Efficient Algorithms for Block-Cyclic Redistribution of Arrays *

Young Won Lim, Prashanth B. Bhat, and Viktor K. Prasanna †

Abstract

The block-cyclic data distribution is commonly used to organize array elements over the processors of a coarse-grained distributed memory parallel computer. In many scientific applications, the data layout must be reorganized at run-time in order to enhance locality and reduce remote memory access overheads. In this paper, we present a general framework for developing array redistribution algorithms. Using this framework, we have developed efficient algorithms that redistribute an array from one block-cyclic layout to another.

Block-cyclic redistribution consists of *index set computation*, wherein the destination locations for individual data blocks are calculated, and *data communication*, wherein these blocks are exchanged between processors. The framework treats both these operations in a uniform and integrated way. We have developed efficient and distributed algorithms for index set computation that do not require any interprocessor communication. To perform data communication in a conflict-free manner, we have developed *direct, indirect, and hybrid* algorithms. In the direct algorithm, a data block is transferred directly to its destination processor. In an indirect algorithm, data blocks are moved from source to destination processors through intermediate relay processors. The hybrid algorithm is a combination of the direct and indirect algorithms.

Our framework is based on a generalized circulant matrix formalism of the redistribution problem and a general purpose distributed memory model of the parallel machine. Our algorithms sustain excellent performance over a wide range of problem and machine parameters. We have implemented our algorithms using MPI, to allow for easy portability across different HPC platforms. Experimental results on the IBM SP-2 and the Cray T3D show superior performance over previous approaches. When the block size of the cyclic data layout changes by a factor of K, the redistribution can be performed in $O(\log K)$ communication steps. This is true even when K is a prime number. In contrast, previous approaches take $O(K)$ communication steps for redistribution.

Our framework can be used for developing scalable redistribution libraries, for efficiently implementing parallelizing compiler directives, and for developing parallel algorithms for various applications. Redistribution algorithms are especially useful in signal processing applications, where the data access patterns change significantly between computational phases. They are also necessary in linear algebra programs, to perform matrix transpose operations.

Keywords: Communication scheduling, Block-Cyclic redistribution, Index computation, Generalized circulant matrix, Direct, indirect, and hybrid algorithms.

*Work supported by DARPA under contract no. DABT63-95-C-0092 and in part by NSF under grant CCR-9317301. A preliminary version of this paper appears in Proc. SPDP 1996 [13].

†Department of EE-Systems, EEB-200C, University of Southern California, Los Angeles, CA 90089-2562. Ph: +1 (213) 740-4483, Fax: +1 (213) 740-4418, email: {lim, prabhat, prasanna}@halcyon.usc.edu, http://ceng.usc.edu/~prasanna/

1 Introduction

The choice of data distribution strongly influences application performance on coarse-grained distributed memory parallel machines. In these machines, access to local data is much faster than access to data located at any of the other nodes. This is because remote data access incurs expensive interprocessor communication overheads. These overheads can be reduced by choosing a data distribution that enhances data locality. However, in many High Performance Computing (HPC) applications, including signal processing, the data access patterns of the application change during the computation [25]. For data locality to be maintained, it is necessary to redistribute the data at intermediate points of the computation. In such scenarios, it is important to perform data redistribution using efficient algorithms. Otherwise, the interprocessor communication overheads due to redistribution would offset the performance benefits resulting from data locality. [15] gives an example of the benefits of interstage data redistribution in a typical HPC application.

Data distribution and redistribution can be specified at various levels of detail in application programs. If a parallelizing compiler is used, application programmers specify data distribution using high level compiler directives. For example, parallel programs developed in HPF use the `ALIGN`, `DISTRIBUTE`, and `REDISTRIBUTE` directives to specify data distribution and redistribution of arrays [9]. When HPC applications are developed using explicit parallel algorithms, the data distribution and redistribution between the processors are managed by the programmer. Message passing calls (such as calls to MPI [18]) are used to perform interprocessor data communication. In either approach, efficient redistribution algorithms are important. When parallelizing compilers are used, data redistribution schemes that efficiently implement the functionality of the directives are required. In the parallel algorithm design approach, efficient redistribution algorithms would be used to implement collective communication routines, such as `MPI_Alltoall`. Alternatively, these algorithms can be used to develop data redistribution libraries built on top of MPI.

In this paper, we present a framework for developing efficient redistribution algorithms. We focus on the class of *block-cyclic* distributions, which are commonly used to partition an array over multiple processors. For a specified block size x, this distribution first partitions the N array elements into blocks of x consecutive elements each. The blocks are then assigned to P processors in a round-robin fashion. Such a distribution is denoted as $cyclic(x)$. The block-cyclic

data redistribution problem consists of reorganizing an array from one block-cyclic distribution to another, *i.e.*, from $cyclic(x)$ to $cyclic(y)$. The most frequently encountered version of this redistribution problem is the $cyclic(x)$ to $cyclic(Kx)$ redistribution. We denote the $cyclic(x)$ to $cyclic(Kx)$ redistribution among P processors as $\Re_x(K, P)$.

The block-cyclic distribution matches the data access patterns of many HPC applications. For example, in signal processing applications, the block-cyclic distribution is the natural choice for radar and sonar data cubes. Many of the commonly occurring communication patterns, such as the corner turn operation [12], can be then viewed as block-cyclic redistribution operations where the block size changes from x to Kx. Further, frequently used data layouts such as the *block* and *cyclic* distributions are special cases of the block-cyclic distribution. HPF directives support block-cyclic data distribution and data redistribution of arrays. ScaLAPACK, a widely used mathematical software for dense linear algebra computations, also uses a block-cyclic distribution for good load balance and computation efficiency [2]. The framework that we present in this paper enables the design of efficient algorithms for various kinds of redistribution problems. In this paper, we consider the $\Re_x(K, P)$ problem. Although we focus on one dimensional arrays, we have also developed techniques for multidimensional arrays [14]. We are currently investigating redistribution from $cyclic(x)$ on a set of source processors to $cyclic(Kx)$ on a distinct set of destination processors [10]. We are also developing algorithms for the general $cyclic(x)$ to $cyclic(y)$ redistribution problem [16].

The block-cyclic redistribution problem has been well studied in the literature. The important case of $\Re_x(K, P)$ has attracted the most attention. The previous research efforts attempt to reduce either of the two main overheads incurred by redistribution algorithms: index computation overheads and interprocessor communication overheads. Index computation overheads are incurred when each processor computes indices of array elements that are to be communicated with the other processors, as well as the destination processors of such array elements. Communication overheads are incurred when the processors exchange array elements. These include software start-up overheads for invocation of the send and receive system calls, transmission costs for sending data over the interconnection network, and overheads due to node contention.

Table 1 compares the performance of some previous research efforts. The direct approach refers to the scheme presented in [23]. Here, an algorithm which takes K communication steps is developed for $\Re_x(K, P)$. In [7], a multiphase approach is derived for the same problem. Here,

K is factored such that $K = k_1 \cdot k_2 \cdots k_n$. The redistribution is then performed in n phases; the i^{th} phase takes k_i steps. A two-phase variant of this scheme is also presented in [7]. When K is a prime number, both the multiphase and two-phase approaches take K communication steps. The indirect approach refers to our new algorithms presented in Section 3. Our indirect algorithms perform $\Re_x(K, P)$ in atmost $\lceil \log K \rceil + 2$ steps [1]. The algorithm is applicable for any value of K, including prime numbers. For example, when K=31 and P=64, the redistribution can be performed in 6 communication steps. In contrast, previous approaches in [23] and [7] take 31 communication steps for this problem. A two-phase strategy can be used for the $cyclic(x)$ to $cyclic(y)$ redistribution problem. Consider the case when x and y are relatively prime, *i.e.* $LCM(x, y) = x.y$. In the first phase, we perform redistribution from $cyclic(x)$ to $cyclic(x.y)$. In the second phase, we perform redistribution from $cyclic(x.y)$ to $cyclic(y)$. Algorithms for $\Re_x(K, P)$ can be used for each of these phases. Previous approaches take a total of $x + y$ communication steps for the 2 phases. Our indirect approach performs this redistribution in $\lceil \log x \rceil + \lceil \log y \rceil + 4$ steps.

An important contribution of our research is a uniform framework for the design and evaluation of these different redistribution algorithms. In this framework, the direct and indirect communication algorithms are special cases of the general class of hybrid algorithms. The redistribution problem is represented as a generalized circulant matrix. This representation allows for the choice of a suitable direct, indirect or hybrid algorithm based on problem parameters such as array size, and block size, as well as parallel machine parameters. Each algorithm consists of a local data reorganization phase and an interprocessor communication phase. The local reorganization phase permutes data blocks within the memory of each processor so that the communication phase can be performed efficiently. Apart from reducing communication overheads, our approach also minimizes index computation overheads by utilizing the periodic characteristics of the block cyclic distribution. Index computation is performed by the processors in a distributed way, and no interprocessor communication is needed.

Implementations of our algorithms on the IBM SP-2 and Cray T3D show superior performance over previous approaches. Our algorithms have been implemented using MPI and can be easily ported to various HPC platforms. Our techniques can be used for developing scalable redistribution libraries, for optimizing ScaLAPACK [2], for implementing the `REDISTRIBUTE`

[1] All logarithms in this paper are to base 2.

Cyclic Redistribution Problem		Direct Approach	Multi-phase Approach	2-phase Approach	Our Indirect Approach
cyclic(x)* → *cyclic(Kx)* *or* *cyclic(Kx)* → *cyclic(x)* *on* *P processors	For a composite $K = \prod_{i=1}^{n} k_i$	K	$\sum_{i=1}^{n} k_i$	$k_1 + k_2$	$\lceil \log K \rceil + 2$
	For a prime K		K	K	
cyclic(x)* → *cyclic(y)* *on* *P processors	For pairwise relatively prime numbers x, y, and P	$x+y$	$x+y$	$x+y$	$\lceil \log x \rceil + \lceil \log y \rceil + 4$

Table 1: Number of communication steps required by various approaches to perform block-cyclic redistribution.

directive in HPF [9], and for developing parallel algorithms for various HPC applications.

The rest of the paper is organized as follows. Section 2 discusses some previous research efforts on block-cyclic redistribution. In Section 3, we present our approach to the redistribution problem and show illustrative examples. Section 4 presents our algorithms and gives proofs of their correctness. In Section 5, we report experimental results on the IBM SP-2 and the Cray T3D. Section 6 concludes the paper.

2 Related Work

Several research efforts have been focused on the problem of redistributing an N element array over P processors from $cyclic(x)$ to $cyclic(Kx)$, *i.e.*, $\Re_x(K, P)$. Many of these research efforts have been targeted towards efficiently implementing high level compiler directives, such as the `REDISTRIBUTE` directive of HPF.

Sadayappan *et. al.* [7] and Walker *et. al.* [23] have proposed algorithms which reduce communication overheads. In [23], a K step schedule is given for $\Re_x(K, P)$. At each step, processors exchange data in a contention free manner: each processor sends data to exactly one processor and receives data from exactly one processor. A similar communication schedule is presented in [7]. Although the communication schedule presented in [23] is based on modular arithmetic and that in [7] is based on tensor products, the resulting communication schedules are similar. Both these communication schedules are special cases of the family of "direct schedules" that can be derived from our framework.

Walker *et. al.* [23] implemented cyclic redistribution algorithms using a *synchronized* and an *unsynchronized* scheme. In the synchronized scheme, each of the K communication steps posts a receive, sends a message, and then waits for the completion of the receive. In the unsynchronized scheme, all the K receives are posted and the K corresponding sends are then performed. The unsynchronized scheme requires temporary buffers of size $\frac{N}{P}$ per processor (as large as the local portion of the array). The synchronized scheme incurs excessive synchronization overheads and is vulnerable to communication hot spots. Implementations on the IBM SP-1 and on the Intel Paragon show that the unsynchronized scheme executes faster than the synchronized scheme. To improve the performance of the synchronized scheme while retaining its low buffer size requirements, random and optimal scheduling algorithms were developed in [23]. However, the optimal schedule in [23] does not minimize the number of communication steps. The schedule only ensures that each communication step is conflict-free. This schedule consists of K steps. Their experimental results show that the performance of the synchronized "optimal" algorithm is comparable to that of the unsynchronized scheme, for small N and K.

Sadayappan *et. al.* [6, 7, 8] presented a multiphase approach for $\Re_x(K, P)$. Here, array redistribution is performed as a sequence of redistributions so that the total communication cost of the sequence is lower than the cost of the direct redistribution. In their scheme, each of the intermediate distributions is a cyclic distribution. The communication schedule for each phase is determined using a tensor product formalism. Using this formalism, each processor computes the set of source processors, the set of destination processors, the local indices of elements to be sent, and the local indices of the elements to be received. The multiphase approach in [7] first factors K into $k_1 \cdot k_2 \cdots k_n$. The redistribution is then performed in n phases. The i^{th} phase consists of k_i communication steps. All the N array elements are moved during each phase. To determine the sequence of intermediate distributions, heuristic optimization algorithms are developed in [7]. When K is prime, this approach takes K communication steps.

Indirect algorithms for collective communication have also been addressed in [1]. Here, interprocessor data communication is performed in a "combine and forward" manner. A library of efficient communication schedules, called CCL, is developed. The library supports operations such as broadcast, scatter, gather, and shift. Communication schedules are derived from a circulant graph representation of strided communication patterns. However, this research did not consider block-cyclic redistribution.

Several other research efforts have focused on reducing index computation overheads. However, communication scheduling is not addressed in these efforts.

In [21, 22], Choudhary *et. al.* present efficient index computation algorithms for the special case when $P \bmod K = 0$. They also consider the redistribution from $cyclic(x)$ to $cyclic(y)$, for general 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 to be used in practice. [22] proposes *gcd* and *lcm* methods. These are two phase algorithms where $cyclic(x)$ is first redistributed to $cyclic(m)$, followed by the redistribution of $cyclic(m)$ to $cyclic(y)$. Here, m can be *gcd* or *lcm* of x and y. It is shown in [22] that multidimensional arrays can be redistributed by applying these algorithms to each dimension of the array separately.

In [20], Banerjee *et. al.* use a line segment formalism to represent a $cyclic(x)$ distribution. The array elements that map to a processor are represented as a set of strided line segments. For every pair of processors, the array elements whose indices are in the intersection of the respective line families are exchanged. Their techniques can compute index sets for arbitrary source and target processor sets and multidimensional arrays.

In [5], Ni *et. al.* assign 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, do not need to be moved. However, index computation becomes complicated. This approach is only applicable when the source and target processor sets for the redistribution are the same.

