Improving Performance of Multi-dimensional Array Redistribution on Distributed Memory Machines

Minyi Guo
University of Tsukuba
305 Tsukuba, Japan
guo@softlab.is.tsukuba.ac.jp

Yoshiyuki Yamashita
University of Tsukuba
305 Tsukuba, Japan
yaman@lang.esys.tsukuba.ac.jp

Ikuo Nakata
University of Information
Science and Library
305 Tsukuba, Japan
nakata@ulis.ac.jp

Array redistribution is required very often in programs on distributed memory parallel computers. It is essential to use efficient algorithms for redistribution, otherwise the performance of the programs may degrade considerably. In this paper, we focus on automatic generation of communication routines for multi-dimensional redistribution. The principal advantage of this work is to gain the ability to handle redistribution between arbitrary source and destination processor sets and between arbitrary source and destination distribution schemes. We have implemented these algorithms using Parallelware communication library. Some optimization techniques for our algorithms are also proposed in this paper. Experimental results show the efficiency and flexibility of our techniques compared to the other redistribution works.

keyword Parallelizing compiler, HPF, Array redistribution, Automatic data distribution, Automatic parallelization and optimization.

1. Introduction

The difficulty of programming for distributed memory parallel computers, such as CP-PACS[11], has been recognized as a major obstacle to their wide-spread acceptance. To address this problem, significant efforts have been aimed at source-to-source parallelizing compilers and languages such as HPF[6]. These techniques or languages provide the programmers with the facilities to select good data distribution schemes or allow annotations to specify the mapping of arrays among the processors of the target distributed memory machine. The block/cyclic distribution is the most general regular data distribution supported in these parallelizing compilers and languages. Furthermore, many applications and kernels, such as ADI(Alternating Direction Integration), 2D FFT(Fast Fourier Transform) and signal processings, require different array distributions at different computation phases for efficient execution on distributed memory machines. The HPF standard provides directives for such dynamic array redistribution. Moreover, it can also dynamically specify the processor subset(see HPF-2[5]), that is, task parallelism is used, where array elements are redistributed between different sets of processors. This means array redistribution may change not only the distribution scheme for the arrays but also the set of processors that the arrays are distributed on. This type of redistribution has been a primary motivation for our work.

In this paper, we propose an algorithm for generation of redistribution routines. Since multi-dimensional arrays appear in most of the applications of data redistribution while most of current works about redistribution are one-dimension oriented, we focus on dealing with multi-dimensional array redistribution. For the sake of simplicity, the case of 2D array is demonstrated in the paper. The primary contributions of our work are that our redistribution algorithm can handle array redistribution between arbitrary source and destination processor sets and between arbitrary source and destination distribution schemes, and are more efficient than some other redistribution algorithms. We also propose some communication optimization techniques to improve the communication performance.

The rest of the paper is organized as follows. Section 2 discusses important related work in the area. In Section 3, we briefly describe regular data redistribution. Our redistribution algorithm and some optimizations are developed in Section 4. Section 5 gives some experimental results for our algorithm and compares the efficiency and flexibility with other works. Finally, we conclude this paper and discuss possible future extensions to our work in Section 6.

2. Related Works

Any technique that handles HPF array assignments [17][14][9] can be used to compile redistribution: the induced communication is one of an array assignment A = B, where B is mapped as the source and A as the target. However, none of these techniques considers handling of different processor sets, multidimensional distributions, communication generation, and local address management. Thus, only the optimized techniques dedicated to their problems have been proposed.

The works in [15],[16] describe a redistribution library for run-time support in a HPF compiler. The routines described treat possible source-target data distributions in a pairwise manner. This allows them to use efficient methods for specific cases of source-target data distributions. A redistribution technique based on a special local data descriptors called pitfalls has been devised[12]. It can treat arbitrary source and target processor sets. However, all of these works are based on 1D redistribution algorithm, there is no capability of solving more complex redistribution applications, such as shape changing redistribution(along some distributed dimensions). In such a case, an expensive run-time resolution approach is used. Further, the approach used for multidimensional array redistribution involves a series of one-dimensional redistributions, which can be costly. Reference [3] describes a redistribution library routine provided as part of the ScaLAPACK library. While this routine can handle arbitrary BLOCK/CYCLIC redistributions of multidimensional arrays, it is not clear whether it can handle arbitrary sending/receiving processor sets or not. Reference [2] proposes a general theoretical framework for data redistribution; this framework is currently being implemented. In [18], a strip mining approach to redistribution is proposed in order to overlap communication with computation and thus reduce the overhead of redistribution. While this is an interesting approach, it only has the ability to handle the special redistribution scheme, applicability to complex redistributions is not clear.

Work by Bixby *et al* [1] and Kremer[10][8] formulates the data redistribution problem in the form of a 0-1 integer programming problem. For each phase, a number of candidate partial data layouts are enumerated along with the estimated execution costs. Since each phase only specifies a partial data distribution, redistribution constraints can cross over multiple phases, thereby requiring the use of 0-1 integer programming. Kalns and Ni[7] present a technique for mapping data to processors in order to minimize the total amount of data that must be communicated during redistribution. A multi-phase redistribution approach is suggested in [13]. The key idea of this work is to perform the redistribution as a sequence of redistributions such that the total cost of the sequence is less than that of direct redistribution. This idea is partly applied in our optimal redistribution algorithm.

