Multi-Phase Redistribution: A Communication-Efficient Approach to Array Redistribution[‡]

S. D. Kaushik[1], C.-H. Huang[1], J. Ramanujam[2], and P. Sadayappan[1]

[1]Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
({kaushik,chh,saday}@cis.ohio-state.edu)

[2]Department of Electrical and Computer Engineering
Louisiana State University
Baton Rouge, LA 70803
(jxr@gate.ee.lsu.edu)

Abstract

Distributed-memory implementations of several scientific applications require array redistribution. Array redistribution is used in languages such as High Performance Fortran to dynamically change the distribution of arrays across processors. Performing array redistribution incurs two overheads - an *indexing overhead* for determining the set of processors to communicate with and the array elements to be communicated, and a *communication overhead* for performing the necessary irregular all-to-many personalized communication. In this paper efficient runtime methods for performing array redistribution are presented. To reduce the indexing overhead, precise closed forms for enumerating the processors to communicate with and the array elements to be communicated are developed for two special cases of array redistribution involving block-cyclically distributed arrays. The general array redistribution problem for block-cyclically distributed arrays can be expressed in terms of these special cases. Using the developed closed forms, a distributed algorithm for scheduling the irregular communication for redistribution is developed. The generated schedule eliminates node contention and incurs the least communication overhead. The scheduling algorithm has an asymptotically lower scheduling overhead than techniques presented in the literature. Based on the developed closed forms, a cost model for estimating the communication and the indexing overhead for array redistribution is developed. Using this model, a multi-phase approach for reducing the communication cost for array redistribution is presented. The key idea is to perform the redistribution as a sequence of redistributions such that the total cost of the sequence is less than that of direct redistribution. Algorithms for determining the sequence of intermediate data distributions which minimizes the total redistribution time are developed. Extensions of the developed closed forms and algorithms to perform array redistribution of multi-dimensional arrays are presented. Experimental results on the Cray T3D and IBM-SP2 demonstrate the validity of the developed cost models and the efficacy of the multi-phase approach for array redistribution.

Keywords: Array Redistribution, Communication Scheduling, Distributed-Memory Computers, Data Distribution, Data Communication, High Performance Fortran, Node Contention.

[‡]J. Ramanujam is supported in part by an NSF Young Investigator Award CCR-9457768, an NSF grant CCR-9210422, and by LEQSF.

1 Introduction

The difficulty of programming distributed-memory machines has been recognized as a major obstacle to their widespread acceptance. To address this problem, languages such as High Performance Fortran (HPF) [6, 17], Fortran D [7], and Vienna Fortran [4] have been defined to be portable parallel programming platforms for such machines. These languages provide the programmer with a single address space and allow annotations to specify the mapping of arrays among the processors of the distributed-memory machine. The block-cyclic distribution is the most general regular data distribution supported in these languages. Array redistribution is used to dynamically change the distribution of an array from a specified source to a specified target distribution. Efficient distributed-memory implementations of several scientific applications including the multi-dimensional fast Fourier transform [28] and the Alternate Direction Implicit (ADI) method for solving two-dimensional diffusion equations require array redistribution. Array redistribution is also required when the distributed arrays passed as arguments to runtime libraries or subroutines have distributions different from those of the formal parameters. Thus efficient methods for performing array redistribution are of great importance for the development of distributed-memory compilers for these languages.

Performing array redistribution incurs an *indexing overhead* for identifying the processors involved in the communication and the array elements to be communicated. Specifically each processor p has to determine the following processor and data index sets:

- *Send processor set* of p: set of processors to which p has to send data.
- *Send data index set* of p to processor q: indices of the array elements which are resident on p under the source distribution but are resident on q after redistribution.
- *Receive processor set* of p: set of processors from which p has to receive data.
- *Receive data index set* of p from processor q: indices of the array elements which are resident on q under the source distribution but are resident on p after redistribution.

A closed form characterization of these sets would reduce the indexing overhead for packing data into messages on the sending processor and unpacking data at the receiving processor. Such a characterization in terms of simple regular sections is possible for array redistributions restricted to block and cyclically distributed arrays [12, 16]. However, in general for the block-cyclic distribution, such a closed form characterization is not possible.

Several methods for performing the indexing for array redistribution have been presented in the literature. Closed form expressions for determining the processor and data index sets for redistributing block and cyclically distributed arrays were developed in [12, 16]. A virtual processor approach for performing redistributions involving block-cyclically distributed arrays was proposed in [12]. The virtual processor approach requires additional memory and computation at runtime to efficiently enumerate the processor and data index sets. Efficient runtime strategies for performing redistribution for special cases involving block-cyclically distributed arrays were presented in [27]; these strategies are applicable to array redistributions in which the block size of the source block-cyclic distribution is a multiple of the block size of the target block-cyclic distribution and vice versa. For the general redistribution problem involving block-cyclically distributed arrays, an approach which incurs a significant indexing overhead per array element was proposed [27]. An array redistribution algorithm which can handle arbitrary source and target processor sets and block-cyclic distributions was presented in [19]. Similar to the virtual processor approach presented in [12], this approach, in general, incurs an additional runtime overhead for solving linear diophantine equations. In this paper, we develop closed form expressions for the processor and data index sets for some special cases of array redistribution involving block-cyclically distributed arrays. The solution to the general array redistribution problem for block-cyclically distributed arrays can be expressed in terms of these special cases. The developed closed forms provide a precise representation for the index sets scanned using the algorithms presented in [27].

Performing array redistribution also incurs a *communication overhead* arising from the *all-to-many personalized* communication among the processors. A processor communicates with several other processors sending distinct array elements to each processor. Several methods for performing the *complete exchange*—a special case of the all-to-many communication in which each processor communicates with every other processor—have been presented in the literature [2, 3, 9, 13, 23, 24, 25]. A multi-phase approach for complete exchange on hypercube-connected multicomputers was presented in [2]. The underlying idea of the multi-phase approach is to combine data to be sent to a set of processors into one message and indirectly send it through this set of processors; each processor appropriates the segment destined to it from this message, repackages the message along with its local data for one of the destination processors and sends it to the next processor in the set. The use of the combining strategy reduces the number of message startups required for the complete exchange but incurs an additional data transmission cost as a data element has to be packed, transmitted and unpacked multiple times. Similar *combining* or *store-and-forward* schemes for two-dimensional meshes were presented in [3, 9, 24, 25]. The issue of scheduling the irregular all-to-many personalized communication for array redistribution has received relatively little attention. Static and run-time scheduling techniques to reduce node and link contention while performing general irregular all-to-many personalized communication were presented in [20, 22]. In the runtime techniques presented therein, each processor constructs a global communication matrix and replicates the computation for scheduling the communication using this matrix. Few methods have exploited the existence of patterns in the irregular communication for array redistribution or developed a multi-phase scheduling scheme to reduce the communication cost for array redistribution. Techniques for scheduling the communication which utilize the fact that the all-to-many communication arises from special forms of array redistribution, were developed in [14]. A communication cost model for redistribution and a multi-phase approach to reduce the communication cost were also developed. In the context of reducing communication cost for array redistribution, Strip-Mining Redistribution, a method in which portions of the array are redistributed in sequence, as opposed to redistributing the entire array in one step to obtain good communication-computation overlap, was presented in [29, 30].

In this paper, we extend the multi-phase approach for array redistribution which was developed in [14]. The underlying idea of the multi-phase approach is to perform array redistribution as a sequence of redistributions so that the total cost of the sequence is lower than the cost of the direct redistribution. Techniques developed in [14] are applicable when the source and the target block-cyclic distributions satisfy certain conditions. Recently, redistribution routines for the special cases when either the block size of the source block-cyclic distribution is a multiple of the block size of the target block-cyclic distribution or vice versa, were presented in [27]. Based on the ideas presented therein, we extend the multi-phase approach presented in [14] to handle arbitrary source and target block-cyclic distributions.

The primary contributions of this paper are as follows. A precise closed form characterization of the processor and data index sets for the special cases when either the source block size is a multiple of the target block size or vice versa, is developed. This closed form characterization facilitates the development of a communication and indexing cost model for array redistribution and a distributed scheduling algorithm which eliminates node contention for the all-to-many personalized communication. The algorithm has an asymptotically lower communication and scheduling overhead than techniques presented in [20, 22]. Using the developed cost model, a multi-phase strategy for performing array redistribution for arbitrary source and target block-cyclic distributions is developed. Algorithms for determining the sequence of intermediate array distributions which minimizes the redistribution time are presented. Extensions of the closed forms and algorithms to perform array redistributions involving multi-dimensional arrays are described. Extensive experimental verification on two state-of-the-art distributed-memory machines—the Cray T3D and the IBM-SP2—demonstrate the efficacy of the multi-phase approach.

The paper is organized as follows. Section 2 briefly describes the regular data distributions supported in HPF and presents a mixed radix representation for the data distributions and index sets for array redistribution. In Section 3, closed forms for the send and receive processor and data index sets for cases when the source block size is a multiple of the target block size and vice versa are presented. A distributed scheduling algorithm and a cost model for the all-to-many personalized communication are developed in Section 4. In Section 5, an indexing cost model developed using the closed forms is presented. Section 6 describes the multi-phase approach and provides algorithms for determining the optimal sequence of intermediate distributions. Experimental results on the Cray T3D and IBM-SP2 are presented in Section 7. Section 8 concludes with a summary and directions for future work.

2 Regular Data Distributions and Array Redistribution

In this section we briefly describe the regular data distributions supported by HPF and present a mixed radix representation for the regular data distributions.

2.1 Data Distributions and Array Redistribution in HPF

HPF supports a two-level mapping of data arrays to the processors of a distributed-memory machine. The language

Table 1: Index mapping functions for regular data distributions.

	BLOCK	CYCLIC	CYCLIC(b)
local to global	$k = p(\lceil N/P \rceil) + l$	$k = lP + p$	$k = (l \text{ div } b)bP + bp + l \bmod b$
global to local	$l = k \bmod \lceil N/P \rceil$	$l = k \text{ div } P$	$l = (k \text{ div } Pb)b + k \bmod b$
global to proc	$p = k \text{ div } \lceil N/P \rceil$	$p = k \bmod P$	$p = (k \text{ div } b) \bmod P$

k global index $0 \leq k \leq N-1$; l local index $0 \leq l < \lceil b * \left(\lceil N/(Pb) \rceil \right)$; p processor $0 \leq p < P$.

introduces a Cartesian grid referred to as a *template*. Arrays are *aligned* with the template using an ALIGN directive and the template is *distributed* among the processors using a DISTRIBUTE directive. Consider an array $A(0 : N-1)$ aligned with a template D such that array element $A(k)$ is aligned with template cell $D(ak+c)$. The template is distributed among P processors using a CYCLIC(b) distribution, such that template cell $D(i)$ is assigned to processor $(i \text{ div } b) \bmod P$. This corresponds to partitioning the template into blocks of size b, which are cyclically assigned to P processors. Effectively, the array A is divided into P arrays, each residing in the local memory of one of the processors. The CYCLIC(b) distribution—referred to as the *block-cyclic* distribution—is the most general regular data distribution supported by HPF. The BLOCK distribution corresponds to a CYCLIC $\left(\lceil \frac{N}{P} \rceil \right)$ distribution while the CYCLIC distribution corresponds to a CYCLIC(1) distribution. An element $A(k)$ of the array A has a *global* and a *local* index. The global index k is the index in the array A as referenced in an HPF program. The portion of the array A located in a processor's local memory is referred to as *A_loc* in the processor's node program generated by an HPF compiler. The index of element $A(k)$ in *A_loc* is its local index. In this paper, we focus on identity alignments, i.e., $A(k)$ is aligned with template cell $D(k)$. Techniques developed in [11] can be used for the array redistribution in the general case. For identity alignments, the relationships between the global index, the local index and the processor index for regular data distributions of a one-dimensional array are shown in Table 1. The indexing for arrays A and *A_loc* begins at zero and the processors are numbered from 0 to $P-1$. Multi-dimensional arrays are distributed by independently specifying the alignments along each dimension onto a multi-dimensional template which is distributed onto a multi-dimensional virtual processor array by specifying the distribution along each dimension. An HPF code segment illustrating the distribution of a two-dimensional array onto a two-dimensional processor mesh is shown in Fig. 1. The PROCESSORS directive declares a 2×2 virtual processor array while the DISTRIBUTE directive distributes the array A(100,100) onto the virtual processor array with a BLOCK distribution along dimension one and a CYCLIC(2) distribution along dimension two.

Array redistribution implies a change in the distribution of the array. For instance, consider an array $B(0 : 14)$ distributed on three processors using a BLOCK distribution. The redistribution of B to a cylic distribution is illustrated in Fig. 2. Fig. 1, illustrates the syntax of the REDISTRIBUTE command. After redistribution, array elements located on a processor under the source distribution may be located on a different processor under the target distribution. Thus a processor has to determine the set of array elements to send to and receive from other processors. A naive strategy for performing redistribution is as follows: each processor scans its local array, determines the global index

```
        REAL A(100,100)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE (BLOCK,CYCLIC(2)) ONTO P::  A
!HPF$ DYNAMIC A
   ...
          FORALL (i=2:98, j=2:98) A(i,j) = A(i-1,j-1) + A(i+1,j+1)
   ...
!HPF$ REDISTRIBUTE A(CYCLIC(2),BLOCK)
```

Figure 1: HPF program segment.

