Efficient Algorithms for Block-Cyclic Array Redistribution between Processor Sets

Neungsoo Park, Viktor K. Prasanna
and C. S. Raghavendra

CENG 98-22

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4483)
September 1998

Efficient Algorithms for Block-Cyclic Array Redistribution between Processor Sets

Neungsoo Park* and Viktor K. Prasanna*
Department of EE-Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562
{neungsoo + prasanna}@halcyon.usc.edu
http://ceng.usc.edu/~prasanna/

C. S. Raghavendra
The Aerospace Corporation
P.O. Box 92957
Los Angeles, CA 90009-2957
raghu@rush.aero.org

Abstract

Run-time array redistribution is necessary to enhance the performance of parallel programs on distributed memory supercomputers. In this paper, we present an efficient algorithm for array redistribution from *cyclic*(x) on P processors to *cyclic*(Kx) on Q processors. The algorithm reduces the overall time for communication by considering the data transfer, communication schedule, and index computation costs. The proposed algorithm is based on a generalized circulant matrix formalism. Our algorithm generates a schedule that minimizes the number of communication steps and eliminates node contention in each communication step. The network bandwidth is fully utilized by ensuring that equal-sized messages are transferred in each communication step. Furthermore, the procedure to compute the schedule and the index sets is extremely fast. It takes $O(max(P,Q))$ time. Therefore, our proposed algorithm is suitable for run-time array redistribution. To evaluate the performance of our scheme, we have implemented the algorithm using C and MPI. The experiments were conducted on the IBM SP2. The experimental results show that the proposed algorithm outperforms well-known algorithms with respect to the total redistribution time including the data transfer and schedule and index computation times.

*This work was supported in part by the US DoD High Performance Computing Modernization Office and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0016. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

1 Introduction

Many High Performance Computing (HPC) applications, including scientific computing and signal processing, consist of several stages [15, 21, 25, 26]. Examples of such applications include the multidimensional Fast Fourier Transform, the Alternative Direction Implicit (ADI) method for solving two-dimensional diffusion equation, and linear algebra solvers. While executing these applications on a distributed memory supercomputer, data distribution is needed for each stage to reduce the performance degradation due to remote memory accesses. As the program execution proceeds from one stage to another, the data access patterns and the number of processors required for exploiting the parallelism in the application may change. These changes usually cause the data distribution in a stage to be unsuitable for the subsequent stage. Data redistribution relocates the data in the distributed memory to reduce the remote access overheads. Since the parameters of redistribution are generally unknown at compile time, run-time data redistribution is necessary. However, the cost of redistribution can offset the performance benefits that can be achieved by the redistribution. Therefore, run-time redistribution must be implemented efficiently to ensure overall performance improvement.

Array data are typically distributed in a *block-cyclic* pattern onto a given set of processors. The block-cyclic distribution with block size x is denoted as *cyclic*(x). A block contains x consecutive array elements. Blocks are assigned to processors in a round-robin fashion. Other distribution patterns, *cyclic* and *block* distribution, are special cases of the block-cyclic distribution. In general, the block-cyclic array redistribution problem is to reorganize an array from one block-cyclic distribution to another, *i.e.*, from *cyclic*(x) to *cyclic*(y). An important case of this problem is redistribution from *cyclic*(x) to *cyclic*(Kx) which arises in many scientific and signal processing applications. This type of data redistribution can occur within a same processor set, or between different processor sets.

Figure 1(a) shows Array **A** with $N = 24$ elements. In the initial distribution of **A**, the block size is 2. The blocks are assigned to $P = 3$ processors in a round-robin fashion as shown in Figure 1(b). If the block size is expanded by a factor of $K = 2$, the new block size becomes 4. Two consecutive blocks are combined into a new block of size 4. These new blocks are then assigned to 3 processors as shown in Figure 1(c). This is an example of the redistribution problem from *cyclic*(x) to *cyclic*(Kx) on the same set of processors. Redistribution from *cyclic*(x) on P processors to *cyclic*(Kx) on Q processors requires the blocks to be redistributed over a different set of processors. Figure 1(d)

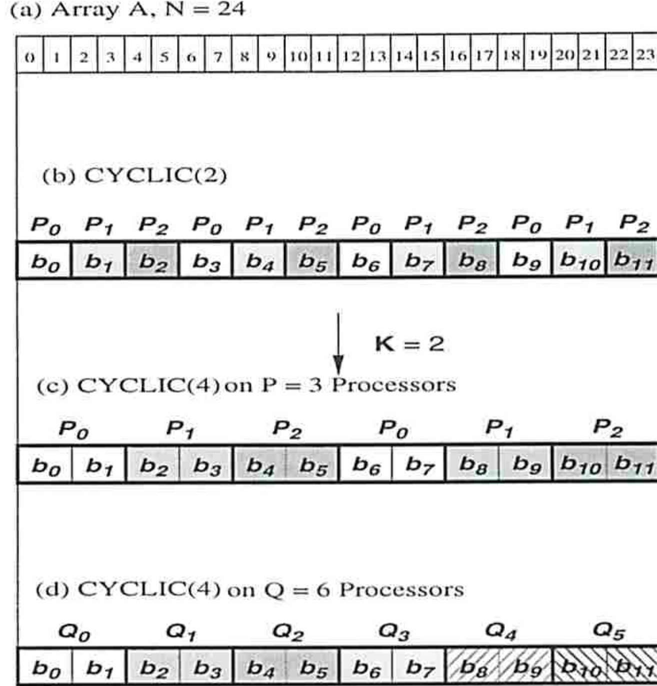


Figure 1: Example of block-cyclic redistributions from *cyclic*(x) on P processors to *cyclic*(Kx) on P and Q processors

shows an example of *cyclic*(4) distribution on $Q = 6$ processors.

Data redistribution from a given initial layout to a final layout consists of four major steps - index computation, communication schedule, message packing and unpacking, and finally data transfer. With a given initial and final data layout, each processor determines the local indices of the data blocks that belong in its memory initially and at the end of redistribution. This calculation from the global indices to local indices of data blocks is called index computation. Data blocks in a source processor need to be moved to destination processor depending on the required final data layout. The parallel communications of data blocks among processors will be determined by a communication schedule. For efficient communication of data blocks, a source processor should pack all blocks destined to a destination processor in one message. Likewise, a destination processor will unpack a received message and place the data blocks in appropriate local memory locations. Finally, the transfer of data blocks over the network incurs a fixed start-up cost plus a transmission cost proportional to the size of a message. All these four steps contribute to the total array redistribution cost.

Table 1 summarizes the key features of the well known data distribution algorithms in the

Table 1: Comparison of various schemes for array redistribution

	Key Features	
	Schedule & Index Computation	Communication
PITFALLS [20]	• No communication scheduling • Index computation using a line segment formalism	• Node contention occurs • Does not minimize the transmission cost
BCC [3]	• No communication scheduling • Efficient index computation • Source and destination sets are same	• Node contention occurs • Does not minimize the transmission cost
Caterpillar [19]	• Simple scheduling algorithm • Index computation by scanning the array segments	• No node contention • Does not minimize the transmission cost and the number of communication steps
Bipartite Matching Scheme [4]	• Large schedule computation overhead Schedule computation time: $O((P+Q)^4)$	• No node contention • Stepwise strategy: minimizes the number of communication steps • Greedy strategy: minimizes the transmission cost
Our Scheme	• Fast schedule and index computations Schedule computation time: $O(max(P,Q))$	• No node contention • Minimizes the number of communication steps and the data transfer cost

literature. All of the known algorithms ignore one or more of the above costs. Some schemes focus only on efficient index set computation and completely ignore scheduling the communication events [3, 9, 20, 22, 23]. Based on the index of a block, these schemes focus on finding its destination processor and generating messages for the same destination. Communication scheduling is not considered. These lead to node contention in performing the communication. This inturn leads to higher data transfer costs as some nodes incur additional delays. Other schemes eliminate node contention by explicitly scheduling the communication events [11, 14, 24]. Although the schemes in [11, 14, 24] have an efficient scheduling algorithm, these were designed for data redistribution on the same processor set. For redistribution between different processor sets, the Caterpillar algorithm was proposed in [19]. It uses a simple round robin schedule to avoid node contention. However, this algorithm does not fully utilize the network bandwidth i.e., the size of the data sent by the nodes in a communication step varies from node to node. This leads to increased data transfer cost. The schemes in [4] reduce the data transfer cost, however, the schedule computation cost is significant. The bipartite graph matching used in [4] takes $O((P+Q)^4)$ time. On a state-of-the-art workstation, this time is in the range of 100's of *msecs* for P and Q of interest. For problems of interest, the schedule computation cost is larger than the data transfer cost. The

algorithm in [4] optimizes the data transfer cost and the number of communication steps for the non all-to-all communication case (which is one of the three cases that occur in performing the redistribution considered here). The algorithm in [4] does not optimize the data transfer cost for the all-to-all communication case with different message sizes. To optimize the data transfer cost, it is necessary that the transferred messages are of equal size in each communication step.

In this paper, we propose a novel and efficient algorithm for data redistribution from $cyclic(x)$ on P processors to $cyclic(Kx)$ on Q processors. Our algorithm uses *optimal* number of communication steps and *fully* utilizes the network bandwidth in each step. The communication schedule is determined using a generalized circulant matrix framework. The schedule computation cost is $O(max(P,Q))$. Our implementations show that the schedule computation time is in the range of 100's of $\mu secs$ when P and Q are in the range 50-100. Each processor computes its own index set and its communication schedule only using a set of equations derived from our generalized circulant matrix formulation. Our experimental results show that the schedule computation time is negligible compared with the data transfer cost for array sizes of interest. The message packing/unpacking cost is the same as that of any scheme that generates an optimal communication schedule. Thus, our scheme minimizes the total time for data redistribution. This makes our scheme attractive for run-time as well as compile-time data redistribution.

Our techniques can be used for implementing scalable redistribution libraries, for implementing `REDISTRIBUTE` directive in HPF [13], and for developing parallel algorithms for supercomputer applications. In particular, these techniques lead to efficient distributed corner turn operation, a key communication kernel needed in parallelizing signal processing applications [15, 21, 25, 26].

Our redistribution scheme has been implemented using MPI and C. It can be easily ported to various HPC platforms. We have performed several experiments to illustrate the improved performance compared with the state-of-the-art. The experiments were performed to determine the data transfer, schedule and index computation costs. In one of these experiments, we used 64 processors on an IBM SP2 which were partitioned into 28 source processors and 36 destination processors. The expansion factor was set to 14. The array size was varied from 2.26 Mbytes to 56.4 Mbytes. Compared with the Caterpillar algorithm, our data transfer times were lower. The ratio of data transfer time of our algorithm to that of the Caterpillar algorithm was between 49.2% and 53.7%. The schedule computation time of the proposed algorithm is much less than that of the bipartite matching scheme [4]. For P and $Q \geq 64$, the schedule computation time of the bipartite

matching scheme is 100's of *msecs*, while that of our algorithm is only 100's of $\mu secs$. For example, when $P = 64$, $Q = 64$, and $K = 32$, the schedule computation time using the bipartite matching scheme is 133.2 *msecs* while that using our algorithm is 178.6 $\mu secs$

The rest of this paper is organized as follows. Section 2 defines the array redistribution problem and reviews prior work. Section 3 explains our table-based framework. It also discusses the generalized circulant matrix formalism for deriving conflict free communication schedules. Section 4 explains our redistribution algorithm and index computation in detail and gives the proofs of their correctness and their complexity. Section 5 reports our experimental results on the IBM SP-2. Concluding remarks are made in Section 6.

2 Background and Related Work

2.1 Problem definition

The block-cyclic distribution, *cyclic*(x), of an array is defined as follows: given an array with N elements, P processors, and a block size x, the array elements are partitioned into contiguous blocks of x elements each. The ith block, b_i, consists of array elements whose indices vary from ix to $(i+1)x - 1$, where i is a global block index and $0 \leq i < \frac{N}{x}$. These blocks are distributed onto P processors in a round-robin fashion. Block b_i is assigned to processor j, P_j, where $j = i \bmod P$.

In this paper, we study the problem of redistributing from *cyclic*(x) on P processors to *cyclic*(Kx) on Q processors, which is denoted as $\Re_x(P, K, Q)$. Figure 1 (d) shows $\Re_2(3, 2, 6)$. Here, initially pairs of consecutive elements of the array are distributed over $P = 3$ source processors. Then, the block size is doubled and the new blocks (of size 4) are redistributed over $Q = 6$ destination processors.

In performing $\Re_x(P, K, Q)$, three classes of communication patterns between source and destination processors arise. One case is the non $all-to-all$ communication: every source processor communicates with some of the destination processors. This case is shown in Table 2, where $N = 30x$, $P = 6$, $Q = 10$ and $K = 3$. Note that the messages are of equal size $(1x)$. The second case is the $all-to-all$ communication with the same message size. In Table 3, all the source processors have messages of the same size $(2x)$ and each source processor communicates with all the destination processors. Here, $N = 120x$, $P = 6$, $Q = 10$ and $K = 4$. The last case is the $all-to-all$ communication with different message sizes. This is shown in Table 4 where $N = 120x$, $P = 8$, $Q = 10$, and $K = 6$. Here half the messages are of size $1x$ while others are of size $2x$.

Table 2: An example of non all-to-all communication. $P = 6$, $Q = 10$ and $K = 3$.

Source \ Destination	0	1	2	3	4	5	6	7	8	9
0	1		1		1		1		1	
1	1		1		1		1		1	
2	1		1		1		1		1	
3		1		1		1		1		1
4		1		1		1		1		1
5		1		1		1		1		1

Table 3: An example of all-to-all communication with same message size. $P = 6$, $Q = 10$, and $K = 4$.

Source \ Destination	0	1	2	3	4	5	6	7	8	9
0	2	2	2	2	2	2	2	2	2	2
1	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2
3	2	2	2	2	2	2	2	2	2	2
4	2	2	2	2	2	2	2	2	2	2
5	2	2	2	2	2	2	2	2	2	2

Table 4: An example of all-to-all communication with different message size. $P = 8$, $Q = 10$, and $K = 6$.

Source \ Destination	0	1	2	3	4	5	6	7	8	9
0	2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2	1
2	1	2	1	2	1	2	1	2	1	2
3	1	2	1	2	1	2	1	2	1	2
4	2	1	2	1	2	1	2	1	2	1
5	2	1	2	1	2	1	2	1	2	1
6	1	2	1	2	1	2	1	2	1	2
7	1	2	1	2	1	2	1	2	1	2

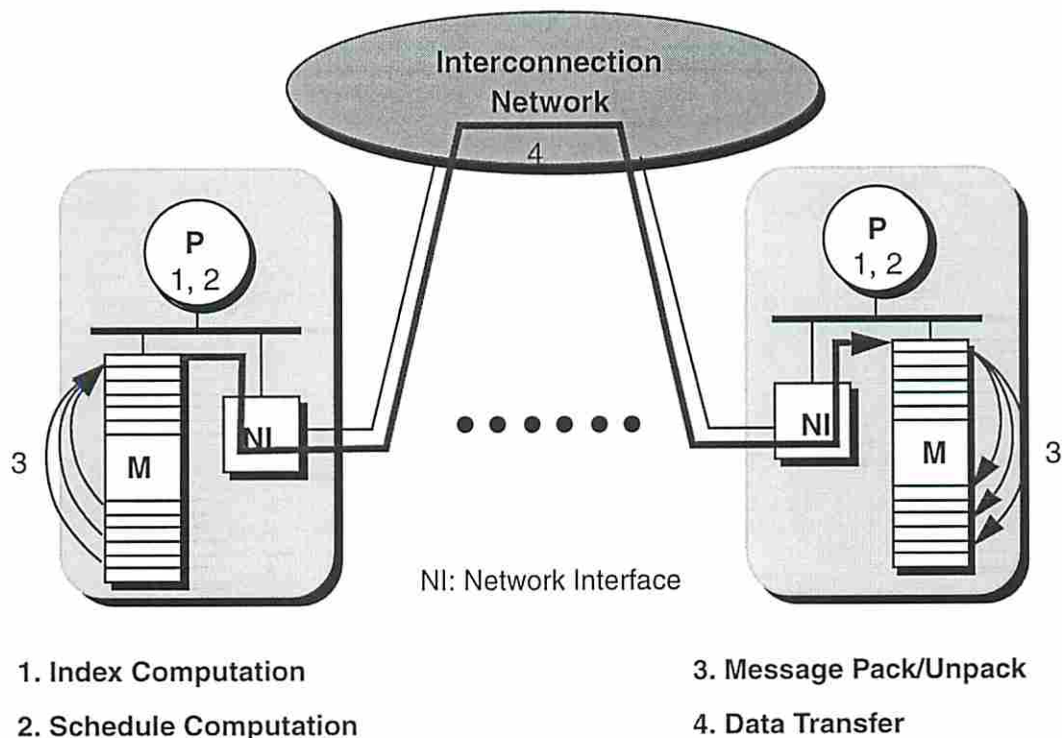


Figure 2: Steps in performing redistribution

2.2 Cost of performing redistribution

We briefly explain the four costs associated with data redistribution:

Index Computation Cost: Each source processor determines the destination processor indices of the array elements that belong to it and computes the local memory location (local index) of each array element. This local index is used to pack array elements into a message. Similarly, each destination processor determines the source processor indices of received messages and computes their local indices to find out the location where the received message is to be stored. The total time to compute these indices is denoted as index computation cost.