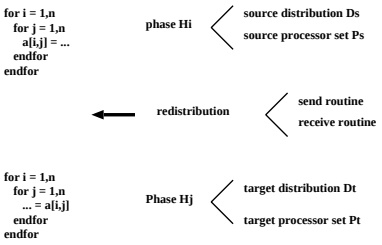

Figure 1. Requirement for redistribution between two phases

3. The Redistribution Problem

Motivated by the need for redistribution, we define the regular redistribution problem[12]. A redistribution $\mathcal{R}$ is the change of a distribution that, given an multi-dimensional array A on a set of source processors $\mathcal{P}_s$ with distribution $\mathcal{D}_s$, transfer all the elements of the array to a set of target processors $\mathcal{P}_t$ with a target distribution $\mathcal{D}_t$. In a general case, $\mathcal{D}_s$ and $\mathcal{D}_t$ can specify arbitrary regular data distributions along each dimension of the array. Therefore, the redistribution routines need to figure out exactly what data needs to be sent(received) by each source(target) processor.

The multi-dimensional redistribution can be divided into two types:

* The shape of the processor grid is not changed. All of the other 1D redistribution algorithms can be applied into this situation where the algorithm is used in each dimension independently.

* The shape of the processor grid is changed. The 1D redistribution can not be simply used to such type of redistribution. For example, to perform the redistribution from (BLOCK, *) to (*,BLOCK), if we apply the 1D redistribution algorithm, we must first redistribute the array from (BLOCK,*) to (*,*), then do from (*,*) to (*,BLOCK). That means each processor must own a global array.

In the following discussion, we concentrate our attention on two-dimensional array redistribution. Our algorithm can be easily generalized to that of higher dimensions. We assume that the global array size is denoted as $n_1 \times n_2$ and local array size is denoted as

Algorithm 1 *send*

```
for i = 1, L1
   for j = 1, L2
      (ig, jg) = local_index_2_global_index((i, j),
                    src_distribution, myid);
      dest_proc = global_index_2_dest_proc((ig, jg),
                    dest_distribution);
      pack(As[i,j], buffer, dest_proc);
   endfor
endfor
for proc = 0, p-1
   send(buffer, proc);
endfor
```

Algorithm 2 *receive*

```
for proc = 0, p-1
   receive(buffer, proc)
   for k = 0, d_proc
      (il, jl) = global_index_2_local_index((ig, jg),
                    dest_distribution, myid);
      unpack(buffer, At[il, jl]);
   endfor
endfor
```

d_{proc}: the length of transmission data between *myid* and *proc*.

Figure 2. The naive redistribution algorithm– send and receive.

$L_1 \times L_2$. All arrays are indexed starting from 1 while processors are numbered starting from 0. The distribution schemes allowed in a dimension are BLOCK, CYCLIC, and All(* in HPF). p processors compose a 2D processor grid $p_1 \times p_2$ where p_1 and p_2 are the number of processors of first and second dimension. *myid* is the logical number of processor executing the program.

It is possible to use a naive approach to perform redistribution. In this approach, at the sending phase, each processor scans its local array, determines the global index from the local index for the source distribution, determines the destination processor for that array element under the target distribution and inserts the element into a buffer reserved for that destination processor. After all the local array elements are scanned and inserted into buffers, a possibly empty buffer is sent to and received from every processor other than itself. The receiving phase is symmetric to the sending phase(Figure 2).

This method involves multiple conversions among global, local, and processor indices for every local array element, which has an unacceptably high indexing and communication overhead thus it cannot be used practically.

4. The Redistribution Algorithm Based on Ordered Index Set

4.1. Ordered Index Set

Consider a two-dimensional array A. Its local elements distributed onto a processor can be represented by its global index set whose elements are pairs of index (i,j). That is, for each processor P_k, there exists a index set

$$IS_{A,k} = \{(i,j) | A[i,j] \textit{ is distributed onto } P_k\}$$

$IS_{A,k}$ is a ordered index set whose elements (i,j) have the lexicographical order.

In the following definitions, we use a 5-tuple to define such an ordered index set.

Definition 1 For a processor P_k, the ordered index set of local array elements distributed onto it are represented by a 5-tuple,

$$D_k =< O, v_1, l_1, v_2, l_2 >$$

where
$O = (i_o, j_o) \in N^2$ is an array index pair which indicates an origin of the coordinate axes;
$v_1, v_2 \in N$ are strides(the distance between the two consecutive elements in the same processor) in 1st-dimension and 2nd-dimension respectively; and
$l_1, l_2 \in N$ are the number of elements in 1st-dimension and 2nd-dimension, respectively. ■

Definition 2 Given a 5-tuple $D =< O, v_1, l_1, v_2, l_2 >$, the ordered index set corresponding to D is defined as follow:

$$IS[D] = \{(i,j) | i = i_o + v_1 * (m_1 - 1), j = j_o + v_2 * (m_2 - 1),$$
$$1 \le m_1 \le l_1, 1 \le m_2 \le l_2 \ \}$$

The minimum element of $IS[D]$ is $O = (i_o, j_o)$, denoted as $\min(IS[D])$. ■

From the above definitions, we can conclude that any BLOCK/CYCLIC(not including block-cyclic) distribution scheme for an array can be expressed by using 5-tuples. Because any local array distributed onto a processor has its minimum index, this is the originator

O in a 5-uple. Any distribution scheme is a permutation of BLOCK, CYCLIC and *. For BLOCK and * distribution scheme, the stride $v_1(v_2)$ is 1 and for CYCLIC the stride $v_1(v_2)$ is p_1 (p_2). The numbers of elements of each dimension, l_1 and l_2, are local array sizes in that dimension.

