Block cyclic array redistribution

Loïc Prylli, Bernard Tourancheau

▶ **To cite this version:**

Loïc Prylli, Bernard Tourancheau. Block cyclic array redistribution. [Research Report] LIP RR-1995-39, Laboratoire de l'informatique du parallélisme. 1995, 2+12p. hal-02101963

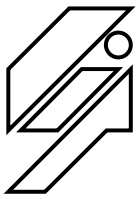
HAL Id: hal-02101963

https://hal-lara.archives-ouvertes.fr/hal-02101963v1

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Laboratoire de l'Informatique du Parallélisme

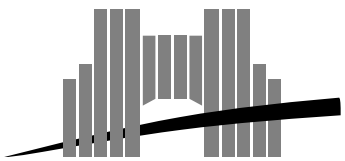
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Block Cyclic Array Redistribution

Loïc Prylli
Bernard Tourancheau

Octobre 1995

Research Report N^{O} 95-39



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Block Cyclic Array Redistribution

Loïc Prylli
Bernard Tourancheau

Octobre 1995

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 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 then made that proves the efficiency of our solution. Timing results on a network of SUN workstations and the Cray T3D using PVM corroborates the results.

Keywords: linear algebra, data redistribution, HPF, block scattered, block-cyclic

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 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 réseaux de stations et Cray T3D en utilisant PVM corroborent nos résultats.

Mots-clés: algèbre linéaire, redistribution de données, HPF, bloc-cyclique

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, 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 [CDW92, DGW92]. 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 and runs for any number of processors, making available the possibility of loading and down-loading from/to one processor to/from many others.

Section 3 introduce 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 a cluster of workstations).

2 Related work

For a long time, redistribution was considered very difficult in the general case, and most implementations were restricting the possible distributions to block or cyclic distributions [CGL+93, AL93, TCF94, ASS93, CP94], or in some implementations all block-sizes had to be multiple of each others to ease some memory access operations.

Some recent work shows that it can be done at compile time in the general case [HKMCS95, SOG94] or describes the access of array elements with different strides [KK95]. But all of these works addressed the compilation techniques for re-distribution of arrays with a fixed 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 block 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 the starting point and which processor it belongs to.

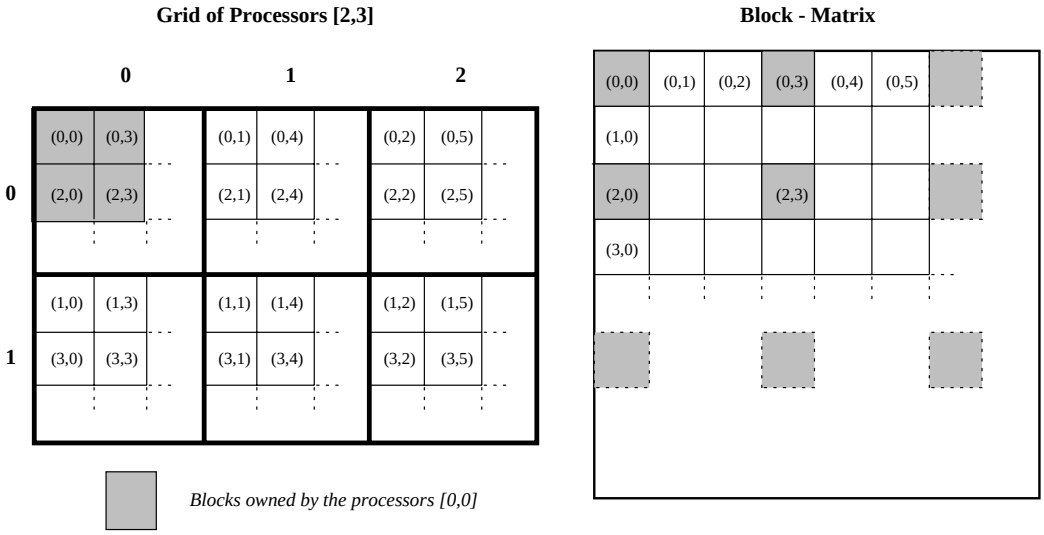


Figure 1: The block cyclic data distribution of a 2D array on a 2×3 grid of processors.

