The Journal of Supercomputing, 12:253–276 (1998)

Efficient Methods for $kr \rightarrow r$ and $r \rightarrow kr$ Array Redistribution[1]

CHING-HSIEN HSU
Department of Information Engineering, Feng Chia University, Taichung, Taiwan 407, ROC, chhsu@iecs.fcu.edu.tw

YEH-CHING CHUNG
Department of Information Engineering, Feng Chia University, Taichung, Taiwan 407, ROC, ychung@iecs.fcu.edu.tw

Abstract. Array redistribution is usually required to enhance algorithm performance in many parallel programs on distributed memory multicomputers. Since it is performed at run-time, there is a performance tradeoff between the efficiency of new data decomposition for a subsequent phase of an algorithm and the cost of redistributing data among processors. In this paper, we present efficient algorithms for BLOCK-CYCLIC(kr) to BLOCK-CYCLIC(r) and BLOCK-CYCLIC(r) to BLOCK-CYCLIC(kr) redistribution. The most significant improvement of our methods is that a processor does not need to construct the send/receive data sets for a redistribution. Based on the *packing/unpacking* information that derived from the BLOCK-CYCLIC(kr) to BLOCK-CYCLIC(r) redistribution and vice versa, a processor can pack/unpack array elements into (from) messages directly. To evaluate the performance of our methods, we have implemented our methods along with the Thakur's methods and the *PITFALLS* method on an IBM SP2 parallel machine. The experimental results show that our algorithms outperform the Thakur's methods and the *PITFALLS* method for all test samples. This result encourages us to use the proposed algorithms for array redistribution.

Keywords: Array redistribution, distributed memory multicomputers, data distribution, runtime support

1. Introduction

The data parallel programming model has become a widely accepted paradigm for programming distributed memory multicomputers. To efficiently execute a data parallel program on a distributed memory multicomputer, an appropriate data decomposition is critical. The data decomposition involves *data distribution* and *data alignment*. The data distribution deals with how data arrays should be distributed. The data alignment deals with how data arrays should be aligned with respect to one another. The purpose of data decomposition is to balance the computational load and minimize the communication overheads.

Many data parallel programming languages such as High Performance Fortran (HPF) [9], Fortran D [6], Vienna Fortran [32], and High Performance C (HPC) [27] provide compiler directives for programmers to specify array distribution. The array distribution provided by those languages, in general, can be classified into two categories, *regular* and *irregular*. The regular array distribution, in general, has three types, BLOCK, CYCLIC,

and BLOCK-CYCLIC(c). The BLOCK-CYCLIC(c) is the most general regular array distribution among them. Dongarra *et al* [5] have shown that these distribution are essential for many dense matrix algorithms design in distributed memory machines. Examples of distributing a one-dimensional array with 18 elements to three processors using BLOCK, CYCLIC, and BLOCK-CYCLIC(c) distribution are shown in Figure 1. The irregular array distribution uses user-defined array distribution functions to specify array distribution.

In some algorithms, such as multi-dimensional fast Fourier transform [28], the Alternative Direction Implicit (ADI) method for solving two-dimensional diffusion equations, and linear algebra solvers [20], an array distribution that is well-suited for one phase may not be good for a subsequent phase in terms of performance. Array redistribution is required for those algorithms during run-time. Therefore, many data parallel programming languages support run-time primitives for changing a program's array decomposition [1, 2, 9, 27, 32]. Since array redistribution is performed at run-time, there is a performance trade-off between the efficiency of a new data decomposition for a subsequent phase of an algorithm and the cost of redistributing array among processors. Thus efficient methods for performing array redistribution are of great importance for the development of distributed memory compilers for those languages.

Array redistribution, in general, can be performed in two phases, the send phase and the receive phase. In the send phase, a processor P_i has to determine all the data sets that it needs to send to other processors (destination processors), pack those data sets into messages, and send messages to their destination processors. In the receive phase, a processor P_i has to determine all the data sets that it needs to receive from other processors (source processors), receive messages from source processors, and unpack elements in messages to their corresponding local array positions. This means that each processor P_i should compute the following four sets.

- Destination Processor Set (DPS[P_i]): the set of processors to which P_i has to send data.
- Send Data Sets $\left(\bigcup_{P_j \in DPS[P_i]} SDS[P_i, P_j]\right)$: the sets of array elements that processor P_i has to send to its destination processors, where SDS[P_i, P_j] denotes the set of array elements that processor P_i has to send to its destination processor P_j.
- Source Processor Set (SPS[P_j]): the set of processors from which P_j has to receive data.

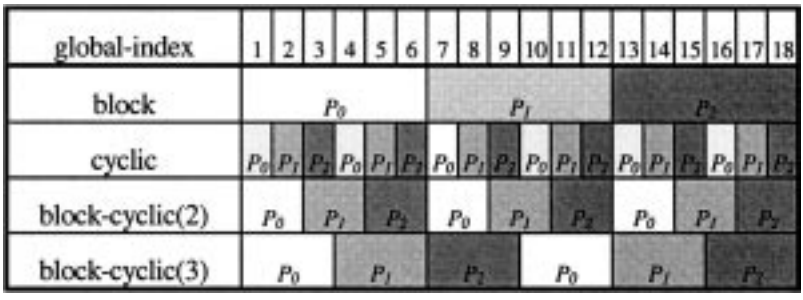


Figure 1. Examples of regular data distribution.

- Receive Data Sets $\left(\bigcup_{P_i \in SPS[P_j]} RDS[P_j, P_i]\right)$: the sets of array elements that P_j has to receive from its source processors, where RDS[P_j, P_i] denotes the set of array elements that processor P_j has to receive from its source processor P_i.

In the send phase, a processor uses the SDS to pack data for each destination processor. In the receive phase, a processor uses the RDS to unpack messages. By determining the send/receive data sets (SDS/RDS) and packing the send data sets into messages, a processor will perform only one **send** operation and one **receive** operation for each processor in its destination processor set (DPS) and its source processor set (SPS), respectively. This implies that the minimum number of send and receive operations required by a processor in a redistribution is equal to the number of processors in its destination processor set and the number of processors in its source processor set, respectively. Using this observation, we know that, to minimize the communication overheads in a redistribution is difficult. On the contrary, to minimize the computation overheads (compute the source/destination processors sets, send/receive data sets, packing, unpacking, etc.) is possible. If a processor can reduce some computation overheads in a redistribution, then the overall performance can be improved.

In this paper, we present efficient methods to perform BLOCK-CYCLIC(*kr*) to BLOCK-CYCLIC(*r*) and BLOCK-CYCLIC(*r*) to BLOCK-CYCLIC(*kr*) redistribution. The most significant improvement of our methods is that a processor does not need to construct the send/receive data sets for a redistribution. Based on the packing/unpacking information that derived from the BLOCK-CYCLIC(*kr*) to BLOCK-CYCLIC(*r*) redistribution and vice versa, a processor can pack/unpack array elements into (from) messages directly. To evaluate the proposed methods, we have implemented our methods along with the Thakur's methods [24, 25] and the *PITFALLS* method [21, 22] on an IBM SP2 parallel machine. The experimental results show that our algorithms outperform the Thakur's methods and the *PITFALLS* method for all test samples.

This paper is organized as follows. In Section 2, a brief survey of related work will be presented. Section 3 presents the algorithms for BLOCK-CYCLIC(*kr*) to BLOCK-CYCLIC(*r*) and BLOCK-CYCLIC(*r*) to BLOCK-CYCLIC(*kr*) redistribution. The performance evaluation and comparisons of array redistribution algorithms that proposed in this paper and in [21, 22, 24, 25] will be given in Section 4. The conclusions will be given in Section 5.

2. Related work

Many methods for performing array redistribution have been presented in the literature. Since techniques of redistribution can be performed either by using the multicomputer compiler technique [26] or using the runtime support technique, we briefly describe the related research in these two approaches.

Gupta et al. [7] derived closed form expressions to efficiently determine the send/receive processor/data sets. They also provided a virtual processor approach [8] for addressing the problem of reference index-set identification for array statements with BLOCK-CYCLIC(c) distribution and formulated active processor sets as closed forms. A recent work in [15] extended the virtual processor approach to address the problem of memory allocation and index-sets identification. By using their method, closed form expressions for index-sets of arrays that were mapped to processors using one-level mapping can be translated to closed form expressions for index-sets of arrays that were mapped to processors using two-level mapping and vice versa. A similar approach that addressed the problems of the index set and the communication sets identification for array statements with BLOCK-CYCLIC(c) distribution was presented in [23]. In [23], the CYCLIC(k) distribution was viewed as a union of k CYCLIC distribution. Since the communication sets for CYCLIC distribution is easy to determine, communication sets for CYCLIC(k) distribution can be generated in terms of unions and intersections of some CYCLIC distributions.

Lee et al. [17] derived communication sets for statements of arrays which were distributed in arbitrary BLOCK-CYCLIC(c) fashion. They also presented closed form expressions of communication sets for restricted block size. In [3], Chatterjee et al. enumerated the local memory access sequence of communication sets for array statements with BLOCK-CYCLIC(c) distribution based on a finite-state machine. In this approach, the local memory access sequence can be characterized by a FSM at most c states. In. [16], Kennedy et al. also presented algorithms to compute the local memory access sequence for array statements with BLOCK-CYCLIC(c) distribution.