3 Our Approach to Redistribution

We shall now examine the block-cyclic redistribution problem in detail. As mentioned in Section 1, we focus on the case where the block size changes by a factor of K, *i.e.*, $\Re_x(K, P)$. Section 3.1 gives an example of such a redistribution problem. Section 3.2 discusses the costs involved in performing block-cyclic redistribution. Section 3.3 explains our table-based framework for the design of efficient redistribution algorithms. Section 3.4 explains the formalism used for developing our communication schedules.

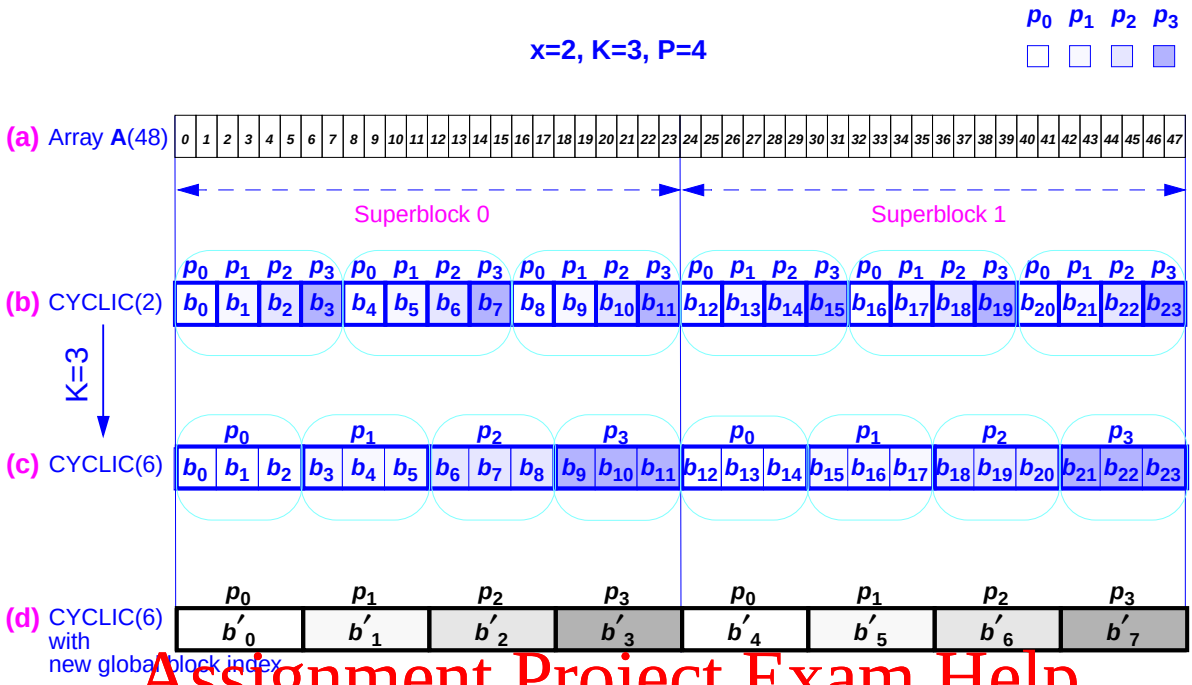


Figure 1: Redistribution from *cyclic*(2) to *cyclic*(6) on 4 processors, array point of view

3.1 The Block-Cyclic Redistribution Problem

The block-cyclic distribution of an array can be defined as follows [23]: given P processors, an array of size N, and a block size x, the distribution first partitions the array elements into contiguous blocks of x items each. b_i is the i^{th} block, $0 \leq i < \frac{N}{x}$ [2]. The blocks are then assigned to processors in a round robin fashion so that b_i is assigned to processor $(i \bmod P)$. We denote a block-cyclic distribution of block size x as *cyclic*(x). We now consider the *cyclic*(x) to *cyclic*(Kx) redistribution problem, $\Re_x(K, P)$. Since this redistribution problem becomes the all-to-all communication problem when $K \geq P$, we consider the case of $K < P$.

Figure 1 shows the example of $\Re_2(3, 4)$. The array **A** which has $N = 48$ elements is shown in Figure 1(a). Figure 1(b) shows the initial distribution, *cyclic*(2). Here, b_i is of size 2 elements, and has a global block index i. The blocks are assigned to $P = 4$ processors in a round robin fashion. If the block size is increased by a factor of $K(= 3)$, *i.e.*, the new block size becomes 6, each set of three consecutive blocks becomes a new block. This increase of block size is shown in Figure 1(c). Figure 1(d) shows the *cyclic*(6) distribution. The blocks have a size of six elements each, and have new global indices.

[2] For simplicity, we assume that x divides N.

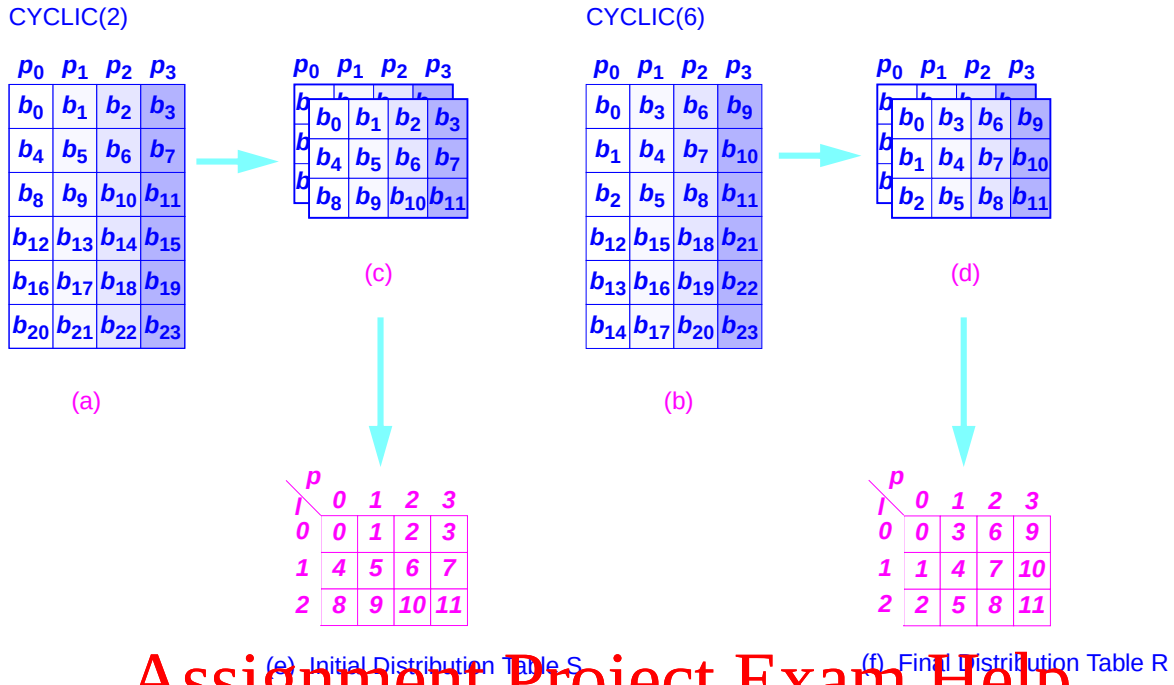


Figure 2: Redistribution from *cyclic*(2) to *cyclic*(6) on 4 processors, processor point of view

From Figures 1(b) and 1(c), a periodicity can be seen in the block movement pattern of $\Re_x(K, P)$. Every set of PK blocks starting at index cPK, $(c = 0, \ldots, \frac{N}{PK} - 1)$, has the same communication pattern. Such a set of PK data blocks is referred to as a *superblock* in [23] and as a *course* in [7]. In this example, each superblock consists of 12 blocks. Figures 1(b) and 1(c) show the two superblocks of the initial and final distributions. To redistribute from *cyclic*(2) to *cyclic*(6), blocks b_0, b_1, b_2 are moved to p_0, blocks b_3, b_4, b_5 are moved to p_1, blocks b_6, b_7, b_8 are moved to p_2, and blocks b_9, b_{10}, b_{11} are moved to p_3. In the next superblock, blocks b_{12} to b_{23} are moved in the same fashion.

Figure 1 shows the array point of view of the redistribution problem. The elements of the array are shown along a single horizontal axis. The processor numbers are then marked above each block. A different representation, where the periodicity of the redistribution pattern and the structure of the superblock can be better observed, is shown in Figure 2. A two-dimensional table, consisting of P columns is used, as shown in Figures 2(a) and Figure 2(b). Column i contains blocks assigned to processor p_i. This representation is therefore called the processor point of view. The table can be further "folded" at each superblock as shown in Figures 2(c) and 2(d). Each "layer" corresponds to a superblock, and consists of K rows and P columns.

The data communication pattern is identical for each layer.

Across different layers, blocks which are located at the same relative position within a superblock are moved in the same way during redistribution. These blocks can therefore be transferred in a single communication step. The MPI derived data type provides an efficient mechanism to handle these blocks as a single unit of data. Without loss of generality, we will consider only the first superblock (the first layer) to illustrate our algorithms in the following. We refer to the tables representing the first superblocks of the initial and the final distribution as the *initial distribution table* and the *final distribution table*, respectively. These are shown by **S** and **R** in Figures 2(e) and Figure 2(f), respectively. It should be noted that the global block indices are in row (column) major order in the initial (final) distribution table. In other words, the entries of **S** and **R** are ordered in row and column major order, respectively. Thus, the cyclic redistribution problem essentially involves reorganizing blocks within each superblock from an initial distribution table **S** to a final distribution table **R**.

So far, we have discussed the $cyclic(x)$ to $cyclic(Kx)$ redistribution problem. A dual relationship exists between the $cyclic(x)$ to $cyclic(Kx)$ and the $cyclic(Kx)$ to $cyclic(x)$ redistributions. In the $cyclic(Kx)$ to $cyclic(x)$ redistribution, the entries of **S** are in column major order, while the entries of **R** are in row major order. Hence, our algorithms for $\Re_x(K, P)$ can be used for $cyclic(Kx)$ to $cyclic(x)$ redistribution, with a few modifications.

3.2 The Cost of Redistribution

As mentioned in Section 1, the data redistribution process incurs two kinds of overheads: index computation overhead and interprocessor communication overhead. A clear understanding of the nature of these overheads will be useful in reducing them.

For each array element, an index computation overhead is incurred in calculating its destination processor and the location of the element within that processor. In HPF, each array element can be referenced by a global index and a local index. The global index is the index of an array element from the array point of view, in Figure 1(a). From the global index, we can calculate the processor number as well as the local memory location, *i.e.*, the local index. These refer to the column and row indices in the two-dimensional table of Figure 2(a). Similarly, the position in the table of Figure 2(b) can be calculated from the global block index, once K is known. However, frequent computation of this information from the global index leads to significant index computation overhead. To reduce the index computation overhead, an intelligent scheme that

utilizes the regular structure of the block-cyclic array distribution will be discussed in Section 4. Observe that explicit index computation must be performed for only one superblock. Across superblocks, the corresponding elements are distributed in an identical fashion. For reasons of efficient data transmission, a redistribution algorithm should perform packing and unpacking of messages. The array elements which have the same destination processor are packed into a single message and sent together. The received message must then be unpacked and elements must be placed in the correct locations in the processor's local memory, according to the desired final distribution.

Interprocessor communication overhead is incurred when data is exchanged between processors of a coarse-grained parallel computer. The communication overheads can be represented using an analytical model of typical distributed memory machines, the General purpose Distributed Memory (GDM) model [24]. Similar models are reported in the literature [1, 3, 4]. The GDM model represents the communication time of a message passing operation using two parameters: the *start-up time* T_d and the *unit data transmission time* τ_d.

The start-up time is incurred once for each communication event. It is independent of the message size to be communicated. This start-up time consists of the transfer request and acknowledgment latencies, context switch latency, and latencies for initializing the message header. The unit data transmission time is the cost of transferring a message of unit length over the network. The total transmission time for a message is proportional to the message size. Thus, the total communication time for sending a message of size m units from one processor to another is modeled as $T_d + m\tau_d$. A permutation of the data elements among the processors, in which each processor has m units of data for another processor, can be performed concurrently in $T_d + m\tau_d$ time. The model assumes that the network is not heavily loaded, and the unit data transmission time is independent of the number of hops the message traverses.

The model does not allow node contention at the source and destination of messages. Thus, a processor can receive a message from only one other processor in every communication step. Similarly, a processor can send a message to only one other processor in each communication step. Contention at switches within the interconnection network is not considered. The interconnection network is modeled as a completely connected graph. Hence, the model does not include parameters for representing network topology. This assumption is realistic because of architectural features such as virtual channels and cut-through routing in state-of-the-art inter-

connection networks. Also, the component of the communication cost that is topology dependent is insignificant compared to the large software overheads included in message passing time.

The model has been used to represent interprocessor communication costs of many general purpose HPC systems such as the IBM SP-2 and the Cray T3D. In the current generation of HPC platforms, the network (hardware) latency is small compared to software overheads in message passing. T_d is in the range of *micro-seconds* and the switch latencies are in the range of *nano-seconds*. Thus, the ratio of T_d to τ_d is in the range of several hundreds to a few thousands. Consequently, the start-up cost is the dominant overhead for small messages. As the message size increases, the data transmission time also becomes significant.

3.3 A Table-Based Framework for Redistribution

The pattern of redistribution of an array from one block-cyclic distribution to another can be represented using the 2-dimensional distribution tables **S** and **R**, introduced in Section 3.1. Each table has the dimensions of a superblock, *i.e.* K rows and P columns for the redistribution problem $\Re_x(K, P)$. The j^{th} column of the table contains the global indices of the blocks that are owned by the processor p_j, $0 \leq j < P$. Since the redistribution pattern is identical for each superblock, it is sufficient to consider the redistribution problem with respect to one superblock.

Each block in the table (**S** or **R**) is identified by a global block index. Note that, if the redistribution parameters (K and P) are given, then each block's location in **S** and **R** can be determined. As a result of redistribution, each block moves from its initial location in **S** to its final location in **R**. Thus, the processor ownership and the local memory location of each block are changed by redistribution. Redistribution can thus be conceptually viewed as a table conversion process from **S** to **R**. This conversion process can be decomposed into independent column and row transformations. In a column transformation, blocks are permuted within a column of the table. This is therefore a local operation within a processor's memory. In a row transformation, blocks within a row are permuted. This operation therefore leads to a change in processor ownership of blocks, and requires interprocessor communication.

