The Journal of Supercomputing, 20, 243–265, 2001
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Framework for Efficient Data Redistribution on Distributed Memory Multicomputers

MINYI GUO e-mail: minyi@u-aizu.ac.jp

Department of Computer Software, The University of Aizu, Aizu-Wakamatsu City, Fukushima, Japan

IKUO NAKATA e-mail: nakata@k.hosei.ac.jp

Faculty of Computer and Information Sciences, Hosei University, Tokyo, Japan

Abstract. Array redistribution is required often in programs on distributed memory parallel computers. It is essential to use efficient algorithms for redistribution; otherwise the performance of the programs will degrade considerably. The redistribution overheads consist of two parts: index computation and inter-processor communication. In this paper, by using a notation for the local data description called an LDD, we propose a framework to optimize the array redistribution algorithm both in index computation and inter-processor communication. That is, our work makes an effort to optimize not only the computation cost but also communication cost for array redistribution algorithms. We present an efficient index computation method and generate a schedule that minimizes the number of communication steps and eliminates node contention in each communication step. Some experiments show the efficiency and flexibility of our techniques.

Keywords: parallelizing compilers, HPF, array redistribution, communication scheduling, distributed memory machines

1. Introduction

Array redistribution problem has recently received considerable attention. This interest is motivated largely by the HPF [8] programming style, in which scientific applications are decomposed into phases. At each phase, there is an optimal distribution of arrays onto the processor grid. Many applications and kernels, such as (ADI) Alternating Direction Integration, 2D (FFT) Fast Fourier Transform and signal processing, require different array distributions at different computation phases for efficient execution on distributed memory machines. Because the optimal distribution changes from phase to phase, array redistribution turns out to be a critical operation.

Basically, the redistribution algorithms consist of two parts: index computation and inter-processor communication. The index computation overheads are incurred when each processor computes indices of array elements that are to be communicated with the other processors, and designates the destination processor numbers of such array elements. The communication overheads are incurred when the processors exchange array elements. These include software start-up overheads for invocation of the send/receive system calls, transmission costs for sending data over the interconnection network, and overheads due to the node

contention. Our previous work attempted to reduce the index computation overheads for two-dimensional redistribution [5, 6]. In this paper, we will extend the algorithms to any multi-dimensional redistribution problem. On the other hand, without proper communication scheduling in a redistribution algorithm, the communication contention may occur, which increases the communication waiting time. In order to solve this problem, in this paper, we also propose a technique to schedule the communication so that it becomes contention-free. Our approach initially generates a communication table to represent the communication relations among sending nodes and receiving nodes. According to the communication table, we then generate another table named communication scheduling table. Each column of communication scheduling table is a permutation of receiving node numbers in each communication step. Thus the communications in our redistribution algorithm are contention-free.

The remainder of the paper is organized as follows. Section 2 discusses some important related work in this area. In Section 3, we discuss our local data description notation called LDD. Section 4 proposes the generalized redistribution algorithm and Section 5 is the communication scheduling algorithm, both based on the LDDs. Section 6 gives some experimental results for our algorithm and compares the efficiency and flexibility with other work. Finally, we conclude this paper and discuss possible future extensions to our work in Section 7.

2. Related work

Any technique that handles HPF array assignments [11, 14, 26, 29] 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, it is not clear that any of the currently implemented compilers have efficient techniques to handle all the regular distributions possible for A and B. Considering only the problem of data redistribution allows us to use many optimizations that may not be applicable in a general compiler framework. Another major obstacle in trying to use current compiler techniques is that they cannot handle the given statement if A and B are distributed on (possibly overlapping) subsets of processors.

Thakur et al. [27, 28] considered a redistribution library for changing the distribution of arrays on a given set of processors. The methods proposed treat possible source-target distributions in a special pairwise manner—redistribution between cyclic(x) and cyclic($k * x$) in one dimension. This prevents them from handling very general source-target distributions in an efficient manner. Further, they proposed an expensive solution for multi-dimensional redistributions. They consider such redistributions to be composed of a series of one-dimensional redistributions, which can lead to a considerable amount of unnecessary communication.

A redistribution technique based on the descriptors called pitfalls has been devised in [23]. It can treat arbitrary source and target processor sets. However, the work has no capability of solving more complex redistribution applications, such as shape changing redistribution—that is, 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. In such a case, an expensive run-time resolution approach is used. Further, the approach used for multi-dimensional array redistribution involves a series of one-dimensional redistributions, which can be costly.

Chung et al. presented a basic-cycle calculation technique to efficiently perform cyclic(x) to cyclic(y) redistribution [2]. Their main idea is to develop closed forms for computing source/destination processors of some specific array elements in a basic-cycle, which is defined as $lcm(x, y)/gcd(x, y)$. These closed forms are then used to efficiently determine the communication sets of a basic-cycle. Then they presented an extended technique called *generalized basic-cycle calculation* method to perform a cyclic(x) over P processors to cyclic(y) over Q processors redistribution [9]. In this method, a generalized basic-cycle is defined as $lcm(xP, yQ)/(gcd(x, y) \times P)$ in the source distribution and $lcm(xP, yQ)/(gcd(x, y) \times Q)$ in the destination distribution. From the source/destination processor/data sets of array elements in the first generalized basic-cycle, a packing/unpacking pattern tables to minimize the data-movement operations was constructed.

Many researches have concentrated mainly on the efficient index computation for generating the communication messages to be exchanged by the processors involved in the redistribution [1, 15, 16, 20, 21]. However, the question of how to efficiently schedule the messages has received little attention. The following are some works concerned with the communication optimization in redistribution.

Lim et al. [17, 18] developed the algorithms that redistribute an array from one block-cyclic scheme to another, where the source and target schemes have the special relation. Their framework is based on a generalized circulant matrix formalism. Through the transform of the rows/columns of the matrix, data communication is performed in a conflict-free manner using direct, indirect, and hybrid algorithms. In a direct algorithm, a data block is transferred directly to its destination processor. In an indirect algorithm, data blocks are moved from source to destination processors through intermediate relay processors. The relay processors combine data blocks with the same destination. A hybrid algorithm is a combination of both direct and indirect algorithms. In their subsequent work [22], Park et al. proposed an extended algorithm that reduces the overall time for communication by considering the data transfer, communication schedule, and index computation costs.

Kalns and Ni [13] presented 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 [10]. Kaushik et al. used the tensor product representation of data distributions and the network contention model by expressing the communication as a sequence of permutations, each of which can be executed in a fixed number of contention-free steps. They developed a multi-phase strategy which performs 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.

Desprez et al. [4] proposed an algorithm for the scheduling of those messages: how to organize the message exchanges into "structured" communication steps that minimize contention. They built a scheduling for moving from a cyclic(r) distribution on a P-processor grid to a cyclic(s) distribution on a Q-processor grid for

a one-dimensional redistribution, where the values of $P, Q, r,$ and s are arbitrary. They considered the size of the communication messages as a term of scheduling. However, they did not give a description of how to get the information about the sources and destinations of communication and the message sizes.

Most works mentioned above have achieved efficiency in one aspect: either index computation optimization or communication optimization. However, they have not developed techniques which reduce overheads both in index computation and inter-node communication. Our aim in this paper is to extend the concepts and results presented in our previous work [5–7], in order to provide a framework for all efficient data redistribution—both in index computation and communication scheduling.

