Efficient Block Cyclic Data Redistribution

Loïc Prylli, Bernard Tourancheau

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Block Cyclic Data Redistribution

Loïc Prylli and Bernard Tourancheau

N 2766

Janvier 1996

PROGRAMME 1

ISSN 0249-6399

Efficient Block Cyclic Data Redistribution

Loïc Prylli and Bernard Tourancheau*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet ReMaP

Rapport de recherche n°2766 — Janvier 1996 — 30 pages

Abstract: Implementing linear algebra kernels on distributed memory parallel computers raises the problem of data distribution of matrices and vectors among the processors. Block-cyclic distribution seems to suit well for most algorithms. But one has to choose a good compromise for the size of the blocks (to achieve a good computation and communication efficiency and a good load balancing). This choice heavily depends on each operation, so it is essential to be able to go from one distribution to another very quickly. We present here the algorithms we implemented in the SCALAPACK library. A complexity study is made that proves the efficiency of our solution. Timing results on the Intel Paragon and the Cray T3D corroborates the results. We show the gain that can be obtained using the good data distribution with 3 numerical kernels and our redistribution routines.

Key-words: parallel computing, parallel linear algebra, personalized all-to-all communication, data redistribution, HPF, block-cyclic data distribution

(Résumé : tsvp)

Le projet *ReMaP* est un projet commun CNRS – ENS Lyon – INRIA. Ce travail est financé par le CNRS contrat PICS, le GDR-PRC PRS action EXEC, la CEE programme EUREKA contrat EUROTOPS.

*Laboratoire LIP, URA CNRS 1398, Ecole Normale Supérieure de Lyon, F - 69364 LYON Cedex 07, e-mail: firstname.lastname@inria.fr

Unité de recherche INRIA Rhône-Alpes
46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1 (France)
Téléphone : (33) 76 57 47 77 – Télécopie : (33) 76 57 47 54

Redistribution efficace des données stockées par blocs entrelacés

Résumé : L'implantation de noyaux d'algèbre linéaire sur les machines parallèles à mémoire distribuée pose le problème du choix de la distribution des données pour les matrices et les vecteurs sur les différents processeurs. Une distribution bloc-cyclique semble convenir pour la plupart des algorithmes, mais un compromis est nécessaire dans le choix de la taille des blocs (pour avoir à la fois des calculs et communications efficaces et une bonne répartition de charge). Le choix optimal est différent pour chaque algorithme, et il est donc essentiel de pouvoir passer d'une distribution à l'autre très rapidement. Nous présentons ici les algorithmes de redistribution que nous avons implantés dans la bibliothèque SCALAPACK. Une étude de complexité vient ensuite prouver l'efficacité des solutions choisies. Les performances obtenues sur Intel Paragon et Cray T3D corroborent nos résultats. Nous montrons le gain obtenu en utilisant une bonne distribution des données avec 3 noyaux de calcul numerique et nos fonctions de redistribution.

Mots-clé : calcul parallèle, algèbre linéaire parallèle, communication en échange total personalisée, redistribution de données, HPF, répartition de données entrelacée par blocs

1 Introduction

This paper describes the solution of the data redistribution problem arising when implementing linear algebra in a distributed system. Although a bit specialized, the problem and its solution contains points of general interest regarding data communication patterns in data parallel languages.

We point out that the paper is not addressing the problem of how to determine a relevant data distribution (even if experimental results are given for 3 numerical kernels), but how to implement a given redistribution. .

The problem of data redistribution occurs as soon as you deal with arrays on parallel distributed memory computer, from vectors to multi-dimensional arrays. It applies both to data-parallel languages such as HPF and to SPMD programs with message-passing. In the first case the redistribution is implicit in array statements like $A = B$ where A et B are two matrices with different distributions. In the second case a library function has to be called to do the same operation or it can also be hidden at the beginning and end of an optimized routine call.

We present here the algorithm and implementation of the redistribution routine that is used in SCALAPACK [4, 7]. Our solution is a dynamic approach in order to construct the communication sets and then efficiently communicate them. Our algorithm uses several strategies depending on the amount of data to be communicated and on the target architecture capabilities in order to be very fast. It runs for any number of processors, making available the possibility of loading and down-loading from/to one processor to/from many others.

Section 2 gives a brief description of related works. Section 3 introduces the SCALAPACK data distribution models and notations used in this paper. Section 4 presents the algorithms that were used for the redistribution of data and section 6 presents timing results obtained on different machines (namely the Cray T3D and the Intel Paragon).

2 Related work

For a long time, redistribution was considered very difficult in the general case, and most implementations are restricting the possible distributions to block or cyclic distributions [3, 2, 11, 1, 5], or in some implementations all block-sizes had to be multiple of each others to ease some memory access operations.

Some recent works show that it can be done at compile time in the general case [8, 10] or describe the access of array elements with different strides [9]. But

generally these works addressed re-distribution of arrays with a fixed number of processors at compile-time, or the compilation of data elements access for block-cyclic distribution. We propose a run-time approach of data-redistribution with no compile-time information and a variable number of processors.

3 Block cyclic data distribution and redistribution

The SCALAPACK library uses the block-cyclic data distribution on a virtual grid of processors in order to reach good load-balance, good computation efficiency on arrays and an equal memory usage between processors. Arrays are wrapped by blocks in all dimensions corresponding to the processor grid. The Figure 1 illustrates the organization of the block-cyclic distribution of a 2D arrays on a 2D grid of P processors.

