Efficient Methods for $kr\rightarrow r$ and $r\rightarrow kr$ Array Redistribution

Yeh-Ching Chung 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 multiprocessors. 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 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 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 multiprocessors, 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 ($\bigcup_{P \in DPS(P_i)} SDS(P_i, P)$) : 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 ($\bigcup_{P \in SPS(P_j)} RDS(P_j, P_i)$) : 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 BLOCK-CYCLIC(kr) to BLOCK-CYCLIC(r) and BLOCK-CYCLIC(r) to BLOCK-CYCLIC(kr) redistribution. In our algorithms, based on the packing and unpacking information that derived from BLOCK-CYCLIC(kr) to BLOCK-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.

Kauhshik 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. Kals et al. [4] proposed a processor mapping technique to minimize 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→r and r→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=mr) redistribution,
- t is divisible by s, i.e., BLOCK-CYCLIC(s=mr) 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→r, r→kr, and s→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→r and r→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→t redistribution on A[i:N] over M processors, the source local array of processor P, denoted by SLA[i:0:N/M−1], is defined as the set of array elements that are distributed to processor P in the source distribution, where 0 ≤ i ≤ M−1. The destination local array of processor P, denoted by DL[A][0:N/M−1], is defined as the set of array elements that are distributed to processor P in the destination distribution, where 0 ≤ j ≤ M−1.

**Definition 3:** Given an s→t redistribution on A[i:N] over M processors, the source processor of an array element in A[i:N] or DL[A][0:N/M−1] is defined as the processor that owns the array element in the source distribution, where 0 ≤ j ≤ M−1. The destination processor of an array element in A[i:N] or SLA[0:N/M−1] is defined as the processor that owns the array element in the destination distribution, where 0 ≤ i ≤ M−1.

**Definition 4:** Given an s→t redistribution on A[i:N] over M processors, we define SG : SLA[i:m] → A[k] is a function that converts a source local array element SLA[i:m] of Pk to its corresponding global array element A[k] and DG : DL[A][n] → A[l] is a function that converts a destination local array element DL[A][n] of Pj to its corresponding global array element A[l], where 1 ≤ k, l ≤ N and 0 ≤ m, n ≤ N/M−1.

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

**Definition 6:** Given an s→t redistribution, a local complete cycle (LCC) of a local array SLA[0:N/M−1] or DL[A][0:N/M−1]) is defined as the least common multiple of s and t, i.e., LCC = lcm(s,t). We define SLA[0:LCC−1] (DL[A][0:LCC−1]) as the first local complete cycle of SLA[0:N/M−1] (DL[A][0:N/M−1]), SLA[0:LCC−1] (DL[A][0:LCC−1]) as the second local complete cycle of SLA[0:N/M−1] (DL[A][0:N/M−1]), and so on.

**Definition 7:** Given an s→t redistribution, for a source processor Pi (or destination processor Pj), a class is defined as the set of array elements in an LCC of SLA, 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 $s \rightarrow r$ 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 $0 \leq i \leq M-1$ and $0 \leq m \leq LCC-1$.

**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 $SLA_i[x\times LCC:x\times LCC+r-1]$, $SLA_i[(x+N/GCC)\times LCC:(x+N/GCC+1)\times LCC]$ and $SLA_i[(x+N/GCC)\times LCC:(x+N/GCC+1)\times LCC]$ are $P_j$, $P_{mod(i+1,M)}$, ..., and $P_{mod(i,k,M)}$, 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 $P_{mod(i+1,M)}$, ..., and $P_{mod(i,k,M)}$, respectively, where $0 \leq i, j \leq M-1$. From Lemma 1, we know that $SLA_i[0:r-1]$, $SLA_i[LCC-r:LCC-1]$, $SLA_i[2\times LCC:2\times LCC+r-1]$, ..., 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:

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

where $\eta$ 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 $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, ..., $SG(SLA_i[a_{\eta-1}])$ is $P_j$, then $SG(SLA_i[a_0])$, $SG(SLA_i[a_1])$, ..., $SG(SLA_i[a_{\eta-1}])$ are in the consecutive local array positions of $DLA_i[0:N/M-1]$, where $0 \leq i, j \leq M-1$, $0 \leq x \leq N/GCC-1$, and $x\times LCC \leq a_0 < a_1 < a_2 < ... < a_{\eta-1} < (x+1)\times LCC$.