3. Redistribution based on local data descriptor

3.1. The redistribution problem

Motivated by the need for the redistribution described in Section 1, we define the regular redistribution problem [27]. A redistribution $\mathcal{R}$ is the set of routines that change the distribution schemes such that, given a 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.

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, 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.

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 resulting in low efficiency.

In the following discussion, we assume that the global array size is denoted as $G_1 \times \cdots \times G_\delta$ and the local array size is denoted as $L_1 \times \cdots \times L_\delta$, where δ is the array dimension. All arrays are indexed starting from 1 while processors are numbered starting from 0. The distribution schemes allowed in a dimension are $BLOCK$, $CYCLIC$, $CYCLIC(b)$ and All (* in HPF). p processors compose a δ-dimensional processor grid $p_1 \times \cdots \times p_\delta$ where $p_i (1 \leq i \leq \delta)$ is the number of processors of the i-th dimension. *myid* is the logical number of the processor executing the program. Because of the limitation of the paper length, we do not give the proofs for the lemmas and theorems in the following sections. Details on proof can be found in [7].

3.2. *Local data descriptor (LDD) for one-dimensional arrays*

Roughly, to perform a redistribution, for any source-target processor pair, one has to look at the set of elements owned by the source processor p_s before redistribution (based on the source distribution scheme $\mathscr{D}_s$) and the set of elements owned by the target processor p_t after redistribution (based on the target distribution $\mathscr{D}_t$). The intersection of these sets is the data that needs to be transferred between the source-target processors. The LDD representation is particularly useful to determine which pair of source-target processors need to communicate.

We first develop the LDD representation for regular distributions of one-dimensional arrays, then extend it to a multi-dimensional case.

Definition 1 A 4-tuple $d = (o, b, s, n)$ is called a LDD (Local Data Descriptor) which describes a set of the global array index for a particular processor. Intuitively, d represents a finite set of equally spaced, equally sized blocks of elements, where

— o is the starting index of the global array elements distributed onto the processor;
— b is the length of the block;
— s is the stride between two consecutive blocks; and
— n is the number of blocks distributed on to the processor.

Consider a one-dimensional array A of size G. Using the notion of LDD, it is possible to represent the set of elements of A owned by a processor under any regular distribution. An LDD $d = (o, b, s, n)$ corresponds a set of the global array index defined as follows:

$$S[d] = \{i \mid o + s * u \leq i < o + b + s * u, 0 \leq u < n\}. \tag{1}$$

The j-th $(0 \leq j < n)$ block of LDD d (denoted by b_j) is given by

$$\begin{aligned} b_j &= (o + s * j, o + s * j + (b - 1)) \\ &= \{i | o + s * j \leq i < o + s * j + (b - 1)\}. \end{aligned} \tag{2}$$

From the above definitions, we can conclude that any block-cyclic distribution scheme for an array can be expressed by using LDDs. For example, let p be the number of processors and the global array size G be the multiple of p, then BLOCK distribution schemes can be expressed by LDDs as $d_k = ((G/p) * k + 1, G/p, 1, 1)$, CYCLIC(4) can be expressed as $d_k = (4 * k + 1, 4, 4 * p, |\frac{G}{4*p}|)$, and CYCLIC is expressed as $d_k = (k + 1, 1, p, G/p)$, for k-th processor.

3.3. *The intersection of LDDs*

Let $d_1 = (o_1, b_1, s_1, n_1)$ and $d_2 = (o_2, b_2, s_2, n_2)$ be two LDDs, and their corresponding array index set be $S[d_1]$ and $S[d_2]$ (namely LDD set) respectively.

The intersection and union of $S[d_1]$ and $S[d_2]$ are as follows:

$$\begin{aligned} S[d_1] \cap S[d_2] = \{i \mid & \max(o_1 + s_1 * u_1, o_2 + s_2 * u_2) \\ & \le i < \min((o_1 + s_1 * u_1) + b_1, (o_2 + s_2 * u_2) + b_2), \\ & 0 \le u_1 < n_1, 0 \le u_2 < n_2\}. \end{aligned} \tag{3}$$

$$\begin{aligned} S[d_1] \cup S[d_2] = \{i \mid & (o_1 + s_1 * u_1 \le i < o_1 + b_1 + s_1 * u_1) \\ & \vee (o_2 + s_2 * u_2 \le i < o_2 + b_2 + s_2 * u_2), \\ & 0 \le u_1 < n_1, 0 \le u_2 < n_2\}. \end{aligned} \tag{4}$$

The examples of the LDDs and their intersections and unions shown in Fig. 1 may help to understand the LDD notion easily.

Lemma 1 *Let* $d_1 = (o_1, b_1, s_1, n_1)$ *and* $d_2 = (o_2, b_2, s_2, n_2)$.

$$S[d_1] \cap S[d_2] = \varnothing \Longleftrightarrow \forall u_1, u_2 (0 \le u_1 < n_1 \wedge 0 \le u_2 < n_2),$$
$$\max(o_1 + s_1 * u_1, o_2 + s_2 * u_2) \ge \min((o_1 + s_1 * u_1) + b_1, (o_2 + s_2 * u_2) + b_2).$$

Lemma 2 *If two blocks which belong to two LDDs* $d_1 = (o_1, b_1, s_1, n_1)$ *and* $d_2 = (o_2, b_2, s_2, n_2)$, *respectively*, *have some common elements*; *that is*, *there exist* u_1^* *and* u_2^* *such that*

$$\max(o_1 + s_1 * u_1^*, o_2 + s_2 * u_2^*) < \min(o_1 + b_1 + s_1 * u_1^*, o_2 + b_2 + s_2 * u_2^*),$$

then all the successive block pairs of stride $s = lcm(s_1, s_2)$ *also have the common elements*.

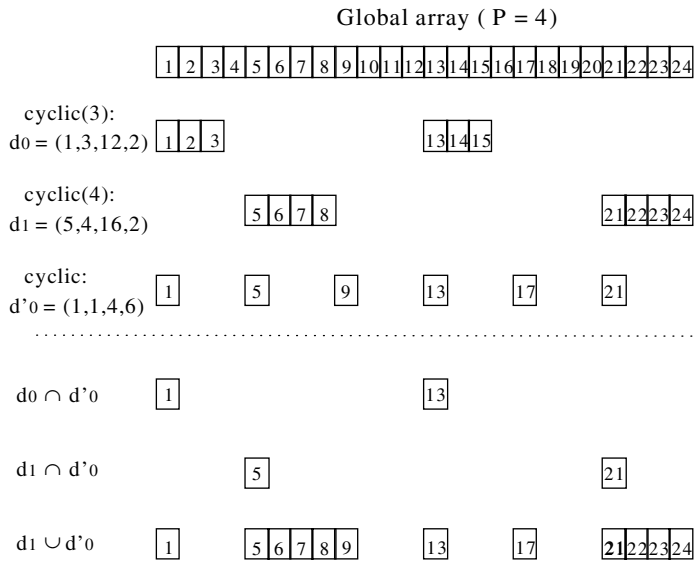

Figure 1. The example of the LDDs and their intersection and union on 4 processors.

Proof: Refer to [7]. □

From the above lemmas we can conclude that the intersection of two LDD sets is the unions of some LDD sets whose stride is s. We have the following theorem.