The distribution of a matrix is defined by four main parameters: a block width size, r; a block height size, s; the number of processor in a row, P_{row}; the number of processors in a column, P_{col} and few others to determine, when a sub-matrix is used, which element of the global matrix is the starting point and which processor it belongs to.

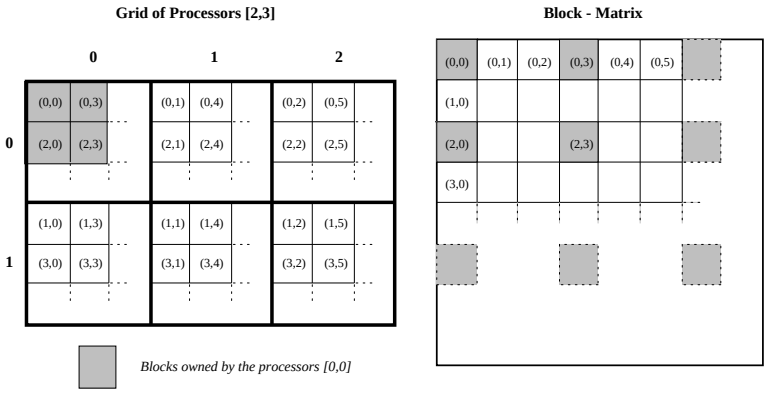

Figure 1: The block cyclic data distribution of a 2D array on a 2×3 grid of processors.

In SCALAPACK, the efficiency of redistribution is crucial as in any data parallel approach because it should be negligible or at least small compared to the computation it was done for. This is especially difficult since the redistribution operation has

to be done dynamically, with no compile-time or static information. This dynamic approach implies that we deal from the beginning with the most general case of redistribution allowed by our constraints, namely cyclic with blocks of size (r, s) on a $P_{row} \times P_{col}$ virtual grid to cyclic with blocks of size (r', s') on a $P'_{row} \times P'_{col}$ virtual grid[1]. Our routines are also usable when dealing with sub-matrices (but do not take into account strides).

Moreover, no latency hiding techniques or overlapping can be used between the redistribution and the previous computation because these routines are independent (remark that it does not prevent the use of these techniques inside the redistribution routine itself, as it is explained in section 4.3).

4 Redistribution algorithm

The whole problem of data redistribution is for each processor to find which data stored locally has to be sent to the others and respectively how much data it will receive from the others and where it will store it locally. When the communication buffers are built (respectively reserved), they have to be transferred between the processors.

4.1 Algorithm in one dimension

Computation of the data sets If we assume that the data are stored contiguously in a block cyclic fashion on the processors, the problem is then to find which data items stored on processor P_i will be send to processor P_j. These data items have to be packed in one message before being sent to P_j in order to avoid start-up delays.

Our algorithm scans at the same time the matrix indices of the data blocks stored on P_i and those that will be stored on P_j. More precisely, we keep two counters, one corresponding to P_i's data location in the global matrix and the other to P_j's one. We increment them progressively by block as in a merge sort in order to determine the overlap areas as shown in Figure 2 (the complexity is linear in the number of blocks). Then we pack the data items corresponding to the overlap areas in one message to be sent to P_j.

[1] Notice that this general case includes the loading and down-loading of data from a processor to a multicomputer and also calls to parallel routines from a sequential code.

Communication of the data sets The communications that occur between the processors correspond to a personalized "all-to-all", i.e. a total exchange with message sizes depending of the processors. The emission buffer is filled and the reception buffers sizes is determined thanks to the computation of the data sets.

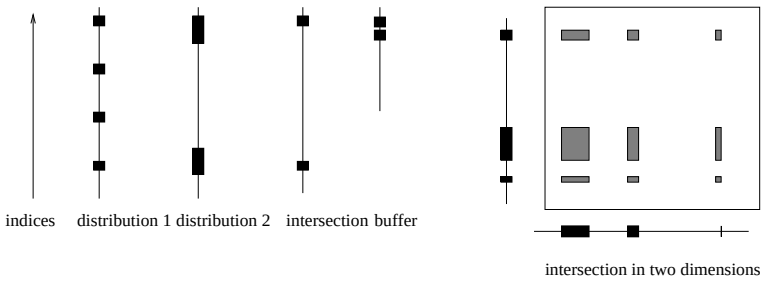

Figure 2: Graphical representation of the research of the intersection of two block-cyclic distributions for a pair of processors. One dimension case on the left, two dimension case on the right.

4.2 General algorithm

The block scanning is done dimension by dimension (see Figure 2) and the overlapping indices are the Cartesian product of the intervals computed in each dimension (there is no limitation in the number of dimensions scanned and the complexity is linear in the sum of the dimension sizes while the packing is obviously linear in the size of the data).

This work is done in each processor in order to send data and respectively to receive data and store them at the right place in local memory. The whole communication is a personalized "all-to-all" as in section 4.1.

4.3 Optimizations

We present in this section several optimizations of the general algorithm that we implemented. The first one reduces the amount of time and memory necessary for the computation of the data sets and the others concern the communication of the resulting buffers depending of the target hardware/OS capabilities.

Scanning : The obvious scanning strategy tests on each processor the whole range of indices of the two data distributions to find the items that belongs to the processor P_i and should be communicated to P_j. But as we will see in proposition 4, the intersection of intervals (that represent the indices of the data items) in a block cyclic distribution is in fact periodic of period lcm(rP,r'P'). So, instead of a full block scanning, the scanning algorithm can stop as soon as it reaches the cyclic bound. Moreover this also reduces the bound on the necessary storage for the intersection patterns.

