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/

Cauligi 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 when the total redistribution time including the data transfer and schedule and index computation times are considered.

*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 [2, 10, 21]. Examples of such applications include the multi-dimensional 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.

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. All these four steps contribute to the total array redistribution cost. We briefly explain these 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 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.

Table 1 summarizes the key features of the well known data distribution algorithms in the literature [4, 13, 14, 15, 22]. 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. 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 [3, 17, 23]. Although the schemes in [3, 17, 23] 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 [11]. It uses a simple round robin schedule to avoid node contention.

	Key Features	
	Schedule & Index Computation	**Communication**
PITFALLS [14]	• No communication scheduling • Index computation using a line segment formalism	• Node contention occurs • Does not minimize the transmission cost
BCC [21]	• No communication scheduling • Efficient index computation • Source and destination sets are same	• Node contention occurs • Does not minimize the transmission cost
Caterpillar [11]	• 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 [5]	• 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

Table 1: Comparison of various schemes for array redistribution.

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 [5] reduce the data transfer cost, however, the schedule computation cost is significant. The bipartite graph matching used in [5] 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 [5] 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 [5] 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 [1], 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.

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 [5]. 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 explains our table-based framework. It also discusses the generalized circulant matrix formalism for deriving conflict free communication schedules. Section 3 explains our redistribution algorithm and index computation. Section 4 reports our experimental results on the IBM SP-2. Concluding remarks are made in Section 5.

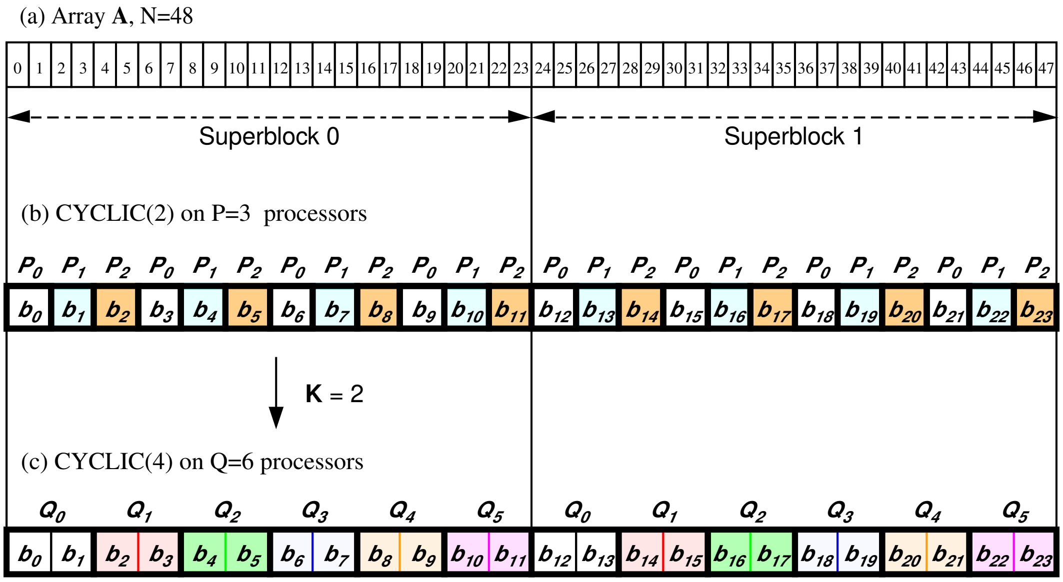


Figure 1: Block-cyclic redistribution from array point of view: (a) the array of elements, (b) $cyclic(x)$ on P processors, (c) from $cyclic(x)$ on P processors to $cyclic(Kx)$ on Q processors. In this example, $x = 2$, $P = 3$, $Q = 6$ and $K = 2$.

2 Our Approach to Redistribution

In this section, we present our approach to block-cyclic redistribution problem. In subsection 2.1, 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 2.3, we discuss the generalized circulant matrix formalism which allows us to compute communication schedule efficiently.

2.1 Array and processor points of view

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 i^{th} 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 $0 \le i < \frac{N}{x}$. 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 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

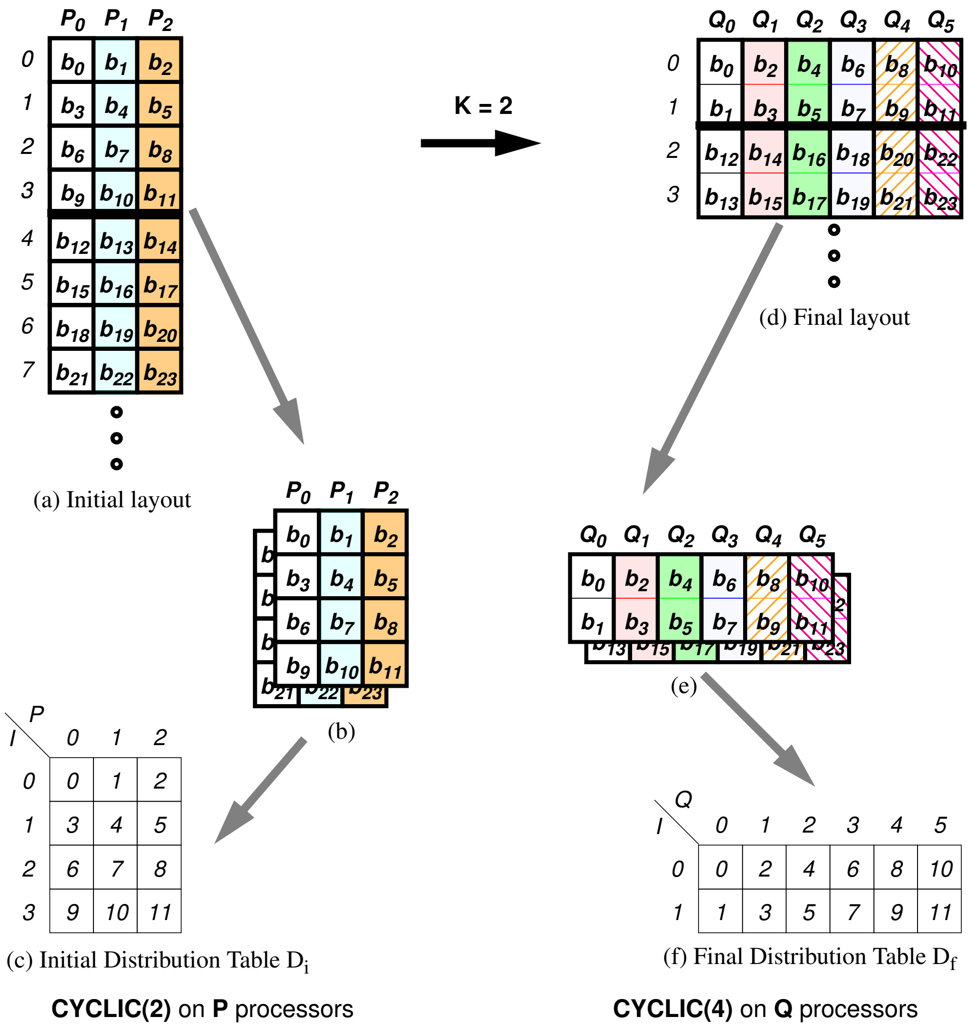


Figure 2: Block-cyclic redistribution from $cyclic(x)$ on P processors to $cyclic(Kx)$ on Q processors from processor point of view. In this example, $P = 3$, $Q = 6$ and $K = 2$.