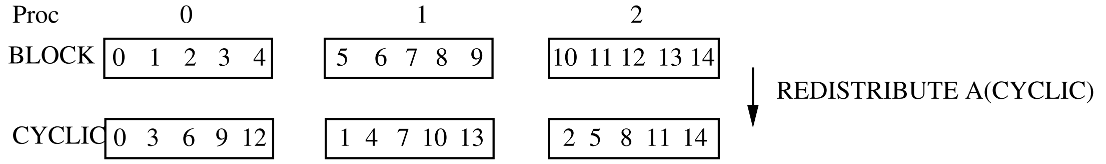

Figure 2: Redistribution of array $A(0:14)$ from BLOCK to CYCLIC distribution.

from the local index for the source distribution, determines the destination processor for that array element under the target distribution and inserts the element into a buffer reserved for that destination processor. After all the local array elements are scanned and inserted into buffers, a possibly empty buffer is sent to and received from every processor other than itself. The unpacking phase is symmetric to the packing phase. This method involves multiple conversions between global, local, and processor indices for every local array element, which has an unacceptably high indexing and communication overhead. An efficient method for performing redistribution is based on using data and processor index sets. Each processor $p,\ 0 \leq p < P$ determines for every processor $q,\ 0 \leq q < P$, the following data index sets:

- $DSend(p,q)$: the set of local indices on p of the array elements which are located on p under the source distribution but are located on q under the target distribution.

- $DRecv(p,q)$: the set of local indices on p of the array elements which are located on q under the source distribution but are located on p under the target distribution.

Two additional sets of processor indices are defined as follows:

- $PSend(p)$: the set of processors $\{q \mid DSend(p,q) \neq \emptyset,\ 0 \leq q < P\}$

- $PRecv(p)$: the set of processors $\{q \mid DRecv(p,q) \neq \emptyset,\ 0 \leq q < P\}$

Using these index sets the node program pseudo-code for array redistribution is as shown in Fig. 3. Note that no particular order of the message sends and receives is implied in Fig. 3. Also if $p \in PSend(p)$, then the message send and receive can be replaced by a local memory copy operation. For array redistributions restricted to BLOCK and CYCLIC distributions, the processor and data index sets can be expressed as simple regular sections [12, 16]. However, in general, for block-cyclic distributions the index sets cannot be expressed as regular sections. In this paper, we

```
/* On each processor p: */
    /* Sending phase */
    for q ∈ PSend(p)
        pack A_loc(DSend(p, q)) into Tmp; send Tmp to q;
    endfor

    /* Receiving phase */
    for q ∈ PRecv(p)
        recv Tmp from q; A_loc(DRecv(p, q)) = Tmp(0 : |DRecv(p, q)| − 1)
    endfor
```

Figure 3: Array redistribution using index sets: node program for processor p.

develop closed form expressions for special cases of array redistribution involving block-cyclic arrays. The general case of array redistribution can be expressed in terms of the special cases.

2.2 Tensor Product Representation

We use the tensor product representation as a mixed radix representation for regular data distributions to develop and represent the index sets for redistribution. We provide a brief overview of the tensor product representation for regular data distributions. The reader is referred to [10] for details.

Let A be an $m \times n$ matrix and B a $p \times q$ matrix. The tensor product of A and B is the block matrix obtained by replacing each element $a_{i,j}$ of matrix A by $a_{i,j}B$. $A \otimes B$ is the $mp \times nq$ matrix defined as

$$A_{m\times n} \otimes B_{p\times q} = \begin{bmatrix} a_{0,0}B & \cdots & a_{0,n-1}B \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B & \cdots & a_{m-1,n-1}B \end{bmatrix}_{mp\times nq}.$$

We use the tensor product of vector bases to represent regular data distributions. A *vector basis* of size m consists of basis vectors e_k^m, $0 \leq k < m$, where e_k^m is a column vector of length m with a one at position k and zeroes elsewhere. We use e_i^m to represent the vector basis of size m. The tensor product $e_i^m \otimes e_j^n$ of two vector bases e_i^m and e_j^n is called a *tensor basis.* From the definition of the tensor product we have $e_i^m \otimes e_j^n = e_{in+j}^{mn}$. Thus the size of the tensor basis $e_i^m \otimes e_j^n$ is mn. Expressing a vector basis e_i^M as the tensor product of vector bases $e_{i_{t-1}}^{m_{t-1}} \otimes \cdots \otimes e_{i_0}^{m_0}$ is called the *factorization* of the vector basis, where $M = m_{t-1} \times \cdots \times m_0$ and $i_k = (i \text{ div } M_{k-1}) \bmod m_k, M_k = \prod_{i=0}^{k} m_i,\ M_{-1} = 1$. For example, some of the possible factorizations of the vector basis e_i^{12} are the tensor basis $e_{i_2}^2 \otimes e_{i_1}^3 \otimes e_{i_0}^2$ where $i_2 = (i \text{ div } 6)$, $i_1 = (i \text{ div } 2) \bmod 3$, and $i_0 = i \bmod 2$ or the tensor basis $e_{i_1}^4 \otimes e_{i_0}^3$ where $i_1 = i \text{ div } 3$ and $i_0 = i \bmod 3$. Factorizing a vector basis in a tensor basis is referred to as factorization of the tensor basis. Expressing a tensor basis $e_{i_{t-1}}^{m_{t-1}} \otimes \cdots \otimes e_{i_0}^{m_0}$ as a vector basis e_i^M, $i = \sum_{k=0}^{t-1} i_k M_{k-1}$ is called *linearization* of the tensor basis. For example, the tensor basis $e_{i_1}^4 \otimes e_{i_0}^3$ can be linearized to give the vector basis $e_{3i_1+i_0}^{12}$. The tensor product basis provides a natural representation for indexing into segmented or multi-dimensional arrays.

2.3 Mixed Radix Representation of Regular Data Distributions

Consider the array $A(0 : N-1)$ distributed over P processors numbered from 0 to $P-1$. We assume that N is a multiple of $P^{\S}$. A mixed radix representation of a regular data distribution of the array A is obtained by associating index k of A with the vector basis element e_k^N and factorizing the vector basis. Associating array indices with the indices of the vector basis helps in identifying the processor index and the local index when the array is partitioned due to the data distribution. For example, under a BLOCK distribution the array is partitioned into segments of size $\frac{N}{P}$. The relationship between the global index k, the processor index p, and the local index l as shown in Table 1 can be represented by the equality

$$e_k^N = e_p^P \otimes e_l^{\frac{N}{P}},$$

where $p = k \text{ div } \frac{N}{P}$ and $l = k \mod \frac{N}{P}$. In the above identity, the index of vector basis e_p^P is associated with the processor index on which element $A(k)$ is located after being distributed using a BLOCK distribution. We distinguish the vector basis corresponding to the processor index by denoting it as ρ and refer to it as a *processor basis*. The index of the vector basis $e_i^{N/P}$ is associated with the local index of $A(k)$ in A_loc. The resulting tensor basis $\rho_p^P \otimes e_l^{N/P}$ provides a mixed radix representation for the BLOCK distribution. A tensor basis in which the vector basis associated with the processor index is denoted as ρ is referred to as a *distribution basis*. Next consider the CYCLIC(B) distribution of array A. The array is divided into N/B blocks of size B each, which are cyclically assigned to P processors. Let $C = \frac{N}{PB}$ be the number of courses (number of blocks) on each processor. The relation between the global index k, the processor index p and the index b within a block is represented by:

$$e_k^N = e_c^C \otimes e_p^P \otimes e_b^B,$$

where $c = k \text{ div } PB$, $p = (k \text{ div } B) \mod P$, and $b = k \mod B$. In the distribution basis, c is the course index and p is the processor index. Note that b does not correspond to the local index l. The local index l is determined by a combination of the course index c and the block index b as shown in Def. 2.1: $l = Bc + b$. We define the indexing functions for the distribution basis which select the vector bases of the distribution which identify the processor index p, the local index l and the global index k of each index of the distributed array. We consider the distribution basis for the block-cyclic distribution as it is the most general regular data distribution.

§ If P does not evenly divide N, then the array is assumed to be padded to $P\lceil\frac{N}{P}\rceil$ elements. No memory is actually allocated for the dummy elements.

Definition 2.1 (Indexing functions) *For a one-dimensional array $A(0:N-1)$ let the distribution basis be $e_c^C \otimes \rho_p^P \otimes e_b^B$, then the global index function* global*, the processor index function* proc*, and the local index function* local *are*

$$\begin{aligned}
\text{global}(e_c^C \otimes \rho_p^P \otimes e_b^B) &= e_{PBc+Bp+b}^N \\
\text{proc}(e_c^C \otimes \rho_p^P \otimes e_b^B) &= \rho_p^P \\
\text{local}(e_c^C \otimes \rho_p^P \otimes e_b^B) &= e_{Bc+b}^{CB}
\end{aligned}$$

Note that since $c = k \text{ div } PB$ and $b = k \mod b$, Def. 2.1 implies the relations between the indices illustrated in Table 1, i.e. $k = PBc + Bp + b$ and $l = Bc + b$.

The distribution basis for a multi-dimensional array can be expressed as a tensor product of the distribution bases for each dimension. Consider an array $A(0 : N_2 - 1, 0 : N_1 - 1)$ stored using a row major order, i.e., element $A(i, j)$ has an index $iN_1 + j$ in linear memory. Since $e_i^{N_2} \otimes e_j^{N_1} = e_{iN_1+j}^{N_2N_1}$, the index (i, j) for the two-dimensional array can be associated with the tensor basis index $e_i^{N_2} \otimes e_j^{N_1}$. Similarly the distribution of each dimension can be independently specified by expressing the vector bases $e_i^{N_2}$ and $e_j^{N_1}$ as distribution bases. Thus the distribution basis for a multi-dimensional array can be represented as the tensor product of the distribution bases for each dimension. For instance, the $(\texttt{CYCLIC}(B_2), \texttt{CYCLIC}(B_1))$ distribution of the $N_2 \times N_1$ array A onto a $P_2 \times P_1$ processor mesh is represented by the distribution basis

$$[e_{c_2}^{C_2} \otimes \rho_{p_2}^{P_2} \otimes e_{b_2}^{B_2}] \otimes [e_{c_1}^{C_1} \otimes \rho_{p_1}^{P_1} \otimes e_{b_1}^{B_1}],$$

where $C_2 = \frac{N_2}{P_2B_2}$ and $C_1 = \frac{N_1}{P_1B_1}$ are the number of courses for block-cyclic distribution along the first and second dimension, respectively. The square parenthesis separate the distribution bases corresponding to each dimension. The indexing functions for the distribution basis of a multi-dimensional array can be defined in terms of those for one-dimensional arrays.

Definition 2.2 (Indexing functions) *For a multi-dimensional array $A(0 : N_{t-1} - 1, \ldots, 0 : N_0 - 1)$, with a distribution basis δ_i for the distribution along dimension i, the global index function* global*, the processor index function* proc *and the local index function* local *are defined as follows:*

$$\begin{aligned}
\text{global}([\delta_{t-1}] \otimes \cdots \otimes [\delta_0]) &= [\text{global}(\delta_{t-1})] \otimes \cdots\cdots \otimes [\text{global}(\delta_0)] \\
\text{proc}([\delta_{t-1}] \otimes \cdots \otimes [\delta_0]) &= [\text{proc}(\delta_{t-1})] \otimes \cdots\cdots \otimes [\text{proc}(\delta_0)] \\
\text{local}([\delta_{t-1}] \otimes \cdots \otimes [\delta_0]) &= [\text{local}(\delta_{t-1})] \otimes \cdots\cdots \otimes [\text{local}(\delta_0)]
\end{aligned}$$

Array redistribution corresponds to a remapping of the processor basis in the distribution basis. If δ_s is the initial distribution basis and δ_t is the distribution basis after the redistribution then $global(\delta_s) = global(\delta_t)$, i.e., the global index is unchanged. However, $proc(\delta_s)$ and $proc(\delta_t)$ may be different, which implies that redistribution involves communication. Also, $local(\delta_s)$ and $local(\delta_t)$ may be different. We refer to δ_s as the source basis, and to δ_t as the target basis. For example, $\delta_s = \rho_p^4 \otimes e_l^4$ represents a `BLOCK` distribution of $A(0 : 15)$ on four processors. Remapping the processor basis gives the new distribution basis $\delta_t = e_p^4 \otimes \rho_l^4$ which corresponds to a `CYCLIC` distribution on four processors. Clearly $global(\delta_s) = global(\delta_t) = e_{4p+l}^{16}$. But $proc(\delta_s) = \rho_p^4$ and $proc(\delta_t) = \rho_l^4$, which implies that communication may be required since p and l can be different for a global array index k. Also $local(\delta_s) = e_l^4$ and $local(\delta_t) = e_p^4$ which implies that the local index of an element can be different after redistribution.