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].

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. Then the communication problem itself occurs on the target computer.

4.1 Computation of data sets in one dimension

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 (the comparison number 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.

4.2 General algorithm for the computation of data sets

The block scanning is done dimension by dimension and the overlapping indices are the Cartesian product of the intervals computed in each dimension. (There is no limitation in the number of dimension 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.

4.3 Optimizations

[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.

Scanning : The obvious scanning strategy tests on each processor every indices of the two data distributions that belongs to the processor and determine if the corresponding data has to be communicated. But as we will see in proposition 3, the intersection of intervals in a block cyclic distribution is in fact periodic of period lcm(rP,r'P'). So, instead a full block scanning, the scanning algorithm stops as soon as it reaches the cyclic bound and moreover it also reduces the bound on the storage necessary for the intersection patterns.

Synchronous communication : In order to avoid OS dead locks due to buffers limitation an algorithm was designed using a blocking receive protocol. Hence to minimize 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 strategy, illustrated in Figure 2, can be compared to a rolling caterpillar of processors: at step d, each processor P_i, $(0 \leq i < P)$ exchanges its data with processor $P_{((P-i-d) mod P)}$.

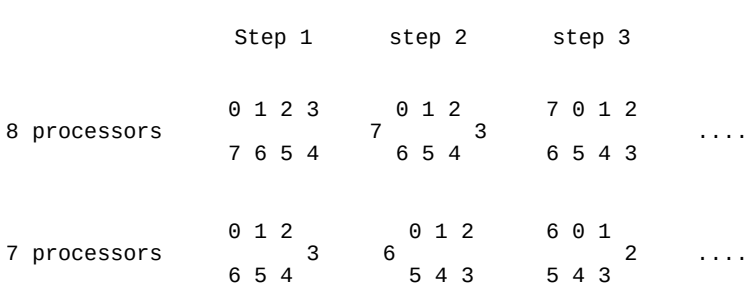


Figure 2: The caterpillar communication method is illustrated with an even (8) and an odd (7) number of processors. The communication occurs between the vertical pairs of processors (a processor alone communicates with itself).

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

Communication pipelining : The pipeline method takes advantage of the possibility of dividing 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 describe in [DT94].

The algorithm 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.

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 every indices of the global matrix. An elementary operation is then the computation of the initial and final owners of a given element. This required 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 an comparisons for all but the first element. This strategy complexity is:

Proposition 1

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

The first improvement we can do is taking into account the block pattern of the distribution for the indices progression. Then the complexity is:

Proposition 2

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

[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.

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 improvements that are done in our algorithm. 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.

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 $E(s, rP, r)$.

Proposition 3 *The intersection of two block cyclic(r) patterns is the union of less than $r \times r'$ cyclic patterns of size 1.*

lemma Let a, a', m, m' be 4 integers.
If $a - a'$ is a multiple of $\gcd(m, m')$, then

$$\exists b, C(a,m) \cap C(a',m') = C(b, \mathrm{lcm}(m,m')),$$

else

$$C(a,m) \cap C(a',m') = \emptyset.$$

The proof is a simple application of Bezout theorem.

Proof 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 void or is a cyclic pattern of period $\mathrm{lcm}(rP, r'P')$, so the intersection is an union of at most $r \times r'$ cyclic patterns $\diamond$

Proposition 4 *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 3. An union of pattern all periodic with the same period $\mathrm{lcm}(rp, r'P')$ is also periodic with the same period $\diamond$

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

Proposition 5 *There will be at most $r \times r'$ items in an interval of length m of the resulting pattern.*

Proof We have seen in proof of proposition 3 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 $\max(M, \mathrm{lcm}(rP, r'P')$. 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, especially for cyclic distributions as in [SOG94], generally for block of reasonable length the previous strategy is better. For instance, see figure 3 where the corresponding complexities are plotted as a function of the average block size $\sqrt{rr'}$.

