# Packing/Unpacking Information Generation for Efficient Generalized *kr*→*r* and *r*→*kr* Array Redistribution

*Ching-Hsien Hsu, Yeh-Ching Chung, and Chyi-Ren Dow*

Department of Information Engineering
Feng Chia University, Taichung, Taiwan 407, R.O.C.
Tel: 886-4-4517250 x3700
Fax: 886-4-4516101
Email: chhsu, ychung, crdow@pine.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 methods to generate the packing/unpacking information for `BOLCK-CYCLIC`(kr) to `BLOCK-CYCLIC`(r) and `BOLCK-CYCLIC`(r) to `BLOCK-CYCLIC`(kr) redistribution with arbitrary source/destination processor sets. The most significant improvement of this paper is that a processor does not need to construct the send/receive data sets for a redistribution. Based on the packing/unpacking information derived from kr→r and r→kr redistributions, 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 PITFALLS method and the Prylli's method on an IBM SP2 parallel machine. The experimental results show that our algorithms outperform the PITFALLS method and the Prylli's method for all test samples.*

Keywords: Array redistribution, packing/unpacking information, distributed memory multicomputers.

## 1. Introduction

In some algorithms, such as multi-dimensional fast Fourier transform, the Alternative Direction Implicit (ADI) method for solving two-dimensional diffusion equations, and linear algebra solvers, 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. 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. Many methods for performing array redistribution were proposed in the literature [2-3, 6-8, 11-14, 16-19]. Due to the page limitation, we will not describe these methods here. The details of these methods can be found in [2].

In this paper, we present efficient methods to generate the packing/unpacking information for `BOLCK-CYCLIC`(*kr*) to `BLOCK-CYCLIC`(*r*) and `BOLCK-CYCLIC`(*r*) to `BLOCK-CYCLIC`(*kr*) redistribution. The proposed methods have the following characteristics.

- Based on the packing/unpacking information that derived from `BOLCK-CYCLIC`(*kr*) to `BLOCK-CYCLIC`(*r*) redistribution and vice versa, a processor can pack/unpack array elements into (from) messages without calculating send/receive processor/data sets.
- The time to generate the packing/unpacking information is independent of the array size involved in a redistribution. Therefore, the indexing overhead is small.
- The generated packing and unpacking information tables are optimized. This optimization can reduce the memory copy time when performing the packing and unpacking processes.
- The proposed methods use an asynchronous communication scheme to send/receive messages in any order. Since the computation and the communication time is overlapped, this leads to a better performance for a redistribution.

The rest of this paper is organized as follows. In Section 2, we will introduce notations and terminology used in this paper. Sections 3 and 4 present the techniques for `BLOCK-CYCLIC`(*kr*) to `BLOCK-CYCLIC`(*r*) and `BLOCK-CYCLIC`(*r*) to `BLOCK-CYCLIC`(*kr*) redistribution, respectively. The performance evaluation will be given in Section 5.

## 2. Preliminaries

In general, a `BLOCK-CYCLIC`(*s*) over *P* processors to `BLOCK-CYCLIC`(*t*) over *Q* processors 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 $kr_{(P)}\rightarrow r_{(Q)}$, $r_{(P)}\rightarrow kr_{(Q)}$, and $s_{(P)}\rightarrow t_{(Q)}$ to represent the first, the second, and the third types of redistribution, respectively. In this section, we present the terminology used in this paper.

Definition 1: Given an $s_{(P)}\rightarrow t_{(Q)}$ redistribution on $A[1:N]$, the *source local array* of processor $P_i$, denoted by $SLA_i[0:N/P-1]$, is defined as the set of array elements that are distributed to processor $P_i$ in the source distribution, where $0 \le i \le P-1$. The *destination local array* of processor $Q_j$, denoted by $DLA_j[0:N/Q-1]$, is defined as the set of array elements that are distributed to processor $Q_j$ in the destination distribution, where $0 \le j \le Q-1$.

Definition 2: Given an $s_{(P)}\rightarrow t_{(Q)}$ redistribution on $A[1:N]$, a *global complete cycle* ($GCC$) of $A[1:N]$ is defined as $GCC = lcm(s\times P, t\times Q)$. 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 3: Given an $s_{(P)}\rightarrow t_{(Q)}$ redistribution on $A[1:N]$, a *local complete cycle* of a local array is defined as $LCC_s = GCC/P$ in the source distribution and $LCC_d = GCC/Q$ in the destination distribution. We define $SLA_i[0:LCC_s-1]$ ($DLA_j[0:LCC_d-1]$) as the first local complete cycle of $SLA_i[0:N/P-1]$ ($DLA_j[0:N/Q-1]$), $SLA_i[LCC_s:2\times LCC_s-1]$ ($DLA_j[LCC_d:2\times LCC_d-1]$) as the second local complete cycle of of $SLA_i[0:N/P-1]$ ($DLA_j[0:N/Q-1]$), and so on.

Definition 4: Given an $s_{(P)}\rightarrow t_{(Q)}$ redistribution, a local complete cycle of a source (destination) local array can be divided into $LCC_s/s$ ($LCC_d/t$) blocks. We define $SLA_i[0:s-1]$ ($DLA_j[0:t-1]$) as the first *source* (*destination*) *section* of $SLA_i[0:LCC_s-1]$ ($DLA_j[0:LCC_d-1]$) of processor $P_i$ ($P_j$), $SLA_i[s:2s-1]$ ($DLA_j[t:2t-1]$) as the second *source* (*destination*) *section* of $SLA_i[0:LCC_s-1]$ ($DLA_j[0:LCC_s-1]$) of processor $P_i$ ($P_j$), and so on.

Definition 5: Given a $s_{(P)}\rightarrow t_{(Q)}$ redistribution, for a source processor $P_i$ (or destination processor $Q_j$), a *class* is defined as the set of array elements with the same destination (or source) processor in a section of $SLA_i$ (or $DLA_j$). The *class size* is defined as the number of array elements in a class.

