Block-cyclic Array Redistribution on Networks of Workstations*

Jack Dongarra[2,3], Frédéric Desprez[1], Antoine Petitet[2], Cyril Randriamaro[1], and Yves Robert[2]

[1] LIP, Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France
[2] Dept. of Computer Science, Univ. of Tennessee, Knoxville, TN 37996-1301, USA
[3] Math. Sciences Section, Oak Ridge Nat. Laboratory, Oak Ridge, TN 37831, USA
e-mail: [desprez, crandria]@lip.ens-lyon.fr
e-mail: [dongarra, petitet, yrobert]@cs.utk.edu

Abstract. This article deals with the run-time comparison between several algorithms (including the MPI_Alltoallv() function call) to perform the redistribution of arrays that are distributed in a block-cyclic fashion over a multidimensional processor grid. The generation of the communication messages to be exchanged by the processors involved in the redistribution is not taken into account. Rather we focus on the *scheduling* of those messages: how to organize the message exchanges into "structured" communication steps that will minimize communication overhead.
Key-words: distributed arrays, redistribution, block-cyclic distribution, scheduling, MPI, HPF, network of workstations.

1 Introduction

Run-time redistribution of arrays that are distributed in a block-cyclic fashion over a multidimensional processor grid is a difficult problem that has recently received considerable attention. This interest is motivated largely by the HPF programming style, in which scientific applications are decomposed into phases. At each phase, there is an optimal distribution of the data arrays onto the processor grid. Typically, arrays are distributed according to a `CYCLIC(r)` pattern along one or several dimensions of the grid. The best value of the distribution parameter r depends on the characteristics of the algorithmic kernel as well as on the communication-to-computation ratio of the target machine [3]. Because

* This work was supported in part by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAH04-95-1-0077, administered by the Army Research Office; by the Department of Energy Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division under Contract DE-AC05-84OR21400; by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615; by the CNRS–ENS Lyon–INRIA project *ReMaP*; and by the Eureka Project *EuroTOPS*. Yves Robert is on leave from Ecole Normale Supérieure de Lyon and is partly supported by DRET/DGA under contract ERE 96-1104/A000/DRET/DS/SR.

the optimal value of r changes from phase to phase and from one machine to another (think of a heterogeneous environment), run-time redistribution turns out to be a critical operation, as stated in [5,9,10] (among others).

Basically, we can decompose the redistribution problem into the following two subproblems:

Message generation The array to be redistributed should be efficiently scanned or processed in order to build up all the messages that are to be exchanged between processors.

Communication scheduling All the messages must be efficiently scheduled so as to minimize communication overhead. A given processor typically has several messages to send, to all other processors or to a subset of these. In terms of MPI collective operations [8], we must schedule something similar to an `MPI_Alltoallv` communication, except that each processor may send messages only to a particular subset of receivers (the subset depending on the sender).

Previous work has concentrated mainly on the first subproblem, message generation. Message generation makes it possible to build a different message for each pair of processors that must communicate, thereby guaranteeing a volume-minimal communication phase (each processor sends or receives no more data than needed). However, the question of how to efficiently schedule the messages has received little attention. Walker and Otto [9] describe how to schedule messages in order to change the array distribution from `CYCLIC(r)` on a P-processor linear grid to `CYCLIC(Kr)` on the same grid. We have extended their work in [2] in order to solve the **general** redistribution problem, that is, moving from a `CYCLIC(r)` distribution on a P-processor grid to a `CYCLIC(s)` distribution on a Q-processor grid. Our goal in this paper is to report a few experiments on a network of workstations, to demonstrate the superiority of our new redistribution algorithms over the current ScaLAPACK redistribution routine [7], as well as over a MPI all-to-all macro-communication.

In Section 2 we precisely describe the redistribution problem which we want to solve, and we provide some examples of redistribution operations which will be used for the experiences. In Section 3 we report on some MPI experiments on a network of workstations that demonstrate the usefulness of our results. Finally, in Section 4, we state some conclusions and directions for future work.