the block movement pattern. For example, in Figure 1, 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 1, superblock size is $lcm(3, 2 \cdot 6) = 12$. In the next superblock, blocks b_{12} to b_{23} are moved 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 i^{th} local block of the j^{th} processor. Figure 2 shows the example of $\Re_2(3, 2, 6)$ from

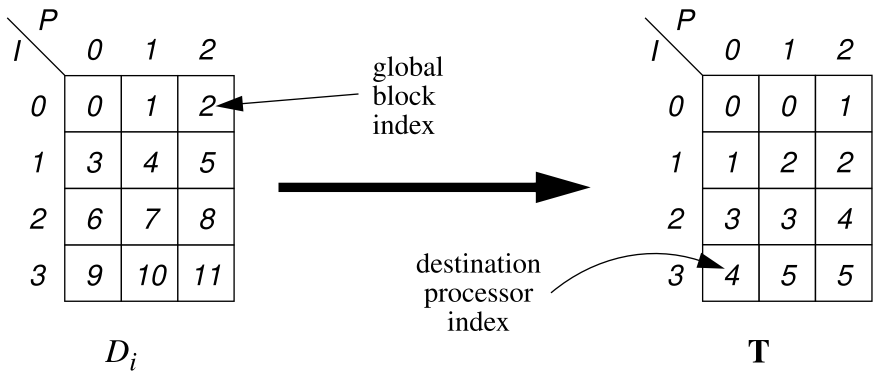


Figure 3: The destination processor table **T**.

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 2(a) and (d) respectively.

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* D_i (*final distribution table* D_f). These are shown in Figure 2(c) and (f), respectively. The cyclic redistribution problem essentially involves reorganizing blocks within each superblock from an initial distribution table D_i to a final distribution table D_f.

2.2 A Table-based framework for redistribution

Given the redistribution parameters, P, K, and Q, each block's location in D_i and D_f can be determined. Through redistribution, each block moves from its initial location in D_i to the final location in D_f. 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 D_i to D_f, which can be decomposed into independent column and row reorganizations.

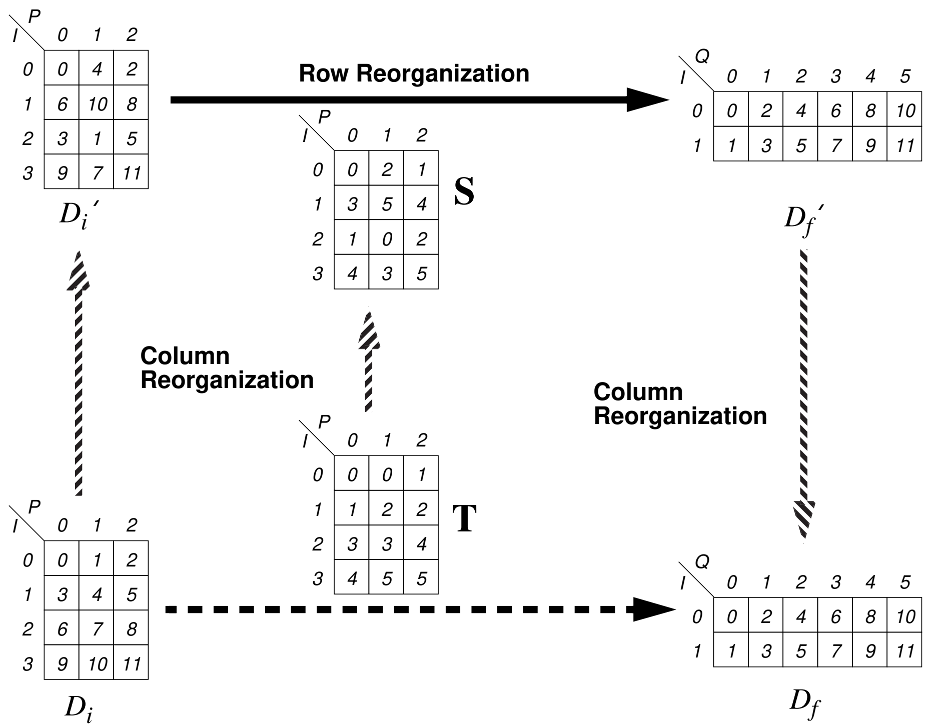


Figure 4: Table conversion process for redistribution.

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 constructed by replacing each block index in the initial distribution table with its destination processor index as shown in Figure 3. This is denoted as *destination processor table* (*dpt*) $\mathbf{T}$. The $(i,j)^{th}$ entry of $\mathbf{T}$ is the destination processor index of i^{th} local block in source processor j and $0 \le i < L_s$ i.e. $\mathbf{T}$ considers only one superblock. It is a $L_s \times P$ matrix. Each row corresponds to a communication step. In our algorithm, during a communication step, a processor sends data to atmost one destination processor. If $Q \ge P$, 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 4 shows our table-based framework for redistribution. To convert the initial distribution table D_i to the final distribution table D_f, (*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 4, 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 the *send communication schedule table*, $\mathbf{S}$. In section 3, we discuss how these reorganizations are performed. $\mathbf{S}$ is a $L_s \times P$ matrix as well. Each entry of $\mathbf{S}$ is a destination processor index and each row corresponds to a contention-free communication step. To maintain the correspondence between D_i and $\mathbf{T}$, the same set of column reorganizations is applied to D_i which results in a distribution table, D'_i corresponding to $\mathbf{S}$. In a communication step, blocks in a row of D'_i are transferred to their destination processors specified by the corresponding entries in $\mathbf{S}$. Referring to Figure 4, 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 $\mathbf{S}$. Such a step is called row reorganization. The distribution table D'_f corresponding to the received blocks in destination processors is reorganized into the final distribution table D_f 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 $\mathbf{S}$ 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 P processors to *cyclic*(x) on Q processors. The redistribution from *cyclic*(Kx) on P processors to *cyclic*(x) on Q 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 P processors to $cyclic(x)$ on Q processors.

2.3 Communication scheduling using generalized circulant matrix

Our framework for communication schedule performs the 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 **S**. 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, $0 \leq l < n$.*

Note that the above definition can be extended to block circulant matrix with the change that "row" is now replaced with "block row".

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 5 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 (*dpt*) **T** is transformed to a generalized circulant matrix **S** with distinct elements in each row.