Theorem 1 *$S[d_1] \cap S[d_2] = S[d^{(1)}] \cup \cdots \cup S[d^{(m)}]$ where $d^{(k)}(1 \le k \le m)$ is a LDD if $\exists(u_1^{(k)}, u_2^{(k)})$ such that*

$$\max(o_1 + s_1 * u_1^{(k)}, o_2 + s_2 * u_2^{(k)}) < \min(o_1 + b_1 + s_1 * u_1^{(k)}, o_2 + b_2 + s_2 * u_2^{(k)}),$$
$$(0 \le u_1^{(k)} < m_1, 0 \le u_2^{(k)} < m_2).$$

The elements of $d^{(k)}$ are defined as

- $o^{(k)} = \max(o_1 + s_1 * u_1^{(k)}, o_2 + s_2 * u_2^{(k)})$
- $b^{(k)} = \min(o_1 + b_1 + s_1 * u_1^{(k)}, o_2 + b_2 + s_2 * u_2^{(k)}) - \max(o_1 + s_1 * u_1^{(k)}, o_2 + s_2 * u_2^{(k)})$
- $s^{(k)} = s = lcm(s_1, s_2)$
- $n^{(k)} = \min(e_1 - e_1^{(k)}, e_2 - e_2^{(k)}) \operatorname{div} s + 1 = \min(s_1 * (n_1 - u_1^{(k)} - 1), s_2 * (n_2 - u_2^{(k)} - 1)) \operatorname{div} s + 1$

where e_1 and e_2 are the last indices of d_1 and d_2:

$$e_1 = o_1 + b_1 - 1 + s_1 * (n_1 - 1), \qquad e_2 = o_2 + b_2 - 1 + s_2 * (n_2 - 1)$$

$e_1^{(k)}$ and $e_2^{(k)}$ are the last indices of the blocks $u_1^{(k)}$ in d_1 and $u_2^{(k)}$ in d_2:

$$e_1^{(k)} = o_1 + b_1 - 1 + s_1 * u_1^{(k)}, \qquad e_2^{(k)} = o_2 + b_2 - 1 + s_2 * u^{(k)}$$

Proof: Refer to [7]. □

For the sake of simplicity, in the following discussion we will use notations $d_1 \cap d_2$ and $d_1 \cup d_2$ to express the same operations onto their corresponding sets, $S[d_1] \cap S[d_2]$ and $S[d_1] \cup S[d_2]$, respectively. For example, the above theorem can be described as

$$d_1 \cap d_2 = d^{(1)} \cup \cdots \cup d^{(m)}. \tag{5}$$

3.4. Local data descriptor for multi-dimensional arrays

Until now we have only considered one-dimensional cases of our LDD representations. Extending these to the multi-dimensional case is trivial and can be done by simply looking at the representations for each dimension and computing the intersections on them independently. Accordingly we can define the Local Data Descriptor of the multi-dimensional case as a 4-tuple of vectors, where each vector is composed of the corresponding elements of one-dimensional LDDs.

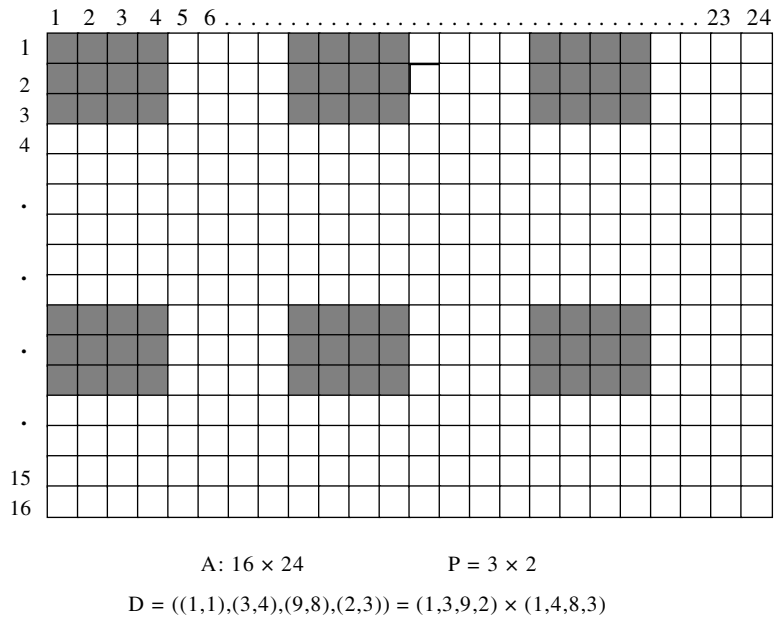

Figure 2. The LDD for multi-dimensional array.

For a multi-dimensional LDD D, its index set $S[D]$ is defined as the Cartesian product of the index sets of each of its dimensional LDD d_i (Fig. 2):

Definition 2 A multi-dimensional LDD is defined as

$$D = (\vec{O}, \vec{B}, \vec{S}, \vec{N}),$$

where $\vec{O} = \langle o_1, \ldots, o_\delta \rangle$, $\vec{B} = \langle b_1, \ldots, b_\delta \rangle$, $\vec{S} = \langle s_1, \ldots, s_\delta \rangle$, $\vec{N} = \langle n_1, \ldots, n_\delta \rangle$, and δ is the number of dimensions. For $1 \leq i \leq \delta$, $d_i = (o_i, b_i, s_i, n_i)$ is called the LDD of i-th dimension.

$$S[D] = S[d_1] \times S[d_2] \times \cdots \times S[d_\delta].$$

We will abbreviate the above formula as

$$D = d_1 \times d_2 \times \cdots \times d_\delta. \tag{6}$$

Also, the intersection of two multi-dimensional LDDs can be computed by a dimension-by-dimension intersection. We have the following corollaries.

Corollary 1

$$\begin{aligned} S[D_i] \cap S[D_j] &= (S[d_{i_1}] \times \cdots \times S[d_{i_\delta}]) \cap (S[d_{j_1}] \times \cdots \times S[d_{j_\delta}]) \\ &= (S[d_{i_1}] \cap S[d_{j_1}]) \times \cdots \times (S[d_{i_\delta}] \cap S[d_{j_\delta}]). \end{aligned}$$

That is, the intersection of two LDDs is the Cartesian product of the intersection of each dimensional LDDs. We can abbreviate this result as

$$\begin{aligned} D_i \cap D_j &= (d_{i_1} \times \cdots \times d_{i_\delta}) \cap (d_{j_1} \times \cdots \times d_{j_\delta}). \\ &= (d_{i_1} \cap d_{j_1}) \times \cdots \times (d_{i_\delta} \cap d_{j_\delta}). \end{aligned}$$

Corollary 2

$$D_i \cap D_j \neq \varnothing \Longleftrightarrow (d_{i_1} \cap d_{j_1} \neq \varnothing) \wedge \cdots \wedge (d_{i_\delta} \cap d_{j_\delta} \neq \varnothing)$$

4. Redistribution algorithm based on LDD

We assume that, before the redistribution, a global array A is distributed into the local arrays according to the source distribution scheme $\mathscr{D}_s$ and the source processor grid $\mathscr{P}_s$. In other words, the global array A is composed of the local arrays which are owned by each processor. A can also be represented as a multi-dimensional LDD

$$D = (\vec{O}, \vec{B}, \vec{S}, \vec{N}),$$

where $\vec{O} = (1, 1, \ldots, 1)$, $\vec{B} = (1, 1, \ldots, 1)$, $\vec{S} = (1, 1, \ldots, 1)$, $\vec{N} = (G_1, \ldots, G_\delta)$. Then it can be denoted as