Schedule Computation Cost: The communication schedule specifies a collection of sender-receiver pairs for each communication step. Since in each communication step, a processor can send at most one message and a processor can receive at most one message, careful scheduling is required to avoid contention while minimizing the number of communication steps. Time to compute this communication schedule can be significant. Reducing this cost is an important criteria in performing run-time redistribution.

Message Packing/Unpacking Cost: At each sender, a message consists of words from different memory locations which need to be gathered in a buffer in the sending node. Typically, this requires

a memory read and a memory write operation to gather the data to form a compact message in the buffer. The time to perform this data gathering at the sender is the message packing cost. Similarly, at the receiving side, each message is to be unpacked and data words need to be stored in appropriate memory locations.

Data Transfer Cost: The data transfer cost for each communication step consists of start-up cost and transmission cost. The start-up cost is incurred by software overheads in each communication operation. The total start-up cost can be reduced by minimizing the number of message transfer steps. The transmission cost is incurred in transferring the bits over the network and depends on the network bandwidth.

2.3 Related work

The array redistribution problem has been the focus of several research efforts [3, 4, 9, 11, 20, 22, 23, 24]. Many of these efforts have targeted efficient implementation of high level compiler directives such as the `REDISTRIBUTE` directive in HPF.

In [20], Banerjee *et al.* use a line segment formalism called PITFALLS to represent the *cyclic*(x) distribution. The array elements that map to a processor are represented as a set of stride line segments. For every pair of processors, the array elements whose indices are in the intersection of the respective line sets are exchanged. Their technique handles arbitrary source and destination processor sets as well as multidimensional arrays. However, PITFALLS does not consider communication scheduling during redistribution. Their main contribution is to determine the elements to be exchanged.

Other previous studies [3, 9, 11, 22, 23, 24] consider redistribution from *cyclic*(x) to *cyclic*(Kx) on the same set of processors. This is a classical redistribution problem. Techniques to compute index sets are proposed in [9, 22, 23]. However, explicit scheduling of the communication to minimize overall redistribution time is not considered.

In [22, 23], Choudhary *et al.* present efficient index computation algorithms for the special case when $P \bmod K \equiv 0$. They also consider redistribution from *cyclic*(x) to *cyclic*(y) on the same processor set, for arbitrary x and y. Although it is possible to explicitly calculate the destination and source processor of each element of the local array, such a scheme is expensive for use in practice as potential node contentions occur in each communication step. In [23], *gcd* and *lcm* methods have been proposed. These are two phase algorithms where *cyclic*(x) is first redistributed to *cyclic*(m), followed by a redistribution from *cyclic*(m) to *cyclic*(y). Here, m can be *gcd* or *lcm* of x and y. In

[23], it is shown that multidimensional arrays can be redistributed by applying these algorithms to each dimension of the array separately.

In [9], Ni *et al.* provide new logical processor numbers (*lpids*) for the target distribution, so as to minimize the amount of data to be communicated during redistribution. Data blocks which have the same *lpids* across source and target distributions, need not be moved. However, index set computation becomes complicated. This approach is not applicable when the number of source and target processors are different.

Sadayappan *et al.* [11] and Walker *et al.* [24] have proposed algorithms which reduce communication overheads. In [24], a K step schedule is derived for *cyclic*(x) to *cyclic*(Kx) redistribution on the same processor set. In each step, processors exchange data in a contention free manner: each processor sends its data to exactly one processor and receives data from exactly one processor. Therefore contention at the destination nodes does not occur. A similar communication schedule is shown in [11]. Although the communication schedule presented in [24] is based on modular arithmetic and that in [11] is based on tensor products, the resulting communication schedules are similar.

In [4], Desprez *et al.* propose a general solution for block cyclic redistribution on arbitrary source and destination processor sets. They can handle redistribution from *cyclic*(x) on P processors to *cyclic*(y) on Q processors, where x, y, P, and Q are arbitrary positive integers. Their method to compute the communication schedule uses bipartite matching. They propose two strategies, stepwise strategy and greedy strategy. In stepwise strategy, they try to minimize the number of communication steps but the total data transfer cost is not optimized. In greedy strategy, the total transmission cost is optimized but the number of communication steps is not minimized. Even though they reduced the data transfer time, the time to compute the communication schedule using bipartite matching is significant. The schedule computation cost is $O((|P| + |Q|)^4$. As the number of processors is increased, the schedule computation time can be larger than the data transfer cost.

In [3], Y.C. Chung *et al.*have proposed the index computation method for redistribution from *cyclic*(x) to *cyclic*(y) on the same processor set. They proposed the basic-cycle calculation (denoted as BCC) technique which is the closed forms for source/destination processor indices of array elements. These closed forms are useful to efficiently determine the communication sets of a basic-cycle. They did not consider the communication schedule in this technique. Therefore, the node

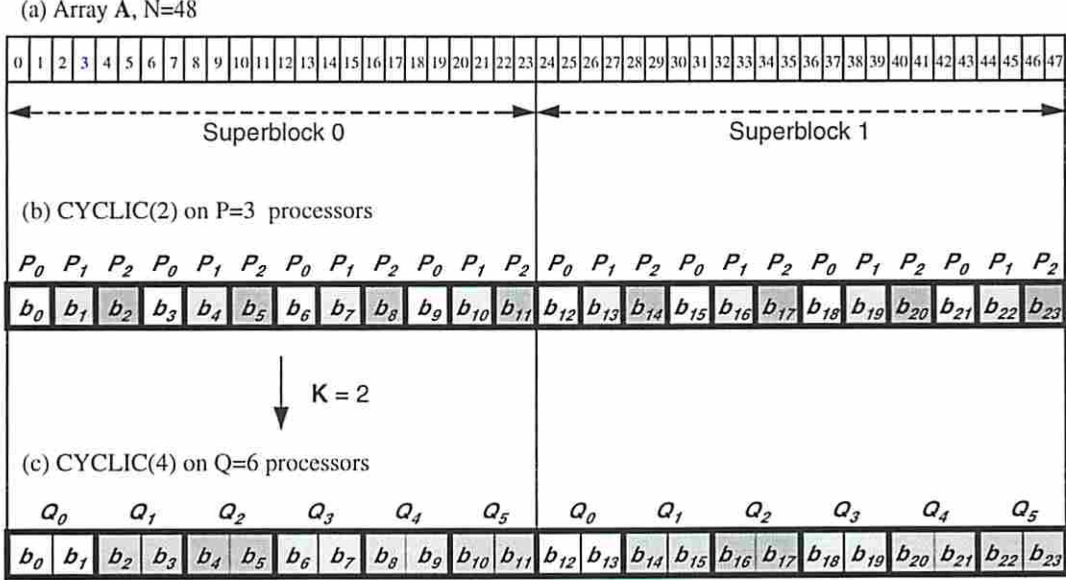


Figure 3: Block-Cyclic Redistribution from array point of view: (a) the array of elements, (b) $cyclic(x)$ on P processors, (c) from $cyclic(x)$ on P processors to $cyclic(Kx)$ on Q processors. In this example, $x = 2$, $P = 3$, $Q = 6$ and $K = 2$.

contention problems exist on the redistribution communications.

3 Our Approach to Redistribution

In this section, we present our approach to block-cyclic redistribution problem. In subsection 3.1, we discuss two views of redistribution and illustrate the concept of a superblock. In the following subsection, we explain our table-based framework for redistribution using the destination processor table and column and row reorganizations. In subsection 3.3, we discuss the generalized circulant matrix formalism which allows us to compute communication schedule efficiently.

3.1 Array and processor points of view

Figure 3 shows $\Re_2(3, 2, 6)$ from the *array point of view*. The elements of the array are shown along a single horizontal axis. The processor indices are marked above each block. For the redistribution $\Re_x(P, K, Q)$, a periodicity can be found in the block movement pattern. For example, in Figure 3, b_0, b_3, b_6, and b_9, which are initially assigned to P_0, are moved to Q_0, Q_1, Q_3, and Q_4 respectively. After that, b_{12} in P_0 is moved to Q_0 again. We find that the communication pattern between b_0 and b_{11} is repeated on other blocks. Such a collection of blocks is called as a superblock. The period of this block movement pattern is $lcm(P, KQ)$, and is the size of the superblock. In Figure 3, superblock size is $lcm(3, 2 \cdot 6) = 12$. In the next superblock, blocks b_{12} to b_{23} are moved

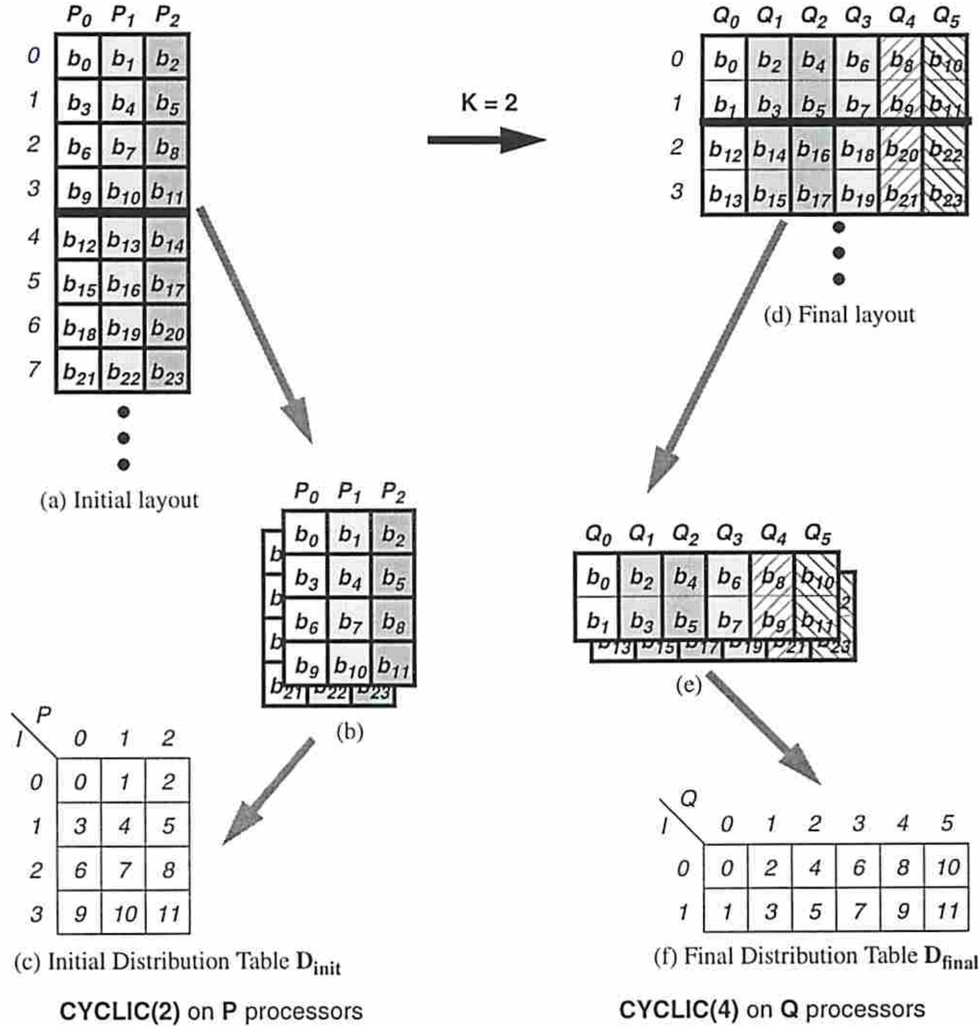


Figure 4: Block-Cyclic Redistribution from $cyclic(x)$ on P processors to $cyclic(Kx)$ on Q processors from processor point of view. In this example, $P = 3$, $Q = 6$ and $K = 2$.

in the same fashion.

From the *processor point of view*, the block-cyclic distribution can be represented by a 2-dimensional table. Each column corresponds to a processor and each row index is a local block index. Each entry in the table is a global block index. Therefore, element (i, j) in the table represents the ith local block of the jth processor. Figure 4 shows the example of $\Re_2(3, 2, 6)$ from the processor point of view. Blocks are distributed on the table in a round-robin fashion. The table corresponding to source processors is denoted as initial layout representing which blocks are initially assigned to which source processors. Similarly, the final layout represents which blocks are assigned to which destination processors. Our problem is to redistribute the blocks from initial layout to final layout. These layouts are shown in Figure 4(a) and (d) respectively.

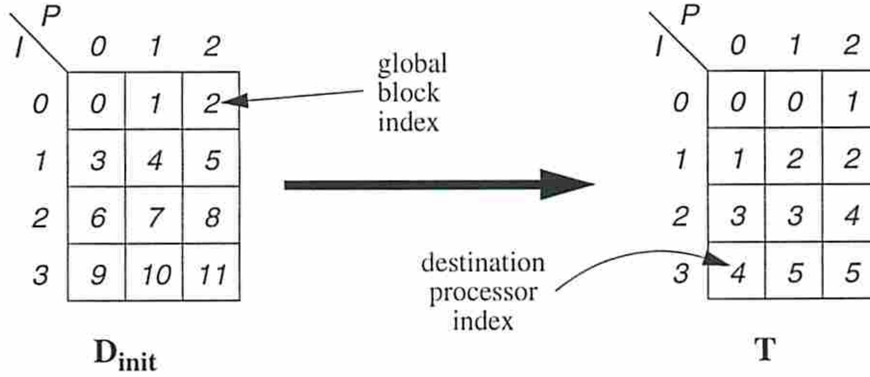


Figure 5: The destination processor table **T**

The initial layout can be partitioned into collections of rows of size $L_s = lcm(P, KQ)/P$. Similarly, the final layout can be partitioned into disjoint collections of rows; each collection having $L_d = lcm(P, KQ)/Q$ rows. Note that each collection corresponds to a superblock. Blocks, which are located at the same relative position within a superblock, are moved in the same way during the redistribution. These blocks can be transferred in a single communication step. The MPI derived data type can handle these blocks as a single block. Without loss of generality, we will consider only the first superblock in the following to illustrate our algorithm. We refer to the tables representing the indices of the blocks within the first superblock in the initial (final) layout as *initial distribution table* $\mathbf{D_{init}}$ (*final distribution table* $\mathbf{D_{final}}$). These are shown in Figure 4(c) and (f), respectively. The cyclic redistribution problem essentially involves reorganizing blocks within each superblock from an initial distribution table $\mathbf{D_{init}}$ to a final distribution table $\mathbf{D_{final}}$.

3.2 A table-based framework for redistribution

Given the redistribution parameters, P, K, and Q, each block's location in $\mathbf{D_{init}}$ and $\mathbf{D_{final}}$ can be determined. Through redistribution, each block moves from its initial location in $\mathbf{D_{init}}$ to the final location in $\mathbf{D_{final}}$. Thus, the processor ownership and the local memory location of each block are changed by redistribution. This redistribution can be conceptually considered as a table conversion process from $\mathbf{D_{init}}$ to $\mathbf{D_{final}}$, which can be decomposed into independent column and row reorganizations. In a column reorganization, blocks are rearranged within a column of the table. This is therefore a local operation within a processor's memory. In a row reorganization, blocks within a row are rearranged. This operation therefore leads to a change in ownership of the blocks, and requires interprocessor communication.