We represent the processor and data index sets for array redistribution in terms of simple regular sections or a tensor basis augmented with ranges for the indices of the vector bases. A regular section $(a : b : s)$ represents the indices $\{i \mid i = a + js,\ 0 \leq j < \lfloor \frac{b-a}{s} \rfloor\}$. A tensor product basis of the form $e_*^C \otimes e_p^b$ represents all the indices $\{i \mid i = i_1 b + p,\ 0 \leq i_1 < C\}$. The symbol $*$ is used to represent all the index values from 0 to $C - 1$ where C is the size of the corresponding vector basis. In the above example, the index i_1 iterates over all the values represented by the $*$ symbol. The set of indices represented by a tensor product basis can be enumerated by a perfect loop nest. A **for** loop is generated for every vector basis in the tensor basis with either a $*$ as index or an index for which a range is specified. If a $*$ is specified, the lower loop bound is 0 and the upper loop bound is equal to one less than the size of the correponding vector basis. If a range is specified then the loop scans the specified range. For example, the indices i specified by $e_*^C \otimes e_y^Y \otimes e_*^b,\ y \in (0 : Y - 1 : P)$ can be enumerated by the following loop nest:

for $i_1 = 0, C - 1$ /* loop for e_*^C */
  for $y = 0, Y - 1, P$ /* loop for $e_y^Y,\ y \in (0 : Y - 1 : P)$ */
    for $i_2 = 0, b - 1$ /* loop for e_*^b */
      $i = i_1 * Y * b + y * b + i_2$
endfor endfor endfor

We now develop the processor and data index sets for two special cases of array redistribution involving block-cyclically distributed arrays.

3 Array Redistribution

In this section, we develop the processor and data index sets for the special cases when the block size of the source block-cyclic distribution is a multiple of the block size of the target block-cyclic distribution and vice versa. We first consider one-dimensional arrays and then describe the extensions for multi-dimensional arrays.

3.1 Array Redistribution of One-Dimensional Arrays

In the following sections we consider an array $A(0:N-1)$ distributed onto P processors.

3.1.1 Array Redistribution from a CYCLIC(Yt) to a CYCLIC(t) distribution

Let the source distribution be CYCLIC(Yt) and the target distribution be CYCLIC(t). Let $C = \frac{N}{PYt}$ be the number of courses in the source block-cyclic distribution. The send processor and data index sets are as follows.

Theorem 3.1 *For an array redistribution from a* CYCLIC(Yt) *distribution to a* CYCLIC(t) *distribution, the send processor set for processor p is*

$$PSend(p) = \begin{cases} ((pY) \mod P : (pY+Y-1) \mod P) & \text{if } Y < P \\ (0:P-1) & \text{if } Y \geq P. \end{cases}$$

For a processor $q \in PSend(p)$, the send data set from processor p to processor q is

$$DSend(p,q) = \begin{cases} e^C_* \otimes e^Y_{y_1} \otimes e^t_* \quad \text{where } y_1 = (q-pY) \mod P & \text{if } Y < P \\ e^C_* \otimes e^Y_{y_1} \otimes e^t_* \quad \text{where } y_1 = ((q-pY) \mod P : Y-1 : P) & \text{if } Y \geq P. \end{cases}$$

Proof: The source distribution is represented by the distribution basis $\delta_s = e^C_{c_1} \otimes \rho^P_{p_1} \otimes e^Y_{y_1} \otimes e^t_{t_1}$ and the target distribution basis is $\delta_t = e^C_{c_2} \otimes e^Y_{y_2} \otimes \rho^P_{p_2} \otimes e^t_{t_2}$. Note that the trailing vector basis has to be further factorized to facilitate the following analysis. Redistribution requires that $global(\delta_s) = global(\delta_t)$ which gives the equality

$$\begin{aligned} c_1 * PYt + p_1 * Yt + y_1 * t + t_1 &= c_2 * PYt + y_2 * Pt + p_2 * t + t_2 \\ \text{or } \; c_1 * PYt + (p_1 * Y + y_1) * t + t_1 &= c_2 * PYt + (y_2 * P + p_2) * t + t_2. \end{aligned}$$

Since $0 \leq t_1, t_2 < t$ and $0 \leq c_1, c_2 < C$, we have $c_1 = c_2$, $t_1 = t_2$ which implies that $p_1 * Y + y_1 = y_2 * P + p_2$.

To determine the send processor set for a processor p_1, we need to determine all the unique values for $p_2 = (y_1 + p_1 * Y) - y_2 * P$ where $0 \leq y_1, y_2 < Y$. Consider the case when $Y < P$. Since $0 \leq p_2 < P$ and $Y < P$, we have $p_2 = (y_1 + p_1 * Y) - y_2 * P = (y_1 + p_1 * Y) \mod P$, $0 \leq y_1 < Y$. Thus $PSend(p_1) = ((p_1 * Y) \mod P : (p_1 * Y + Y - 1) \mod P)$, if $Y < P$. Note that $PSend(p_1)$ may include a wrap-around depending on whether $(p_1 * Y) \mod P > (p_1 * Y + Y - 1) \mod P$, i.e., the set of processors in $PSend(p_1)$ will be $((p_1 * Y) \mod P : P-1) \cup (0 : (p_1 * Y + Y - 1) \mod P)$.

Consider the case when $Y \geq P$. Let $x = p_1 Y \mod P$. Since $p_1 Y < PY$ and $0 \leq y_2 < Y$, there exists a y_2', such that $p_1 Y - y_2' P = x$. Fig. 4 illustrates the relation among the values. Since $p_2 = y_1 + (p_1 * Y - y_2 * P)$, p_2 can take values from x to $P-1$ for $y_2 = y_2'$ and $y_1 \in (0 : P-x-1)$. Similarly p_2 can take values from 0 to $x-1$ for $y_2 = y_2' - 1$ and $y_1 \in (P-x : P-1)$. Thus when $Y \geq P$, $PSend(p_1) = (0 : P-1)$.

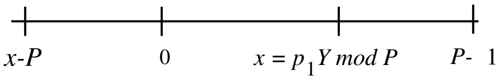

Figure 4: CYCLIC(Yt) to CYCLIC(t) redistribution: Construction of $PSend(p)$ for $Y \geq P$.

We now determine the send data index set from a processor p_1 to a processor p_2. We obtain all the unique values for y_1 such that $y_1 = y_2 * P + (p_2 - p_1 * Y)$. The smallest non-negative value for y_1 is given by $y_1 = (p_2 - p_1 * Y) \mod P$. If $Y < P$, then there is just a single value for y_1. If $Y \geq P$, then $y_1 \in ((p_2 - p_1 * Y) \mod P : Y - 1 : P)$. Since the data send index set consists of local indices on processor p_1, $DSend(p_1, p_2) = local(e_*^C \otimes e_{p_1}^P \otimes e_{y_1}^Y \otimes e_*^t) = e_*^C \otimes e_{y_1}^Y \otimes e_*^t$. Thus we have

$$DSend(p,q) \quad = \quad \begin{cases} e_*^C \otimes e_{y_1}^Y \otimes e_*^t \quad \text{where } y_1 = (q - pY) \mod P & \text{if } Y < P \\ e_*^C \otimes e_{y_1}^Y \otimes e_*^t \quad \text{where } y_1 = ((q - pY) \mod P : Y - 1 : P) & \text{if } Y \geq P. \end{cases}$$

□

Note that the processor send set may involve a wrap-around at $P - 1$ if $(pY) \mod P > (pY + Y - 1) \mod P$. The receive processor and data index sets for a CYCLIC(Yt) to CYCLIC(t) array redistribution are given by 3.2 shown below.

Theorem 3.2 *For an array redistribution from a* CYCLIC(Yt) *distribution to a* CYCLIC(t) *distribution, the receive processor set for a processor p is*

$$PRecv(p) \quad = \quad \begin{cases} (p : YP - 1 : P) \text{ div } Y & \text{if } Y < P \\ (0 : P - 1) & \text{if } Y \geq P. \end{cases}$$

For a processor $q \in PRecv(p)$, the receive data index set is

$$DRecv(p,q) \quad = \quad \begin{cases} e_*^C \otimes e_{y_2}^Y \otimes e_*^t \quad \text{where } y_2 = \left\lceil \frac{qY-p}{P} \right\rceil & \text{if } Y < P \\ e_*^C \otimes e_{y_2}^Y \otimes e_*^t \quad \text{where } y_2 = \left(\left\lceil \frac{qY-p}{P} \right\rceil : \left\lfloor \frac{qY-p+Y-1}{P} \right\rfloor \right) & \text{if } Y \geq P. \end{cases}$$

Proof: Following an argument similar to Theorem 3.1, we have $c_1 = c_2$, $t_1 = t_2$ and $p_1 * Y + y_1 = y_2 * P + p_2$. For the receive processor set, we need to determine the unique values for p_1, where $p_1 * Y + y_1 = y_2 * P + p_2 = ((y_2 * P + p_2) \text{ div } Y) * Y + (y_2 * P + p_2) \mod Y$. It follows that $p_1 = (y_2 * P + p_2) \text{ div } Y$, $0 \leq y_2 < Y$. Thus we have $p_1 \in (p_2 : YP - 1 : P) \text{ div } Y$. When $Y \geq P$ the set $(p_2 : YP - 1 : P) \text{ div } Y$ includes all processors $(0 : P - 1)$.

To determine the receive data index set for a processor p_2 from a processor p_1, we need to determine the set of all unique y_2 such that $y_2 * P = (p_1 * Y - p_2) + y_1$, $0 \leq y_1 < Y$. The smallest non-negative value for y_2 is $\left\lceil \frac{p_1 * Y - p_2}{P} \right\rceil$. When $Y < P$, there is a single value for y_2. If $Y \geq P$, y_2 can take values in the range $\left\lceil \frac{p_1 * Y - p_2}{P} \right\rceil : \left\lfloor \frac{p_1 * Y - p_2 + Y - 1}{P} \right\rfloor$. Since, the data receive index set consists of local indices on processor p_2, $DRecv(p_2, p_1) = local(e_*^C \otimes e_{p_2}^P \otimes e_{y_2}^Y \otimes e_*^t) = e_*^C \otimes e_{y_2}^Y \otimes e_*^t$. Thus we have

$$DRecv(p,q) = \begin{cases} e_*^C \otimes e_{y_2}^Y \otimes e_*^t \quad \text{where } y_2 = \left\lceil \frac{qY-p}{P} \right\rceil & \text{if } Y < P \\ e_*^C \otimes e_{y_2}^Y \otimes e_*^t \quad \text{where } y_2 = \left(\left\lceil \frac{qY-p}{P} \right\rceil : \left\lfloor \frac{qY-p+Y-1}{P} \right\rfloor \right) & \text{if } Y \geq P. \end{cases}$$

□

3.1.2 Array Redistribution from a CYCLIC(s) to a CYCLIC(Ys) distribution

Let the source distribution be CYCLIC(s), the target distribution be CYCLIC(Ys), and let $C = \frac{N}{PYs}$ denote the number of courses in the source block-cyclic distribution. The processor and data index sets for a CYCLIC(s) to CYCLIC(Ys) redistribution can be determined in a manner similar to that for a CYCLIC(Yt) to a CYCLIC(t) redistribution. The send processor and data index sets for a redistribution from a CYCLIC(s) to a CYCLIC(Ys) distribution are equivalent to the receive processor and data index sets for the CYCLIC(Yt) to a CYCLIC(t) redistribution, respectively and vice versa. We provide the processor and data index sets without proofs.

Theorem 3.3 *For an array redistribution from a* CYCLIC(s) *distribution to a* CYCLIC(Ys) *distribution, the send processor set for a processor p is*

$$PSend(p) = \begin{cases} (p : PY-1 : P) \text{ div } Y & \text{if } Y < P \\ (0 : P-1) & \text{if } Y \geq P. \end{cases}$$

For a processor $q \in PSend(p)$, the send data set from p to q is

$$DSend(p,q) = \begin{cases} e_*^C \otimes e_{y_1}^Y \otimes e_*^s \quad \text{where } y_1 = \left\lceil \frac{qY-p}{P} \right\rceil & \text{if } Y < P \\ e_*^C \otimes e_{y_1}^Y \otimes e_*^s \quad \text{where } y_1 = \left(\left\lceil \frac{qY-p}{P} \right\rceil : \left\lfloor \frac{qY-p+Y-1}{P} \right\rfloor \right) & \text{if } Y \geq P. \end{cases}$$

Theorem 3.4 *For an array redistribution from a* CYCLIC(s) *distribution to a* CYCLIC(Ys) *distribution, the receive processor set for a processor p is*

$$PRecv(p) = \begin{cases} ((pY) \bmod P : (pY+Y-1) \bmod P) & \text{if } Y < P \\ (0 : P-1) & \text{if } Y \geq P. \end{cases}$$

For a processor $q \in PRecv(p)$, the receive data set from p to q is

$$DRecv(p,q) = \begin{cases} e_*^C \otimes e_{y_2}^Y \otimes e_*^s \quad \text{where } y_2 = (q-pY) \bmod P & \text{if } Y < P \\ e_*^C \otimes e_{y_2}^Y \otimes e_*^s \quad \text{where } y_2 = ((q-pY) \bmod P : Y-1 : P) & \text{if } Y \geq P. \end{cases}$$

We now develop the index sets for redistribution of multi-dimensional arrays.