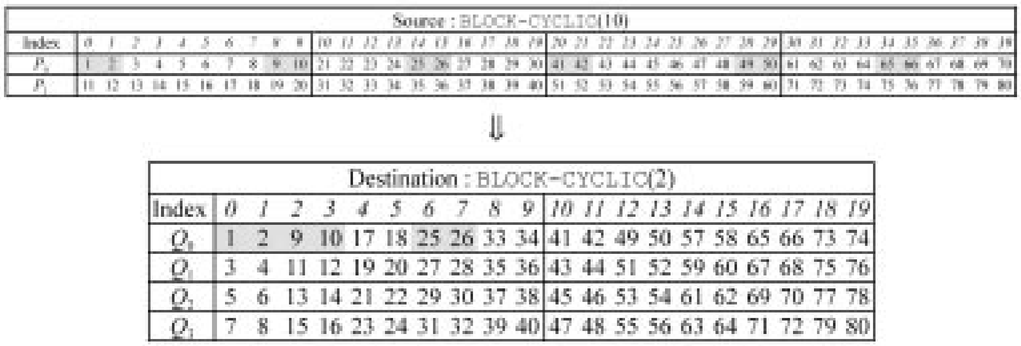
Fig. 1 shows a BLOCK-CYCLIC(10) over two processors ($P$=2) to BLOCK-CYCLIC(2) over four processors ($Q$=4) redistribution on a one-dimensional array $A[1:80]$. In Fig. 1, the global complete cycle ($GCC$) is 40. The local complete cycle is $LCC_s$=20 in the source distribution and $LCC_d$=10 in the destination distribution. For source processor $P_0$, there are two sections (size = 10) in each local complete cycle. In the first section, there are four classes $SLA_0[0, 1, 8, 9]$, $SLA_0[2, 3]$, $SLA_0[4, 5]$ and $SLA_0[6, 7]$. The size of these four classes $SLA_0[0, 1, 8, 9]$, $SLA_0[2, 3]$, $SLA_0[4, 5]$ and $SLA_0[6, 7]$ are equal to 4, 2, 2, and 2, respectively. In the second section, there are four classes $SLA_0[14, 15]$, $SLA_0[16, 17]$, $SLA_0[10, 11, 18, 19]$ and $SLA_0[12, 13]$. The size of these four classes $SLA_0[14, 15]$, $SLA_0[16, 17]$, $SLA_0[10, 11, 18, 19]$ and $SLA_0[12, 13]$ are equal to 2, 2, 4, and 2, respectively.

To perform the redistribution shown in Fig. 1, in general, a processor needs to compute the send/receive data sets and the destination/source processor set. A naive way to get those sets is to scan every array element once and to compute those sets. Since the redistribution is performed at run-time, if an array size is very large, the time to determine those sets by scanning every array element once may greatly offset the performance of a program by performing the redistribution. Many methods use the repetitive nature of global complete cycle [11] to construct the communication sets only for the first global complete cycle. However, these methods can not handle the cases when the source and the destination processor sets are different. In [13, 14], even these methods can handle arbitrary number of source and destination processors, they still have one shortcoming. In these methods, each processor needs to find out all intersections between source and destination distribution with all other processors. The computation time depends on the number of intersections. When the difference of the block size of the source distribution and that of the destination distribution is large, the number of intersections becomes large as well. For example, in a BLOCK-CYCLIC(12) over two processors to BLOCK-CYCLIC(2) over four processors array redistribution, source processor $P_0$ will send $SLA_0[0, 1, 8, 9]$ to the destination processor $Q_0$ in the first local complete cycle. To get the address sequence of $SLA_0[0, 1, 8, 9]$, $P_0$ needs to compute two intersections, [0,1] and [8,9]. If the source distribution factor was scaled from BLOCK-CYCLIC(12) to BLOCK-CYCLIC(120), a processor will need to compute twenty intersections which will demand a lot of computation time. In fact, for $kr_{(P)}\rightarrow r_{(Q)}$ and $r_{(P)}\rightarrow kr_{(Q)}$ 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 sections, we will describe how to derive the packing and unpacking information for $kr_{(P)}\rightarrow r_{(Q)}$ and $r_{(P)}\rightarrow kr_{(Q)}$ array redistribution.

## 3. $kr_{(P)}\rightarrow r_{(Q)}$ Array Redistribution

### 3.1 Send Phase

We first use the example shown in Fig. 1 to describe our method. From Fig. 1, we have some observations.



Source : BLOCK-CYCLIC(10)

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_0$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| $P_1$ | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |

| Index | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_0$ | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| $P_1$ | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |

$\Downarrow$

Destination : BLOCK-CYCLIC(2)

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Q_0$ | 1 | 2 | 9 | 10 | 17 | 18 | 25 | 26 | 33 | 34 | 41 | 42 | 49 | 50 | 57 | 58 | 65 | 66 | 73 | 74 |
| $Q_1$ | 3 | 4 | 11 | 12 | 19 | 20 | 27 | 28 | 35 | 36 | 43 | 44 | 51 | 52 | 59 | 60 | 67 | 68 | 75 | 76 |
| $Q_2$ | 5 | 6 | 13 | 14 | 21 | 22 | 29 | 30 | 37 | 38 | 45 | 46 | 53 | 54 | 61 | 62 | 69 | 70 | 77 | 78 |
| $Q_3$ | 7 | 8 | 15 | 16 | 23 | 24 | 31 | 32 | 39 | 40 | 47 | 48 | 55 | 56 | 63 | 64 | 71 | 72 | 79 | 80 |

Fig. 1: A `BLOCK-CYCLIC`(10) over two processors to `BLOCK-CYCLIC`(2) over four processors array redistribution on a one-dimensional array $A[1:80]$.

- Observation 3.1: Each local complete cycle have the same communication patterns. For example, for source processor $P_0$, array elements $SLA_0[0]$, $SLA_0[1]$, $SLA_0[8]$, $SLA_0[9]$, $SLA_0[14]$, and $SLA_0[15]$ in the first $LCC_s$ of $SLA_0$ will be sent to destination processor $Q_0$. In this example, $LCC_s$ is equal to 20. From Fig. 1, we can see that array elements $SLA_0[0+20]$, $SLA_0[1+20]$, $SLA_0[8+20]$, $SLA_0[9+20]$, $SLA_0[14+20]$, and $SLA_0[15+20]$ in the second $LCC_s$ of $SLA_0$ will also be sent to destination processor $Q_0$.
- Observation 3.2: For each source processor $P_i$, every $r$ elements of a class have consecutive local array positions in $SLA_i$. For example, for source processor $P_0$, array elements $SLA_0[0, 1, 8, 9]$ form a class in the first section of $SLA_0[0{:}LCC_s-1]$. Since $r$ is equal to two, we an see that $SLA_0[0, 1]$ and $SLA_0[8, 9]$ are in the consecutive local array positions of $SLA_0$. Array elements $SLA_0[14, 15]$ form a class in the second section of $SLA_0[0{:}LCC_s-1]$. We also see that $SLA_0[14, 15]$ are in the consecutive local array positions of $SLA_0$.
- Observation 3.3: For each source processor $P_i$, if the class size of a class is larger than $r$, then the difference of the indices of array elements in the same position of the $i$th and the $(i+1)$th $r$ array elements of the class is $Qr$. For example, for source processor $P_0$, the class size of $SLA_0[0, 1, 8, 9]$ is four. Since $r$ is equal to two, the first array elements in the first and the second $r$=2 array elements of the class are $SLA_0[0]$ and $SLA_0[8]$, respectively. The difference of their indices is $Qr$=8. So are $SLA_0[1]$ and $SLA_0[9]$.