Thakur et al. [24, 25] presented algorithms for run-time array redistribution in HPF programs. For BLOCK-CYCLIC(kr) to BLOCK-CYCLIC(r) redistribution (or vice versa), in most cases, a processor scanned its local array elements once to determine the destination (source) processor for each block of array elements of size r in the local array. In [21, 22], Ramaswamy and Banerjee used a mathematical representation, PITFALLS, for regular data redistribution. The basic idea of PITFALLS is to find all intersections between source and target distributions. Based on the intersections, the send/receive processor/data sets can be determined and general redistribution algorithms can be devised. In [10], an approach for generating communication sets by computing the intersections of index sets corresponding to the LHS and RHS of array statements was also presented. The intersections are computed by a scanning approach that exploits the repetitive pattern of the intersection of two index sets.

Kaushik et al. [13, 14] proposed a multi-phase redistribution approach for BLOCK-CYCLIC(s) to BLOCK-CYCLIC(t) redistribution. The main idea of multi-phase redistribution is to perform a redistribution as a sequence of redistribution such that the communication cost of data movement among processors in the sequence is less than that of direct redistribution. Based on the closed form representations, a cost model for estimating the communication and the indexing overheads for array distribution was developed. From the cost model, algorithms for determining the sequence of intermediate array distribution that minimize the total redistribution time were presented.

Instead of redistributing the entry array at one time, a strip mining approach was presented in [30]. In this approach, portions of array elements were redistributed in sequence in order to overlap the communication and computation. In [31], a spiral mapping technique was proposed. The main idea of this approach was to map formal processors onto actual processors such that the global communication can be translated to the local communication in a certain processor group. Since the communication is local to a processor group, one can reduce communication conflicts when performing a redistribution. Kalns and Ni [11, 12] proposed a processor mapping technique to minimize the amount of data exchange for BLOCK to BLOCK-CYCLIC(c) redistribution and vice versa. Using the data to logical processors mapping, they show that the technique can achieve the maximum ratio between data retained locally and the total amount of data exchanged. In [18], a generalized circulant matrix formalism was proposed to reduce the communication overheads for BLOCK-CYCLIC(r) to BLOCK-CYCLIC(kr) redistribution. Using the generalized circulant matrix formalism, the authors derived direct, indirect, and hybrid communication schedules for the cyclic redistribution with the block size changed by an integer factor k. They also extended this technique to solve some multidimensional redistribution problems [19].

Walker et al. [29] used the standardized message passing interface, MPI, to express the redistribution operations. They implemented the BLOCK-CYCLIC array redistribution algorithms in a synchronous and an asynchronous scheme. Since the excessive synchronization overheads incurred from the synchronous scheme, they also presented the random and optimal scheduling algorithms for BLOCK-CYCLIC array redistribution. The experimental results show that the performance of synchronized method with optimal scheduling algorithm is comparable to that of the asynchronous method.

3. Efficient methods for BLOCK-CYCLIC(kr) to BLOCK-CYCLIC(r) and BLOCK-CYCLIC(r) to BLOCK-CYCLIC(kr) redistribution

In general, a BLOCK-CYCLIC(s) to BLOCK-CYCLIC(t) redistribution can be classified into three types:

- s is divisible by t, i.e. BLOCK-CYCLIC($s = kr$) to BLOCK-CYCLIC($t = r$) redistribution,
- t is divisible by s, i.e. BLOCK-CYCLIC($s = r$) to BLOCK-CYCLIC($t = kr$) redistribution,
- s is not divisible by t and t is not divisible by s.

To simplify the presentation, we use $kr \rightarrow r$, $r \rightarrow kr$, and $s \rightarrow t$ to represent the first, the second, and the third types of redistribution, respectively, for the rest of the paper. In this section, we first present the terminology used in this paper and then describe efficient methods for $kr \rightarrow r$ and $r \rightarrow kr$ redistribution.

Definition 1: Given a BLOCK-CYCLIC(s) to BLOCK-CYCLIC(t) redistribution, BLOCK-CYCLIC(s), BLOCK-CYCLIC(t), s, and t are called the *source distribution*, the *destination distribution*, the *source distribution factor*, and the *destination distribution factor* of the redistribution, respectively.

Definition 2: Given an $s \rightarrow t$ redistribution on $A[1:N]$ over M processors, the *source local array* of processor P_i, denoted by $SLA_i[0:N/M-1]$, is defined as the set of array elements that are distributed to processor P_i in the source distribution, where $0 \leq i \leq M-1$. The *destination local array* of processor P_j, denoted by $DLA_j[0:N/M-1]$, is defined as the set of array elements that are distributed to processor P_j in the destination distribution, where $0 \leq j \leq M-1$.

Definition 3: Given an $s \rightarrow t$ redistribution on $A[1:N]$ over M processors, the *source processor* of an array element in $A[1:N]$ or $DLA_j[0:N/M-1]$ is defined as the processor that owns the array element in the source distribution, where $0 \leq j \leq M-1$. The *destination processor* of an array element in $A[1:N]$ or $SLA_i[0:N/M-1]$ is defined as the processor that owns the array element in the destination distribution, where $0 \leq i \leq M-1$.

Definition 4: Given an $s \rightarrow t$ redistribution on $A[1:N]$ over M processors, we define $SG:SLA_i[m] \rightarrow A[k]$ is a function that converts a source local array element $SLA_i[m]$ of P_i to its corresponding global array element $A[k]$ and $DG:DLA_j[n] \rightarrow A[l]$ is a function that converts a destination local array element $DLA_j[n]$ of P_j to its corresponding global array element $A[l]$, where $1 \leq k, l \leq N$ and $0 \leq m, n \leq N/M-1$.

Definition 5: Given an $s \rightarrow t$ redistribution on $A[1:N]$ over M processors, a global complete cycle (GCC) of $A[1:N]$ is defined as M times the least common multiple of s and t, i.e., $GCC = M \times lcm(s, t)$. We define $A[1:GCC]$ as the first global complete cycle of $A[1:N]$, $A[GCC+1:2 \times GCC]$ as the second global complete cycle of $A[1:N]$, and so on.

Definition 6: Given an $s \rightarrow t$ redistribution, a *local complete cycle* (LCC) of a local array $SLA_i[0:N/M-1]$ (or $DLA_j[0:N/M-1]$) is defined as the least common multiple of s and t, i.e., $LCC = lcm(s, t)$. We define $SLA_i[0:LCC-1]$ ($DLA_j[0:LCC-1]$) as the first local complete cycle of $SLA_i[0:N/M-1]$ ($DLA_j[0:N/M-1]$), $SLA_i[LCC:2 \times LCC-1]$ ($DLA_j[LCC:2 \times LCC-1]$) as the second local complete cycle of of $SLA_i[0:N/M-1]$ ($DLA_j[0:N/M-1]$), and so on.

Definition 7: Given an $s \rightarrow t$ redistribution, for a source processor P_i (or destination processor P_j), a *class* is defined as the set of array elements in an LCC of SLA_i (DLA_j) with the same destination (or source) processor. The *class size* is defined as the number of array elements in a class.

Given a one-dimensional array $A[1:30]$ and $M = 3$ processors, Figure 2(a) shows a BLOCK-CYCLIC($s = 10$) to BLOCK-CYCLIC($t = 2$) redistribution on A over M pro-

Source : BLOCK-CYCLIC(10)										
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
P_0	1	2	3	4	5	6	7	8	9	10
P_1	11	12	13	14	15	16	17	18	19	20
P_2	21	22	23	24	25	26	27	28	29	30

Destination : BLOCK-CYCLIC(2)										
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
P_0	1	2	7	8	13	14	19	20	25	26
P_1	3	4	9	10	15	16	21	22	27	28
P_2	5	6	11	12	17	18	23	24	29	30

(a)

$\text{DPS}[P_0] = \{P_0, P_1, P_2\}$

$\text{SDS}[P_0, P_0] = \{(l_0, A[1]), (l_1, A[2]), (l_6, A[7]), (l_7, A[8])\}$, $\text{SDS}[P_0, P_1] = \{(l_2, A[3]), (l_3, A[4]), (l_8, A[9]), (l_9, A[10])\}$, $\text{SDS}[P_0, P_2] = \{(l_4, A[5]), (l_5, A[6])\}$

$\text{SPS}[P_0] = \{P_0, P_1, P_2\}$

$\text{RDS}[P_0, P_0] = \{(l_0, A[1]), (l_1, A[2]), (l_2, A[7]), (l_3, A[8])\}$, $\text{RDS}[P_0, P_1] = \{(l_4, A[13]), (l_5, A[14]), (l_6, A[19]), (l_7, A[20])\}$, $\text{RDS}[P_0, P_2] = \{(l_8, A[25]), (l_9, A[26])\}$

(b)

Figure 2. (a) A BLOCK-CYCLIC (10) to BLOCK-CYCLIC (2) redistribution on a one-dimensional array $A[1:30]$ over 3 processors. (b) The send/receive data sets and the source/destination processor sets that are computed by processor P_0.

cessors. In this paper, we assume that the local array index starts from 0 and the global array index starts from 1. In Figure 2(a), we use the italic numbers and the normal numbers to represent the local array indices and the global array indices, respectively. In Figure 2(a), the global complete cycle (*GCC*) is 30 and the local complete cycle (*LCC*) is 10. For source processor P_0, array elements $SLA_0[0, 1, 6, 7]$, $SLA_0[2, 3, 8, 9]$, and $SLA_0[4, 5]$ are classes in the first *LCC* of SLA_0. The size of three classes $SLA_0[0, 1, 6, 7]$, $SLA_0[2, 3, 8, 9]$, and $SLA_0[4, 5]$ are equal to 4, 4 and 2 respectively.