3.2 Array Redistribution of Multi-dimensional Arrays

The redistribution techniques for multi-dimensional arrays differ based on whether the shape of the data and processor array change during redistribution. We have the following cases:

- *Shape Preserving Redistribution* : These routines are applicable when the shape of the data and processor array are not changed during the array redistribution, i.e., the redistribution only involves changing the distribution along the dimensions of the array. For instance, the redistribution considered in the program segment in Fig. 1 is a shape preserving redistribution. The index sets for the multi-dimensional array redistribution are developed using those developed for the one-dimensional arrays.

- *Shape Modifying Redistribution* : In general, when either the data or processor array is reshaped during redistribution, techniques developed in [12] are required to efficiently enumerate the index sets. However, for particular source and target distributions it is possible to develop closed form expressions for the index sets. Techniques for these are addressed in [15].

We develop the index sets for the shape preserving redistribution. Consider an array $A(0 : N_{t-1}-1, \ldots, 0 : N_0-1)$ distributed onto a $P_{t-1} \times \cdots \times P_0$ processor mesh using a regular data distribution along each dimension. The distribution basis is of the form $[\delta_{t-1}] \otimes \cdots \otimes [\delta_0]$ where $\delta_i,\ 0 \le i < t$ is the distribution basis for dimension i. After redistribution, let the distribution basis be $[\delta'_{t-1}] \otimes \cdots \otimes [\delta'_0]$. Since the redistribution is shape-preserving, the shape of the data array does not change and the sizes of δ_i and δ'_i are equal. Also the sizes of *proc*(δ_i) and *proc*(δ'_i) are equal as the processor array is not reshaped. Thus the analysis of Section. 3.1 can be independently applied for each dimension to determine the index sets for that dimension. The index sets for the multi-dimensional array redistribution are given by the tensor product of the corresponding index sets for each dimension. For instance, consider a processor $(p_{t-1}, \ldots, p_0),\ 0 \le p_i < P_i$. Let $PSend_i(p_i)$ denote the processor send set corresponding to dimension i of the array. Then the processor send set for processor $(p_{t-1}, \ldots, p_0)$ for the multi-dimensional array redistribution is $PSend_{t-1}(p_{t-1}) \otimes \cdots \otimes PSend_0(p_0)$. The index sets can be scanned using loop nests as shown in Section. 2. Thus the node code on processor $(p_{t-1}, \ldots, p_0)$ for the multi-dimensional array redistribution is as shown in Fig. 5. Note that for the array dimensions i which are not distributed, $PSend_i$ is empty and $DSend$ includes the entire dimension.

4 A Communication Cost Model for Array Redistribution

Modeling the communication cost for the all-to-many personalized communication for array redistribution requires accounting for the message startup and transmission costs and the overhead arising due to node and link contention.

```
/* Sending phase */
for q_{t-1} ∈ PSend_{t-1}(p_{t-1})
    ⋮
    for q_0 ∈ PSend_0(p_0)
        pack A_loc(DSend_{t-1}(p_{t-1}, q_{t-1}), ..., DSend_0(p_0, q_0)) into Tmp; send Tmp to (q_{t-1}, ..., q_0);
endfor ··· endfor

/* Receiving phase */
for q_{t-1} ∈ PRecv_{t-1}(p_{t-1})
    ⋮
    for q_0 ∈ PRecv_0(p_0)
        recv Tmp from (q_{t-1}, ..., q_0);
        A_loc(DRecv_{t-1}(p_{t-1}, q_{t-1}), ..., DRecv_0(p_0, q_0)) = Tmp;
endfor ··· endfor
```

Figure 5: Node program on processor $(p_{t-1}, \ldots, p_0)$ for multi-dimensional array redistribution.

For worm-hole routed and circuit-switched networks, the cost of transmitting a large message is nearly independent of the distance between the source and target nodes [18]. *Node contention* arises as a processor can typically receive a single message at a time. *Link contention* arises when two or more message paths need to share the same interconnection network edge. Node contention can be avoided by scheduling the all-to-many personalized communication as a sequence of permutations. For several current architectures with adaptive/randomized wormhole routing, the impact of node contention on the communication cost is more significant than that of link contention [21]. For such architectures, it is reasonable to model the cost of all-to-many personalized communication by factoring the communication into a sequence of permutations and determining the cost of each permutation.

In the absence of link contention, the cost of transmitting the largest message in a permutation gives the approximate cost of the permutation. If the message startup cost is t_s, the network bandwidth is $\frac{1}{t_e}$, and the largest message size in the permutation is M, then the cost of the permutation is approximately $t_s + Mt_e$. We first model the communication cost for redistribution of one-dimensional arrays and then extend the model to multi-dimensional arrays.

4.1 Array Redistribution of One-Dimensional Arrays

We model the cost of a redistribution from a CYCLIC(Yt) distribution to a CYCLIC(t) distribution. The cost for the CYCLIC(s) to CYCLIC(Ys) case can be similarly modeled.

4.1.1 CYCLIC(Yt) to CYCLIC(t) Redistribution

Using the send processor and data index sets developed in Theorem 3.1, the total number of message startups required and the volume of data to be communicated by a processor p, we have:

$$|PSend(p)| = \min(P,Y), \quad |DSend(p,q)| \approx \frac{N}{P\min(P,Y)}, \forall q \in PSend(p).$$

A factoring of the all-to-many communication into permutations will contain at least $\min(P,Y)$ permutations. We show that for the CYCLIC(Yt) to CYCLIC(t) redistribution, $\min(P,Y)$ is a tight lower bound on the number of permutations necessary to perform the communication for redistribution.

Theorem 4.1 *A tight lower bound on the number of permutations required to cover the communication for the* CYCLIC(Yt) *to* CYCLIC(t) *redistribution is* $\min(P,Y)$.

Proof: Construct the $P \times P$ global communication matrix $COMM$ for the CYCLIC(Yt) to CYCLIC(t) redistribution, where $COMM(i,j) = 1$ if processor i sends data to processor j else $COMM(i,j) = 0$. From Theorem 3.1, it follows that each row and each column of $COMM$ has exactly $k = \min(P,Y)$ non-zero entries. From the Birkhoff-Von Neumann Theorem [1], we have

> *If M is an $m \times m$ matrix of nonnegative integers such that all the elements of each row and column add up to k then M is a sum of k permutation matrices.*

Thus $COMM$ can be expressed as the sum of $\min(P,Y)$ permutation matrices. $\square$

Since the largest message size for each permutation is $\frac{N}{P\min(P,Y)}$, the cost of each permutation is $t_s + \frac{N}{P\min(P,Y)}t_e$. The total communication cost for the CYCLIC(Yt) to CYCLIC(t) redistribution is

$$\min(P,Y) * \left(t_s + \frac{N}{P\min(P,Y)}t_e\right) \quad = \quad \min(P,Y)t_s + \frac{N}{P}t_e. \tag{4.1}$$

While Theorem 4.1 demonstrates the existence of a tight lower bound, it does not provide an algorithm for scheduling the communication as a sequence of permutations to attain the lower bound. A graph theoretic framework based on expressing the communication as a bipartite graph and determining complete matchings of this graph [1, 5] can be used to determine the permutations. However, these schemes require additional computation to construct the communication matrix $COMM$, and would incur the added expense of determining k complete matchings. We develop a distributed routing algorithm for scheduling the communication which has $\theta(\min(P,Y))$ overhead. The algorithm uses the additional information obtained by the closed form characterization of the communication sets and avoids the construction of the global communication matrix $COMM$.

When $Y \geq P$, the communication is a complete exchange which can be factorized into P permutations in a straightforward fashion. We focus on the case when $Y < P$. From Theorem 3.1, when $Y < P$, the first processor that

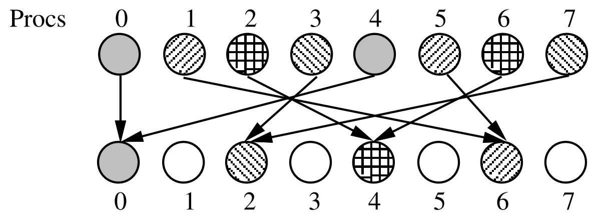

Figure 6: First destination processors for array redistribution of $A(96)$ from CYCLIC(12) to CYCLIC(2) on eight processors.

p communicates with is $(pY) \mod P$. We refer to processors q such that $q = (rY) \mod P$ for some $0 \leq r < P$ as *first destination processors*. After the first destination processor $(pY) \mod P$, processor p communicates with every consecutive processor (with a possible wrap-around at $P-1$) up to and including $(pY+Y-1) \mod P$. If each processor communicates with a distinct first processor, then clearly every consecutive processor it communicates with will be distinct and the all-to-many communication can be performed as a sequence of permutations. However it is possible that several processors communicate with the same first destination processor. For instance, consider an array $A(96)$ distributed on eight processors using a CYCLIC(12) distribution and redistributed to a CYCLIC(2) distribution. The first processor that each processor communicates with is shown in Fig. 6. Lemma 4.1 shows that the processors can be partitioned into groups, such that all processors in a group communicate with the same first destination processor.

Lemma 4.1 *All processors* $p + \frac{k*\text{lcm}(P,Y)}{Y}, \quad 0 \leq k < \text{gcd}(P,Y)$ *have the same first destination processor. The number of such distinct groups is* $\frac{\text{lcm}(P,Y)}{Y}$.

Proof: Follows by noting that $(p + \frac{k*\text{lcm}(P,Y)}{Y}) * Y \mod P = (pY + k*\text{lcm}(P,Y)) \mod P = (pY) \mod P$. Each processor $p,\ 0 \leq p < \frac{\text{lcm}(P,Y)}{Y}$ determines a different group. $\square$

Let $G(p)$ denote the group of processors $p + \frac{k*\text{lcm}(P,Y)}{Y}$ where $0 \leq k < \text{gcd}(P,Y)$. There are totally $\frac{\text{lcm}(P,Y)}{Y}$ such groups, each group identified by a $p \in \left(0 : \frac{\text{lcm}(P,Y)}{Y} - 1\right)$. From Lemma 4.1, it follows that the processors in $G(p)$ communicate with the processors in $(pY \mod P : (pY \mod P + Y - 1) \mod P)$. Clearly not all processors are first destination processors. Lemma 4.2 characterizes the first destination processors.

Lemma 4.2 *All processors communicate first with a processor* r*, such that* $\text{gcd}(P,Y) \mid r$.

Proof: Follows by noting that a processor p first communicates with processor $pY \mod P$, and for some k, $pY \mod P = pY - kP = \text{gcd}(P,Y)\left(\frac{pY}{\text{gcd}(P,Y)} - \frac{kP}{\text{gcd}(P,Y)}\right)$. $\square$

Thus processors which are not multiples of $\text{gcd}(P,Y)$ are not first destination processors. If we ensure that the processors in $G(p)$ communicate with distinct first processors then the communication will be scheduled as a sequence of permutations. Since $G(p)$ contains $\text{gcd}(P,Y) - 1$ processors excluding p and the processors $((pY \mod P + 1) \mod P : (pY \mod P + \text{gcd}(P,Y) - 1) \mod P)$ are not first destination processors, by ensuring that processor $p + k\frac{\text{lcm}(P,Y)}{Y}$ first communicates with processor $pY \mod P + k$ we can guarantee that the communication is performed as a sequence of permutations.

Input: Processor index p, total number of processors P, source block size Yt, target block size t.

Output: Sequence of destination processors for scheduling all-to-many communication as sequence of permutations.

begin
$p_f = p * Y \mod P$; $p_l = (p * Y + Y - 1) \mod P$;
$p_{sf} = p_f + (p * Y)$ div (lcm(P, Y)); $p_s = p_{sf}$;
for $i = 0,\ \min(P, Y) - 1$
  Pack $A\_loc(DSend(p, p_s))$ into tmp_sbuf;
  Send(p_s, tmp_sbuf);**Recv**(p_r, tmp_rbuf);
  Unpack tmp_rbuf into $B\_loc(DRecv(p, p_r))$;
  $p_s = p_s + 1$;
  if $(p_f \le p_l \wedge p_s > p_l)$ **then** $p_s = p_f$;
  else if $(p_f > p_l \wedge p_s = P)$ **then** $p_s = 0$;
  else if $(p_f > p_l \wedge p_s = p_l + 1)$ **then** $p_s = p_f$;
  endif
endfor

Figure 7: Distributed algorithm for performing communication for CYCLIC(Yt) to CYCLIC(t) redistribution.

The scheduling algorithm is shown in Fig. 7. In the algorithm, the send is non-blocking. The blocking receive is performed so as to receive any incoming message and the source processor p_r is determined from the received message. In Fig. 7, the first processor p_{sf} that p communicates with is determined as $pY \mod P + \frac{pY}{\text{lcm}(P,Y)}$. As the computation within the loop is fixed, the complexity of the algorithm is $\theta(\min(P, Y))$.