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

where $D_k, k \in \{0, \ldots, p-1\}$ is the local LDD of each processor under $\mathscr{D}_s$, and $D_i \cup D_j$ is the abbreviated form of $S[D_i] \cup S[D_j]$. On the other hand, after the redistribution, the global array A can also be composed of the local arrays owned by each processor, which A is distributed into the local arrays according to the destination distribution scheme $\mathscr{D}_t$ and the destination processor grid $\mathscr{P}_t$. It also can be denoted as

$$D = \widetilde{D}_0 \cup \widetilde{D}_1 \cup \cdots \cup \widetilde{D}_{\tilde{p}-1},$$

where $\widetilde{D}_k, k \in \{0, \ldots, \tilde{p}-1\}$ is the local LDD of each processor under the target distribution. Because

$$\begin{aligned} D = D \cap D &= (D_0 \cup \cdots \cup D_{p-1}) \cap (\widetilde{D}_0 \cup \cdots \cup \widetilde{D}_{\tilde{p}-1}) \\ &= \bigcup_{k=0}^{p-1} D_k \cap (\widetilde{D}_0 \cup \cdots \cup \widetilde{D}_{\tilde{p}-1}). \end{aligned}$$

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

Similarly, we can also get the following formula,

$$D = \bigcup_{k=0}^{\tilde{p}-1} \widetilde{D}_k \cap (D_0 \cup \cdots \cup D_{p-1}).$$

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

Theorem 2 *Let D_i be the LDD of a source processor P_i under the source distribution scheme $\mathcal{D}_s$, and $\widetilde{D}_j$ be the LDD of a target processor $\widetilde{P}_j$ under the target distribution scheme $\mathcal{D}_t$. In a redistribution from $(\mathcal{P}_s, \mathcal{D}_s)$ to $(\mathcal{P}_t, \mathcal{D}_t)$, where $\mathcal{P}_s$ and $\mathcal{P}_t$ are the source and target processor set respectively, the data that each processor P_i $(P_i \in \mathcal{P}_s)$ should communicate with target processors $\widetilde{P}_j, (\widetilde{P}_j \in \mathcal{P}_t)$, are indicated by the intersection of D_i and $\widetilde{D}_j$.*

Figure 3 shows an example for the intersection of a global array A's LDDs before and after redistribution, where $\mathcal{P}_s = 4 \times 1, \mathcal{D}_s = (block, *)$, and $\mathcal{P}_t = 2 \times 2, \mathcal{D}_t = (block, block)$.

The redistribution routine includes two parts, the sending routine and receiving routine. The sending routine is executed by the source processors. It analyzes the communication requirements for each array and packs the message into a contiguous buffer for each destination processor that needs data from the calling source processor; it then sends message to the corresponding destination processor. The receiving routine essentially does the reverse.

The algorithm frameworks of the sending and receiving routines are shown in Figures 4 and 5, respectively.

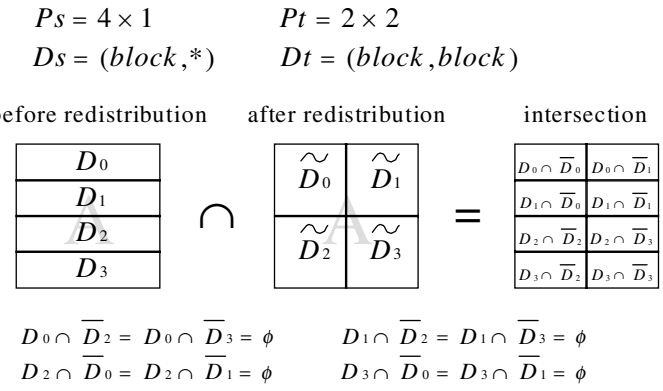

Figure 3. Intersection of a global array A's LDDs before and after redistribution on 4 processors. $\mathcal{P}_s = 4 \times 1, \mathcal{D}_s = (block, *)$, and $\mathcal{P}_t = 2 \times 2, \mathcal{D}_t = (block, block)$.

Algorithm 1 Send routine

```
if myid ∈ P_s then
  get my source LDD D' = (O', B', S', N')
  /* D' is determined by myid, the source distribution scheme */
  for P̃_k ∈ P_t, k = 0, 1, ..., p̃ − 1
    get each target LDD D''_0, D''_1, ..., D''_{p̃−1};
    /* D''_k is determined by P_k and the destination distribution scheme */
    compute the intersection of D' ∩ D''_k
    D_1 = D' ∩ D''_0, D_2 = D' ∩ D''_1, ...,
    if myid ≠ P̃_k then
      if D_k ≠ φ then
        /* D_k is global index set, it should be converted to
        local index under source distribution scheme */
        convert global (o_g^1, ..., o_g^δ) to local (o_l^1, ..., o_l^δ);
        compute the local stride m_1^1, ..., m_1^δ of
        source local array As for each dimension;
        pack the local D_k into buffer;
        send(buffer, P̃_k);
      endif
    endif
  endfor
endif
```

Figure 4. The redistribution algorithm of the send routine.

5. Communication scheduling

We only optimize the cost of index computation in the algorithms shown in Figures 4 and 5, without taking into consideration the reduction of the communication cost. Figure 6(a) shows the sequence of events that can occur during a redistribution involving four source and four destination processors using the above redistribution algorithm. We observed that each destination processor receives all of its messages simultaneously; this may lead to communication contention. The communication contention problem can be described as follows: For a set of processors, since the receiving processor typically can receive messages from only one processor at once, if there are more than two of sending processors they may have to wait for other processors to complete their communication, in this case we say that the communication (or node) contention occurred. The communication contention has a deteriorating effect on the total time required for communication [7].

Using the communication scheduling we can avoid the communication contention in redistribution operations. For instance, we can use simple schemes to get rid of the contention shown in Fig. 6(a); the effect of such schemes is illustrated in Fig. 6(b).

Algorithm 2 Receive routine

```
if myrid ∈ P_t then
  allocate destination local array At;
  get my destination LDD D'' = (O'', B'', S'', N'');
  /* D'' is determined by myid, destination distribution scheme */
  for P_k ∈ P_s, k = 0, 1, ..., p − 1
    get each source LDD D'_0, D'_1, ..., D'_{p−1};
    /* D''_k is determined by P_k and source distribution scheme */
    compute the intersection of D' ∩ D''_k
    D_0 = D' ∩ D''_0, D_1 = D' ∩ D''_1, ...,
    if myid ≠ P_k then
      if D_k ≠ φ then
        receive(buffer, P_k);
        compute the local stride m_2^1, ..., m_2^δ of At:
        unpack the buffer into At,
      endif
    else
      /* memory copy */
      convert the global index (i_1, ..., i_δ) to the local index
      (i_s^1, ..., i_s^δ) under the source distribution D_s,
      and (i_t^1, ..., i_t^δ) under the destination distribution D_t;
      At[i_t, j_t] = As[i_s, j_s];
    endif
  endfor
endif
if myid ∈ P_s then
  free(As);
endif
```

Figure 5. The redistribution algorithm of the receive routine.

5.1. One dimensional scheduling algorithm

The avoidance of contention for all-to-all communication is so trivial that we focus on the contention-free communication scheduling in the case of all-to-many communication in the following discussions. For one dimension, we assume that the redistribution is processed from a cyclic(b) distribution on a p-processor grid to a cyclic(b') distribution on a $\tilde{p}$-processor grid and $b = \beta * b'$(the case of $b' = \beta * b$ is similar).