Given a $kr_{(P)} \to r_{(Q)}$ redistribution, for a source processor $P_i$, if the destination processor for a class $SLA_i[\beta_0, \beta_1, \ldots, \beta_{\alpha-1}]$ is $Q_j$, where $\beta_0, \beta_1, \ldots, \beta_{\alpha-1}$ are indices of array elements in the class, $\beta_0 < \beta_1 < \ldots < \beta_{\alpha-1}$, $\beta_0$ is the first index of array elements in the class, and $\alpha$ is the class size; according to Observation 3.1, source processor $P_i$ will pack array elements $SLA_i[\beta_0, \ldots, \beta_0+r-1]$, $SLA_i[\beta_0+Qr, \ldots, \beta_0+r-1+Qr]$, ..., and $SLA_i[\beta_0+(\alpha/r-1)\times Qr, \ldots, \beta_0+r-1+(\alpha/r-1)\times Qr]$ to the message which will be sent to destination processor $Q_j$. From Observation 3.2, we know that array elements $SLA_i[\beta_0, \ldots, \beta_{\alpha-1}]$, $SLA_i[LCC+\beta_0, \ldots, LCC+\beta_{\alpha-1}]$, …, and $SLA_i[(N/GCC-1)\times LCC+\beta_0, \ldots, (N/GCC-1)\times LCC+\beta_{\alpha-1}]$ have the same destination processor. Therefore, if we know the class size and the index of the first array element in a class, according to Observations 3.1 and 3.2, we can pack array elements in $SLA_i$ to messages directly without computing the send data sets and the destination processor set. For example, in Fig. 1, for source processor $P_0$, array elements $SLA_0[0, 1, 8, 9]$ form a class in the first section of $SLA_0[0{:}LCC_s-1]$. Since the class size is equal to 4 and its first array element's index is equal to 0, according to Lemma 2, processor $P_0$ will pack array elements $SLA_0[0, 1]$ and $SLA_0[8, 9]$ to message $msg_0$ which will be sent to destination processors $Q_0$. In the second section of $SLA_0[0{:}LCC_s-1]$, array elements $SLA_0[14, 15]$ form a class. The class size is equal to 2 and its first array element's index is equal to 14. Processor $P_0$ packs array elements $SLA_0[14, 15]$ to the message $msg_0$. According to Observation 3.1, each local complete cycle has the same communication patterns. $SLA_0[20, 21, 28, 29]$, and $SLA_0[34, 35]$ will also be packed to messages $msg_0$ as shown in Fig. 2(a). Messages $msg_1$, $msg_2$ and $msg_3$ that will be sent to destination processors $Q_1$, $Q_2$, and $Q_3$, respectively, by source processor $P_0$ can be packed in a similar manner and are shown in Fig. 2(b).

Given a $kr_{(P)} \to r_{(Q)}$ redistribution, if we denote the class size and the index of the first array element in a class as *CS* and *FI*, respectively, we can gather these information to form a *packing information table* (*PIT*). Fig. 3 shows the packing information table of source processor $P_0$ for the redistribution shown in Fig. 1. Since each local complete cycle has two sections in the redistribution shown in Fig. 1, there are two entries of packing information (*CS* and *FI*) for each message. According to the packing information table, a source processor can pack array elements to messages directly without calculating the send processor/data sets.

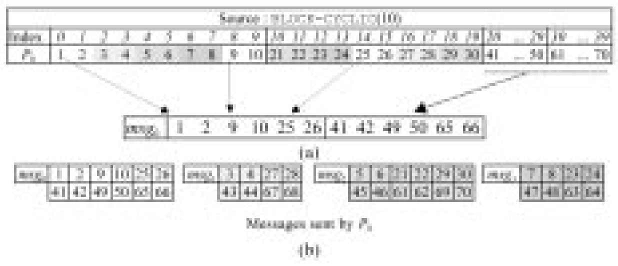


Fig. 2: Packing array elements to messages for the example shown in Fig. 1. (a) Message $msg_0$ packed by source processor $P_0$. (b) Messages packed by source processor $P_0$.

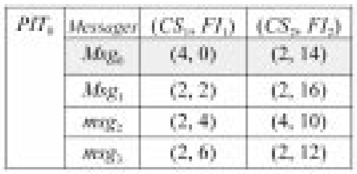
In the following, we describe how to derive the packing information table for $kr_{(P)} \rightarrow r_{(Q)}$ redistribution.

Given a $kr_{(P)} \rightarrow r_{(Q)}$ redistribution, for each source processor $P_i$, a local complete cycle ($LCC_s$) can be divided into $m$ sections, where $m = \frac{LCC_s}{kr}$. A source processor $P_i$ can construct the packing information table by the following steps:

1. For each section $u$, do steps 2 to 4, where $u$ = 1 to $m$.
2. Calculate the destination processor $Q_{j_u}$ for the first array element in the $u$th section by the following Equation,

$$Rank(Q_{j_u}) = k \times ((u-1) \times P + i) \bmod Q, \qquad (1)$$

   where $u$ = 1 to $m$.
3. The indices of the first array elements ($FI$) of classes that will be sent to destination processor $Q_{j_u}$, $Q_{(j_u+1) \bmod Q}$, $Q_{(j_u+2) \bmod Q}$, …, $Q_{(j_u+Q-1) \bmod Q}$ in the $u$th section are equal to $\beta+0$, $\beta+r$, $\beta+2r$, …, $\beta+(Q-1)\times r$, respectively, where $\beta = (u-1)\times kr$.
4. The class size ($CS$) of classes that will be sent to destination processors $Q_{j_u}$, $Q_{(j_u+1) \bmod Q}$, $Q_{(j_u+2) \bmod Q}$, …, $Q_{(j_u+kmodQ-1) \bmod Q}$ is equal to $base+r$. The class size ($CS$) of classes that will be sent to other destination processors in the $u$th section are equal to $base$, where $base = \lfloor \frac{k}{Q} \rfloor \times r$, and $kmodQ = mod(k, Q)$



| $PIT_0$ | Messages | $(CS_1, FI_1)$ | $(CS_2, FI_2)$ |
|---|---|---|---|
| | $Msg_0$ | (4, 0) | (2, 14) |
| | $Msg_1$ | (2, 2) | (2, 16) |
| | $msg_2$ | (2, 4) | (4, 10) |
| | $msg_3$ | (2, 6) | (2, 12) |

Fig. 3: The packing information table of source processor $P_0$ for the redistribution shown in Fig. 1.