Example 1 Let $p = p_1 \times p_2$ be number of processors, each processor P_k is identified by its coordinates (k_1, k_2) with $k_1 \in \{0..p_1 - 1\}$ and $k_2 \in \{0..p_2 - 1\}$, and assume that n_1 and n_2 are divisible by p_1 and p_2 respectively. The followings are some distribution schemes expressed in 5-tuple.
(BLOCK,*):

$$D_k =< ((n_1/p_1) * k_1 + 1, 1), 1, n_1/p_1, 1, n_2 >$$

(BLOCK,BLOCK):

$$D'_k =< (k_1 * (n_1/p_1) + 1, k_2 * (n_2/p_2) + 1),$$
$$1, n_1/p_1, 1, n_2/p_2 >$$

(CYCLIC,CYCLIC):

$$D''_k =< (k_1 + 1, k_2 + 1), p_1, n_1/p_1, p_2, n_2/p_2 > \quad \blacksquare$$

Definition 3 The minimum common element of two 5-tuples D' and D'' $O_{min} = minO(D', D'')$ is defined as

$$O_{min} \in IS[D'] \wedge O_{min} \in IS[D''], \text{ and}$$
$$\forall d.d \in IS[D'] \wedge d \in IS[D''] \Rightarrow d \geq O_{min}.$$

If there exists no common element of D' and D'', we denote

$$O_{min} = (\perp, \perp). \quad \blacksquare$$

Definition 4 Let $D' =< O', v'_1, l'_1, v'_2, l'_2 >, D'' =< O'', v''_1, l''_1, v''_2, l''_2 >$, the intersection of D' and D'' is defined as

$$D = D' \cap D'' =< O, v_1, l_1, v_2, l_2 > .$$

The values of O, v_1, l_1, v_2, l_2 are defined as follows respectively,

$$O = (i_o, j_o) = minO(D', D''),$$

if $O = (\perp, \perp)$ then $IS[D] = \emptyset$, we denote $D = \phi$,

$$v_1 = \text{lcm}(v'_1, v''_1),\ v_2 = \text{lcm}(v'_2, v''_2)$$

$$l_1 = \min(e'_1 - i_o, e''_1 - i_o) \text{ div } \text{lcm}(v'_1, v''_1) + 1$$

where e'_1, e''_1 are the end indices of the 1st-dimension in $IS[D']$ and $IS[D'']$ respectively. $e'_1 = i'_o + v'_1 * (l'_1 - 1)$, $e''_1 = i''_o + v''_1 * (l''_1 - 1)$

$$l_2 = \min(e'_2 - j_o, e''_2 - j_o) \text{ div } \text{lcm}(v'_2, v''_2) + 1$$

where e'_2, e''_2 are the end indices of the 2nd-dimension in $IS[D']$ and $IS[D'']$ respectively. $e'_2 = j'_o + v'_2 * (l'_2 - 1)$, $e''_2 = j''_o + v''_2 * (l''_2 - 1)$

Because $v_1, v_2, l_1, l_2 \in N$ and $O \in N^2$, D is also a 5-tuple we defined in Definition 1. $\blacksquare$

Theorem 1 Let $D' =< O', v'_1, l'_1, v'_2, l'_2 >$ and $D'' =< O'', v''_1, l''_1, v''_2, l''_2 >$ are arbitrary two 5-tuples, we have

$$IS[D' \cap D''] = IS[D'] \cap IS[D'']$$

Proof. $IS[D'] \cap IS[D''] = \emptyset \Longleftrightarrow \neg\exists d.d \in IS[D'] \wedge d \in IS[D''] \Longleftrightarrow O = (\perp, \perp)\ of\ D' \cap D'' \Longleftrightarrow IS[D' \cap D''] = \emptyset$.
If $S = IS[D'] \cap IS[D''] \neq \emptyset$, Let $O_{min} = (i_o, j_o) \in S$. From the definition 2,

$$IS[D'] = \{(i', j') | i' = i'_o + v'_1 * (m'_1 - 1),$$
$$j' = j'_o + v'_2 * (m'_2 - 1),$$
$$1 \leq m'_1 \leq l'_1, 1 \leq m'_2 \leq l'_2\}$$

and

$$IS[D''] = \{(i'', j'') | i'' = i''_o + v''_1 * (m''_1 - 1),$$
$$j'' = j''_o + v''_2 * (m''_2 - 1)$$
$$1 \leq m''_1 \leq l''_1, 1 \leq m''_2 \leq l''_2\}.$$

Since $(i_o, j_o) \in IS[D'] \wedge (i_o, j_o) \in IS[D'']$, we have,

$$i_o = i'_o + v'_1 * t'_1, \quad j_o = j'_o + v'_2 * t'_2$$

and

$$i_o = i''_o + v''_1 * t''_1, \quad j_o = j''_o + v''_2 * t''_2$$

We also assume that, in set S, the successor of i_o and j_o in its own dimension is i_1 and $j_1((i_1, j_1) \in S)$, then,

$$i_1 = i'_o + v'_1 * (t'_1 + s'_1), \quad j_1 = j'_o + v'_2 * (t'_2 + s'_2)$$

$$i_1 = i''_o + v''_1 * (t''_1 + s''_1), \quad j_1 = j''_o + v''_2 * (t''_2 + s''_2)$$