We construct a communication matrix (table) *COM* for the redistribution $(\mathscr{D}_s, \mathscr{P}_s)$ to $(\mathscr{D}_t, \mathscr{P}_t)$. A "1" in the (i, j) entry represents the fact that a processor P_i needs to communicate to a processor P_j. That is, $COM[i, j] = 1$ if and only if a processor P_i sends data to a processor P_j. According to the usage of LDDs, let d_i and d_j be

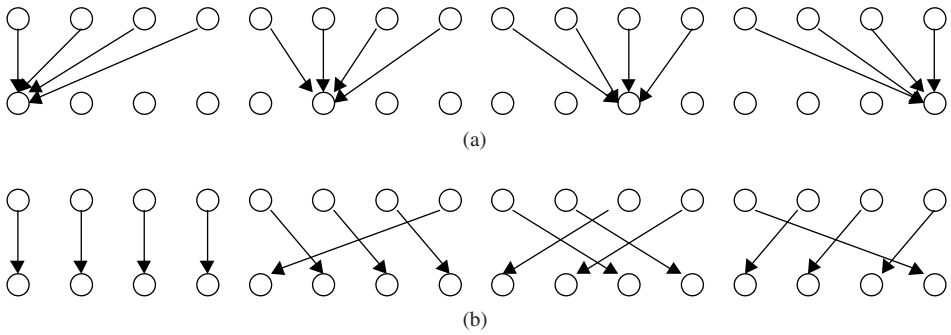

Figure 6. Communication contention for array redistribution.

the LDDs of processors P_i and P_j, respectively, then

$$COM[i, j] = \begin{cases} 1, & d_i \cap d_j \neq \varnothing \\ 0, & d_i \cap d_j = \varnothing. \end{cases} \tag{7}$$

Our goal is to generate an algorithm that derives another table from *COM*, called the communication scheduling table $CS[i, k]$, where $0 \leq i < p$, $0 \leq k < \mathcal{K}$ ($\mathcal{K}$ is the number of communication steps), and if $COM[i, j] = 1$, there exists a k, $0 \leq k < \mathcal{K}$ such that $CS[i, k] = j$ and each column of *CS* is a (partial) permutation of processors $0, 1, \ldots, \tilde{p} - 1$. Figure 7 shows the examples of the communication table and its corresponding communication scheduling table.

Theorem 3 *In a redistribution operation, if a sending processor P_i needs to send (needs not send) a message to a receiving processor P_j, then for another sending processor $P_{i'}$, $0 \leq i' < p$, there certainly exists a receiving processor $P_{j'}$ such that $P_{i'}$ needs to send (not send) a message to $P_{j'}$, and $j' = (j + (i' - i) * b/b') \bmod \tilde{p}$. Using the*

	0	1	2	3	4
0	1	1	0	1	0
1	1	0	1	0	1
2	0	1	0	1	1
3	1	0	1	1	0
4	0	1	1	0	1

(a) Communication table COM

	0	1	2
0	3	0	1
1	2	4	0
2	1	3	4
3	0	2	3
4	4	1	2

(b) Communication scheduling table CS

Figure 7. Examples of the communication table and the communication scheduling table.

notion of communication table, this means

$$COM[i, j] = 0\ (or = 1) \Longleftrightarrow COM[i', j'] = 0\ (or = 1),$$

and $j' = (j + (i' - i) * b/b') \bmod \tilde{p}$.

Proof: Refer to [7]. □

Definition 3 According to Theorem 2, two entries (i, j) and (i', j') of *COM* are called the symmetrical if and only if

$$j' = \left(j + (i' - i) * \frac{b}{b'}\right) \bmod \tilde{p}.$$

Figure 8 shows two communication tables with $p = \tilde{p} = 6$, $\mathcal{K} = 4$ and $p = \tilde{p} = 5$, $\mathcal{K} = 3$, respectively. The entries enclosed in a circle are symmetrical entries.

Corollary 3 *For a sending processor* P_i, *if there exist* j_1 *and* j_2, *such that* $COM[i, j_1] = 0$ *and* $COM[i, j_2] = 0$ *and* $(j_2 - j_1) \bmod \tilde{p} = d$, *then for each other sending node* $P_{i'}$, *there certainly exist* j_1' *and* j_2', *such that* $COM[i', j_1'] = 0$ *and* $COM[i', j_2'] = 0$ *and* $(j_2' - j_1') \bmod \tilde{p} = d$, *where d is a constant, namely distance*.

Definition 4 A scheduled sending vector of a processor P_i is a sequence of destination processors:

$$SV_i = (P_{j_0}, P_{j_1}, \ldots, P_{j_{\mathcal{K}-1}}),$$

where $COM[i, j_k] = 1$, $0 \le k < \mathcal{K}$, $\mathcal{K}$ is the number of the communication steps, $j_k = (j_0 + k) \bmod \mathcal{K}$[1], and $j_u \prec j_v$[2] when $u < v$. $SV_i^k (= P_{j_k})$ represents the k-th entry in vector SV_i. SV_i^0 is called the *start element* of SV_i.

cyclic(8) → cyclic(2) P = 6

	0	1	2	3	4	5
0	①	1	1	1	0	0
1	1	1	0	0	①	1
2	0	0	①	1	1	1
3	①	1	1	1	0	0
4	1	1	0	0	①	1
5	0	0	①	1	1	1

cyclic(6) → cyclic(2) P = 5

	0	1	2	3	4
0	1	1	1	⓪	0
1	1	⓪	0	1	1
2	0	1	1	1	⓪
3	1	1	⓪	0	1
4	⓪	0	1	1	1

○ express the corresponding entries.

Figure 8. Examples of communication tables and the symmetrical entries for one dimensional redistribution. (a) *cyclic*(8) to *cyclic*(2), $P = 6$. (b) *cyclic*(6) to *cyclic*(2), $P = 5$.

According to the above definition, the communication scheduling table CS is composed of some SV_i. SV_i is the i-th row of CS. From Corollary 3, if the start elements of two sending vectors are different, then all the elements at the same entry are different for these two sending vectors. Therefore in the following discussions, we only focus on the algorithm that generates the start elements.

From Theorem 3, if $COM[i_0, j_0] = 1$ $(or = 0)$ for a sending node P_{i_0}, then there certainly exists corresponding j_k for each other sending node P_{i_k} such that $COM[i_k, j_k] = 1 (or = 0), 1 \leq k < P$. These j_k form a group $(j_0, \ldots, j_{P-1})$, called the *relative group RG*.

Definition 5 Suppose J^* is a "1" entry in the row i of COM for a sending node P_i, the k-th "1" entry J following J^* in the row i of COM is defined as that $COM[i, J] = 1, J^* \prec J$, and there exist $J^{(1)}, \ldots, J^{(k-1)}$ such that $J^* \prec J^{(1)} \prec \cdots \prec J^{(k-1)} \prec J, COM[i, J^{(l)}] = 1, 1 \leq l \leq k-1$.