3 Efficient Redistribution Algorithms

Before discussing communication schedule algorithm for redistribution, we classify communication patterns into 3 classes for the redistribution problem $\Re_x(P, K, Q)$ (See [5] for an alternative formulation for the $cyclic(x)$ to $cyclic(y)$ problem) according to the following Lemma. Let G be $gcd(P, KQ)$.

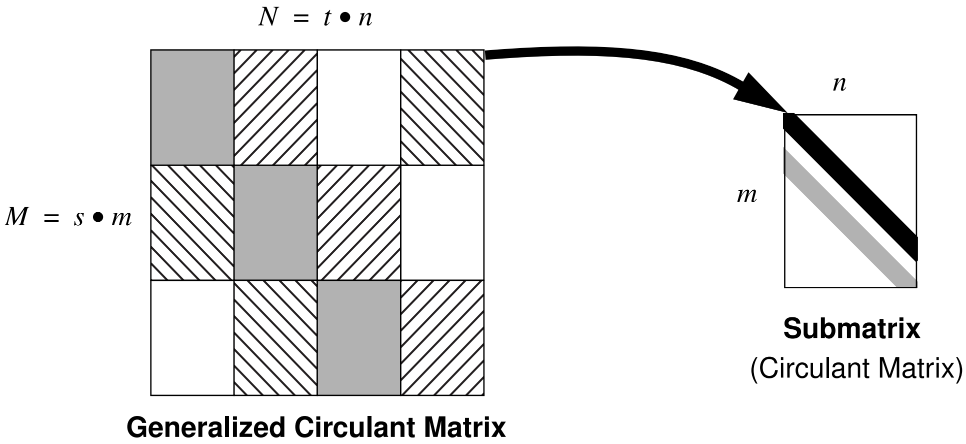


Figure 5: Generalized circulant matrix.

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 $K \neq \alpha G$.*

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.

3.1 Non all-to-all communication

Given the redistribution parameters P, Q, and K, we get the $L_s \times P$ initial distribution table D_i 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* $\mathbf{T}$, every K_1^{th} 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 reorganizing blocks in each column of the block matrix and reorganizing individual columns within a submatrix by appropriate amounts. This results in a generalized circulant matrix which is our communication schedule matrix $\mathbf{S}$. In

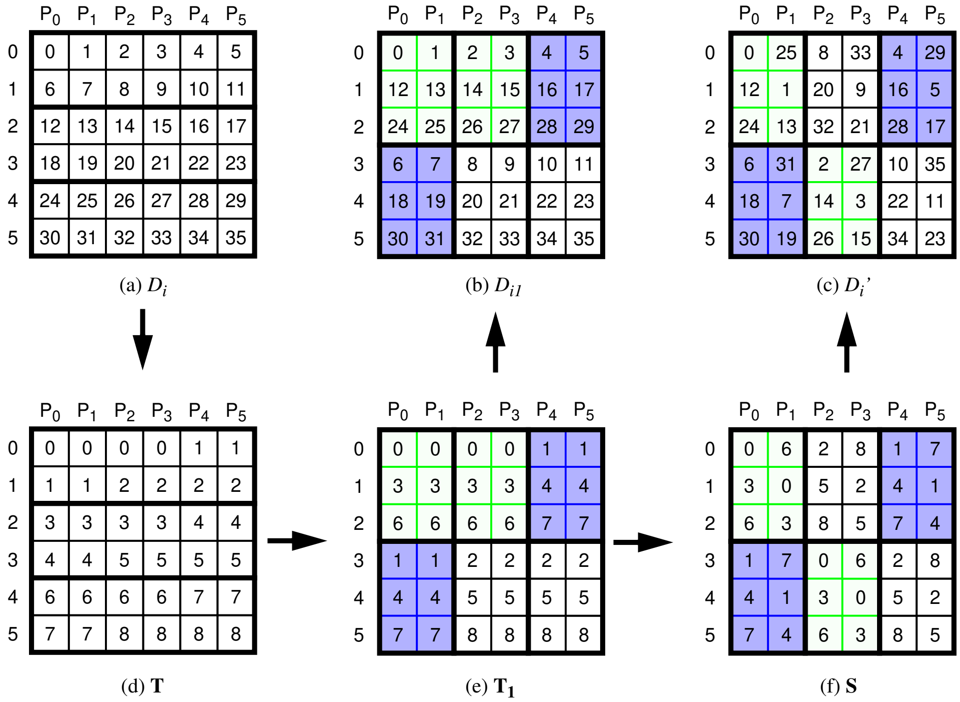


Figure 6: Steps of column reorganization.

this procedure, the K identical values in row 0 of the *dpt* $\mathbf{T}$ are distributed to K distinct rows, and hence, row 0 has distinct values. Since $\mathbf{S}$ is a generalized circulant matrix and all the elements in each row are distinct, we achieve a conflict-free schedule. A rigorous proof of this fact that any *dpt* $\mathbf{T}$ can be transformed to a generalized circulant matrix using these column reorganizations can be found in [25]. With this schedule, in each communication step, P source processors transfer an equal-size message to P distinct destination processors. It ensures that the network bandwidth is fully utilized. The number of communication steps is also minimized. Therefore the data transfer cost is minimized. In the above reorganization, an element is moved within its column. So, it does not incur any interprocessor communication. Figure 6 shows an example where *dpt* $\mathbf{T}$ of $\Re_x(6,4,9)$ is converted to a generalized circulant matrix form $\mathbf{S}$ by column reorganizations. In this example, $L_s = 6$, $G_1 = 2$, $K_1 = 2$, $P_1 = 3$, and $Q_1 = 3$. Figure 6(a) shows the initial distribution table, D_i, and Figure 6(d) shows the corresponding *dpt* $\mathbf{T}$. Rows of D_i and $\mathbf{T}$ are shuffled, as shown in Figure 6(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 are shown in Figure 6(c) and (f). Figure 6(f) is a generalized circulant matrix $\mathbf{S}$ which gives a communication schedule.

While the *dpt* $\mathbf{T}$ is converted to the send communication schedule table $\mathbf{S}$, the same set of reorganizations are applied to the initial data distribution table D_i. It is converted to D'_i as shown in Figure 6. It can be expensive to reorganize large amount of data within a local memory. 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* $\mathbf{D_s}$. Each entry of $\mathbf{D_s}$ is the local block index of the corresponding entry of D'_i. Each entry of $\mathbf{S}$, $\mathbf{S}(i,j)$, points to the destination processor of the corresponding entry of $\mathbf{D_s}$, $\mathbf{D_s}(i,j)$. Our scheme computes the schedule and data index set at the same time.