## 3.2 Receive Phase

We use the same example shown in Fig. 1 to describe our method in the receive phase. In Fig. 1, for source processor $P_0$, array elements $SLA_0$[0, 1, 8, 9] form a class in the first section of $SLA_0[0{:}LCC_s-1]$. Array elements $SLA_0$[14, 15] form a class in the second section of $SLA_0[0{:}LCC_s-1]$. We have the following observation.

- <u>Observation 3.4</u>: For each destination processor, each local complete cycle have the same communication patterns. For example, for destination processor $Q_0$, the source processor of array elements $DLA_0[0]$, $DLA_0[1]$, $DLA_0[2]$, $DLA_0[3]$, $DLA_0[6]$, and $DLA_0[7]$ in the first $LCC_d$ of $DLA_0$ is $P_0$. In this example, $LCC_d$ is equal to 10. From Fig. 1, we can see that the source processor of array elements $DLA_0[0+10]$, $DLA_0[1+10]$, $DLA_0[2+10]$, $DLA_0[3+10]$, $DLA_0[6+10]$, and $DLA_0[7+10]$ in the second $LCC_d$ of $DLA_0$ is also $P_0$.
- <u>Observation 3.5</u>: For each source processor, array elements in the same class of a source local array will be in the consecutive array positions of a destination local array in the destination distribution. For example, for source processor $P_0$, array elements $SLA_0$[0, 1, 8, 9] = $A$[1, 2, 9, 10] are in the same class. In the destination distribution, $A$[1, 2, 9, 10] are redistributed to $DLA_0$[0, 1, 2, 3]. So is class $SLA_0$[14, 15] = $A$[25, 26] that will be redistributed to $DLA_0$[6, 7].

Given a $kr_{(P)} \rightarrow r_{(Q)}$ redistribution, in the send phase, for a source processor $P_i$, message $msg_j$ that will be sent to destination processor $Q_j$ is packed class by class in an ascending order. According to Observations 3.3 and 3.4, for a destination processor $Q_j$, if we know the class sizes and the positions to place the first array elements of classes, we can unpack elements in messages to $DLA_j$ without calculating the receive processor/data sets. For example, for the redistribution shown in Fig. 1, the message $msg_0$ that will be sent from source processor $P_0$ to destination processor $Q_0$ is given in Fig. 2(a). In Fig. 2(a), $msg_0[0{:}3] = SLA_0$[0, 1, 8, 9] and $msg_0[4{:}5] = SLA_0$[14, 15] are classes in the first and the second sections of $SLA_0[0{:}LCC_s-1]$, respectively. The class sizes of $SLA_0$[0, 1, 8, 9] and $SLA_0$[14, 15] are 4 and 2 respectively. To unpack $msg_0$, the positions of the first array elements of $msg_0[0{:}3]$ and $msg_0[4{:}5]$ are 0 and 6 in $DLA_0$, respectively. According to Observation 3.4, $Q_0$ unpacks the $msg_0[0{:}3]$ to $DLA_0[0{:}3]$ and $msg_0[4{:}5]$ to $DLA_0[6{:}7]$. From the Observation 3.1, we know that each local complete cycle has the same communication patterns. Since $LCC_d$ = 10, $msg_0[6{:}9]$ and $msg_0[10{:}11]$ will be unpacked to $DLA_0[10{:}13]$ and $DLA_0[16{:}17]$, respectively, as shown in Fig. 4(a). Fig. 4(b) shows the unpacking process of destination processor $Q_0$.

According to above descriptions, we can gather the information of class sizes ($CS$) and the positions ($FI$) of destination local arrays to place the first array elements of classes into an *unpacking information table* ($UPIT$). Fig. 5 shows the unpacking information table of destination processor $Q_0$ for the redistribution shown in Fig. 1.

Based on the unpacking information table, we can unpack elements in messages to destination local arrays without calculating the receive processor/data sets.

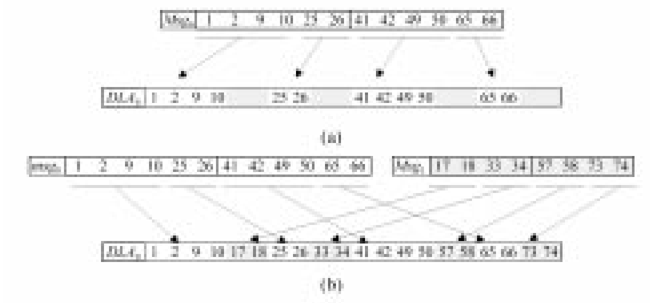
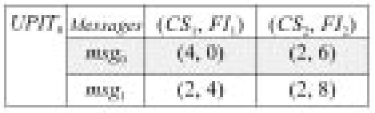


Fig. 4: (a) Destination processor $Q_0$ unpacks messages $msg_0$ (b) Destination processor $Q_0$ unpacks messages $msg_1$



| $UPIT_0$ | Messages | $(CS_1, FI_1)$ | $(CS_2, FI_2)$ |
|---|---|---|---|
| | $msg_0$ | (4, 0) | (2, 6) |
| | $msg_1$ | (2, 4) | (2, 8) |

Fig. 5: The unpacking information table of destination processor $Q_0$ for the redistribution shown in Fig. 1.

Given a $kr_{(P)}\rightarrow r_{(Q)}$ redistribution, a destination processor $Q_j$ can construct the unpacking information table by the following steps:

1. The values of $CS$ in the unpacking information table shown in Fig. 6 can be determined by the following Equation,

$$\alpha_{a,b} = (\lfloor k/Q \rfloor + \Gamma[mod((j+Q-mod((b+(a-1)\times P)\times k,Q)),Q) < mod(k,Q)]) \times r \qquad (2)$$

   Where $m=\frac{LCC_s}{kr}$, $1 \le a \le m$, $0 \le b \le P-1$, and $\Gamma[e]$ is called Iverson's function. If the value of $e$ is true, then $\Gamma[e] = 1$; otherwise $\Gamma[e] = 0$.

