}$$

This block decomposition gives the following equations:

$$L_{00}X_{00} = I_{00} \quad (5) \qquad L_{10}X_{00} + L_{11}X_{10} = I_{10} \quad (7)$$
$$L_{00}X_{01} = I_{01} \quad (6) \qquad L_{10}X_{01} + L_{11}X_{11} = I_{11} \quad (8)$$

This equations lead to the following recursive algorithm:

1. Solve the triangular resolution $L_{00}X_{00} = I_{00}$ (equation (5)) and solve $L_{00}X_{01} = I_{01}$ (equation (6)).

2. Determine X_{10} from equation (7) and (8):

   - Compute matrices $I'_{10} = I_{10} - L_{10}X_{00}$, and $I'_{11} = I_{11} - L_{10}X_{01}$

3. Solve recursively $L_{11}X_{10} = I'_{10}$ and $L_{11}X_{11} = I'_{11}$

2.3 Parallelization in ScaLAPACK

In ScaLAPACK the parallel LU factorization corresponds to the `pdgetrf` function. This parallelization is based on a data-parallel approach: the matrix is distributed on processors and the computation is distributed according to the *owner compute rule*. The matrix is decomposed in $k \times k$ blocks.

As noticed above, at each recursive application of the right looking algorithm, the left and upper part of the matrix is factorized (modulo a permutation in the lower left part of the matrix). So, for *load balancing*, a cyclic distribution of the data is used.

As shown in figure 2, the matrix is distributed *block cyclic* on a (virtual) grid of p rows and q columns of processors. The *block decomposition* of the algorithm (shown in Figure 1) is corresponding to the *block distribution* of the matrix. So step 1a of the algorithm is computed by one column of p processors; step 2 is computed by one row of the q processors ; step 3a is computed by the whole grid. Pivoting step 1b (resp. 4) is executed concurrently by computation step 1a (resp. 3b).

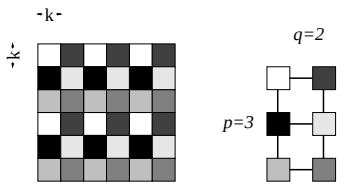


Figure 2: Data-distribution for ScaLAPACK. $k \times k$ blocks are cyclically distributed among $p \times q$ processors .

Now let us describe more precisely the different steps of the algorithm. Step 1a is implemented by ScaLAPACK function `pdgetf2`, which factorize a block of columns. For each diagonal element of the upper block (i.e. A_{00}) the following operations are applied:

1. determine pivot by a *reduce* communication primitive and *exchange* the pivot row with the current row;
2. *broadcast* the pivot row on columns of processors;
3. *scale*, i.e. divide, the column under pivot by the pivot value and update the matrix elements on the right of the column.

Step 2 of the algorithm is implemented by ScaLAPACK function `pdtrsm`: the left-upper block (i.e. A_{00}) is *broadcast* to the processors line followed by a (BLAS) triangular solve.

Step 3a is implemented by ScaLAPACK function `pdgemm`: the blocks corresponding to U_{01} are broadcast on columns (of processors); the blocks corresponding to L_{10} are broadcast on rows (of processors); then the blocks are multiplied to update A'.

In ScaLAPACK the function `pgdtrsm` implement the parallel substitution. The matrix is distributed like the LU decomposition (Figure 2). Step 1 of the algorithm is computed by one row of q processors; step 2 is computed by the whole grid.

The Figure 3 describes the different steps of the algorithm. Step 1 is implemented by ScaLAPACK function `pdtrsm`, which perform a parallel triangular substitution with a square matrix. First the algorithm broadcast the data needed for the triangular solve (step 1a) and compute it (step 1b). In another words this step solve equations (5) and (6). Step 2 is called by ScaLAPACK with function `pdgemm`, which upgrades the rest of the matrix by multiplication. We can see again two parts, data communication part (step 2a) and the computing part (step 2b). This step compute equations (7) and (8) in the same time.

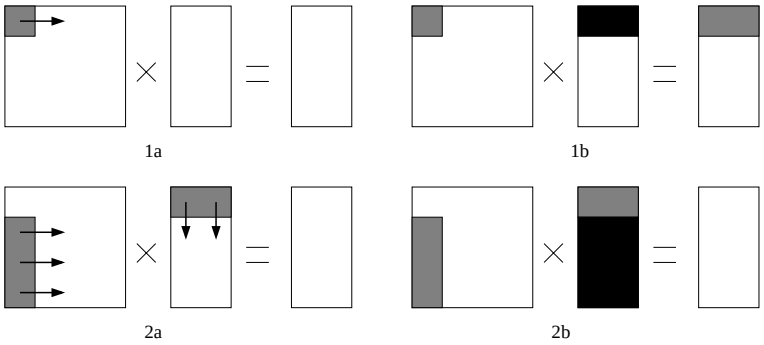


Figure 3: One stage of block triangular solve *(forward substitution).*

3 Parallel out-of-core algorithm

Now, we consider the situation where matrix A is too large to fit in main memory. First we present the parallel out-of-core LU factorization (*left-right looking* algorithm) and the parallel out-of-core triangular substitution.

3.1 Parallel out-of-core LU factorization

We present the parallel out-of-core left right looking LU factorization algorithm used by the ScaLAPACK routine `pfdgetrf` for parallel out-of-core LU factorization [6]. Similar algorithms are also described in [11, 8]. In the algorithm the matrix is divided in blocks of columns called *superblocks*. The width of the superblock is determined by the amount of physical available memory.

