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

Yeh-Ching Chung[2] and Ching-Hsien Hsu

Department of Information Engineering
Feng Chia University, Taichung, Taiwan 407, ROC
Tel : 886-4-4517250 x2706
Fax : 886-4-4515517
Email : ychung, chhsu@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 algorithms for array redistribution. The most significant improve-ment of our algorithms is that a processor does not need to construct the send/receive data sets for a redistribution. Based on the packing/unpack-ing information that derived from the BLOCK-CYCLIC(kr) to BLOCK-CYCLIC(r) redistribution (or 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 Thakur's methods on an IBM SP2 parallel machine. The results show that the execution time of our algorithms is approximately 5% to 27% faster than that of Thakur's methods.

Key words: array redistribution, distributed memory multicomputers, data distribution.

1. Introduction

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 will be sent to destination processors, pack those data sets, and send those packed data sets to their destination processors. In the receive phase, a processor P_i has to determine all the data sets that will be received from source processors, receive those data sets, and unpack data elements in those data sets 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 ($\bigcup_{P_j \in DPS[P_i]}$ SDS[P_i, P_j]) : 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.
- Receive Data Sets ($\bigcup_{P_i \in SPS[P_j]}$ RDS[P_j, P_i]) : 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.

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 data among processors. Thus efficient methods for performing array redistribution are of great importance for the development of distributed memory compilers. In this paper, we present efficient methods to perform BOLCK-CYCLIC(kr) to BOLCK-CYCLIC(r) and BOLCK-CYCLIC(r) to BOLCK-CYCLIC(kr) redistribution. In our algorithms, based on the packing and unpacking information that derived from BOLCK-CYCLIC(kr) to BOLCK-CYCLIC(r) (and vice versa) redistribution, a processor can pack and unpack array elements without calculating the send/receive data sets. Therefore, the computation overheads can be reduced greatly.

The paper is organized as follows. In Section 2, a brief survey of related work will be presented. Section 3 presents the algorithms for array redistribution. The performance evaluation and comparisons of redistribution algorithms that proposed in this paper and in [11, 12] will be given in Section 4.

[1] The work of this paper was partially supported by NSC of R.O.C. under contract NSC-86-2213-E035-023.
[2] The correspondence addressee.

2. Related Work

Many methods for performing array redistribution have been presented in the literature. Gupta *et al.* [2] derived closed form expressions to and virtual processor approach for addressing the problem of reference index-set identification for array statements with BLOCK-CYCLIC(c) distribution. A similar approach was presented in [10]. In [1], 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. Kennedy *et al.* [6] also presented algorithms to compute the local memory access sequence for array statements with BLOCK-CYCLIC(c) distribution.

Thakur *et al.* [11,12] presented algorithms for run-time array redistribution in HPF programs. In [8,9], Ramaswamy *et al.* used a mathematical representation, PITFALLS, for regular data redistribution. The basic idea of PITFALLS is to find all intersections between source and target distributions. In [3], 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.

Kaushik *et al.* [5] proposed a multi-phase redistribution approach for BLOCK-CYCLIC(s) to BLOCK-CYCLIC(t) redistribution. In [14], portion of array elements were redistributed in sequence in order to overlap the communication and computation. In [15], a spiral mapping technique was proposed to reduce communication conflicts when performing a redistribution. Kalns *et al.* [4] proposed a processor mapping technique to minimizes the amount of data exchange for BLOCK to BLOCK-CYCLIC(c) redistribution and vice versa. In [7], a generalized circulant matrix formalism was proposed to reduce the communication overheads for BLOCK-CYCLIC(r) to BLOCK-CYCLIC(kr) redistribution. Walker *et al.* [13] used the standardized message passing interface, MPI, to express the redistribution operations.

3. Efficient Methods for $kr \rightarrow r$ and $r \rightarrow kr$ Redistribution

In general, the 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 $0 \le i \le M-1$. 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 $0 \le j \le M-1$.

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 $0 \le j \le M-1$. 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 $0 \le i \le M-1$.

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 $1 \le k, l \le N$ and $0 \le m, n \le N/M-1$.

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 with the same destination (or source) processor. The *class size* is defined as the number of array elements in a class.

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

Due to the page limitation, we omit the proof of lemmas presented in this paper.

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 $SLA_i[m+N/M\times LCC]$ have the same destination processor, where $0 \le i \le M-1$ and $0 \le m \le LCC-1$.

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 $0 \le x \le N/GCC-1$ and $0 \le i, j \le M-1$.