To perform the redistribution shown in Figure 2(a), in general, a processor needs to compute the send data sets, the receive data sets, the source processor set, and the destination processor set. Figure 2(b) illustrates these sets that are computed by processor P_0 for the redistribution shown in Figure 2(a). In Figure 2(b), element $(l_4, A[5])$ in $\text{SDS}[P_0, P_2]$ denotes that the source local array element with index = 4 of P_0 is $A[5]$, which will be sent to processor P_2. Element $(l_8, A[25])$ in $\text{RDS}[P_0, P_2]$ denotes that the array element $A[25]$ that received from P_2 should be put in $DLA_0[8]$. In the send phase, processor P_0 sends data to P_0, P_1, and P_2. The sets of array elements that P_0 will send to P_0, P_1, and P_2 are $\{A[1], A[2], A[7], A[8]\}$, $\{A[3], A[4], A[9], A[10]\}$, and $\{A[5], A[6]\}$, respectively. Since a processor has known the send data set for each destination processor, it only needs to pack these data into messages and send messages to their corresponding destination processors. In the receive phase, processor P_0 receives messages from P_0, P_1,

and P_2. When it receives messages from its source processors, it unpacks these messages by placing elements in messages to their appropriate local array positions according to the receive data sets.

The method mentioned above is not efficient at all. When the array size is large, the computation overheads is great in computing the send/receive data sets. In fact, for $kr \rightarrow r$ and $r \rightarrow kr$ array redistribution, we can derive packing and unpacking information that allows one to pack and unpack array elements without calculating the send/receive data sets. In the following subsections, we will describe how to derive the packing and unpacking information for $kr \rightarrow r$ and $r \rightarrow kr$ array redistribution.

3.1. $kr \rightarrow r$ redistribution

3.1.1. Send phase.

Lemma 1: Given an $s \rightarrow t$ redistribution on $A[1:N]$ over M processors, $SLA_i[m]$, $SLA_i[m + LCC]$, $SLA_i[m + 2 \times LCC]$, …, and $SLA_i[m + (N/GCC - 1) \times LCC]$ have the same destination processor, where $0 \leq i \leq M - 1$ and $0 \leq m \leq LCC - 1$.

Proof In a $kr \rightarrow r$ redistribution, $GCC = M \times lcm(s,t)$ and $LCC = lcm(s,t)$. In the source distribution, for a source processor P_i, if the global array index of $SLA_i[m]$ is α, then the global array indices of $SLA_i[m + LCC]$, $SLA_i[m + 2 \times LCC]$, …, and $SLA_i[m + (N/GCC - 1) \times LCC]$ are $\alpha + GCC$, $\alpha + 2 \times GCC$, …, and $\alpha + (N/GCC - 1) \times GCC$, respectively, where $0 \leq i \leq M - 1$, $0 \leq m \leq LCC - 1$. Since $GCC = M \times lcm(s,t)$ and $LCC = lcm(s,t)$, in the destination distribution, if $A[\alpha]$ is distributed to the destination processor P_j, so are $A[\alpha + GCC]$, $A[\alpha + 2 \times GCC]$, …, and $A[\alpha + (N/GCC - 1) \times GCC]$, where $0 \leq j \leq M - 1$ and $1 \leq \alpha \leq GCC$. ■

Lemma 2: Given a $kr \rightarrow r$ redistribution on $A[1:N]$ over M processors, for a source processor P_i and array elements in $SLA_i[x \times LCC:(x + 1) \times LCC - 1]$, if the destination processor of $SLA_i[x \times LCC]$ is P_j, then the destination processors of $SLA_i[x \times LCC: x \times LCC + r - 1]$, $SLA_i[x \times LCC + r: x \times LCC + 2r - 1]$, …, $SLA_i[x \times LCC + (k - 1) \times r: x \times LCC + kr - 1]$ are P_j, $P_{mod(j+1,M)}$, …, $P_{mod(j+k-1,M)}$, respectively, where $0 \leq x \leq N/GCC - 1$ and $0 \leq i, j \leq M - 1$.

Proof In a $kr \rightarrow r$ redistribution, LCC is equal to kr. In the source distribution, for each source processor P_i, array elements in $SLA_i[x \times LCC:(x + 1) \times LCC - 1]$ have consecutive global array indices, where $0 \leq x \leq N/GCC - 1$ and $0 \leq i \leq M - 1$. Therefore, in the destination distribution, if $SLA_i[x \times LCC]$ is distributed to processor P_j, then $SLA_i[x \times LCC: x \times LCC + r - 1]$ will be distributed to P_j. Since the destination distribution is in BLOCK-CYCLIC(r) fashion, $SLA_i[x \times LCC + r: x \times LCC + 2r - 1]$, $SLA_i[x \times LCC + 2r: x \times LCC + 3r - 1]$, …, $SLA_i[x \times LCC + (k - 1) \times r: x \times LCC + kr - 1]$ will be distributed to processor $P_{mod(j+1,M)}$, $P_{mod(j+2,M)}$, …, $P_{mod(j+k-1,M)}$, respectively, where $0 \leq x \leq N/GCC - 1$ and $0 \leq i, j \leq M - 1$. ■

Given a $kr \rightarrow r$ redistribution on $A[1:N]$ over M processors, for a source processor P_i, if the destination processor for the first array element of SLA_i is P_j, according to Lemma 2, array elements in $SLA_i[0:r-1]$, $SLA_i[r:2r-1]$, …, and $SLA_i[LCC-r:LCC-1]$ will be sent to destination processors P_j, $P_{mod(j+1,M)}$, …, and $P_{mod(j+k-1,M)}$, respectively, where $0 \leq i, j \leq M-1$. From Lemma 1, we know that $SLA_i[0:r-1]$, $SLA_i[LCC:LCC+r-1]$, $SLA_i[2 \times LCC:2 \times LCC+r-1]$, …, and $SLA_i[(N/GCC-1) \times LCC:(N/GCC-1) \times LCC+r-1]$ have the same destination processor. Therefore, if we know the destination processor of $SLA_i[0]$, according to Lemmas 1 and 2, we can pack array elements in SLA_i to messages directly without computing the send data sets and the destination processor set. For example, a BLOCK-CYCLIC(6) to BLOCK-CYCLIC(2) redistribution on $A[1:24]$ over $M = 2$ processors is shown in Figure 3(a). In this example, for source processor P_0, the destination processor of $SLA_0[0]$ is P_0. According to Lemma 2, $SLA_0[0, 1, 4, 5]$ and $SLA_0[2, 3]$ should be packed to messages msg_0 and msg_1 which will be sent to destination processors P_0 and P_1, respectively. From Lemma 1, $SLA_0[6, 7, 10, 11]$, and $SLA_0[8, 9]$ will also be packed to messages msg_0 and msg_1, respectively. Figure 3(b) shows the messages packed by each source processor.

Given a $kr \rightarrow r$ redistribution over M processors, for a source processor P_i, the destination processor for the first array element of SLA_i can be computed by the following equation:

$$\eta = mod(rank(P_i) \times k, M) \tag{1}$$

where η is the destination processor for the first array element of SLA_i and $rank(P_i)$ is the rank of processor P_i.

3.1.2. Receive phase.

Lemma 3: Given a $kr \rightarrow r$ redistribution on $A[1:N]$ over M processors, for a source processor P_i and array elements in $SLA_i[x \times LCC:(x+1) \times LCC-1]$, if the destination processor of $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, …, $SG(SLA_i[a_{\gamma-1}])$ is P_j, then $SG(SLA_i[a_0])$,

Source : BLOCK-CYCLIC(6)

index	0	1	2	3	4	5	6	7	8	9	10	11
p_0	1	2	3	4	5	6	13	14	15	16	17	18
p_1	7	8	9	10	11	12	19	20	21	22	23	24

Destination : BLOCK-CYCLIC(2)

index	0	1	2	3	4	5	6	7	8	9	10	11
p_0	1	2	5	6	9	10	13	14	17	18	21	22
p_1	3	4	7	8	11	12	15	16	19	20	23	24

msg_0	1	2	5	6	13	14	17	18
msg_1	3	4	15	16				

Messages sent by P_0
(a)

msg_0	9	10	21	22				
msg_1	7	8	11	12	19	20	23	24

Messages sent by P_1
(b)

Figure 3. (a) A BLOCK-CYCLIC(6) to BLOCK-CYCLIC(2) redistribution on $A[1:24]$ over $M = 2$ processors. (b) Messages packed by source processors.

$SG(SLA_i[a_1])$, …, $SG(SLA_i[a_{\gamma-1}])$ are in the consecutive local array positions of $DLA_j[0:N/M-1]$, where $0 \le i, j \le M-1$, $0 \le x \le N/GCC-1$, and $x \times LCC \le a_0 < a_1 < a_2 <\ldots< a_{\gamma-1} < (x+1) \times LCC$.

Proof In a $kr \rightarrow r$ redistribution, LCC is equal to kr. In the source distribution, for each source processor P_i, array elements in $SLA_i[x \times LCC:(x+1) \times LCC-1]$ have consecutive global array indices, where $0 \le i \le M-1$ and $0 \le x \le N/GCC-1$. Therefore, in the destination distribution, if $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, …, $SG(SLA_i[a_{\gamma-1}])$ will be distributed to processor P_j and $SG(SLA_i[a_0]) = DG(DLA_j[\alpha])$, then $SG(SLA_i[a_1]) = DG(DLA_j[\alpha+1])$, $SG(SLA_i[a_2]) = DG(DLA_j[\alpha+2])$, …, $SG(SLA_i[a_{\gamma-1}]) = DG(DLA_j[\alpha+\gamma-1])$, where $0 \le i, j \le M-1$, $0 \le x \le N/GCC-1$ and $x \times LCC \le a_0 < a_1 < a_2 <\ldots< a_{\gamma-1} < (x+1) \times LCC$. ■

Lemma 4: Given a $kr \rightarrow r$ redistribution on $A[1:N]$ over M processors, for a source processor P_i, if $SLA_i[a]$ and $SLA_i[b]$ are the first array element of $SLA_i[x \times LCC:(x+1) \times LCC-1]$ and $SLA_i[(x+1) \times LCC:(x+2) \times LCC-1]$, respectively, with the same destination processor P_j and $SG(SLA_i[a]) = DG(DLA_j[\alpha])$, then $SG(SLA_i[b]) = DG(DLA_j[\alpha+kr])$, where $0 \le i, j \le M-1$, $0 \le x \le N/GCC-2$, and $0 \le \alpha \le N/M-1$.