Like the previous parallel algorithm, the matrix is logically block cyclically distributed on the $p \times q$ grid of processors. But only blocks of the current superblock are in main memory, the others are on disk.

The parallel out-of-core algorithm is an extension of the parallel in-core algorithm. It factorizes the matrix from left to right, superblock by superblock. Each time a new superblock of the matrix is fetched in memory (called the *active* superblock), all previous pivoting and update of a *history of the right-looking algorithm* are applied to the active superblock. To do this update, each superblock lying on the left of the active superblock is read again. Once the update is finished, the right-looking algorithm resumes on the updated superblock, and the factorized active superblock is written on the disk. Once the last superblock is factorized, the matrix is read again to apply the remaining row pivoting of the recursive phases (step 4).

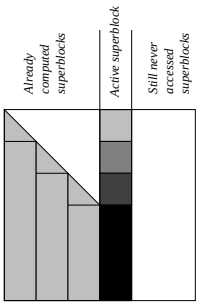


Figure 4: Superblock updating for left-looking out-of-core algorithm

The update of each active superblock is summarized in Figure 4. When a left superblock is considered (called the *current* superblock), the update consists in applying row pivoting to the active superblock and:

1'. reading the under-diagonal part of a current superblock;

2'. computing the U_{01} part of the active superblock by a triangular solve (function `pdtrsm`);

3'. updating A_{11}, i.e. sub the product of U_{01} part of the active superblock by L_{10} of the current superblock (function `pdgemm`).

3.2 Parallel out-of-core triangular substitution

Superblock decomposition of the matrix is also used. The algorithm performs this substitution between two out-of-core matrices. The X matrix is computed superblock per superblock. For each superblock, the whole L matrix is also read superblock per superblock. In another words for each superblock of X the algorithm performs a triangular *out-of-core* substitution with L.

Let L_0, L_1,...,L_B the superblocks of the matrix L. Let X_0, X_1,...,X_B the superblocks of the matrix X. The Figure 5 shows a *macroscopic view* of one step of the out-of-core triangular substitution. For each superblock i of X and I, the parallel computation described in this figure is applied for each superblock j of L. Note that in the real program X and I are the same matrix.

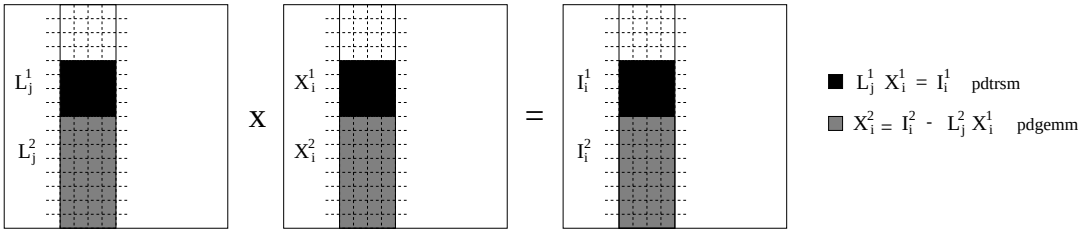


Figure 5: Macroscopic view of one step the out-of-core triangular substitution

The parallel computation done in this step is similar to the original parallel triangular substitution (section 2.2). The difference is the matrix is rectangular instead of square. In first, the algorithm perform a parallel triangular substitution on the diagonal $L_j^1 X_i^1 = I_i^1$. Using the parallel algorithm previously described in section 2.2. In second, the algorithm compute a parallel matrix multiplication to upgrade X_i^2, with $X_i^2 = I_i^2 - L_j^2 X_i^1$.

4 Performances Prediction

In this section we present an architectural model for cluster and a execution time prediction of the parallel out-of-core matrix inversion algorithm previously described.

4.1 Architectural Model

Our architectural model of a cluster is a distributed memory machine with an interconnection network and one disk on each node. Each node stores its blocks on its own disk. Let characterize this kind of architecture by some constants representing the computation time, the communication time and the IO time.

Computation time. It is usually based on the time required for the computation of one floating point operation on one processor and is represented by a constant α. In fact, this time is not constant and depends on processor memory hierarchy and on the kind of computation. For instance a matrix multiplication algorithm exhibits good cache reuse whereas product of a vector by a scale has poor temporal locality. So we distinguish three times for floating point operations which appear in the algorithm: α_g for matrix multiplication, α_t for triangular solve, and α_s for scaling of vectors.