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 $0 \le i, j \le M-1$. 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 set and the destination processor set.

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:

$$\eta = mod(rank(P_i)\times k, M) \qquad (1)$$

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 $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, ..., $SG(SLA_i[a_{\gamma-1}])$ 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 $x\times LCC \le a_0 < a_1 < a_2 < \ldots < a_{\gamma-1} < (x+1)\times LCC$.

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 $0 \le \alpha \le N/M-1$.

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 $0 \le \alpha \le N/M-1$. 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 send data set and the source processor set.

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:

$$\gamma = (\lfloor k/M \rfloor + \llbracket mod((rank(P_j)+M - mod(rank(P_i)\times k, M)), M) < mod(k, M) \rrbracket) \times r \qquad (2)$$

$$\alpha = (\lfloor rank(P_i)\times k/M \rfloor + \llbracket rank(P_j) < mod(rank(P_i)\times k, M) \rrbracket) \times r \qquad (3)$$

Where $rank(P_i)$ and $rank(P_j)$ are the ranks of processors P_i and P_j. The notation "$\llbracket\ \rrbracket$" in equations (2) and (3) is called *Iverson's function*. It is defined as follows :

$\llbracket f(x) \rrbracket = 1$ when $f(x)$ is true

$\llbracket f(x) \rrbracket = 0$ when $f(x)$ is false

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

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

/* Send phase */

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

/* Packing data sets */

4. $index = 1$; $length_\delta = 1$, where $\delta = 0, \ldots, M-1$;

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

3.2 Method for $r \rightarrow kr$ 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 $SLA_j[0{:}N/M-1]$, where $0 \le i, j \le M-1$, $0 \le x \le N/GCC-1$, and $x \times LCC \le a_0 < a_1 < a_2 < \ldots < a_{n-1} < (x+1) \times LCC$.

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 $SLA_i[0{:}LCC\text{-}1]$ (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_{j1}, P_{j2}, P_{j3},..., and P_{ju}, 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 sizes 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_{j1}, array elements $SLA_i[|C_1| : |C_1|+|C_2|-1]$ will be sent to destination processor P_{j2},..., 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_{ju}. From Lemma 1, we know that $SLA_i[0{:}|C_1|-1]$, $SLA_i[LCC{:}LCC+|C_1|-1]$, $SLA_i[2LCC{:}2LCC+|C_1|-1]$,..., 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 $(|C_1|, P_{j1})$, $(|C_2|, P_{j2})$,..., and $(|C_u|, P_{ju})$, we can pack array elements in SLA_i to messages directly without computing the send data set and the destination processor set.

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 $SLA_i[0{:}LCC\text{-}1]$ whose destination processor is P_j can be computed by equation (4). Equations (4) and (5) are given as follows:

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

$$\varphi = \lfloor rank(P_i)/k \rfloor \quad (5)$$

Where $rank(P_i)$ and $rank(P_j)$ are the ranks of processors P_i and P_j. The notation "$[\![\ ]\!]$" in equation (4) is called *Iverson's function.*

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, and $SG(SLA_i[a_0]) = DG(DLA_j[v])$, 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 $0 \le v \le N/M-1$.

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 $0 \le \beta \le N/M-1$. 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 set and the source processor set. 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:

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

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

Where $rank(P_i)$ and $rank(P_j)$ are the ranks of processors P_i and P_j. The notation "[]" in equation (6) is called *Iverson's function.*

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_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** ($index <= max\_index$)
9. { $t = 1$;
10. **while**(($t <= m$)&&($index <= max\_index$))
11. { $j = j_t$; $l = 1$;
12. **while** (($l <= |C_j|$) && ($index <= max\_index$))
13. { $out\_buffer_j[length_j{+}{+}] = SLA_i[index{+}{+}]$;
14. $l{+}{+}$; }
15. $t{+}{+}$; }}
16. Send out_buffer_j to processor P_j, where $j = j_1, j_2, \ldots j_m$.