Proof In a $kr \rightarrow r$ redistribution, GCC and LCC are equal to Mkr and kr, respectively. In the source distribution, for a source processor P_i, if $SLA_i[a]$ and $SLA_i[b]$ are the first array element of $SLA_i[x \times LCC:(x+1) \times LCC-1]$ and $SLA_i[(x+1) \times LCC:(x+2) \times LCC-1]$, respectively, with the same destination processor P_j, according to Lemma 1, $SLA_i[b] = SLA_i[a+LCC]$, where $0 \le i, j \le M-1$ and $0 \le x \le N/GCC-2$. Furthermore, if $SG(SLA_i[a])$ is $A[u]$, then $SG(SLA_i[a+LCC])$ is $A[u+GCC]$, where $1 \le u \le N$. In the destination distribution, since $LCC = kr$ and $GCC = Mkr$, the number of array elements distributes to each destination processor in a global complete cycle of $A[1:N]$ is kr. Therefore, if $A[u] = DG(DLA_j[\alpha])$, then $A[u+GCC] = DG(DLA_j[\alpha+kr])$, where $0 \le \alpha \le N/M-1$. ■

Given a $kr \rightarrow r$ redistribution on $A[1:N]$ over M processors, for a destination processor P_j, if the first element of a message (assume that it was sent by source processor P_i) will be unpacked to $DLA_j[\alpha]$ and there are γ array elements in $DLA_j[0:LCC-1]$ whose source processor is P_i, according to Lemmas 3 and 4, the first γ array elements of the message will be unpacked to $DLA_j[\alpha:\alpha+\gamma-1]$, the second γ array elements of the message will be unpacked to $DLA_j[\alpha+kr:\alpha+kr+\gamma-1]$, the third γ array elements of the message will be unpacked to $DLA_j[\alpha+2kr:\alpha+2kr+\gamma-1]$, and so on, where $0 \le i, j \le M-1$ and $0 \le \alpha \le LCC$. Therefore, for a destination processor P_j, if we know the values of γ (the number of array elements in $DLA_j[0:LCC-1]$ whose source processor is P_i) and α (the position to place the first element of a message in DLA_j), we can unpack elements in messages to DLA_j without computing the receive data sets and the source processor set. For the redistribution shown in Figure 3(a), Figure 4 shows how a destination processor P_0 unpacks messages using the unpacking information (values of α and γ). In this example, for destination processor P_0, values of (α, γ) for messages msg_0 and msg_1 that are received from source processors P_0 and P_1 are (0, 4) and (4, 2),

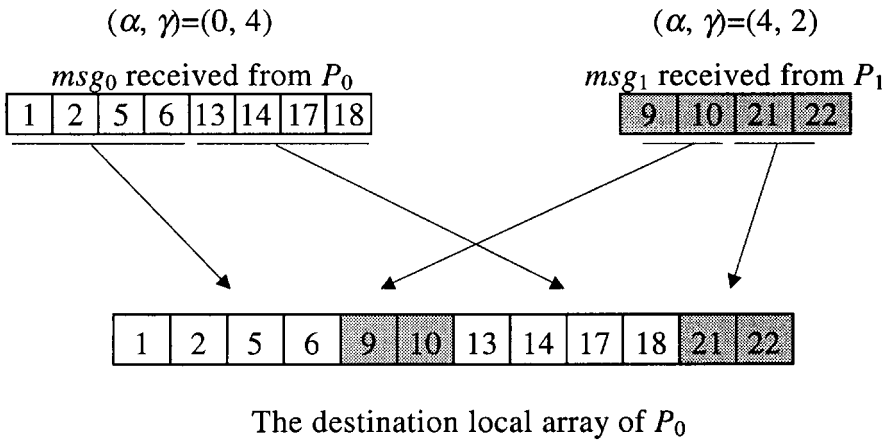


Figure 4. Unpack messages using the unpacking information.

respectively. Therefore, destination processor P_0 unpacks the first 4 elements of msg_0 to $DLA_0[0:3]$ and the second 4 elements of msg_0 to $DLA_0[6:9]$. The first and the second 2 elements of msg_1 will be unpacked to $DLA_0[4:5]$ and $DLA_0[10:11]$, respectively.

Given a $kr \rightarrow r$ redistribution on $A[1:N]$ over M processors, for a destination processor P_j, the values of α and γ can be computed by the following equations:

$$\gamma = (\lfloor k/M \rfloor + \Gamma[mod((rank(P_j) + M - mod(rank(P_i) \times k, M)), M) < mod(k, M)]) \times r \quad (2)$$

$$\alpha = (\lfloor rank(P_i) \times k/M \rfloor + \Gamma[(rank(P_j) < mod(rank(P_i) \times k, M)]) \times r \quad (3)$$

Where $rank(P_i)$ and $rank(P_j)$ are the ranks of processors P_i and P_j. $\Gamma[e]$ is called Iverson's function. If the value of e is true, then $\Gamma[e] = 1$; otherwise $\Gamma[e] = 0$.

The $kr \rightarrow r$ redistribution algorithm is described as follows.

Algorithm $kr \rightarrow r\_redistribution(k, r, M)$

/***Send phase***/
1. $i = MPI\_Comm\_rank(\ )$;
2. max_local_index = the length of the source local array of processor P_i;
3. the destination processor of $SLA_0[0]$ is $\eta = (k \times i)\ mod\ M$;

/***Packing data sets***/
4. $index = 1$; $length_\delta = 1$, where $\delta = 0, \ldots, M - 1$;
5. **while** ($index \leq max\_local\_index$)
6. { $\delta = \eta$; $j = 1$;
7. **while** (($j \leq k$) && ($index \leq max\_local\_index$))
8. { $l = 1$;

```
9.              while ((l ≤ r) && (index ≤ max_local_index))
10.             { out_buffer_δ[length_δ++] = SLA_i[index++];
11.               l++; }
12.             j++; if (δ = M) δ = 0 else δ++; }
13.     }
14. Send out_buffer_δ to processor P_δ, where δ = 0, …, M − 1;
/*Receive phase*/
15. max_cycle = max_local_index / kr;
16. Repeat m = min (M, k) times
17.     Receive message buffer_in_i from source processor P_i;
18.     Calculate the value of γ for message buffer_in_i using Equation (2);
19.     Calculate the value of α for message buffer_in_i using Equation (3);
        /*Unpacking messages*/
20.     index = α; length = 1; j = 0;
21.     while (j ≤ max_cycle)
22.       { index = α + j × kr; l = 1;
23.         while (l ≤ γ)
24.         { DLA_i[index++] = buffer_in_i[length++];
25.           l++; }
26.         j++; }
end_of_kr → r_redistribution
```

3.2. $r \rightarrow kr$ redistribution

3.2.1. Send phase.

Lemma 5: Given an $r \rightarrow kr$ redistribution on $A[1:N]$ over M processors, for a source processor P_i and array elements in $SLA_i[x \times LCC:(x+1) \times LCC - 1]$, if the destination processor of $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, …, $SG(SLA_i[a_{n-1}])$ is P_j, then $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, …, $SG(SLA_i[a_{n-1}])$ are in the consecutive local array positions of $SLA_i[0:N/M-1]$, where $0 \le i, j \le M-1$, $0 \le x \le N/GCC - 1$, and $x \times LCC \le a_0 < a_1 < a_2 < \ldots < a_{n-1} < (x+1) \times LCC$.

Proof In an $r \rightarrow kr$ redistribution, LCC is equal to kr. For a destination processor P_j, array elements in $DLA_j[x \times LCC:(x+1) \times LCC - 1]$ have consecutive global array indices, where $0 \le j \le M-1$ and $0 \le x \le N/GCC - 1$. Therefore, in the source distribution, if $DG(DLA_j[a_0])$, $DG(DLA_j[a_1])$, …, $DG(DLA_j[a_{n-1}])$ are distributed to source processor P_i in the source distribution, and $DG(DLA_j[a_0]) = SG(SLA_i[v])$, then $DG(DLA_j[a_1]) = SG(SLA_i[v+1])$, $DG(DLA_j[a_2]) = SG(SLA_i[v+2])$, …, $DG(DLA_j[a_{n-1}]) = SG(SLA_i[v+n-1])$, where $0 \le i, j \le M-1$, $0 \le x \le N/GCC - 1$, and $x \times LCC \le a_0 < a_1 < a_2 < \ldots < a_{n-1} < (x+1) \times LCC$. ■

Given an $r \rightarrow kr$ redistribution on $A[1:N]$ over M processors, for a source processor P_i, if the destination processor for the first array element of SLA_i is P_j and there are u classes,