2 The redistribution problem

Consider an array $X[0...M-1]$ of size M that is distributed according to a block cyclic distribution `CYCLIC(r)` onto a linear grid of P processors (numbered from $p = 0$ to $p = P-1$). Our goal is to redistribute X using a `CYCLIC(s)` distribution on Q processors (numbered from $q = 0$ to $q = Q-1$). The new grid of Q processors can be identical to, or disjoint of, the original grid of P processors. The actual total number of processors in use is an unknown value between $\text{Max}(P, Q)$ and $(P+Q)$. For simplicity, assume that the size M of

X is a multiple of $L = lcm(Pr, Qs)$, the least common multiple of Pr and Qs: this is because the redistribution pattern repeats after each slice of L elements. Therefore, assuming an even number of slices in X will enable us (without loss of generality) to avoid discussing side effects.

We define redistribution schedules as a succession of communication steps. At each step, each sender should send no more than one message; symmetrically, each receiver should receive no more than one message. More precisely, consider a redistribution with parameters r, s, P, and Q:

- The **communication grid** is a $P \times Q$ table with a nonzero entry $length(p, q)$ in position (p, q) if and only if p has to send a message to q. Message lengths are given for a single slice vector, i.e. a vector of $L = lcm(Pr, Qs)$ components.
- A **communication step** is a collection of pairs $\{(p_1, q_1), (p_2, q_2), \ldots, (p_t, q_t)\}$ such that $p_i \neq p_j$ for $1 \leq i < j \leq t$, $q_i \neq q_j$ for $1 \leq i < j \leq t$, and $length(p_i, q_i) > 0$ for $1 \leq i \leq t$. The cost of a communication step is the maximum value of its entries, in other words, $\max\{length(p_i, q_i);\ 1 \leq i \leq t\}$ In other words, the cost of each communication step is (proportional to) the length of the longest message involved in this step.
- A schedule is a succession of communication steps such that each nonzero entry in the communication grid appears in one and only one of the steps. The cost of a schedule may be evaluated in two ways:
  1. the **number of steps** NS, which is simply the number of communication steps in the schedule; or
  2. the **total cost** TC, which is the sum of the cost of each communication step (as defined above).

The motivation for evaluating schedules via their number of steps or via their total cost is detailed in [2].

Example 1 Our first example is with $P = Q = 7$ processors, $r = 3$, and $s = 2$. All communications are summarized by the communication grid in Table 1. We see that each processor has only 4 messages to send, so that a complete all-to-all communication is not required. Also, message lengths are either 1 or 2 when redistributing a single-slice vector of length 42 (messages would be of length k and $2k$ for a vector of size $42 \times k$).

Any schedule requires at least 4 steps (each step being complete), and the total cost is at least equal to 6 (the sum of message lengths in each row or column). Using the mathematical techniques described in [2], we succeed in deriving a schedule meeting both bounds, i.e. with $NS = 4$ and $TC = 6$, as illustrated in Table 1. The costs for steps (a, b, c, d) are $(1, 2, 2, 1)$. This schedule, which is guaranteed to be both step-optimal and cost-optimal, is built as follows: let $g = \gcd(Pr, Qs) = 7$. At step k, $-2 = 1 - r \leq k \leq s - 1 = 1$, processor p, $0 \leq p < P$, sends a message to processor $q = s^{-1}(pr - k) \bmod g = 4(3p - k) = 5p + 3k \bmod 7$ (s^{-1} denotes the inverse of s modulo g, $2^{-1} \bmod 7 = 4$).

Table 1. Communication schedule for $P = Q = 7$, $r = 3$, and $s = 2$. Message lengths are indicated for a vector X of size $L = 42$.