Asynchronous (receive) communication : In that case, the communication algorithm is simple. The sizes of the messages to be received are computed first. Then, the asynchronous receives are posted followed by the sends. There is no assumption on the target computer ability to receive messages *"from any"* .

Synchronous communication : In order to avoid OS dead locks due to the fixed number of communication buffers for instance, we designed an algorithm that is using a blocking receive protocol. Moreover, in order to minimize the processor idle time, exchanges are built, i.e. all receive function calls have the corresponding send function calls posted before or at the same time (this removes the *"non-blocking sends"* assumption on the target computer).

This strategy, illustrated in Figure 3, can be compared to a rolling caterpillar of processors where at a given step d, each processor P_i, $(0 \leq i < P)$ exchanges its data with processor $P_{((P-i-d) mod P)}$.

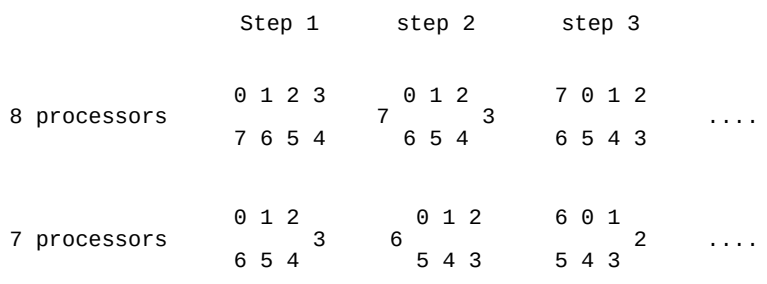

Figure 3: The caterpillar communication method is illustrated with an even (8) and an odd (7) number of processors. The communication occurs between the vertical pairs of processors (a processor alone communicates with itself).

Communication pipelining : The pipeline method takes advantage of the possibility of dividing the work in small units. Instead of waiting for all the information from another processor, each processor P_i receives a small packet of elements, and in the same time packs a small packet of elements to be sent and unpacks the elements it just received. This is an overlapping strategy close to the work described in [6].

This optimization is implemented within the caterpillar method where for each step there are several send/receive exchanges. This overlap between communication and computation improved the timings on machines with rather slow communications[2].

5 Complexity Study

We consider the redistribution of a multidimensional array of size $M_1 \times M_2 \times ... \times M_D$. The data distributions are defined by data block sizes $r_1 r_2 \cdots r_D$ and a processor grid of size $P_1 \times P_2 \times \cdots \times P_D$ as in section 3. A prime will indicate the parameters in the target distribution.

For the sake of simplicity, we decompose the study in 2 parts corresponding to the contribution of indices computation and data moves and transfers:

Proposition 1 *The total complexity is* $T_{redistrib} = T_{scan} + T_{comm}$

5.1 Scanning complexity

The block scanning is done for each processor in order to send the data and respectively to receive the data and to stored them at the right place in local memory[3].

The obvious scanning strategy is testing all indices of the global matrix. An elementary operation is then the computation of the initial and final owners of a given element. This requires a modulo operation as the data distribution is cyclic. But as we repeat it for adjacent items, this computation can be decomposed and transformed in just a few additions and comparisons for all but the first element. This strategy complexity is:

Proposition 2

$$T^1_{scan} = O\left(\prod_{i=1..D} M_i\right)$$

[2] i.e. LAN of workstations and "old" parallel computers

[3] At any time, the computation is done with global indices of the matrix but only local indices (corresponding to the local part of the matrix) are necessary to access the data stored in each processor.

processor elementary operations.

In order to improve the complexity, we take into account the block pattern of the distribution for the indices progression (described in 4.1). Then the complexity is:

Proposition 3

$$T^2_{scan} = O\left(\sum_{i=1..dim} \frac{M_i}{r_i P_i} + \frac{M_i}{r'_i P'_i}\right)$$

Proof *The scanning is done block by block independently in each dimension (cf. §4.1). There are* $\frac{M_i}{r_i P_i}$ *(resp.* $\frac{M_i}{r'_i P'_i}$*) blocks on the original (resp. final) distribution for the source (resp. final) processor. These memory locations are tested like in the "merge sort" algorithm, hence the complexity is equal to the total number of blocks.* $\diamond$

We describe in the following the reduction of the scanning complexity obtained with the optimizations implemented in our algorithm. The aim is to reduce the scanned interval to its periodic pattern as proposed in section 4.3. Let us have a look at the problem in one dimension, where block cyclic(r) means a cyclic data distribution by blocks of size r. We decompose the cyclic patterns to their canonical form (proposition 4) in order to be able to compute their intersection (proposition 5) and its size (proposition 6). Then the results are given in propositions 7 and 8.

Notation: $E(s,l,b) = \{x | x = s + kl + j, k \in Z, j \in [0..b-1]\}$ and $C(s,l) = E(s,l,1)$. Then, in the one-dimensional case, the set of items owned by a processor is a pattern $E(s, rP, r)$, and $C(s, rP)$ in the more particular case of a cyclic distribution.

Proposition 4 *The intersection of two block cyclic(r) patterns is the union of less than* $r \times r'$ *cyclic patterns of size 1.*

Proof ***Lemma :*** *Let* a, a', m, m' *be 4 integers.*
If $(a - a')$ *is a multiple of* $gcd(m, m')$
then $\exists b, C(a,m) \cap C(a',m') = C(b, lcm(m,m'))$,
else $C(a,m) \cap C(a',m') = \emptyset$.
The proof is a simple application of Bezout theorem.