C_1, C_2, C_3, …, and C_u in $SLA_i[0{:}LCC - 1]$ (assume that the indices of local array elements in these classes have the order $C_1 < C_2 < C_3 <\ldots< C_u$ and the destination processors of C_1, C_2, C_3, …, and C_u are P_{j_1}, P_{j_2}, P_{j_3}, …, and P_{j_u}, respectively), according to Lemma 5, we know that

$$j_1 = j,$$

$$j_2 = mod((|C_1| \times M)/kr + j_1, M),$$

$$j_3 = mod((|C_2| \times M)/kr + j_2, M),$$

$$\vdots$$

$$j_u = mod((|C_{u-1}| \times M)/kr + j_{u-1}, M),$$

where $1 \le u \le min(k, M)$ and $|C_1|$, …, $|C_{u-1}|$ are class size of C_1, …, C_{u-1}, respectively. This means that array elements $SLA_i[0{:}|C_1| - 1]$ will be sent to destination processor P_{j_1}, array elements $SLA_i[|C_1| : |C_1| + |C_2| - 1]$ will be sent to destination processor P_{j_2}, …, and array elements $SLA_i[|C_1| + |C_2| +\ldots+ |C_{u-1}| : |C_1| + |C_2| +\ldots+ |C_u| - 1]$ will be sent to destination processor P_{j_u}. From Lemma 1, we know that $SLA_i[0{:}\ |C_1| - 1]$, $SLA_i[LCC{:}LCC + |C_1| - 1]$, $SLA_i[2 \times LCC{:}2 \times LCC + |C_1| - 1]$, …, and $SLA_i[(N/GCC - 1) \times LCC{:}(N/GCC - 1) \times LCC + |C_1| - 1]$ have the same destination processor. Therefore, if we know the destination processor of $SLA_i[0]$ and the values of $(|C_1|, P_{j_1})$, $(|C_2|, P_{j_2})$, …, and $(|C_u|, P_{j_u})$, we can pack array elements in SLA_i to messages directly without computing the send data sets and the destination processor set. For example, a BLOCK-CYCLIC(2) to BLOCK-CYCLIC(6) redistribution on $A[1{:}24]$ over $M = 2$ processors is shown in Figure 5(a). In this example, for source processor P_0, the destination processor of $SLA_0[0]$ is P_0. There are two classes C_1 and C_2 in $SLA_i[0{:}LCC - 1]$. The destination processors of C_1 and C_2 are P_0 and P_1, respectively. The size of classes C_0 and C_1 are 4 and 2, respectively. According to Lemma 5, $SLA_0[0, 1, 2, 3]$, and $SLA_0[4,$

Source : BLOCK-CYCLIC(2)

index	0	1	2	3	4	5	6	7	8	9	10	11
p_0	1	2	5	6	9	10	13	14	17	18	21	22
p_1	3	4	7	8	11	12	15	16	19	20	23	24

Destination : BLOCK-CYCLIC(6)

index	0	1	2	3	4	5	6	7	8	9	10	11
p_0	1	2	3	4	5	6	13	14	15	16	17	18
p_1	7	8	9	10	11	12	19	20	21	22	23	24

msg_0	1	2	5	6	13	14	17	18
msg_1	9	10	21	22				

Messages sent by P_0
(a)

msg_0	3	4	15	16				
msg_1	7	8	11	12	19	20	23	24

Messages sent by P_1
(b)

Figure 5. (a) A BLOCK-CYCLIC(2) to BLOCK-CYCLIC(6) redistribution on $A[1{:}24]$ over $M = 2$ processors. (b) Messages packed by source processors.

5] should be packed to messages msg_0 and msg_1 which will be sent to destination processors P_0 and P_1, respectively. From Lemma 1, $SLA_0[6, 7, 8, 9]$, and $SLA_0[10, 11]$ will also be packed to messages msg_0 and msg_1, respectively. Figure 5(b) shows the messages packed by each source processor.

Given an $r \rightarrow kr$ redistribution on $A[1:N]$ over M processors, for a source processor P_i, the destination processor for the first array element of SLA_i can be computed by equation (5) and the number of array elements in $SLA_i[0:LCC-1]$ whose destination processor is P_j can be computed by equation (4). Equations (4) and (5) are given as follows:

$$|C_j| = (\lfloor k/M \rfloor + \Gamma[mod(rank(P_i) + M - mod(rank(P_j) \times k,M)),M) < mod(k,M)]) \times r \tag{4}$$

$$\varphi = \lfloor rank(P_i)/k \rfloor \tag{5}$$

Where φ is the destination processor for the first array element of SLA_i and $rank(P_i)$ and $rank(P_j)$ are the ranks of processors P_i and P_j, respectively. $\Gamma[e]$ is the Iverson's function defined in Equations 2 and 3.

3.2.2. Receive phase.

Lemma 6: Given an $r \rightarrow kr$ redistribution on $A[1:N]$ over M processors, for a source processor P_i and array elements in $SLA_i[x \times LCC:(x+1) \times LCC - 1]$, if the destination processor of $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, …, $SG(SLA_i[a_{n-1}])$ is P_j, then array elements of $SG(SLA_i[a_0])$, …, $SG(SLA_i[a_{r-1}])$; $SG(SLA_i[a_r])$, …, $SG(SLA_i[a_{2r-1}])$; …; and $SG(SLA_i[a_{n-r}])$, …, $SG(SLA_i[a_{n-1}])$ are in the consecutive local array positions of $DLA_j[0:N/M-1]$, where $0 \le i, j \le M-1$, $0 \le x \le N/GCC - 1$ and $x \times LCC \le a_0 < a_1 < a_2 < \ldots < a_{n-1} < (x+1) \times LCC$. Furthermore, if $SG(SLA_i[a_0]) = DG(DLA_j[v])$, then $SG(SLA_i[a_r]) = DG(DLA_j[v + Mr])$, $SG(SLA_i[a_{2r}]) = DG(DLA_j[v + 2Mr])$, …, and $SG(SLA_i[a_{n-r}]) = DG(DLA_j[v + (n/r - 1) \times Mr])$, where $0 \le v \le N/M - 1$.

Proof In an $r \rightarrow kr$ redistribution, $GCC = Mkr$ and $LCC = kr$. In the source distribution, for each source processor P_i, every r array elements in $SLA_i[x \times LCC:(x+1) \times LCC - 1]$ have consecutive global array indices, where $0 \le i \le M-1$ and $0 \le x \le N/GCC - 1$. Since $LCC = kr$, in the destination distribution, if $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, …, $SG(SLA_i[a_{n-1}])$ will be distributed to processor P_j, then array elements of $SG(SLA_i[a_0])$, …, $SG(SLA_i[a_{r-1}])$; $SG(SLA_i[a_r])$, …, $SG(SLA_i[a_{2r-1}])$; …; and $SG(SLA_i[a_{n-r}])$, …, $SG(SLA_i[a_{n-1}])$ are in the consecutive local array positions of $DLA_j[0:N/M-1]$, where $0 \le i, j \le M-1$, $0 \le x \le N/GCC - 1$ and $x \times LCC \le a_0 < a_1 < a_2 < \ldots < a_{n-1} < (x+1) \times LCC$.

Since in the source distribution, for each source processor P_i, every r array elements in $SLA_i[x \times LCC:(x+1) \times LCC - 1]$ have consecutive global array indices, if $SG(SLA_i[a_0]) = A[\beta]$, then $SG(SLA_i[a_r]) = A[\beta + Mr]$, $SG(SLA_i[a_{2r}]) = A[\beta + 2Mr]$, …, and $SG(SLA_i[a_{n-r}]) = A[\beta + (n/r - 1) \times Mr]$, where $1 \le \beta \le N-1$, $0 \le x \le N/GCC$

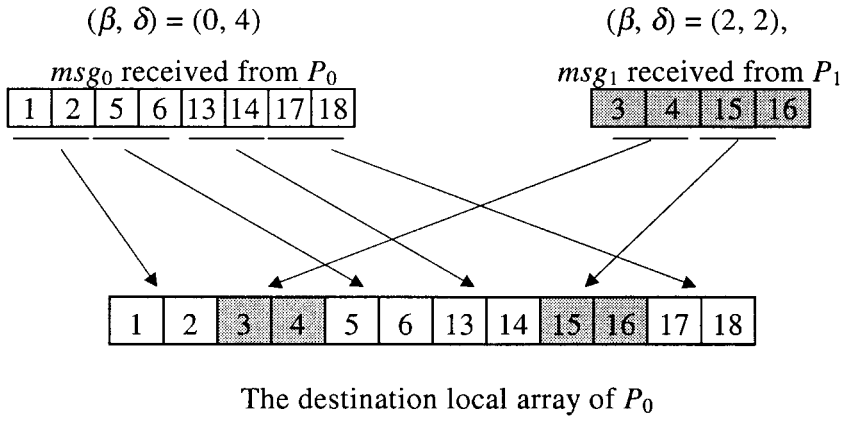


Figure 6. Unpack messages using the unpacking information.