The overall communication pattern of the redistribution can be defined by the destination processor table, shown in Figure 3(a). The j^{th} column of the destination processor table **P** contains all the destination processor indices of blocks in the processor p_j, $0 \leq j < P$. **P** can be obtained by replacing each entry in **S** with the destination processor index of the corresponding block. Figure 3(b) shows the set of communication events in $\Re_x(K, P)$. It can be seen that every

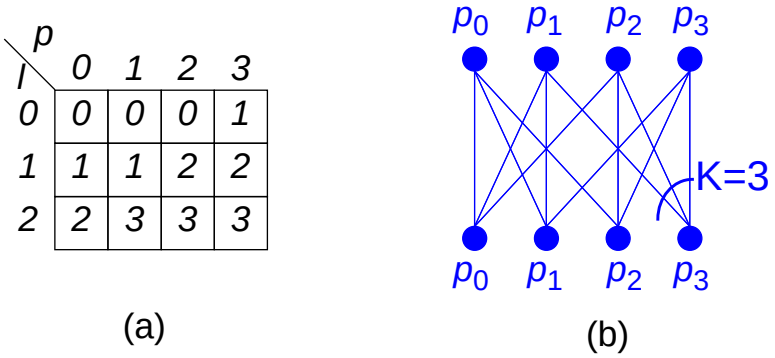


Figure 3: (a) Destination processor table (b) Communication pattern for $\Re_x(3,4)$

processor communicates with K processors (including itself). Since a processor can send data to only one other processor at a time, it is evident that this communication pattern must be broken down into a set of communication events. This constitutes communication scheduling.

The destination processor table (*dpt*) defines the required set of communication events in redistribution. One way to perform the row transformations is to regard the i^{th} row of $\mathbf{P}$ as the set of events to take place in the i^{th} communication step. Thus, the $(i,j)^{th}$ entry of $\mathbf{P}$ will be the destination processor of the i^{th} communication event at the processor p_j. In general, this can lead to node contention, since multiple processors can attempt to send messages to a single processor in the same communication step. However, if every row of $\mathbf{P}$ is a permutation of the processor indices $\{0, 1, \ldots, P-1\}$, then every processor has a distinct destination processor in each communication step. Thus, node contention can be avoided. This observation is the motivation for the column transformations.

Figure 4 shows the key steps in our table conversion approach to redistribution. The lower portion of the figure shows the desired overall result: the initial distribution table $\mathbf{S}$ is to be converted to the final distribution table $\mathbf{R}$ by a communication pattern specified by the *dpt* $\mathbf{P}$. Our approach consists of first transforming $\mathbf{S}$ to an intermediate form $\mathbf{S}'$ by applying only column transformations. Remember that the $(i,j)^{th}$ entry of $\mathbf{P}$ gives the destination processor number for the $(i,j)^{th}$ entry of $\mathbf{S}$. To maintain this correspondence between $\mathbf{S}$ and $\mathbf{P}$, the same set of column transformations are also applied to $\mathbf{P}$. This results in the table $\mathbf{P_s}$.

$\mathbf{S}'$ is then converted to $\mathbf{R}'$, by applying row transformations based on the communication pattern specified by $\mathbf{P_s}$. $\mathbf{R}'$ is then converted to $\mathbf{R}$ by applying an additional set of column transformations. The key idea of the framework is to choose a $\mathbf{P_s}$ such that the required row

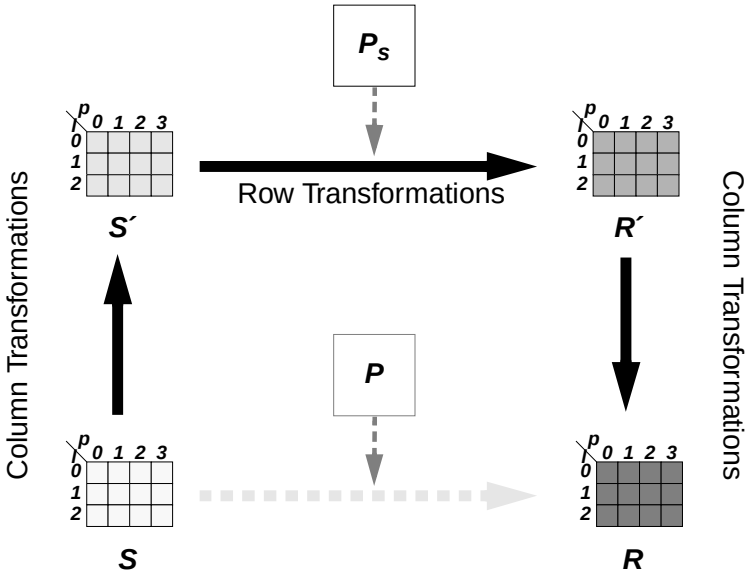


Figure 4: Overview of our redistribution steps

transformations (communication events) can be performed efficiently.

For example, assume that the local data reorganization (column transformation) creates a $\mathbf{P_s}$ wherein the entries along each diagonal are the same. The communication events (row transformations) will then be required to vertically align these diagonals, *i.e.* elements which were present along a diagonal of $\mathbf{P_s}$ are to be moved into a single column. It will be shown in Section 3.4 that this can be achieved in a logarithmic number of steps by circularly shifting rows of $\mathbf{P_s}$ in a strided way. Since $\mathbf{P_s}$ has K rows, the communication can be performed in $O(\log K)$ steps.

In the following section, we shall define the notion of a generalized circulant matrix. Once $\mathbf{P_s}$ is in generalized circulant matrix form, a class of efficient and contention free communication schedules for $\Re_x(K, P)$ can be easily derived.

3.4 Communication Scheduling for Efficient Redistribution

As discussed in the previous section, our approach to redistribution consists of local rearrangement of data within each processor as well as interprocessor communication. Our framework for communication scheduling utilizes the notion of a generalized circulant matrix. We now explain the characteristics of such a matrix and how it is used to develop communication schedules.

Definition: An $m \times n$ matrix ($m \leq n$) is a *circulant matrix* if row k = row 0 circularly right shifted k times, where $0 \leq k < m$.

Definition: Given an $M \times N$ matrix ($M \leq N$), suppose the matrix can be partitioned into submatrices of size $s \times t$, where $M = m \cdot s$ and $N = n \cdot t$, for some $s, t > 0$ and $m \leq n$. The matrix is a *generalized circulant matrix* if row k of submatrices is the same as row 0 of submatrices circularly right shifted k times ($0 \leq k < m$), and each submatrix is either a circulant matrix or a generalized circulant matrix.

The generalized circulant matrix formalism and the *dpt* provide a systematic and flexible framework for developing contention free communication schedules. In Figure 4, the initial column transformations convert $\mathbf{P}$ to a generalized circulant matrix form, $\mathbf{P_s}$. Using this form of $\mathbf{P_s}$, we have derived three kinds of schedules for $\Re_x(K, P)$. These are the direct, indirect and hybrid schedules.

An example of a $K \times P$ generalized circulant matrix $\mathbf{C}$ is shown in Eq (1). The matrix $\mathbf{C}$ is partitioned into submatrices C_j of size $G \times G$. Here $G = gcd(K, P)$, $K = K'G$, $P = P'G$ and $K \leq P$. (The choice of parameters in this example will become clear in Section 4.) Thus, the matrix $\mathbf{C}$ has K' submatrix rows and P' submatrix columns. Each submatrix row is the circularly right shifted version of the first submatrix row $[\ C_0 \quad C_1 \quad \cdots \quad C_{P'-1}\ ]$. Therefore, the matrix $\mathbf{C}$ can have at most P' distinct submatrices. Each submatrix C_j, $0 \leq j < P'$ itself is a $G \times G$ circulant matrix.

$$\mathbf{C} = \begin{bmatrix} C_0 & C_1 & \cdots & C_{P'-1} \\ C_{P'-1} & C_0 & \cdots & C_{P'-2} \\ C_{P'-2} & C_{P'-1} & \cdots & C_{P'-3} \\ \vdots & \vdots & & \vdots \\ C_{(P'-K'+1) mod P'} & C_{(P'-K'+2) mod P'} & \cdots & C_{P'-K'} \end{bmatrix} \qquad C_j = \begin{bmatrix} c_{j,0} & c_{j,1} & \cdots & c_{j,G-1} \\ c_{j,G-1} & c_{j,0} & \cdots & c_{j,G-2} \\ \vdots & \vdots & & \vdots \\ c_{j,1} & c_{j,2} & \cdots & c_{j,0} \end{bmatrix} \quad (1)$$

We make the following two observations about generalized circulant matrices: (i) Using the above notation, all the K' submatrices (C_j's) along each diagonal are identical, (ii) If all the P elements in a certain row i are distinct, then every row is a distinct permutation of the elements in row i. The graphical representation of Eq (1) in Figure 5 clearly shows these properties. The submatrices along each diagonal of Figure 5(a) are identical, depicted by their uniform shading. Within each submatrix, the elements of each diagonal are also identical, as shown in Figure

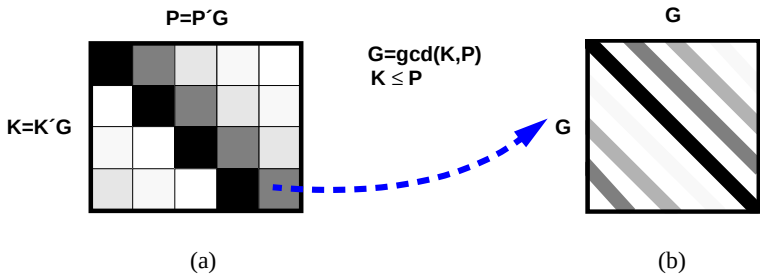


Figure 5: $K \times P$ generalized circulant matrix

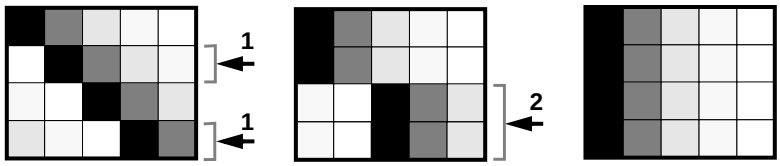


Figure 6: Cyclic shift operations

5(b). The diagonals in the generalized circulant matrix can be aligned vertically by circular shift operations. Figure 6 shows the first two steps of such an alignment. In the first step, the second and the fourth rows of submatrices are circularly left shifted. In the second step, the third and the fourth rows of submatrices are circularly left shifted with the stride increased by a factor of two. At this point, each column of submatrices has identical elements. The circular shift operations can be applied recursively to each submatrix. As a result of these operations, each of the diagonals within the submatrices will be vertically aligned.

Consider the example of $\Re_x(6, 9)$. Figure 7 shows the column transformations which convert the *dpt* $\mathbf{P}$ to its generalized circulant matrix form $\mathbf{P_s}$. Figure 7(a) shows the initial distribution table, while Figure 7(b) shows the corresponding *dpt*, $\mathbf{P}$. In this example, $G = 3, K' = 2$, and $P' = 3$. In Figure 7(a) and 7(b), $\mathbf{S}$ and $\mathbf{P}$ can be partitioned into submatrices of size 2×9. Next, the rows in these submatrices are shuffled, as shown in Figure 7(c) and 7(d). Now we can partition the shuffled $\mathbf{S_1}$ and the corresponding *dpt* into submatrices of size 3×3. The diagonalization of these 3×3 submatrices and diagonalization of elements in each submatrix are shown in Figures 7(e) and 7(f). This results in the generalized circulant matrix $\mathbf{P_s}$. It consists of 2 rows and 3 columns of 3×3 submatrices. Observe that the elements within each column of $\mathbf{P}$ (or $\mathbf{S}$) are the same as the elements in the corresponding column of $\mathbf{P_s}$ (or $\mathbf{S'}$). Hence, no interprocessor communication is incurred during this column transformation phase.