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

Given a $kr \rightarrow r$ redistribution on $A[1:N]$ over $M$ processors, for a destination processor $P_i$, the first element of a message (assume that it was sent by source processor $P_j$) will be unpacked to $DLA_i[\alpha]$ and there are $\gamma$ array elements in $DLA_i[0:LCC-1]$ whose source processor is $P_j$, according to Lemmas 3 and 4, the first $\gamma$ array elements of the message will be unpacked to $DLA_i[0:\alpha+\gamma-1]$, the second $\gamma$ array elements of the message will be unpacked to $DLA_i[\alpha+kkr:\alpha+2\gamma-1]$, the third $\gamma$ array elements of the message will be unpacked to $DLA_i[\alpha+2\gamma:\alpha+3\gamma-1]$, and so on, where $0 \leq i, j \leq M-1$ and $0 \leq \alpha \leq M-1$. Therefore, for a destination processor $P_i$, if we know the values of $\gamma$ (the number of array elements in $DLA_i[0:LCC-1]$ whose source processor is $P_j$) and $\alpha$ (the position to place the first element of a message in $DLA_i$), we can unpack elements in messages to $DLA_i$ 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 $P_i$, the values of $\alpha$ and $\gamma$ can be computed by the following equations:

$$\gamma = \left\lfloor \frac{k}{M} \right\rfloor + \left\lfloor \frac{\text{mod}(rank(P_i)\times M-k, M)}{\text{mod}(k,M)} \right\rfloor \times 2$$

$$\alpha = \left\lfloor \frac{\text{rank}(P_i)\times k}{M} \right\rfloor + \left\lfloor \frac{\text{mod}(\text{rank}(P_i)\times k, M)}{M} \right\rfloor \times r$$

Where $\text{rank}(P_i)$ and $\text{rank}(P_j)$ are the ranks of processors $P_i$ and $P_j$. The notation "[1]" in equations (2) and (3) is called Iverson's function. It is defined as follows:

$$f(x) = 1$$

when $f(x)$ is true

$$f(x) = 0$$

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. $\text{max\_index} = \text{the length of the source local array of processor } P_i;$
3. the destination processor of $SLA_i[0]$ is $\eta = (k\times i) \mod M$;
/* Packing data sets */
4. $index = 1; length_3 = 1,$ where $\delta = 0, ..., M-1;$
while (index <= max_index)
7. { δ = η; j = 1;
8. while (j <= k) \&\& (index <= max_index))
9. { i = 1;
10. while ((i <= r) \&\& (index <= max_index))
11. { out_buffer[index++] = SLA[index];
12. i++;
13. if (δ = M) \ δ = 0 else δ++;
}
14. j++;
15. }
16. Send out_buffer to processor P_δ, where δ = 0, ...
17. M-1;

/* Receive phase */
18. max_cycle = max_index / kr,
19. Repeat m = min (M, k) times
20. Receive message buffer_in from source processor P_i;
21. Calculate the value of γ for message buffer_in using Equation (2);
22. Calculate the value of α for message buffer_in using Equation (3);
23. /* Unpacking messages */
24. index = α; length = 1; j = 0;
25. while (j <= max_cycle)
26. { index = α + j*kr; l = 1;
27. while (l <= γ) { DLAs[index+l] = buffer_in[length++];
28. l++;
29. }
30. j++;
31. }

3.2 Method for r→kr Redistribution
3.2.1 Send Phase

Lemma 5: Given an r→kr redistribution on A[1:N] over M processors, for a source processor P_i, and array elements in SLA[x×LCC::(x+1)×LCC-1], if the destination processor of SG(SLA[α_i]), SG(SLA[α_1]), ..., SG(SLA[α_n]) is P_j, then SG(SLA[α_j]), SG(SLA[α_1]), ..., SG(SLA[α_n]) are in the consecutive local array positions of SLA[0:M-1], where 0 ≤ i, j ≤ M-1, 0 ≤ x ≤ N/GCC-1, and x×LCC ≤ α_0 < α_1 < α_2 < ... < α_n < (x+1)×LCC.