Let $E(s, rP, r)$ and $E(s', r'P', r')$ be two block-cyclic patterns.

$$E(s, rP, r) = \bigcup_{i=0..r-1} C(s+i, rP)$$

then we have:

$$E(s, rP, r) \bigcap E(s', r'P', r') = \bigcup_{(i,j)\in[0..r-1,0..r'-1]} C(s+i, rP) \cap C(s'+j, r'P')$$

By application of the lemma, each individual intersection is empty or is a cyclic pattern of period $lcm(rP, r'P')$, so the intersection is an union of at most $r \times r'$ cyclic patterns. $\diamond$

Proposition 5 *The intersection of two block cyclic(r) patterns is periodic of period $lcm(rP, r'P')$ (where P and P' are the number of processors in each distribution).*

Proof *This is derived from the proof of proposition 4. An union of patterns all periodic with the same period $lcm(rp, r'P')$ is also periodic with the same period* $\diamond$

Hence we can use that property to speedup the computation and decrease the memory needs.

Proposition 6 *There will be at most $r \times r'$ items in an interval of length m of the resulting pattern.*

Proof *We have seen in the proof of proposition 4 that the resulting pattern is the union of at most $r \times r'$ cyclic patterns, of identical period. In one period, we have one representative of each component of the union* $\diamond$

In the following, **let** b **be** $\max(M, \mathbf{lcm}(rP, r'P'))$. Thanks to the previous propositions, as described in 4.3, we can stop the scanning as soon as we reach the global index b because the construction of the intersection patterns is completed. Moreover we have a bound of $r \times r'$ on the number of descriptors of the intersection pattern (this will be important for the required storage).

In practice this optimization is only interesting when we have very small block sizes or very large matrices. Although there are cases where it is more interesting

to analytically determine the pattern following proposition 4 (for instance for cyclic distributions as in [10]), generally for block of reasonable length[4] our strategy is better. For instance, see Figure 4 where the corresponding complexities are plotted as a function of the average block size $\sqrt{rr'}$ (the square root of the initial distribution block size times the final distribution block size).

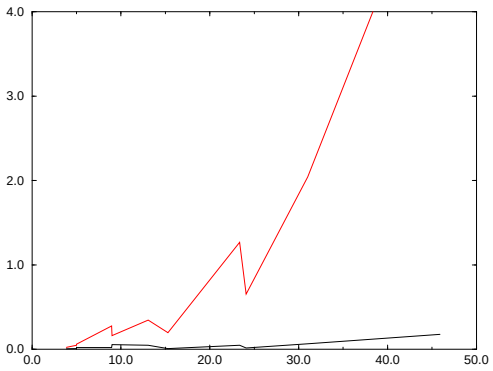

Figure 4: Comparison of the two scanning strategies in one dimension. The time (milliseconds) is plotted against different random block sizes (the abscissa axis represents the average block size). Each value is obtained from the mean of one thousand iterations of the same scanning on a SUN Sparc5 workstation.

Notice that with the preceding optimizations, the complexity is independent of the matrix size (except for small matrices where $M < \mathrm{lcm}(rP, r'P')$) :

Proposition 7 *The final scanning complexity in one dimension is:*

$$T^3_{scan} = O\left(\frac{b}{rP} + \frac{b}{r'P'}\right)$$

where b *is the* $\max(M, lcm(rP, r'P'))$

[4] i.e. > 1 for classical parallel computers

Proof *From proposition 6, the scanning interval is reduced to* $\frac{b}{rP}$ *by the pre-computation of the cyclic pattern intersections.* $\diamond$

Finally, the result can be straightforwardly extended to multidimensional arrays :

Proposition 8

$$T^4_{scan} = O\left(\sum_i \frac{b_i}{r_i P_i} + \frac{b_i}{r'_i P'_i}\right)$$

where $b_i = \max(M_i, lcm(r_i P_i, r'_i P'_i))$

Proof *Derived from proposition 7 by adding the cost in each dimension.* $\diamond$

Notice that the scanning duration is greatly reduced by our optimization but the packing one remains the same.

5.2 Packing and communication complexity

The packing consists of "move" operations to build a buffer, the buffer is then sent. Following the classical linear transfer time model, we describe the communication cost complexity for multi-dimensional arrays in terms of move and transfer time only (there is always $P-1$ startup delays in our algorithm) :

Proposition 9

$$T_{comm} = O\left(\prod_{i=1..D} \frac{M_i}{P_i P'_i}\right)$$

move and transfer operations for a pair of processors (and each processor will exchange with all the others).

Proof *All elements that are to be sent to a partner must be packed in one buffer to reduce startup overhead. The corresponding cost is proportional to the number of elements per processor, that is on average* $\prod_{i=1..D} \frac{M_i}{P_i P'_i}$. $\diamond$

5.3 Comparison between scanning and packing/communication complexities

In order to study the relative importance of each part of our algorithm (scanning and packing/communication), we present in the following the behavior of their ratio when M is increasing. The ratio R of the complexities of the scanning and the moves and transfers is:

Proposition 10

$$\begin{aligned} R &= \frac{T_{scan}}{T_{move}} = O\left(\sum_i \frac{P \prod_{j \neq i} \frac{P_j}{M_j}}{r_i}\right) \\ &= O\left(\sum_i \frac{P^2 M_i}{M P_i r_i}\right) = \frac{P^2}{M} O\left(\sum_i \frac{M_i}{P_i r_i}\right) \end{aligned}$$

where $P = \prod P_i$ is the total number of processors.