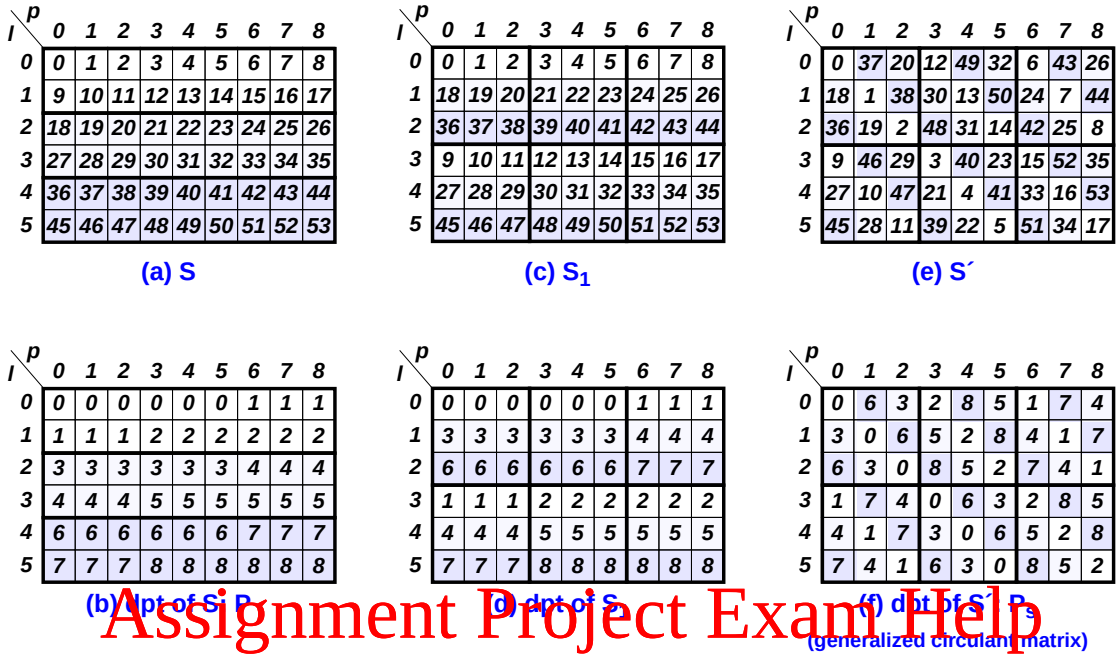


Figure 7: Steps of column transformation

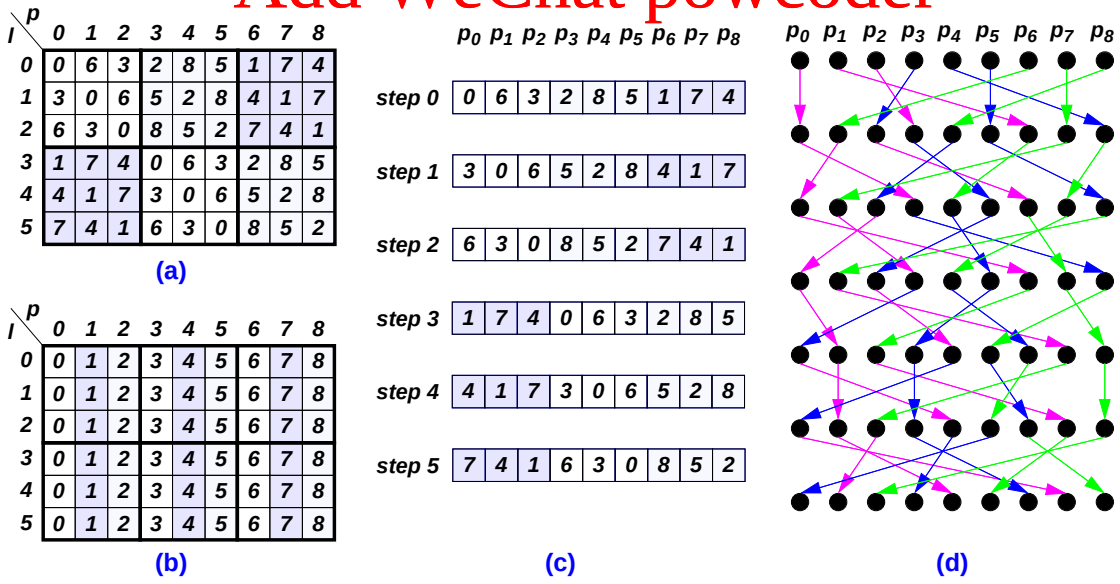


Figure 8: Communication steps and patterns in the direct schedule

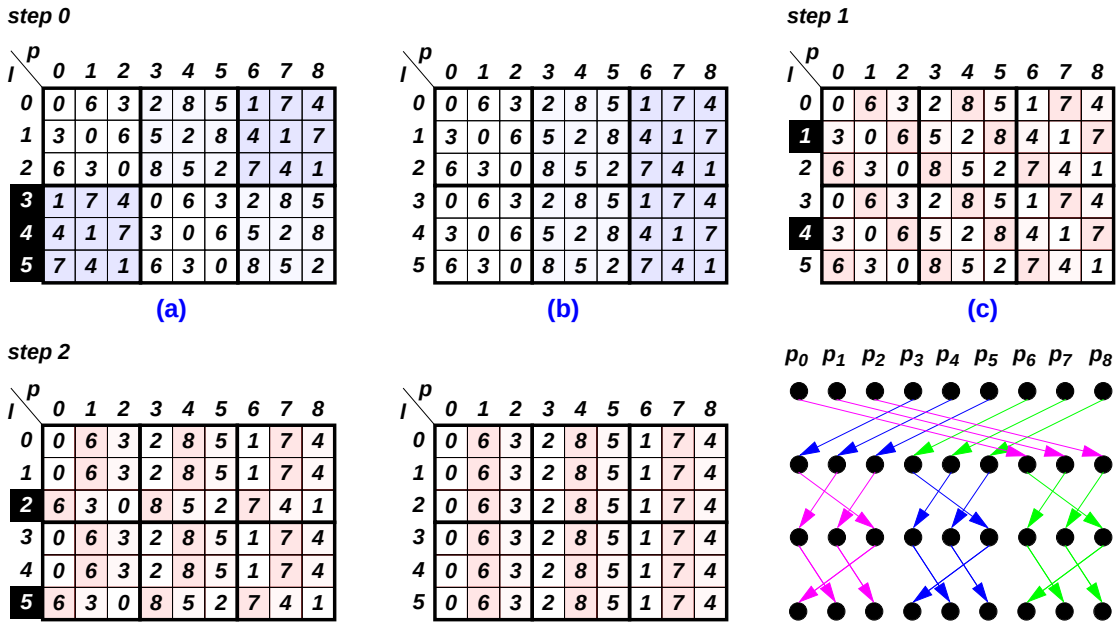


Figure 9: Communication steps and patterns in the indirect schedule. In (a), submatrices of size 3×3 are moved as a unit. In (c) and (d), elements within a submatrix are circularly shifted.

Figures 8, 9, and 10 illustrate the direct, indirect, and hybrid schedules for the example of $\Re_x(6, 9)$. Note that $\mathbf{P_s}$ is in the generalized circulant matrix form, as shown in Figures 8(a) and 9(a). In the direct schedule, the i^{th} communication event permutes the i^{th} row of $\mathbf{P_s}$, as shown in Figures 8(c) and Figure 8(d). For example, the 0^{th} row of $\mathbf{P_s}$ is (0, 6, 3, 2, 8, 5, 1, 7, 4). Thus, processors $(p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8)$ send the blocks which correspond to the 0^{th} row to processors (p_0, p_6, p_3, p_2, p_8, p_5, p_1, p_7, p_4). After 6 steps, Figure 8(b) is obtained. In each step of the direct schedule, each processor sends only one message and also receives only one message, as shown in Figure 8(d). This is therefore a communication schedule wherein each step is contention free. Figure 8(c) and Figure 8(d) show only one of the many possible direct schedules for this example. Observe that the K rows of $\mathbf{P_s}$ specify K communication steps. These steps can be performed in any order. Hence, there are $K!$ possible direct schedules that can be derived from $\mathbf{P_s}$.

In the indirect schedule, each diagonal of submatrices is first vertically aligned. In Fig-

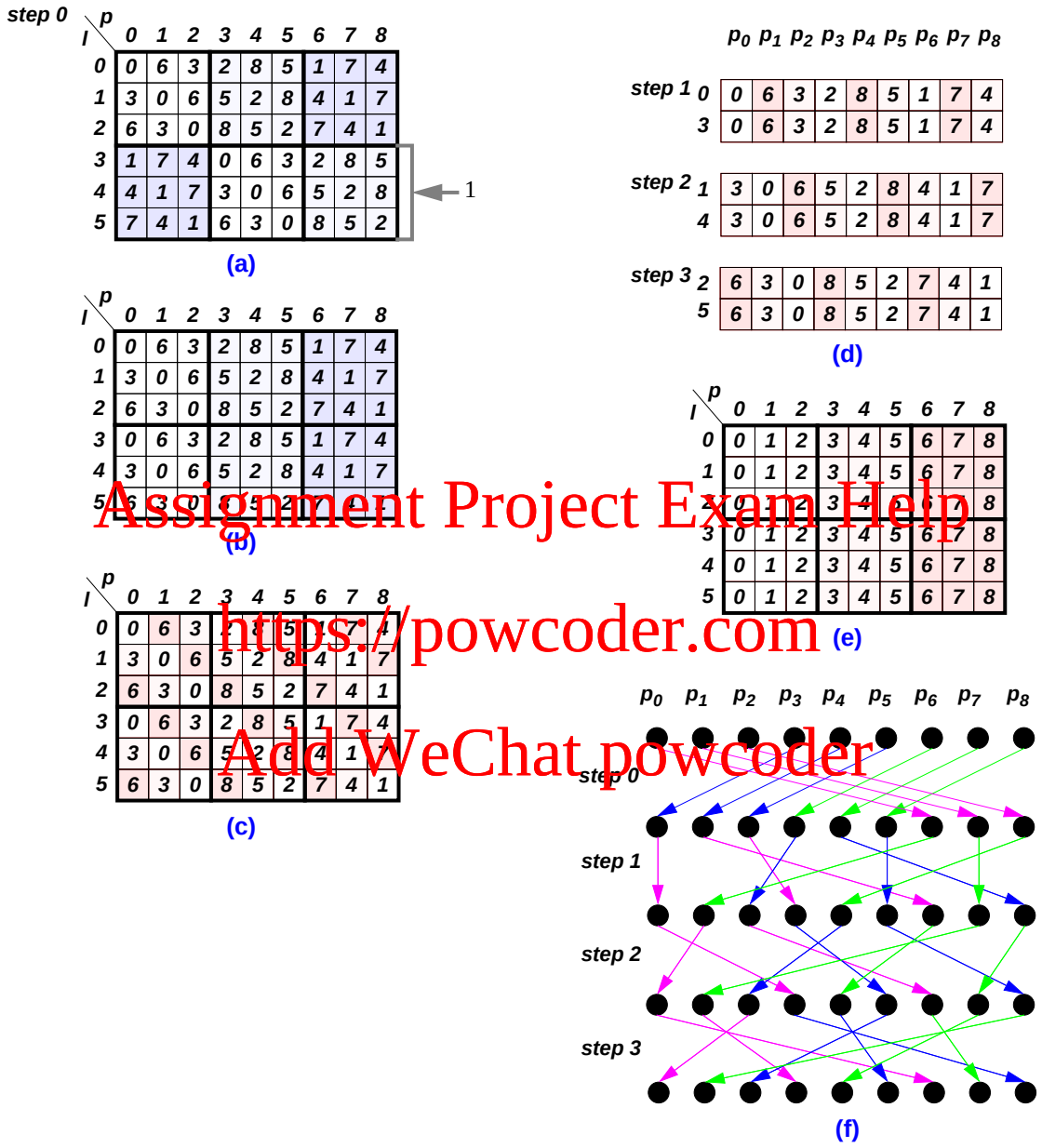


Figure 10: Communication steps and patterns in the hybrid schedule with $d = 1$. The indirect schedule is used for step 0 of the schedule (a). The direct schedule is used for steps 1, 2, and 3 (d)

ure 9(a), there are two rows of submatrices as shown by the thick lines. Rows 0, 1, and 2 are the first row of submatrices, while rows 3, 4, and 5 are the second row of submatrices. These two rows are aligned in $\lceil \log 2 \rceil$ steps, by circularly shifting the 3^{rd}, 4^{th}, and 5^{th} rows to the left, as shown in Figures 9(a) and (b). Next, the diagonal elements within each submatrix of size 3×3 are aligned. Figures 9 (c) and (d) show the $\lceil \log 3 \rceil$ circular shift operations performed in each group of three processors. Thus, after $\lceil \log 2 \rceil + \lceil \log 3 \rceil = 3$ steps, every element is vertically aligned, as shown in Figure 9(e). Figure 9(f) shows the communication pattern of the circular shift operations.

For large sized messages, the transmission cost constitutes most of the total communication time. In such situations, the direct schedule is the most efficient. In the direct communication schedule, messages are sent directly from the source processor to the destination processor. For small sized messages, the start-up costs dominate the transmission costs in the total communication time. This situation is often encountered, for example, when redistributing a small sized array or redistributing an array among a large set of processors. The excessive start-up costs can be reduced by sending messages to their destination processors through intermediate "relay" processors. At these intermediate processors, messages which are destined to the same processor, are combined and forwarded. Thus, the indirect transfer of messages refers to reorganizing the initial communication pattern. The indirect schedule reduces the start-up costs; however, it increases the total message volume to be transferred. For large sized messages, the communication time of the indirect schedule increases faster than the direct schedule due to the extra data transmission.

A hybrid schedule with degree of indirection d, performs the first d steps of the indirect schedule. Each data block of size 2^d that has the same destination is then transferred using a direct schedule. This schedule thus balances the start-up and data transmission costs. The desired degree of indirection d can be computed based on problem specific parameters (P, K, N) and machine specific parameters (T_d and τ_d) to minimize the overall redistribution cost. Note that the indirect schedule is a hybrid schedule with $d = \lceil \log K' \rceil + \lceil \log G \rceil$ and the direct schedule is a hybrid schedule with $d = 0$. Figure 10 shows the steps of the hybrid algorithm for $\Re_x(6, 9)$. Here, a value of $d = 1$ is selected. In step 0 (Figure 10(a)), the lower row of 3×3 submatrices is shifted to the left by one column. This is identical to the first step of the indirect algorithm. At this point, both rows of submatrices are vertically aligned, as shown in Figure 10(b) and

Figure 10(c). Next, blocks that have the same destination are combined and moved together using the direct schedule. Observe in Figure 10(c) that rows 0 and 3 are identical. Similarly, rows 1 and 4 are identical, and so are rows 2 and 5. Figure 10(d) shows how these rows are combined. In general, sets of 2^d rows will be identical to one another, and can therefore be combined. Three steps of the direct schedule are then performed with this larger block size. In each step, the communication pattern is a permutation of the P processors. The final distribution is shown in Figure 10(e). The communication patterns for all the steps of the hybrid schedule are shown in Figure 10(f).

The next section gives pseudocode for these algorithms and analyzes their complexity.

4 Redistribution Algorithms and Their Complexity

As already discussed, we view the redistribution problem $\Re_x(K, P)$ as a process of converting a table from row major order $\mathbf{S}$ to column major order $\mathbf{R}$. We now present results that establish the correctness and complexity of this operation.

The table conversion process consists of an initial set of column transformations (local reorganization) followed by a set of row transformations (interprocessor communication events). Theorem 1 shows that the initial *dpt* $\mathbf{P}$ can be converted into a suitable generalized circulant matrix form $\mathbf{P_s}$ by column transformations alone. Note that the column transformations convert $\mathbf{S}$ into $\mathbf{S'}$ in the same way as they convert $\mathbf{P}$ into $\mathbf{P_s}$. Theorem 2 gives formulae for efficiently computing the entries of $\mathbf{P_s}$. This represents the index set computation. Theorem 3 analyzes the complexity of the direct, indirect, and hybrid communication schedules for performing the row transformations.

In the following, i/K denotes the integer quotient of the division. Also, $G = gcd(K, P)$, $K = K'G$, $P = P'G$, $i_1 = i/G$, $i_2 = i \bmod G$, $j_1 = j/G$, and $j_2 = j \bmod G$.

Theorem 1 *The initial dpt* $\mathbf{P}$ *of* $\Re_x(K, P)$ *can be reorganized via column transformations to obtain a* $\mathbf{P_s}$ *such that* (i) $\mathbf{P_s}$ *is a generalized circulant matrix, and* (ii) *Every row of* $\mathbf{P_s}$ *is a distinct permutation of* $\{0, 1, \ldots, P-1\}$.

Proof: Consider the initial distribution table $\mathbf{S}$. Each element of $\mathbf{S}$, $\mathbf{S}(i, j)$ has the value

$$\mathbf{S}(i, j) = iP + j, \tag{2}$$

since the data blocks are in row major order. The corresponding elements of the *dpt* are obtained from $\mathbf{S}$ by $\mathbf{S}(i,j)/K$. In the following, we reorganize the elements in each column of $\mathbf{S}$ in two stages, obtaining $\mathbf{S_1}$ and then $\mathbf{S'}$ (Figure 7). We then show that $\mathbf{P_s}$, obtained by $\mathbf{P_s}(i,j) = \mathbf{S'}(i,j)/K$, is in the generalized circulant matrix form. We also show that row 0 of $\mathbf{P_s}$ is a permutation of processor indices. Since $\mathbf{P_s}$ is a generalized circulant matrix, it follows that every row of $\mathbf{P_s}$ is a distinct permutation of processor indices.