Simplify the above formulas, we have,

$$i_1 = i_o + v'_1 * s'_1, \quad j_1 = j_o + v'_2 * s'_2$$

$$i_1 = i_o + v''_1 * s''_1, \quad j_1 = j_o + v''_2 * s''_2$$

hence,

$$v'_1 * s'_1 = v''_1 * s''_1, \quad v'_2 * s'_2 = v''_2 * s''_2$$

must be satisfied. The solutions are

$$s'_1 = \text{lcm}(v'_1, v''_1)/v'_1, \quad s''_1 = \text{lcm}(v'_1, v''_1)/v''_1$$

$$s'_2 = \text{lcm}(v'_2, v''_2)/v'_2, \quad s''_2 = \text{lcm}(v'_2, v''_2)/v''_2$$

That is,

$$i_1 = i_o + \text{lcm}(v'_1, v''_1) * 1, \quad j_1 = j_o + \text{lcm}(v'_2, v''_2) * 1$$

Let the number of elements in 1st-dimension and 2nd-dimension of S is l_1 and l_2 respectively The other elements (i_{m_1}, j_{m_2}) in S can be similarly expressed as

$$i_{m_1} = i_o + \mathrm{lcm}(v_1', v_1'') * m_1, \quad j_{m_2} = j_o + \mathrm{lcm}(v_2', v_2'') * m_2$$

where $m_1 \in \{0..l_1 - 1\}$ and $m_2 \in \{0..l_2 - 1\}$. Moreover, consider in $IS[D']$, we can also obtain

$$i_{m_1} \leq i_o' + v_1' * (l_1' - 1) = i_o + \mathrm{lcm}(v_1', v_1'') \frac{l_1' - t_1' - 1}{s_1'}$$

$$j_{m_2} \leq j_o' + v_2' * (l_2' - 1) = j_o + \mathrm{lcm}(v_2', v_2'') \frac{l_2' - t_2' - 1}{s_2'}$$

The same reason applying to $IS[D'']$, we have

$$i_{m_1} \leq i_o + \mathrm{lcm}(v_1', v_1'') \frac{l_1'' - t_1'' - 1}{s_1''},$$

$$j_{m_2} \leq j_o + \mathrm{lcm}(v_2', v_2'') \frac{l_2'' - t_2'' - 1}{s_2''},$$

This means that l_1, l_2 must satisfy

$$l_1 - 1 = \min(\lfloor \frac{l_1' - (t_1' + 1)}{s_1'} \rfloor, \lfloor \frac{l_1'' - (t_1'' + 1)}{s_1''} \rfloor)$$

$$l_2 - 1 = \min(\lfloor \frac{l_2' - (t_2' + 1)}{s_2'} \rfloor, \lfloor \frac{l_2'' - (t_2'' + 1)}{s_2''} \rfloor)$$

Replace the values of t_i', t_i'' and $s_i', s_i''(i = 1, 2)$ to the above formulas, we obtain

$$l_1 = \min(e_1' - i_o, e_1'' - i_o) \text{ div } \mathrm{lcm}(v_1', v_1'') + 1$$

where $e_1' = i_o' + v_1' * (l_1' - 1),\ e_1'' = i_o'' + v_1'' * (l_1'' - 1)$

$$l_2 = \min(e_2' - j_o, e_2'' - j_o) \text{ div } \mathrm{lcm}(v_2', v_2'') + 1$$

where $e_2' = j_o' + v_2' * (l_2' - 1),\ e_2'' = j_o'' + v_2'' * (l_2'' - 1)$. Hence,

$$\begin{aligned} S = \{(i,j) | & i = i_o + \mathrm{lcm}(v_1', v_1'') * (m_1 - 1), \\ & j = j_o + \mathrm{lcm}(v_2', v_2'') * (m_2' - 1), \\ & 1 \leq m_1 \leq l_1, 1 \leq m_2 \leq l_2\} \\ = IS[D' \cap D''] & \end{aligned}$$

■

In the following discussion, we sometimes use a 5-tuple D to indicate its ordered index set IS[D] when there is no confusion in the context.

Example 2 For the distributions $D_k, D_k', D_k'', k = (k_1, k_2), k_i \in \{0..p_i - 1\}, i = 1, 2$ given in example 1, assume $p = 4 \times 1$ when distribution is (BLOCK,*) and $p = 2 \times 2$ when the distributions are (BLOCK,BLOCK) or (CYCLIC,CYCLIC).

$$D_{0,0} \cap D_{0,0}' =< (1,1), 1, n_1/4, 1, n_2/2 >$$

$$D_{0,0} \cap D_{0,1}' =< (1, n_2/2 + 1), 1, n_1/4), 1, n_2/2 >$$

$$D_{1,0} \cap D_{1,0}' = \phi$$

$$D_{0,0}' \cap D_{0,1}'' =< (1,2), 2, n_1/2, 2, n_2/2 >$$

■ *11*

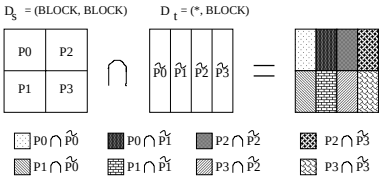

Figure 3. The intersection of (BLOCK,BLOCK) and (*,BLOCK) with 4 processors

Figure 3 shows the intersections of each pair of source processor P_i and destination processor $\tilde{P}_j$. when $\mathcal{P}_s = 2 \times 2$, $\mathcal{D}_s$ = (BLOCK,BLOCK), $\mathcal{P}_t = 1 \times 4$, $\mathcal{D}_t$ = (*, BLOCK).