Notice that with the preceding optimizations, the complexity is independent of the matrix size :

Proposition 6 *The final scanning complexity in one dimension is:*

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

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

Notice that the scanning duration is greatly reduced by our optimization but the packing will remain the same.

Proposition 7 *The scanning is* $\sum_i \frac{b_i}{r_i P_i} + \frac{b_i}{r'_i P'_i}$ *where* $b_i = \max(M_i, lcm(r_i P_i, r'_i P'_i))$

Proof Derived from proposition 6 by adding the cost in each dimension. $\diamond$

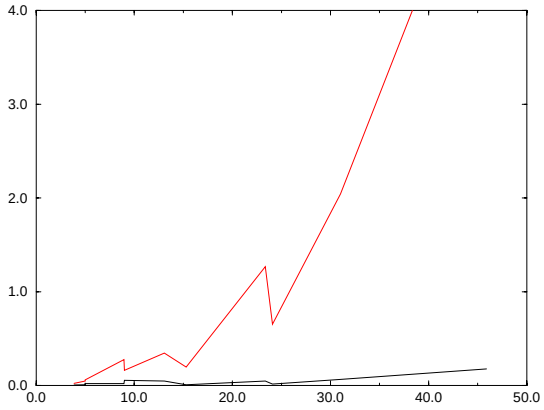


Figure 3: Comparison of the two scanning strategies. The time (milliseconds) is plotted against different random block sizes (the abscissa axe represents the square root of the initial distribution block size times the final distribution block size). Each value is obtained from the mean of one thousand iterations of the same scanning on a workstation.

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 its complexity:

For the sake of clarity, we consider here a 2D-matrix, but the results can be extended obviously to more than two dimensions.

The communication cost is :

Proposition 8

$$T_{com} = \prod_{i=1..D} \frac{M_i}{P_i P'_i}$$

move and transfer operation 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, that is on average

$$\prod_{i=1..D} \frac{M_i}{P_i P_i'}$$

.

5.3 Comparison between scanning and packing/communication complexities

In the general case,

Proposition 9 *The ratio between the scanning and the copy is*

$$O\left(\sum i \frac{P \prod i \neq j \frac{P_j}{M_j}}{r_i}\right)$$

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

Proof: Just coming from the ratio of proposition 7 and 8. $\diamond$

If the "lcm" variant (cf. section 4.3) is applied, asymptotically, (and this is true even for matrices of moderate size, say $\prod M_i$ ten times bigger than the number of processors.), the redistribution cost is roughly equal to two memory copies (one when we send and one when we receive) and the transfer time of the items owned locally. The scanning is negligible.

Proposition 10 *When*

$$\prod M_i \to \infty \Rightarrow \sum i \frac{P \prod i \neq j \frac{P_j}{M_j}}{r_i} \to 0$$

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

Proposition 11 *The total complexity is* $T_{comm} = T_{scan} + T_{copy}$

6 Timing results

The experiments corroborate very well our expectations, the computing of the data sets is negligible compared to the communication and packing, and the global routine execution time is very good.

6.1 On a LAN of workstation using PVM

The PVM machine was composed of 4 workstations and the tests were ran during a Saturday night (no fever). We generate random tests in a range that is reasonable for the target algorithm using a $N \times N$ matrix ($1 \leq r < \sqrt{\frac{N}{P_{row}}}$ and $1 \leq s < \sqrt{\frac{N}{P_{col}}}$), and compare them to the LU decomposition on the same matrix size in Figure 4.