The destination processor of each block in the initial distribution table is determined by the redistribution parameters and its global block index. A send communication events table is con-

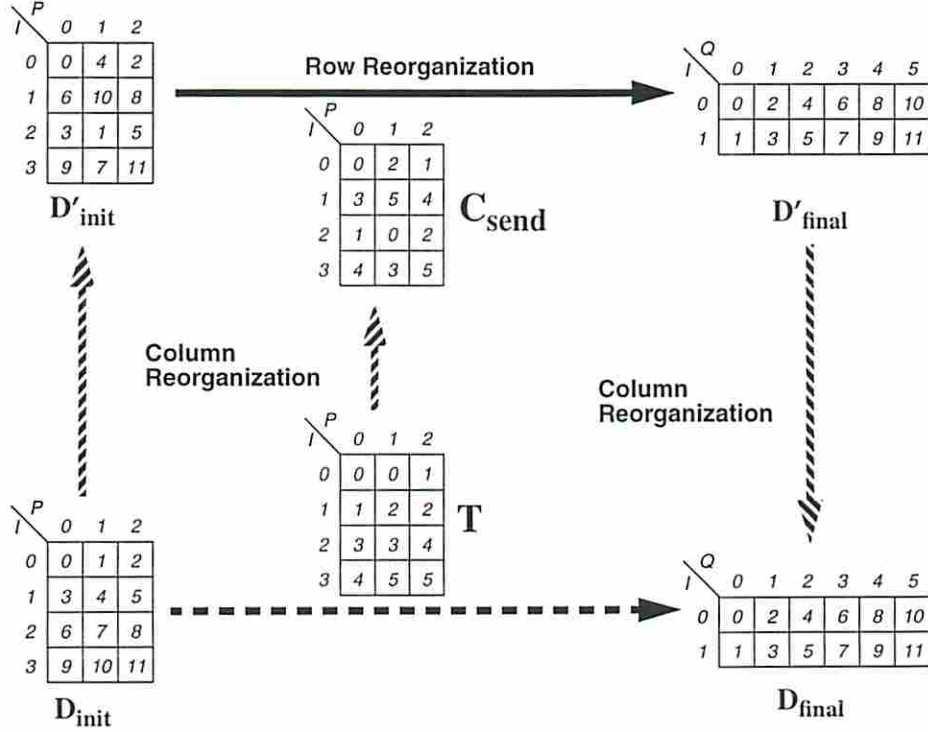


Figure 6: Table conversion process for redistribution

structed by replacing each block index in the initial distribution table with its destination processor index as shown in Figure 5. This is denoted as *destination processor table* (*dpt*) $\mathbf{T}$. The $(i, j)^{th}$ entry of $\mathbf{T}$ is the destination processor index of ith local block in source processor j and $0 \leq i < L_s$ i.e. $\mathbf{T}$ considers only one superblock. It is a $L_s \times P$ matrix. Each row corresponds to a communication step. On the other hand, each destination processor has to know from which source processors it receives messages. Source processor index of each block in the final distribution is also determined by the redistribution parameters and its global block index. A receive communication events table can be constructed by replacing each global block index in $\mathbf{D_{final}}$ with its source processor index. In our algorithm, during a communication step, a processor sends data to atmost one destination processor. If $Q \geq P$, atmost P processors in the destination processor set can receive data and the other destination processors remain idle during that communication step. Therefore, each communication step can have at most P communicating pairs. On the other hand, if $Q < P$, only Q destination processors can receive data at a time. The maximum number of communicating pairs in a communication step is $min(P, Q)$. Without loss of generality, in the following discussion we assume that $Q \geq P$.

Figure 6 shows our table-based framework for redistribution. To convert the initial distribution

table $\mathbf{D_{init}}$ to the final distribution table $\mathbf{D_{final}}$, *dpt* $\mathbf{T}$ can be used. But, the use of $\mathbf{T}$ itself as a communication schedule is not efficient. It leads to node contention, since several processors try to send their data to the same destination processor in a communication step. For example, in Figure 6, during step 0, both source processors 0 and 1 try to communicate with destination processor 0. However, if every row of $\mathbf{T}$ consists of P distinct destination processor indices among $\{0, 1, \cdots, Q-1\}$, node contention can be avoided in each communication step. This is the motivation for the column reorganizations.

To eliminate node contention, the *dpt* $\mathbf{T}$ is reorganized by column reorganizations. The reorganized table is called a *send communication schedule table*, $\mathbf{C_{send}}$. In section 4, we discuss how these reorganizations are performed. $\mathbf{C_{send}}$ is a $L_s \times P$ matrix as well. Each entry of $\mathbf{C_{send}}$ is a destination processor index and each row corresponds to a contention-free communication step. To maintain the correspondence between $\mathbf{D_{init}}$ and $\mathbf{T}$, the same set of column reorganizations is applied to $\mathbf{D_{init}}$ which results in a distribution table, $\mathbf{D'_{init}}$ corresponding to $\mathbf{C_{send}}$. In a communication step, blocks in a row of $\mathbf{D'_{init}}$ are transferred to their destination processors specified by the corresponding entries in $\mathbf{C_{send}}$. Referring to Figure 6, in the first communication step, source processors 0, 1 and 2 transfer blocks 0, 4 and 2 to destination processors 0, 2 and 1 respectively as specified by $\mathbf{C_{send}}$. Such a step is called row reorganization. The distribution table $\mathbf{D'_{final}}$ corresponding to the received blocks in destination processors is reorganized into the final distribution table $\mathbf{D_{final}}$ by another set of column reorganizations. (For this example, we do not need this operation.) The received blocks are then stored in the memory locations of the destination processors. The key idea is to choose a $\mathbf{C_{send}}$ such that the required row reorganizations(communication events) can be performed efficiently and it supports easy-to-compute contention-free communication scheduling.

So far, we have discussed a redistribution problem from *cyclic*(x) on P processors to *cyclic*(Kx) on Q processors. A dual relationship exists between the problem from *cyclic*(x) on P processors to *cyclic*(Kx) on Q processors and the problem from *cyclic*(Kx) on Q processors to *cyclic*(x) on P processors. The redistribution from *cyclic*(Kx) on Q processors to *cyclic*(x) on P processors is the redistribution with reverse direction of the redistribution $\Re_x(P, K, Q)$. Its send (receive) communication schedule table is the same as the receive (send) communication schedule table of $\Re_x(P, K, Q)$. Therefore, our scheme for $\Re_x(P, K, Q)$ can be extended to the redistribution problem from *cyclic*(Kx) on Q processors to *cyclic*(x) on P processors.

In the following subsection, we define generalized circulant matrix. Using the generalized cir-

0	3	2	1
1	0	3	2
2	1	0	3
3	2	1	0

(a) $m = n = 4$

0	3	2	1
1	0	3	2
2	1	0	3

(b) $m = 3, n = 4$

0	3	2
1	0	3
2	1	0
3	2	1

(c) $m = 4, n = 3$

Figure 7: Circulant matrix examples

culant matrix, we derive efficient and contention free communication schedule for $\Re_x(P, K, Q)$.

3.3 Communication scheduling using generalized circulant matrix

Our framework for communication schedule performs local rearrangement of data within each processor as well as interprocessor communication. The local rearrangement of data, which we call column reorganization, results in a send communication schedule table $\mathbf{C_{send}}$. We will show that for any P, K and Q, the send communication schedule is indeed a generalized circulant matrix which avoids node contention.

Definition 1 *An $m \times n$ matrix is a* circulant matrix *if it satisfies the following properties:*

1. *If $m \leq n$, row k = row 0 circularly right shifted k times, $0 \leq k < m$.*
2. *If $m > n$, column l = column 0 circularly down shifted l times, $0 \leq l < n$.*

Figure 7 shows a circulant matrix. Note that the above definition can be extended to block circulant matrices by changing "row" to "row block".

Definition 2 *An $M \times N$ matrix is a* generalized circulant matrix *if the matrix can be partitioned into blocks of size $m \times n$, where $M = s \cdot m$ and $N = t \cdot n$, for some s, $t > 0$ such that the block matrix forms a circulant matrix and each block is either a circulant matrix or a generalized circulant matrix.*

Figure 8 illustrates a generalized circulant matrix. There are two observations about the generalized circulant matrix: *(i) the s blocks along each block diagonal are identical, and (ii) if all the elements in row 0 are distinct, then in each row all elements are distinct.*

We will show that through our approach the destination processor table $\mathbf{T}$ is transformed to a generalized circulant matrix $\mathbf{C_{send}}$ with distinct elements in each row.

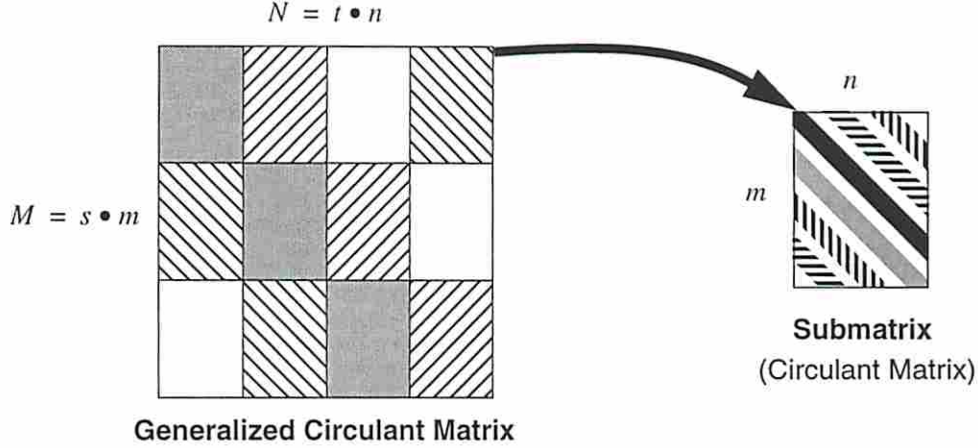


Figure 8: Generalized circulant matrix

4 Efficient Redistribution Algorithms

In this section, we discuss the communication patterns that arise in performing array redistribution, develop the algorithms for our table-based approach to obtain the send schedule, and present correctness proofs of our techniques.

4.1 Communication patterns of redistribution

Consider the movement of the global block i in the redistribution problem $\Re_x(P, K, Q)$. It is the mth local block of source processor p, where $m = i/P$ and $p = i \bmod P$.

$$i = P \times m + p \tag{1}$$

After redistribution, K consecutive global blocks become one block in the new layout. Therefore, global block i is k^{th} block in the i_K^{th} new global block, where $k = i \bmod K$ and $i_K = i/K$. The i_K^{th} new global block is at n^{th} new local block of destination processor q, where $n = i_K/Q$ and $q = i_K \bmod Q$.

$$i = (Q \times n + q) \times K + k \tag{2}$$

We can derive the following redistribution equation.

$$i = P \times m + p = (Q \times n + q) \times K + k \tag{3}$$

For example, consider the 16th global block in $\Re_x(3, 2, 6)$ of Figure 3. Here, $i = 16$, $P = 3$, $p = 1$ and $m = 5$. All block indices start from 0. Also, $K = 2$, $k = 0$, $i_K = 8$, $n = 1$ and $q = 2$.

Let $L = lcm(P, KQ)$ and $G = gcd(P, KQ)$. As discussed in Section 3.1, global blocks i and $L+i$ are mapped to the same source processor p and redistributed to the same destination processor q, because L is a least common multiplier of P and KQ. The boundaries for Eq. (3) are

$$\begin{aligned} 0 \le p &< P \\ 0 \le q &< Q \\ 0 \le k &< K \end{aligned} \tag{4}$$

From the redistribution equation, we can classify communication patterns into 3 classes for the redistribution problem $\Re_x(P, K, Q)$ (See [4] for an alternative formulation for the *cyclic*(x) to *cyclic*(y) problem) according to the following Lemma.

Lemma 1 *The communication pattern induced by* $\Re_x(P, K, Q)$ *requires: (i) non all-to-all communication if* $G > K$, *(ii) all-to-all communication with a fixed message size if* $K = \alpha G$, *where* α *is an integer greater than 0, and (iii) all-to-all communication with different message sizes if* $G < K$ *and* $K \neq \alpha G$.

Proof: The redistribution equation, Eq. (3), can be rewritten as

$$p - Kq = (nQK - mP) + k$$

which can be expressed as $p - Kq = \lambda G + k$ because $nQK - mP$ is a multiple of G and λ is an arbitrary integer. For any p and q, if there is at least one block satisfying the above equation, this redistribution requires all-to-all communication. When both sides are divided by G, their remainders are

$$(p - Kq) \bmod G = k \bmod G \tag{5}$$

Assume that $G \le K$. There is at least one k satisfying Eq. (5), since k is between $[0, K-1]$. Especially, when $K = \alpha G$ for any integer α, Eq. (5) has α solutions for any (p, q) pair. Therefore, this redistribution requires all-to-all communication with the same message size. In the initial distribution table, α blocks in a column are transfered to the same destination processor. If $K > G$ and $K \neq \alpha G$, then it requires all-to-all communication with different message sizes. Assuming that $G > K$, we can find a processor pair (p, q) exchanging no message. When $(p - Kq) \bmod G \ge K$, none of k satisfies Eq. (5). For example, consider $p = 0$ and $q = Q - 1$ pair which gives

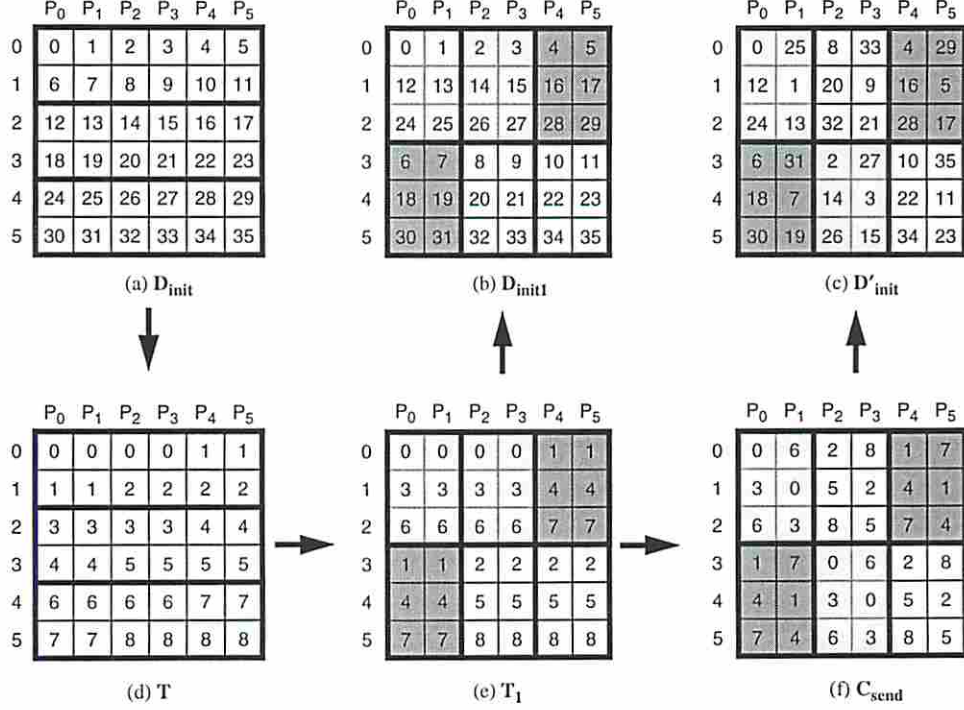
$(p - Kq) \bmod G = K$. The right hand side has a maximum value of $K - 1$. Therefore, processor 0 doesn't send any message to processor $Q - 1$. □

Among these three cases, the case of all-to-all processor communication with the same message size can be optimally scheduled using a trivial round-robin schedule. However, it is non trivial to achieve the same message size between all pairs of nodes in a communication step for all-to-all case with different message sizes. Therefore, we focus on the two cases of redistribution requiring scheduling of non all-to-all communication and all-to-all communication with different message sizes.

4.2 Non all-to-all communication

We first explain how *dpt* $\mathbf{T}$ is transformed to the send communication schedule table $\mathbf{C_{send}}$. Given the redistribution parameters P, Q, and K, we get the $L_s \times P$ initial distribution table $\mathbf{D_{init}}$ and its *dpt* $\mathbf{T}$. Let $G_1 = gcd(P, K)$, $K_1 = \frac{K}{G_1}$ and $P_1 = \frac{P}{G_1}$. In the *dpt*, every K_1 row has a similar pattern. It has different destination processor index. We shuffle the rows such that rows having similar pattern are adjacent resulting in the shuffled *dpt* $\mathbf{T_1}$. The shuffled *dpt* $\mathbf{T_1}$ is divided into Q_1 slices in the row direction, $Q_1 = \frac{L_s}{K_1}$. It is divided into P_1 slices in the column direction. Now, *dpt* $\mathbf{T_1}$ can be considered as a $K_1 \times P_1$ block matrix made of $Q_1 \times G_1$ submatrices. This block matrix is then converted into a generalized circulant matrix by reorganization of submatrices within columns and rotating individual columns within submatrix by appropriate amounts. This generalized circulant matrix can then be used as a communication schedule table $\mathbf{C_{send}}$. In this procedure, the K identical values in row 0 of the *dpt* are distributed to K distinct rows, and hence, row 0 has distinct values. Since $\mathbf{C_{send}}$ is a generalized circulant matrix, all rows are distinct and we achieve a conflict-free schedule.