The column reorganization in any column j of $\mathbf{S}$ proceeds as follows:

(a) Stage 1 ($\mathbf{S}$ to $\mathbf{S_1}$): We observe that elements whose row indices differ by a multiple of K' have the same modulo value with respect to K. Consider rows i and $i'' = i + lK', 0 < l < G$. From Eq (2),

$$\begin{aligned}
\mathbf{S}(i,j) &= iP + j \\
\mathbf{S}(i'',j) &= i''P + j \\
&= (i + lK')P + j \\
\mathbf{S}(i'',j) - \mathbf{S}(i,j) &= lK'P \\
&= lK'P'G \\
\mathbf{S}(i + lK', j) - \mathbf{S}(i,j) &= lP'K \\
&\equiv 0 \pmod{K}
\end{aligned} \tag{3}$$

In each column, for each r, $0 \leq r < K'$, there are exactly G elements such that $i \bmod K' = r$. In the first stage, these elements are brought together by moving row i to row $i' = (i \bmod K')G + i/K'$. This constitutes the new table $\mathbf{S_1}$. Thus

$$\mathbf{S}(i,j) = \mathbf{S_1}((i \bmod K')G + i/K', j) \tag{4}$$

Figure 7 shows the tables $\mathbf{S}$ and $\mathbf{S_1}$ for $\Re_x(6,9)$. Here $G = 3$ and $K' = 2$. Consider the table $\mathbf{S}$ in Figure 7(a). Rows 0, 2, 4 are moved to positions 0, 1, 2 in $\mathbf{S_1}$, respectively. Similarly, rows 1, 3, 5 are brought together in rows 3, 4, and 5 of $\mathbf{S_1}$, as shown in Figure 7(c).

$\mathbf{S_1}$ has $K' \times P'$ submatrices, each of size $G \times G$, as shown in Eq (6) and Figure 7(c). These submatrices are denoted by $\mathbf{B}_{i_1,j_1}$, $(0 \leq i_1 < K', 0 \leq j_1 < P')$. The element $\mathbf{S_1}(i,j)$ is contained in submatrix $\mathbf{B}_{i_1,j_1}$, and is located at position (i_2, j_2) within $\mathbf{B}_{i_1,j_1}$. Note that $0 \leq i_1 < K', 0 \leq j_1 < P', 0 \leq i_2, j_2 < G$. The value of any element $\mathbf{S_1}(i,j)$ can be computed by

the reverse of Eq (4):

$$\begin{aligned}\mathbf{S_1}(i,j) &= \mathbf{S}((i \bmod G)K' + i/G, j) \\ &= \mathbf{S}(i_2K' + i_1, j) \\ &= (i_2K' + i_1)P + j \\ &= (i_1P' + j_1)G + (i_2K'P + j_2) \end{aligned} \tag{5}$$

Eq (5) has 2 components. The first component depends only on (i_1, j_1) and is fixed for all elements of submatrix $\mathbf{B}_{i_1,j_1}$. The second component depends on the position (i_2, j_2) within each submatrix. Eq (7) shows the elements in submatrix $\mathbf{B}_{i_1,j_1}$. The fixed component is shown outside the matrix while the variable component is shown at the position of the respective element. This equation can be validated using the example of Figure 7(c).

$$\mathbf{S_1} = \begin{bmatrix} \mathbf{B}_{0,0} & \mathbf{B}_{0,1} & \cdots & \mathbf{B}_{0,P'-1} \\ \mathbf{B}_{1,0} & \mathbf{B}_{1,1} & \cdots & \mathbf{B}_{1,P'-1} \\ \vdots & \vdots & & \vdots \\ \mathbf{B}_{K'-1,0} & \mathbf{B}_{K'-1,1} & \cdots & \mathbf{B}_{K'-1,P'-1} \end{bmatrix} \tag{6}$$

$$\mathbf{B_{i_1,j_1}} = (i_1P' + j_1)G + \begin{bmatrix} 0 & 1 & \cdots & G-1 \\ K'P & K'P+1 & \cdots & K'P+G-1 \\ \vdots & \vdots & & \vdots \\ (G-1)K'P & (G-1)K'P+1 & \cdots & (G-1)(K'P+1) \end{bmatrix} \tag{7}$$

(b) Stage 2 ($\mathbf{S_1}$ to $\mathbf{S}'$): We can transform $\mathbf{S_1}$ into $\mathbf{S}'$ by moving the element at position (i', j) of $\mathbf{S_1}$ to position (i, j) of $\mathbf{S}'$ such that

$$i_1 = (i'_1P' + j_1) \bmod K' \tag{8}$$

$$i_2 = (j_2 + i'_2) \bmod G \tag{9}$$

(Recall that $i = i_1G + i_2$.)

$\mathbf{P_s}$ will have the following form, where $\mathbf{P_s}(i,j) = \mathbf{S}'(i,j)/K$:

$$\mathbf{P_s} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \cdots & \mathbf{A}_{0,P'-1} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{1,P'-1} \\ \vdots & \vdots & & \vdots \\ \mathbf{A}_{K'-1,0} & \mathbf{A}_{K'-1,1} & \cdots & \mathbf{A}_{K'-1,P'-1} \end{bmatrix} \tag{10}$$

$$\mathbf{A}_{i_1,j_1} = \frac{(i_1'P' + j_1)}{K'} + \begin{bmatrix} 0 & (G-1)P' & \cdots & P' \\ P' & 0 & \cdots & (2)P' \\ \vdots & \vdots & & \vdots \\ (G-1)P' & (G-2)P' & \cdots & 0 \end{bmatrix} \tag{11}$$

We now show that $\mathbf{P_s}$ is a generalized circulant matrix. Each $G \times G$ submatrix $\mathbf{A}_{i_1,j_1}$ of Eq (10) consists of two components, shown in Eq (11). These are referred to as the base and the offset components. Figure 11 shows the base and offset components of $\mathbf{S}'$ and $\mathbf{P_s}$, for $\Re_x(6,9)$. The rows and columns of the base component are indexed by i_1 and j_1, respectively. The rows and columns of the offset component are indexed by i_2 and j_2, respectively. Thus, the entries of the base component are independent of i_2 and j_2. The entries within any $G \times G$ submatrix of $\mathbf{P_{base}}$ are therefore all identical to one another, and are given by $(i_1'P' + j_1)/K'$. Similarly, the entries of the offset component are independent of i_1 and j_1. All the $G \times G$ submatrices are therefore identical to one another. Each of these is a $G \times G$ circulant matrix. This fact can be proved as follows: in the offset term of Eq (7), each row has the same quotient when divided by K. All elements of row i_2 have a quotient of i_2P' when divided by K. The transformation $i_2 = (j_2 + i_2') \bmod G$ of Eq (9) converts each of these G rows into a diagonal in the offset term of Eq (11). Thus, this is a circulant matrix whose rows are obtained by shifting $[0\ (G-1)P'\ \cdots\ P']$ circularly.

Consider a row major indexing $r = i_1K' + j_1$ imposed on submatrices $\mathbf{B}_{i_1,j_1}$ of Eq (6). Here, $0 \le r < P'K'$. We refer to each collection of K' adjacent submatrices starting at global index lK' as a *run*. Run l contains submatrices whose indices range from lK' to $(l+1)K' - 1$. Within a run, $r \bmod K'$ ranges from 0 to $K' - 1$. During the transformation from $\mathbf{S_1}$ to $\mathbf{S}'$, the k^{th} submatrix of the run moves to the position k within its column, $0 \le k < K'$. Thus, these K' submatrices of a run constitute a diagonal of submatrices of $\mathbf{S}'$. From Eq (5), it can be seen that the first component $(i_1'P' + j_1)G$ yields the same quotient when divided by K, for each of the K' submatrices in a run. Since the elements of $\mathbf{P_s}(i,j)$ are obtained by $\mathbf{S}'(i,j)/K$, each diagonal of $\mathbf{P_s}$ contains identical submatrices.

We now show that row 0 of $\mathbf{P_s}$ is a permutation of processor numbers. Since $i_1 = 0$ for this row, Eq (8) gives $(i_1'P' + j_1) \bmod K' = 0$. Therefore, $(i_1'P' + j_1) \in \{0, K', \ldots (P'-1)K'\}$ and $(i_1'P' + j_1)/K' \in \{0, 1, \ldots P' - 1\}$. The first row of the second term of Eq (11) is given by $[0\ (G-1)P'\ \cdots\ P']$. Thus, it is easy to see that row 0 of $\mathbf{P_s}$ is a permutation of the processor ids.

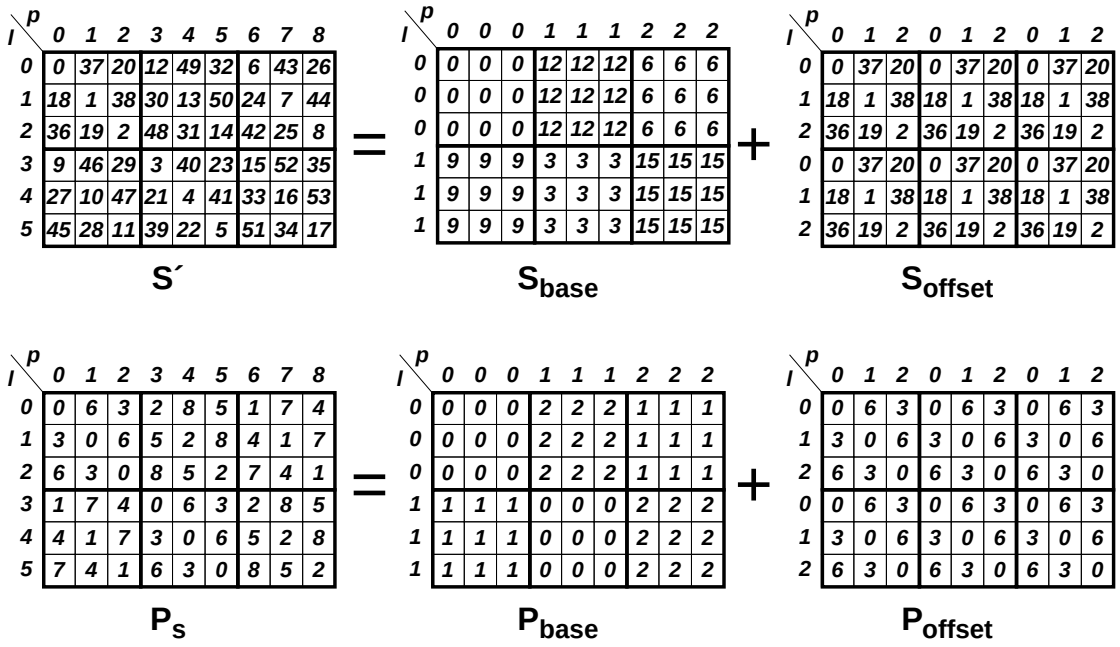


Figure 11: Decomposition of $\mathbf{S}'$ and $\mathbf{P_s}$

Thus, $\mathbf{P_s}$ is a generalized circulant matrix. Also, every row is a distinct permutation of processor numbers.

Note that, if $G = 1$, then $K' = K$, $P' = P$, $\mathbf{S_1} = \mathbf{S}$. Also, the second term in Eq (7) and Eq (11) will reduce to 0.

□

With reference to Figure 4, observe that $\mathbf{P_s}$ helps to specify the row transformations that convert $\mathbf{S}'$ to $\mathbf{R}'$. The initial column transformations convert $\mathbf{S}$ to $\mathbf{S}'$ and $\mathbf{P}$ to $\mathbf{P_s}$. Although Section 3 indicates that the column transformations 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. This is achieved using a *send data location table* $\mathbf{D_s}$. Each entry of $\mathbf{D_s}$, $\mathbf{D_s}(i,j)$ shows that the block corresponding to the entry $\mathbf{S}'(i,j)$ was initially located at the row $\mathbf{D_s}(i,j)$ and the column j in $\mathbf{S}$. Therefore, $\mathbf{S}'(i,j) = \mathbf{S}(\mathbf{D_s}(i,j),j)$. Similarly, $\mathbf{P_s}(i,j) = \mathbf{P}(\mathbf{D_s}(i,j),j)$. The row transformations are specified by the *destination processor table* $\mathbf{P_s}$ with respect to $\mathbf{S}'$. Each entry of $\mathbf{P_s}$, $\mathbf{P_s}(i,j)$ shows that the block corresponding to the entry $\mathbf{S}'(i,j)$ must be moved to the row i and the column $\mathbf{P_s}(i,j)$ in $\mathbf{R}'$. In other words, $\mathbf{S}'(i,j) = \mathbf{R}'(i,\mathbf{P_s}(i,j))$.

In an analogous way, the *source processor table* $\mathbf{P_r}$ specifies the row transformation with respect to $\mathbf{R'}$ by $\mathbf{R'}(i,j) = \mathbf{S'}(i, \mathbf{P_r}(i,j))$. Another set of column transformations are required to convert $\mathbf{R'}$ to $\mathbf{R}$. The *receive data location table* $\mathbf{D_r}$ specifies such column transformations by $\mathbf{R'}(i,j) = \mathbf{R}(\mathbf{D_r}(i,j), j)$.

Theorem 2 gives the formulae to compute the individual entries of $\mathbf{P_s}$ and $\mathbf{D_s}$ efficiently. Each row of $\mathbf{P_r}$ is the inverse permutation of $\mathbf{P_s}$. Since $\mathbf{P_r}$ and $\mathbf{D_r}$ can be computed in a similar manner as $\mathbf{P_s}$ and $\mathbf{D_s}$, we do not discuss them further.