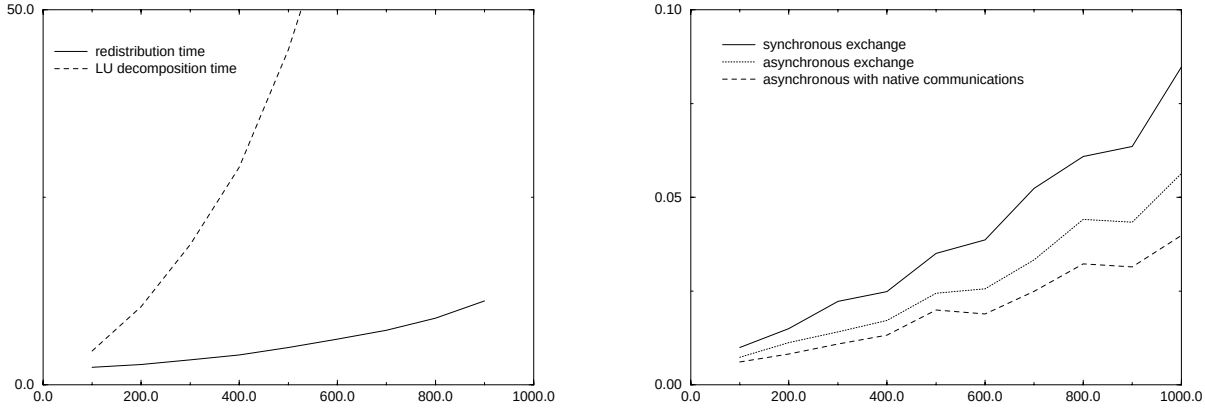


Figure 4: Timings of the redistribution tests in seconds as a function of the matrix size. On the left the average of 20 random data-distributions in seconds compared to (the best) LU decomposition on a 4 nodes LAN of SUN Sparc-ELC. On the right similar experiments with a 32 nodes Cray T3D.

The LU decomposition is by far more costly than the redistributions, moreover, the worst timing of LU decomposition is twice the one plotted while there is no big differences between redistribution times.

6.2 On a Cray T3D parallel machine

The Cray T3D proposed a home-made version of PVM based on the Cray native primitives. We show on Figure 4 the two algorithms described before implemented using this Cray PVM version and the asynchronous algorithm (best one) implemented directly with the Cray native shared memory primitives.

The results show the very good communication performances achieved by this machine, especially with the shared memory communications.

The timings are very good indeed in comparison to classical computation duration, for instance the benchmark of LU decomposition of a 1600×1600 matrix is 1.7 second on this machine[4].

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 elapsed time, the redistribution of data has to be very efficient.

Our results shows that the redistribution of data can efficiently perform in practice with our algorithms (the redistribution timings are very small compared to the computation timings corresponding to matrix operations like LU decomposition).

Our algorithms are implemented within the SCALAPACK library. They compute all redistributions and are not limited to a set of block-cyclic redistributions. They are also usable when dealing with sub-matrices (but cannot take into account strides that are not 1 nor the array leading dimension).

Our complexity analysis shows that the scanning is negligible for arrays commonly used. 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.

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

References

[AL93] 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.

[ASS93] G. Agrawal, A. Sussman, and J. Saltz. Compiler and runtime support for structured and block structured applications. pages 578–587, 1993.

[CDW92] 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*

[4](from the LINPACK benchmark database)

and Tools For Parallel Scientific Computing, pages 3–15. Elsevier, 1992.

[CGL+93] 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.

[CP94] 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.

[DGW92] 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.

[DT94] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Cmputation Subroutines. *Future Generation Computer Systems*, 10:279–284, 1994.

[HKMCS95] 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.

[KK95] 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.

[SOG94] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generating communications for array statements: design implementation and evaluation. *JPDC*, 21:150–159, 1994.

[TCF94] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in hpf programs. In *Scalable High-Performance Computing Conference*. IEEE, May 1994.