Given an r→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 is P_j, and there are u classes, C_1, C_2, C_3, ..., and C_u in SLA[0:LCC-1] (assume that the indices of local array elements in these classes have the order C_1 < C_2 < C_3 < ... < C_u and the destination processors of C_1, C_2, C_3, ..., and C_u are P_1, P_2, P_3, ..., and P_u, respectively), according to Lemma 5, we know that j_1 = j_2 = \text{mod}(\langle C_u \times M \rangle / kr + j_1, M), j_3 = \text{mod}(\langle C_u \times M \rangle / kr + j_2, M), ...
\text{mod}(\langle C_u \times M \rangle / kr + j_{u-1}, M), where 1 ≤ u ≤ \text{min}(k, M) and \langle C_1, ..., C_u \rangle are class sizes of C_1, ..., C_u, respectively. This means that array elements SLA[0:|C_1|-1] will be sent to destination processor P_{j_1}, array elements SLA[|C_1|:|C_2|-1] will be sent to destination processor P_{j_2}, ..., and array elements SLA[|C_1|+|C_2|+...+|C_{u-1}|:|C_1|+|C_2|+...+|C_u|-1] will be sent to destination processor P_{j_u}. From Lemma 1, we know that SLA[0:|C_1|-1], SLA[|C_1|:|C_2|-1], SLA[|C_1|+|C_2|+...+|C_{u-1}|:|C_1|+|C_2|+...+|C_u|-1] have the same destination processor. Therefore, if we know the destination processor of SLA[0] and the values of \langle C_1, P_1 \rangle, \langle C_2, P_2 \rangle, ..., and \langle C_u, P_u \rangle, we can pack array elements in SLA to messages directly without computing the send data set and the destination processor set.

Given an r→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 can be computed by equation (5) and the number of array elements in SLA[0:LCC-1] whose destination processor is P_j can be computed by equation (4). Equations (4) and (5) are given as follows:

\[|C_i| = \langle \lfloor k/M \rfloor \times \text{mod}(\text{rank}(P_i) + M - \text{mod}(\text{rank}(P_j) + M, M)) \times \text{mod}(k, M) \rangle \times k \times \text{rank}(P_j) / k \]  
\[v = \lfloor \text{rank}(P_0) / k \rfloor \]  
Where \text{rank}(P_i) and \text{rank}(P_j) are the ranks of processors P_i and P_j. The notation "\lfloor x \rfloor" in equation (4) is called Iverson's function.

3.2.2 Receive Phase

Lemma 6: Given an r→kr redistribution on A[1:N] over M processors, for a source processor P_i, and array elements in SLA[0:LCC::(x+1)×LCC-1], if the destination processor of SG(SLA[α_i]), SG(SLA[α_1]), ..., SG(SLA[α_n]) is P_j, and SG(SLA[α_0]) = DG(DLAs[\nu]), then SG(SLA[α_0]) = DG(DLAs[\nu+M_i]), SG(SLA[α_1]) = DG(DLAs[\nu+2M_i]), ..., and SG(SLA[α_n]) = DG(DLAs[\nu+(n+1)×M_i]), where 0 ≤ \nu ≤ N/M-1.

Given an r→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 DLAs[\beta] and there are \delta array elements in DLAs[0:LCC-1] whose source processor is P_i. According to Lemma 6, the first \delta array elements of this message will be unpacked to DLAs[\beta+\nu×M_i], DLAs[\beta+M_i×M_i], DLAs[\beta+2M_i×M_i], ..., and DLAs[\beta+(\delta-1)×M_i], the second \delta array elements of the message will be unpacked to DLAs[\beta+kr×M_i+\nu×M_i], DLAs[\beta+kr×M_i+M_i], DLAs[\beta+kr×M_i+2M_i], ..., and DLAs[\beta+kr+(\delta-1)×M_i].
so on, where $0 \leq \beta \leq N/M-1$. Therefore, if we know the values of $\delta$ (the number of array elements in $DLA[0:LCC-1]$ whose source processor is $P_i$) and $\beta$ (the position to place the first element of a message in $DLA$), we can unpack messages to $DLA$, 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 $P_i$, the values of $\beta$ and $\delta$ can be computed by the following equations:

$$
\delta = (i/k) M + \text{mod}((M+\text{rank}(P_i)-
\text{mod}((\text{rank}(P_i) \times k, M), M ) \times k, M) ) \times r
$$

$$
\beta = \text{mod}((M + \text{rank}(P_i) -
\text{mod}((\text{rank}(P_i) \times k, M), M ) \times k, M) ) \times r
$$