In the above reorganizations, an element is moved within its column. So, it does not incur any interprocessor communication. Figure 9 shows an example where *dpt* $\mathbf{T}$ of $\Re_x(6, 4, 9)$ is converted to generalized circulant matrix form $\mathbf{C_{send}}$ by column reorganizations. In this example, $L_s = 6$, $G_1 = 2$, $K_1 = 2$, $P_1 = 3$, and $Q_1 = 3$. Figure 9(a) shows the initial distribution table, $\mathbf{D_{init}}$, and (d) shows the corresponding *dpt* $\mathbf{T}$. Rows of $\mathbf{D_{init}}$ and $\mathbf{T}$ are shuffled, as shown in Figure 9(b) and (e). Now we can partition the shuffled tables into submatrices of size 3×2. The diagonalization of submatrices and diagonalization of elements in each submatrix is shown in Figure 9(c) and (f). Figure 9(f) is a generalized circulant matrix, $\mathbf{C_{send}}$. In the following theorem, we will formally show that this procedure correctly obtains a generalized circulant matrix which is the send communication



(a) $\mathbf{D_{init}}$ (b) $\mathbf{D_{init1}}$ (c) $\mathbf{D'_{init}}$ (d) $\mathbf{T}$ (e) $\mathbf{T_1}$ (f) $\mathbf{C_{send}}$

Figure 9: Steps of Column rearrangement

schedule table $\mathbf{C_{send}}$.

Theorem 1 *The initial* dpt $\mathbf{T}$ *of* $\Re_x(P, K, Q)$ *with* $P \leq Q$ *can be reorganized via column reorganizations to a send communication schedule table* $\mathbf{C_{send}}$ *such that (i)* $\mathbf{C_{send}}$ *is a generalized circulant matrix and (ii) Every row of* $\mathbf{C_{send}}$ *has* P *distinct numbers from the set of* $\{0, 1, 2, \cdots, Q-1\}$.

Proof: In the initial distribution table $\mathbf{D_{init}}$, each element, $\mathbf{D_{init}}(a, j)$, is assigned a global block index, $aP + j$, where $0 \leq a < L_s$ and $0 \leq j < P$. The corresponding *dpt* $\mathbf{T}$ of $\mathbf{D_{init}}$ is obtained by replacing each element by $(\mathbf{D_{init}}(a, j)/K) \bmod Q$. First, we show that the *dpt* $\mathbf{T}$ can be converted to a generalized circulant matrix $\mathbf{C_{send}}$. Then every row of $\mathbf{C_{send}}$ consists of P distinct numbers from the set of $\{0, 1, 2, \cdots, Q-1\}$. Therefore, $\mathbf{C_{send}}$ can be used as a send communication schedule table.

The following are the two major steps in our approach to converting the *dpt* $\mathbf{T}$ to a generalized circulant matrix.

1. **Stage 1:** We will prove that the *dpt* $\mathbf{T}$ corresponding to the initial distribution table $\mathbf{D_{init}}$ has some rows with similar patterns. These rows can be brought together in this stage. It results in the intermediate *dpt* $\mathbf{T_1}$. Corresponding operations can be performed on $\mathbf{D_{init}}$

resulting in the intermediate distribution table $\mathbf{D_{init1}}$. $\mathbf{T_1}$ and $\mathbf{D_{init1}}$ can be represented in block matrix form.

2. **Stage 2:** By considering $\mathbf{T_1}$ as a block matrix, we will prove that by reorganization of submatrices within columns and circular shift of elements within columns of submatrices, we will obtain a generalized circulant matrix, which is our send communication schedule table $\mathbf{C_{send}}$.

We now show mathematically that these reorganizations can be performed correctly.

Stage 1 ($\mathbf{D_{init}}$ to $\mathbf{T_1}$): In $\Re_x(P, K, Q)$, the D_i matrix will have L_s rows and P columns. With $G_1 = gcd(P, K)$, we observe that every K_1^{th} row will have the same modulo value with respect to K, where $K_1 = \frac{K}{G_1}$.

$$\mathbf{D_{init}}(a, j) - \mathbf{D_{init}}(a \bmod K_1, j) \equiv (a/K_1)K_1P \equiv 0 \pmod{K} \tag{6}$$

There are exactly Q_1 rows such that $a \bmod K_1 = a_1$, where $0 \leq a_1 < K_1$. The corresponding Q_1 rows in *dpt* $\mathbf{T}$ have the same pattern but their destination processor indices are different. In the first stage, these rows are gathered by moving row a to row $a' = (a \bmod K_1)Q_1 + a/K_1$, where $0 \leq a' < L_s$. In $\mathbf{D_{init1}}$, for a given a' the global index of an element can be determined by first mapping to original row, the corresponding $a = (a' \bmod Q_1)K_1 + a'/Q_1 = a_2'K_1 + a_1'$, where $0 \leq a_1' < K_1$ and $0 \leq a_2' < Q_1$. From this row shuffling, any element of $\mathbf{D_{init1}}(a', j)$ is then given by:

$$\begin{aligned}
\mathbf{D_{init1}}(a', j) &= \mathbf{D_{init}}((a' \bmod Q_1)K_1 + a'/Q_1, j) \\
&= \mathbf{D_{init}}(a_2'K_1 + a_1', j) \\
&= (a_2'K_1 + a_1')P + j
\end{aligned}$$

With $P_1 = \frac{P}{G_1}$, $\mathbf{D_{init1}}$ can be considered as $K_1 \times P_1$ block matrix and each submatrix is a $Q_1 \times G_1$ matrix. Let us denote these submatrices by $\mathbf{A}_{a_1', j_1}$, where $0 \leq a_1' < K_1$ and $0 \leq j_1 < P_1$. $\mathbf{D_{init1}}$ can be represented as,

$$\mathbf{D_{init1}} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \cdots & \mathbf{A}_{0,P_1-1} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{1,P_1-1} \\ \vdots & \vdots & & \vdots \\ \mathbf{A}_{K_1-1,0} & \mathbf{A}_{K_1-1,1} & \cdots & \mathbf{A}_{K_1-1,P_1-1} \end{bmatrix} \tag{7}$$

Let $j_1 = j/G_1$ and $j_2 = j \bmod G_1$ for any j. Recall that the global block index of any element is $\mathbf{D_{init1}}(a', j) = (a'_2 K_1 + a'_1)P + j$, which can be written as $(a'_1 P_1 + j_1)G_1 + (a'_2 K_1 P + j_2)$ by replacing j by $j_1 G_1 + j_2$. Now, let us consider a submatrix $\mathbf{A}_{a'_1, j_1}$. Element $\mathbf{D_{init1}}(a', j)$ is in submatrix $\mathbf{A}_{a'_1, j_1}$, and is located at (a'_2, j_2) within $\mathbf{A}_{a'_1, j_1}$. This matrix can be written as,

$$\mathbf{A}_{a'_1, j_1} = (a'_1 P_1 + j_1)G_1 + \begin{bmatrix} 0 & 1 & \cdots & G_1 - 1 \\ K_1 P & K_1 P + 1 & \cdots & K_1 P + G_1 - 1 \\ \vdots & \vdots & & \vdots \\ (Q_1 - 1)K_1 P & (Q_1 - 1)K_1 P + 1 & \cdots & (Q_1 - 1)K_1 P + G_1 - 1 \end{bmatrix} \quad (8)$$

The corresponding *dpt* $\mathbf{T_1}$ of $\mathbf{D_{init1}}$ is obtained by replacing each element by $(\mathbf{D_{init1}}(a', j)/K) \bmod Q$. This can be represented again in block matrix form as,

$$\mathbf{T_1} = \begin{bmatrix} \mathbf{B}_{0,0} & \mathbf{B}_{0,1} & \cdots & \mathbf{B}_{0,P_1-1} \\ \mathbf{B}_{1,0} & \mathbf{B}_{1,1} & \cdots & \mathbf{B}_{1,P_1-1} \\ \vdots & \vdots & & \vdots \\ \mathbf{B}_{K_1-1,0} & \mathbf{B}_{K_1-1,1} & \cdots & \mathbf{B}_{K_1-1,P_1-1} \end{bmatrix} \quad (9)$$

The expanded form of a submatrix can be written as,

$$\mathbf{B}_{a'_1, j_1} = \left(\frac{(a'_1 P_1 + j_1)}{K_1} + \begin{bmatrix} 0 & 0 & \cdots & 0 \\ P_1 & P_1 & \cdots & P_1 \\ \vdots & \vdots & & \vdots \\ (Q_1 - 1)P_1 & (Q_1 - 1)P_1 & \cdots & (Q_1 - 1)P_1 \end{bmatrix} \right) \bmod Q \quad (10)$$

Thus, we have completed the proof for stage 1. Further, we can see that each block matrix has a base value and a fixed offset matrix.

Stage 2 ($\mathbf{T_1}$ to $\mathbf{C_{send}}$): We can transform $\mathbf{T_1}$ to $\mathbf{C_{send}}$ by reorganizing submatrices within columns of the block matrix and circularly shifting elements within columns of submatrices. First we will show that $Q_1 \times G_1$ block matrices can be reorganized to obtain a block-wise circulant matrix. Next, we show the elements within a submatrix can be converted to circulant matrix.

Now consider the base $b = \frac{(a'_1 P_1 + j_1)}{K_1}$ of $\mathbf{B}_{a'_1, j_1}$ in Eq. (10), where $0 \leq b < P_1$. We refer to each collection of K_1 adjacent submatrices as a run in row-major order. Run r contains submatrices whose base value is r. Through these column reorganizations of the block matrix $\mathbf{T_1}$, the kth submatrix of a run moves to the kth row in its column, where $0 \leq k < K_1$. Thus, K_1 submatrices of each run are aligned into a diagonal. In this column reorganization of the block matrix, submatrix

$\mathbf{B}_{a'_1,j_1}$ in $\mathbf{T_1}$ moves to $\mathbf{C}_{i_1,j_1}$ in $\mathbf{C_{send}}$. The relationship between a'_1 and i_1 is as follows.

$$i_1 = (a'_1 P_1 + j_1) \bmod K_1 \tag{11}$$

The resultant block matrix is a block-wise circulant matrix as shown as follows.

$$\mathbf{C_{send}} = \begin{bmatrix} \mathbf{C}_{0,0} & \mathbf{C}_{0,1} & \cdots & \mathbf{C}_{0,P_1-1} \\ \mathbf{C}_{1,0} & \mathbf{C}_{1,1} & \cdots & \mathbf{C}_{1,P_1-1} \\ \vdots & \vdots & & \vdots \\ \mathbf{C}_{K_1-1,0} & \mathbf{C}_{K_1-1,1} & \cdots & \mathbf{C}_{K_1-1,P_1-1} \end{bmatrix} \tag{12}$$

Similarly, we can convert each submatrix to a circulant matrix. Elements within the j_2^{th} column of submatrix $\mathbf{B}_{a'_1,j_1}$ are circularly shifted by j_2 positions. The relationship between a'_2 and i_2 is as follows.

$$i_2 = (a'_2 + j_2) \bmod Q_1 \tag{13}$$

Each reoganized submatrix is shown below.

$$\mathbf{C}_{i_1,j_1} = \left(\frac{(a'_1 P_1 + j_1)}{K_1} + \begin{bmatrix} 0 & (Q_1-1)P_1 & \cdots & (Q_1-G_1+1)P_1 \\ P_1 & 0 & \cdots & (Q_1-G_1+2)P_1 \\ \vdots & \vdots & & \vdots \\ (Q_1-1)P_1 & (Q_1-2)P_1 & \cdots & (Q_1-G_1)P_1 \end{bmatrix} \right) \bmod Q \tag{14}$$

The resulting matrix is a block-wise circulant matrix and its submatrices are also circulant matrices. Therefore, it is a generalized circulant matrix. This matrix can be used as a send communication schedule table $\mathbf{C_{send}}$. The same column reorganizations are applied to the $\mathbf{D_{init1}}$. The reorganized distribution table, $\mathbf{D'_{init}}$ is in the following form:

$$\mathbf{D'_{init}} = \begin{bmatrix} \mathbf{D}_{0,0} & \mathbf{D}_{0,1} & \cdots & \mathbf{D}_{0,P_1-1} \\ \mathbf{D}_{1,0} & \mathbf{D}_{1,1} & \cdots & \mathbf{D}_{1,P_1-1} \\ \vdots & \vdots & & \vdots \\ \mathbf{D}_{K_1-1,0} & \mathbf{D}_{K_1-1,1} & \cdots & \mathbf{D}_{K_1-1,P_1-1} \end{bmatrix} \tag{15}$$

$$\mathbf{D}_{i_1,j_1} = (a'_1 P_1 + j_1)G_1 + \begin{bmatrix} 0 & (Q_1-1)K_1P+1 & \cdots & (Q_1-G_1+1)K_1P+1 \\ K_1P & 1 & \cdots & (Q_1-G_1+2)K_1P+1 \\ \vdots & \vdots & & \vdots \\ (Q_1-1)K_1P & (Q_1-2)K_1P+1 & \cdots & (Q_1-G_1)K_1P+G_1-1 \end{bmatrix} \tag{16}$$

We will show that every row of $\mathbf{C_{send}}$ has P distinct numbers from the set of $\{0, 1, 2, \cdots, Q-1\}$. Consider the 0th row of $\mathbf{C_{send}}$. For row $i_1 = 0$, it implies $(a'_1 P_1 + j_1) \bmod K_1 = 0$. Then, $(a'_1 P_1 + j_1) \in \{0, K_1, \cdots, (P_1-1)K_1\}$ since $0 \le a'_1 < K_1$ and $0 \le j_1 < P_1$. Therefore, $\frac{(a'_1 P_1 + j_1)}{K_1} \in$

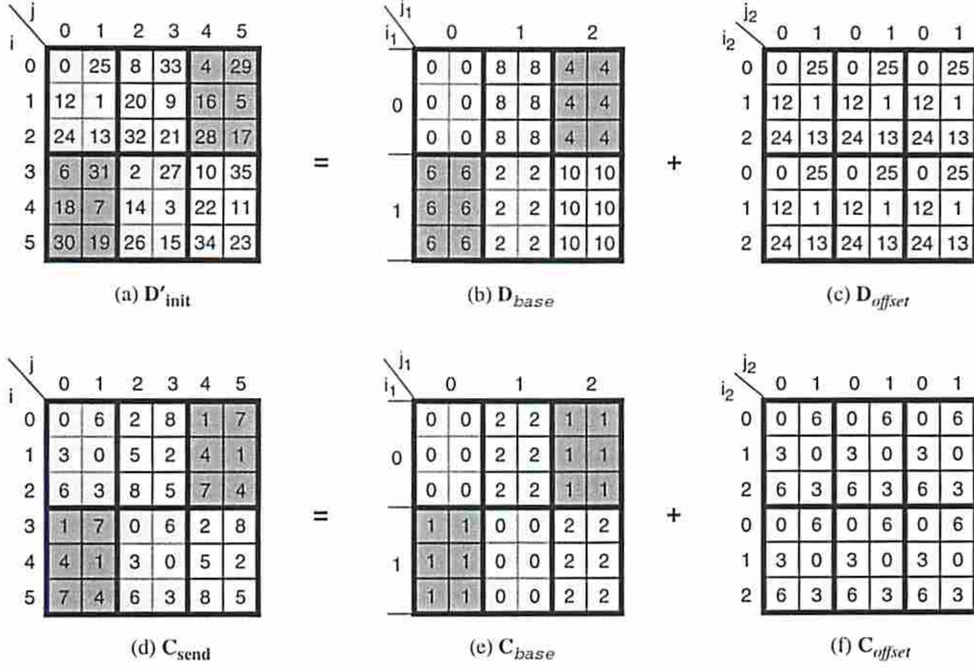


Figure 10: Decomposition of $\mathbf{D}'_{\text{init}}$ and $\mathbf{C}_{\text{send}}$

$\{0, 1, \cdots, (P_1 - 1)\}$. In the first row, the second term of Eq. (14) is $([0\ (Q_1-1)P_1\ \cdots\ (Q_1 - G_1+1)P_1]) \bmod Q$. Since $Q_1 = \frac{Q}{gcd(Q,P_1)}$, $Q_1P_1 \bmod Q = 0$. Therefore, the second term is $[0\ Q - P_1\ \cdots\ Q-(G_1-1)P_1]$. By summing the base and offset values, elements in the first row will have a value in range 0 to $P_1 - 1$ or in range $Q-(G_1-1)P_1$ to $Q-1$. Since $P_1 - 1 < Q-(G_1-1)P_1$, row 0 of the send communication table $\mathbf{C}_{\text{send}}$ consists of P distinct destination processor indices. Since $\mathbf{C}_{\text{send}}$ is a generalized circulant matrix, every row of $\mathbf{C}_{\text{send}}$ consists of P distinct destination processor indices. □

The send communication schedule table $\mathbf{C}_{\text{send}}$ is $K_1 \times P_1$ block matrix. Each block is $Q_1 \times G_1$ submatrix. The submatrix $\mathbf{B}_{i_1,j_1}$ of Eq. (16) consists of two parts. These can be referred as base and offset respectively. The row and column of the base are indexed as i_1 and j_1 respectively. Also, the row and column of the offset are indexed as i_2 and j_2 respectively. Thus, entries of the base matrix is independent of i_2 and j_2. All entries within $Q_1 \times G_1$ submatrix of $\mathbf{C}_{base}$ have the same value, given by $(\frac{(a'_1P_1+j_1)}{K_1}) \bmod Q = \frac{(a'_1P_1+j_1)}{K_1}$. Similarly, the entries of the offset are independent of i_1 and j_1. Therefore, all $Q_1 \times G_1$ submatrices of $\mathbf{C}_{offset}$ are identical to one another. Each is a $Q_1 \times G_1$ circulant matrix. Figure 10 shows the base and the offset of $\mathbf{D}'_{\text{init}}$ and $\mathbf{C}_{\text{send}}$ for $\Re_x(6,4,9)$.