2. The values of $FI$ in the unpacking information table shown in Fig. 6 can be determined as follows,

$$\boxed{\text{Section 1}}\quad \begin{array}{llcl} & \beta_{1,0} & = & \beta_{1,P-1}+\alpha_{1,P-1} \\ & \beta_{1,1} & = & \beta_{1,0}+\alpha_{1,0} \\ & & \vdots & \\ & \beta_{1,i_1-1} & = & \beta_{1,i_1-2}+\alpha_{1,i_1-2} \\ \rightarrow & \beta_{1,i_1} & = & 0 \\ & \beta_{1,i_1+1} & = & \beta_{1,i_1}+\alpha_{1,i_1} \\ & & \vdots & \\ & \beta_{1,P-2} & = & \beta_{1,P-1}+\alpha_{1,P-1} \end{array}$$

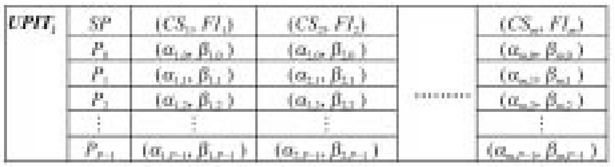
$$\boxed{\text{Section 2}}\quad \begin{array}{llcl} & \beta_{2,0} & = & \beta_{2,P-1}+\alpha_{2,P-1} \\ & \beta_{2,1} & = & \beta_{2,0}+\alpha_{2,0} \\ & & \vdots & \\ & \beta_{2,i_2-1} & = & \beta_{2,i_2-2}+\alpha_{2,i_2-2} \\ \rightarrow & \beta_{2,i_2} & = & \beta_{1,i_1-1}+\alpha_{1,i_1-1} \\ & \beta_{2,i_2+1} & = & \beta_{2,i_2}+\alpha_{2,i_2} \\ & & \vdots & \\ & \beta_{2,P-2} & = & \beta_{2,P-1}+\alpha_{2,P-1} \end{array}$$

$$\vdots$$

$$\boxed{\text{Section } m}\quad \begin{array}{llcl} & \beta_{m,0} & = & \beta_{m,P-1}+\alpha_{m,P-1} \\ & \beta_{m,1} & = & \beta_{m,0}+\alpha_{m,0} \\ & & \vdots & \\ & \beta_{m,i_m-1} & = & \beta_{m,i_m-2}+\alpha_{m,i_m-2} \\ \rightarrow & \beta_{m,i_m} & = & \beta_{m-1,i_{m-1}-1}+\alpha_{m-1,i_{m-1}-1} \\ & \beta_{m,i_m+1} & = & \beta_{m,i_m}+\alpha_{m,i_m} \\ & & \vdots & \\ & \beta_{m,P-2} & = & \beta_{m,P-1}+\alpha_{m,P-1} \end{array}$$

Where $i_1$, $i_2$, …, $i_m$ represent the ranks of the source processors for array elements $DLA_j[0]$, $DLA_j[\beta_{1,i_1-1}+\alpha_{1,i_1-1}]$, …, $DLA_j[\beta_{m-1,i_{m-1}-1}+\alpha_{m-1,i_{m-1}-1}]$. They can be determined by the following Equation,

$$Rank(\ sp(DLA_j[x])\ ) = mod\left(\left\lfloor \frac{x\times Q+j}{k} \right\rfloor, P\right) \qquad (3)$$



| $UPIT_j$ | SP | $(CS_1, FI_1)$ | $(CS_2, FI_2)$ | | $(CS_m, FI_m)$ |
|---|---|---|---|---|---|
| | $P_0$ | $(\alpha_{1,0}, \beta_{1,0})$ | $(\alpha_{2,0}, \beta_{2,0})$ | ……… | $(\alpha_{m,0}, \beta_{m,0})$ |
| | $P_1$ | $(\alpha_{1,1}, \beta_{1,1})$ | $(\alpha_{2,1}, \beta_{2,1})$ | | $(\alpha_{m,1}, \beta_{m,1})$ |
| | $P_2$ | $(\alpha_{1,2}, \beta_{1,2})$ | $(\alpha_{2,2}, \beta_{2,2})$ | | $(\alpha_{m,2}, \beta_{m,2})$ |
| | ⋮ | ⋮ | ⋮ | | ⋮ |
| | $P_{P-1}$ | $(\alpha_{1,P-1}, \beta_{1,P-1})$ | $(\alpha_{2,P-1}, \beta_{2,P-1})$ | | $(\alpha_{m,P-1}, \beta_{m,P-1})$ |

Fig. 6: An unpacking information table for $kr_{(P)}\rightarrow r_{(Q)}$ redistribution with $LCC_s=mkr$.

## 4 $r_{(P)}\rightarrow kr_{(Q)}$ array redistribution

### 4.1Send Phase

In an $r_{(P)}\rightarrow kr_{(Q)}$ redistribution, for each source processor, the method to pack array elements is similar to that of a destination processor to unpack array elements in a $kr_{(P)}\rightarrow r_{(Q)}$ redistribution. Therefore, we only describe how to derive a packing information table for $r_{(P)}\rightarrow kr_{(Q)}$ redistribution. The form of packing information table is the same as that shown in Fig. 6. Given an $r_{(P)}\rightarrow kr_{(Q)}$ redistribution, a source processor $P_i$ can construct the packing information table by the following steps:

1. The values of $CS$ in the packing information table can be determined by the following Equation,

$$\alpha_{a,b} = (\lfloor k/P \rfloor + \Gamma[mod((i+P-mod(\ b\times k,P)),P) < mod(k,P)]) \times r \qquad (4)$$

   Where $m=\frac{LCC_s}{kr}$, $1 \le a \le m$, $0 \le b \le Q-1$.

2. The values of $FI$ in the packing information table were

determined by the same way as that described for Fig. 6. The ranks of the destination processors for array elements $SLA_i[0]$, $SLA_i[\beta_{1,j_1-1}+\alpha_{1,j_1-1}]$, …, $SLA_i[\beta_{m-1,j_{m-1}-1}+\alpha_{m-1,j_{m-1}-1}]$ can be determined by the following Equation,

$$Rank(dp(SLA_i[x])) = mod\left(\left\lfloor \frac{x\times P+i}{k}\right\rfloor, Q\right) \quad (5)$$

## 4.2 Receive Phase

In an $r_{(P)}\to kr_{(Q)}$ redistribution, for each destination processor, the method to unpack array elements is similar to that of a source processor to pack array elements in a $kr_{(P)}\to r_{(Q)}$ redistribution. Therefore, we only describe how to derive the unpacking information table for $r_{(P)}\to kr_{(Q)}$ redistribution.