Where $\text{rank}(P_i)$ and $\text{rank}(P_i)$ are the ranks of processors $P_i$ and $P_j$. The notation "$\text{mod}"$ 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. $i = MPI_{\text{Comm}}(rank)$;
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 $j_2, j_3, \ldots, j_w$;
6. Calculate class size $|C_j|, k$ 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.    \{ $i = 1$;
10.       while ((i <= m) && (index <= max_index))
11.          \{ $j = j_i; l = 1$;
12.             while ((l <= |C_j|)) && (index <= max_index))
13.                \{ out_buffer[length_j++ = SLA_i[index++];
14.             \}
15.          \}
16.       Send out_buffer to processor $P_j$, where $j = j_i, j_2, \ldots, j_w$;
/* 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 $\delta$ for $buffer_in_i$ using Equation (6);
21. Calculate the value of $\beta$ for $buffer_in_i$ using Equation (7);
/* Unpacking data sets */
22. $index \times \beta; length = 1; j = 0; count = 0$;
23. while ($i <= max_cycle$
24. \{ count = 1; index = $\beta + j \times kr$;
25. while (count <= $\delta$)
26. \{ $l = 1$;
27. while (l <= r) \{ DLA_i[index++];

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$\rightarrow$ (and vice-versa) redistribution with various array size and distribution factors. In Table 1, the execution time of ours in BLOCKCYCLIC(10) to BLOCKCYCLIC(2) redistribution is about 11% to 23% faster than that of thakur. For BLOCKCYCLIC(2) to BLOCKCYCLIC(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$\rightarrow$ 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$\rightarrow$ 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


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.

<table>
<thead>
<tr>
<th>SIZE</th>
<th>BLOCK-CYCLIC(10)</th>
<th>BLOCK-CYCLIC(2)</th>
<th>Improvement</th>
<th>SIZE</th>
<th>BLOCK-CYCLIC(2)</th>
<th>BLOCK-CYCLIC(10)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>72000</td>
<td>12.621</td>
<td>14.423</td>
<td>12.5%</td>
<td>72000</td>
<td>15.378</td>
<td>17.45</td>
<td>11.9%</td>
</tr>
<tr>
<td>144000</td>
<td>14.662</td>
<td>17.536</td>
<td>16.4%</td>
<td>144000</td>
<td>22.358</td>
<td>24.465</td>
<td>8.6%</td>
</tr>
<tr>
<td>216000</td>
<td>17.541</td>
<td>19.719</td>
<td>11%</td>
<td>216000</td>
<td>27.905</td>
<td>29.504</td>
<td>5.4%</td>
</tr>
<tr>
<td>288000</td>
<td>18.364</td>
<td>23.71</td>
<td>22.5%</td>
<td>288000</td>
<td>29.602</td>
<td>32.457</td>
<td>8.8%</td>
</tr>
<tr>
<td>360000</td>
<td>26.058</td>
<td>33.858</td>
<td>23%</td>
<td>360000</td>
<td>32.306</td>
<td>38.24</td>
<td>15.5%</td>
</tr>
</tbody>
</table>

Time unit : ms

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

<table>
<thead>
<tr>
<th>SIZE</th>
<th>BLOCK to CYCLIC</th>
<th>CYCLIC to BLOCK</th>
<th>Improvement</th>
<th>SIZE</th>
<th>BLOCK to CYCLIC</th>
<th>CYCLIC to BLOCK</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>72000</td>
<td>10.834</td>
<td>13.221</td>
<td>18.1%</td>
<td>72000</td>
<td>8.469</td>
<td>10.425</td>
<td>18.8%</td>
</tr>
<tr>
<td>144000</td>
<td>11.025</td>
<td>14.133</td>
<td>22%</td>
<td>144000</td>
<td>11.533</td>
<td>14.826</td>
<td>22.2%</td>
</tr>
<tr>
<td>216000</td>
<td>15.121</td>
<td>19.223</td>
<td>21%</td>
<td>216000</td>
<td>17.258</td>
<td>21.127</td>
<td>18.3%</td>
</tr>
<tr>
<td>288000</td>
<td>16.275</td>
<td>21.123</td>
<td>23%</td>
<td>288000</td>
<td>20.072</td>
<td>25.912</td>
<td>22.5%</td>
</tr>
<tr>
<td>360000</td>
<td>19.897</td>
<td>27.287</td>
<td>27.1%</td>
<td>360000</td>
<td>27.977</td>
<td>35.913</td>
<td>22.1%</td>
</tr>
</tbody>
</table>

Time unit : ms