Through algebraic manipulations, the above procedure gives the following two equations to directly compute the individual entries of $\mathbf{S}$ and $\mathbf{D_s}$. In these equations, $i_1 = i/K_1$ denotes the quotient of integer division and $i_2 = i \bmod K_1$ denotes the remainder of the integer division. Similarly, $j_1 = j/G_1$ and $j_2 = j \bmod G_1$.

$$\mathbf{S}(i,j) = \{\{n(j_1 - i_1)\} \bmod P_1 + \{(i_2 - j_2) \bmod Q_1\}P_1\} \bmod Q \quad (1)$$

$$\mathbf{D_s}(i,j) = \{m(j_1 - i_1)\} \bmod K_1 + \{(i_2 - j_2) \bmod Q_1\}K_1 \quad (2)$$

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

The proof of correctness of the above mathematical formulations can be found in [25]. The above formulae for computing the communication schedule and index set for redistribution are extremely efficient compared with the methods presented in [5], 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 $O(max(P,Q))$. The amortized cost to compute a step in the communication schedule and index set computation is $O(1)$. 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 total data transfer cost.

3.2 All-to-all communication with different message sizes

The all-to-all communication case arises if $G(= G_1G_2) < K$ 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* is a $L_s \times P$ matrix, where $L_s = K_1Q_1$. Since $Q = Q_1G_2$ and $G_2 \leq K_1$, $L_s \geq Q$. Therefore, each column has more entries than Q destination processors. In each column, several blocks are transferred to

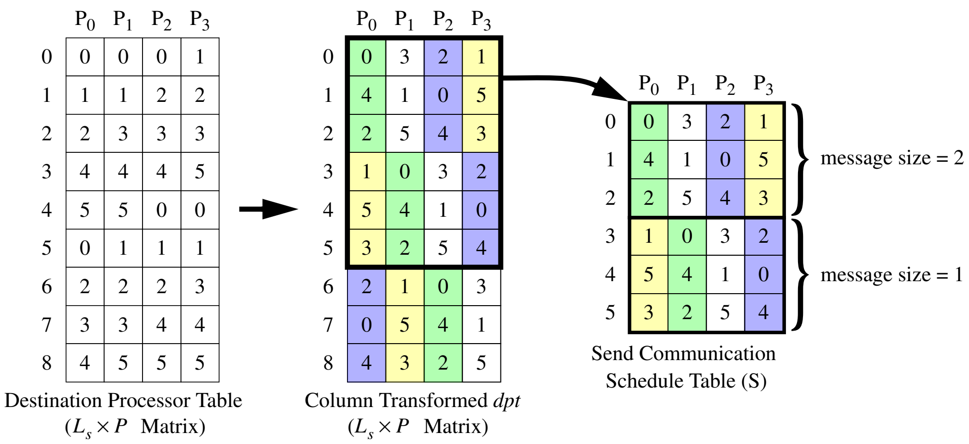


Figure 7: Example illustrating an all-to-all case with different message sizes: $\Re_x(4, 3, 6)$.

the same destination. The column reorganizations as stated in Section 3.1 are applied to the *dpt* $\mathbf{T}$, which results in a generalized circulant matrix which 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 $\mathbf{S}$. 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 $\mathbf{S}$ 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 7 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 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 the $\mathbf{S}$ table. The 3^{rd} row is folded onto the 1^{st} row. Hence, the message size in the 1^{st} row is 2 and that in the 2^{nd} row is 1. If K_1 is a multiple of G_2, the message size in every row will be the same. Therefore, the network bandwidth is fully utilized by sending equal sized messages in each communication step.

3.3 Data transfer cost

In distributed memory model, the communication cost has two parameters which are start-up time and transfer time. The start-up time, T_s, is incurred once for each communication event and is independent of the 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 total transmission time for a message is proportional to its size. Thus, the total communication 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 redistribution. Using this distributed memory model, the performance of our algorithm can be analyzed as follows. Assume that an array with N elements is initially distributed $cyclic(x)$ on P processors and then redistributed to $cyclic(Kx)$ on Q processors. Using our algorithms, the communication costs for performing $\Re_x(P, K, Q)$ are *(i)* $L_sT_s + \frac{N}{P}\tau_d$ in the case of non all-to-all communication, and *(ii)* $QT_s + \frac{N}{P}\tau_d$ in the case of all-to-all communication. The proof of this analysis will be found in [25].

4 Experimental Results

Our experiments were conducted on the IBM SP2. The algorithms were written in C and MPI.

Table 2 shows a comparison of the proposed algorithm with the Caterpillar algorithm[11] and the bipartite matching scheme[5] with respect to the data transfer cost and schedule and index computation costs. For the all-to-all communication case with an 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, it is not considered in Table 2. In Table 2, 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

Table 2: 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 [11]	$Q\left(T_s + \tau_d \frac{M}{L_s}\right)$	$O(Q)$	$QT_s + \tau_d \sum_{i=0}^{Q-1} m_i$	$O(Q)$
Bipartite matching scheme [5]	$L_s\left(T_s + \tau_d \frac{M}{L_s}\right)$	$O((P+Q)^4)$	$QT_s + \tau_d M$	$O((P+Q)^4)$
Our algorithm	$L_s\left(T_s + \tau_d \frac{M}{L_s}\right)$	$O(Q)$	$QT_s + \tau_d M$	$O(Q)$

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

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 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 4.1 reports experimental results of the overall redistribution time of our algorithm and the Caterpillar algorithm. Subsection 4.2 shows experimental results for the data transfer time of our algorithm and the Caterpillar algorithm. Subsection 4.3 compares our algorithm and the bipartite matching scheme with respect to the schedule computation time.