Proof *Just coming from the ratio of proposition 8 and 9.* $\diamond$

Then looking at the limit :

Proposition 11 *If $\prod M_i \to \infty$ then $R = \sum_i \frac{P \prod_{j \neq i} \frac{P_j}{M_j}}{r_i} \to 0$*

Proof *The ratio $\frac{P_j}{M_j} \to 0$* $\diamond$

Asymptotic proposition 11 means in practise that as soon as M is large enough[5], if our optimizations are applied (cf. section 4.3), the redistribution cost is roughly equal to two memory copies (one when the data is sent and one when it is received) plus the transfer time of the items owned locally, i.e. the scanning is negligible.

6 Timing results

In this section, we want to show the efficiency of our algorithms and the correctness of our complexity results but also to demonstrate the real gain that is obtained in

[5] This is true even for matrices of moderate size, for instance when $\prod M_i$ is ten times bigger than the number of processor Ps.

practice for linear algebra kernel computation using our redistribution routines. We choose for that 3 routines of the SCALAPACK library: the matrix multiply (MM), LU factorization (LU) and triangular solve (TS). In the SCALAPACK library, the routines run for any virtual grid of processors and any size of square blocks.

Our experiments were ran on the Cray T3D and on the Intel Paragon parallel computers. We use a compiled version of the SCALAPACK and BLACS libraries on top of the vendor optimized BLAS on the Paragon, and a compiled version of the redistribution routine with a vendor implementation of the LU, MM, TS, BLACS and BLAS routines on the T3D.

For a given number of processors (16) and a given matrix size (1024×1024), we first determine the best block size and grid shape for each computation kernel. Our tests show the timings as a function of the block sizes for virtual processors grids of all shape : 1×16, 2×8, 4×4, 8×2, 16×1 processors grids. For clarity, the block sizes are chosen power of 2, but our redistribution routines run for any sizes.

We show in Figure 5 the redistribution elapse time for random block sizes and random grid shapes on the Paragon[6].

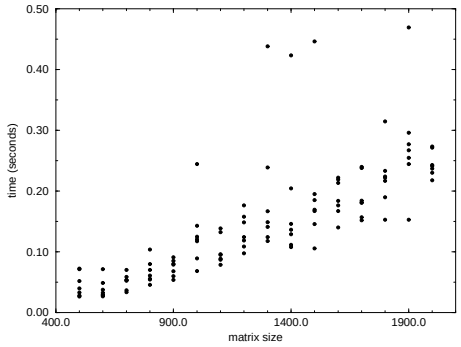

Figure 5: Redistribution elapse time for different matrix sizes on random grids of 14 to 16 processors with random block sizes.

[6] We were not able to run tests with non power of 2 grids on the T3D because of vendor BLACS limitation on this machine

We then present the efficiency of our solutions, with a comparison of the timings with and without redistributions of the data (see the corresponding algorithms on Figure 6) for the 3 linear algebra computation kernels (LU, MM, TS) on the multicomputer .

Algorithm	**Algo. with redistribution**
Computation_kernel(A-datadist)	Data_redistribution(A-datadist,A-bestforkernel) Computation_kernel(A-bestforkernel) Data_redistribution(A-bestforkernel,A-datadist)

Figure 6: Algorithms for one computation kernel without and with data redistribution.

With that results, we then study an example with the 3 linear algebra operations executed sequentially in the same parallel program (see the corresponding algorithms on Figure 7). The number of systems solved is set up to one fourth of the matrix size in order for that routine to have an execution time comparable to the 2 others).

Remark that if several arrays are implied in the computation kernel, then several data redistributions must be called (in the Matrix Multiply for instance). Hence, the redistribution solution includes 6 calls to the redistribution routine (i.e. 2 for the MM kernel, 1 for the LU kernel, 2 for the TS kernel, plus one to go back to the initial data distribution).

6.1 On an Intel Paragon parallel machine

On the Intel Paragon, Figure 8 shows that the best performance of the matrix multiply is obtained with a 4×4 virtual grid with the biggest block size (64 for our problem).

For the LU decomposition, the best result is obtained with blocks of size 8 on a 1×16 virtual grid of processors.

For the solve of 1024 right hand sides, the best result is obtain with the biggest block size (64) on the 16×1 virtual grid of processors.

The grid shape and block sizes are very different for these 3 routines (see Table 1) and the timings differences are not negligible. This tend to demonstrate that the redistribution improvement can lead to an important gain in elapse time.

Knowing these best configurations, we ran the tests corresponding to the algorithm of Figure 6 for the 3 numerical kernels with 16 processors. We plotted with

Algorithm	**Algo. with redistribution**
	Data_redistribution(A-datadist,A-bestforMM)
	Data_redistribution(B-datadist,B-bestforMM)
Matrix_multiplication(A-datadist,B-datadist)	Matrix_multiplication(A-bestforMM,B-bestforMM)
/ result is assumed stored in A */*	Data_redistribution(A-bestforMM, A-bestforLU)
LU_factorization(A-datadist)	LU_factorization(A-bestforLU)
/ results are assumed stored in A */*	Data_redistribution(A-bestforLU, A-bestforTS)
/ C is the solution matrix */*	Data_redistribution(C-datadist,C-bestforTS)
Triangular_solve(A-datadist,C-datadist)	Triangular_solve(A-bestforTS,C-bestforTS)
/ result is assumed stored in A */*	Data_redistribution(C-bestforTS, C-datadist)

Figure 7: Algorithms using several numerical kernel without and with data redistributions