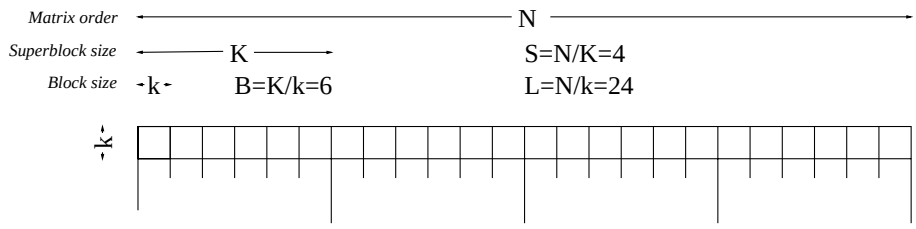


Figure 6: Data-distribution for out-of-core LU decomposition

Communication time. As usual, the communication time is represented by the $\beta + V\tau$ model, where β is the startup time and τ is the time to transmit one unit of datum and V is the volume of data to communicate. We consider only broadcast communication in our model. The constants β and τ are dependent on the topology of the virtual grid: β_p^q is the startup time for a column of p processors to broadcast data[1] on their rows, and $1/\tau_p^q$ represents the throughput. Similarly β_q^p and τ_q^p denote time for one row of q processors to broadcast data on their columns. These functions depend on the communication network. For instance, for a cluster of workstations with a switch, the broadcast can be implemented by a tree diffusion. Then $\beta_p^q = \log_2 q \times \beta$ and $\tau_p^q = \log_2 q \times \frac{\tau}{p}$ where β is the startup communication time for one node and $1/\tau$ the throughput of the medium. With a hub (i.e. a bus), the model is: $\beta_p^q = p(q-1) \times \beta$ and $\tau_p^q = \tau$ if $q > 1$, $\tau_p^q = 0$ if $q = 1$.

IO time. The IO time is based on the throughput of a disk. Let τ^{io} be the time to read or write one word for one disk, then $\tau_p^{io} = \frac{\tau^{io}}{p}$ is the time to read or write p words in parallel for p independent disks.

4.2 Modeling

To model the algorithm, we estimate the time used by each function. For each function, we distinguish computation time and communication time, and we distinguish the intrinsic cost time of the parallel right-looking algorithm and cost time introduced by the out-of-core extension.

Let N be the matrix order, K be the column width of superblock, the block size is $k \times k$. The grid of processors is composed of p rows of q columns. We have the following constraints for the different constants: N is multiple of K and K is multiple of k and q. Let $L = \frac{N}{k}$ be the block width of the matrix, $S = \frac{N}{K}$ the number of superblocks, and $B = \frac{K}{k}$ be the block width of a superblock. The Figure 7 shows the blocks used in the three main functions for a fixed active and current superblock.

In the next section we recall performance prediction model for the out-of-core parallel LU factorization presented in [3]. Then we present a performance prediction model for the parallel out-of-core triangular substitution.

4.3 LU Factorization modeling

The Figure 8 collects costs of the different steps of the algorithm. For the sake of simplicity, we don't consider the pivoting cost in our analysis. This cost is mainly the cost of reduce operations for each element of the diagonal and the cost of row interchanges, plus the cost of re-read/write of the matrix. This time can be easily integrated in the analysis if necessary.

[1] Data are equi-distributed on processors.

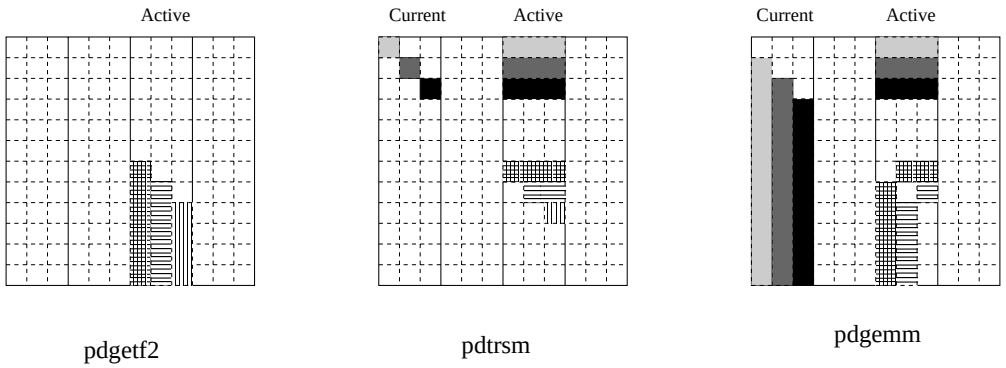


Figure 7: Blocks involved in the 3 main functions of the factorization. Figure shows blocks from the active superblock and from a current one (there are 4 superblocks).

`pdgetf2` **cost :** Step 1 of the algorithm (ScaLAPACK function `pdgetf2`) is applied on block columns (of width k) under the diagonal. There are L such blocks. This computation is independent of the superblock size. For the computation cost, we distinguish the computation of blocks on the diagonal (9) and the computation on the blocks under the diagonal (10). The total computation time for `pdgetf2` function for the whole $N \times N$ matrix is (11). For communications in `pdgetf2`, for each block on the diagonal and for each element on the diagonal, the right part is broadcasted to the processor column (12).

`pdtrsm` **cost :** Step 2 of the algorithm (computation of U_{01}) is applied on each block of row lying on right the diagonal: there is a triangular solve for each. The computation cost of a triangular solve between two blocks of size $k \times k$ is $\alpha_t k^3$. The total computation cost for every `pdtrsm` applied by the algorithm is (13). The communication cost for `pdtrsm` is the broadcast of diagonal blocks onto the processors row. One broadcast is done during the factorization of the active superblock (14), and another one is needed during the future updates (15).

`pdgemm` **cost.** Step 3 of the algorithm updates the trailing sub-matrix A'. The computation is mainly matrix multiply plus broadcast. For a trailing sub-matrix of order H, there are $(\frac{H}{k})^2$ block multiplications of size $k \times k$. The cost of such a multiplication is $2\alpha_g k^3$. The total computation cost is (16). For the communication cost, we distinguish cost of factorization of the active superblock and cost of the update of the active superblock. For the factorization of the superblock, the cost is the broadcast of one row of blocks and the broadcast of one columns of blocks (17). The Figure 4 illustrates the successive updates for an active superblock. Each block of column under diagonal in left superblock read are broadcasted (18). In the same time symmetric row of blocks of the current superblock are broadcasted (19).