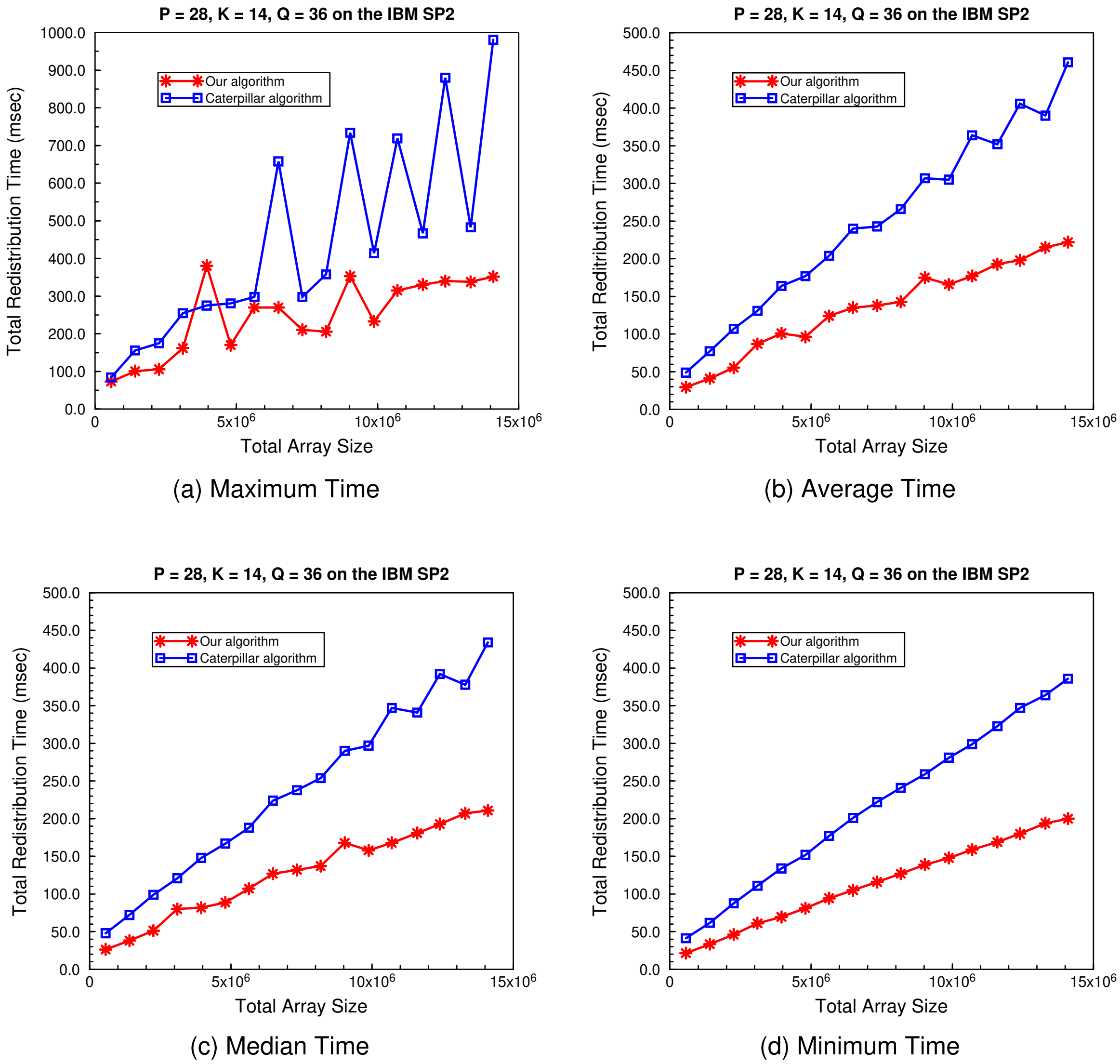
```
for (j=0; j<n1; j++) {
   ts = MPI_Wtime()
   /* redistribution routine */
   compute schedule and index set
   for (i=0; i<n2; i++) {
      if (source processor) { /* source processor */
         pack message
         send message to a destination processor
      } else {                /* destination 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 8: Steps for measuring the redistribution time.

4.1 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 computation 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 8. 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. The redistribution was executed `n1` times. Each processor measures `node_time[j]` in the j^{th} 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



(a) Maximum Time (b) Average Time

(c) Median Time (d) Minimum Time

Figure 9: The maximum, average, median, and minimum total redistribution time for $\Re_2(28, 14, 36)$.

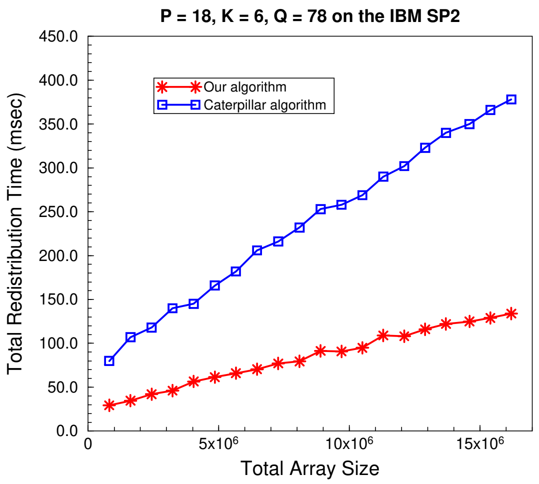
average `node_time` over $(P+Q)$ processors is computed and saved as `tavg`. The measured value is stored in an array `T`, as shown in Figure 8. After the redistribution is performed `n1` times, the maximum, minimum, median, and average total redistribution time are computed over `n1` runs. In our experiments, `n1` was set to 20.

In Figure 9, the total redistribution time of our algorithm and the Caterpillar algorithm are 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 9(a) shows the maximum time (`Tmax` in Figure 8). It was observed that there was a large variance in the measured values. Figure 9(b) shows the results of the average time (`Tavg` in Figure 8). Figure 9(c) shows the results using the median time (`Tmed` in Figure 8). 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 9(d) shows the minimum time for redistribution (`Tmin` in Figure 8). 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.

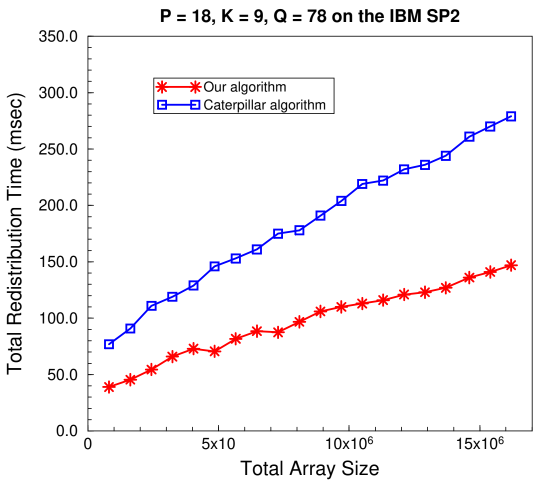