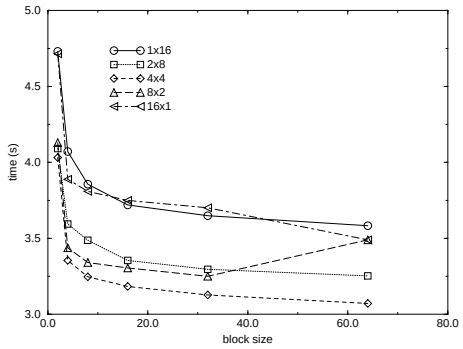

Figure 8: Matrix multiply elapse time on the Intel Paragon with 16 processors in different grid shapes and different square block sizes.

dots on figures 11, 12 and 13 all the timings without redistributions obtained with all different virtual grids and block sizes combinations for several matrix sizes. The best

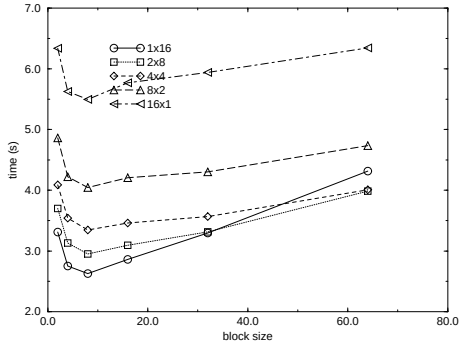

Figure 9: LU decomposition elapse time on the Intel Paragon with 16 processors in different grid shapes and different square block sizes.

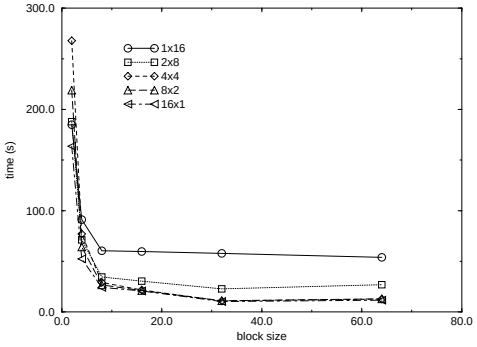

Figure 10: Triangular Solve elapse time on the Intel Paragon with 16 processors in different grid shapes and different square block sizes.

and worst cases of the same computations with the redistributions are represented as curves on the same graph.

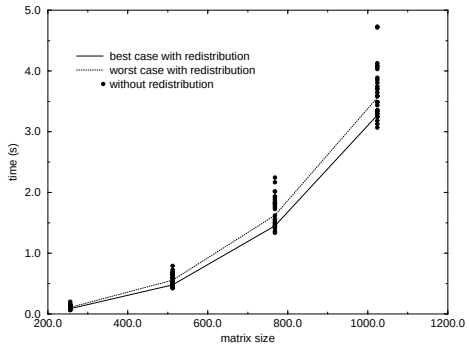

Figure 11: Matrix Multiply elapse time on the Intel Paragon with 16 processors for all grid shapes and block sizes without data redistributions compared with redistribution to the best configuration.

The results are extremely good because there is only slight differences between the best and worst case with the redistributions. Hence this performance stays in the same range close to the optimal. On the contrary, a bad distribution choice can lead to double the execution time for LU and multiply by five the time of the solves.

To conclude, we present in Figure 14 the tests corresponding to the algorithm of Figure 7 where the numerical kernels are chained in the same parallel program.

The results proves that in all cases, it is better to use the redistributions than to stay with the same one from the beginning to the end. Moreover, the first two curves of the tests are the worst possible case while using redistribution because first we assume that none of the array is in the good distribution at the beginning, and second, we redistribute the arrays at the end in order to go back to the initial data repartition. These 2 steps are not mandatory in efficient programming (for instance if the initial distribution is well chosen and if there is no assumption on the data distribution during the rest of the program). Thus the gain obtained for

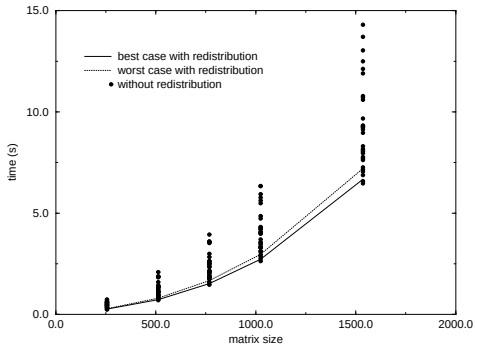

Figure 12: LU decomposition elapse time on the Intel Paragon with 16 processors for all grid shapes and block sizes without data redistributions compared with redistribution to the best configuration.

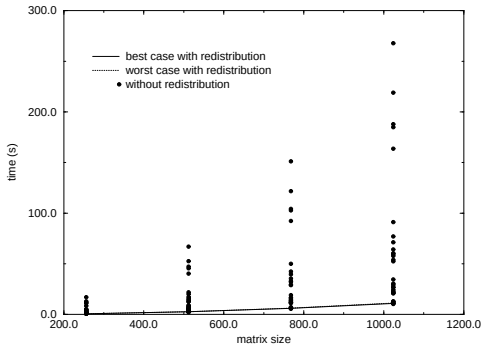

Figure 13: Triangular Solve elapse time on the Intel Paragon with 16 processors for all grid shapes and block sizes without data redistributions compared with redistribution to the best configuration.