Sender/Recv.	0	1	2	3	4	5	6	Nbr. of msg.
0	2	1	-	1	2	-	-	4
1	-	1	2	-	-	2	1	4
2	2	-	-	2	1	-	1	4
3	-	2	1	-	1	2	-	4
4	1	-	1	2	-	-	2	4
5	1	2	-	-	2	1	-	4
6	-	-	2	1	-	1	2	4
Nbr. of msg.	4	4	4	4	4	4	4	

(a) Communication grid

Sender/Recv.	0	1	2	3	4	5	6
0	c	a	-	d	b	-	-
1	-	d	b	-	-	c	a
2	b	-	-	c	a	-	d
3	-	c	a	-	d	b	-
4	a	-	d	b	-	-	c
5	d	b	-	-	c	a	-
6	-	-	c	a	-	d	b

(b) Communication steps

Example 2 Our second example is with $P = Q = 8$, $r = 3$, and $s = 2$. It is similar to the previous one, in that not each processor need to communicate to every other processor. The schedule is represented in Table 2. Computing a step-optimal cost-optimal schedule is theoretically more difficult in this case (technically, this is because s is not relatively prime with $g = \gcd(Pr, Qs) = 8$. However, we know from [2] that we can use either a graph-based algorithm, that will always lead to a step-optimal schedule, or a greedy heuristic, whose aim is to minimize the total cost of the resulting schedule. At each step, the greedy schedule selects compatible pairs of senders/receivers such that the sum of the message lengths is maximized (in graph theory, we say we construct a maximum weighted matching). For instance in Table 2, the first step (step a) is composed of eight pairs of messages of length 2. The costs for steps (a, b, c, d) are $(2, 2, 1, 1)$. It turns out that the final scheduling is both step-optimal and cost-optimal[1].

Example 3 Our third and last example is with $P = Q = 6$ processors, $r = 8$, and $s = 3$. The schedule is represented in Table 3. The costs for steps (a, b, c, d, e, f) are $(5, 5, 5, 4, 4, 4)$. We include this example for the sake of fairness: since a full all-to-all communication operation must be implemented, we need at least 6 communication steps. Given the message lengths, for any step-optimal schedule, the total cost will be at least $5 + 5 + 5 + 4 + 4 + 4 = 27$ and at worst $6 \times 5 = 30$. There is thus little space for improvement over a standard all-to-all macro-communication. Still, our schedule is both step optimal and cost-optimal.

[1] We would need to use more workstations than we actually have at our disposal, to find examples for which the graph-based algorithm or the greedy algorithm would not lead to optimal solutions. See [2] for such an example with $P = 15$, $Q = 6$, $r = 2$ and $s = 3$.

Table 2. Communication schedule for $P = Q = 8$, $r = 3$, and $s = 2$. Message lengths are indicated for a vector X of size $L = 48$.

Sender/Recv.	0	1	2	3	4	5	6	7	Nbr. of msg.
0	2	1	-	-	2	1	-	-	4
1	-	1	2	-	-	1	2	-	4
2	1	-	-	2	1	-	-	2	4
3	1	2	-	-	1	2	-	-	4
4	-	-	2	1	-	-	2	1	4
5	2	-	-	1	2	-	-	1	4
6	-	2	1	-	-	2	1	-	4
7	-	-	1	2	-	-	1	2	4
Nbr. of msg.	4	4	4	4	4	4	4	4	

(a) Communication grid

Sender/Recv.	0	1	2	3	4	5	6	7
0	a	c	-	-	b	d	-	-
1	-	d	b	-	-	c	a	-
2	d	-	-	b	c	-	-	a
3	c	b	-	-	d	a	-	-
4	-	-	a	d	-	-	b	c
5	b	-	-	c	a	-	-	d
6	-	a	d	-	-	b	c	-
7	-	-	c	a	-	-	d	b

(b) Communication steps

Table 3. Communication schedule for r $P = Q = 6$, $r = 8$, and $s = 3$. Message lengths are indicated for a vector X of size $L = 144$.

Sender/Recv.	0	1	2	3	4	5	Nbr. of msg.
0	5	3	5	3	5	3	6
1	4	4	4	4	4	4	6
2	3	5	3	5	3	5	6
3	5	3	5	3	5	3	6
4	4	4	4	4	4	4	6
5	3	5	3	5	3	5	6
Nbr. of msg.	6	6	6	6	6	6	

(a) Communication grid

Sender/Recv.	0	1	2	3	4	5
0	a	d	c	e	b	f
1	b	f	e	c	d	a
2	e	c	d	a	f	b
3	c	e	b	f	a	d
4	f	b	a	d	c	e
5	d	a	f	b	e	c

(b) Communication steps

3 MPI Experiments

This section presents the results for runs on a network of SPARC workstations. All the machines are connected by one bus, separated from the global network with a hub. We use the MPI communication library, with the programming environment LAM 6.1.