/* Receive phase */

17. $max\_cycle = max\_index$ divided by kr
18. **Repeat** $m = min(M, k)$ times
19. Receive message $buffer\_in_i$ from source processors P_i.
20. Calculate the value of δ for $buffer\_in_i$ using Equation (6);
21. Calculate the value of β for $buffer\_in_i$ using Equation (7);

/* Unpacking data sets */

22. $index = \beta$; $length = 1$; $j = 0$; $count = 0$;
23. **while** ($j <= max\_cycle$)
24. { $count = 1$; $index = \beta + j \times kr$;
25. **while** ($count <= \delta$) { $l = 1$;
26. **while** ($l <= r$) { $DLA_i[index{+}{+}] = buffer\_in[length{+}{+}]$;
27. $count{+}{+}$; $l{+}{+}$; }
28. $index += (M-1) \times r$; }
29. $j{+}{+}$; }

*end_of_*$r \rightarrow kr\_redistribution$

---

4. Performance Evaluation and Experimental Results

To evaluate the performance of the proposed algorithms, we have implemented the proposed algorithms on an 64-nodes IBM SP2 parallel machine along with those proposed in [11, 12]. All of the algorithms were written in C + MPI.

The experimental results were shown in Table 1 and Table 2. In Table 1 and Table 2, the *ours* represents the algorithms proposed in this paper while *thakur* represents the algorithms proposed in [11, 12]. Table 1 gives the execution time and the percentages of the performance improvement of *ours* over *thakur* for $kr \rightarrow r$ (and vice-versa) redistribution with various array size and distribution factors. In Table 1, the execution time of *ours* in BLOCK-CYCLIC(10) to BLOCK-CYCLIC(2) redistribution is about 11% to 23% faster than that of *thakur*. For BLOCK-CYCLIC(2) to BLOCK-CYCLIC(10) redistribution, the execution time of *ours* is about 5% to 15% faster than that of *thakur*. For the cases of k = 25, 50, and 100, we have similar observations as those of Table 1 (Due to the page limitation, we did not show the results here). Table 2 gives the execution time and the percentages of the performance improvement of *ours* over *thakur* for BLOCK to CYCLIC (and vice-versa) redistribution. From Table 2, we can see that the execution time of *ours* is about 18% to 27% faster than that of *thakur*.

6. Conclusions

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 Thakur's methods on an IBM SP2 parallel machine. The results show that the execution time of our algorithms is approximately 5% to 27% faster than that of Thakur's methods.

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] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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*, Mclean, VA. Pp. 342-349, Feb. 1995.

[9] S. Ramaswamy, B. Simons, and P. Banerjee, "Optimization for Efficient Array Redistribution on Distributed Memory Multicomputers," *JPDC,* Vol. 38, pp. 217-228, 1996.

[10] 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.

[11] 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.

[12] Rajeev. Thakur, Alok. Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution, " *IEEE TPDS*, vol. 7, no. 6 , JUNE 1996.

[13] David W. Walker, Steve W. Otto, "Redistribution of BLOCK-CYCLIC Data Distributions Using MPI," Technical Report ORNL/TM-12999, Computer Science and Mathematics Division, Oak Ridge National Laboratory, 1995.

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

[15] A. Wakatani and M. Wolfe, "Optimization of Array Redistribution for Distributed Memory Multicomputers, " In *Parallel Computing(submitted)*, 1994.

Table 1 : The percentages of the performance improvement of *ours* over *thakur's* for BLOCK-CYCLIC(10) to BLOCK-CYCLIC(2) redistribution and vice-versa.

BLOCK-CYCLIC(10) to BLOCK-CYCLIC(2)				BLOCK-CYCLIC(2) to BLOCK-CYCLIC(10)			
SIZE	ours	thakur's	Improvement	SIZE	ours	thakur's	Improvement
72000	12.621	14.423	12.5%	72000	15.378	17.45	11.9%
144000	14.662	17.536	16.4%	144000	22.358	24.465	8.6%
216000	17.541	19.719	11%	216000	27.905	29.504	5.4%
288000	18.364	23.71	22.5%	288000	29.602	32.457	8.8%
360000	26.058	33.858	23%	360000	32.306	38.24	15.5%

Time unit : ms

Table 2 : The percentages of the performance improvement of *ours* over *thakur's* for BLOCK to CYCLIC redistribution and vice-versa.

BLOCK to CYCLIC				CYCLIC to BLOCK			
SIZE	ours	thakur's	Improvement	SIZE	ours	thakur's	Improvement
72000	10.834	13.221	18.1%	72000	8.469	10.425	18.8%
144000	11.025	14.143	22%	144000	11.533	14.826	22.2%
216000	15.121	19.223	21%	216000	17.258	21.122	18.3%
288000	16.275	21.123	23%	288000	20.072	25.912	22.5%
360000	19.897	27.287	27.1%	360000	27.977	35.913	22.1%

Time unit : ms