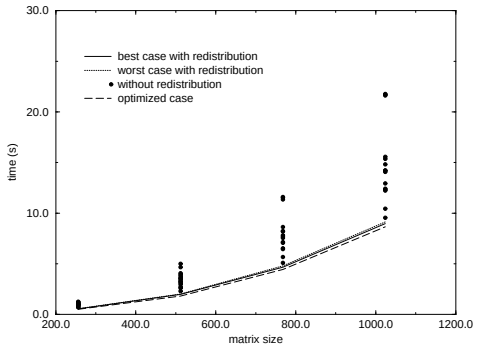

Figure 14: Comparison of the succession of the 3 numerical kernels on the Intel Paragon with 16 processors for all grid shapes and block sizes with and without data redistributions and in the optimized case.

this computation is bigger for this optimized case, which is represented in the lowest curve of Figure 14.

6.2 On an Cray T3D parallel machine

For the Cray T3D, Figure 15 shows that the best performance of the matrix multiply is obtained with a 4×4 virtual grid with the biggest block size (64 for our problem) as on the Paragon.

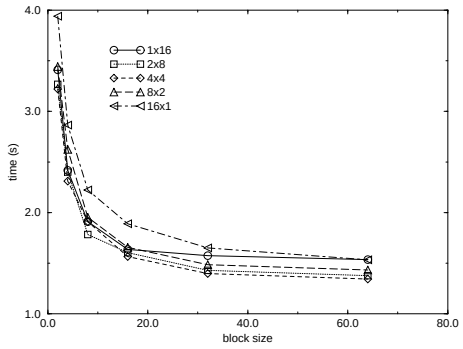

Figure 15: Matrix Multiply elapse time on the Cray T3D with 16 processors in different grid shapes and different square block sizes.

For the LU decomposition, the best result is obtained with blocks of size 16 on a 1×16 virtual grid of processors.

For the solve of 1024 right hand sides, the best result is obtain with the biggest block size (64) on the 16×1 virtual grid of processors.

Again, the grid shape and block sizes are very different for these 3 routines (see Table 1), so the redistribution algorithm can lead to an important gain in elapse time.

As for the Paragon, we ran the tests corresponding to the algorithms of Figure 6 for the 3 numerical kernels with 16 processors. We plotted on figures 18, 19 and 20 all the timings without redistributions obtained with all different virtual grids and

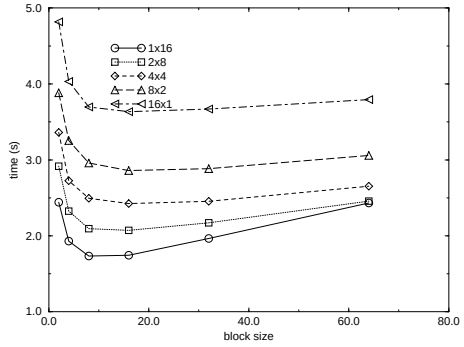

Figure 16: LU decomposition elapse time on the Cray T3D with 16 processors in different grid shapes and different square block sizes.

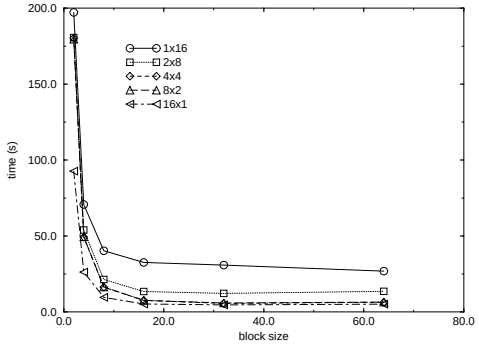

Figure 17: Triangular Solve elapse time on the Cray T3D with 16 processors in different grid shapes and different square block sizes.

block sizes combinations for several matrix sizes. The best and worst cases of the same computations with the redistributions are represented as curves on the same graph.

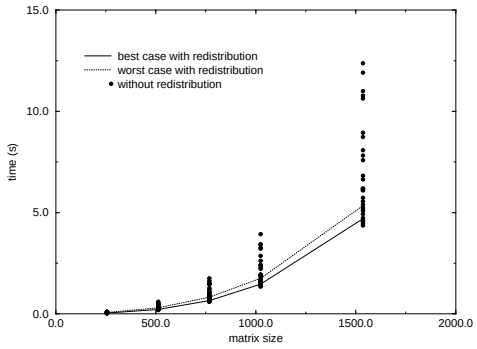

Figure 18: Matrix Multiply elapse time on the Cray T3D with 16 processors for all grid shapes and block sizes without data redistributions compared with redistribution to the best configuration.

Here again, the use of redistribution allow to have an elapse time almost independent of the initial distribution and very close from the optimal. Basically, the redistribution is useless only when the matrix is already in the optimal configuration.

To conclude, we present in Figure 21 the tests corresponding to the algorithm in Figure 7 where the numerical kernels are chained in the same parallel program. Here again, there is no single data distribution that permit to reach the optimal performance obtained by the redistribution between each operation. The results of the optimized programming without initial and final redistributions are also plotted.

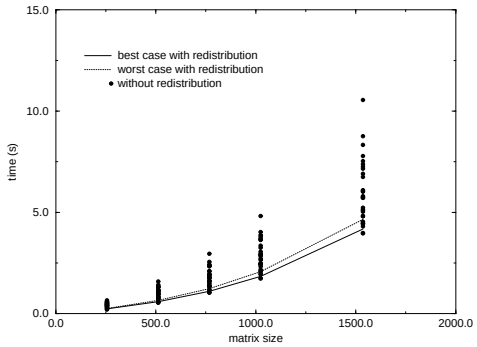

Figure 19: LU decomposition elapse time on the Cray T3D with 16 processors for all grid shapes and block sizes without data redistributions compared with redistribution to the best configuration.