Given an $r_{(P)}\to kr_{(Q)}$ redistribution, since a local complete cycle of destination local array can be divided into $m$ sections, where $m=\frac{LCC_d}{kr}$, processor $Q_j$ can construct the unpacking information table by the following steps:

1. For each section $u$, do steps 2 to 4, where $u$ = 1 to $m$.
2. Calculate the source processor $P_{i_u}$ for the first array element in the $u$th section by the following Equation,

$$Rank(P_{i_u}) = k \times((u\text{-}1)\times Q+ j) \bmod P \quad (6)$$

   where $u$ = 1 to $m$.
3. The index of the first array element ($FI$) of the class which received from source processor $P_{i_u}$, $P_{(i_u+1)\bmod P}$, $P_{(i_u+2)\bmod P}$, …, $P_{(i_u+P-1)\bmod P}$ are equal to $\beta$+0, $\beta+r$, $\beta+2r$, …, $\beta+(Q-1)\times r$, respectively, where $\beta = (u-1)\times kr$.
4. The class size ($CS$) for source processor $P_{i_u}$, $P_{(i_u+1)\bmod P}$, $P_{(i_u+2)\bmod P}$, …, $P_{(i_u+kmodP-1)\bmod P}$ are equal to $base+r$. The class size ($CS$) for other source processors in the $u$th section are equal to $base$, where $base = \lfloor \frac{k}{P}\rfloor\times r$, and $kmodP = mod(k, P)$.

## 5. Experimental Results

To evaluate the performance of the proposed methods, we have implemented our methods along with the *PITFALLS* method and the *Prylli's* method 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 $kr_{(P)}\to r_{(Q)}$ and $r_{(P)}\to kr_{(Q)}$ array redistribution. Time was measured by using *MPI_Wtime*(). The experimental results were shown in Fig. 7 and Fig. 8. In Fig. 7 and Fig. 8, *Krr* represents the algorithms proposed in this paper. *Pitfalls* and *Scala* represent the *PITFALLS* method and the *Prylli's* method, respectively.

Fig. 7 gives the performance of these algorithms to perform $kr_{(P)}\to r_{(Q)}$ and $r_{(P)}\to kr_{(Q)}$ redistribution with various array size, where $k$ = 5, $P$ = 50 and $Q$ = 40. In Fig. 7(a), the execution time of these three algorithms has the order $T(Krr) < T(Scala) < T(Pitfall)$. From Table 1, for the $kr\to r$ redistribution, we can see that the indexing time of the *Krr* method is smaller than that of the *Prylli's* method and the *PITFALLS* method. This is because that the *PITFALLS* method and the *Prylli's* method need to spend time on communication sets calculation while the *Krr* method does not. Moreover, the time for the *Krr* method to generate the packing/unpacking information tables is quite small. Therefore, the indexing time of the *Krr* method is less than that of the *PITFALLS* method and the *Prylli's* method.

For the packing/unpacking part, the packing and unpacking information tables of the *Krr* method are optimized, that is, every consecutive local array elements that have the same source (destination) processor in a local complete cycle of a local array will have only one (*CS*, *FI*) entry in the packing (unpacking) information table. This optimization can reduce the memory copy time when performing the packing and unpacking processes. Therefore, we can see that the packing/unpacking time of the *Krr* method is less than that of the *PITFALLS* method and the *Prylli's* method.

For the communication part, all of these three methods use asynchronous communication schemes. The computation and the communication overheads can be overlapped. However, the *Krr* method unpacks any received messages in the receive phase while the *PITFALLS* method and the *Prylli's* method unpack messages in a specific order. Therefore, the communication time of the *Krr* method is less than or equal to that of the *PITFALLS* and the *Prylli's* methods.

Fig. 7(b) presents the execution time of these algorithms for the $r\to kr$ redistribution. The execution time of these three algorithms has the order $T(Krr) < T(Scala) < T(Pitfall)$. In Table 1, for the $r\to kr$ redistribution, we can see that the computation and the communication time of the *Krr* method is less than that of the *PITFALLS* method and the *Prylli's* method. The reasons are the same as those described for Fig. 7(a).

For the cases when $k$ is equal to 25, 50, and 100, we have similar observations as those described for Fig. 7.

Fig. 8 gives the execution time of these algorithms to perform BLOCK→CYCLIC and CYCLIC→BLOCK redistribution with various array sizes. In this case, the value of $k$ is equal to *Array_size*/*P* (or *Array_size*/*Q*). From Fig.s 8(a) and 8(b), we can see that the execution time of these three algorithms has the order $T(Krr) << T(Scala) < T(Pitfall)$ for both BLOCK→CYCLIC and CYCLIC→BLOCK redistribution. In Table 2, for both BLOCK→CYCLIC and CYCLIC→BLOCK redistribution, the indexing time of theses three algorithms has the order $T_{index}(Krr) << T_{index}(Scala) < T_{index}(PITFALLS)$. The *PITFALLS* and the *Prylli's* methods have very large indexing time compared to that of the *Krr* method. The reason is that the indexing time of these two methods depends on the number of intersections between source and destination distributions. In this case, there are *Array_size*/*P* and *Array_size*/*Q* intersections between each source and destination processor in the BLOCK→CYCLIC and CYCLIC→BLOCK redistribution, respectively. Therefore, a processor needs to compute $\lfloor Array\_size/P \rfloor \times P$ (or $\lfloor Array\_size/Q \rfloor \times Q$) intersections that demand a lot of computation time when array size is large.

From the above performance analysis and experimental results, we have the following remarks.

1. The indexing time of the *PITFALLS* method and the *Prylli's* method depends on the value of *k* while the *Krr* method does not. When the value of *k* increases, the indexing time of the *PITFALLS* method and the *Prylli's* method increases as well. However, The indexing time of these three methods is independent to the array size.
2. Since the packing and unpacking information tables of the *Krr* method are optimized, the packing/unpacking time of the *Krr* method is less than that of the *PITFALLS* method and the *Prylli's* method. When the array size increases, the difference of the packing/unpacking time between the *Krr* method and the *PITFALLS* method or the *Prylli's* method becomes large.

All of these three methods use asynchronous communication schemes. However, the *Krr* method unpacks any received messages in the receive phase while the *PITFALLS* method and the *Prylli's* method unpack messages in a specific order. Therefore, the communication time of the *Krr* method is less than or equal to that of the *PITFALLS* and the *Prylli's* methods.

## 6. Conclusions

In this paper, we have presented efficient methods to generate the packing/unpacking information for BOLCK-CYCLIC(*kr*) to BLOCK-CYCLIC(*r*) and BOLCK-CYCLIC(*r*) to BLOCK-CYCLIC(*kr*) redistribution with arbitrary source/destination processor sets. The most significant improvement of this paper is that a processor does not need to construct the send/receive processor/data sets for a redistribution. Based on the packing/unpacking information, 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 *PITFALLS* method and the *Prylli's* method on an IBM SP2 parallel machine. The experimental results show that our algorithms outperform the *PITFALLS* method and the *Prylli's* method and for all test samples.