The scheduling techniques for all-to-many personalized communication presented in [21] address a more general problem—scheduling general irregular all-to-many personalized communication—and hence require the construction of the global communication matrix $COMM$ at runtime. Each processor then replicates the computation for the partitioning of $COMM$ into permutations. The scheduling overhead is $O(PY log^2 Y)$. Also for the all-to-many communication for the CYCLIC(Yt) to CYCLIC(t) redistribution, the techniques presented therein are not guaranteed to partition $COMM$ into exactly $\min(P, Y)$ permutations.

4.1.2 CYCLIC(s) to CYCLIC(Ys) Array Redistribution

The communication cost for the CYCLIC(s) to CYCLIC(Ys) redistribution can be similarly modeled. From Theorem 3.3, we have, for a processor p,

$$|PSend(p)| = \min(P, Y), \quad |DSend(p, q)| \approx \frac{N}{P \min(P, Y)}, \forall q \in PSend(p).$$

Thus every processor communicates with $\min(P, Y)$ processors and the size of each message is approximately $\frac{N}{P \min(P,Y)}$. The communication cost of the array redistribution is determined to be equal to the cost in Eq. 4.1. Since the receive processor set for the CYCLIC(s) to CYCLIC(Ys) redistribution is equivalent to the send processor set for the CYCLIC(Ys) to CYCLIC(s) redistribution, the communication can be similarly partitioned into a sequence of $\min(P, Y)$ permutations using the algorithm in Fig. 7. Some additional computation cost is required for identification

```
p_f = (p * Y) mod P; p_l = (p * Y + Y - 1) mod P;
for p_s = p_f, p_l
    y_1 = (p_s - p * Y) mod P; l = y_1 * t; cnt = 0;
    for c = 0, C - 1
        for i = l, l + t - 1
            buffer[cnt] = A_loc[i]; cnt = cnt + 1;
        endfor
    l = l + Y * t;
endfor
```

Figure 8: Packing data for array redistribution: node program for processor p.

of the inverse. The scheduling algorithm has a complexity of $O(P\min(P,Y))$.

4.2 Array Redistribution of Multi-dimensional Arrays

For shape preserving multi-dimensional array redistribution, the communication cost estimates are obtained by combining the cost estimates for each dimension. The total number of message startups required and the volume of data communicated in a message is the product of the sizes of the processor and data send sets for each dimension, respectively. This observation follows directly from Fig. 5. The distributed routing algorithm is obtained by applying the algorithm in Fig. 7 independently for each dimension to determine the next processor along that dimension to communicate with. Thus for an r-dimensional array $A(0:N_{r-1}-1,\ldots,0:N_0-1)$ which is redistributed such that the redistribution along dimension i is either of the form CYCLIC($Y_i t_i$) to CYCLIC(t_i) or CYCLIC(s_i) to CYCLIC($Y_i s_i$), the communication cost for the redistribution evaluates to

$$\big(\min(P_{r-1},Y_{r-1})\times\cdots\times\min(P_0,Y_0)\big)\,t_s+\left(\frac{N_{r-1}\times\cdots\times N_0}{P_{r-1}\times\cdots\times P_0}\right)t_e \tag{4.2}$$

We now develop an indexing cost model for array redistribution.

5 An Indexing Cost Model for Array Redistribution

In this section, we develop a cost model for the indexing cost for array redistribution. The indexing overhead includes the cost of packing and unpacking the array elements into and from message buffers. This overhead includes a base cost for copying the entire local array into a buffer and back and an additional indexing cost for determining the elements to copy. The base cost is independent of the indexing strategy used and depends only on the processor architecture, while the additional indexing cost is directly dependent on the how good the indexing scheme is. We use the total number of contiguous sections of data elements to be packed into and unpacked from a message buffer as a measure of the additonal indexing cost. Consider the sending phase of the redistribution from a CYCLIC(Yt) to a CYCLIC(t) distribution on a processor p using the sets developed in Theorem 3.1 shown in Fig. 8. Each contiguous section to be packed corresponds to an instance $(l, l+t-1)$ of the innermost loop. Clearly, a fixed cost is incurred for determining

Table 2: Indexing cost measures for special forms of array redistribution.

Array Redistribution	# Send Sections	# Recv Sections
CYCLIC(Yt) $\rightarrow$ CYCLIC(t)	$\frac{N}{PYt}$ if $Y < P$ $\frac{N}{P^2t}$ if $Y \geq P$	$\frac{N}{PYt}$ if $Y < P$ $\frac{N}{P^2t}$ if $Y \geq P$
CYCLIC(s) $\rightarrow$ CYCLIC(Ys)	$\frac{N}{PYs}$ if $Y < P$ $\frac{N}{P^2s}$ if $Y \geq P$	$\frac{N}{PYs}$ if $Y < P$ $\frac{N}{P^2s}$ if $Y \geq P$

the lower bound l for each section. Since consecutive elements are accessed, each section has good spatial locality and on an average a fixed cache penalty is incurred for each section. If the total number of sections to be packed/unpacked is S, then the indexing cost would be approximately $k * S$, for a constant k whose value depends on the additional computation per section and the processor specifications.

The number of sections to be packed and unpacked for the redistribution from a CYCLIC(Yt) distribution to a CYCLIC(t) is determined from Theorems 3.1 and 3.2. A similar evaluation for the redistribution from a CYCLIC(s) to a CYCLIC(Ys) distribution can be performed using Theorems 3.3 and 3.4. These estimates are shown in Table. 2.

The impact of the indexing cost on the total redistribution time will depend on the characteristics of the processor architecture and communication network. If the communication time dominates the redistribution time, then the indexing time need not be accounted for in the cost model. The index cost for redistribution of multi-dimensional arrays can be similarly estimated.

6 A Multiphase Approach for Array Redistribution

In this section, we describe a multi-phase approach for performing array redistribution. We first consider the redistribution from a CYCLIC(Yt) to a CYCLIC(t) distribution and then the general CYCLIC(s) to CYCLIC(t) redistribution. The redistribution from a CYCLIC(s) to a CYCLIC(Ys) distribution is similar to the CYCLIC(Yt) to a CYCLIC(t) case.

6.1 Redistribution from a CYCLIC(Yt) to a CYCLIC(t) distribution

Consider an array $A(0:319)$ distributed onto 16 processors. A redistribution of A from a CYCLIC(20) distribution to a CYCLIC(2) distribution corresponds to the CYCLIC(Yt) $\rightarrow$ CYCLIC(t) case with $Y = 10$ and $t = 2$. The communication cost of the redistribution is $10t_s + 20t_e$. However, if the redistribution is performed in two phases, CYCLIC(20) $\rightarrow$ CYCLIC(10) $\rightarrow$ CYCLIC(2), the total communication cost is $(2t_s + 20t_e) + (5t_s + 20t_e) = 7t_s + 40t_e$. If the message startup cost t_s is significant compared to the element transmission cost t_e for the underlying architecture, then the two-phase redistribution would require a smaller communication cost.

In general, the multi-phase strategy for redistribution of an array $A(0:N-1)$ distributed onto P processors, from a

$\texttt{CYCLIC}(Yt)$ to a $\texttt{CYCLIC}(t)$ distribution, corresponds to performing the redistribution as a sequence: $\texttt{CYCLIC}(Yt) \rightarrow \texttt{CYCLIC}(\frac{Y}{Y_1}t) \rightarrow \texttt{CYCLIC}(\frac{Y}{Y_1 \times Y_2}t) \rightarrow \cdots \rightarrow \texttt{CYCLIC}(\frac{Y}{Y_1 \times \cdots \times Y_{k-1}}t) \rightarrow \texttt{CYCLIC}(t)$ where $Y = Y_1 \times \cdots \times Y_{k-1} \times Y_k$. Phase i corresponds to a redistribution from a $\texttt{CYCLIC}(\frac{Y}{Y_1 \times \cdots \times Y_i}t)$ distribution to a $\texttt{CYCLIC}(\frac{Y}{Y_1 \times \cdots \times Y_{i+1}}t)$ distribution and has a communication cost of $Y_{i+1}t_s + \frac{N}{P}t_e$. The total cost of the k-phase redistribution is given by

$$\sum_{i=0}^{k-1} \left(Y_{i+1}t_s + \frac{N}{P}t_e \right) \quad = \quad \left(\sum_{i=1}^{k} Y_i \right) t_s + k\frac{N}{P}t_e. \tag{6.1}$$

Thus the multi-phase strategy trades off a decrease in the number of message startups with an increase in the data volume transmitted. Choosing the set of intermediate data distributions is equivalent to choosing a multiplicative factorization $Y = Y_1 \times \cdots \times Y_k$. The choice of the sequence of data distributions with the lowest execution time will depend on the actual values of the message startup time t_s and the data transmission time t_e. We formulate the problem of determining the intermediate distributions as the following optimization problem:

> Given the machine parameters t_s, t_e and P, and the distribution parameters N and Y, determine k and a factorization of $Y = Y_1 \times \cdots \times Y_k$ which minimizes the total redistribution cost in Eq. 6.1.

For systems with a small number of processors, the total number of factorizations of Y is small, and an exhaustive search would be acceptable. However, for a large P and Y the development of algorithms (or heuristics) to solve the optimization problem is essential. When Y is a perfect power r^p, finding a multiplicative partition of Y is equivalent to finding an additive partition of p. We represent an additive partition of p by an l-tuple $(p_{l-1}, \ldots, p_0)$ where $p = p_0 + \cdots + p_{l-1}$. The communication cost for this tuple is given by

$$\left(\sum_{i=0}^{l-1} r^{p_i} \right) t_s + l\frac{N}{P}t_e. \tag{6.2}$$

Finding the optimal solution corresponds to finding the tuple that minimizes the value of Eq. 6.2. The total number of tuples is given by the total number of additive partitions of p, $a(p) = \frac{1}{4\sqrt{3}p}e^{\pi\sqrt{2/3}\sqrt{p}}$ [8] which is exponential. The optimization problem is solved in $\theta(\log Y)$ time as follows. A *min-tuple* of size l (as constructed below) is proved to have the the lowest value for Eq. 6.2 among all possible tuples of size l (Refer Appendix A for the proof). Thus among all tuples of size l, only the value of Eq. 6.2 for the min-tuple need be considered. Let $m(p, l)$ denote the min-tuple of size l for p. We have

$$m(p,l) \quad = \quad (\overbrace{\left\lceil \frac{p}{l} \right\rceil, \ldots, \left\lceil \frac{p}{l} \right\rceil}^{p \bmod l}, \overbrace{\left\lfloor \frac{p}{l} \right\rfloor, \ldots, \left\lfloor \frac{p}{l} \right\rfloor}^{l - p \bmod l})$$

The algorithm generates min-tuples of sizes $l,\ 1 \leq l \leq p$. The optimal tuple is found by comparing values of Eq. 6.2 for all the min-tuples in $O(p)$ time. If Y is not a perfect power, heuristics presented in Appendix B are used. The

Table 3: Indexing cost measures for the two-phase array redistribution.

	Common Multiple m		Common Divisor d	
# Send Sections	$\frac{2N}{Pm}$	if $Y_1 < P \wedge Y_2 < P$	$\frac{N}{P}(\frac{1}{s} + \frac{1}{t})$	if $Y_1 < P \wedge Y_1 < P$
	$\frac{N}{Pm} + \frac{2N}{P^2t}$	if $Y_1 < P \wedge Y_2 \geq P$	$\frac{N}{Ps} + \frac{N}{P^2d}$	if $Y_1 < P \wedge Y_2 \geq P$
	$\frac{N}{P^2s} + \frac{N}{Pm}$	if $Y_1 \geq P \wedge Y_2 < P$	$\frac{N}{P^2d} + \frac{N}{Pt}$	if $Y_1 \geq P \wedge Y_2 < P$
	$\frac{N}{P^2}(\frac{1}{s} + \frac{1}{t})$	if $Y_1 \geq P \wedge Y_2 \geq P$	$\frac{2N}{P^2d}$	if $Y_1 \geq P \wedge Y_2 \geq P$

For common multiple $Y_1 = \frac{m}{s},\ \ Y_2 = \frac{m}{t},$ For common divisor $Y_1 = \frac{t}{d},\ \ Y_2 = \frac{s}{d}$

multi-phase approach for the CYCLIC(s) to CYCLIC(Ys) redistribution can be similarly developed.

6.2 CYCLIC(s) to CYCLIC(t) Redistribution

We now consider the general CYCLIC(s) to CYCLIC(t) redistribution where neither s is a multiple of t nor t a multiple of s. This redistribution can be performed in two stages such that each of the two stages corresponds to a either a CYCLIC(s) to CYCLIC(Ys) redistribution or a CYCLIC(Yt) to CYCLIC(t) redistribution. The block size of the intermediate block-cyclic distribution is chosen as either a common multiple m or a common divisor d of s and t. The choice of the divisor/multiple should be such that the total communication cost is reduced. A similar approach was concurrently proposed in [26].