3.1 Description

Experiments have been performed on a network of workstations SPARC 4 and 5, with a C program calling routines from the MPI library. MPI has been chosen for portability and re-usability reasons. Schedules are composed of steps, and each step generates at most one send and/or one receive per processor. Hence we used only one-to-one communication primitives from MPI, namely MPI_Send() and MPI_Receive(). No explicit buffering function is called (like MPI_Bsend()) to avoid a loss in performance.

Our goal was to compare three redistribution schemes:

- The first scheme is a simple call to the MPI_Alltoallv() function from MPI libraries, which implements a full all-to-all communication with variable message sizes.
- The second scheme, namely the "caterpillar algorithm", is the current redistribution algorithm of ScaLAPACK [7]: a total exchange is organized between processors, which are arranged as a (virtual) caterpillar. The total exchange is implemented as a succession of steps. At each step, processors are arranged into pairs that perform a send/receive operation. Then the caterpillar is shifted so that new exchange pairs are formed.
- The final scheme corresponds to the communication schedules described in the previous section.

To run our scheduling algorithm, we proceed as follows: (i) Compute schedule steps using the results of Section 2; (ii) Pack all the communication buffers; (iii) Perform a barrier synchronization; (iv) Start the timer; (v) Execute the communications using either the MPI_Alltoallv() function, or the caterpillar algorithm, or our redistribution algorithms; (vi) Stop the timer; (vii) Unpack all buffers; (viii) Check that the redistribution is correct. The maximum of the timers is taken over all processors. We emphasize that we do not take the cost of message generation into account: we compare communication costs only.

We use the same physical processors for the input and the output processor grid. Results are not very sensitive to having the same grid, or disjoint grids for senders and receivers.

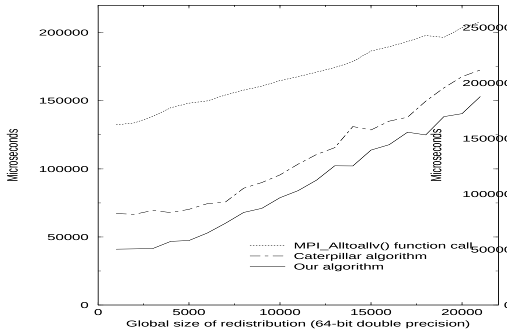
3.2 Results

Back to Example 1 In Figure 1(a), we see as expected that there is a significant difference between the caterpillar and our redistribution algorithm. This is due to the fact that a full all-to-all is not required to perform the redistribution. Hence, during each step of the caterpillar algorithm, there are some idle processors, which must wait for other processors which are still communicating.

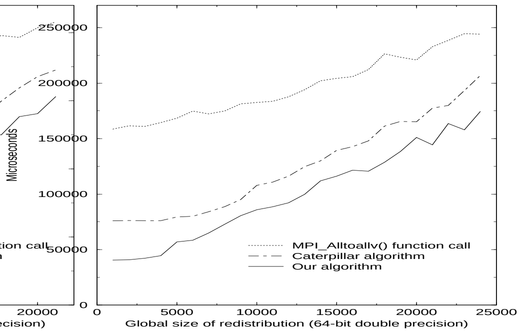
The caterpillar algorithm generates 7 steps of communication, each of cost 2. Our redistribution algorithm has 4 steps and a total cost of 6, hence the potential gain lies between $4/7 \approx 0.53$ for short messages (the ratio of the number of start-ups) and $6/14 \approx 0.43$ for long messages (the ratio of total message lengths).