4.2. The Redistribution Algorithm

Before the redistribution, the global array A is distributed into the local arrays according to the source distribution scheme $\mathcal{D}_s$ and the source processor grid $\mathcal{P}_s$. In other words, the global array A is composed of the local arrays which are owned by each processor. The global array A can also be represented as a 5-tuple $D =< (1,1), 1, n_1, 1, n_2 >$, then it can be denoted as

$$D = D_0 \cup D_1 \cup \ldots \cup D_{p-1}$$

where $D_k, k \in \{0..p-1\}$ is the local 5-tuple of each processor under the source distribution, and $D_i \cup D_j$ is the abbreviated form of $IS[D_i] \cup IS[D_j]$. Similarly, after the redistribution, the global array A is composed of the local arrays owned by each processor with the destination distribution scheme $\mathcal{D}_t$ and the destination processor grid $\mathcal{P}_t$, it also can be marked as

$$D = \tilde{D}_0 \cup \tilde{D}_1 \cup \ldots \cup \tilde{D}_{\tilde{p}-1}$$

where $\tilde{D}_k, k \in \{0..\tilde{p}-1\}$ is the local 5-tuple of each processor under the target distribution. Since

$$D = D \cap D = (D_0 \cup \ldots \cup D_p) \cap (\tilde{D}_0 \cup \ldots \cup \tilde{D}_{\tilde{p}})$$

$$= \bigcup_{k=0}^{p} D_k \cap (\tilde{D}_0 \cup \ldots \cup \tilde{D}_{\tilde{p}})$$

That is, for the source processor P_k, if we get the intersections of D_k and each destination index set, we can obviously know which array elements should be sent to which processor.

Similarly, we can also get the following formula,

$$D = \bigcup_{k=0}^{\tilde{p}} \tilde{D}_k \cap (D_0 \cup \ldots \cup D_p)$$

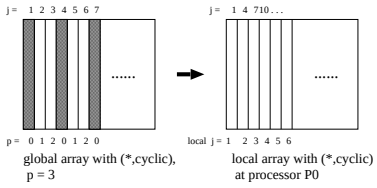

Figure 4. Global intervals and local intervals

That means at the receiving phase, for the destination processor $\tilde{P}_k$, the elements received from the source processor P_j are intersection of $\tilde{D}_k$ and D_j. This results in the following theorem.

Theorem 2 Let D_i be the ordered index set of source processor P_i under the source distribution scheme $\mathcal{D}_s$, and $\tilde{D}_j$ be the ordered index set of target processor $\tilde{P}_j$ under the target distribution scheme $\mathcal{D}_t$, In a redistribution from $(\mathcal{P}_s, \mathcal{D}_s)$ to $(\mathcal{P}_t, \mathcal{D}_t)$, the data that each processor P_i $(0 \leq i < p)$ should communicate with target processor $\tilde{P}_j, (0 \leq j < \tilde{p})$, are indicated by the intersection of D_i and $\tilde{D}_j$, ■

The redistribution routine includes two parts, the send routine and the receive routine. The send routine is executed by the source processors. It analyzes the communication requirements for each array and pack the message into a contiguous buffer for each destination processor that needs data from the calling source processor; then sends message to the corresponding destination processor. The receive phase essentially does the reverse. Since an element of ordered index set is a global index of the array, in order to pack(unpack) the local array elements into(from) a buffer, we have to generate local addresses for accessing elements of a local array during redistribution. We use the formula shown in the following table for generating local origin coordinate O.

block	cyclic	all
$l = (g-1)\%(n_i/p_i)+1$	$l = (g-1)/p_i+1$	$l = g$

l:local index, g:global index, n_i: global array size of the dimension i, p_i: the number of processors of dimension i.

Furthermore, the ordered index set of local array has not always the same stride as one of global array. For example, if the array is distributed with CYCLIC in a dimension, the stride is p_i(the number of processors of this dimension), but the stride of local array

Algorithm 3 *send routine*
if *myid* $\in \mathcal{P}_s$ **then**
  get my source indexset $D' =< O', v'_1, l'_1, v'_2, l'_2 >$
  / D' is determined by myid, the source distribution scheme */*
  for $\tilde{P}_k \in \mathcal{P}_t$, $k = 1, 2, \ldots, \tilde{p}$
    get each target indexset $D''_1, D''_2, \ldots, D''_{\tilde{p}}$;
    / D''_k is determined by P_k and the destination distribution scheme */*
    compute the intersection of $D' \cap D''_k$
    $D_1 = D' \cap D''_1, D_2 = D' \cap D''_2, \ldots,$
    if *myid* $\neq \tilde{P}_k$ **then**
      if $D_k \neq \phi$ **then**
        / D_k is global index set, it should be convertedd to local index under source distribution scheme */*
        convert global $O(i_g, j_g)$ *to local* $O(i_l, j_l)$
        / compute the local stride* s_1, s_2 *of source local array As: */*
          $s_1 = lcm(v'_1, v''_1)/v'_1$;
          $s_2 = lcm(v'_2, v''_2)/v'_2$;
        pack the local D_k *into buffer,*
        send(buffer, $\tilde{P}_k$*);*
      endif
    endif
  endfor
endif

Figure 5. The redistribution algorithm of the send routine

is 1(see Figure 4). To compute the stride for packing(unpacking) local array, we use local strides

$$s_1 = \text{lcm}(v'_1, v''_1)/v'_1 \qquad s_2 = \text{lcm}(v'_2, v''_2)/v'_2$$

for packing and

$$t_1 = \text{lcm}(v'_1, v''_1)/v''_1 \qquad t_2 = \text{lcm}(v'_2, v''_2)/v''_2$$

for unpacking.

The algorithms of the send routine and the receive routine are shown in Figure 5 and Figure 6, respectively.