IO cost : The IO cost corresponds to the read/write of the active superblock (20) and the read of left superblocks (21).

4.4 Parallel out-of-core substitution modeling

After the LU factorization, the algorithm makes two out-of-core triangular substitutions. The cost of forward and backward substitution are the same. We give the cost for one triangular substitution. Costs are sum up on Figure 9

pdgetf2:

$$\alpha_s \sum_{j=1}^{j=k} \Big(\sum_{i=1}^{i=j-1} (2i-2) + \sum_{i=j+1}^{j=k} (2j-1) \Big) \quad (9)$$

$$+ \frac{\alpha_s}{p} \sum_{j=1}^{j=L} \Big(k \times j + j \sum_{i=2}^{i=k} 2(i-1) \Big) \quad (10)$$

$$= \frac{\alpha_s}{p} \Big(\frac{Nk^2}{6} + \frac{N^2 k}{2} - \frac{Nk}{2} - \frac{N}{6} \Big) \quad (11)$$

$$L \times \Big(k\beta_1^p + \frac{k(k+1)}{2} \tau_1^p \Big) = N \times (\beta_1^p + \frac{(k+1)}{2} \tau_1^p) \quad (12)$$

pdtrsm:

$$\frac{1}{q} \sum_{i=1}^{i=L-1} \alpha_t k^3 \times i = \frac{\alpha_t}{2q} \times (N^2 k - N k^2) \quad (13)$$

$$(S(B-1) + \sum_{i=1}^{S-1} (i \times B)) \times (\beta_p^q + k^2 \tau_p^q)$$

$$= S(B-1)(\beta_p^q + k^2 \tau_p^q) + \quad (14)$$

$$\frac{S(S-1)}{2} (\beta_p^q + k^2 \tau_p^q) \quad (15)$$

pdgemm:

$$\frac{1}{pq} \sum_{i=1}^{i=L-1} i^2 \times 2\alpha_g k^3$$

$$= \frac{\alpha_g}{pq} \Big(\frac{N^3 + Nk^2}{3} - N^2 k \Big) \quad (16)$$

$$S((B-1)\beta_q^p + \frac{B(B-1)}{2} k^2 \tau_q^p)$$

$$+ \sum_{i=1}^{i=S} ((B-1)\beta_p^q + \frac{B(B-1)}{2} k^2 \tau_p^q$$

$$+ (i-1)B(B-1)k^2 \tau_p^q) \quad (17)$$

$$\sum_{i=1}^{i=S-1} i \times (B\beta_p^q + \frac{B(B-1)}{2} k^2 \tau_p^q + (i-1)B^2 k^2 \tau_p^q)$$

$$= \frac{N(N-K)(6\beta_p^q + (Kk + 4Nk - 3k^2)\tau_p^q)}{12Kk} \quad (18)$$

$$\frac{S(S-1)}{2} (B\beta_q^p + K^2 \tau_q^p) \quad (19)$$

IO:

$$2SNK\tau_{pq}^{io} \quad (20)$$

$$\sum_{i=1}^{i=S-1} i \times (\frac{B(B-1)}{2} k^2 \tau_{pq}^{io} + (i-1)B^2 k^2 \tau_{pq}^{io})$$

$$= \frac{N(N-K)((Kk + 4Nk - 3k^2)\tau_{pq}^{io})}{12Kk} \quad (21)$$

Figure 8: Costs of the different steps of the left-right looking algorithm for out-of-core LU factorization

.

pgdtrsm:

$$S^2 \Big(\frac{B.(k^2 K + kK - K)}{q} \alpha_t + \frac{\sum_{i=1}^{i=B-1} 2iKk^2}{pq} \alpha_g \Big) \quad (22)$$

$$S \sum_{i=1}^{i=B-1} \Big(\frac{k^2}{2} \tau_p^q + \beta_p^q \Big) \quad (23)$$

$$S(S-1) \sum_{i=1}^{i=B-1} \Big(\frac{k^2}{2} \tau_p^q + \beta_p^q \Big) \quad (24)$$

$$S^2 \sum_{i=1}^{i=B-1} \big(ik^2 \tau_p^q + \beta_p^q \big) \quad (25)$$

$$S^2 (B-1) \big(kK\tau_q^p + \beta_q^p \big) \quad (26)$$

pdgemm:

$$S \sum_{i=1}^{i=S-1} \frac{(2iK^3)}{pq} \alpha_g \quad (27)$$

$$\sum_{i=1}^{i=S-1} \big(iBkK\tau_p^q + B\beta_p^q \big) \quad (28)$$

$$(S-1) \sum_{i=1}^{i=S-1} \big(iBkK\tau_p^q + B\beta_p^q \big) \quad (29)$$

$$S \sum_{i=1}^{i=S-1} (BkK\tau_q^p + B\beta_q^p) \quad (30)$$

IO:

$$S(NK\tau_{pq}^{io} + \beta_{pq}^{io}) \quad (31)$$

$$2S \sum_{i=1}^{i=S} ((N - K(i-1))K\tau_{pq}^{io} + \beta_{pq}^{io}) \quad (32)$$

Figure 9: Costs of the different steps of parallel out-of-core substitution algorithm.