Algorithm 3 Because the sending vector SV can be determined according to SV^0, we only need to find the start element SV_i^0 for each sending node P_i according to the following steps:

1. Find out a "1" entry as the first "1" entry J_0^* for sending node P_0 where $COM[0, J_0^*] = 1$ and its relative group $RG = (J_0^*, \ldots, J_{p-1}^*)$. (Take Fig. 8 as an example. $RG = (0, 4, 2, 0, 4, 2)$ and $RG' = (0, 3, 1, 4, 2)$ for the communication tables shown in Fig. 8(a) and Fig. 8(b), respectively.)
2. If some J^*s in RG are equal to each other, i.e., $J_{i_1}^* = J_{i_2}^* = \cdots = J_{i_n}^*, (J_{i_u}^* \in RG, 0 \leq i_u < P)$, then put the sending node $P_{i_1}, \ldots, P_{i_n}$ into a sub-group SN. Thus P sending nodes can be divided into m sub-groups $SN_0, \ldots, SN_{m-1}$, each sub-group SN_i has the same number of elements n, where $p = m * n$.
   (For Fig. 8(a), $SN_0 = (P_0, P_3), SN_1 = (P_1, P_4), SN_2 = (P_2, P_5)$.)
3. If the number of elements of all the sub-groups is one, that is, there are total p sub-groups, then the start element of each row i is the first "1" entry J_i^*.
   (For Fig. 8(b), $SN_0 = (P_0), SN_1 = (P_1), SN_2 = (P_2), SN_3 = (P_3), SN_4 = (P_4)$. Hence the start elements are $(0, 3, 1, 4, 2)$.)
4. Otherwise, for the sending nodes $P_{i_0}, \ldots, P_{i_{n-1}}$ in a subgroup, the start element is the first "1" entry, 2nd "1" entry, $\ldots$, n-th "1" entry for the row $i_0, \ldots, i_{n-1}$, respectively. (Hence, the start elements of (P_0, P_3), (P_1, P_4) and (P_2, P_5) are $(0, 1)$, $(4, 5)$ and $(2, 3)$, respectively.)
5. All such sequences of the processor number which begin at the start element SV^0 are composed of sending vectors SV, which are the rows of the scheduling table CS.

Figure 9 gives some examples of generation of the scheduling tables from the communication tables, according to Algorithm 3. The entries enclosed in a triangular are the start elements of the sending vectors.

Theorem 4 *Any column of a CS generated from Algorithm* 3 *is a permutation of the processor numbers* $(0, 1, \ldots, \tilde{p} - 1)$.

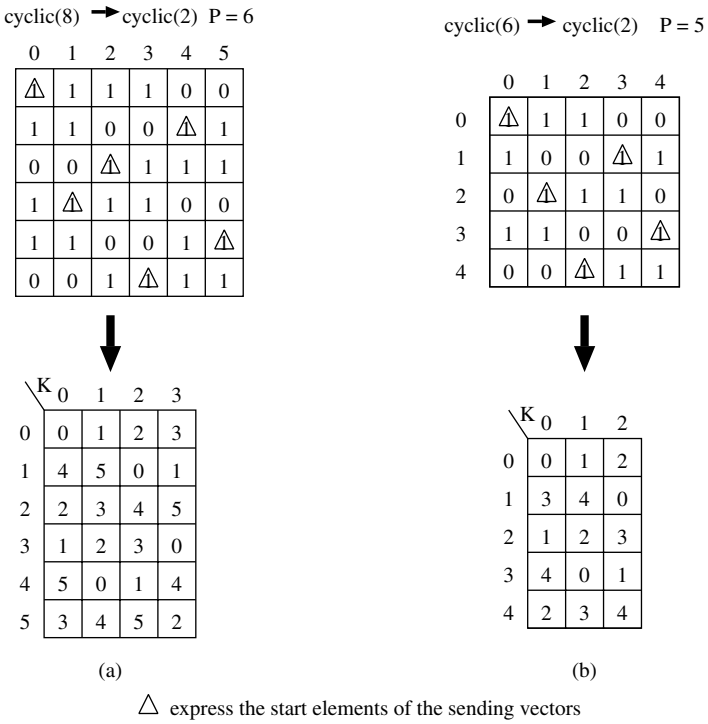

Figure 9. Generation of scheduling tables from communication tables.

Proof: *Refer to* [7]. □

5.2. Multi-dimensional scheduling algorithm

For a multi-dimensional case, if the redistribution is the "shape retaining" case, it can be performed by simply looking at the representations for each dimension and performing redistributions dimension-by-dimension independently. In this section we only consider the "shape changing" case. For the sake of simplicity, in the following discussion, we use 2-dimensional case to explain the multi-dimensional redistribution problems. We assume that the processor grid is $P = P_1 \times P_2$ before the redistribution and $P' = P'_1 \times P'_2$ after the redistribution. Hence the processor P_i can be represented in two-dimensional coordinate, that is $i = (i_1, i_2)$.

With respect to 2D array redistribution, as we proposed in Lemma 2 of Section 3,

$$D_i \cap D_j \neq \varnothing \Longleftrightarrow d_{i_1} \cap d_{j_1} \neq \varnothing \ \wedge \ d_{i_2} \cap d_{j_2} \neq \varnothing,$$

where $i = (i_1, i_2)$ is the source processor number and $j = (j_1, j_2)$ is the target processor number. Using the notation of the communication table COM, it can be represented as

$$COM[i, j] = COM[(i_1, i_2), (j_1, j_2)] = COM_1[i_1, j_1] \wedge COM_2[i_2, j_2],$$

where COM_1 and COM_2 are the communication tables corresponding to the first and second dimension, respectively.

The algorithm for determining the start element of each sending node is composed of the 1D algorithms applied repeatedly. The algorithm is as follows.

Algorithm 4 Similar to Algorithm 3, we also consider the start elements only.

1. First consider COM_1. Applying Algorithm 3, we can get the start elements $(J_1^0, \ldots, J_1^{P_1-1})$ for $I_1^0, \ldots, I_1^{P_1-1}$, such that,

   $$COM[(I_1^0, i_2), (J_1^0, j_2)], \ldots, COM[(I_1^{P_1-1}, i_2), (J_1^{P_1-1}, j_2)]$$

   form P_1 number of sub-communication tables. All these sub-communication tables are equal to another communication table COM_2 but no overlapping rows and columns in COM.
2. Then consider sub-communication tables

   $$COM[(I_1^0, i_2), (J_1^0, j_2)], \ldots, COM[(I_1^{P_1-1}, i_2), (J_1^{P_1-1}, j_2)].$$

   Applying Algorithm 3 again,

   2.1. if $P_2 \leq P_2'$ we can directly apply Algorithm 3 to tables

   $$COM[(I_1^0, i_2), (J_1^0, j_2)], \ldots, COM[(I_1^{P_1-1}, i_2), (J_1^{P_1-1}, j_2)],$$

   and obtain the start elements $(J_2^0, \ldots, J_2^{P_2-1})$ for $I_2^0, \ldots, I_2^{P_2-1}$.

   2.2. If $P_2 > P_2'$, it is possible there are not enough columns to get start elements in COM_2, then we compound two sub-communication tables $COM[(I_1, i_2), (J_1, j_2)]$ and $COM[(I_1, i_2)], (\hat{J}_1, j_2)]$ into a sub-table and use the Algorithm 3 to it, where $COM_1[I_1, J_1] = 1$ and $COM_1[I_1, \hat{J}_1] = 1$ and $\hat{J}_1$ is the first "1" entry following J_1.
3. The pairs (J_1^u, J_2^v) are the start elements for each sending node $(I_1^u, I_2^v)(0 \leq (u, v) < (P_1, P_2))$.