## References

[1] 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," *JPDC*, Vol. 26, pp. 72-84, 1995.

[2] Y.-C Chung, C.-H Hsu and S.-W Bai, "A Basic-Cycle Calculation Technique for Efficient Dynamic Data Redistribution," *IEEE Trans. on PDS*, vol. 9, no. 4, pp. 359-377, April 1998.

[3] Frederic Desprez, Jack Dongarra, and Antoine Petitet," Scheduling Block-Cyclic Array Redistribution," *IEEE Trans. on PDS*, vol. 9, no. 2, Feb. 1998.

[4] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan, "On Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines," *JPDC*, Vol. 32, pp. 155-172, 1996.

[5] 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.

[6] Ching-Hsien Hsu and Yeh-Ching Chung, "Efficient Methods for kr→r and r→kr array Redistribution,"*The Journal of Supercomputing*, vol. 12, no. 2, May 1998.

[7] Edgar T. Kalns, and Lionel M. Ni, "Processor Mapping Technique Toward Efficient Data Redistribution, " *IEEE Trans. on PDS*, vol. 6, no. 12 , December 1995.

[8] S. D. Kaushik, C. H. Huang, J. Ramanujam, and P. Sadayappan, "Multiphase array redistribution: Modeling and evaluation," In *Proc. of IPPS*, pp. 441-445, 1995.

[9] S. D. Kaushik, C. H. Huang, and P. Sadayappan, "Efficient Index Set Generation for Compiling HPF Array Statements on Distributed-Memory Machines," *JPDC*, Vol. 38, pp. 237-247, 1996.

[10] K. Kennedy, N. Nedeljkovic, and A. Sethi, "Efficient address generation for block-cyclic distribution," In *Proc. of Intl Conf. on Supercomputing*, pp. 180-184, July 1995.

[11] Young Won Lim, Prashanth B. Bhat, and Viktor, K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," *Proc. of the Eighth IEEE SPDP*, pp. 74-83, 1996.

[12] Y. W. Lim, N. Park, and V. K. Prasanna, "Efficient Algorithms for Multi-Dimensional Block-Cyclic Redistribution of Arrays," *Proc. of the 26th ICPP*, pp. 234-241, 1997.

[13] L. Prylli and B. Touranchean, "Fast runtime block cyclic data redistribution on multiprocessors," *JPDC*, Vol. 45, pp. 63-72, Aug. 1997.

[14] S. Ramaswamy, B. Simons, and P. Banerjee, "Optimization for Efficient Array Redistribution on

Distributed Memory Multicomputers," *JPDC,* Vol. 38, pp. 217-228, 1996.

[15] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross," Generating communication for array statements: Design, implementation, and evaluation," *JPDC,* Vol. 21, pp. 150-159, 1994.

[16] Rajeev. Thakur, Alok. Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution, " *IEEE Trans. on PDS*, vol. 7, no. 6 , June, 1996.

[17] David W. Walker, Steve W. Otto, "Redistribution of BLOCK-CYCLIC Data Distributions Using MPI," Concurrency: Practice and Experience, vol. 8, no. 9, pp. 707-728, Nov. 1996.

[18] Akiyoshi Wakatani and Michael Wolfe, "A New Approach to Array Redistribution: Strip Mining Redistribution," In *Proc. of Parallel Architectures and Languages*, July 1994.

[19] Akiyoshi Wakatani and Michael Wolfe, "Optimization of Array Redistribution for Distributed Memory Multicomputers, " short communication, *Parallel Computing*, Vol. 21, Number 9, pp. 1485-1490, September 1995.

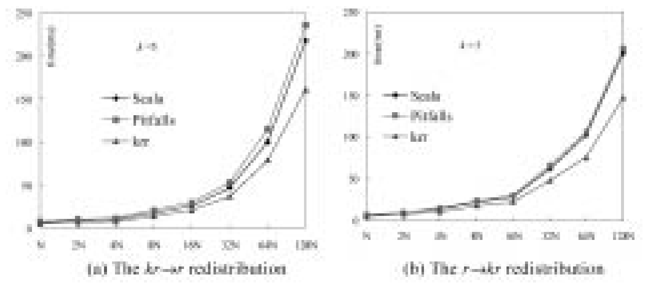


Fig. 7: Performance of different algorithms to execute a BLOCK-CYCLIC(10) to BLOCK-CYCLIC(2) redistribution and vice versa on a 50-node SP2. (*N*=1M single precision).

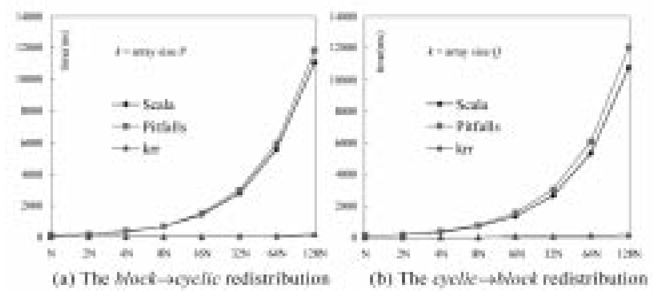
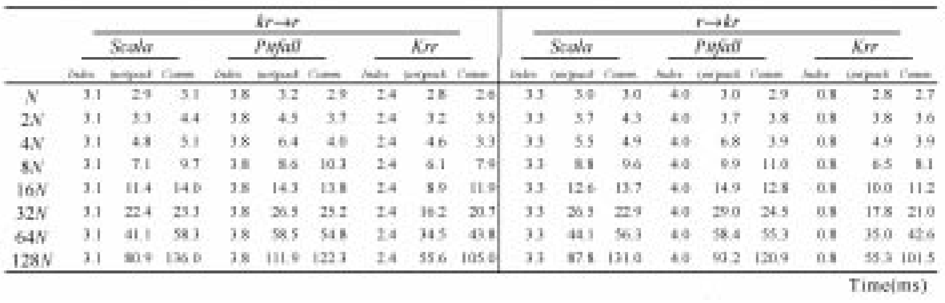


Fig. 8: Performance of different algorithms to execute a BLOCK to CYCLIC redistribution and vice versa on a 50-node SP2. (*N*=1M single precision).

Table 1: The indexing, packing/unpacking, and communication time for Fig. 7.