Theorem 2 *A destination processor table in generalized circulant matrix form* ($\mathbf{P_s}$) *and the corresponding send data location table* $\mathbf{D_s}$ *can be constructed as follows:*

$$\mathbf{P_s}(i,j) = \{n(j_1 - i_1)\} \bmod P' + P'\{(i_2 - j_2) \bmod G\} \tag{12}$$

$$\mathbf{D_s}(i,j) = \{m(j_1 - i_1)\} \bmod K' + K'\{(i_2 - j_2) \bmod G\} \tag{13}$$

where n *and* m *are solutions to* $nK' - mP' = 1$

Proof: From Theorem 1, the $(i,j)^{th}$ element of $\mathbf{S'}$ is

$$\mathbf{S'}(i,j) = (i'_1 P' + j_1)G + (i'_2 K' P + j_2) \tag{14}$$

where

$$i_1 = (i'_1 P' + j_1) \bmod K' \tag{15}$$

$$i_2 = (j_2 + i'_2) \bmod G \tag{16}$$

Let $a = (i'_1 P' + j_1)$. From Eq (15) and the definition of a,

$$a = X K' + i_1 = Y P' + j_1 \tag{17}$$

Hence,

$$XK' - YP' = (j_1 - i_1) \tag{18}$$

We wish to find X and Y in the range $0 \le X < P', 0 \le Y < K'$. Since $gcd(K', P') = 1$, we can find m, n using the Euclid algorithm such that

$$nK' - mP' = 1 \tag{19}$$

Comparing Eq (18) and Eq (19),

$$X = \{n(j_1 - i_1)\} \bmod P' \tag{20}$$

$$Y = \{m(j_1 - i_1)\} \bmod K' \tag{21}$$

Because $0 \leq i_2, j_2 < G$, Eq (16) becomes $i'_2 = (i_2 - j_2) \bmod G$. Therefore, $\mathbf{P_s}(i,j) = \mathbf{S}'(i,j)/K = [(XK' + i_1)G + \{(i_2 - j_2) \bmod G\}K'P + j_2]/K$ becomes Eq (12) since $i_1 G + j_2 < K$. Similarly, we can prove Eq (13). □

Note that the above formulae for computing the entries of $\mathbf{P_s}$ and $\mathbf{D_s}$ are computationally efficient compared to the approach used in [23]. Our algorithms as well as the scheme in [23] compute $gcd(P, K)$. We use the extended Euclid algorithm to find m and n as a byproduct of computing the gcd. Using the above formulae, we can then compute entries of $\mathbf{P_s}$ and $\mathbf{D_s}$ with very little computational overhead, and without any interprocessor communication.

The following corollary gives a more efficient scheme to compute $\mathbf{P_s}$ and $\mathbf{D_s}$. Using these formulae, processor p_j can iteratively compute column j of $\mathbf{P_s}$ and $\mathbf{D_s}$. Eq (12) and Eq (13) are used to compute $\mathbf{P_s}(0,j)$ and $\mathbf{D_s}(0,j)$. The formulae below can be converted into a simple `for` loop which computes the remaining entries in column j. Thus, expensive operations like mod and multiplication can be reduced in index computation. The formulae can also reduce the space requirements for storing the tables. When a direct schedule is used, only a single entry of the table needs to be stored at a time.

Corollary 1 *Elements within adjacent rows of* $\mathbf{P_s}$ *and* $\mathbf{D_s}$ *have the following relation:*

$$\mathbf{P_s}(i+1,j) = \begin{cases} (\mathbf{P_s}(i,j) + P') \bmod P, & \text{if } ((i+1) \bmod G) \neq 0 \\ (\mathbf{P_s}(i,j) + P' + (-n) \bmod P') \bmod P, & \text{otherwise} \end{cases}$$

$$\mathbf{D_s}(i+1,j) = \begin{cases} (\mathbf{D_s}(i,j) + K') \bmod K, & \text{if } ((i+1) \bmod G) \neq 0 \\ (\mathbf{D_s}(i,j) + K' + (-m) \bmod K') \bmod K, & \text{otherwise} \end{cases}$$

where $i = 0, 1, \ldots, K-2$ *and* $j = 0, 1, \ldots, P-1$, *and* $nK' - mP' = 1$.

□

We now give the pseudocode for the interprocessor communication operations (the row transformations) in Figure 12. Theorem 3 analyzes the complexity of these communication algorithms.

Direct Communication Schedule
G=gcd(K,P), K=GK′, P=GP′

for stage= 0 **to** K-1 **do**
  dst = **Ps**[stage]
  src = **Pr**[stage]

  Send elements at **Ds**[stage] to dst and receive elements from src and store at **Dr**[stage]
endfor

Indirect Communication Schedule
G=gcd(K,P), K=GK′, P=GP′
$j_1 = P_{id} / G$, $j_2 = P_{id} \bmod G$
$i_1 = i / G$, $i_2 = i \bmod G$

<Cyclic Shift 1>

stride = 1
for stage = 0 **to** $\lceil \log K' \rceil$ **do**
  src = $\{(j_1+stride) \bmod P'\}*G + j_2$
  dst = $\{(j_1-stride) \bmod P'\}*G + j_2$

  Send elements at **Ds**[i] to dst and receive elements from src and store at **Dr**[i] such that (i_1 & stride) != 0

  stride = stride*2
endfor

<Cyclic Shift 2>

stride = 1
for stage = 0 **to** $\lceil \log G \rceil$ **do**
  src = $j_1*G + \{(j_2+stride) \bmod G\}$
  dst = $j_1*G + \{(j_2-stride) \bmod G\}$

  Send elements at **Ds**[i] to dst and receive elements from src and store at **Dr**[i] such that (i_2 & stride) != 0

  stride = stride*2
endfor

Hybrid Communication Schedule
d = level of indirection
G=gcd(K,P), K=GK′, P=GP′
$j_1 = P_{id} / G$, $j_2 = P_{id} \bmod G$
$i_1 = i / G$, $i_2 = i \bmod G$

if (d < $\lceil \log K' \rceil$) **then** $nstage_1$=d, $nstage_2$=0
else $nstage_1$=$\lceil \log K' \rceil$, $nstage_2$=d-$\lceil \log K' \rceil$

$nstage_1$ steps of <Cyclic Shift 1>
stride = 1
for stage = 0 **to** $nstage_1$-1 **do**
  src = $\{(j_1+stride) \bmod P'\}*G + j_2$
  dst = $\{(j_1-stride) \bmod P'\}*G + j_2$
  Send elements at **Ds**[i] to dst and receive elements from src and store at **Dr**[i] such that (i_1 & stride) != 0
  stride = stride*2
endfor
$nstage_2$ steps of <Cyclic Shift 2>
stride = 1
for stage = 0 **to** $nstage_2$-1 **do**
  src = $j_1*G + \{(j_2+stride) \bmod G\}$
  dst = $j_1*G + \{(j_2-stride) \bmod G\}$
  Send elements at **Ds**[i] to dst and receive elements from src and store at **Dr**[i] such that (i_2 & stride) != 0
  stride = stride*2
endfor
$\lceil K/2^d \rceil$ steps of direct schedule
for stage = 0 **to** $\lceil K/2^d \rceil$-1 **do**
  dst = **Ps**[stage]
  src = **Pr**[stage]
  Send elements at **Ds**[stage] to dst and receive elements from src and store at **Dr**[stage]
endfor

Multiphase Algorithm
$K_1 \bullet K_2 \bullet\bullet\bullet K_n$=prime decomposition(K)
for i = 1 **to** n **do**
  Direct Communication Schedule with K_i and P
endfor

Figure 12: Direct, indirect, hybrid, and multiphase algorithms

Theorem 3 *If the destination processor table of size $K \times P$ is in generalized circulant matrix form as shown in Eq(1) and if every row is a permutation of* $\{0, 1, \ldots, P-1\}$ *, the redistribution specified by the dpt can be performed in a contention free manner in* (i) K *communication steps using a direct schedule,* (ii) $\lceil \log K \rceil + 2$ *communication steps using an indirect schedule, and* (iii) $d + \lceil \frac{K}{2^d} \rceil$ *communication steps using a hybrid schedule with d degree of indirection.*

Proof: Consider a destination processor table $\mathbf{P_s}$, which is in generalized circulant matrix form and each row is a permutation. The communication schedules in all our algorithms are specified as a sequence of row transformations on $\mathbf{P_s}$ such that in its final form, column j of $\mathbf{P_s}$ contains all the K elements whose value is j. A row transformation permutes the elements within a row of $\mathbf{P_s}$. Consider a transformation that moves an element from position (i, j) to position (i, k) in row i of $\mathbf{P_s}$. This corresponds to an interprocessor communication event where processor p_j sends a data block to processor p_k. The number of communication steps required by the algorithm is equal to the number of transformations performed on $\mathbf{P_s}$. We therefore prove the theorem in terms of the number of transformations required to bring $\mathbf{P_s}$ to its desired final form.

Direct Schedule: We know that each row of $\mathbf{P_s}$ is a distinct permutation of processor numbers. Transformation i permutes the elements in row i $(0 \leq i < K)$ of $\mathbf{P_s}$ so that each element labeled j is moved to column j, $(0 \leq j < P)$. After K such transformations, $\mathbf{P_s}$ is in the desired form.

Indirect Schedule: $\mathbf{P_s}$ consists of $K' \times P'$ submatrices, each of size $G \times G$. The algorithm first moves these submatrices such that each block diagonal of $\mathbf{P_s}$ is vertically aligned in $\lceil \log K' \rceil$ transformations. Next, the G elements within each submatrix are aligned vertically in $\lceil \log G \rceil$ transformations. At this point, all elements labeled x are in the same column, say column k. In the last transformation, entire columns are permuted so that column j contains all the elements labeled j, $0 \leq j < P$. (In many applications, this transformation is not needed.) Thus, $\lceil \log K' \rceil + \lceil \log G \rceil + 1 \leq \lceil \log K \rceil + 2$ communication steps are required in the indirect schedule.

Hybrid Schedule: This algorithm performs the first d steps of the indirect algorithm ($0 \leq d \leq (\lceil \log K' \rceil + \lceil \log G \rceil)$) and aligns data blocks of size 2^d vertically. A direct schedule consisting of $\lceil \frac{K}{2^d} \rceil$ steps is then performed. The total number of communication steps required in the hybrid schedule is thus $d + \lceil \frac{K}{2^d} \rceil$. □

Corollary 2 *Consider an array with N elements, each of size b bytes, distributed over P processors. Using the GDM model, the communication complexities for $\Re_x(K, P)$ are* (i) $KT_d + \frac{N}{P} b \tau_d$

time using a direct schedule, (ii) $(\lceil \log K \rceil + 2)T_d + ((\lceil \log K \rceil + 1)\frac{N}{2P} + \frac{N}{P})b\tau_d$ *time using the indirect schedule,* (iii) $(d + \lceil \frac{K}{2^d} \rceil)T_d + (\frac{Nd}{2P} + \frac{N}{P})b\tau_d$ *time using the hybrid schedule.*

□

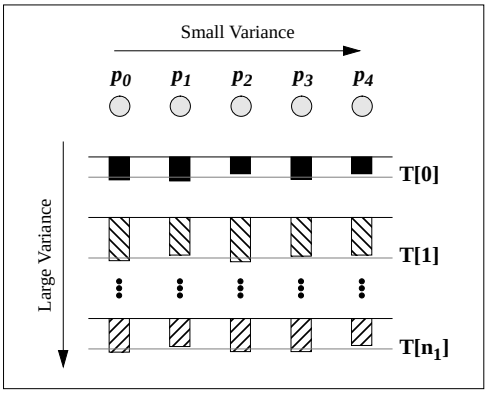
5 Experimental Results

In the previous sections, we have discussed our approach to the design of efficient redistribution algorithms for $\Re_x(K, P)$. We also showed pseudocode for the direct, indirect, and hybrid communication schedules and analyzed their complexity in Theorem 3. This section shows timing results from implementations of our redistribution algorithms on the IBM SP-2 and the Cray T3D, which are coarse-grain distributed memory parallel machines. The architecture of these machines is typical of current generation coarse grained parallel architectures. The results reported in this section are therefore representative of the entire class of distributed memory parallel machines. In terms of the LogGP model, $T_d = 40\mu$sec and $\tau_d = 35$ nsec/byte for the IBM SP-2; $T_d = 93\mu$ sec and $\tau_d = 43$ nsec/byte for the Cray T3D. The anticipated variations in performance due to different values of T_d and τ_d are also discussed in this section.

Our methodology for evaluating performance is shown in Figure 13. The redistribution algorithms are coded using C and MPI. Hence, the developed code is easily portable across a number of HPC systems. In our implementations, MPI function calls are only used for node to node send and receive operations. We have developed our collective communication algorithms using these basic primitives. We shall also show performance comparisons with an MPI collective communication routine. The redistribution time is measured using `MPI_Wtime`. First, the average time for `n2` consecutive runs of the redistribution subroutine is measured on each processor (`node_time`). The average of `node_time` over P processors is then computed (`tavg`). Our experimental results show that the variance of `node_time` over P processors is insignificant. To reduce the errors due to OS interference, we repeat the measurement of `tavg` `n1` times. The measured values are stored in an array `T`, as shown in Figure 13(b). Finally, the quantities Tmax=max{T[i]}, Tmin=min{T[i]}, Tmed=median{T[i]}, and Tavg=avg{T[i]} are calculated.

Figure 14 shows the difference between these performance metrics through an example. Here, the performance of direct, indirect, and hybrid schedule (with $d = 1$) are measured on a 128 node IBM SP-2. The array size used is 1.6 Mbytes. The expansion factor K is varied in the range $3 \leq K \leq 128$. Figure 14(a) shows the maximum value (`Tmax` in Figure 13(a)) for each

```
for (i=0; i<n1; i++) {
   ts = MPI_Wtime
   for (j=0; j<n2; j++) {
      redistribution subroutine
   }
   te = MPI_Wtime
   node_time = (te-ts) / n2
   tavg = average of node_time over P nodes.
T[i] = tavg
}
compute Tmax = max {T[i]}, Tmin = min {T[i]}
        Tmed = median {T[i]}, Tavg = avg {T[i]}
```



(a) (b)

Figure 13: Set up for measuring the redistribution time.

experiment. Observe that there are a few points whose maximum value is very large. These arise due to OS interference during a few of the trials. The effect of these outliers is also seen in Figure 14(b) which shows the average time (`Tavg` in Figure 13(a)). These cannot be eliminated unless the experiment is repeated a large number of times. Figure 14(c) shows that this effect can be reduced by using the median value `Tmed` instead of `Tavg`. Figure 14(d) shows the best observed communication time `Tmin`. This plot eliminates the variance due to OS interference, and allows an accurate comparison of the relative performance of the redistribution algorithms. In the remaining plots in this section, we shall show only `Tmin`.

From Figure 14(d), it can be seen that the indirect schedule takes the least amount of communication time, while the direct schedule takes the longest. The communication time for the hybrid schedule is in between these values. As the value of K increases, the difference in the communication times of the schedules becomes larger. This performance trend is to be expected for any fixed array size. The direct schedule takes K steps, while the indirect schedule takes only $\lceil \log K \rceil + 2$ steps. Hence, the difference in the number of communication steps for these schedules increases significantly with increase in K.