Using the cost metrics developed for the special cases, the communication cost for the two-stage CYCLIC$(s) \rightarrow$ CYCLIC$(m) \rightarrow$ CYCLIC(t) distribution is $\left(\frac{m}{s}t_s + \frac{N}{P}t_e\right) + \left(\frac{m}{t}t_s + \frac{N}{P}t_e\right) = m\left(\frac{1}{s} + \frac{1}{t}\right)t_s + 2\frac{N}{P}t_e$. Clearly the choice of a multiple m which minimizes $m\left(\frac{1}{s} + \frac{1}{t}\right)$ should provide the lowest communication cost. Since the least common multiple $l = \mathrm{lcm}(s,t)$ minimizes the expression, the two-stage redistribution using CYCLIC(l) as the intermediate distribution should have the lowest communication cost. Similarly, the cost for the two-stage redistribution using a common divisor d can be estimated as $\frac{1}{d}(s+t)t_s + 2\frac{N}{P}t_e$. It follows that the intermediate distribution CYCLIC(g) where $g = \mathrm{gcd}(s,t)$ should have the lowest communication cost among all common divisors of s and t.

Since we are interested in finding the scheme which requires the lowest communication cost, a question that arises is which of the two-stage redistribution strategies has lower communication cost. As shown by Lemma 6.1, the common divisor and common multiple approach using the lcm and gcd, respectively, should have equal communication cost.

Lemma 6.1 *For $s, t \in \mathcal{Z}^+$, and $l = \mathrm{lcm}(s,t)$, $g = \mathrm{gcd}(s,t)$, $l\left(\frac{1}{s} + \frac{1}{t}\right) = \frac{1}{g}(s+t)$.*

Proof. Since $st = lg$, we have $\frac{s}{g} + \frac{t}{g} = \frac{sl+tl}{gl} = \frac{sl+tl}{st} = l\left(\frac{1}{s} + \frac{1}{t}\right)$. $\square$

However, the indexing costs for the two schemes are different. The indexing costs for the two phases are composed in Table 3. For any common multiple m and divisor d of s and t, we have $s \leq m,\ t \leq m$ and $s \geq d,\ t \geq d$. Thus from Table 3 the common divisor approach will have greater indexing overhead than common multiple approach for any divisor d and multiple m which have equal communication costs. Thus the common multiple approach using lcm(s,t)

is expected to be the preferred two-stage scheme for performing redistribution from a CYCLIC(s) to a CYCLIC(t) distribution.

Once the common divisor/multiple has been identified, the multi-phase approach can be independently applied for each of the two stages. For example, the CYCLIC(36) $\rightarrow$ CYCLIC(80) redistribution can be performed as the two-stage sequence: CYCLIC(36) $\rightarrow$ CYCLIC(720) $\rightarrow$ CYCLIC(80). The first redistribution CYCLIC(36) $\rightarrow$ CYCLIC(720) corresponds to the CYCLIC(s) $\rightarrow$ CYCLIC(Ys) case while the second corresponds to the CYCLIC(Yt) $\rightarrow$ CYCLIC(t) case. The multi-phase approach can now be independently applied to the CYCLIC(36) $\rightarrow$ CYCLIC(720) and CYCLIC(720) $\rightarrow$ CYCLIC(80) redistributions performing them as CYCLIC(36) $\rightarrow$ CYCLIC(180) $\rightarrow$ CYCLIC(720) and CYCLIC(720) $\rightarrow$ CYCLIC(240) $\rightarrow$ CYCLIC(80), respectively. Using the multi-phase approach the number of message startups is halved while the data transmission cost is doubled.

The multi-phase approach can be similarly used for the redistribution of multi-dimensional arrays. Consider array $A(0 : N_{t-1} - 1, \ldots, 0 : N_0 - 1)$ distributed on a $P_{t-1} \times \cdots \times P_0$ virtual processor mesh. Consider a shape-preserving redistribution such that redistribution along dimension i is either of the form CYCLIC($Y_i t_i$) to CYCLIC(t_i) or CYCLIC(s_i) to CYCLIC($Y_i s_i$). Let $Y = \min(P_{t-1}, Y_{t-1}) \times \cdots \times \min(P_0, Y_0)$. The communication cost of the redistribution is $Y t_s + \frac{N_{t-1} \times \cdots \times N_0}{P_{t-1} \times \cdots \times P_0} t_e$. The problem of determining the intermediate distributions in the multi-phase approach corresponds to finding an appropriate factorization of Y which minimizes the total cost for the redistribution. The cost function is similar to that optimized in Eq. 6.1 and the optimization problem can be solved using the algorithms/heuristics for the single dimensional case.

7 Performance Results

In this section, we present experimental results to demonstrate the efficacy of the multiphase approach for array redistribution. The experiments were performed on two state-of-the-art distributed-memory machines, a 32-node Cray T3D and a 64-node IBM-SP2. The node programs on the Cray T3D used the PVM-message passing library, while the MPL message passing library with user space communication was used on the IBM-SP2. Wall-clock times for redistributing arrays were measured on each machine, and the maximum finish time among all nodes reported.

We first evaluate the performance of the multi-phase approach for the CYCLIC(Yt) to CYCLIC(t) form of array redistribution. We compare the performance of the direct redistribution with a two-phase and a three-phase redistribution. We then present similar measurements for the CYCLIC(s) to CYCLIC(Ys) form. The general CYCLIC(s) to CYCLIC(t) form of array redistribution is expressed in terms of these two special forms and an improved performance for either of the special forms directly implies better performance for the general form. For the CYCLIC(s) to CYCLIC(t) form of redistribution we compare the effect of choice of common multiples and divisors on the total redistribution time for the two-stage approach.

Fig. 9 presents the execution times on the Cray T3D for the CYCLIC(240) to CYCLIC(8) and the CYCLIC(192)

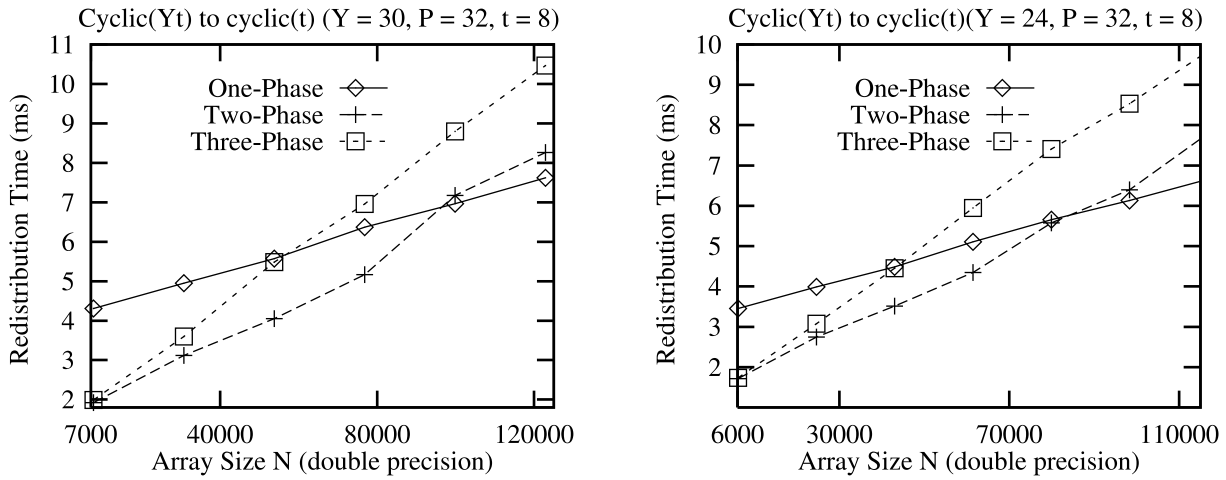

Figure 9: Execution times for CYCLIC(240) to CYCLIC(8) and CYCLIC(192) to CYCLIC(8) redistributions on 32 node T3D.

---

to $\texttt{CYCLIC}(8)$ redistributions of an array of double precision (eight-byte) numbers, respectively. Execution times for single-phase (1-P), two-phase (2-P), and three-phase (3-P) redistributions are presented. For the $\texttt{CYCLIC}(240)$ to $\texttt{CYCLIC}(8)$ redistribution, $Y = 30$, and the 2-P and the 3-P redistributions correspond to the factorizations: $Y = 6 \times 5$ and $Y = 2 \times 3 \times 5$, respectively. For the $\texttt{CYCLIC}(192)$ to $\texttt{CYCLIC}(8)$ redistribution, $Y = 24$, and the 2-P and the 3-P redistributions correspond to the factorizations: $Y = 6 \times 4$ and $Y = 2 \times 3 \times 4$, respectively. We make the following observations.

- The relationship between the rates of increase in execution time with array size is: 3-P $>$ 2-P $>$ 1-P. This follows by noting that for a fixed Y the component of the message startup time in the communication time is nearly fixed. An increase ΔN in the array size should lead to an increase of $\Delta N t_e$ in execution time for the 1-P redistribution, $2\Delta N t_e$ for the 2-P redistribution, and $3\Delta N t_e$ for the 3-P redistribution.

- For both redistributions and for all the array sizes considered, the 2-P redistribution performs better than the 3-P redistribution. This follows by noting that the Cray T3D has a low t_s/t_e ratio. For the array sizes considered, a decrease of one message startup for the 3-P redistribution does not offset the increase in data transmission cost for the additional phase.

- For the $\texttt{CYCLIC}(240)$ to $\texttt{CYCLIC}(8)$ redistribution, the 2-P redistribution performs better than the 1-P redistribution up to an array size of approximately $100K$. However, for the $\texttt{CYCLIC}(192)$ to $\texttt{CYCLIC}(8)$ redistribution the cross-over occurs at a smaller array size of $85K$. This follows by noting that the reduction in the number of message startups for the 2-P stragegy for $Y = 30$ is 19 while that for $Y = 24$ is 14. Hence the increase in data transmission cost will offset the reduction in startups at a smaller array size for $Y = 24$.

Experiments for different block sizes t, show a similar relationship between the one-phase, two-phase, and three-phase redistributions.

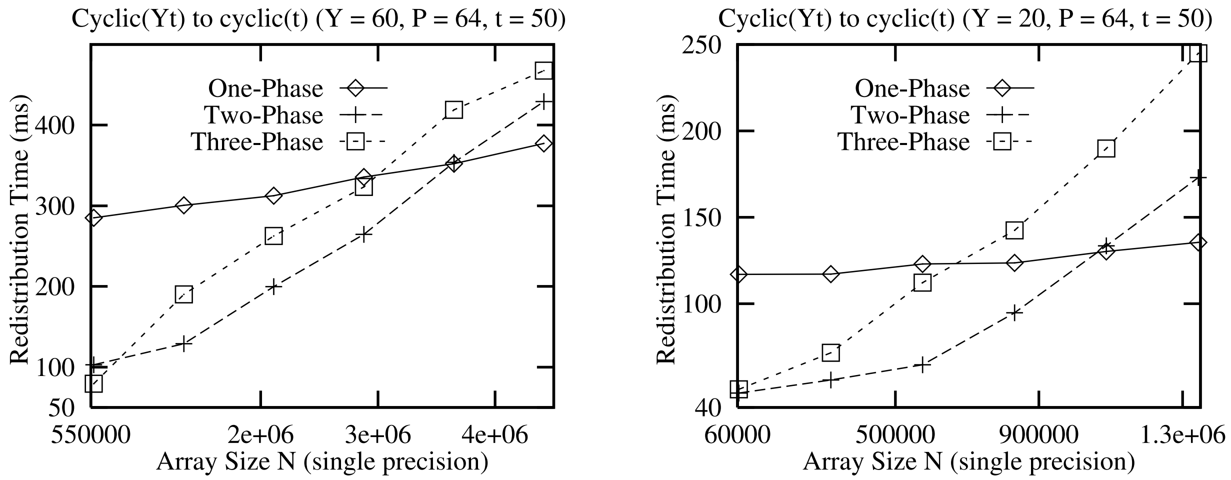

Figure 10: Execution times for `CYCLIC`(3000) to `CYCLIC`(50) and `CYCLIC`(1000) to `CYCLIC`(50) redistribution on IBM-SP2.

Fig. 10 presents the times on the IBM-SP2 for the `CYCLIC`(3000) to `CYCLIC`(50) and the `CYCLIC`(1000) to `CYCLIC`(50) redistributions of an array of single precision (four-byte) numbers, respectively. For the `CYCLIC`(3000) to `CYCLIC`(50) redistribution, $Y = 60$, and the 2-P and the 3-P redistributions correspond to the factorizations: $Y = 6 \times 10$ and $Y = 5 \times 4 \times 3$, respectively. For the `CYCLIC`(1000) to `CYCLIC`(50) redistribution, $Y = 20$, and the 2-P and the 3-P redistributions correspond to the factorizations $Y = 5 \times 4$ and $Y = 5 \times 2 \times 2$, respectively. We observe the following:

- The relations between rates of increase in execution time with array size are: 3-P $>$ 2-P $>$ 1-P.
- Clearly for $Y = 20$, the 3-P redistribution cannot have better performance than the 2-P redistribution for any array size as no reduction in message startups is obtained. For the `CYCLIC`(3000) to `CYCLIC`(50) case ($Y = 60$), the 3-P redistribution has better performance than the 2-P redistribution upto an array size of $650K$. This follows by noting that the IBM-SP2 has a higher t_s/t_e ratio than the Cray T3D. For small array sizes the decrease in communication time due to a reduction in the number of message startups offsets the increase in data transmission cost due to an additional phase.
- The cross-over between 2-P and 1-P redistributions occurs at a array size of $3.6M$ for the `CYCLIC`(3000) to `CYCLIC`(50) redistribution, while the cross-over for the `CYCLIC`(1000) to `CYCLIC`(50) occurs at a smaller array size of $1M$. This difference arises since the 2-P redistribution for $Y = 60$ reduces the number of message startups by 44, while the 2-P redistribution for $Y = 20$ reduces the startups by 11. The 2-P redistribution performs better than the 1-P redistribution for a larger array size range on the SP2 than on the Cray T3D, due to the large t_s/t_e ratio.