For example, consider an array redistribution (BLOCK, BLOCK) to (BLOCK, BLOCK) on $P = 2 \times 4$ to $P' = 4 \times 2$. The sub-communication tables COM_1 and COM_2, the communication table COM, and the CS table derived from Algorithm 4 are shown in Fig. 10(a), (b), (c) and (d), respectively, where $COM[(i_1, i_2), (j_1, j_2)]$ and $CS[(i_1, i_2), k]$ are represented as $COM[4i_1 + i_2, 2j_1 + j_2]$ and $CS[4i_1 + i_2, k]$.

6. Experimental results

For purpose of performance evaluation of our optimized redistribution algorithms and comparison with other redistribution work, in this section, we present the experimental evaluation for these techniques. All the experiments are implemented on

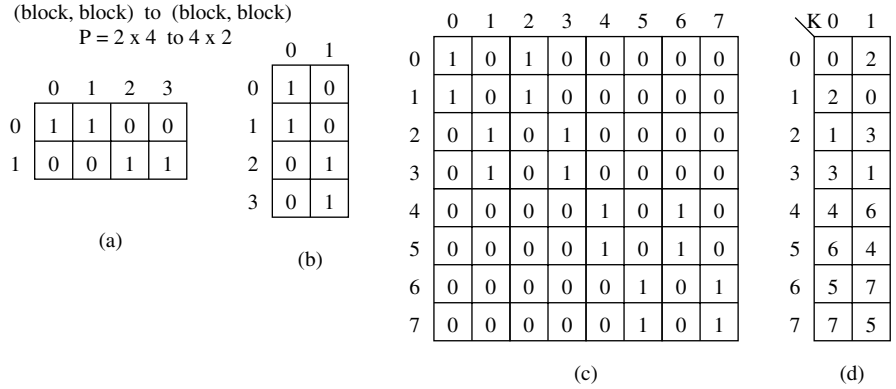

Figure 10. Example for multi-dimensional redistribution.

CP-PACS [3], a 2048-processor MIMD distributed memory parallel computer developed at the University of Tsukuba. The node programs are written in C, using PARALLELWARE[3] programming environment, a commercially available package that extends C and FORTRAN77 with a portable communication library.

6.1. Experimental results for efficient index computation

The redistribution routines implemented on CP-PACS require information about the source and target processor grids and source and target distribution schemes for each of arrays being redistributed. To compare with other redistribution works, we also implemented Thakur's one-dimensional algorithm described in [27, 28] and the naive approach described in Section 3. Because Thakur's algorithm can only handle a "shape retaining redistribution" and only focuses on index computation optimization, the routines used in this subsection are only optimized in index computation. The experimental results of communication scheduling are shown in next subsection.

Figure 11 shows the results of the one-dimensional redistribution from CYCLIC(12) to CYCLIC(5) with the data size $n = 120{,}000$, where *Naive*, T_{algo}, and G_{algo} indicate the naive approach, Thakur's algorithm and our algorithm, respectively. The main difference of these algorithms is that the Thakur's algorithms do the redistribution from CYCLIC(12) to CYCLIC(5) through two phases—first the redistribution from CYCLIC(12) to CYCLIC(60), then from CYCLIC(60) to CYCLIC(5)[4], but ours do the redistribution from CYCLIC(12) to CYCLIC(5) directly.

The above figure suggests that our algorithm can get much better performance than the naive approach. On the other hand, with respect to the redistribution from CYCLIC(x) to CYCLIC(y), if $\max(x, y)$ is not a multiple of $\min(x, y)$, our algorithm gets the better performance than Thakur's algorithm because data is communicated twice. However, when x is a multiple of y, the performance of our

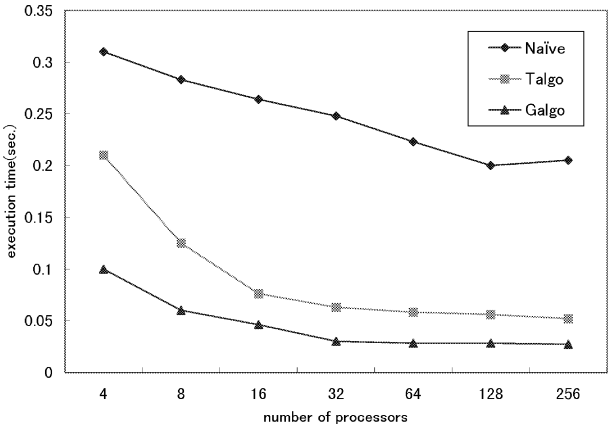

Figure 11. Performance of the three algorithms for one-dimensional redistribution from CYCLIC(12) to CYCLIC(5) on CP-PACS(data size: 120,000).

algorithm is almost close to Thakur's algorithm because it can do the redistribution directly.

For the multi-dimensional redistribution, we first perform an experiment of executing the "shape retaining" redistribution from (BLOCK, BLOCK) to (CYCLIC, CYCLIC), as the typical case for redistribution in two dimensions with the array size of 1024×1024. Figure 12 shows the results of this experiment by using the algorithms *Naive*, T_{algo}, and G_{algo}.

To show the efficiency and flexibility for the "shape changing" redistribution, we performed experiments on such cases as when either the source processor grid differs from the target processor grid, and when at least one dimension of the array is collapsed before or after redistribution. Table 1 shows some experimental results with array sizes of 300×300 and 600×600. A comparison of performance is made between the naive approach and our algorithm.

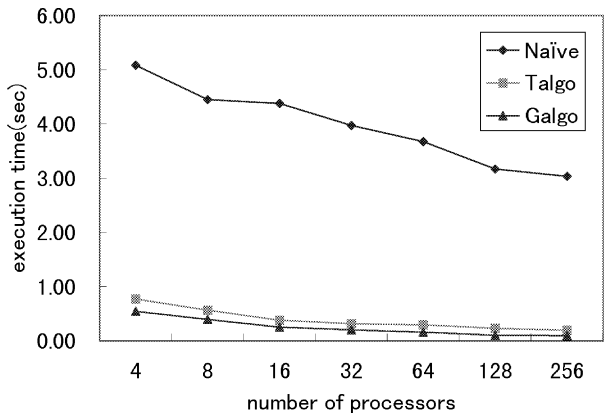

Figure 12. Performance of the three algorithms for the "shape retaining" redistribution from (BLOCK,BLOCK) to (CYCLIC,CYCLIC) on CP-PACS (data size: 1024×1024).

Table 1. Results for the "shape changing" redistribution on CP-PACS (time in seconds)

Redistributions	Array Size	P_s	P_t	Naive	G_{algo}
(CYCLIC, BLOCK)	300×300	3×3	5×2	0.46	0.11
to (BLOCK, CYCLIC)		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)	300×300	4×5	10×1	0.24	0.03
to (BLOCK,*)		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,*)	300×300	20×1	1×20	0.19	0.09
to (*,BLOCK)		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

These figures and table indicate:

- Our algorithm works better than the naive algorithm for the "shape retaining redistribution," irrespective of the number of processors or data size. In addition, our algorithm can get higher performance than Thakur's algorithm in a general sense.
- For the "shape changing redistribution," our algorithm works well for all of the redistributions on arbitrary processor sets. It performs better than the naive algorithm, and the speedup ranges from 200% to 900%.