4.3. Optimizations of the Algorithm

The algorithms in Figure 5 and Figure 6 are the original algorithms derived from Theorem 2. In the following SubSections, we give some improvements for our algorithm.

Algorithm 4 *receive routine*
if $myid \in \mathcal{P}_t$ **then**
  allocate destination local array At;
  get my destination indexset
  $D'' =< O'', v_1'', l_1'', v_2'', l_2'' >$
  / D'' is determined by myid, destination distribution scheme */*
  for $P_k \in \mathcal{P}_s$, $k = 1, 2, \ldots, p$
    get each source indexset $D_1', D_2', \ldots, D_p'$;
    / D_k'' is determined by P_k and the source distribution scheme */*
    compute the intersection of $D' \cap D_k''$
    $D_1 = D' \cap D_1'', D_2 = D' \cap D_2'', \ldots,$
    if $myid \neq P_k$ **then**
      if $D_k \neq \phi$ **then**
        receive(buffer, P_k*);*
        compute the local interval t_1, t_2 *of At:*
          $t_1 = lcm(v_1', v_1'')/v_1''$;
          $t_2 = lcm(v_2', v_2'')/v_2''$;
        unpack the buffer into At,
      endif
    else
      / memory copy */*
      convert the global index (i, j) *to the local index* (i_s, j_s) *under the source distribution* D_s*, and* (i_t, j_t) *under the destination distribution* D_t;
      $At[i_t, j_t] = As[i_s, j_s]$;
    endif
  endfor
endif
if $myid \in \mathcal{P}_s$ **then**
  free(As);

Figure 6. The redistribution algorithm of the receive routine

4.3.1. Multi-phase Redistribution

In the algorithms shown in Figure 5 and Figure 6, each source(target) processor has to communicate with all of target(source) processors and simultaneously communicate with only one target(source) processor in the send(receive) phase (see Figure 7(a)). The number of the communications are $O(\tilde{p}_1 \times \tilde{p}_2)$ for the send phase and $O(p_1 \times p_2)$ for the receive phase. In [13], a multi-phase approach is adopted to reduce the communication cost. Applying the method to our algorithm, we first redistribute the array along the first dimension from $\mathcal{P}_s$ to $\mathcal{P}_m$, where the number of processors of the first and the second dimensions of $\mathcal{P}_m$ are the same as the first dimension of $\mathcal{P}_t$ and the second dimension of $\mathcal{P}_s$ respectively, that is, the processor grid of $\mathcal{P}_m$ is $p_1 \times \tilde{p}_2$. We then redistribute the array along the second dimension from $\mathcal{P}_m$ to $\mathcal{P}_t$. For this approach, processors in one dimension can simultaneously communicate with other target(source) processors in same dimension(Figure 7(b)). The communication times are $O(\tilde{p}_1 + \tilde{p}_2)$ for the send phase and $O(p_1 + p_2)$ for the receive phase.

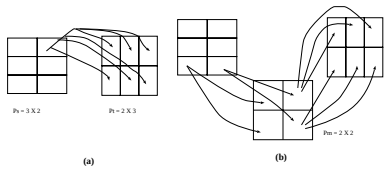

Figure 7. The number of communication times before and after optimization.

4.3.2. Communication Scheduling

During a redistribution using the algorithms of Figure 5 and Figure 6, we see that each source processor sends its messages to the target processor P_0 at the step 1, and sends the messages to P_1 at step 2, and so on. This results in that each target processor receives all of its messages simultaneously, which may lead to communication bottlenecks. We avoid such bottlenecks by scheduling the communication starting with P_{myid} at first, then with P_{myid+1} at second step, and so on. This method is to have that each source processor starts with a different target processor index and each target processor receives its own messages at the same time[12].

4.3.3. Communication Aggregation

It is difficult to improve the performance of the redistribution problem when the shape of the processor grid is changed. In accordance with widely used(ADI, FFT) redistribution (X,*) to (*,X) where X can be block or cyclic distribution(the shape of the processor grid is changed from $p \times 1$ to $1 \times p$), the strip-mining technique has been proposed in [18]. The authors reduce the redistribution overhead by overlapping the communication with computation. Here, we propose a communication aggregation technique to reduce the overhead by decreasing the frequencies of communication.

With respect to the redistribution (X,*) to (*,X), Each processor requires p times to communicate with

$p-1$ other processors and each time transmit one intersection of two ordered index set. Each intersection has the size $(n_1 \times n_2)/p^2$, which is $1/p$ of the local array size. However, if each source node sends m intersection to m target nodes simmultaneously, the number of communication times can be reduced. In our algorithm, we replace only one $D' \cap D''_k$ by union of some neighbour intersections. The algorithm is modified as follows(D_k is denoted as the intersection $D' \cap D''_k$):

For each sorurce processor,

Step 1: broadcast $D_{myid} \cup D_{myid+1} \cup \ldots \cup D_{myid+m-1}$
to $P_{myid}, \ldots P_{myid+m-1}$

Step 2: broadcast $D_{myid+m} \cup D_{myid+m+1} \cup \ldots \cup D_{myid+2m-1}$
to $P_{myid+m}, \ldots P_{myid+2m-1}$

.....

Step s: broadcast $D_{myid+s*m} \cup D_{myid+s*m+1} \cup \ldots \cup D_{myid+(s+1)*m-1}$
to $P_{myid+s*m}, \ldots P_{myid+(s+1)*m-1}$

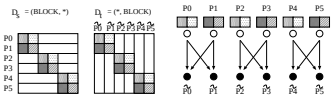

Figure 8. The first step of communication aggregation.