To perform triangular substitution, two main functions are called. Triangular substitution on square blocks (`pgdtrsm`) and upgrade the rest of the matrix by blocks multiplication (`pdgemm`). For each function we distinguish two parts: communication and computation.

`pgdtrsm` cost : The computation time for out-of-core triangular substitution is equal to the blocks in-core version. The algorithm performs S triangular substitution for each superblock of the first matrix, i.e. S^2 triangular substitution (22). The `pgdtrsm` communication consists of diagonal blocks broadcast to the row of processor. This block is send S^2 times. Thus, this cost is divided into the active superblocks communication (23) and the out-of-core overhead communication (24). The data communicated for upgrade the rest of matrix are broadcast through processors (25). The cost is like in-core version. The column broadcast is (26).

`pdgemm` cost : The upgrade multiplication is applied under the diagonal. For each L superblock, the height is decreased by K (27). Broadcasts needed for multiplication out-of-core upgrade are sent through the processors row. The size of this blocks is K for the width. The height is determined by the diagonal. We assume that broadcast is performed by blocks. The active blocks cost is (28) and the out-of-core overhead is (29). The row block $K \times K$ is broadcast on processors column like in in-core version (30).

IO cost : IO accesses correspond to reading and writing the superblocks of result matrix (31) and reading superblocks of matrix L and U (32).

4.5 Experimental validation of the analytical model

To validate our prediction model, we ran the ScaLAPACK out-of-core factorization program on a cluster of 8 PC-Celeron running Linux and interconnected by a Ethernet switch. Each node has 96 Mb of physical memory. The model described in the previous sections is instanced with the following constants (experimental measurements):

$$1/\alpha_g = 237\ Mflops, 1/\alpha_t = 123\ Mflops, 1/\alpha_s = 16\ Mflops,$$

$$\beta_p^q = \beta_q^p = 1.7\ ms, 1/\tau_p^q = 1/\tau_q^p = 1.1\ Mo/s, 1/\tau_{io} = 1.8\ Mo/s$$

The Figure 10 show the comparison between the running time and the predicted time (in *italics*) of the program. This Figure shows the comparison between the different functions of matrix inverse algorithm (for the triangular substitution we distinguished the communication and the computation time). For computation and communication, running time was close to the predicted time. There were some differences for IO times. It is mainly due to our rough model of IO: IO performances are more difficult to model because access file performances depend on the layout of the file on the disk (fragmentation).

5 Out-of-core overhead analysis

In comparison with the standard *in-core* algorithm, the overhead of the *out-of-core* algorithm is the extra IO cost and broadcasts (of columns) cost for the update of the active superblock: for each active superblock, left superblocks must be read and broadcast once again!

In [3], we demonstrated that with a correct distribution and with the overlap of IO by computation, it is possible to avoid the out-of-core overhead of the algorithm: we achieve the performance of

M	K	$p \times q$	LU	`pgdtrsm` IO	`pgdtrsm` Comm. Active	`pgdtrsm` Comm Update	`pgdtrsm` Computation	Execution time
12288	3072	1x8	29m39/*24m25* 693 Mflp/*844 Mflp*	10m16/*8m23*	3m6/*3m02*	8m13/*9m05*	21m21/*15m32*	1h36/*1h28* 856 Mflp/*928 Mflp*
1,2 Go / 4SB		2x4	21m17/*18m30* 983 Mflp/*1114 Mflp*	9m39/*8m23*	1m27/*1m32*	3m4/*3m25*	20m27/*15m41*	1h12/*1h08* 1128 Mflp/*1208 Mflp*
		4x2	18m50/*17m10* 1088 Mflp/*1200 Mflp*	8m52/*8m23*	2m5/*2m15*	2m25/*2m17*	19m50/*16m00*	1h08/*1h06* 1200 Mflp/*1232 Mflp*
		8x1	19m57/*18m51* 1019 Mflp/*1093 Mflp*	8m9/*8m23*	5m22/*5m57*	4m5/*4m33*	19m50/*16m38*	1h21/*1h22* 1016 Mflp/*1008 Mflp*
20480	2048	1x8	2h37/*2h08* 610 Mflp/*742 Mflp*	1h7/*46m37*	11m44/*8m21*	1h16/*1h15*	2h34/*1h11*	10h38/*8h06* 592 Mflp/*784 Mflp*
3,3 Go / 10SB		2x4	1h06/*1h29* 706 Mflp/*1067 Mflp*	1h2/*46m37*	4m28/*4m15*	27m27/*26m34*	2h05/*1h12*	6h23/*5h42* 992 Mflp/*1112 Mflp*
		4x2	1h50/*1h17* 1208 Mflp/*1238 Mflp*	1h2/*46m37*	5m53/*6m15*	11m55/*11m28*	1h56/*1h13*	6h21/*5h05* 1000 Mflp/*1248 Mflp*
		8x1	1h43/*1h18* 917 Mflp/*1212 Mflp*	55m46/*46m37*	13m45/*16m24*	13m41/*15m14*	1h49/*1h14*	6h25/*5h38* 984 Mflp/*1128 Mflp*
27648	1024	1x8	7h59/*7h58* 492 Mflp/*490 Mflp*	2h7/*3h25*	14m23/*14m59*	6h2/*6h38*	4h4/*2h56*	1d04h/*1d07h* 536 Mflp/*504 Mflp*
6,1 Go / 27SB		2x4	4h34/*4h51* 857 Mflp/*805 Mflp*	1h59/*3h25*	7m16/*7m45*	2h3/*2h15*	3h51/*2h57*	16h45/*18h58* 928 Mflp/*824 Mflp*
		4x2	3h20/*3h45* 1174 Mflp/*1040 Mflp*	2h2/*3h25*	9m50/*11m18*	39m41/*43m30*	3h47/*2h58*	12h43/*14h58* 1224 Mflp/*1040 Mflp*
		8x1	3h02/*3h34* 1290 Mflp/*1097 Mflp*	2h4/*3h25*	26m35/*29m12*	26m53/*30m00*	3h46/*3h02*	12h29/*15h02* 1248 Mflp/*1040 Mflp*