$- 1$, and $x \times LCC \leq a_0 < a_1 < a_2 < \ldots < a_{n-1} < (x + 1) \times LCC$. Since the destination processor of $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, …, $SG(SLA_i[a_{n-1}])$ is P_j, in the destination distribution, if $A[\beta] = DG(DLA_j[v])$, we have $A[\beta + Mr] = DG(DLA_j[v + Mr])$, $A[\beta + 2Mr] = DG(DLA_j[v + 2Mr])$, …, and $A[\beta + (n/r - 1) \times Mr] = DG(DLA_j[v + (n/r - 1) \times Mr])$, where $0 \leq v \leq N/M - 1$. ■

Given an $r \rightarrow kr$ redistribution on $A[1:N]$ over M processors, for a destination processor P_j, if the first array element of the message (assume it was sent by source processor P_i) will be unpacked to $DLA_j[\beta]$ and there are δ array elements in $DLA_j[0:LCC - 1]$ whose source processor is P_i. According to Lemma 6, the first δ array elements of this message will be unpacked to $DLA_j[\beta:\beta + r - 1]$, $DLA_j[\beta + Mr:\beta + Mr + r - 1]$, $DLA_j[\beta + 2Mr:\beta + 2Mr + r - 1]$, …, and $DLA_j[\beta + (\delta/r - 1) \times Mr:\beta + (\delta/r - 1) \times Mr + r - 1]$; the second δ array elements of the message will be unpacked to $DLA_j[\beta + kr:\beta + kr + r - 1]$, $DLA_j[\beta + kr + Mr:\beta + kr + Mr + r - 1]$, $DLA_j[\beta + kr + 2Mr:\beta + kr + 2Mr + r - 1]$, …, and $DLA_j[\beta + kr + (\delta/r - 1) \times Mr:\beta + kr + (\delta/r - 1) \times Mr + r - 1]$, and so on, where $0 \leq i, j \leq M - 1$ and $0 \leq \beta \leq N/M - 1$. Therefore, if we know the values of δ (the number of array elements in $DLA_j[0:LCC - 1]$ whose source processor is P_i) and β (the position to place the first element of a message in DLA_j), we can unpack messages to DLA_j without computing the receive data sets and the source processor set. For the redistribution shown in Figure 5(a), Figure 6 shows how a destination processor P_0 unpacks messages using the unpacking information (values of β and δ). In this example, for destination processor P_0, values of (β, δ) for messages msg_0 and msg_1 that are received from source processors P_0 and P_1 are (0, 4) and (2, 2), respectively. Therefore, destination processor P_0 unpacks the first 2 elements of msg_0 to $DLA_0[0:1]$ and the second 2 elements of msg_0 to $DLA_0[4:5]$. The third and the fourth 2 elements of msg_0 will be unpacked to $DLA_0[6:7]$ and $DLA_0[10:11]$, respectively. The first and second 2 elements of msg_1 will be unpacked to $DLA_0[2:3]$ and $DLA_0[8:9]$, respectively.

Given an $r \rightarrow kr$ redistribution on $A[1:N]$ over M processors, for a destination processor P_j, the values of β and δ can be computed by the following equations:

$$\delta = (\lfloor k/M \rfloor + \Gamma[mod((M + rank(P_i) - mod(rank(P_j) \times k, M)), M) < mod(k, M)]) \times r \quad (6)$$

$$\beta = mod(M + rank(P_i) - mod(rank(P_j) \times k, M), M) \times r \quad (7)$$

Where $rank(P_i)$ and $rank(P_j)$ are the ranks of processors P_i and P_j, respectively. $\Gamma[e]$ is the Iverson's function defined in Equations 2 and 3.

The $r \rightarrow kr$ redistribution algorithm can be described as follows.

Algorithm $r \rightarrow kr\_redistribution(k, r, M,)$

/***Send phase***/
1. $i = MPI\_Comm\_rank(\ )$;
2. max_local_index = the length of the source local array of processor P_i;
3. the destination processor of $SLA_i[0]$ is $\varphi = i/k$;
4. $m = min(k, M)$; $j_1 = \varphi$;
5. Calculate $j_2, j_3, \ldots j_m$;
6. Calculate class size $|C_{jw}|$ using Equation (4), where $w = 1, \ldots, m$;
/***Packing data sets***/
7. $index = 1$; $length_j = 1$, where $j = 0, \ldots, M - 1$;
8. **while** ($index \leq max\_local\_index$)
9. { $t = 1$;
10. **while**(($t \leq m$)&&($index \leq max\_local\_index$))
11. { $j = j_t$; $l = 1$;
12. **while**(($l \leq |C_j|$) && ($index \leq max\_local\_index$))
13. { $out\_buffer_j[length_j{+}{+}] = SLA_i[index{+}{+}]$;
14. $l{+}{+}$;}
15. $t{+}{+}$; }
16. }
17. Send out_buffer_j to processor P_j, where $j = j_1, j_2, \ldots j_m$.
/***Receive phase***/
18. max_cycle = max$_local_index$ divided by kr
19. **Repeat** $m = min\ (M, k)$ times
20. Receive message $buffer\_in_i$ from source processors P_i.
21. Calculate the value of δ for $buffer\_in_i$ using Equation (6);
22. Calculate the value of β for $buffer\_in_i$ using Equation (7);
/***Unpacking data sets***/
23. $index = \beta$; $length = 1$; $j = 0$; $count = 0$;
24. **while** ($j \leq max\_cycle$)
25. { $count = 1$; $index = \beta + j \times kr$;
26. **while** ($count \leq \delta$)

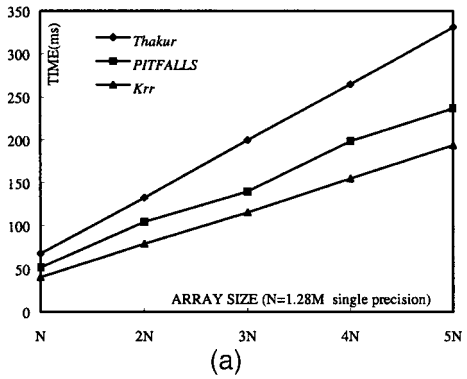
```
27.      { l = 1;
28.        while (l ≤ r)
29.          { DLA_i[index++] = buffer_in[length++];
30.                count ++; l++; }
31.        index + = (M − 1) × r; }
32.      j++; }
end_of_kr → r_redistribution
```

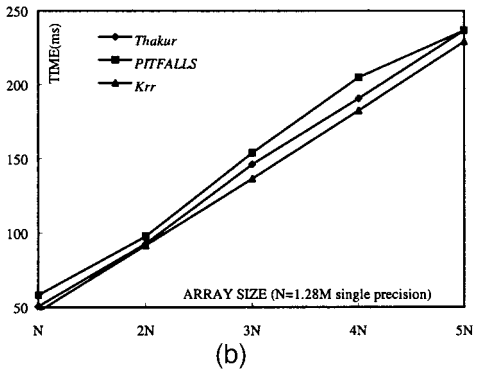
4. Performance evaluation and experimental results

To evaluate the performance of the proposed algorithms, we have implemented the proposed methods along with the Thakur's methods [24, 25] and the *PITFALLS* method [21, 22] on an IBM SP2 parallel machine. All algorithms were written in the single program multiple data (SPMD) programming paradigm with C + MPI codes. To get the experimental results, we have executed those programs for different kinds of $kr \rightarrow r$ and $r \rightarrow kr$ array redistribution with various array size N on a 64-node IBM SP2 parallel machine, where $N \in \{1.28M, 2.56M, 3.84M, 5.12M, 6.4M\}$ and $k \in \{5, 25, 50, 100, N/64\}$. For a particular redistribution, all algorithms were executed 20 times. The mean time of the 20 tests was used as the time of an algorithm to perform the redistribution. Time was measured by using *MPI_Wtime*( ). The single-precision array was used for the test. The experimental results were shown in Figure 7 to Figure 11. In Figure 7 to Figure 11, the *Krr* represents the algorithms proposed in this paper. The *Thakur* and the *PITFALLS* represent the algorithms proposed in [24, 25] and [21,22], respectively.

Figure 7 gives the execution time of these algorithms to perform BLOCK-CYCLIC(10) to BLOCK-CYCLIC(2) and BLOCK-CYCLIC(2) to BLOCK-CYCLIC(10) redistribution with various array size, where $k = 5$. In Figure 7(a), the execution time of these three algorithms has the order $T(Krr) < T(PITFALLS) < T(Thakur)$. From Figure 7(c), for the $kr \rightarrow r$ redistribution, we can see that the computation time of these three algorithms has the order $T_{comp}(Krr) < T_{comp}(Thakur) < T_{comp}(PITFALLS)$. For the *PITFALLS* method, a processor needs to find out all intersections between source and destination distribution with all other processors involved in the redistribution. Therefore, the *PITFALLS* method requires additional computation time at communication sets calculation. For the *Thakur's* method, a processor needs to scan its local array elements once to determine the destination (source) processor for each block of array elements of size r in the local array. The *Thakur's* method also requires additional computation time at communication sets calculation. However, for the *Krr* method, based on the packing/unpacking information derived from the $kr \rightarrow r$ redistribution, it can pack/unpack array elements to/from messages directly without calculating the communication sets. Therefore, the computation time of the *Krr* method is the lowest one among these three methods.