With reference to Figure 6, observe that $\mathbf{C}_{\text{send}}$ helps to specify the row reorganizations that convert $\mathbf{D}'_{\text{init}}$ to $\mathbf{D}'_{\text{final}}$. The initial column reorganizations convert $\mathbf{D}_{\text{init}}$ to $\mathbf{D}'_{\text{init}}$ and $\mathbf{T}$ to $\mathbf{C}_{\text{send}}$. Although Section 3 indicates that the column reorganizations reorganize data within a processor,

this operation can be expensive for large array sizes. Instead, the reorganization can be done by maintaining pointers to the elements of the array. Each source processor has a table which points to the data blocks to be packed in a communication step. It is denoted as *send data location table* $\mathbf{D_{send}}$. Each entry of $\mathbf{D_{send}}$ is the local block index of the corresponding entry of $\mathbf{D'_{init}}$. Therefore, $\mathbf{D_{send}}(i,j) = \mathbf{D'_{init}}(i,j)/P$. Each entry of $\mathbf{C_{send}}$, $\mathbf{C_{send}}(i,j)$, points to the destination processor of the corresponding entry of $\mathbf{D_{send}}$, $\mathbf{D_{send}}(i,j)$. Our scheme computes the schedule and data index set at the same time.

The *receive communication schedule table* $\mathbf{C_{recv}}$ and the *receive data location table* $\mathbf{D_{recv}}$ can be computed in a similar way. They are not discussed further.

Theorem 2 gives the formulae to compute the individual entries of $\mathbf{C_{send}}$ and $\mathbf{D_{send}}$ efficiently. Referring to Figure 10, in the communication schedule table, $\mathbf{C_{send}}$, i indices represent the communication steps and j indices represent the source processor indices. Each entry $\mathbf{C_{send}}(i,j)$ refers to the destination processor to which the jth processor communicates in the ith communication step. In the following theorem, i_1, i_2, j_1, and j_2 refer to the indices defined in the proof of Theorem 1. (See Eq. (11) and Eq. (13))

Theorem 2 *Given redistribution parameters,* K, P, *and* Q, *let* $G_1 = gcd(P,K)$, $P_1 = P/G_1$, $K_1 = K/G_1$, $G_2 = gcd(P_1,Q)$, *and* $Q_1 = Q/G_2$. *A send communication schedule table* $\mathbf{C_{send}}$ *and the send data location table* $\mathbf{D_{send}}$ *in the generalized circulant matrix form can be computed as follows:*

$$\mathbf{C_{send}}(i,j) = \{\{n(j_1 - i_1)\} \bmod P_1 + \{(i_2 - j_2) \bmod Q_1\}P_1\} \bmod Q \quad (17)$$

$$\mathbf{D_{send}}(i,j) = \{m(j_1 - i_1)\} \bmod K_1 + \{(i_2 - j_2) \bmod Q_1\}K_1 \quad (18)$$

where n *and* m *are solutions of* $nK_1 - mP_1 = 1$.

Proof: From the proof of Theorem 1, the $(i,j)^{th}$ entry of $\mathbf{D'_{init}}$ is shown as follows.

$$\mathbf{D'_{init}}(i,j) = \mathbf{D_{init1}}(a',j) = (a'_1 P_1 + j_1)G_1 + (a'_2 K_1 P + j_2) \quad (19)$$

From Eq. (11) and Eq. (13),

$$i_1 = (a'_1 P_1 + j_1) \bmod K_1 \quad (20)$$

$$i_2 = (a'_2 + j_2) \bmod Q_1 \quad (21)$$

Let $t = (a'_1 P_1 + j_1)$. From Eq. (20),

$$t = XK_1 + i_1 = YP_1 + j_1$$

where $0 \leq X < P_1$ and $0 \leq Y < P_1$. Hence,

$$XK_1 - YP_1 = (j_1 - i_1) \tag{22}$$

We have to solve a Diophantine equation to find X and Y. Since $gcd(K_1, P_1) = 1$, we can find m and n using the Euclid algorithm such that

$$nK_1 - mP_1 = 1 \tag{23}$$

From Eq. (22) and Eq. (23),

$$X = \{n(j_1 - i_1)\} \bmod P_1 \tag{24}$$

$$Y = \{m(j_1 - i_1)\} \bmod K_1 \tag{25}$$

Eq. (21) becomes $a'_2 = (i_2 - j_2) \bmod Q_1$. By replacing a'_1 and a'_2 with i_1 and i_2, Eq. (19) can be rewritten as follows

$$\mathbf{D}'_{\mathbf{init}}(i,j) = (XK_1 + i_1)G_1 + \{(i_2 - j_2) \bmod Q_1\}K_1 P + j_2$$

Therefore, $\mathbf{C}_{\mathbf{send}}(i,j) = (\mathbf{D}'_{\mathbf{init}}(i,j)/K) \bmod Q$ gives Eq. (17) since $i_1 G_1 + j_1 < K$. Similarly, we can prove Eq. (18). □

The above formulae for computing the communication schedule and index set for redistribution are extremely efficient compared with the methods presented in [4], which use a bipartite matching algorithm. Furthermore, using our formulae, each processor computes only entries which it needs in its send communication schedule table. Hence, the schedule and index set computation can be performed in a distributed way and the total cost of computing the schedule and index set is $O(max(P,Q))$. Our scheme minimizes the number of communication steps and avoids node contention. In each communication step, equal-sized messages are transferred. Therefore, our scheme minimizes the data transfer cost.

Corollary 1 *In the proposed redistribution algorithm, the costs of index set and communication schedule computations are* $O(max(P,Q))$. *The amortized cost to compute a step in the communication schedule and index set computation is* $O(1)$.

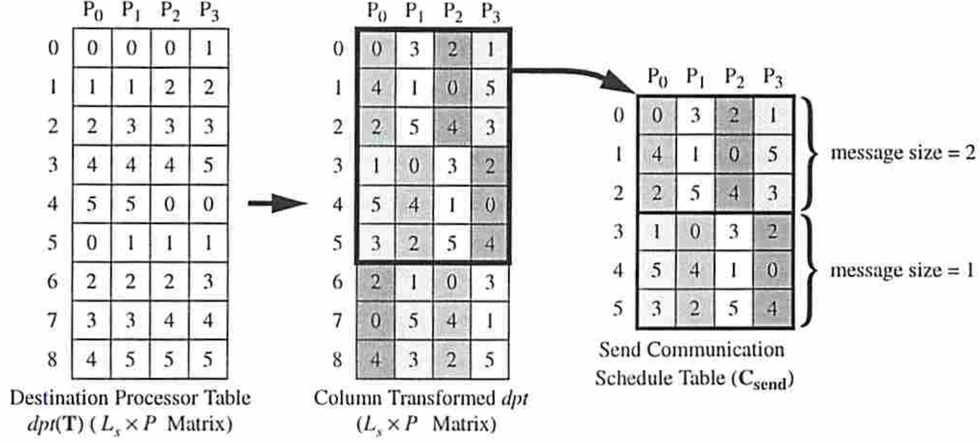


Figure 11: Example illustrating an all-to-all case with different message sizes: $\Re_x(4,3,6)$

4.3 All-to-all communication

The all-to-all communication case arises if $G(= G_1G_2) \leq K$ as stated in Lemma 1, where $G_1 = gcd(P,K)$ and $G_2 = gcd(P_1,Q)$. From the first superblock, the *dpt* $\mathbf{T}$ is constructed. The *dpt* $\mathbf{T}$ is a $L_s \times P$ matrix, where $L_s = K_1Q_1$. Since $Q = Q_1G_2$ and $G_2 \leq K_1$, the number of rows in *dpt* $\mathbf{T}$ $L_s \geq Q$. Therefore, each column has more entries than Q destination processors. In each column, several blocks are transferred to the same destination. The column reorganizations as stated in Section 4.2 are applied to the *dpt* $\mathbf{T}$, which results in a generalized circulant matrix and is a $K_1 \times P_1$ circulant block matrix. Each block is a $Q_1 \times G_1$ submatrix which is also a circulant matrix. In the block matrix, the first G_2 blocks in each column are distinct. Blocks in every G_2^{th} row have the same entries but different circular-shifted patterns. These blocks can be folded onto the blocks in their first row. Therefore, the first G_2 rows in the block matrix only are used in determining a send communication schedule table $\mathbf{C_{send}}$. It is a $Q \times P$ generalized circulant matrix. Since blocks in every G_2^{th} row are folded onto blocks in their first row, for all-to-all communication case with different message sizes, blocks in the first $(K_1 \bmod G_2)$ rows of $\mathbf{C_{send}}$ have size $\lceil \frac{K_1}{G_2} \rceil$, whole blocks in the remaining rows have size $\lfloor \frac{K_1}{G_2} \rfloor$.

Figure 11 shows an example of the send communication schedule table of $\Re_x(4,3,6)$, generated for all-to-all case with different message sizes. In this example, each processor has more entries than 6 destination processors. The corresponding *dpt* is a $L_s \times P$ matrix, where $L_s = 9$ and $P = 4$. Applying column reorganizations on it results in a generalized circulant matrix, which can be considered as a $K_1 \times P_1$ block matrix, where $K_1 = 3$ and $P_1 = 4$. Each block is a $Q_1 \times G_1$ matrix, where $Q_1 = 3$ and $G_1 = 1$. The first $G_2 = 2$ rows are used as a send communication

schedule table $\mathbf{C_{send}}$. The 3rd row is folded onto the 1st row. Hence, the message size in the 1st row is 2 and that in the 2nd row is 1. If K_1 is a multiple of G_2, the message size in every step is the same. Therefore, the network bandwidth is fully utilized by sending equal sized messages in each communication step.

Theorem 3 summarizes the above intuition and shows that the send communication schedule table for all-to-all communication can be obtained as a generalized circulant matrix. Further, it ensures that equal-sized messages are transferred in every communication step.

Theorem 3 *If a redistribution problem $\Re_x(P, K, Q)$ requires all-to-all communication, the send communication schedule table is a $Q \times P$ matrix which consists of the first $\{0, 1, \cdots, Q-1\}^{th}$ rows in a $L_s \times P$ generalized circulant matrix as computed by the equations of Theorem 2. Equal-sized messages are transferred in each communication step.*

Proof: From Theorem 1, the *dpt* $\mathbf{T}$ is rearranged into a generalized circulant matrix form by column reorganizations. It is computed by Eq. (17) in Theorem 2. The generalized circulant matrix is a $K_1 \times P_1$ circulant block matrix. Each block matrix is a $Q_1 \times G_1$ circulant matrix. We show that every G_2^{th} row in the circulant block matrix consist of the same indices in each column. Therefore, we prove that the send communication schedule table is represented as $Q \times P$ generalized circulant matrix and equal-sized messages are transferred in each communication step.

Consider a generalized circulant matrix computed by Eq. (17) in Theorem 2. The second part of Eq. (17) is related to an offset of the $Q_1 \times G_1$ block matrix. For processor j, the second part is $qP_1 \bmod Q$, where $q = (i_2 - j_2) \bmod Q_1$ and $0 \le q, i_2 < Q_1$. Since $qP_1 \bmod Q = qP_1 - \{(qP_1/Q) \times Q\} = G_2\{qP_2 - Q_1(qP_1/Q)\}$, $qP_1 \bmod Q$ is a multiple of G_2. Therefore, the second part of Eq. (17) is an element in *set* $\{0, G_2, 2G_2, \cdots, (Q_1 - 1)G_2\}$. Similarly, the first part of Eq. (17) is related to the base value of each $Q_1 \times G_1$ block matrix. A base value is determined by $\{\{n(j_1 - i_1)\} \bmod P_1\} \bmod Q = \{n(j_1 - i_1)\} \bmod P_1$, for processor j and $0 \le i_1 < K_1$. For processor j, consider the base value of the block matrix at the first row and the block matrix at the G_2^{th} row. For the block matrix at the first row, the base value in Eq. (17) is computed as follows,

$$\begin{aligned}(nj_1) \bmod P_1 &= nj_1 - \{(nj_1)/P_1\}P_1 \\ &= nj_1 - \lambda_1 G_2\end{aligned}$$

where $\lambda_1 = [(nj_1)/P_1]P_2$ and $P_1 = P_2G_2$. For the block matrix in the G_2^{th} row, the base value in

Eq. (17) is also computed as follows,

$$\begin{aligned} n(j_1 - G_2) \bmod P_1 &= n(j_1 - G_2) - \{n(j_1 - G_2)/P_1\}P_1 \\ &= (nj_1) - \{n - [n(j_1 - G_2)/P_1]P_2\}G_2 \\ &= (nj_1) - \lambda_2 G_2 \end{aligned}$$

where $\lambda_2 = n - [n(j_1 - G_1)/P_1]P_2$. Even though their base values are different, the difference of both base values are a multiple of G_2. When the base value is divided by G_2, their remainder is the same. Therefore, the $Q_1 \times G_1$ block matrices with these base values consist of the same destination processor indices, because offset values are multiples of G_2. So, block matrices at every G_2^{th} row in each column consist of the same entries in $\mathbf{C}_{\mathbf{send}}$. In the case of all-to-all communication with same message size, K_1 is a multiple of G_2. Therefore, for all-to-all communication case, the send communication schedule table is a $Q \times P$ matrix which consists of the first $\{0, 1, \cdots, Q-1\}^{th}$ rows in a $L_s \times P$ generalized circulant matrix. Every G_2^{th} row in the circulant block matrix is folded into its first row. Therefore, equal-sized messages are transferred in a communication step. □

Corollary 2 *For the redistribution problem $\Re_x(P, K, Q)$, the proposed algorithm minimizes the data transfer cost in both non all-to-all communication case and all-to-all communication case.*

4.4 Data transfer cost

We now give the pseudo-code for our algorithm used in the redistribution problem in Figure 12. Theorem 4 analyzes the complexity of this communication algorithm.

In distributed memory model, the data transfer cost has two parameters which are start-up time and transmission time. The start-up time, τ_s, is incurred once for each communication event and is independent of the communicated message size. Generally, the start-up time consists of the transfer request and acknowledgment latencies, context switch latency, and latencies for initializing the message header. The unit transmission time, τ_d, is the cost of transferring a message of unit length over the network. The transmission time for a message is proportional to its size. Thus, the data transfer time for sending a message of size m units from one processor to another is modeled as $\tau_s + m\tau_d$. In this model, a reorganization of the data elements among the processors, in which each processor has m units of data for another processor, also takes $\tau_s + m\tau_d$ time. This model assumes that there is no node contention. This is ensured by our communication schedules for

Algorithm of redistribution from *cyclic(x)* on *P* processors to *cyclic(Kx)* on *Q* processors
(this algorithm is described with respect to each processor i.e. each processor executes it)

```
// Assume that each processor knows P, K, Q and its ID # i.e. processor #, pid
// comm_pattern = 0 if non all-to-all communication; = 1 if all-to-all communication with same message
//                  = 2 if all-to-all communication with different message

comm_pattern = 0
message_size = 1
G1 = gcd(P,K); K1 = K / G1; P1 = P / G1
Ls = lcm(P, KQ) / P
G2 = gcd(P1, Q); Q1 = Q / G2
j1 = pid / G1; j2 = pid mod G1

// Determination of Communication Pattern

if (G > K) // non all-to-all case
    maxStep = Ls - 1
else if (G < K)  // all-to-all case
    maxStep = Q - 1
    if (K = wG where w is an integer > 0)
        comm_pattern = 1
    else comm_pattern = 2
    endif
endif

// if the processor is in sending mode

if (Send Processor Set)

    // Index Computation and Schedule Determination

    for step = 0 to maxStep
        i1 = step / Q1
        i2 = step mod Q1
        Dsend[step] = {m (j1 - i1)} mod K1 + {(i2 - j2) mod Q1} K1
        Csend[step] = {{n(j1- i1)} mod P1 + {(i2-j2) mod Q1} P1} mod Q
           where m and n are solutions of nK1 - mP1 = 1
    endfor

    // Message Packing and Data Transmission

    for step = 0 to maxStep
        dst = Csend[step]
        if (comm_pattern = 1)
            message_size = w
```

(continued on next page)

Figure 12: High-level description of the Algorithm

Algorithm of redistribution from *cyclic(x)* on *P* processors to *cyclic(Kx)* on *Q* processors (continued)

```
        else if (comm_pattern = 2)
            limit = (K1 mod G2) Q1
            if (step < limit)
                message_size = ceiling(K1 / G2)
            else
                message_size = floor(K1 / G2)
            endif
        endif
        Pack data at Dsend[step] with messages of message_size into send-buffer
              and send them to dst
    endfor

// if the processor is in receiving  mode

else if (Receive Processor Set)

    // Index Computation and Schedule Determination
    // The equations of Drecv and Crecv are derived from the column reorganizaitons similar to those used
       for Dsend and Csend

    for step = 0 to maxStep
       compute  Drecv[step] and Crecv[step].
    endfor

    // Message Unpacking and Data Receiving

    for step = 0 to maxStep
        src = Crecv[step]
        if (0  ≤ src < P)
            Receive messages of  messsge_size from src and unpack them to Drecv[step]
        else
            idle
        endif
    endfor
endif
```

Figure 13: High-level description of the Algorithm (continued)

redistribution. Theorem 4 shows the data transfer cost for our redistribution algorithms using the distributed memory model.