The redistribution $\Re_2(28, 14, 36)$ is a non all-to-all communication case. In the non all-to-all communication case, the messages in each communication step are of the same size. The total number of communication steps is 18 in our algorithm, while it is 36 in the Caterpillar algorithm. Therefore, the redistribution time of our algorithm is theoretically 50% of that of the Caterpillar algorithm. In the experimental results shown in Figure 9(d), the redistribution time of our algorithm is between 51.8% and 55.1% of that of the Caterpillar algorithm.

Figure 10 shows several experimental results for the non all-to-all communication case. Figure 10(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. Out experimental results confirm these. Similar reduction in time were achieved in the other experimental results shown in Figure 10.

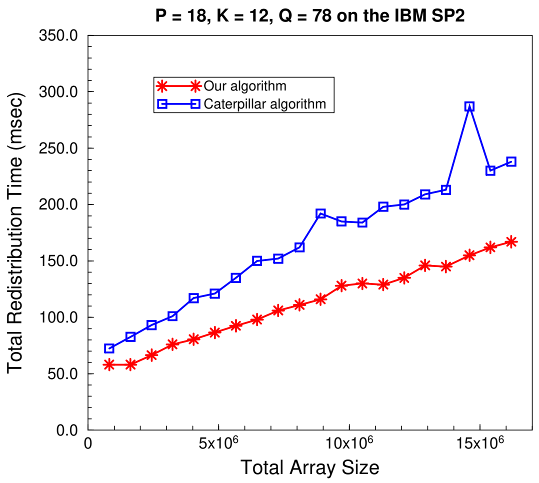
Figure 11 compares the overall redistribution time for the all-to-all communication case with different message sizes. Figure 11(a) reports the experimental results for $\Re_4(28, 6, 36)$. The array



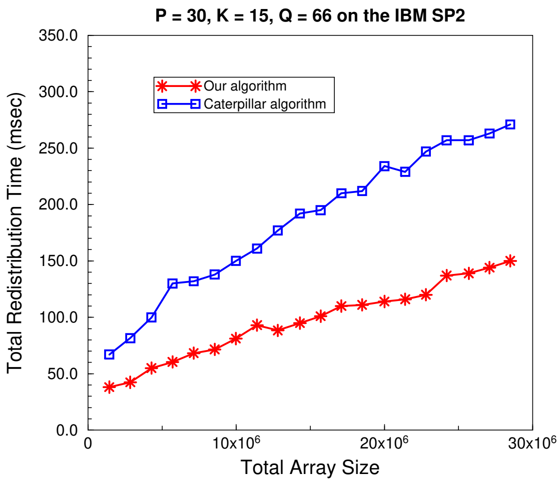
(a) $\Re_{16}(18,6,78)$



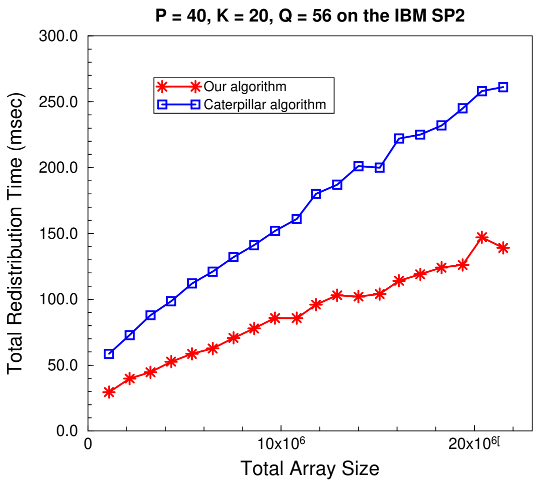
(b) $\Re_{16}(18,9,78)$



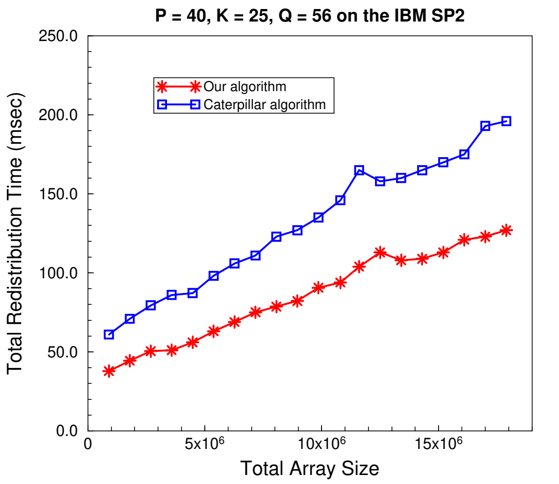
(c) $\Re_{16}(18,12,78)$



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

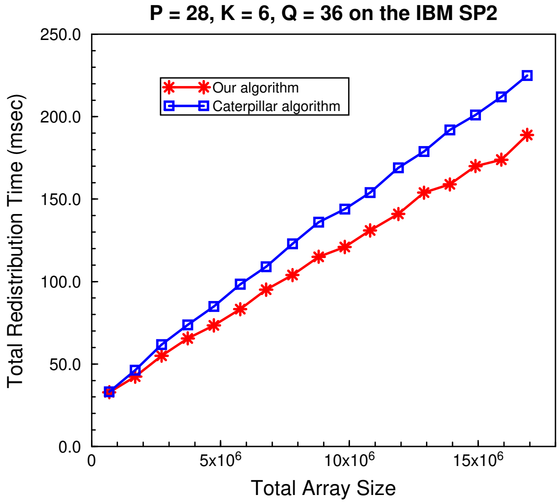


(e) $\Re_{16}(40,20,56)$

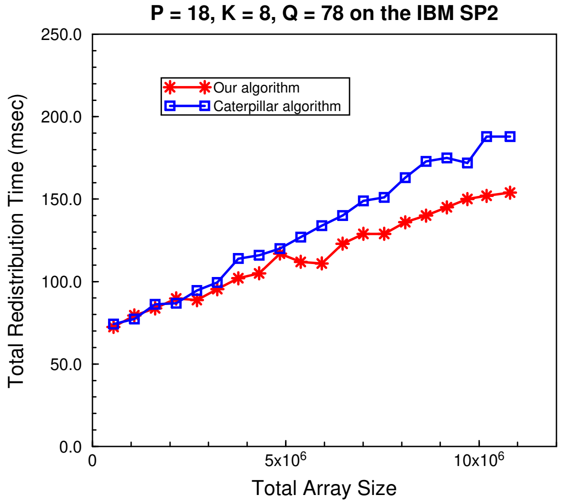


(f) $\Re_{16}(40,25,56)$

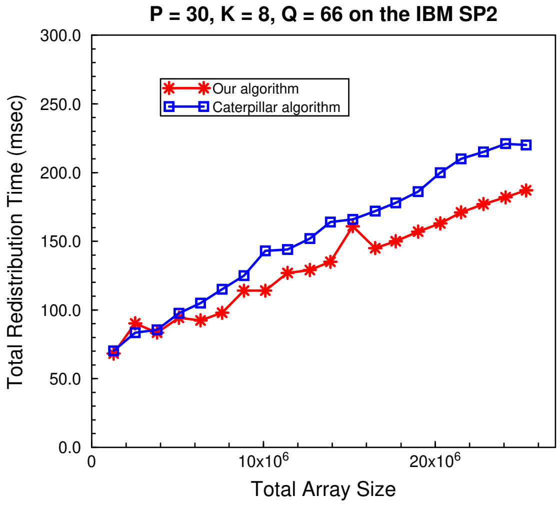
Figure 10: Total redistribution time for non all-to-all communication cases.



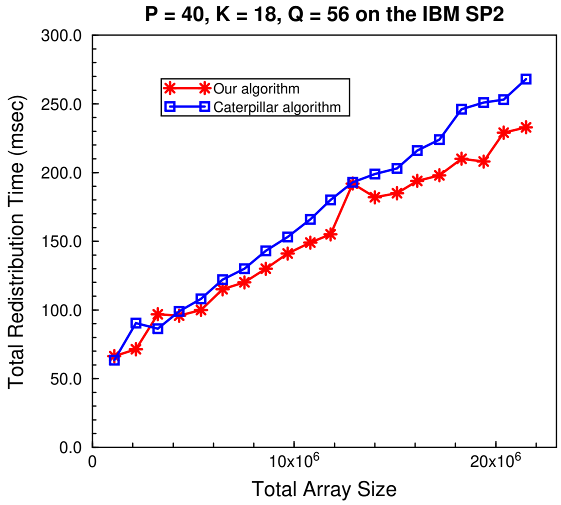
(a) $\Re_4(28,6,36)$



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



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



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

Figure 11: 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 12: Steps in measuring the data transfer time.