For the same case, the communication time of these three algorithms has the order $T_{comm}(Krr) < T_{comm}(PITFALLS) < T_{comm}(Thakur)$. For the *Krr* method and the *PITFALLS* method, both methods use asynchronous communication schemes. The computation and the communication overheads can be overlapped. However, the *Krr* method unpacks any





	$kr \rightarrow r$						$r \rightarrow kr$					
	Thakur		*PITFALLS*		*Krr*		*Thakur*		*PITFALLS*		*Krr*	
	Comp.	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*
N	36.364	31.795	41.660	10.170	32.126	8.529	41.534	8.933	42.043	15.052	38.183	8.582
2*N*	75.133	57.815	81.918	22.843	57.657	21.754	67.439	25.301	70.945	27.761	65.766	25.832
3*N*	103.638	96.154	114.089	26.120	92.106	23.863	101.317	44.727	104.990	48.896	93.624	42.967
4*N*	148.964	115.740	167.094	42.100	119.046	36.100	134.124	56.519	145.050	59.785	114.663	54.100
5*N*	195.568	135.642	175.198	61.680	144.551	49.408	169.880	66.725	163.399	75.375	162.268	67.003

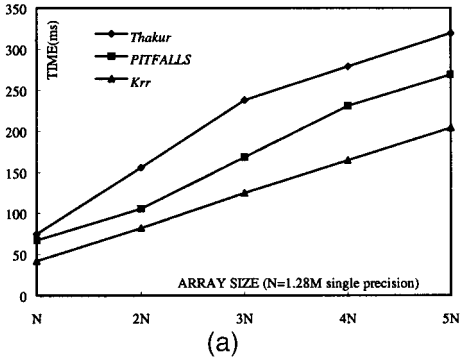
Time(ms)

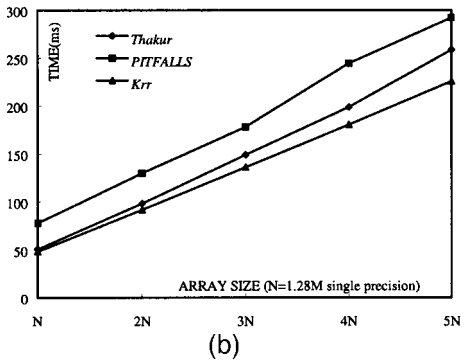
(c)

Figure 7. Performance of different algorithms to execute a BLOCK-CYCLIC(10) to BLOCK-CYCLIC(2) redistribution and vice versa with various array size (N = 1.28Mbytes) on a 64-node SP2. (a) The $kr \rightarrow r$ redistribution. (b) The $r \rightarrow kr$ redistribution. (c) The computation time and the communication time of (a) and (b).

received messages in the receiving phase while the *PITFALLS* method unpacks messages in a specific order. Therefore, the communication time of the *Krr* method is less than or equal to that of the *PITFALLS* method. For the *Thakur's* method, due to the algorithm design strateagy, it uses a synchronous communication scheme in the $kr \rightarrow r$ redistribution. In a synchronous communication scheme, the computation and the communication overheads can not be overlapped. Therefore, the *Thakur's* method has higher communication overheads than those of the *Krr* method and the *PITFALLS* method.

Figure 7(b) presents the exetution time of these algorithms for the $r \rightarrow kr$ redistribution. The execution time of these three algorithms has the order $T(Krr) < T(Thakur) < T(PITFALLS)$. In Figure 7(c), for the $r \rightarrow kr$ redistribution, the computation time of these three algorithms have the order $T_{comp}(Krr) < T_{comp}(Thakur) < T_{comp}(PITFALLS)$. The reason is similar to that described for Figure 7(a).





	kr→*r*						*r*→*kr*					
	Thakur		*PITFALLS*		*Krr*		*Thakur*		*PITFALLS*		*Krr*	
	Comp.	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*
N	34.932	39.847	43.805	23.501	28.336	13.536	34.413	16.346	48.389	29.418	31.594	16.384
2*N*	70.091	85.882	73.827	31.833	56.374	25.425	68.778	29.285	85.114	44.655	63.102	28.704
3*N*	118.017	120.472	135.841	32.920	94.381	30.287	99.957	48.886	124.429	53.545	94.800	41.287
4*N*	141.519	137.484	173.722	57.684	113.757	42.100	152.308	46.638	160.021	84.648	126.625	42.100
5*N*	179.806	139.634	209.926	59.450	151.592	52.920	191.431	67.190	219.808	72.622	158.830	66.876

Time(ms)

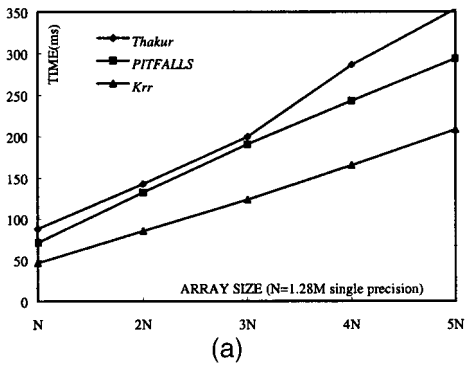
(c)

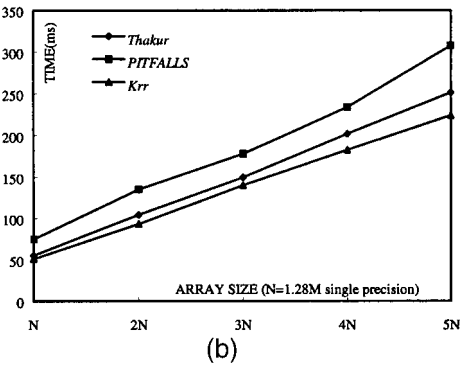
Figure 8. Performance of different algorithms to execute a BLOCK-CYCLIC(50) to BLOCK-CYCLIC(2) redistribution and vice versa with various array size (N = 1.28M single precision) on a 64-node SP2. (a) The $kr \rightarrow r$ redistribution. (b) The $r \rightarrow kr$ redistribution. (c) The computation time and the communication time of (a) and (b).

For the communication time, the *Thakur's* method and the *Krr* method have similar communication overheads and are less than that of the *PITFALLS* method. In the $r \rightarrow kr$ redistribution, all these three algorithms use asynchronous communication schemes. However, the *Krr* method and the *Thakur's* method unpack any received message in the receiving phase while the *PITFALLS* method unpacks messages in a specific order. Therefore, the *PITFALLS* method has more communication overheads than those of the *Krr* method and the *Thakur's* method.

Figures 8, 9, and 10 are the cases when k is equal to 25, 50, and 100, respectively. From Figure 8 to Figure 10, we have similar observations as those described for Figure 7.

Figure 11 gives the execution time of these algorithms to perform BLOCK to CYCLIC and vice versa redistribution with various array size. In this case, the value of k is equal to $N/64$. From Figure 11(a) and (b), we can see that the execution time of these three algorithms has the order $T(Krr) < T(Thakur) \ll T(PITFALLS)$ for both $kr \rightarrow r$ and $r \rightarrow kr$ redistribution. In Figure 11(c), for both $kr \rightarrow r$ and $r \rightarrow kr$ redistribution, the compu-





	$kr \rightarrow r$						$r \rightarrow kr$					
	Thakur		*PITFALLS*		*Krr*		*Thakur*		*PITFALLS*		*Krr*	
	Comp.	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*
N	34.996	52.990	69.069	19.960	28.638	17.704	35.690	19.942	48.628	26.227	31.754	19.590
2*N*	70.299	72.605	92.376	40.094	58.666	26.697	67.776	36.661	71.928	63.404	64.235	29.450
3*N*	101.153	98.645	148.98	41.841	85.748	37.670	101.702	48.266	122.045	56.000	96.599	43.544
4*N*	135.619	151.076	172.798	70.446	115.733	42.100	155.148	47.329	188.372	45.633	125.682	42.100
5*N*	189.154	164.476	201.033	92.933	140.428	68.043	188.093	63.568	223.770	84.105	165.061	59.070

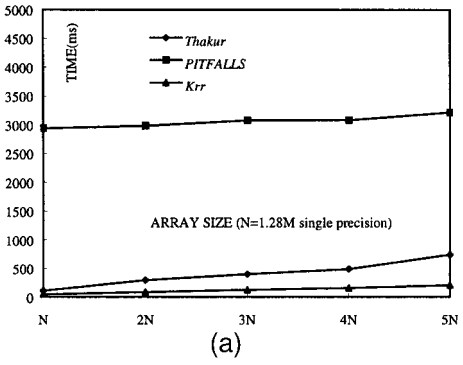
Time(ms)

(c)

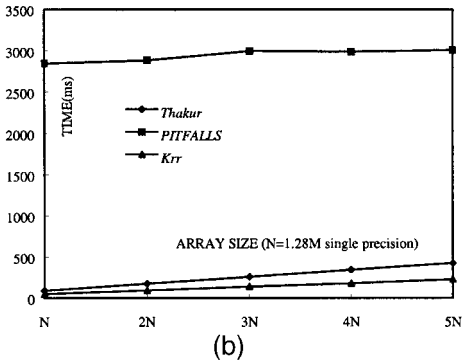
Figure 9. Performance of different algorithms to execute a BLOCK-CYCLIC(100) to BLOCK-CYCLIC(2) redistribution and vice versa with various array size (N = 1.28M single precision) on a 64-node SP2. (a) The $kr \rightarrow r$ redistribution. (b) The $r \rightarrow kr$ redistribution. (c) The computation time and the communication time of (a) and (b).

tation time of theses three algorithms has the order $T_{comp}(Krr) < T_{comp}(\text{Thakur}) \ll T_{comp}(PITFALLS)$. The *PITFALLS* method has very large computation time compared to those of the *Krr* method and the *Thakur's* method. Th reason is that each processor needs to find out all intersections between source and destination distribution with all other processors in the *PITFALLS* method. The computation time of the *PITFALLS* method depends on the number of intersections. In this case, there are $N/64$ intersections between each source and destination processor. Therefore, a processor needs to compute $\lfloor N/64 \rfloor \times 64$ intersections which demands a lot of computation time when N is large. For the communication overheads, we have similar observations as those described for Figure 7(b).

From the above performance analysis and experimental results, we can see that the *Krr* method outperforms the *Thakur's* method and the *PITFALLS* method for all test samples.