Theorem 4 *Consider an array with N elements initially distributed* cyclic(x) *on P processors. That needs to be redistributed to* cyclic(Kx) *on Q processors. Using our algorithms, the data transfer costs for performing $\Re_x(P, K, Q)$ are* (i) $L_s T_s + \frac{N}{P}\tau_d$ *in the case of non all-to-all communication pattern, and* (ii) $QT_s + \frac{N}{P}\tau_d$ *in the case of all-to-all communication pattern.*

Proof: Data transfer cost is considered as a sum of total start-up cost and total transmission cost. Consider our send communication schedule table $\mathbf{C_{send}}$, which is in generalized circulant matrix form and each row consists of distinct elements. The communication schedule is specified as a sequence of row reorganization on $\mathbf{C_{send}}$. A row reorganization moves elements to their destination processor specified by $\mathbf{C_{send}}$. This corresponds to an interprocessor communication event. The number of communication steps is equal to the number of row reorganizations on $\mathbf{C_{send}}$. We prove the theorem in terms of the number of reorganizations required to convert $\mathbf{C_{send}}$ to its desired final form. The total start-up cost is proportional to the number of communication steps. Therefore, it is in proportion to the number of row reorganizations. Also, communication pairs in each communication step communicate messages of the same size. Therefore, source processors have the same total transmission cost without wasting the network bandwidth. The total transmission cost is proportional to the size of an array assigned to each processor.

(i) non all-to-all communication: We know that each row of $\mathbf{C_{send}}$ represents communication step i. The total number of communication steps is L_s. In each step, processor pairs communicate one block per superblock between them. Therefore, data transfer cost can be estimated as follows,

$$\begin{aligned} data\ transfer\ cost &= L_s \cdot (T_s + \frac{N}{PL_s}\tau_d) \\ &= L_s T_s + \frac{N}{P}\tau_d \end{aligned}$$

(ii) all-to-all communication: From Theorem 3, we need only Q rows in the generalized circulant matrix form to develop the communication schedule table $\mathbf{C_{send}}$. We know that each row of $\mathbf{C_{send}}$ represents communication step i. The total number of communication steps is L_s. In each step, equal-sized messages are communicated between processor pairs. The message size in the first r rows on $\mathbf{C_{send}}$ is $q + 1$ blocks per superblock, where $r = L_s \bmod Q$ and $q = L_s/Q$. The message

Table 5: Comparison of data transfer cost and schedule and index computation costs of the Caterpillar algorithm, bipartite matching scheme and our algorithm.

	Non all-to-all communication		All-to-all communication with different message sizes	
	Data transfer cost	Schedule and index computation cost	Data transfer cost	Schedule and index computation cost
Caterpillar algorithm [19]	$Q\left(T_s + \tau_d \frac{M}{L_s}\right)$	$O(Q)$	$QT_s + \tau_d \sum_{i=0}^{Q-1} m_i$	$O(Q)$
Bipartite matching scheme [4]	$L_s\left(T_s + \tau_d \frac{M}{L_s}\right)$	$O((P+Q)^4)$	$QT_s + \tau_d M$	$O((P+Q)^4)$
Our algorithm	$L_s\left(T_s + \tau_d \frac{M}{L_s}\right)$	$O(Q)$	$QT_s + \tau_d M$	$O(Q)$

Note: $L_s < Q$ for the non all-to-all communication case, $M = N/P$ and m_i is the maximum transfered data size in communication step i.

size in remaining rows is q blocks per superblock. Therefore, data transfer cost can be estimated as follows,

$$\begin{aligned} data\ transfer\ cost &= QT_s + \{r(q+1) + (Q-r)q\}\frac{N}{PL_s}\tau_d \\ &= QT_s + L_s\frac{N}{PL_s}\tau_d \\ &= QT_s + \frac{N}{P}\tau_d \end{aligned}$$

□

5 Experimental Results

Our experiments were conducted on the IBM SP2 at Maui High Performance Computing Center. The algorithms were written in C and MPI. MPI communication primitives were used for interprocessor communication.

Table 5 shows a comparison of the proposed algorithm with the Caterpillar algorithm[19] and the bipartite matching scheme[4] with respect to the data transfer cost and schedule and index

computation costs. For the all-to-all communication case with equal-sized messages, the data transfer cost is the same in each communication step for all three algorithms. Also, the schedule computation can be performed in a simple way. Hence, the all-to-all communication case with equal-sized messages is not considered in Table 5. In Table 5, M is the size of the array assigned to each source processor ($M = \frac{N}{P}$). For the non all-to-all communication case, $L_s < Q$, where $L_s = \frac{lcm(P,KQ)}{P}$. Our algorithm as well as the bipartite matching scheme perform less number of communication steps compared with the Caterpillar algorithm. For the all-to-all communication case with different message sizes, the messages transmitted in a communication step are of the same size in the bipartite matching scheme as well as our algorithm. Therefore, the network bandwidth is fully utilized and the total transmission cost is $\tau_d M$. In the Caterpillar algorithm, the transmission cost in a communication step is dominated by the largest message transferred in that step. Let m_i denote the size of the largest message sent in a communication step i. Note that $\sum_{i=0}^{Q-1} m_i \geq M$. The total start-up cost of all the algorithms is QT_s since the number of communication steps is the same. On the other hand, the total transmission cost of the bipartite matching scheme and our algorithm is $\tau_d M$ which is less than that of the Caterpillar algorithm. The Caterpillar algorithm as well as our algorithm perform the schedule and index computation in $O(Q)$ time. However, the schedule and index computation cost in the bipartite matching scheme is $O((P+Q)^4)$.

To evaluate the total redistribution cost and the data transfer cost, we consider 3 different scenarios corresponding to the relative size of P and Q: (Scenario 1); $P \ll Q$, (Scenario 2); $Q \geq 2P$, and (Scenario 3); $P < Q < 2P$. In our experiments, we choose $P = 18$ and $Q = 78$ for Scenario 1, $P = 30$ and $Q = 66$ for Scenario 2, and $P = 46$ and $Q = 50$ for Scenario 3. The array consisted of single precision integers. The size of each array element is 4 bytes. The array size was chosen to be a multiple of the size of a superblock to avoid padding using dummy data.

The rest of this section is organized as follows. Subsection 5.1 reports experimental results of the overall redistribution time of our algorithm and the Caterpillar algorithm. Subsection 5.2 shows experimental results for the data transfer time of our algorithm and the Caterpillar algorithm. Subsection 5.3 compares our algorithm and the bipartite matching scheme with respect to the schedule computation time.

5.1 Total redistribution time

In this subsection, we report experimental results for the total redistribution time of our algorithm and the Caterpillar algorithm. The total redistribution time consists of the schedule com-

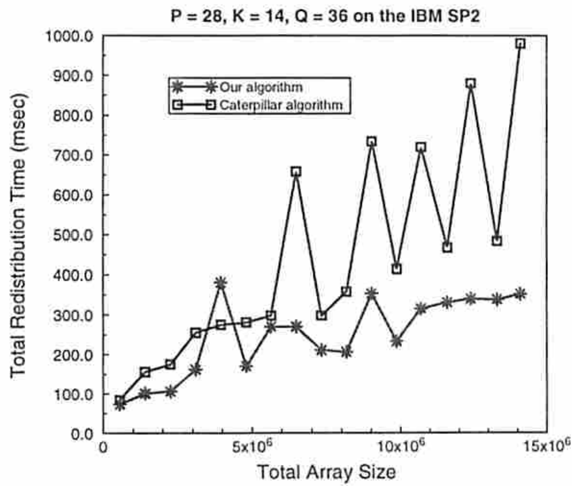
```
for (j=0; j<n1; j++) {
   ts = MPI_Wtime()
   /* redistribution routine */
   compute schedule and index set
   for (i=0; i<n2; i++) {
      if (source processor) { /* source processor */
         pack message
         send message to a destination processor
      } else {                /* destination processor */
         receive message from a source processor
         unpack message
      }
   }
   node_time[j] = MPI_Wtime() - ts
   compute tavg from node_time of each node
   T[j] = tavg
}

compute Tmax = max{T[j]}, Tmin = min{T[j]},
               Tmed = median{T[j]}, Tavg = average{T[j]}
```

Figure 14: Steps for measuring the redistribution time.

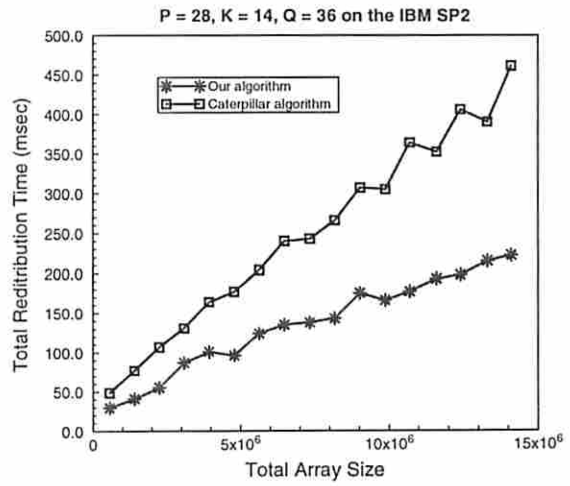
putation time, index computation time, packing/unpacking time, and data transfer time. In our experiments, the source and the destination processor sets were disjoint. In each communication step, each sender packs a message before sending it and each receiver unpacks the message after receiving it. Pack operations in the source processors and unpack operations in the destination processors were overlapped, *i.e.*, after sending their message in communication step i, senders start to pack a message for communication in step $(i + 1)$ and receivers start to unpack the message received in step i.

Our methodology for measuring the total redistribution time is shown in Figure 14. The time was measured using the `MPI_Wtime()` call. `n1` is the number of runs. A run is an execution of redistribution. `n2` is the number of communication steps. Each processor measures `node_time[j]` in the jth run. Generally, source and destination processors which do not perform an interprocessor communication in the last step, complete the redistribution earlier than the processors which receive a message and unpack it. A barrier synchronization, `MPI_Barrier()`, was performed at the end of redistribution. After measuring `node_time`, the average `node_time` over $(P + Q)$ processors is computed and saved as `tavg`. The measured value is stored in an array `T`, as shown in Figure 14. After the redistribution is performed `n1` times, the maximum, minimum, median, and average total redistribution time are computed over `n1` runs. In our experiments, `n1` was set to 20.

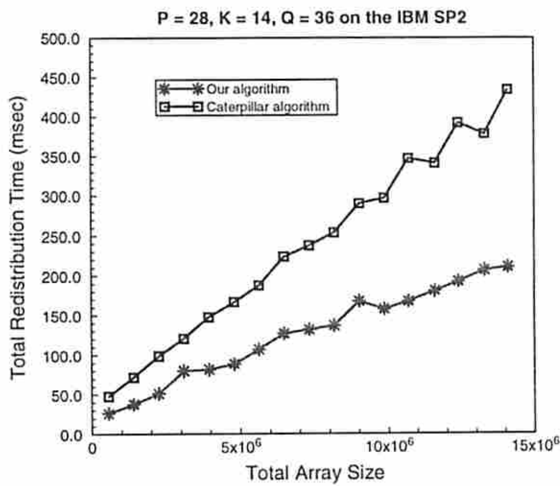
In Figure 15, the total redistribution time of our algorithm and the Caterpillar algorithm are



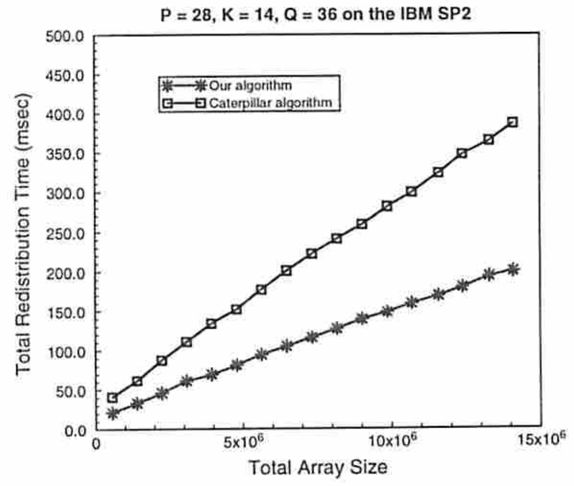
(a) Maximum Time



(b) Average Time



(c) Median Time



(d) Minimum Time

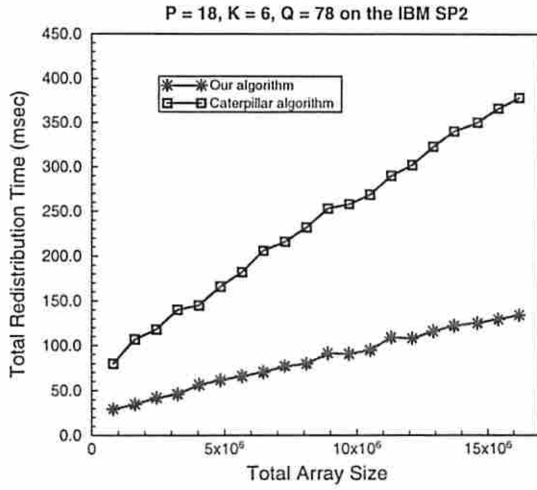
Figure 15: The maximum, average, median, and minimum total redistribution time for $\Re_2(28, 14, 36)$.

compared on the IBM SP2. In these experiments, 64 nodes were used; 28 were source processors and 36 were destination processors. The total number of array elements (in single precision) was varied from 564,480 (2.26 Mbytes) to 14,112,000 (56.4 Mbytes). K was set to 14. Figure 15(a) shows the maximum time (`Tmax` in Figure 14). Over `n1` runs, a large variance in the measured values was observed. Figure 15(b) shows the results of the average time (`Tavg` in Figure 14). Figure 15(c) shows the results using the median time (`Tmed` in Figure 14). There is still a variance in the measured values. However, this is smaller than the variance found in the average and the maximum time. Figure 15(d) shows the minimum time for redistribution (`Tmin` in Figure 14). This plot is a more accurate observation of the redistribution time since the minimum time has the smallest component due to OS interference and other effects related to the environment. In the remaining plots in this section, we show `Tmin` only.

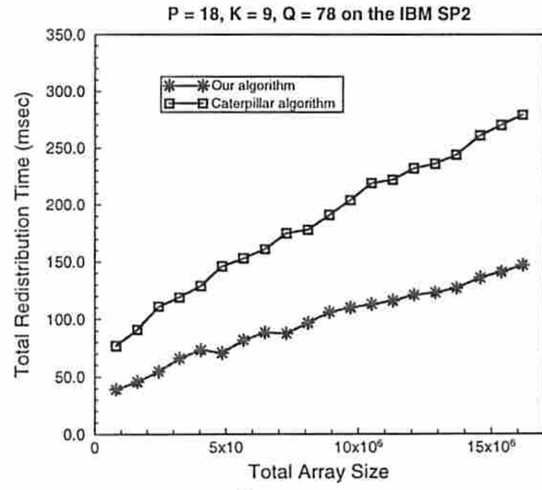
The redistribution $\Re_2(28, 14, 36)$ is a non all-to-all communication case. In the non all-to-all communication case, the messages in each communication step are of the same size. The total number of communication steps is 18 in our algorithm, while it is 36 in the Caterpillar algorithm. Therefore, the redistribution time of our algorithm is theoretically 50% of that of the Caterpillar algorithm. In the experimental results shown in Figure 15(d), the redistribution time of our algorithm is between 51.8% and 55.1% of that of the Caterpillar algorithm.