	Paragon		T3D	
Kernel	BBS	BGS	BBS	BGS
MM	64	4×4	64	4×4
LU	8	1×16	16	1×16
TS	64	16×1	64	16×1

Table 1: Best Block Size (BBS) and Best Grid Shape (BGS) for the different parallel computers of our test while running the Matrix Multiply (MM), LU decomposition (LU) and Triangular Solves (TS) computation kernels.

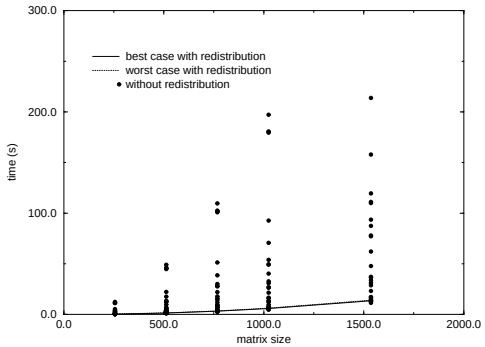

Figure 20: Triangular Solve elapse time on the Cray T3D with 16 processors for all grid shapes and block sizes without data redistributions compared with redistribution to the best configuration.

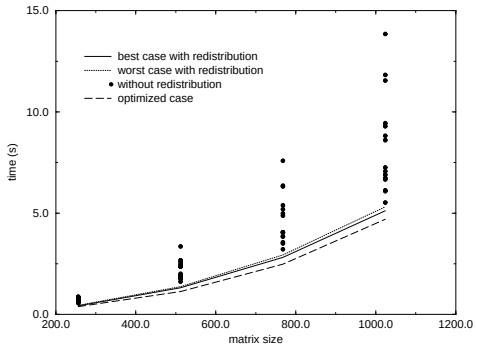

Figure 21: Comparison of the succession of the 3 numerical kernels on the Cray T3D with 16 processors for all grid shapes and block sizes with and without data redistributions and in the optimized case.

7 Conclusion

In the general case, the redistribution of data is useful to improve the efficiency of parallel linear algebra routines. But to ensure a gain on the elapse time, the redistribution of data has to be very efficient.

Our complexity analysis shows that the scanning is negligible for arrays of common size. Then with our optimizations, it is sufficient to know the distribution parameters only at runtime and there is no more constraints about providing all distribution parameters at compile-time.

Tests were run on a Cray T3D and on an Intel Paragon. All the experiments corroborate very well our expectations, the computation of the data sets is negligible compared to the communication and packing, and the global redistribution routine execution time is very good.

These results show that the redistribution of data with our algorithms can efficiently perform in practice giving up to 80% gain and assuring on the average that the computation time stays very close to the optimal even with bad initial data distribution choices (for both block sizes and grid shape).

From our results, a first improvement should be the integration of these redistribution routines in the linear algebra kernel themselves (before and after the computation) as soon as they provide a gain, i.e. when the matrix size is bigger than a given point depending of the platform.

Moreover, our results are encouraging for the frequent use of this redistribution library routines in explicit parallel programming, master/slave schemes or in compiled codes generated by HPF compilers.

Our algorithms are implemented within the SCALAPACK library and are accessible for test and use on the W^3 site `http://www.netlib.org/scalapack/index.html`.

References

[1] G. Agrawal, A. Sussman, and J. Saltz. Compiler and runtime support for structured and block structured applications. pages 578–587, 1993.

[2] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Conference on programming language design and implemnetation*, Albuquerque, NM, June 1993. ACM SIGPLAN.

[3] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S. H. Teng. Generating local addresses and communication sets for data-parallel programs. In *Symposium on Principles and practice of parallel programming*, San diego, CA, May 1993. ACM SIGPLAN.

[4] J. Choi, J.J. Dongarra, and D.W. Walker. The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools For Parallel Scientific Computing*, pages 3–15. Elsevier, 1992.

[5] P. Crooks and R. H. Perrott. Language construct for data partitioning and distribution. Technical report, dept of C.S., The Queen's Univ of Belfast, Belfast BT7 INN, Northern Ireland, 1994.

[6] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Cmputation Subroutines. *Future Generation Computer Systems*, 10:279–284, 1994.

[7] J.J. Dongarra, R. Van De Geijn, and R.C. Whaley. Two Dimensional Basic Linear Algebra Communication Subprograms. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools For Parallel Scientific Computing*, pages 17–29. Elsevier, September 1992.

[8] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. Technical Report TR95521-S, CRPC, Rice Univ., Houston, TX 77005, March 1995.

[9] A. Sethi K. Kennedy, N. Nedeljkovic. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Principles and practice of parallel programming*, volume 30 of *ACM SIGPLAN NOTICES*, pages 102–111, Santa Barbara, CA, July 1995. ACM Press.

[10] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generating communications for array statements: design implementation and evaluation. *JPDC*, 21:150–159, 1994.

[11] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in hpf programs. In *Scalable High-Performance Computing Conference*. IEEE, May 1994.

Biographies

LOÏC PRYLLI is a Ph.D Candidate and assistant professor at the ENS-Lyon, France. He received his M.S. degrees in Computer Science from ENS-Lyon in 1994, he also received a M.S. degree in Mathematics in 1995. His research interests includes parallel and distributed algorithms, parallel operating systems, and environment for parallel programming.

BERNARD TOURANCHEAU is a senior research associate of CNRS at the LIP in ENS-Lyon, France. He received his Habilitation degree from University of Lyon in 1994, his Ph.D. degree in Computer Science from University of Grenoble in 1989. He was a research associate at the University of Tennessee, Knoxville from 1992 to 1994. His major interests are parallel and distributed algorithms, environment for parallel programming. He is a member of ieee and acm.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399