(a)



(b)

	$B \rightarrow C$						$C \rightarrow B$					
	Thakur		*PITFALLS*		*Krr*		*Thakur*		*PITFALLS*		*Krr*	
	Comp.	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*
N	60.995	47.957	2604.27	23.433	25.217	21.403	71.295	20.927	2752.87	32.108	31.453	20.512
$2N$	132.927	61.863	2737.06	37.795	49.658	35.874	141.443	34.662	2774.36	43.613	57.855	34.786
$3N$	207.400	91.779	2829.74	49.372	73.716	48.813	218.876	39.928	2834.00	64.959	101.388	37.1490
$4N$	332.264	126.576	2784.76	59.221	99.822	42.100	277.103	45.309	2839.74	82.462	106.374	42.100
$5N$	308.744	134.895	2921.13	93.937	123.557	81.721	348.445	78.006	2904.35	96.881	156.440	72.246

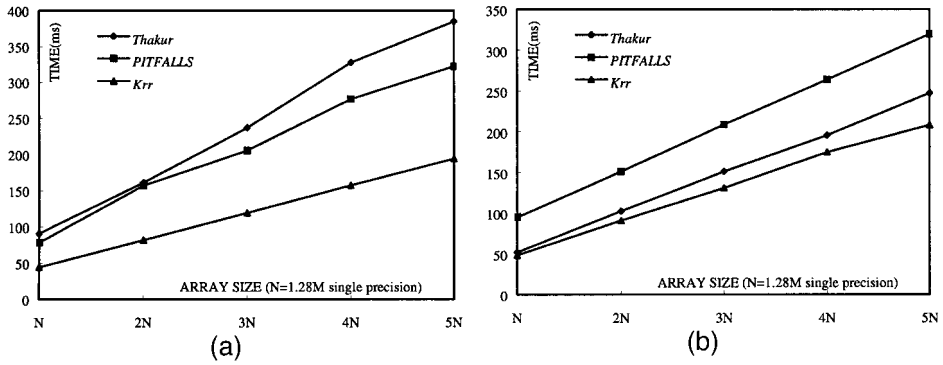
Time(ms)

(c)

Figure 11. Performance of different algorithms to execute a BLOCK to CYCLIC redistribution and vice versa with various array size (N = 1.28M single precision) on a 64-node SP2. (a) The BLOCK to CYCLIC redistribution. (b) The CYCLIC to BLOCK redistribution. (c) The computation time and the communication time of (a) and (b).

5. Conclusions

Array redistribution is usually used in data-parallel programs to minimizing the run-time cost of performing data exchange among different processors. Since it is performed at run-time, efficient methods are required for array redistribution. In this paper, we have presented efficient algorithms for $kr \rightarrow r$ and $r \rightarrow kr$ redistribution. The most significant improvement of our algorithms is that a processor does not need to construct the send/receive data sets for a redistribution. Based on the packing/unpacking information that derived from the $kr \rightarrow r$ and $r \rightarrow kr$ redistribution, a processor can pack/unpack array elements to (from) messages directly. To evaluate the performance of our methods, we have implemented our methods along with the Thakur's method and the *PITFALLS* method on an IBM SP2 parallel machine. The experimental results show that our algorithms outperform the Thakur's methods and the *PITFALLS* method. This result encourages us to use the proposed algorithms for array redistribution.



	$kr \rightarrow r$						$r \rightarrow kr$					
	Thakur		*PITFALLS*		*Krr*		*Thakur*		*PITFALLS*		*Krr*	
	Comp.	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*	*Comp.*	*Comm.*
N	34.242	56.689	45.490	33.120	28.089	16.140	33.440	18.615	67.028	27.673	29.489	18.615
2N	69.344	92.207	100.996	56.120	53.568	28.459	65.311	36.997	104.210	46.929	58.536	32.359
3N	123.528	113.691	137.049	68.470	76.972	42.646	112.608	38.567	142.419	66.427	92.456	38.358
4N	148.702	179.254	194.375	82.695	103.479	42.100	150.642	45.269	181.675	82.260	123.773	42.100
5N	187.916	197.152	234.207	88.120	135.932	58.247	181.544	65.916	233.561	85.919	148.389	59.917

Time(ms)

(c)

Figure 10. Performance of different algorithms to execute a BLOCK-CYCLIC(200) to BLOCK-CYCLIC(2) redistribution and vice versa with various array size (N = 1.28M single precision) on a 64-node SP2. (a) The $kr \rightarrow r$ redistribution. (b) The $r \rightarrow kr$ redistribution. (c) The computation time and the communication time of (a) and (b).

Acknowledgements

The work of this paper was partially supported by NSC under contract NSC87-2213-E-035-011.

References

1. S. Benkner. Handling block-cyclic distribution arrays in Vienna Fortran 90. In *Proceeding of Intl. Conf. on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 1995.
2. B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic data distribution in Vienna Fortran. *Proc. of Supercomputing'93*, pp. 284–293. Nov. 1993.
3. S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating Local Address and Communication Sets for Data Parallel Programs. *Journal of Parallel and Distributed Computing*, vol. 26, pp. 72–84. 1995.

4. Y.-C Chung, C.-S Sheu and S.-W Bai. A Basic-Cycle Calculation Technique for Efficient Dynamic Data Redistribution. In *Proceedings of Intl. Computer Symposium on Distributed Systems*, pp. 137–144. Dec. 1996.
5. J. J. Dongarra, R. Van De Geijn, and D. W. Walker. A look at scalable dense linear algebra libraries. Technical Report ORNL/TM-12126 from Oak Ridge National Laboratory. Apr. 1992.
6. 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.
7. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On the Generation of Efficient Data Communication for Distributed-Memory Machines. *Proc. of Intl. Computing Symposium*, pp. 504–513. 1992.
8. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines. *Journal of Parallel and Distributed Computing*, vol. 32, pp. 155–172. 1996.
9. High Performance Fortran Forum. High Performance Fortran Language Specification(version 1.1). Rice University. November 1994.
10. S. Hiranandani, K. Kennedy, J. Mellor-Crammey, and A. Sethi. Compilation technique for block-cyclic distribution. In *Proc. ACM Intl. Conf. on Supercomputing*, pp. 392–403. July 1994.
11. Edgar T. Kalns, and Lionel M. Ni. Processor Mapping Technique Toward Efficient Data Redistribution. *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 12. December 1995.
12. E. T. Kalns and L. M. Ni, DaReL: A portable data redistribution library for distributed-memory machines. In *Proceedings of the 1994 Scalable Parallel Libraries Conference II*. Oct. 1994.
13. S. D. Kaushik, C. H. Huang, R. W. Johnson, and P. Sadayappan. An Approach to communication efficient data redistribution. In *Proceeding of International Conf. on Supercomputing*, pp. 364–373. July 1994.
14. S. D. Kaushik, C. H. Huang, J. Ramanujam, and P. Sadayappan. Multiphase array redistribution: Modeling and evaluation. In *Proceeding of International Parallel processing Symposium*, pp. 441–445. 1995.
15. S. D. Kaushik, C. H. Huang, and P. Sadayappan. Efficient Index Set Generation for Compiling HPF Array Statements on Distributed-Memory Machines. *Journal of Parallel and Distributed Computing*, vol. 38, pp. 237–247. 1996.
16. K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distribution. In *Proceeding of International Conf. on Supercomputing*, pp. 180–184, Barcelona. July 1995.
17. P-Z. Lee and W. Y. Chen. Compiler techniques for determining data distribution and generating communication sets on distributed-memory multicomputers. *29th IEEE Hawaii Intl. Conf. on System Sciences*, Maui, Hawaii. pp.537–546. Jan 1996.
18. Young Won Lim, Prashanth B. Bhat, and Viktor K. Prasanna. Efficient Algorithms for Block-Cyclic Redistribution of Arrays. *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pp. 74–83. 1996.
19. Y. W. Lim, N. Park, and V. K. Prasanna. Efficient Algorithms for Multi-Dimensional Block-Cyclic Redistribution of Arrays. *Proceedings of the 26th International Conference on Parallel Processing*, pp. 234–241. 1997.
20. L. Prylli and B. Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *Journal of Parallel and Distributed Computing*, vol. 45, pp. 63–72. Aug. 1997.
21. S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. Frontier'95: *The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pp. 342–349. Mclean, VA., Feb. 1995.
22. S. Ramaswamy, B. Simons, and P. Banerjee. Optimization for Efficient Array Redistribution on Distributed Memory Multicomputers. *Journal of Parallel and Distributed Computing*, vol. 38, pp. 217–228. 1996.
23. J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, vol. 21, pp. 150–159. 1994.

24. R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. *Proc. 1994 Scalable High Performance Computing Conf.*, pp. 309–316. May 1994.
25. Rajeev. Thakur, Alok. Choudhary, and J. Ramanujam. Efficient Algorithms for Array Redistribution. *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 6. June 1996.
26. A. Thirumalai and J. Ramanujam. HPF array statements: Communication generation and optimization. 3th workshop on Languages, Compilers and Run-time system for Scalable Computers, Troy. NY. May 1995.
27. V. Van Dongen, C. Bonello and C. Freehill. High Performance C - Language Specification Version 0.8.9. Technical Report CRIM-EPPP-94/04–12. 1994.
28. C. Van Loan. Computational Frameworks for the Fast Fourier Transform. *SIAM*, 1992.
29. David W. Walker, and Steve W. Otto. Redistribution of BLOCK-CYCLIC Data Distributions Using MPI. *Concurrency: Practice and Experience*, 8.9:707–728, Nov. 1996.
30. Akiyoshi Wakatani and Michael Wolfe. A New Approach to Array Redistribution: Strip Mining Redistribution. In *Proceeding of Parallel Architectures and Languages* Europe. July 1994.
31. Akiyoshi Wakatani and Michael Wolfe. Optimization of Array Redistribution for Distributed Memory Multicomputers. In *Parallel Computing* (submitted). 1994.
32. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - A Language Specification Version 1.1. ICASE Interim Report 21. ICASE NASA Langley Research Center, Hampton, Virginia 23665. March, 1992.