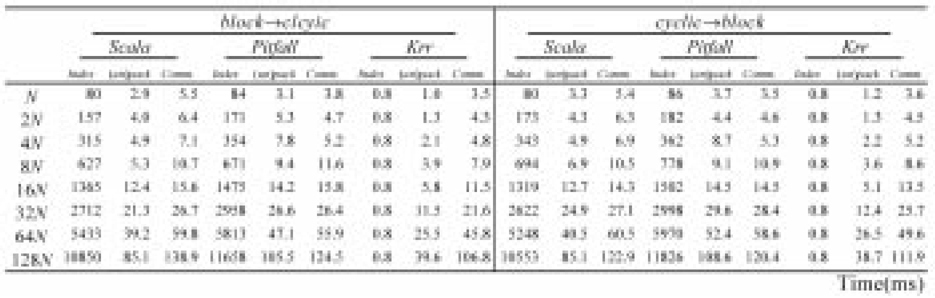


| | *kr→r* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | *Scala* | | | *Pitfall* | | | *Krr* | | |
| | Index | (un)pack | Comm. | Index | (un)pack | Comm. | Index | (un)pack | Comm. |
| *N* | 3.1 | 2.9 | 3.1 | 3.8 | 3.2 | 2.9 | 2.4 | 2.8 | 2.6 |
| 2*N* | 3.1 | 3.3 | 4.4 | 3.8 | 4.5 | 3.7 | 2.4 | 3.2 | 3.5 |
| 4*N* | 3.1 | 4.8 | 5.1 | 3.8 | 6.4 | 4.0 | 2.4 | 4.6 | 5.3 |
| 8*N* | 3.1 | 7.1 | 9.7 | 3.8 | 8.6 | 10.3 | 2.4 | 6.1 | 7.9 |
| 16*N* | 3.1 | 11.4 | 14.0 | 3.8 | 14.3 | 13.8 | 2.4 | 8.9 | 11.9 |
| 32*N* | 3.1 | 22.4 | 23.3 | 3.8 | 26.5 | 25.2 | 2.4 | 16.2 | 20.7 |
| 64*N* | 3.1 | 41.1 | 58.3 | 3.8 | 58.5 | 54.8 | 2.4 | 34.5 | 43.8 |
| 128*N* | 3.1 | 80.9 | 136.0 | 3.8 | 111.9 | 122.3 | 2.4 | 55.6 | 105.0 |

| | *r→kr* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | *Scala* | | | *Pitfall* | | | *Krr* | | |
| | Index | (un)pack | Comm. | Index | (un)pack | Comm. | Index | (un)pack | Comm. |
| *N* | 3.3 | 3.0 | 3.0 | 4.0 | 3.0 | 2.9 | 0.8 | 2.8 | 2.7 |
| 2*N* | 3.3 | 3.7 | 4.3 | 4.0 | 3.7 | 3.8 | 0.8 | 3.8 | 3.6 |
| 4*N* | 3.3 | 5.5 | 4.9 | 4.0 | 6.8 | 3.9 | 0.8 | 4.9 | 3.9 |
| 8*N* | 3.3 | 8.8 | 9.6 | 4.0 | 9.9 | 11.0 | 0.8 | 6.5 | 8.1 |
| 16*N* | 3.3 | 12.6 | 13.7 | 4.0 | 14.9 | 12.8 | 0.8 | 10.0 | 11.2 |
| 32*N* | 3.3 | 26.3 | 22.9 | 4.0 | 29.0 | 24.5 | 0.8 | 17.8 | 21.0 |
| 64*N* | 3.3 | 44.1 | 56.3 | 4.0 | 58.4 | 55.3 | 0.8 | 35.0 | 42.6 |
| 128*N* | 3.3 | 87.8 | 131.0 | 4.0 | 93.2 | 120.9 | 0.8 | 55.3 | 101.5 |

Time(ms)

Table 2: The indexing, packing/unpacking, and communication time for Fig. 8.



| | *block→clcyic* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | *Scala* | | | *Pitfall* | | | *Krr* | | |
| | Index | (un)pack | Comm. | Index | (un)pack | Comm. | Index | (un)pack | Comm. |
| *N* | 80 | 2.9 | 5.5 | 84 | 3.1 | 3.8 | 0.8 | 1.0 | 3.5 |
| 2*N* | 157 | 4.0 | 6.4 | 171 | 5.3 | 4.7 | 0.8 | 1.3 | 4.3 |
| 4*N* | 315 | 4.9 | 7.1 | 354 | 7.8 | 5.2 | 0.8 | 2.1 | 4.8 |
| 8*N* | 627 | 5.3 | 10.7 | 671 | 9.4 | 11.6 | 0.8 | 3.9 | 7.9 |
| 16*N* | 1365 | 12.4 | 15.6 | 1475 | 14.2 | 15.8 | 0.8 | 5.8 | 11.5 |
| 32*N* | 2712 | 21.3 | 26.7 | 2958 | 26.6 | 26.4 | 0.8 | 11.5 | 21.6 |
| 64*N* | 5433 | 39.2 | 59.8 | 5813 | 47.1 | 55.9 | 0.8 | 25.3 | 45.8 |
| 128*N* | 10850 | 85.1 | 138.9 | 11658 | 105.5 | 124.5 | 0.8 | 39.6 | 106.8 |

| | *cyclic→block* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | *Scala* | | | *Pitfall* | | | *Krr* | | |
| | Index | (un)pack | Comm. | Index | (un)pack | Comm. | Index | (un)pack | Comm. |
| *N* | 80 | 3.3 | 5.4 | 86 | 3.7 | 3.5 | 0.8 | 1.2 | 3.6 |
| 2*N* | 175 | 4.3 | 6.5 | 182 | 4.4 | 4.6 | 0.8 | 1.3 | 4.3 |
| 4*N* | 343 | 4.9 | 6.9 | 362 | 8.7 | 5.3 | 0.8 | 2.2 | 5.2 |
| 8*N* | 694 | 6.9 | 10.5 | 778 | 9.1 | 10.9 | 0.8 | 3.6 | 8.6 |
| 16*N* | 1319 | 12.7 | 14.3 | 1582 | 14.5 | 14.5 | 0.8 | 5.1 | 13.5 |
| 32*N* | 2622 | 24.9 | 27.1 | 2998 | 29.6 | 28.4 | 0.8 | 12.4 | 23.7 |
| 64*N* | 5248 | 40.3 | 60.5 | 5970 | 52.4 | 58.6 | 0.8 | 26.5 | 49.6 |
| 128*N* | 10553 | 85.1 | 122.9 | 11826 | 108.6 | 120.4 | 0.8 | 38.7 | 111.9 |

Time(ms)