Figure 10: Comparison of experimental and theoretical (in *italic*) running time and performances of the out-of-core matrix inverse algorithm. M is the matrix order, K the superblock wide, p the number of row of the processor grid, q the number of columns, S is then number of superblocks. The size of the matrix in Gigabyte is given in the first column. Times are given in days (d), hours (h), minutes (m) and seconds (s).The last column shows the real and predicted performances in Mflops (Mflp).

the in-core one. In the following, we demonstrate this result hold for the parallel out-of-core matrix inversion.

This overhead cost of matrix inversion is represented by equations (24) and (29) for communications and (32) for IO. It is easy to show that if $K = N$ (i.e. $S = 1$) then this cost is equal to zero: it is the *in-core* algorithm execution time.

The overhead cost is $O(N^3)$, and is non negligible. In the following paragraph, we will show how to reduce this overhead cost. Let $O_C = (24) + (29)$ be the overhead communication cost and $O_{IO} = (32)$ be the overhead IO cost.

5.1 Reducing overhead communication cost

As shown by the model and experimental results, the topology of the grid of processors has a great influence on the overhead communication cost:

Fact 1 *If the number of columns q is equal to 1, then* $O_C = 0$*!*

If there is only one column of processors, there is no broadcast of column during the update. If we consider a communication model where broadcast cost is increasing with the number of processors, then the greater the number of columns is, the greater O_C is.

Figure 11 shows the influence of topology on the performances for the matrix inversion. In the same figure, there are plots for the predicted performances of the in-core algorithm. The constants are the constant of our small PC-Celeron cluster. The communication and IO costs for the update are prominent. The figure 12 shows the IO and communication ratio on the total execution time in this case.

5.2 Overlapping IO and computations

A trivial way to avoid the IO overhead is to overlap this IO by the computation. In [3], we presented an overlapping scheme for LU factorization. During updates of the active superblock, the left superblocks are read from left to right. An overlapping scheme is to read the next left superblock during the update of active superblock with the current one: if the time for this update is greater than the time for reading the next superblock, then the overhead IO cost is avoided.

In [3] we obtained the following result for the LU part. Let M be the amount of memory devoted to a superblock in one processor. For a matrix order N the width of a superblock is then $K = \frac{pgM}{N}$. Let O_{IO}^o the overhead IO cost not overlapped in this new scheme.

Theorem 1 *If the number of columns of processor q is equal to 1 and if*

$$pM \geq N \frac{t_{io}}{2\alpha}$$

then

$$O_{IO}^o = 0$$

We propose a similar overlapping scheme for the triangular substitution. When the algorithm perform the computation part, ie. triangular solve and matrix multiplication, between L_j, X_i and I_i, the next superblock L_{j+1} is prefetched. If the prefetch time is less than the computation time, then the overhead IO cost is null. We proved this theorem also holds for inversion (see Annexe).