size was varied from 677,376 (2.71 Mbytes) to 16,934,400 (67,7 Mbytes). For this case, 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 during the other half one block messages are sent. The Caterpillar algorithm does not attempt to 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 11(b), (c), and (d)).

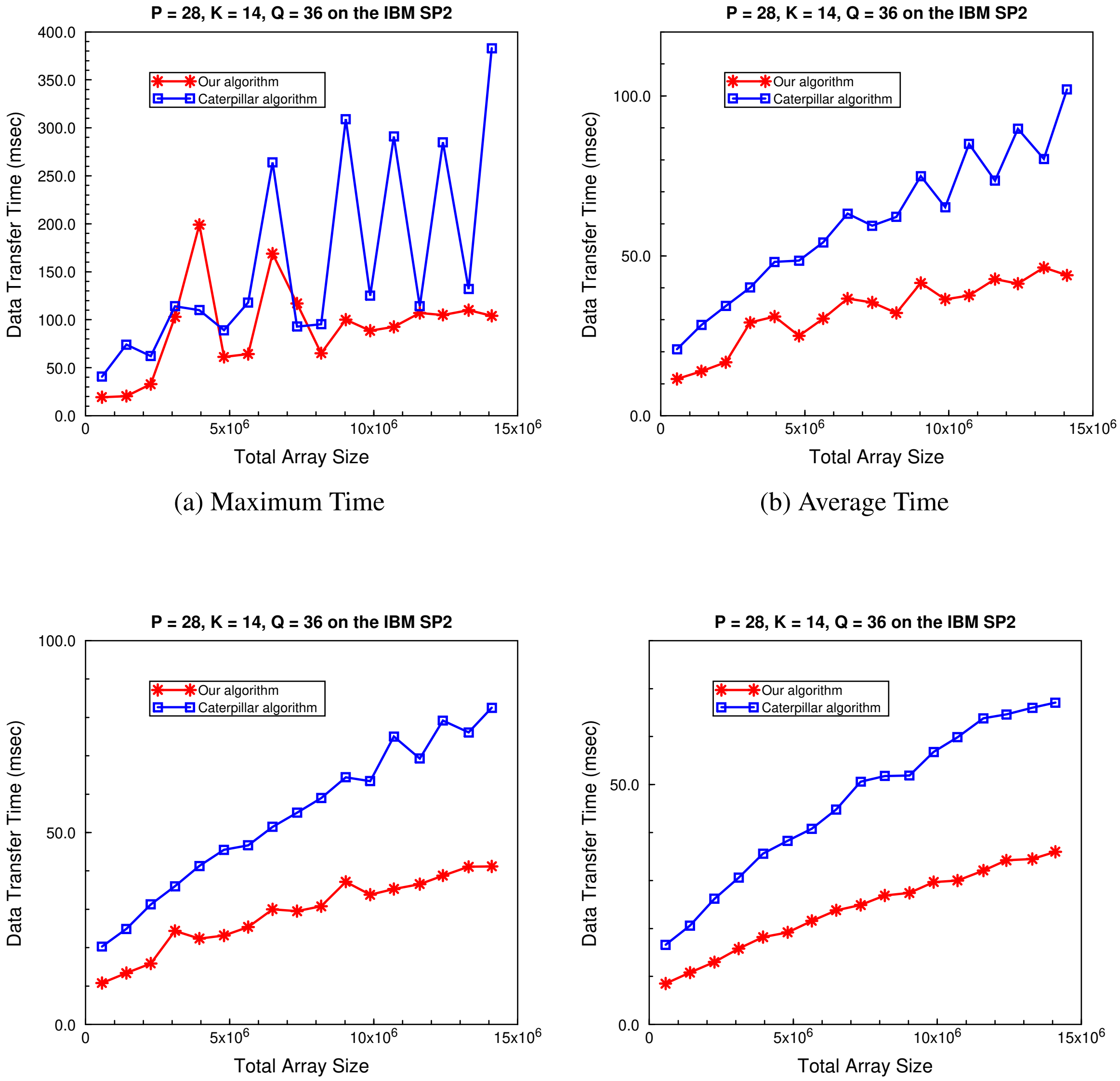
4.2 Data transfer time

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 4.1. 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 12.

In Figure 13, the data transfer time of our algorithm and that of the Caterpillar algorithm are reported. The experiments were performed on the IBM SP2. Figure 13(a) reports the maximum data transfer time (`Tmax` in Figure 12). A large variation in the measured values was observed. Figure 13(b) and (c) show the average time (`Tavg` in Figure 12) and the median time (`Tmed` in Figure 12) of the data transfer time, respectively. These values are computed using the maximum time(`Tmax`). Figure 13(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 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 13(d)), the redistribution time of our algorithm is between 49.2% and 53.7% of that of the Caterpillar algorithm. Figure 14 shows several experimental results for the non all-to-all communication case. Similar reductions in time were achieved in these experiments.

Figure 15 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



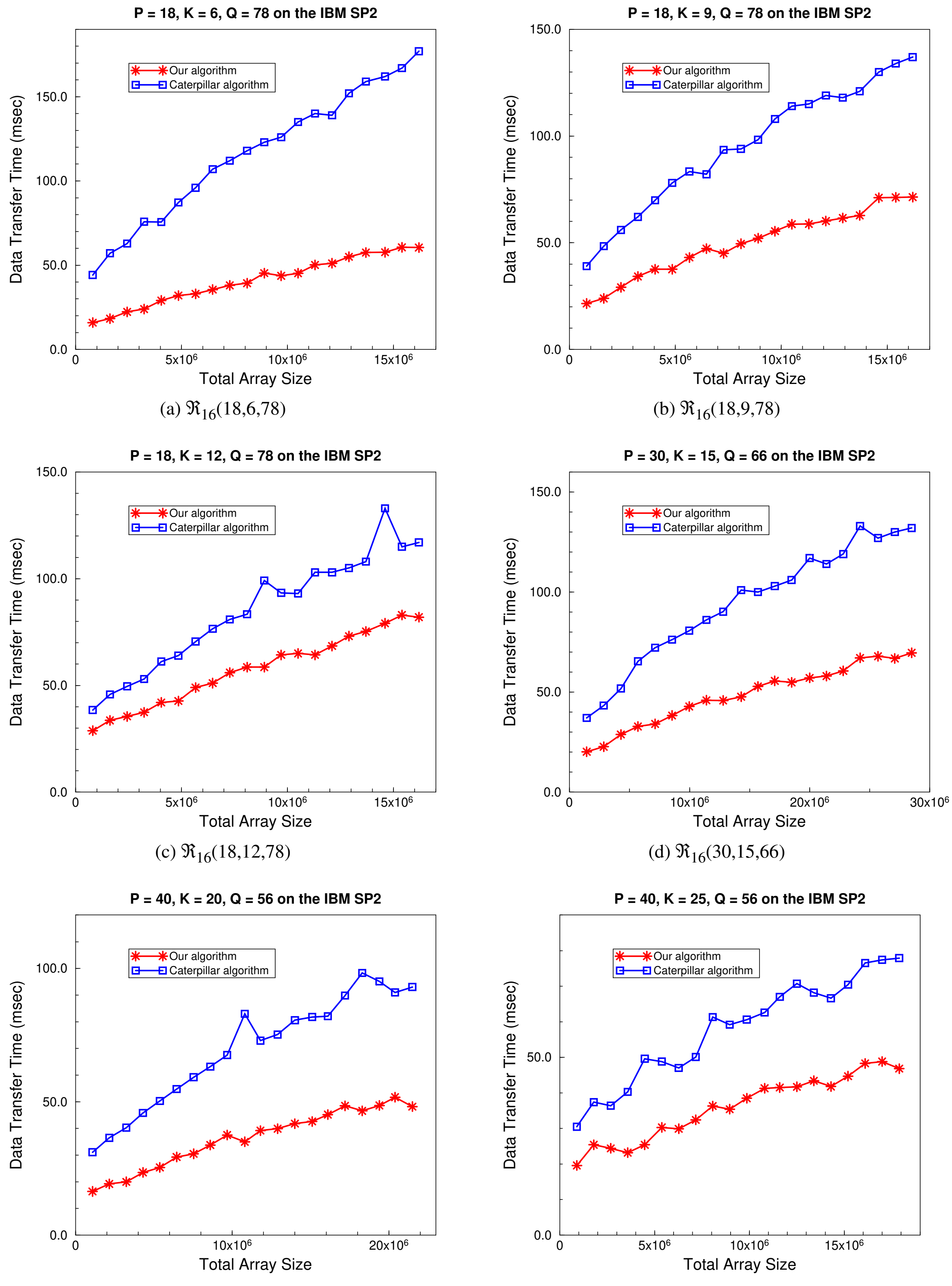
(a) Maximum Time

(b) Average Time

(c) Median Time

(d) Minimum Time

Figure 13: Maximum, average, median, and minimum data transfer times for $\Re_2(28, 14, 36)$.



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

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

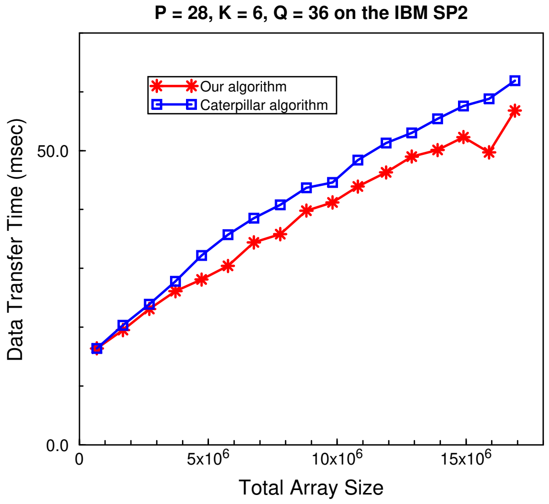
(c) $\Re_{16}(18,12,78)$

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

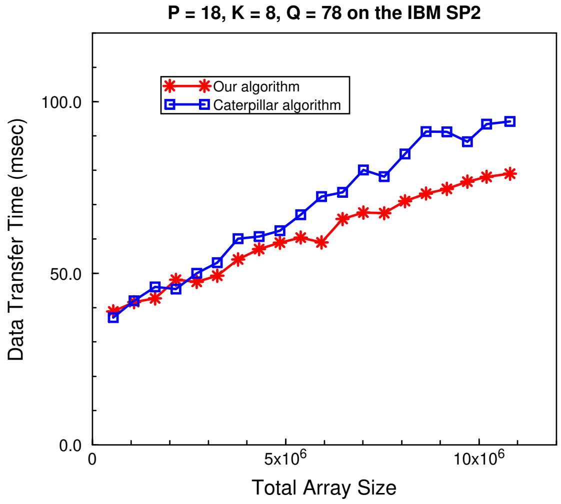
(e) $\Re_{16}(40,20,56)$

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