Figure 16 shows several experimental results for the non all-to-all communication case. Figure 16(a), (b), and (c) show results for $P = 18$ and $Q = 78$. $K = 6$, 9, and 12 were used. The number of communication steps using our algorithm is 26, 39, and 52, respectively. The number of communication steps using the Caterpillar algorithm is 78. Therefore, the redistribution time of our algorithm can be expected to be reduced by 67%, 50%, and 33% when compared with that of the Caterpillar algorithm. Our experimental results confirm these. Similar reductions in time were achieved in the other experimental results shown in Figure 16.

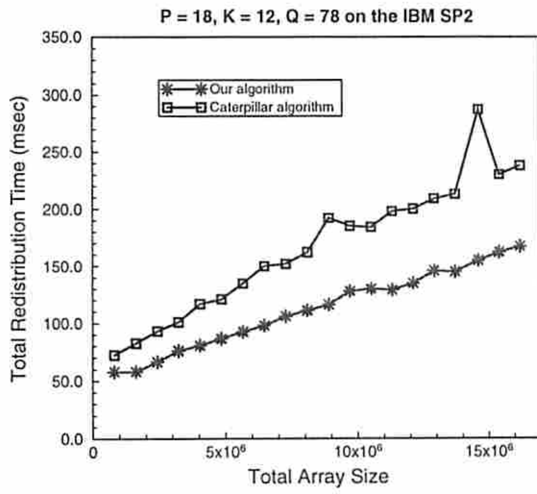
Figure 17 compares the overall redistribution time for the all-to-all communication case with different message sizes. Figure 17(a) reports the experimental results for $\Re_4(28, 6, 36)$. The array size was varied from 677,376 (2.71 Mbytes) to 16,934,400 (67,7 Mbytes). For this case, both the algorithms have the same number of steps (36). Within a superblock, half the messages are two blocks while the other half are one block. In our algorithm, equal-sized messages are transferred in each communication step. Therefore, during half the steps, two block messages are sent while during the other half one block messages are sent. The Caterpillar algorithm does not attempt to



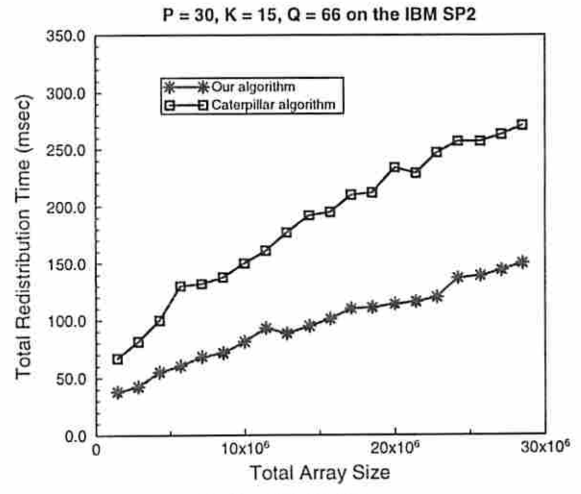
(a) $\mathfrak{R}_{16}(18,6,78)$



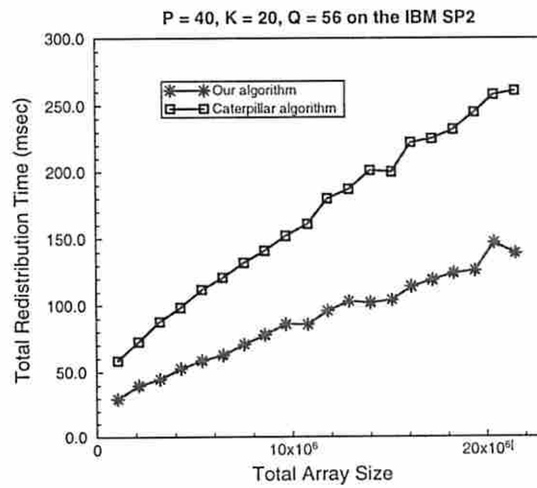
(b) $\mathfrak{R}_{16}(18,9,78)$



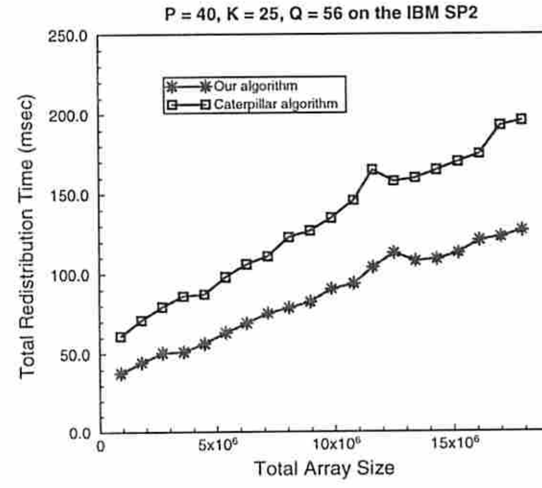
(c) $\mathfrak{R}_{16}(18,12,78)$



(d) $\mathfrak{R}_{16}(30,15,66)$

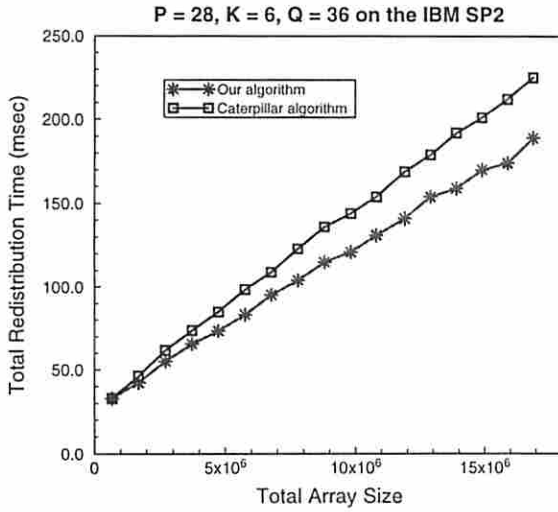


(e) $\mathfrak{R}_{16}(40,20,56)$

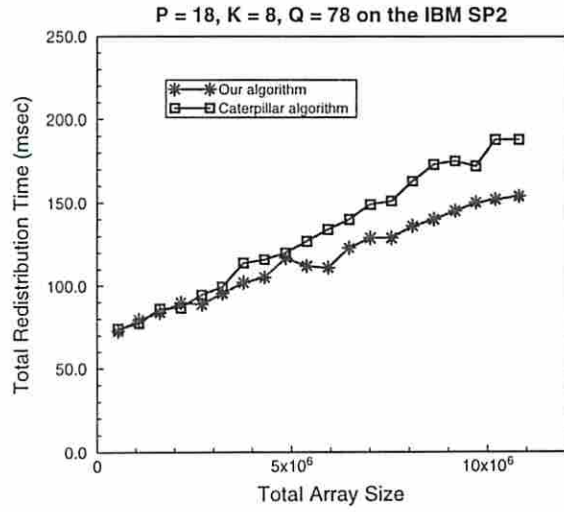


(f) $\mathfrak{R}_{16}(40,25,56)$

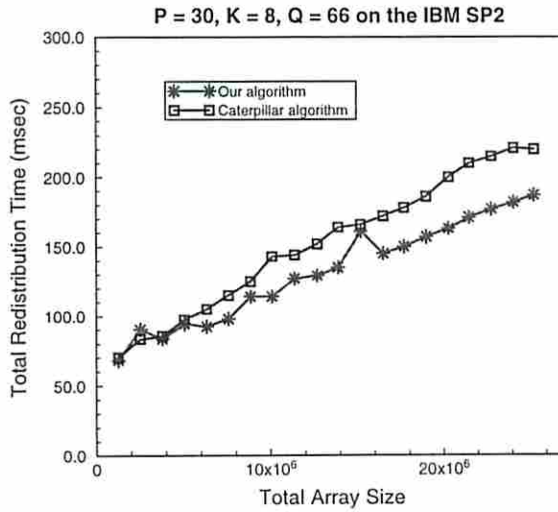
Figure 16: Total redistribution time for non all-to-all communication cases.



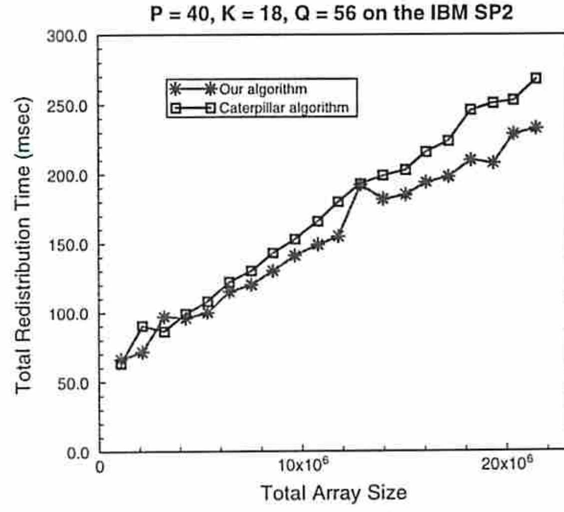
(a) $\Re_4(28,6,36)$



(b) $\Re_{16}(18,8,78)$



(c) $\Re_{16}(30,8,66)$



(d) $\Re_8(40,18,56)$

Figure 17: Total redistribution time for all-to-all communication cases with different message sizes.

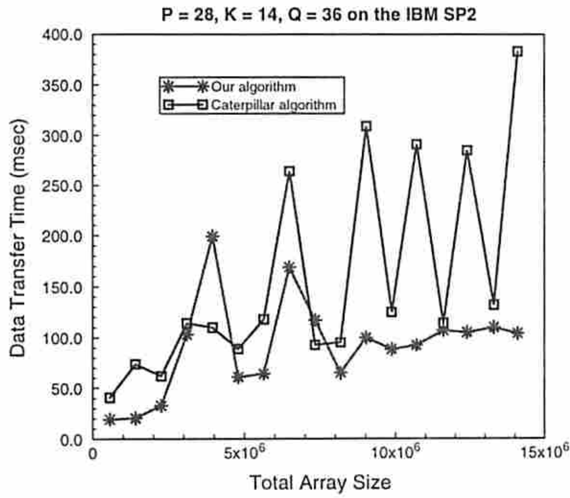
```
for (j=0; j<n1; j++) {
   /* redistribution routine */
   compute schedule and index set
   node_tr[j] = 0;
   for (i=0; i<n2; i++) {
      if (source processor) { /* source processor */
         pack message
         ts = MPI_Wtime()
         send message to a destination processor
         node_tr[j] = tr[j] + MPI_Wtime() - ts
      } else {              /* destination processor */
         ts = MPI_Wtime()
         receive message from a source processor
         node_tr[j] = tr[j] + MPI_Wtime() - ts
         unpack message
      }
   }
   compute tavg from node_tr of each node
   Tr[j] = tavg
}

compute Tmax = max{Tr[j]}, Tmin = min{Tr[j]},
              Tmed = median{Tr[j]}, Tavg = average{Tr[j]}
```

Figure 18: Steps in measuring the data transfer time.

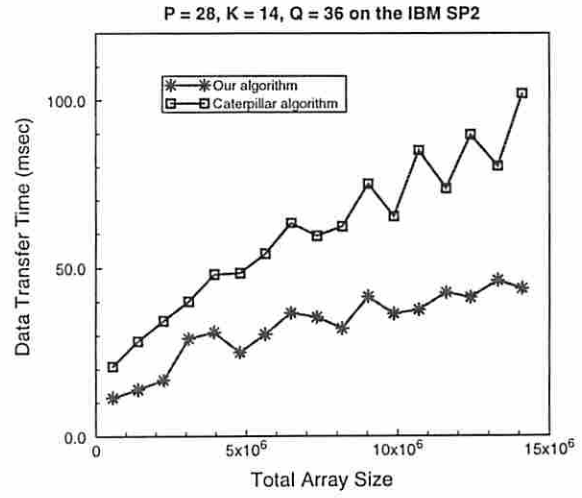
schedule the communication operations to send equal-sized messages. Therefore, the redistribution time in each step is determined by the time to transfer the largest message. Theoretically, the total redistribution time of our algorithm is reduced by 25% compared with that of the Caterpillar algorithm. In our experiments, we achieved up to 17.9% reduction in redistribution time. When the array size is small, both algorithms have approximately the same performance since the start-up cost dominates the overall data transfer cost. As the array size is increased, the reduction in the time to perform the distribution using our algorithm improves. For other scenarios, we obtained similar results (See Figure 17(b), (c), and (d)).

5.2 Data transfer time

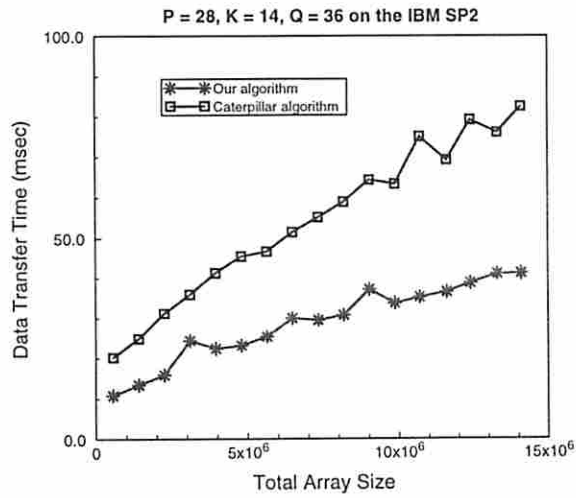
In this subsection, we report the experimental results of the data transfer time of our algorithm and the Caterpillar algorithm. The experiments were performed in the same manner as discussed in Subsection 5.1. The data sets used in these experiments are the same as those used in the previous subsection. The data transfer time of each communication step is first measured. Then the total data transfer time is computed by summing up the measured time for all the communication steps. The methodology for measuring the time is shown in Figure 18.



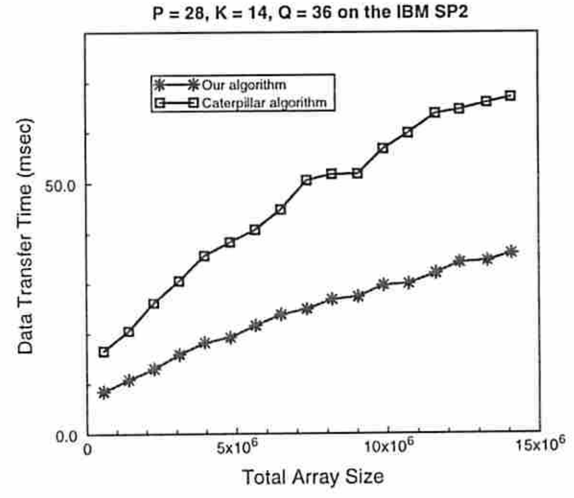
(a) Maximum Time



(b) Average Time



(c) Median Time



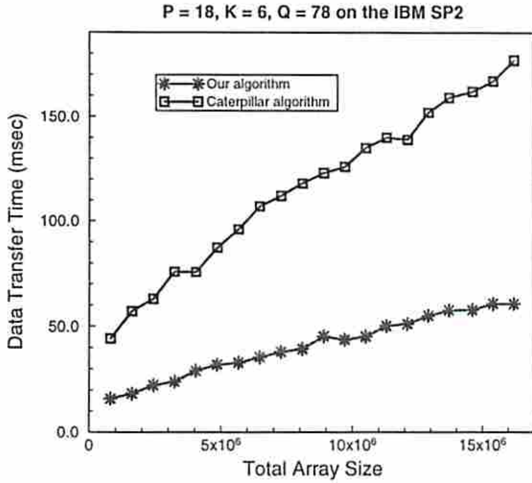
(d) Minimum Time

Figure 19: Maximum, average, median, and minimum data transfer times for $\Re_2(28, 14, 36)$.

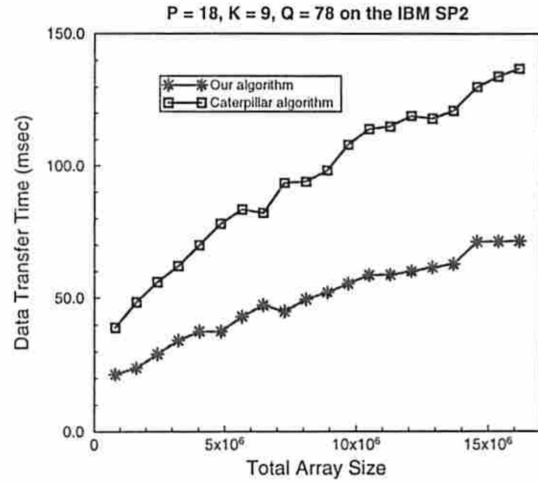
In Figure 19, the data transfer time of our algorithm and that of the Caterpillar algorithm are reported. The experiments were performed on the IBM SP2. Figure 19(a) reports the maximum data transfer time (`Tmax` in Figure 18). A large variance in the measured values was observed. Figure 19(b) and (c) show the average time (`Tavg` in Figure 18) and the median time (`Tmed` in Figure 18) of the data transfer time, respectively. These values are computed using the maximum time(`Tmax`). Figure 19(d) shows the minimum data transfer time(`Tmin`). This plot is a more accurate observation of the data transfer time since the minimum time has the smallest component due to OS interference and other effects related to the environment. Therefore, it is a more accurate comparison of the relative performance of the redistribution algorithms. In the remainder of this section, we show plots corresponding to `Tmin` only.