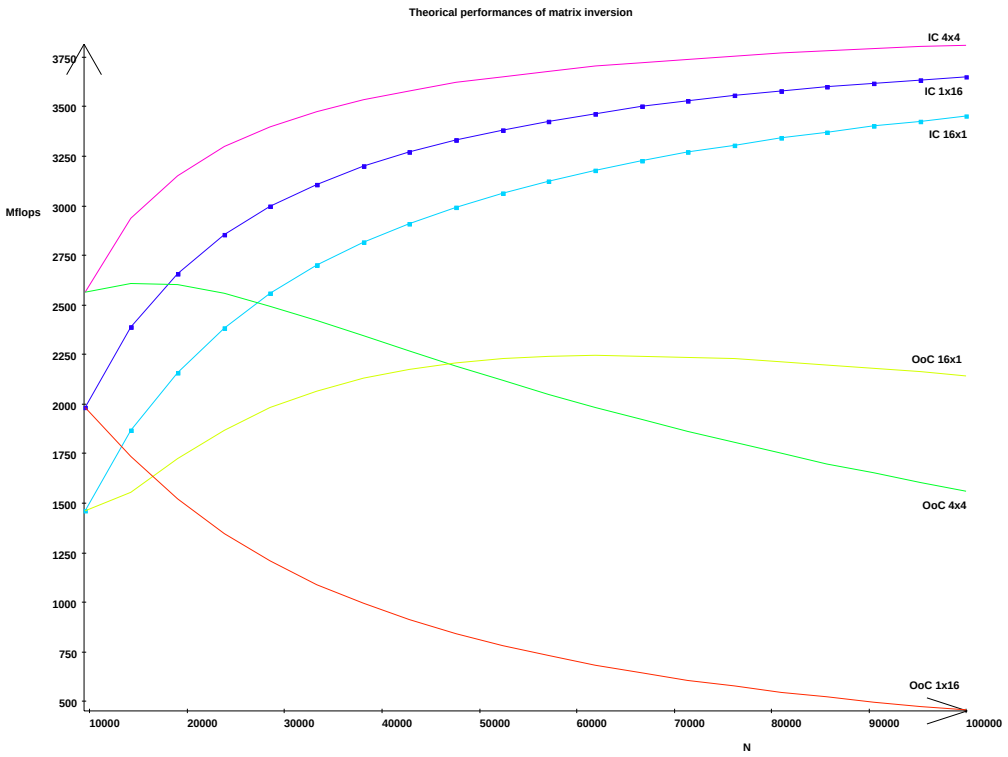


Figure 11: Theoretical performances of the matrix inversion on a cluster of 16 PC-Celeron/Linux (64Mb) interconnected by Fast-Ethernet switch: comparison of the parallel in core (IC) right-looking algorithm and the parallel out-of-core (OoC) left-right looking algorithm, with 3 kinds of topology (1×16, 4×4, and 16×1). N is the matrix order.

5.3 Toward a more general overlaping scheme

To avoid the communication overhead, we considered only one dimentional distribution (processor column). With IO overlaping, the performance of the OoC algorithm is then equal to the performance of the IC algorithm with the same distribution. Unfortunately, for the IC algorithm one column distribution is the worst distribution (Fig. 11):

- the parallelism is reduced because the computation of U_{01} in the LU part (see Section 2.3) is done by only one processor;
- due to partial pivoting the step 1a of LU decomposition (see Section 2.3) is fine grained and involves small communication (so a lot of communication latencies).

In [4], it is show that best performances are obtained with a grid width few rows. So, to improve performance, we have to use a two dimentional distribution. To avoid communication update overhead, another overlaping scheme must be introduced. They are two approaches.

- The first is to reuse the work of Desprez *et al.* [7] where computation and communication are pipelined in basic ScaLAPACK routines (like pdtrsm and pdgemm).
- The second is to read ahead the second next superblock and communicate the next left superblock during the update of the active superblock with the current one.

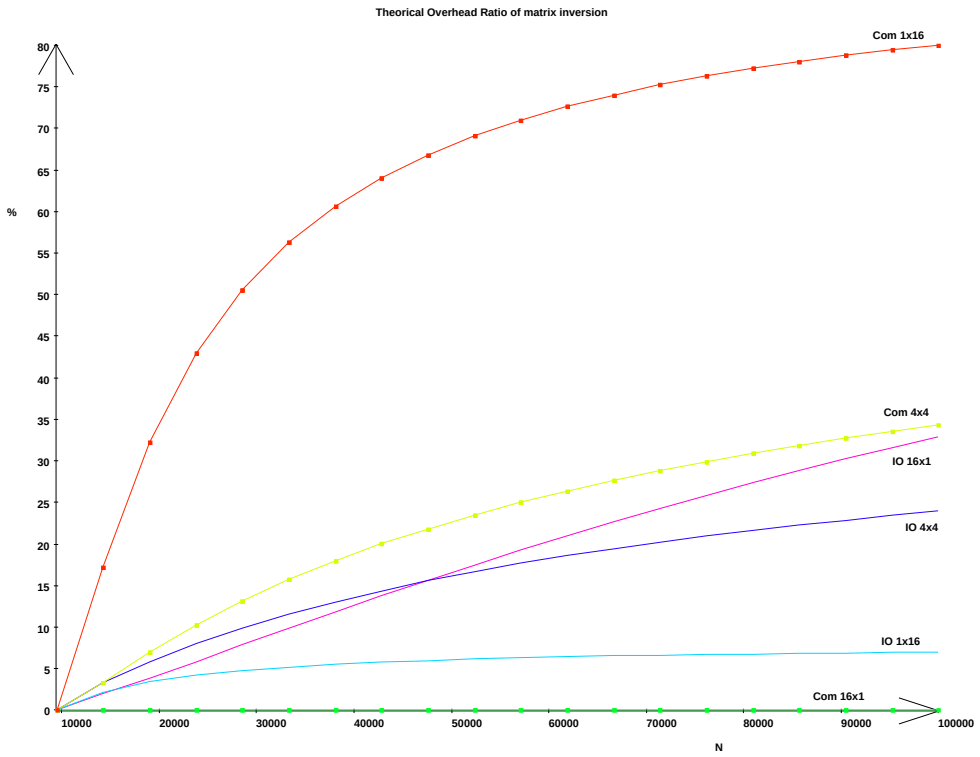


Figure 12: Matrix Inversion: Overhead ratio for communication and IO.

The last approach requires more memory for the prefetched and communicated superblock. But in our description we considered that width of the active and the width of current superblock are equal. An idea to reduce the need for physical memory is to specify different widths for the active and the current superblock during the update: increase width of active superblock (i.e. computation time) and reduce width of current superblocks (i.e. read and communication time). These widths are function of differents architectural constants (network and IO throughput, computation rate).

6 Conclusion

In this paper we have presented a parallel out-of-core matrix inversion algorithm with a performance prediction model of it. Performance model was validated by experiments. This algorithm is derived from the ScaLAPACK *out-of-core* LU factorization. Thanks to this modelling, we isolated the overhead introduced by the out-of-core version. We observed that the best virtual topology to suppress the communication overhead is one column of processor. We show that a straightforward overlapping scheme of the IO by the computations allows us to reduce the IO overhead of the algorithm. We determined the memory size which is necessary to avoid the IO overhead. The memory size needed is proportional to the square root of the matrix size.