Experiments for different block sizes t, show a similar relationship between the one-phase, two-phase and three-phase redistributions.

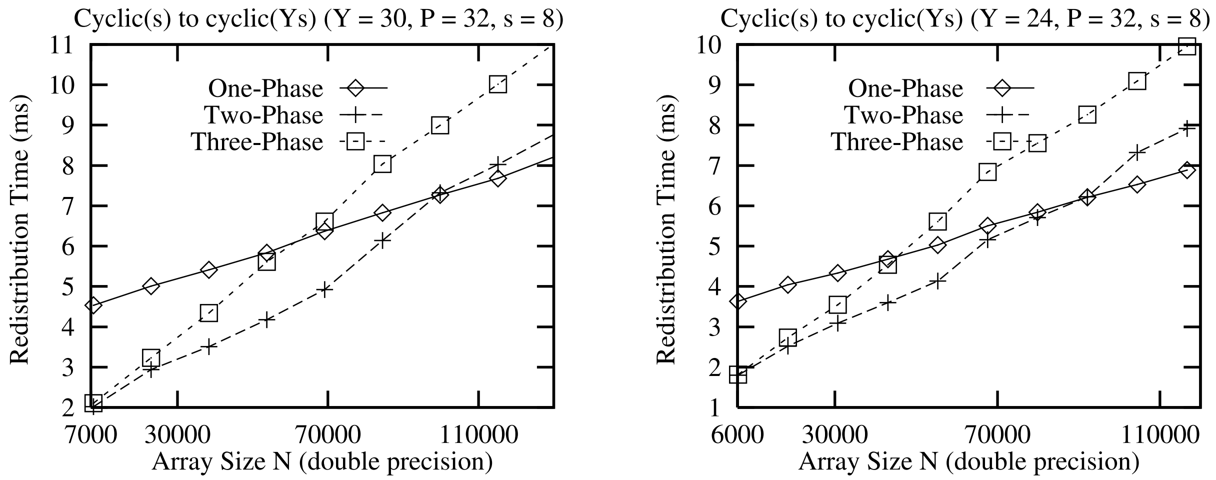

Figure 11: Execution times for CYCLIC(8) to CYCLIC(240) and CYCLIC(8) to CYCLIC(192) redistribution on a 32 node T3D.

The relationship between the one-phase, two-phase, and three-phase redistribution strategies for the CYCLIC(s) to CYCLIC(Ys) form of redistribution is similar to that for the CYCLIC(Yt) to CYCLIC(t) form of redistribution. Plots of the single and multi-phase strategies for a CYCLIC(8) to CYCLIC(240) and CYCLIC(8) to CYCLIC(192) redistribution on the Cray T3D and a CYCLIC(50) to CYCLIC(3000) and a CYCLIC(50) to CYCLIC(1000) on the IBM-SP2, are shown in Fig. 11 and Fig. 12, respectively. The key-points to note are:

- On the Cray T3D, the cross-over between the 2-P and the 1-P redistributions occurs at an array size of $100K$ for the CYCLIC(8) to CYCLIC(240) redistribution, and at $87K$ for the CYCLIC(8) to CYCLIC(192) redistribution.
- On the IBM-SP2, the cross-over between the 2-P and the 1-P redistributions occurs at an array size of $4M$ for the CYCLIC(50) to CYCLIC(3000) redistribution, and at $1M$ for the CYCLIC(8) to CYCLIC(192) redistribution. The cross-over between the 2-P and 3-P occurs at a data size of $600K$ for the CYCLIC(50) to CYCLIC(3000) redistribution. For the CYCLIC(50) to CYCLIC(1000) redistribution, the 2-P redistribution performs better than the 3-P redistribution for all array sizes considered.

We now evaluate the effect of the choice of common multiples and divisors on the general CYCLIC(s) to CYCLIC(t) redistribution. Table. 4 shows the two-stage redistribution times for various source and target block sizes on the Cray T3D. The redistribution times for various multiples and divisors are presented. It can be observed that the two-stage redistributions using the lcm l and the gcd g as block sizes for the intermediate distribution perform better than those using a common multiple greater than the lcm and a common divisor smaller than the gcd. The relationship between the two-stage strategies for multiples is: $t_l < t_{l*2} < t_{l*4}$. Similarly, for the divisors we have: $t_g < t_{g/2} < t_{g/4}$. These relations can be explained by noting that the two-stage redistribution using either the lcm or the gcd as the block size of the intermediate block-cyclic distribution has the lowest communication cost among all common divisors and multiples, as discussed in Section 6. Furthermore it can be observed that $t_{l*2} < t_{g/2}$ and $t_{l*4} < t_{g/4}$. While the

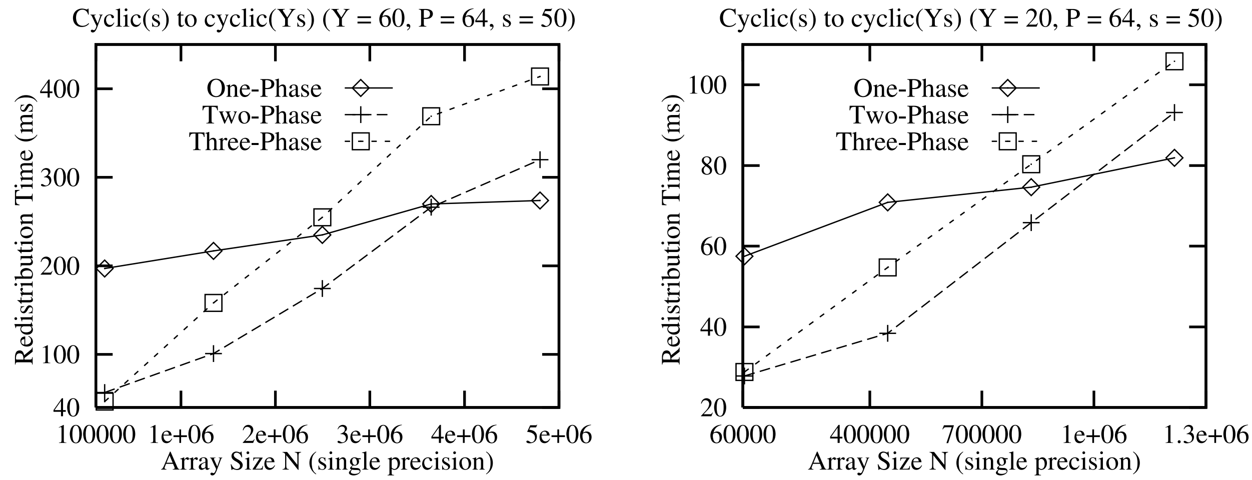

Figure 12: Execution times for CYCLIC(50) to CYCLIC(3000) and CYCLIC(50) to CYCLIC(1000) redistribution on IBM-SP2.

$N = 552960$							
s	t	l	$l*2$	$l*4$	g	$g/2$	$g/4$
120	180	22.4	24.2	27.2	22.5	24.7	28.2
120	540	24.2	27.7	34.5	24.5	28.3	36.3
240	180	23.1	25.3	29.4	23.4	25.7	30.4
240	540	24.9	28.7	36.5	25.2	29.3	38.6
480	180	24.2	27.4	34.0	24.2	28.0	35.4
480	540	25.9	31.0	40.9	26.2	31.5	43.5

$N = 2211840$							
s	t	l	$l*2$	$l*4$	g	$g/2$	$g/4$
120	180	84.6	87.5	91.1	85.3	89.8	94.8
120	540	86.4	91.0	105.9	87.5	93.3	112.9
240	180	86.0	88.2	92.5	86.6	90.5	96.6
240	540	87.7	91.7	107.0	88.8	94.0	114.5
480	180	86.5	89.9	96.7	87.3	92.1	102.6
480	540	88.3	93.5	110.3	89.5	95.6	120.5

Table 4: Execution times (ms) for CYCLIC(s) to CYCLIC(t) redistribution on 32-processor Cray T3D. $l = \text{lcm}(s, t)$, $g = \text{gcd}(s, t)$.

two-stage methods using the lcm and the gcd have nearly equal performance for a small array size, for a larger array size $t_l < t_g$. Although these schemes are predicted to have the same communication cost, a two-stage redistribution using a multiple will have a lower indexing cost than the redistribution using a divisor; this is explained by the indexing cost model presented in Section 5.

Execution times for the CYCLIC(s) to CYCLIC(t) form of array redistribution on the IBM-SP2 are presented in Table 5. Similar relationships between various multiples and divisors are observed.

8 Conclusion

In this paper, we have presented a multi-phase approach for performing communication-efficient data redistribution for block-cyclically distributed arrays. We have developed precise closed form expressions for the send and receive processor and data index sets for two special cases where the block size of the source block-cyclic distribution is a multiple of the block size of the target block-cyclic distribution and vice versa. These closed forms facilitate the

$N = 1105920$							
s	t	l	$l*2$	$l*4$	g	$g/2$	$g/4$
120	180	317	333	341	320	324	343
120	540	304	478	707	314	487	715
240	180	351	281	462	366	271	448
240	540	344	424	798	360	431	806
480	180	292	374	696	292	370	738
480	540	289	521	1048	308	558	1055

$N = 4423680$							
s	t	l	$l*2$	$l*4$	g	$g/2$	$g/4$
120	180	759	1013	1115	757	1040	1137
120	540	886	1126	1155	889	1159	1182
240	180	847	1126	1324	925	1133	1361
240	540	1022	1208	1344	1047	1236	1416
480	180	960	1265	1113	963	1363	1231
480	540	706	772	849	732	813	881

Table 5: Execution times (ms) for CYCLIC(s) to CYCLIC(t) redistribution on IBM-SP2.

development of a distributed scheduling algorithm for performing the all-to-many personalized communication, and a communication and indexing cost model for array redistribution. The algorithm eliminates node contention, incurs minimum communication overhead and has an asymptotically lower complexity than the algorithms presented in the literature. Based on this model, we demonstrate that the use of the multi-phase redistribution strategy can reduce the total cost for array redistribution. Performance results on the Cray T3D and the IBM-SP2 demonstrate that the multi-phase strategy can improve performance over the single phase strategy for array redistribution.

Acknowledgments

We would like to thank the Ohio Supercomputer Center for access to the Cray T3D and the Maui High Performance Computing Center for access to the IBM-SP2.

References

[1] K. P. Bogart. *Introductory Combinatorics*. Harcourt Brace Jovanovich Inc., Florida, second edition, 1990.

[2] S. Bokhari. Multiphase complete exchange on a circuit switched hypercube. In *Proc. of Intl. Conf. on Parallel Processing*, volume I, pages 525–538, Aug. 1991.

[3] S. Bokhari and H. Berryman. Complete exchange on a circuit switched mesh. In *Proc. of Scalable High Performance Computing Conference*, pages 300–306, May 1992.

[4] B. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortran – a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Language, Compilers and Runtime Environments for Distributed Memory Machines*, pages 39–62. 1992.

[5] N. Deo. *Graph Theory with Applications to Engineering and Computer Science*. Prentice Hall of India, 1987.

[6] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[7] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran-D language specification. Technical Report TR-91-170, Dept. of Computer Science, Rice University, Dec. 1991.

[8] E. Grosswald. *Topics from the Theory of Numbers*. Birkhauser, Boston, 1984.

[9] S. Gupta, S. Hawkinson, and B. Baxter. A binary interleaved algorithm for complete exchange on a mesh architecture. Intel Technical Report, Intel Corporation, 1993.

[10] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. A methodology for the generation of data distributions to optimize communication. In *Proc. of IEEE Symposium on Parallel and Distributed Processing*, pages 436–441, Dec. 1992.

[11] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical Report OSU-CISRC-4/94-TR19, Department of Computer and Information Science, The Ohio State University., Apr. 1994.

[12] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On the generation of efficient data communication for distributed-memory machines. In *Proc. of Intl. Computing Symposium*, pages 504–513, Dec. 1992.

[13] S. L. Johnsson and C. T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computer Systems*, C-38(9):1249–1268, Sep. 1989.

[14] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan. An approach to communication efficient data redistribution. In *Proc. of Intl. Conf. on Supercomputing*, pages 364–373, Jul. 1994.