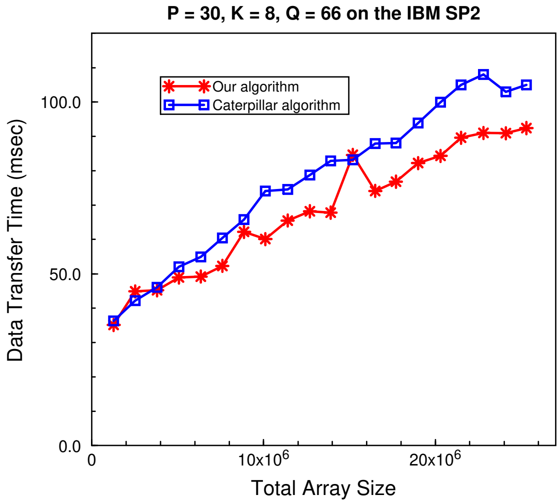
Figure 14: Data transfer time for non all-to-all communication cases.



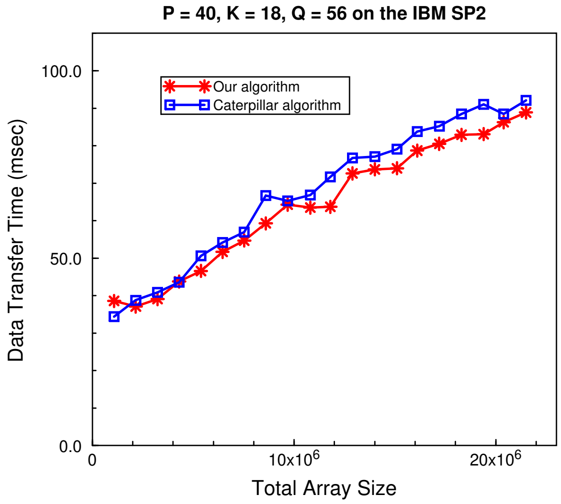
(a) $\Re_4(28,6,36)$



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



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



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

Figure 15: Data transfer time for all-to-all communication cases with different message sizes.

Table 3: Comparison of schedule computation time ($\mu secs$): The procedure in [23] was used for bipartite matching.

P and **Q**	**K**	**Our Scheme**	**Bipartite Matching Scheme [5]**
P = 16 Q = 16	4	30.114	866.386
	8	50.866	2389.655
	12	77.692	3959.604
P = 32 Q = 32	8	51.539	7453.028
	16	94.923	16761.141
	24	146.758	25613.034
P = 64 Q = 64	16	94.245	62443.372
	32	178.569	133212.592
	48	276.587	207762.816
P = 128 Q = 128	32	175.960	534768.000
	64	345.812	1167586.304
	96	576.755	1842415.104

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 15(b), (c), and (d).

4.3 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 [5] 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 cost 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 3. 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.

5 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 [5] 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] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

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

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.*, Apr 1996.

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

[22] 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.

[23] 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.

[24] 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.*

[25] N. Park, V.K. Prasanna, and C. Raghavendra Efficient Algorithms for Block-Cyclic Array Redistribution between processor Sets. *Technical Report, Dept. of EE-Systems, University of Southern California, August 1998.*