To see if this result is practicable, consider a small cluster of PC-Celeron with 16 nodes and with a Fast-Ethernet switch. To invert a 80 Gigabytes matrix (a 100000 matrix order) we need 26 Mega Byte of memory per superblock (active, current and prefetched) per node, i.e. 78 Mbyte per node! The predicted execution time to factorize the matrix is 11 days without overlapping, and 9

days otherwise[2]. If we substitute the Intel Celeron processors of 237 Mflops by Digital Alpha AXP processors of 757 Mflops, then the needed memory size per processor is 252 Mbytes! The predicted computation time is about 6 days without overlapping and about 4 days otherwise (1.5 faster). This last time is the estimated time for the in-core algorithm with the same topology (i.e. one column of 16 processors). With a better topology for the in-core algorithm (4 columns of 4 processors), the in core algorithm takes 3 days to factorize the matrix, but the memory needed by node is 5 Gigabytes: 20 times greater than the memory necessary for the out-of-core version!

This result demonstrate that for some compute bound algorithm, like the matrix inverse computation, cluster computing is able to compete whith clasical supercomputer. A supercomputer has usualy more memory than commodity cluster, but thanks to the out-of-core computation technics, this advantage disappears.

We plan to integrate the general overlapping scheme (communication and IO) described above.

Acknowledgment: Special thanks to Olivier Cozette (LaRIA) for his help in the development.

References

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

[2] Eddy Caron, Olivier Cozette, Dominique Lazure, and Gil Utard. Virtual Memory Management in Data Parallel Applications. In *Proc. of HPCN'99 (High Performance Computing and Networking)*. Springer, April 1999.

[3] Eddy Caron, Dominique Lazure, and Gil Utard. Performance modeling and analysis of parallel out-of-core matrix factorization. In *HiPC'2000, 7th International Conference on High Performance Computing. Bangalore, India*, December 17-20 2000.

[4] 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 Performaces. Technical Report UT-CS-95, Department of Computer Science, University of Tennessee, 1995.

[5] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner. Performance issues of the parallel PAM-CRASH code. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(1):3–11, Spring 1997.

[6] Ed F. d'Azevedo and Jack J. Dongarra. The design and implementation of the parallel out-of-core scalapack LU, QR and cholesky factorization routines. Technical Report UT-CS-97-347, Department of Computer Science, University of Tennessee, January 1997.

[7] F. Desprez, S. Domas, and B. Tourancheau. Optimization of the ScaLAPACK LU factorization routine using Communication/Computation overlap. In *Europar'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 3–10. Springer Verlag, August 1996.

[8] Wesley C. Reiley and Robert A. van de Geijn. POOCLAPACK : Parallel Out-of-Core Linear Algebra Package. Technical report, Department of Computer Sciences, The University of Texas, Austin, Draft 27 October 1999.

[9] J.M. Del Rosario and A. Choudhary. High performance I/O for massively parallel computers: Problems and Prospects. *IEEE Computer*, 27(3):59–68, 1994.

[2] Checkpointing is implicit in out-of-core algorithms, so we can deal with fault tolerances!

[10] Rajeev Thakur, Ewing Lusk, and William Gropp. I/O characterization of a portable astrophysics application on the IBM SP and Intel Paragon. Technical Report MCS-P534-0895, Argonne National Laboratory, August 1995. Revised October 1995.

[11] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 28–40, Philadelphia, May 1996. ACM Press.

Annexe: proof of Theorem 1

Let consider the resource needed to achieve such a total overlapping: We verify that the Theorem 1 also holds for the triangular substitution.

Proof 1 *From modelling we know that the reading time of the next left superblock is (33) where H is the superblock height.*

$$(H-K)K\tau_{pq}^{io} + \beta_{pq}^{io} \tag{33}$$

We underestimate the update computation time, and we consider only the main cost of this update: the `pdgemm` *part.*

$$\frac{2(H-1)K^2}{pq}\alpha_g \tag{34}$$

Let add the communication cost. We assume, from theorem 1, we have just one processor column ($q = 1$):

$$BkK\tau_q^p + B\beta_q^p \tag{35}$$

Now consider the situation where the IO is overlapped by computation (i.e. $O_{IO}^o = 0$), that is $\frac{(34)+(35)}{(33)} \geq 1$. Note that if $\frac{(34)}{(33)} \geq 1$ then $\frac{(34)+(35)}{(33)} \geq 1$. So we restrict the problem to the following: determinate for which superblock width K:

$$\frac{\frac{2(H-1)K^2}{pq}\alpha_g}{(H-K)K\tau_{pq}^{io}} \geq 1 \tag{36}$$

$$\frac{H-1}{H-K} \times \frac{2K\alpha_g}{pq\tau_{pq}^{io}} \geq 1 \tag{37}$$

The first part of this expression is always greater than 1. We know that $q = 1$ and $\tau_{pq}^{io} = \frac{t_{io}}{pq}$. We can evaluate boundary for K:

$$\frac{2K\alpha_g}{t_{io}} \geq 1 \tag{38}$$

i.e.

$$K \geq \frac{t_{io}}{2\alpha_g} \tag{39}$$

To sum up $K = pqM/N$ and $q = 1$, if $pM \geq N\frac{t_{io}}{2\alpha}$ then $\frac{(34)}{(33)} \geq 1$, in another words $O_{IO}^o = 0$.