The redistribution $\Re_2(28, 14, 36)$ is a non all-to-all communication case. The messages in each communication step are of the same size. The total number of communication steps is 18 using our algorithm, where as the total number of steps is 36 using the Caterpillar algorithm. Therefore, the data transfer time of our algorithm is theoretically 50% of that of the Caterpillar algorithm. In the experimental results (see Figure 19(d)), the redistribution time of our algorithm is between 49.2% and 53.7% of that of the Caterpillar algorithm. Figure 20 shows several experimental results for the non all-to-all communication case. Similar reductions in time were achieved in these experiments.

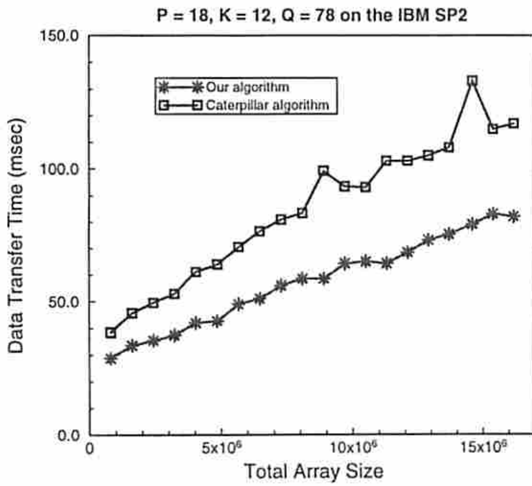
Figure 21 reports the experimental results for the all-to-all communication case with different message sizes. The data transfer time in the all-to-all communication case is sensitive to network contention since every source processor communicates with every destination processor. For $\Re_4(28, 6, 36)$, both algorithms have the same number of steps (36). Within a superblock, half the messages are two blocks while the other half are one block. In our algorithm, equal-sized messages are transferred in each communication step. Therefore, during half the steps, two block messages are sent while one block messages are sent during the other half. The Caterpillar algorithm does not attempt to send equal-sized messages in each communication step. Therefore, the data transfer time in each step is determined by the time to transfer the largest message. Theoretically, the data transfer time of our algorithm is reduced by 25% when compared with that of the Caterpillar algorithm. In experiments with large message sizes, we achieved up to 15.5% reduction. With small messages, both algorithms have approximately the same performance since the start-up time dominates the data transfer time. Other experimental results are reported in Figure 21(b), (c), and (d).



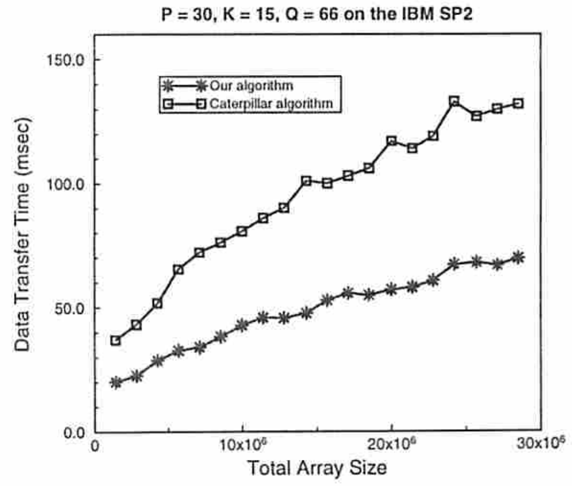
(a) $\Re_{16}(18,6,78)$



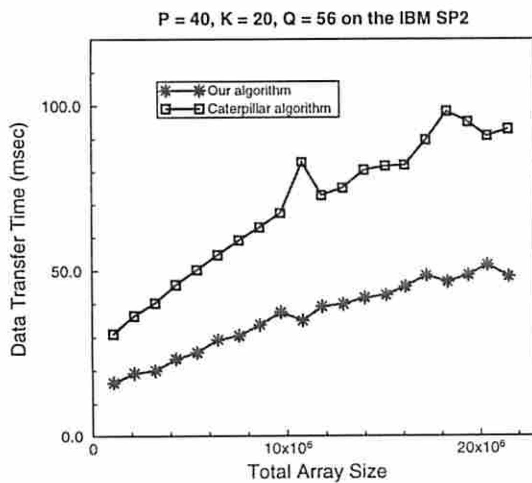
(b) $\Re_{16}(18,9,78)$



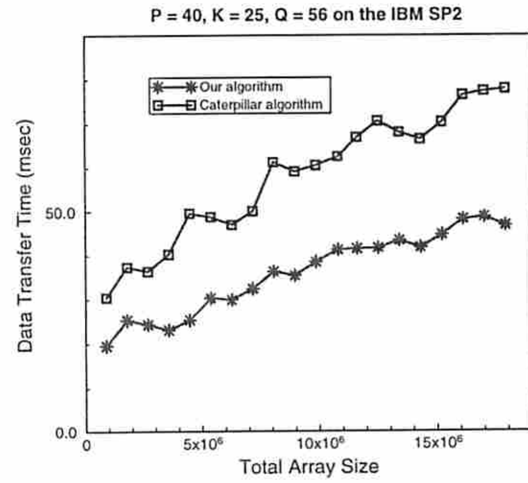
(c) $\Re_{16}(18,12,78)$



(d) $\Re_{16}(30,15,66)$

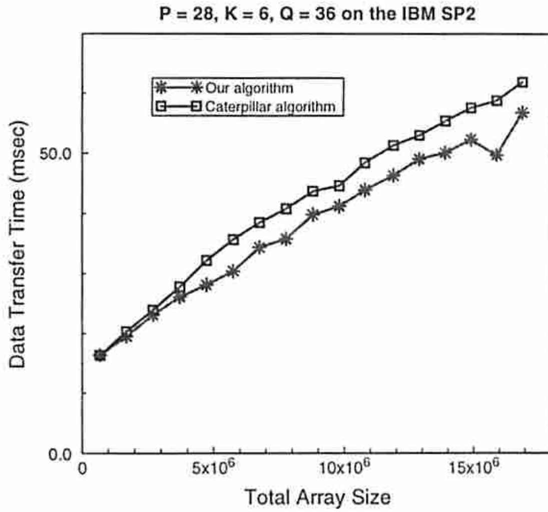


(e) $\Re_{16}(40,20,56)$

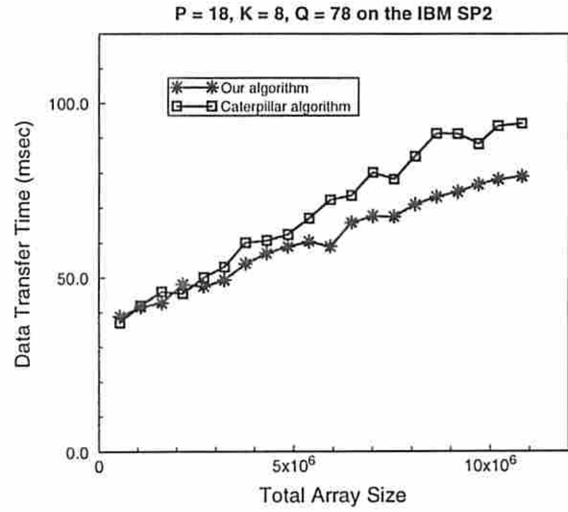


(f) $\Re_{16}(40,25,56)$

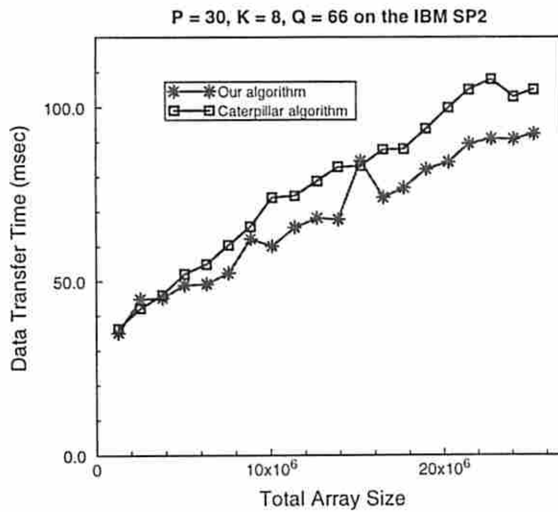
Figure 20: Data transfer time for non all-to-all communication cases.



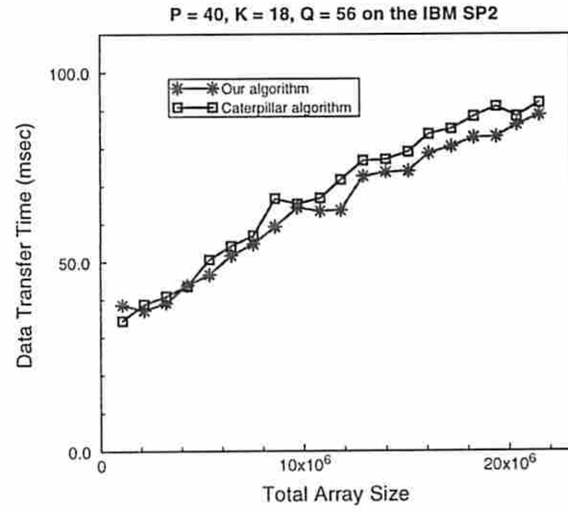
(a) $\Re_4(28,6,36)$



(b) $\Re_{16}(18,8,78)$



(c) $\Re_{16}(30,8,66)$



(d) $\Re_8(40,18,56)$

Figure 21: Data transfer time for all-to-all communication cases with different message sizes.

Table 6: Comparison of schedule computation time ($\mu secs$): The procedure in [8] was used for bipartite matching.

P and Q	K	Our Scheme	Bipartite Matching Scheme [5]
P = 16 Q = 16	4	30	866
	8	51	2390
	12	78	3960
P = 32 Q = 32	8	52	7453
	16	94	16761
	24	147	25613
P = 64 Q = 64	16	94	62443
	32	179	133213
	48	277	207763
P = 128 Q = 128	32	176	534768
	64	346	1167586
	96	577	1842415

5.3 Schedule computation time

The time for computing the schedule in the Caterpillar algorithm as well as in our algorithm is negligible compared with the total redistribution time. Even though the schedule in the Caterpillar algorithm is simpler than ours, the Caterpillar algorithm needs time for index computation to identify the blocks to be packed in a communication step. This time is approximately the same as our schedule computation time.

The schedule computation time of the bipartite matching scheme [4] is much higher than that of the Caterpillar algorithm and our algorithm. It is in the range of hundreds of *msecs* which is quite significant. The schedule computation time of the bipartite matching scheme increases rapidly as the number of processors increases. On the other hand, our algorithm computes the communication schedule efficiently. Each processor computes its entries in the send communication schedule table. Thus, the schedule is computed in a distributed way. The schedule computation time is in the range of 100's of $\mu secs$. The comparison between our scheme and the bipartite matching scheme with respect to the schedule computation time is shown in Table 6. Here, the time for our scheme includes the index computation time. For the bipartite matching scheme, the time shown is the schedule computation time only.

6 Conclusions

In this paper, we showed an efficient algorithm for performing redistribution from *cyclic*(x) on P processors to *cyclic*(Kx) on Q processors. The proposed algorithm was represented using generalized circulant matrix formalism. Our algorithm minimizes the number of communication steps and avoids destination node contention in each communication step. The network bandwidth is fully utilized by ensuring that messages of the same size are transferred in each communication step. Therefore, the total data transfer cost is minimized.

The schedule and index computation costs are also important in performing run-time redistribution. In our algorithm, the schedule and the index sets are computed in $O(max(P,Q))$ time. This computation is extremely fast compared with the bipartite matching scheme in [4] which takes $O((P+Q)^4)$ time. Our schedule and index computation times are small enough to be negligible compared with the data transfer time making our algorithms suitable for run-time data redistribution.

Acknowledgment

We would like to thank the staff at MHPCC for their assistance in evaluating our algorithms on the IBM SP-2. We also would like to thank Manash Kirtania for his assistance in preparing this manuscript.

References

[1] J. Bruck, C.-H. Ho, S. Kipnis, and Weathersby. Efficient Algorithms for All-to-All Communications in Muti-Port Message-Passing Systems. In *6th Annual ACM Symp. on Para. Alg. and Arch.*, pages 298–309, July 1994.

[2] J. Choi, J. Dongarra, and D. Walker. Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers. Technical Report ORNL/TM-12309, Oak Rigde National Laboratory, October 1993.

[3] Y.C. Chung, C.H. Hsu, and S.W. Bai. A Basic-Cycle Calculation Technique for Efficient Dynamic Data Redistribution. In *IEEE Tans. on Parallel and Distributed Systems*, Vol. 9, No. 4, April 1998.

[4] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro and Y. Robert. Scheduling Block-Cyclic Array Redistribution. University of Tennessee Computer Science Technical Report, UT-CS-97-349, LAPACK Working Note 120, February 1997.

[5] J. Dongarra and *et al.* ScaLAPACK: A Portable Linear Algebra Library of Distributed Memory Computers - Design Issues and Performance. Technical Report LAPACK Working Note 95, Oak Rigde National Laboratory, 1995.

[6] R.A. Games. Benchimarking Methodology for Real-Time Embedded Scalabla High Performance Computing. MITRE *Technical Report MTR96B0000020, March 1996.*

[7] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation Techniques for Block-Cyclic Distributions. In *Proc. of Intl. Conf. on Supercomputing*, pages 392–403, July 1994.

[8] R. Jonker and A. Volgenant. A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems. on *http://207.158.230.188/assignment.html.*

[9] E.T. Kalns and L.M. Ni. Processor Mapping Techniques Toward Efficient Data Redistribution. *Proc. of International Parallel Processing Symposium*, April 1994.

[10] S.D. Kaushik, C.-H. Huang, R.W. Johnson, and P. Sadayappan. An Approach to Communication Efficient Data Redistribution. In *Proc. of Intl. Conf. on Supercomputing*, pages 364–373, July 1994.

[11] S.D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multiphase Array Redistribution: Modeling and Evaluation. Technical Report OSU-CISRC-9/94-TR52, Ohio State University, September 1994.

[12] S.D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multiphase Array Redistribution: Modeling and Evaluation. In *Proc. of Intl. Parallel Processing Symposium*, pages 441–445, 1995.

[13] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[14] Y.W. Lim, P.B. Bhat, and V.K. Prasanna. Efficient Algorithms for Block-Cyclic Redistribution of an Array. In *Proc. IEEE Symposium on Parallel and Distributed Processing*, October 1996.

[15] Y.W. Lim and V.K. Prasanna. Scalable Portable Implementations of Space-Time Adaptive Processing. In *10th Inter. Conf. High Perf. Comp.*, 1996.

[16] W. Liu, W. Kostis, and V.K. Prasanna. Communication Issues in Heterogeneous Embedded Systems. In *Proc. of Workshop on Para. and Dist. Real Time Sys.*, April 1996.

[17] W. Liu and V.K. Prasanna. Design of Application Software for Embedded Signal Processing. In *IEEE Signal Processing Magazine*, September 1998.

[18] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, University of Tennessee, Knoxville, May 1994.

[19] L. Prylli and B. Tourancheau. Fast Runtime Block Cyclic Data Redistribution on Multiprocessors. *Journal of Parallel and Distributed Computing*, volume 45, August 1997

[20] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Proc. of 5th Symp. Frontiers of Massively Parallel Computation, McLean, VA*, pages 342–349, Feburary 1995.

[21] J. Suh, M. Ung, and V.K. Prasanna. Parallel Implementation of Synthetic Aperture Radar on High Performance Computing Platforms. *International Conference on Algorithms And Architectures for Parallel Processing '97*, December 1997.

[22] R. Thakur, A. Choudhary, and G. Fox. Runtime Array Redistribution in HPF Programs. In *Proc. of Scalable High Performance Computing Conference*, pages 309–316, May 1994.

[23] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient Algorithms for Array Redistribution. To appear in IEEE Trans. on Parallel and Distributed Systems

[24] D.W. Walker and S.W. Otto. Redistribution of Block-Cyclic Data Distributions Using MPI. Technical Report ORNL/TM-12999, ORNL, June 1995.

[25] C.-L. Wang, P.B. Bhat, and V.K. Prasanna. High-Performance Computing for Vision. *Proceedings of IEEE*, 84:931–946, 1996.

[26] J. Ward. Space-Time Adaptive Processing for Airborne Radar. Technical Report 1015, MIT Lincoln Lab., December 1994.