Figure 8 shows the first step where $p = 6$ and $m = 2$. Although the above improvement reduces the frequencies of the communication, the transmission data size and pack(unpack) buffer size are increased. Because the communication start-up time is much larger than unit data transmission time on distributed memory machines, the optimized algorithm can achieve good performance.

4.4. Performance Estimation of the Algorithms

In order to compare the performance of the proposed algorithms, in this section we estimate the computation and communication costs of the naive algorithm, the 1D redistribution algorithm in [15] and [16], and our algorithm, where all the costs include the send and receive parts. The various quantities used in the estimation formulas are:

- T_{pk} — the time for packing/unpacking an array element to(from) a buffer;
- T_{gl} — the time for computing the conversions among the global, local index and the processor index;
- d_k — the data size which communicate with processor P_k, where

$$L_1 \times L_2 = \Sigma_{k=0}^{p-1} d_k \times P_k;$$

- $\bar{d}$ — the average transmission data size; that is, $\bar{d} = (\Sigma_{k=0}^{p-1} d_k)/p$;
- Δ_c — some computations other than T_{pk} and T_{gl};
- Δ_p — the number of processors that do not cause the communication;
- T_{comp} — computation cost;
- $T_{comm} = T_s + d * T_d$ — communication cost, where T_s is the start-up time and T_d is the unit data transmission time.

The followings are the estimation formulas of the algorithms.

♠ Naive algorithm:

$$T_{comp} \approx (L_1 * L_2) * (3T_{gl} + 2T_{pk})$$

$$T_{comm} \approx p * (T_s + \bar{d} * T_d)$$

♡ 1D algorithm for the shape of the processor grid retaining redistribution:

$$T_{comp} \approx (L_1 + L_2) * (2T_{gl} + 2T_{pk})$$

$$T_{comm} \approx p_1 * (T_s + \bar{d}^2 * T_d) + p_2 * ((T_s + \bar{d}^1 * T_d)$$

$$= (p_1 + p_2) * T_s + (p_1 * \bar{d}^2 + p_2 * \bar{d}^1) * T_d$$

♢ Our algorithm for the shape of the processor grid retaining redistribution:

$$T_{comp} \approx p * (4T_{gl} + 2T_{pk} + \Delta_c)$$

$$T_{comm1} \approx (p_1 + p_2 - \Delta_p) * T_s + (p_1 * \bar{d}^2 + p_2 * \bar{d}^1) * T_d$$

♣ Our algorithm for the shape of the processor grid changing redistribution:

$$T_{comp} \approx (p_1 + p_2) * (4T_{gl} + 2T_{pk} + \Delta_c)$$

$$T_{comm2} \approx (p - \Delta_p) * (T_s + \bar{d} * T_d)$$

especially, with the communication aggregation for redistribution (X,*) to (*,X), the communication cost of our algorithm becomes

$$T_{comm3} = \frac{p}{m} * (T_s + m * \frac{n_1 \times n_2}{p^2} * T_d)$$

where m is the number of neighbour intersections.

Table 1. Results for "the shape of the processor grid retaining redistribution" on CP-PACS(time in second).

Redistributions	Algorithms	Number of processors						
		4	8	16	32	64	128	256
(CYCLIC,CYCLIC)	Naive	4.37	4.31	3.45	3.38	3.07	2.75	2.48
to	1D algo	0.49	0.36	0.26	0.20	0.18	0.12	0.09
(CYCLIC,BLOCK)	G_{algo}	0.30	0.21	0.15	0.13	0.11	0.09	0.07
(BLOCK,BLOCK)	Naive	5.08	4.45	4.38	3.97	3.67	3.17	2.29
to	1D algo	0.67	0.46	0.28	0.21	0.19	0.13	0.09
(CYCLIC,CYCLIC)	G_{algo}	0.54	0.39	0.25	0.20	0.16	0.10	0.09

5. Implementation and Experimental Results

The redistribution routines described in Section 4 have been implemented on CP-PACS, a 2048-processor MIMD distributed memory supercomputer developed at University of Tsukuba. These routines require the information about the source and target processor grids and source and target distribution schemes for each of arrays being redistributed. For purpose of performance evaluation of our algorithm and comparison with other redistribution works, we also implemented one-dimensional algorithms proposed in [15] and [16] and the naive approach described in Section 3.1. As we have discussed in Section 2 and Section 3, however, the one-dimensional algorithm can only handle "the shape of the processor grid retaining redistribution".

Table 1 shows the results of executing two kinds of source-target distribution pairs: (CYCLIC, CYCLIC) to (CYCLIC, BLOCK) and (BLOCK, BLOCK) to (CYCLIC, CYCLIC), as the typical cases for redistribution in only one dimension and all two dimensions with the array size of 1024×1024, where Naive, 1D algo and G_{algo} indicate the naive approach, one-dimensional algorithm and our algorithm(without communication scheduling optimization) respectively.

To show the efficiency and flexibility for the "the shape of the processor grid changing redistribution", we experiment on some cases when either the source processor grid is different from the target processor grid, or at least one dimension of the array is collapsed before or after redistribution. Table 2 shows some experimental results with array sizes of 300×300 and 600×600. The comparison of the performances is done between the naive approach and our algorithm(without optimizations), since one-dimensional algorithm can not handle this type of redistribution.