[15] S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multiphase array redistribution: Modeling and Evaluation. Technical Report OSU-CISRC-9/94-TR52, Department of Computer and Information Science, The Ohio State University., Sep. 1994.

[16] C. Koelbel. Compile-time generation of communication for scientific programs. In *Proc. of Supercomputing '91*, pages 101–110, Nov. 1991.

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

[18] L. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 3:62–76, Feb. 1993.

[19] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed-memory multicomputers. Technical Report UILA-ENG-94-2213, CRHC-94-09, CRHPC, Coordinated Science Laboratory, University of Illinios, Apr. 1994.

[20] S. Ranka and J.-C. Wang. Static and runtime scheduling of unstructured communication. *Intl. Journal of Computing Systems in Engineering*, 1993.

[21] S. Ranka, J.-C. Wang, and M. Kumar. Personalized communication avoiding node contention on distributed memory systems. In *Proc. of Intl. Conf. on Parallel Processing*, volume I, pages 241–243, Aug. 1993.

[22] S. Ranka, J.-C. Wang, and M. Kumar. Irregular personalized communication on distributed memory systems. *Journal of Parallel and Distributed Computing*, 1995. To appear.

[23] D. Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Proc. of Distributed Memory Computing Conference*, pages 398–403, Aug. 1991.

[24] N. S. Sundar, D. N. Jayasimha, D.K. Panda, and P. Sadayappan. Complete exchange in 2D meshes. In *Proc. of Scalable High Performance Computing Conference*, pages 406–413, May 1994.

[25] R. Thakur and A. Choudhary. All-to-all communication on meshes with wormhole routing. In *Proc. of Intl. Parallel Processing Symposium*, pages 525–538, Apr. 1994.

[26] R. Thakur and A. Choudhary. Efficient algorithms for array redistribution. Technical Report SCCS-601, Northeast Parallel Architectures Center, Jun. 1994. Revised Oct. 1994.

[27] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. In *Proc. of Scalable High Performance Computing Conference*, pages 309–316, May 1994.

[28] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, 1992.

[29] A. Wakatani and M. Wolfe. A new approach to array redistribution: Strip Mining Redistribution. In *Proc. of Parallel Architectures and Languages Europe (PARLE 94)*, Jul. 1994.

[30] A. Wakatani and M. Wolfe. Effectiveness of message strip-mining for regular and irregular communication. In *Intl. Conference on Parallel and Distributed Computing Systems*, Oct. 1994.

A Proof of Optimality of Min-Tuple for $Y = r^k$

We now prove the optimality of the *min-tuple* presented in Section 6. We first formally define the *min-tuple*. Let $\mathcal{T}(k,l)$ denote the set of all *ordered l-tuples* corresponding to additive partitions of k. An ordered l-tuple $(k_{l-1},\ldots,k_0)$ in $\mathcal{T}(k,l)$ satisfies the relation that $\sum_{i=0}^{l-1} k_i = k$ and $k_i \geq k_{i-1},\ 1 \leq i < l$. Given a tuple $s \in \mathcal{T}(k,l)$, let s_i represent component i of s. The communication cost for the ordered l-tuple $s = (s_{l-1},\ldots,s_0)$ as given by Eq. 6.1 is $C(s) = (\sum_{i=0}^{l-1} r^{s_i}) * t_s + l * \frac{N}{P} t_e$. We now define some relations between two tuples in $\mathcal{T}(k,l)$ and based on this relation impose a linear irreflexive order on $\mathcal{T}(k,l)$.

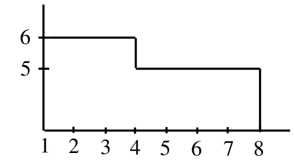

Figure 13: Min-Tuple $m(44, 8) = (6, 6, 6, 6, 5, 5, 5, 5)$

Definition A.1 *Let $c \in [0 : l-1]$; then for any two ordered tuples $r, s \in \mathcal{T}(k, l)$, $r \prec_c s \iff ((\forall j \in [c+1 : l-1] : r_j = s_j) \wedge r_c < s_c)$.*

Lemma A.1 *Given the set of tuples $\mathcal{T}(k, l)$ we have.*

1. *$\forall c \in [0 : l-1], \prec_c$ is an irreflexive partial order on $\mathcal{T}(k, l)$.*
2. *$\forall r, s \in \mathcal{T}(k, l)$, either $r = s$ or there exists a unique $c \in [0 : l-1]$ such that either $r \prec_c s$ or $s \prec_c r$.*
3. *$\forall r, s, u \in \mathcal{T}(k, l)$ and $c,\ c' \in [0 : l-1]$, $r \prec_c s \wedge s \prec_{c'} u \Rightarrow r \prec_{c''} u$ where $c'' = max(c, c')$.*

We now define the lexicographic relation $\prec$ between two ordered l-tuples.

Definition A.2 (Lexicographic Order) $\forall r, s \in \mathcal{T}(k, l) : r \prec s \Leftrightarrow \exists c \in [0 : l-1] : r \prec_c s$.

The relation $\prec$ on $\mathcal{T}(k, l)$ is similar to the lexicographical order $<$ defined on $\mathcal{Z}^n$. The relations $\preceq, \succ$, and $\succeq$ are defined equivalently. Using Lemma A.1 and Def. A.2, we have:

Lemma A.2 *The lexicographical order $\prec$ on $\mathcal{T}(k, l)$ is an irreflexive linear order.*

From Lemma A.2 and noting that the set $\mathcal{T}(k, l)$ is finite, it is evident that $\mathcal{T}(k, l)$ has a unique minimum. We refer to the minimum as the *min-tuple*, and denote it as $m(k, l)$. Noting that $k = \lceil \frac{k}{l} \rceil (k \ mod \ l) + \lfloor \frac{k}{l} \rfloor (l - k \ mod \ l)$ the *min-tuple* can be proven to be.

$$m(k, l) = (\overbrace{\left\lceil \frac{k}{l} \right\rceil, \ldots, \left\lceil \frac{k}{l} \right\rceil}^{k \ mod \ l}, \overbrace{\left\lfloor \frac{k}{l} \right\rfloor, \ldots, \left\lfloor \frac{k}{l} \right\rfloor}^{l-k \ mod \ l})$$

The proof follows by noting that

- Any tuple s such that $s_{l-1} > \lceil \frac{k}{l} \rceil$ is greater than the min-tuple and
- The min-tuple is the only tuple with largest component equal to $\lceil \frac{k}{l} \rceil$.

The values of the components of the min-tuple have a characteristic property, i.e., there is at most one "jump" of size one in the component values. For instance, the min-tuple $m(44, 8) = (6, 6, 6, 6, 5, 5, 5, 5)$ can be represented as shown in Fig. 13. The area under the solid-line curve is 44. This property is stated in Lemma A.3 and follows directly from the definition of the min-tuple and noting that $\lceil \frac{k}{l} \rceil - \lfloor \frac{k}{l} \rfloor \leq 1$.

Lemma A.3 $0 \leq m(k, l)_i - m(k, l)_{i-1} \leq 1, \ \ 0 < i \leq l-1$

This property of the *min-tuple* leads to the following lemma.

Lemma A.4 *Let $t \in \mathcal{T}(k, l)$ be any tuple $t \neq m(k, l)$ and let $c \in [0 : l-1]$ be the largest index such that $m(k, l)_c \geq t_c$. Then $m(k, l)_i \leq t_i, \ c < i < l$ and $m(k, l)_i \geq t_i, 0 \leq i \leq c$.*

The proof follows by considering the three possible relations between the tuple t and $m(k, l)$ as shown in Fig. 14. The lemma simply states that once the plot for the tuple t goes below that of the mintuple, it remains below. Using Lemma A.4 we prove that the min-tuple $m(k, l)$ has the lowest cost in $\mathcal{T}(k, l)$. We first introduce the concept of *adjacent tuples*.

Definition A.3 (Adjacent Tuples) *Two ordered tuples $r, s \in \mathcal{T}(k, l)$ are adjacent if they differ in exactly two components, i.e., $r_{c1} \neq s_{c1}$, $r_{c2} \neq s_{c2}$, and $r_i = s_i, \ \forall i, \ i \neq c_1, \ i \neq c_2$.*

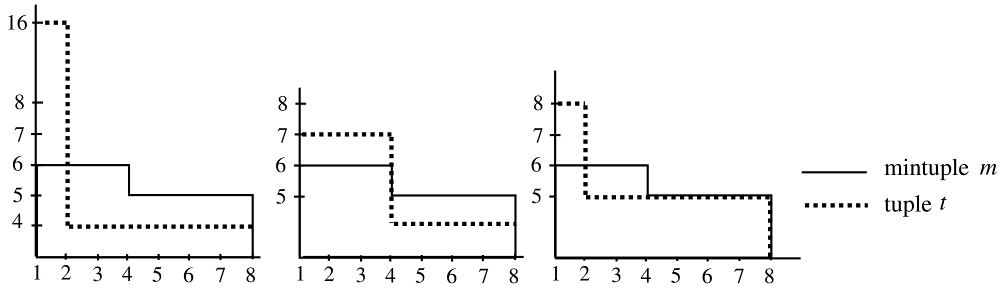

Figure 14: Relation between tuple t and min-tuple m.

Note that the adjacent tuples have to differ in exactly two components and that $r_{c_1} - s_{c_1} = s_{c_2} - r_{c_2}$, where c_1 and c_2 are the indices in which r and s differ. The relation between the costs of two adjacent tuples is provided in the following lemma.

Lemma A.5 *If $r, s \in \mathcal{T}(k,l)$ are adjacent tuples, then $r \prec s \Rightarrow C(r) < C(s)$.*

Proof: Since $r \prec s$ and r, s are adjacent, it follows that r and s are of the form $r = (x_{l-1}, \ldots, x_p, \ldots, x_q, \ldots, x_0)$ and $s = (x_{l-1}, \ldots, x_p + c, \ldots, x_q - c, \ldots, x_0)$, where $c > 0$. Consider $C(s) - C(r)$. We have $C(s) - C(r) = r^{x_p+c} + r^{x_q-c} - r^{x_p} - r^{x_q} = r^{x_p}(r^c - 1) - r^{x_q-c}(r^c - 1) = (r^c - 1)(r^{x_p} - r^{x_q-c})$. Since $x_p \geq x_q$ and $c > 0$ we have $C(s) - C(r) > 0$. □.

Given any tuple $t \in \mathcal{T}(k,l)$, it is possible to construct a chain of adjacent tuples starting at t and ending in the min-tuple $m(k,l)$. We have the following lemma.

Lemma A.6 *Given a tuple $t \in \mathcal{T}(k,l)$, there exists a chain of adjacent tuples $t, t_1, t_2, \ldots, t_r, m(k,l)$ in $\mathcal{T}(k,l)$, such that $t \succeq t_1$, $t_i \succeq t_{i+1}, 1 \leq i < r-1$, and $t_r \succeq m(k,l)$.*

For example, the chain for the tuple $(9,7,6,5,4,2) \in \mathcal{T}(33,6)$ is: $(9,7,6,5,4,2)$, $(9,6,6,5,5,2)$, $(6,6,6,5,5,5)$. The chain can be constructed by iteratively constructing t_{i+1} from t_i and $m(k,l)$ using Lemma A.4. Thus we have

Theorem A.1 *The min-tuple $m(k,l)$ has the lowest cost among all tuples $t \in \mathcal{T}(k,l)$.*

Proof: Follows from Lemma A.5 and Lemma A.6. □

B Heuristics for General Y

We consider different cases of the general Y. Depending on the relationship between t_s and t_e we can identify optimal solutions or use heuristics.

We first consider the case when $t_s >> \frac{N}{P} t_e$. This case occurs when redistributing small arrays or for architectures with large message setup times. The cost function in Eq. 6.1 is

$$\sum_{i=0}^{k-1} \left(Y_{i+1} t_s + \frac{N}{P} t_e \right) \approx \sum_{i=0}^{k-1} \left(Y_{i+1} \right) t_s$$

Thus we are interested in finding a factorization of $Y = Y_1 \times \cdots \times Y_k$ which minimizes $\sum_{i=1}^{k} Y_i$. We have

Lemma B.1 $\forall r, s \in \mathcal{Z},\ rs \geq r + s$

From Lemma B.1, it follows that the prime factorization of Y will have the lowest communication time. This case is at one extreme at which the data transmission cost is negligible compared to the setup cost. The other extreme corresponds to the case when the setup cost is negligible. For this case the single phase redistribution will have lowest cost.

For the other cases, we use a greedy heuristic to minimize the objective function in Eq. 6.1. The heuristic starts with the prime factorization of Y. At each stage the algorithm combines a pair of factors, such that this grouping leads to the largest reduction in the objective function. If Y is composed of t prime factors each step requires $O(t^2)$ steps, as atmost t^2 pairs have to be examined for combining. This grouping is continued until no further reduction in the value of the objective function is possible. The grouping will terminate in at most $t-1$ steps. Thus, the heuristic has a $O(t^3)$ complexity. If no reduction is possible in the first step then the prime factorization is the optimal. However, in the general case the heuristic may get into a local minima and may not yield the optimal solution.