To estimate the performance of our algorithms on a distributed environment, we also conducted some experiments on a workstation cluster connected with network [6]. The results of the experiments match the one on a massively parallel computer.

6.2. Experimental results for communication scheduling

In this subsection, we concentrate on the results that demonstrate the usefulness of the communication scheduling optimizations we presented in Section 5. We use our earlier algorithm without scheduling for the sake of comparison.

We carried out an experiment in one-dimensional case with a communication step $\mathcal{K} = 3$. Figure 13 shows the result of the experiment. The curve "without scheduling" represents our redistribution algorithm without a communication scheduling, and "with scheduling" represents the algorithm with a communication scheduling optimization presented in Section 5.

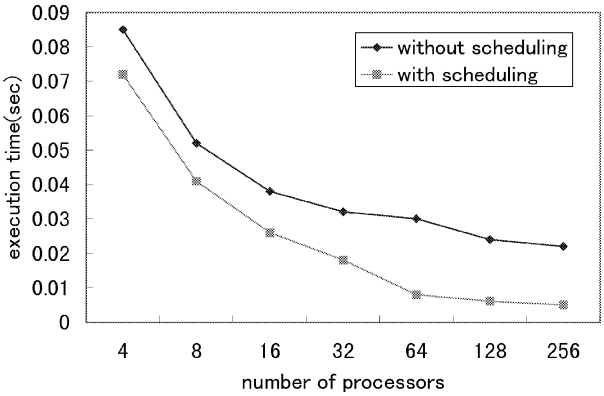

Figure 13. Comparison of the performance with and without communication scheduling in the redistribution algorithm on CP-PACS (data size: 120,000, $\mathscr{K} = 3$).

Figure 13 shows that the algorithm with the communication scheduling optimization achieves better performance than the former algorithm. The performance improvement becomes more appreciable as the number of processors increases. This means it is vital to use communication scheduling in redistribution algorithms.

7. Conclusions and future work

In this paper, we have focused on producing a framework of optimization both in index computation and in inter-processor communication for redistribution algorithms. In order to build different data for each pair of processors that must communicate, and further guarantee there is no more redundant data to be communicated, we have presented a model to describe the local data distributed onto the local memories. The array elements distributed onto a processor can be represented by a 4-tuple in each dimension. The 4-tuple, called the Local Data Descriptor (LDD), can describe all of the block-cyclic distribution schemes. Our redistribution is based on the notion of LDD and the intersection of two LDDs. The communication optimization is achieved through scheduling communication to avoid the node contention.

In the future, we would like to extend our scheduling algorithm to the one which considers the communication message size, because different sending processors may send the messages in different lengths to receiving processors [4]. It is easy to add such an extention into our algorithms because the block length of the intersection of two LDDs is the message size, which can be the entries of the communication table.

Notes

1. Because we assume $b = \beta * b'$ in p. 260 all sending processors redistribute data to some neighbouring receiving processors.

2. $\prec$ is a special ascending order designated as "$<$" but taken a round of $\tilde{p}$. For example, if $\tilde{p} = 6$ and $j_0 = 3$, then $3 \prec 4 \prec 5 \prec 0 \prec 1 \prec 2$.
3. PARALLELWARE is a trademark of Nippon Steel Corporation. The trademark of the same software in America is Express.
4. Because their algorithms can only redistribute between CYCLIC(kx) and CYCLIC(x).

References

1. R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the 1994 International Conference on Parallel Archs. and Compilation Techniques*, Montreal, Canada, Aug. 1994.
2. Y. Chung, C. Hsu, and S. Bai. A basic-cycle calculation technique for efficient dynamic data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):359–377, 1988.
3. K. Nakazawa, H. Nakamura, T. Boku, I. Nakata, and Y. Yamashita. CP-PACS: a massively parallel processor at the University of Tsukuba. *Parallel Computing*, 25(13–14):1635–1661, 1999.
4. F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Scheduling block-cyclic array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):192–205, 1998.
5. M. Guo, Y. Yamashita, and I. Nakata. Efficient implementation of multi-dimensional array redistribution. *IEICE Transactions on Information and Systems*, E81-D(11):1195–1204, 1998.
6. M. Guo, Y. Yamashita, and I. Nakata. Improving performance of multi-dimensional array redistribution on distributed memory machines. In *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Orlando, Fla. March 1998.
7. M. Guo. Efficient techniques for data distribution and redistribution in *parallelizing compilers*. Ph.D. Thesis, University of Tsukuba, Japan, July 1998.
8. HPF Forum. *High Performance Fortran Language Specification*, version 2.0 ed. Rice University, Houston, Texas, 1996.
9. C. Hsu, S. Bai, Y. Chung, and C. Yang. A generalized basic-cycle calculation method for efficient array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1201–1216, 2000.
10. S. D. Kaushik, C.-H. Huang, R. W. Johmson, and P. Sadayappan. An approach to communication-efficient data redistribution. In *Proceedings of the 8th ACM International Conference on Supercomputing*, Manchester, U.K., July 1994.
11. S. D. Kaushik, C.-H. Huang, and P. Sadayappan. Efficient index set generation for compiling HPF array statements on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 38(2):237–247, 1996.
12. S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase redistribution: a communication-efficient approach to array redistribution. Technical report, The Ohio State University, 1995.
13. 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.
14. K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proceedings of the International Conference on Supercomputing*, Barcelona, July 1995.
15. K. Kennedy and U. Kremer. Automatic data layout for high performance Fortran. In *Proceedings of Supercomputing'95*, San Diego, Calif., Dec. 1995.
16. U. Kremer. NP-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, Dec. 1993.
17. Y. W. Lim, P. B. Bhat, and V. Prasanna. Efficient algorithms for block-cyclic redistribution of arrays. *IEEE Symposium on Parallel and Distributed Processing*, Oct. 1996.
18. Y. W. Lim, N. Park, and V. Prasanna. Efficient algorithms for multi-dimensional block-cyclic redistribution of arrays. In *Proceedings of the 26th International Conference on Parallel Processing*, Bloomingdale, IL, Aug. 1997.
19. 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).

20. D. J. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. *In Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1995.
21. D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Dynamic data partitioning for distributed-memory multicomputers. *Journal of Parallel and Distributed Computing*, No. 38:158–175, 1996.
22. N. Park, V. K. Prasanna, and C. S. Raghavendra. Efficient algorithms for block-cyclic array redistribution between processor sets. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1217–1239, 1999.
23. S. Ramaswamy, B. Simons, and P. Banerjee. Optimizations for efficient array redistribution on distributed memory multicomputers. *Journal of Parallel and Distributed Computing*, 38:217–228, 1996.
24. S. Ranka, J.-C., Wang, and G. Fox. Static and run-time algorithms for all-to-many personalized communication on permutation networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1266–1274, (1994).
25. S. Ranka, R. Shankar, and K. Alsabti. Many-to-many personalized communication with bounded traffic. In *Proceedings of Frontiers'95*, 1995.
26. J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: design, implementation, and evaluation, *Journal of Parallel and Distributed Computing*, pp. 150–159, 1994.
27. R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. In *Proceedings Scalable High Performance Computing Conference*, May 1994, pp. 309–316.
28. R. Thakur, A. Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587–593, 1996.
29. E. H. Tseng and J. L. Gaudiot. Communication generation for aligned and cyclic(k) distributions using integer lattice. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):136–146, 1999.