To show the validation of our new optimizations we presented – communication scheduling and aggregation, we timed our unoptimized algorithm and a optimized version for the redistribution (BLOCK,*) to (*,BLOCK) with array size of 1024×1024. Table 3 shows the results of our study. From these tables, we can make the following observations:

- Our algorithm can work better than the naive and one-dimensional algorithms for the "the shape of the processor grid retaining redistribution".
- For the "the shape of the processor grid changing redistribution", our algorithm can work well for all of the redistributions on arbitrary processor sets. It performs better than the naive algorithm, and as the array size increases, the performance improvement becomes more appreciable.
- The optimizations we proposed provide significant performance benefits.
- Contrasting with the estimation formulas that we proposed in Section 4.4, it seems that all of the experimental results is approximately consistent with performance estimations.

Table 3. Results for communication optimization on CP-PACS(time in second).

Algorithm	Number of processors					
	8	16	32	64	128	256
no optimization	1.82	1.61	1.41	1.07	0.89	1.09
optimized	0.39	0.28	0.22	0.18	0.13	0.11

6. Conclusions

Array redistribution can provide higher performance than purely static distributions for programs. But most of current algorithms are one-dimension based, therefore they cannot efficiently handle some multi-dimensional array redistribution. In this paper, we proposed a multi-dimensional array redistribution al-

Table 2. Results for "the shape of the processor grid changing redistribution" on CP-PACS(time in second).

Redistributions	Array Size	P_s	P_t	naive	G_{algo}
(CYCLIC,BLOCK) to (BLOCK,CYCLIC)	300×300	3×3	5×2	0.46	0.11
		6×12	10×5	0.27	0.14
		15×10	5×6	0.32	0.22
	600×600	3×3	5×2	2.21	0.60
		6×12	10×5	1.47	0.48
		15×10	5×6	1.21	0.54
(BLOCK,CYCLIC) to (BLOCK,*)	300×300	4×5	10×1	0.24	0.03
		10×6	120×1	0.40	0.13
		10×20	200×1	0.27	0.21
	600×600	4×5	10×1	1.09	0.11
		10×6	120×1	1.01	0.17
		10×20	200×1	0.89	0.28
(BLOCK,*) to (*,BLOCK)	300×300	20×1	1×20	0.19	0.09
		100×1	1×50	0.21	0.17
		150×1	1×150	0.17	0.14
	600×600	20×1	1×20	1.89	0.63
		100×1	1×50	0.98	0.43
		150×1	1×150	0.85	0.36

gorithm. It can deal with any redistribution between arbitrary source and destination processor sets and between arbitrary source and destination distribution schemes. The generated SPMD code has been experimented on CP-PACS and the results are shown. In the future, we would like to extend our algorithm to irregular redistribution problem, such as redistribution between linear distribution schemes described in [4].

References

[1] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proc. of the 1994 Int'l Conf. on Parallel Archs. and Compilation Techniques*, Montreal, Canada, Aug. 1994.

[2] F. Coelho and C. Ancourt. Optimal compilation of hpf remappings. *Journal of Parallel and Distributed Computing*, 38:229–236, 1996.

[3] J. Dongarra, L. Prylli, C. Randriamaro, and B. Tourancheau. Array redistribution in scalapack using pvm. In *Proceedings of Euro-PVM'95*, Lyon, France, Sept. 1995.

[4] M. Guo, Y. Yamashita, and I. Nakata. An efficient data distribution technique for distributed memory parallel computers. In *Joint Symposium on Parallel Processing 1997*, Kobe, Japan, May 1997.

[5] HPF Forum, Rice University, Houston, Texas. *HPF-2 Scope of Activities and Motivating Applications*, version 0.8 edition, Nov. 1994.

[6] HPF Forum, Rice University, Houston, Texas. *High performance Fortran Language Specification*, version 2.0 edition, Nov. 1996.

[7] E. T. Kalns and L. M. Ni. Processor mapping techniques toward efficient data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1234–1247, 1995.

[8] K. Kennedy and U. Kremer. Automatic data layout for high performance fortran. In *Proceedings of Supercomputing'95*, San Diego, CA, Dec. 1995.

[9] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proc. International Conference on Supercomputing*, Barcelona, July 1995.

[10] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distribute-memory machines in the d programming environment. In C. W. Kessler, editor, *Automatic Parallelization–New Approaches to Code Generation, Data Distribution, and Performance Prediction*, Vieweg Advanced Studies in Computer Science, pages 136–147. Verlag Vieweg, Wiesbaden, Germany, 1993.

[11] K. Nakazawa, H. Nakamura, and T. Boku. The architecture of massively parallel processor cp-pacs. *Journal of Information Processing Society of Japan*, 37(1):18–28, 1996. (in Japanese).

[12] S. Ramaswamy, B. Simons, and P. Banerjee. Optimizations for efficient array redistribution on distributed memery multicomputers. *Journal of Parallel and Distributed Computing*, 38:217–228, 1996.

[13] J. R. S.D. Kaushik, C.-H. Huang and P. Sadayappan. Multi-phase redistribution: A communication-efficient

approach to array redistribution. Technical report, The Ohio Sate University, 1995.

[14] J. Stichnoth, D. O'Hallaron, and T. Gross. Grnerating communication for array statements: Design, implementation, and evaluation. *J. Parallel and Distributed Computing*, pages 150–159, 1994.

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

[16] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587–593, 1996.

[17] A. Thirumalai and J. Ramanujam. Hpf array statements: Communication generation and optimization. In *Proc. 3rd Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, Troy, NY, May 1995.

[18] A. Wakatani and M. Wolfe. A new approach to array redistribution: Strip mining redistribution. In *Proceedings of Parallel Architectures and Languages Europe*, July 1994.