Figure 15 shows the result of a similar experiment on the Cray T3D. The array size used is 0.4 Mbytes. The figure shows only `Tmin`. The performance trend is similar to that in Figure 14. This plot also shows the performance of the multiphase algorithm. The multiphase algorithm improves upon the direct algorithm for composite values of K. However, the communication time of the multiphase algorithm varies considerably with the value of K, since the communication time depends upon the factors of K. When K is prime, the algorithm degenerates to the direct algorithm. In comparison, our indirect algorithm does not show such fluctuations. It is uniformly

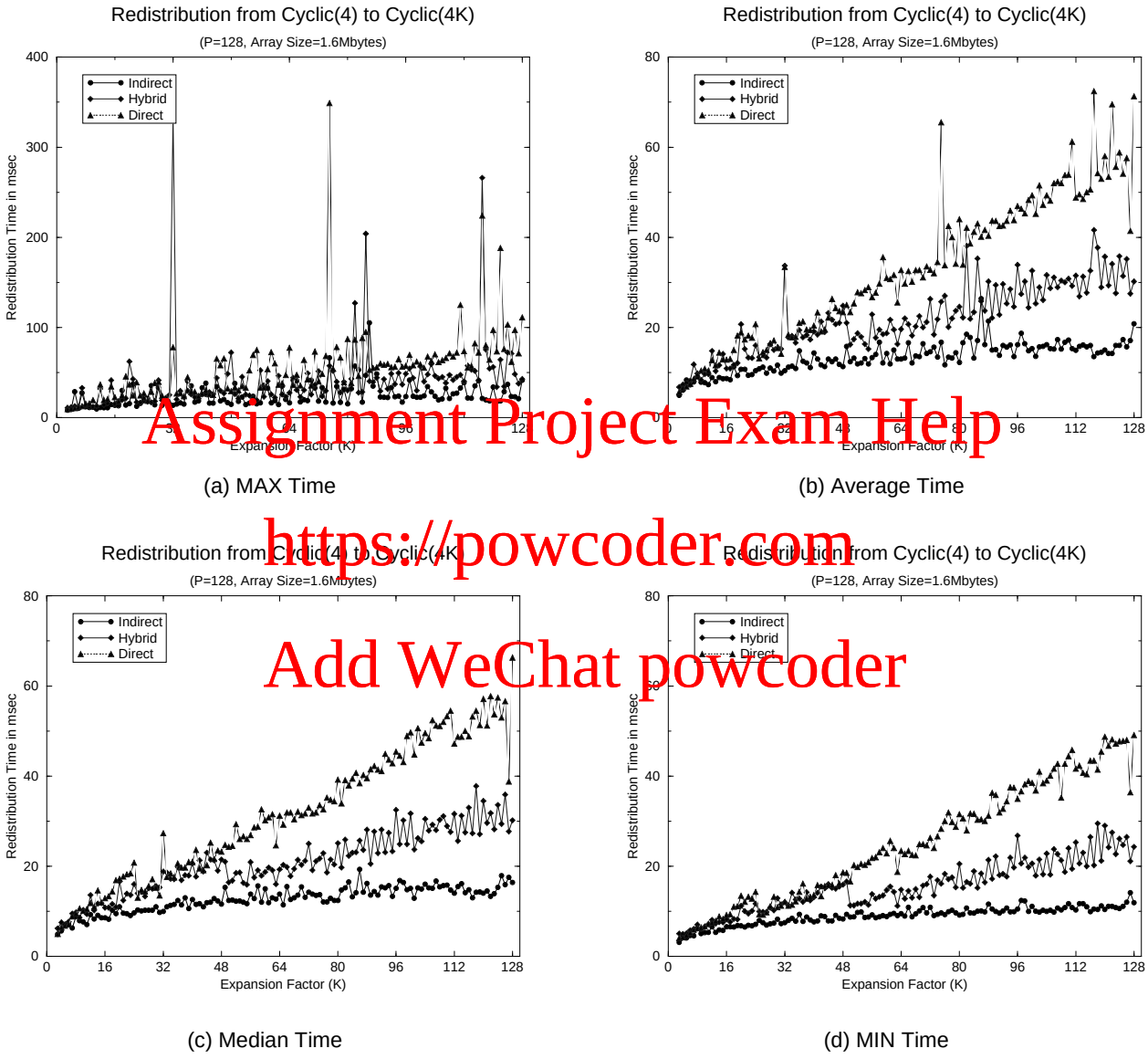


Figure 14: The maximum, average, median, and minimum times for redistributing an array of size 1.6 Mbytes on the IBM SP-2 (`n1`=40, `n2`=20).

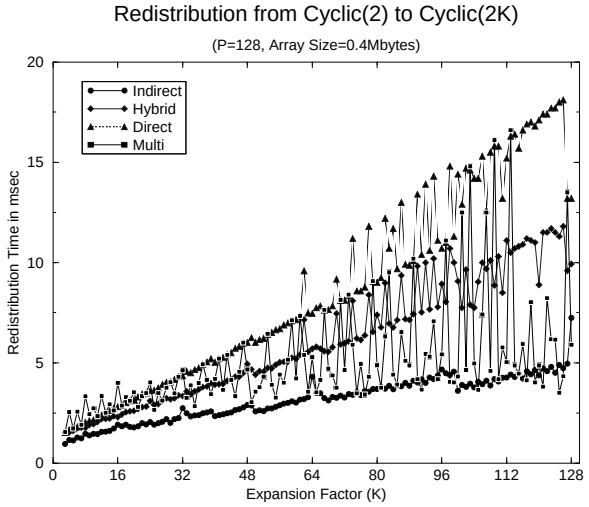
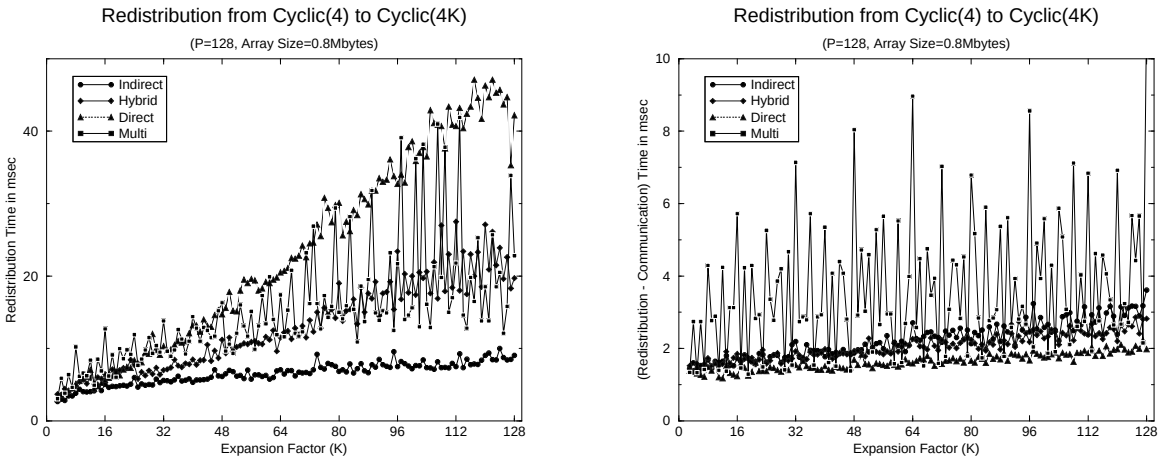


Figure 15: Timing results (`Tmin`) for direct, indirect, hybrid and multiphase algorithms on the Cray T3D.

applicable for prime and composite values of K. Apart from the fact that our algorithms use fewer communication steps, at most $\lceil \frac{N}{2P} \rceil$ array elements are moved in each step per processor (N is the total number of array elements). In comparison, the multiphase approach moves the entire array in every phase. Thus $\lceil \frac{N}{P} \rceil$ array elements are moved in each communication phase by each processor.

Figure 16(a) shows the timing performance when the array size is reduced to 0.8 Mbytes on the IBM SP-2. We know that the total communication time consists of two components: start-up costs and data transmission costs. From Theorem 3 and its corollary, the data transmission cost is dependent upon the array size, while the start-up cost depends only on K, P, and d. Therefore, the performance characteristics in Figure 16(a) are similar to Figure 14(d). The reduction in array size reduces the amount of data sent in each communication step, but does not reduce the number of communication steps. However, the value of the communication times are reduced for all the algorithms. The reduction is largest for the indirect algorithm.

Figure 16(b) shows the difference between the redistribution time and the communication time only. This includes the time for computing index sets and for buffer copying operations (*i.e.*, packing and unpacking), but not the interprocessor communication time. By comparing Figure 16(a) with Figure 16(b), it can be seen that the interprocessor communication time constitutes the major portion of the time for the direct schedule. The indirect schedule reduces this communication cost since it consists of fewer communication steps. The index computation



(a) Redistribution Time

(b) Redistribution Time - Communication Time

Figure 16: Comparison of direct, hybrid, multi-phase, and indirect schedules on the IBM SP-2.

cost for the multiphase algorithm is significant. This is because the factors of K must be computed.

In Figures 14, 15, and 16, `MPI_Sendrecv` is used in all the communication steps, *i.e.*, the communication schedules are *synchronous*. Each communication step is completed before the next communication step begins. In the case of direct and hybrid schedules, we can employ an *asynchronous* approach by using `MPI_Irecv` and `MPI_Waitany`. Each processor first posts K non-blocking receive commands, and then invokes send communication routines. An asynchronous approach cannot be used for the indirect schedule. This is because the messages sent by a processor in any step must include data received by the processor in the previous step. The timing results for these algorithms are shown in Figure 17. By comparing Figure 17(a) with Figure 16(a), and Figure 17(b) with Figure 15, it can be inferred that by reducing the synchronization overheads, the communication performance of the direct and hybrid algorithms can be improved. However, the performance of the asynchronous direct algorithm is still worse than that of the synchronous indirect algorithm.

Cyclic redistribution can also be performed using the MPI collective communication routine `MPI_Alltoall`. However, the message sizes between every pair of processors in $\Re_x(K, P)$ are not identical. Dummy data can be sent to equalize the message sizes. Figure 18 compares the performance of redistribution using `MPI_Alltoall` with other approaches. The redistribution time using `MPI_Alltoall` is very large for small values of K, because of large amounts of dummy data transfer.

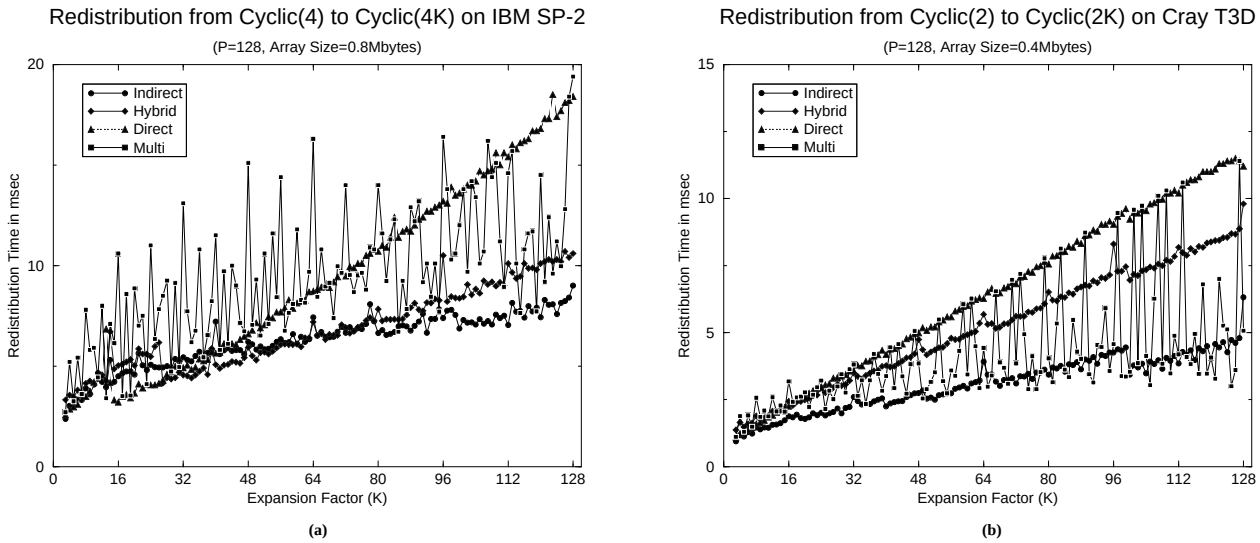


Figure 17: Comparison of *asynchronous* implementation of direct, hybrid, multi-phase, and *synchronous* indirect schedules on the (a) IBM SP-2 and (b) Cray T3D.

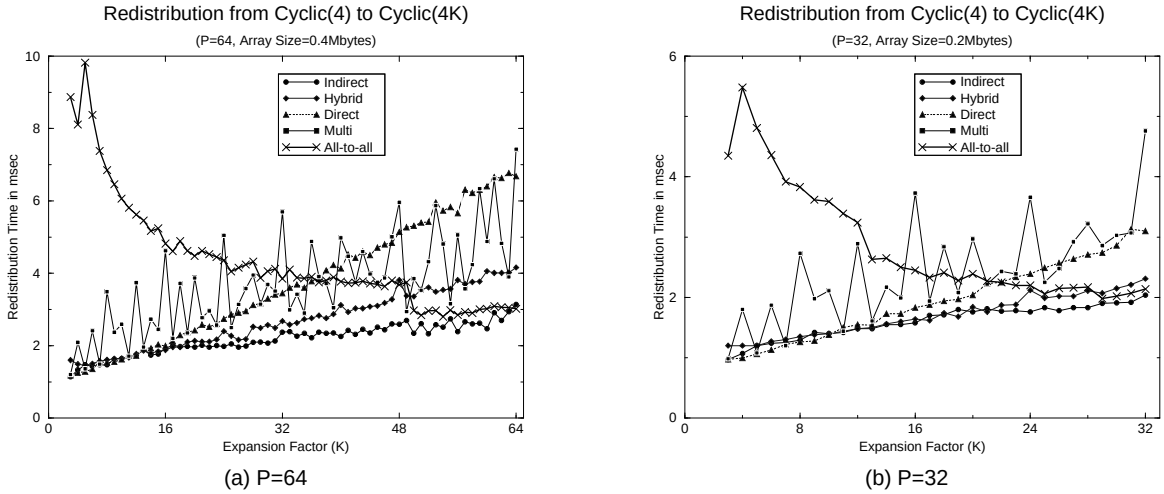


Figure 18: The comparison of *all-to-all*, *asynchronous* direct, hybrid, multi-phase, and *synchronous* indirect schedules on the IBM SP-2.