The best experimental ratio is 0.6. We pay some overhead for network contentions (each step of our algorithm has more messages than each step in the caterpillar). Also, a more accurate simulation of caterpillar algorithm shows that its total is only 13: since there is no synchronization barrier between each step, steps can be partially overlapped.

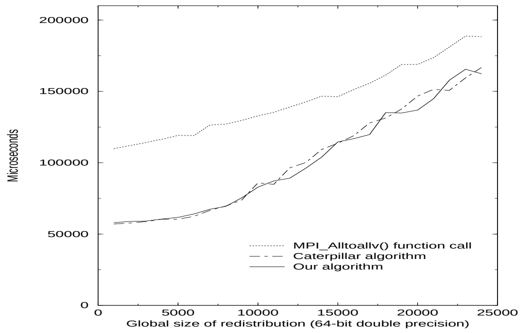
The MPI_Alltoallv() function call has the largest cost. Because communications are performed via TCP/IP, the MPI_Alltoallv() function cannot use low-level routines. We must pay some overhead for system calls and routing tables computation.



(a) Redistribution from $P = 7$, $r = 3$ to $Q = 7$, $s = 2$.



(b) Redistribution from $P = 8$, $r = 3$ to $Q = 8$, $s = 2$.



(c) Redistribution from $P = 6$, $r = 8$ to $Q = 6$, $s = 3$.

Fig. 1. Time measurements for MPI_Alltoallv(), caterpillar and our new schedule for different vector sizes.

Back to Example 2 Figure 1(b) shows results which are similar to Example 1. The caterpillar algorithm generates 8 steps, each of cost 2. Our redistribution algorithm has 4 steps and a total cost of 6, hence the potential gain lies between $4/8 = 0.5$ for short messages and $6/16 \approx 0.38$ for long messages. The best experimental ratio we obtain is 0.54. Here, the problem of contentions is amplified by the number of processors, and simulating the caterpillar algorithm leads to a communication cost of 14.

In both examples, we can say that experimental results are in good accordance with theoretical predictions, even though contention obviously plays a role. Also, modelling asynchronous macro-communications such as the caterpillar is not so easy.

Back to Example 3 As expected in this example (see Figure 1(c)), our redistribution scheme is quite comparable to the caterpillar. We pay the same number of communication start-ups, because both algorithms have the same number of steps. The totlal cost of our algorithm is 27, and the caterpillar one is 29 (result from a simulation).

4 Conclusion

This paper extends our previous paper [2] and provides additional evidence of the interest of sophisticated redistribution schedules. The experimental comparison with the caterpillar algorithm is encouraging, even though we noted that rough performance estimators such as the number of steps and the total cost can be impaired by contentions and asynchronism.

Future work will be devoted to finding schedules that take contentions into account, and to further exploring both the theoretical and experimental performances of simple heuristics such as the greedy algorithm.

References

1. J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996. (also LAPACK Working Note #95).
2. Frédéric Desprez, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert. Scheduling block-cyclic array redistribution. Research Report 97-349, Computer Science Department, University of Tennessee at Knoxville, February 1997. Also available as LAPACK Working Note 120, http://www.netlib.org/lapack/lawns.
3. J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.
4. Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins, 2 edition, 1989.
5. E. T. Kalns and L. M. Ni. Processor mapping techniques towards efficient data redistribution. *IEEE Trans. Parallel Distributed Systems*, 6(12):1234–1247, 1995.
6. Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.
7. L. Prylli and B. Tourancheau. Efficient block-cyclic data redistribution. In *EuroPar'96*, volume 1123 of *Lectures Notes in Computer Science*, pages 155–164. Springer Verlag, 1996.
8. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the complete reference*. The MIT Press, 1996.
9. David W. Walker and Steve W. Otto. Redistribution of block-cyclic data distributions using MPI. *Concurrency: Practice and Experience*, 8(9):707–728, 1996.
10. Lei Wang, James M. Stichnoth, and Siddhartha Chatterjee. Runtime performance of parallel array assignment: an empirical study. In *1996 ACM/IEEE Supercomputing Conference*. `http://www.supercomp.org/sc96/proceedings`, 1996.