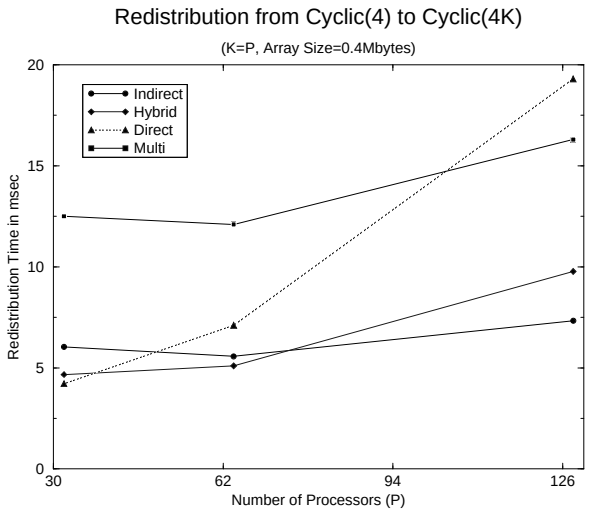


Figure 19: Comparison of *asynchronous* implementation of direct, hybrid, multiphase, and *synchronous* indirect schedules when the array size is fixed and the number of processors is varied (IBM SP-2).

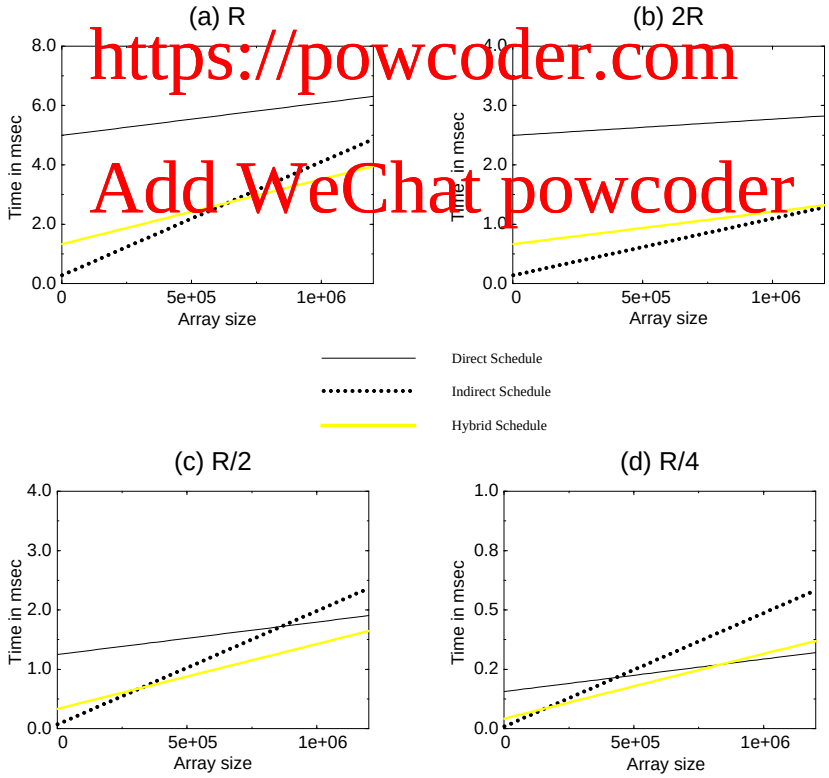


Figure 20: Predicted time for redistribution using direct, indirect, and hybrid algorithms when the ratio of machine parameters ($R = \frac{T_d}{\tau_d}$) changes due to advances in technology (P=128, K=128).

Figure 19 shows the effect of varying the number of processors on $\Re_x(K, P)$. A fixed array size of 0.4 Mbytes is used and K is chosen to be equal to P. For small values of P, the portion of the array within each processor is large. This results in large data transmission costs. The direct algorithm performs better than the indirect algorithm in this situation. As the number of processors is increased, the start-up costs become dominant. The communication time of the direct algorithm increases significantly, and the indirect algorithm yields the best performance.

The above plots have been obtained based on experiments performed on the IBM SP-2 and the Cray T3D. However, the machine parameters (T_d, τ_d) change from one machine to another and also change with technological advances. Advances in networking technology will reduce the unit transmission time τ_d. Progress in processor technology and communication software can reduce the startup time T_d. Note that τ_d also depends on the processor technology and will improve with processor technology. We believe that in future message passing systems the time for buffer copy, for invoking a system call, and for communication protocol processing can be reduced but it cannot be completely ignored. Figure 20 shows the influence of these changes on our algorithms. We assume that the number of processors $P = 128$, and $K = P$. Let $R = \frac{T_d}{\tau_d}$. Figure 20(a) shows the case when $T_d = 40\mu$sec and $\tau_d = 35$nsec/byte. These are the machine parameters for the IBM SP-2. The horizontal axis shows N, the number of elements in the array. The size of each element is assumed to be 4 bytes. For the hybrid schedule, a value of $d = 2$ is chosen. Figure 20 (b) shows the performance when T_d and τ_d are reduced by factors of 2 and 4 respectively. Thus, the ratio becomes $2R$. In Figure 20 (c), T_d and τ_d are reduced by factors of 4 and 2 respectively. In Figure 20 (d), T_d and τ_d are reduced by factors of 32 and 8 respectively. The plots show the performance of our communication schedules for various message sizes. It can be inferred that there always exists a range of message size where the startup costs dominate. This range depends on the machine features and array size.

6 Conclusion

In this paper, we have proposed a uniform framework for block cyclic redistribution of arrays. Arrays are one of the most commonly used data structures in many HPC applications such as signal processing and dense linear algebra. The block cyclic distribution represents typical data access patterns in these applications. Redistribution is necessary when the data access patterns of the application change at intermediate points in the computation. We have focused on an

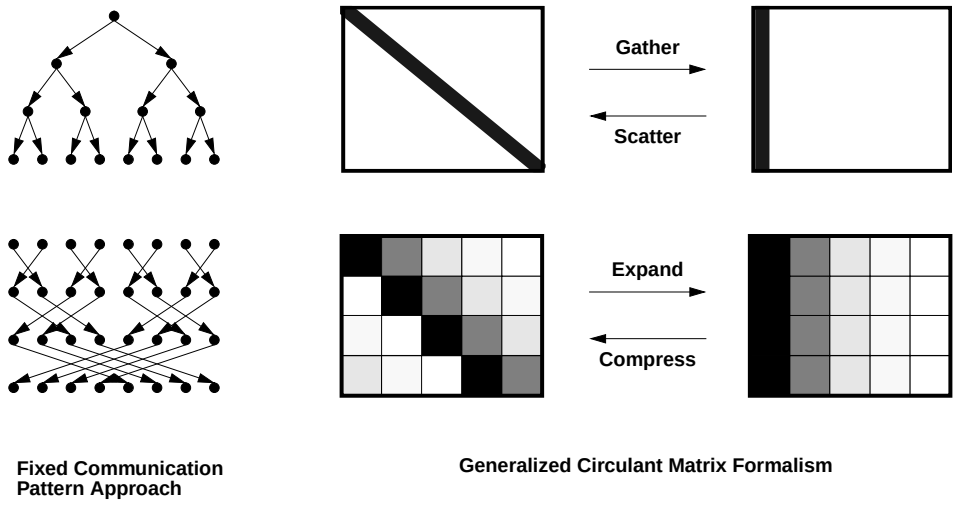


Figure 21: Fixed communication pattern approach vs. generalized circulant matrix formalism

important case of block cyclic redistribution, $\Re_x(K, P)$, where the block size changes by a factor K. Our framework can be uniformly applied to all instances of $\Re_x(K, P)$. This is in contrast to some previous approaches which are inefficient for certain parameter values *eg.*, when K is prime.

In our framework, the redistribution algorithm is viewed as a table conversion process. The destination processor table is first transformed into a generalized circulant matrix form by column-wise reorganization. This corresponds to diagonally arranging the entries which have the same destination processor. Efficient communication schedules are then derived based on this generalized circulant matrix. We discussed the direct, indirect, and hybrid communication schedules. The direct schedule is effective for large sized arrays while the indirect schedule is more suitable for smaller array sizes. The hybrid schedule is a combination of the direct and indirect schedules. We have developed formulae that estimate the performance of each schedule based on problem and machine parameters. Based on our framework, we have also developed efficient algorithms for index set computation that do not involve any interprocessor communication.

We have used the GDM model to estimate the performance of our redistribution algorithms. This model represents interprocessor communication events with a start-up cost component and a data transmission cost component. Experimental results on real HPC platforms validate the effectiveness of the GDM model and our communication schedules.

Our framework can be extended to derive communication schedules for other communication patterns by using a different set of column and row transformations. In the indirect schedule, entries along a diagonal of the destination processor table are aligned vertically in a logarithmic number of circular shift operations. In the block-cyclic redistribution from $cyclic(x)$ to $cyclic(Kx)$, P diagonals are aligned simultaneously in $\lceil \log K \rceil + 1$ steps. A similar communication pattern is used in many "combine-and-forward" redistributions. As shown in Figure 21, gather and scatter algorithms, which are developed using a binary tree, can also be viewed as aligning a single diagonal into a vertical line or *vice versa*.

In certain irregular problems, the index sets are not only determined at run time but they are also random in nature (variability in message size, destination and source processors, *etc*). To reduce this variability, a combine-and-forward schedule based on a fixed communication pattern is commonly used. Such schedules are not adequately general. We believe that the generality, uniformity, and efficacy of our framework can improve the performance of the data remapping problems in irregular applications. Extension of our indirect algorithms to these irregular problems will result in performance improvement over other approaches. The fixed communication approach also yields low performance for small sized messages. Thus, current ScaLAPACK implementations that utilize such an approach provide poor performance for small matrix sizes. In the general array assignment problems of parallelizing compilers, only some of the array elements are communicated. Our indirect schedules can be directly applied to both of these problems, to improve the performance.

Our research has been motivated by the need for utilizing coarse grained parallel computers in high performance signal processing environments. Typical applications are Radar and Sonar, including techniques such as Space-Time Adaptive Processing (STAP). In the area of communication scheduling, we have also developed redistribution algorithms for multidimensional arrays [14]. We are currently developing techniques for redistribution between distinct sets of source and destination processors [19]. Apart from communication scheduling, we have also developed tecniques for scheduling computational tasks in such applications on processors of HPC systems. Most signal processing applications consist of a pipeline of computational stages, through which a continuous stream of input data is processed. We have developed techniques to implement such software pipelines on coarse grained parallel architectures [10, 11, 17].

References

[1] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir, "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers," *IEEE Transactions on Parallel and Distributed Systems,* (6)2: pp. 154-164, Feb. 1995.

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

[3] S.E. Hambrusch and A.A. Khokhar, "C^3: A Parallel Model for Coarse-Grained Machines," *Journal of Parallel and Distributed Computing,* (32)2: pp. 139-154, Feb. 1996.

[4] J. JáJá and K. Ryu, "The Block Distributed Memory Model for Shared Memory Multiprocessors," *Proc. International Parallel Processing Symposium,* April 1994, pp. 752-756.

[5] E.T. Kalns and L.M. Ni, "Processor Mapping Techniques Toward Efficient Data Redistribution," *Proc. International Parallel Processing Symposium,* April 1994, pp. 469-476.

[6] S.D. Kaushik, C.-H. Huang, R.W. Johnson, and P. Sadayappan, "An Approach to Communication Efficient Data Redistribution," *Proc. International Conference on Supercomputing,* July 1994, pp. 364 - 373.

[7] 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.

[8] S.D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan, "Multiphase Array Redistribution: Modeling and Evaluation," *Proc. International Parallel Processing Symposium,* April 1995, pp. 441-445.

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

[10] M. Lee and V.K. Prasanna, "High Throughput-Rate Parallel Algorithms for Space Time Adaptive Processing," *Proc. 2nd International Workshop on Embedded HPC Systems and Applications (EHPC '97),* April 1997.

[11] M. Lee, W. Liu, and V.K. Prasanna, "A Mapping Methodology for Designing Software Task Pipelines for Embedded Signal Processing," *Proc. 3rd International Workshop on Embedded HPC Systems and Applications (EHPC '98),* April 1998.

[12] Y.W. Lim, P.B. Bhat, and V.K. Prasanna, "Efficient Data Remapping Algorithms for Embedded Signal Processing Applications," *Proc. 10th International Conference on High Performance Computers,* June 1996.

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

[14] Y.W. Lim, N. Park and V.K. Prasanna, "Efficient Algorithms for Multi-Dimensional Block-Cyclic Redistribution of Arrays," *Proc. International Conference on Parallel Processing,* Aug. 1997, pp. 234-241.

[15] Y.W. Lim and V.K. Prasanna, "Scalable Portable Implementations of Space-Time Adaptive Processing," *Proc. 10th International Conference on High Performance Computers,* June 1996.

[16] Y.W. Lim and V.K. Prasanna, "Efficient Algorithms for General Block-Cyclic Redistribution," *Technical Report,* University of Southern California, Dept. of EE-Systems, Aug. 1996.

[17] W. Liu and V.K. Prasanna, "Design of Application Software for Embedded Signal Processing," *Invited article, IEEE Signal Processing Magazine,* to appear.

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

[19] N. Park, V.K. Prasanna, and C.S. Raghavendra, "Efficient Communication Schedules for Block-Cyclic Array Redistribution between Processor Sets," *Technical Report,* University of Southern California, Dept. of EE-Systems, Jan. 1998.

[20] S. Ramaswamy and P. Banerjee, "Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers," *Proc. 5th Symposium on Frontiers of Massively Parallel Computation,* Feb. 1995, pp. 342-349.

[21] R. Thakur, A. Choudhary, and G. Fox, "Runtime Array Redistribution in HPF Programs," *Proc. Scalable High Performance Computing Conference,* May 1994, pp. 309-316.

[22] R. Thakur, A. Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution," *IEEE Transactions on Parallel and Distributed Systems,* (7)6:587-594, June 1996.

[23] D.W. Walker and S.W. Otto, "Redistribution of Block-Cyclic Data Distributions using MPI," *Technical Report,* ORNL/TM-12999, Oak Ridge National Labs, June 1995.

[24] C.-L. Wang, P.B. Bhat, and V.K. Prasanna, "High-Performance Computing for Vision," *Proceedings of the IEEE,* (84)7: pp. 931-946, July 1996.

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