Available online at www.sciencedirect.com

SCIENCE DIRECT®

J. Parallel Distrib. Comput. 65 (2005) 553–563

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Improving communication scheduling for array redistribution

Minyi Guo[a, b, *], Yi Pan[b]

[a]Department of Computer Software, The University of Aizu, Aizu-Wakamatsu City, Fukushima 965-8580, Japan
[b]Department of Computer Science, Georgia State University, University Plaza, Atlanta, GA 30303, USA

Received 29 August 2002; received in revised form 15 October 2004; accepted 7 December 2004

Abstract

Many scientific applications require array redistribution when the programs run 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. If there is no communication scheduling in a redistribution routine, the inter-processor communication will incur a larger communication idle time when there exists node contention and/or difference among message lengths during one particular communication step. In order to solve this problem, in this paper, we propose an efficient scheduling scheme that not only minimizes the number of communication steps and eliminates node contention, but also minimizes the difference of message lengths in each communication step. Thus, the communication idle time is reduced in redistribution routines.

Keywords: Automatic parallelization; Array redistribution; Communication scheduling; Block-cyclic distribution; MPI; Distributed memory multicomputers

1. Introduction

The array redistribution problem has recently received considerable attention. This interest is motivated largely by the observation that the most of scientific applications consist of several separated nested loops, which are decomposed into phases in the HPF [6] programming style. At each phase, there is an optimal distribution of arrays onto the processor grid. Many applications and kernels, such as alternating direction implicit (ADI), two-dimensional fast fourier transform (FFT) and signal processing, require different array distributions at different computation phases for efficient execution on distributed memory machines. For example, in our previous study, we found that we had to parallelize different loops in different phases due to different data dependence in those phases in a computational electromagnetics (CEM) application [19]. In order to reduce the number of remote accesses, efficient data redistribution is necessary. Because the optimal distribution changes from phase to phase, array redistribution turns out to be a critical operation.

A redistribution problem can simply be described as: A set of routines that change the distribution schemes such that, given a multi-dimensional array *A* on a set of source processors with a source distribution scheme, they transfer all the elements of the array to a set of target processors with a target distribution scheme. Therefore, the redistribution routines need to figure out exactly what data need to be sent (received) by each source (target) processor.

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 it also designates the destination processor numbers of such array elements. When the processors exchange the array elements, the communication overheads will occur, which 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. For the first aspect, many researchers have mainly concentrated on the efficient index computation for generating the messages to be exchanged by the processors involved in the redistribution [2,4,7,10,20,26]. These

* Corresponding author. Fax:+81 242 37 2744
E-mail address: minyi@u-aizu.ac.jp (M. Guo).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.12.001

research projects precisely built a different message for each pair of processors that must communicate each other. This effort guarantees that each processor sends or receives no redundant data. However, until now researchers rarely pay attention to the question of how to schedule the communication in redistribution routines except researches described in [3,8,20,24], where, however, they only focused on reducing the data transmission cost for a special case. Our previous work [5] studied the communication scheduling to avoid node contention. However, because we did not align the message length in each communication step, it seems that the algorithms cannot completely minimize the communication idle time (refer to the motivating examples in Section 2).

In this paper, we focus on the communication scheduling for one-dimensional redistribution that redistributes from *cyclic*(x) on p source processors(numbered from 0 to $p-1$), to *cyclic*(y) on q (numbered from 0 to $q-1$) target processors. For the sake of simplicity, assume that the size of A is a multiple of $L = \mathrm{lcm}(xp, yq)$, because the redistribution pattern repeats after each section of L elements (we call this consecutive L elements a *slice* in a global array). We improve our previous scheduling algorithms [5] so that the new algorithms guarantee that the message communications are not only node contention-free, but they also minimize the total length of messages in all communication steps.

Our method can be extended to multi-dimensional arrays easily. We have implemented our algorithms on a massively parallel MIMD machine. The experimental results show that the algorithms provide performance improvement for some redistribution samples.

The rest of the paper is organized as follows: Section 2 gives some motivating examples and describes the scheduling problem. Related work is discussed in Section 3. Section 4, the core of this paper, proposes some algorithms for improving communication according to different redistribution patterns. The experimental evaluations will be shown in Section 5. Finally, Section 6 presents our conclusions.

2. Preliminaries, problem description and motivating examples

The following is an example template of a simple algorithm for array redistribution:

```
for $dest = 0$ to $q-1$
    compute all the elements that must send to dest;
    pack the data into buffer[dest];
endfor
for $dest = 0$ to $q-1$ other than me
    send(buffer[dest], dest);
endfor
for $src = 0$ to $p-1$ other than me
    receive(buffer, src);
    unpack the buffer;
endfor
```

where *me* is the logical number of the processor executing the program. Note that there is no explicit scheduling in this method: all messages are sent almost simultaneously to the same target process. This approach induces a tremendous amount of waiting time for all receiving processors. Our experiment shows that this communication pattern results in overall performance degradation [5].

There are three main possibilities for implementing the communications induced by a redistribution operation [3]:

Asynchronous (*non-blocking*) *communications.* Each source processor communicates with a destination processor by using non-blocking communication primitives. There is no obvious barrier and synchronous communication steps. This asynchronous strategy requires buffering for all the messages to be received.

Synchronous communications. A synchronized algorithm involves communication phases and steps. At each step, each participating processor posts a request, sends a message, and then waits for the completion of the receive. This strategy may result in situations where some processors may have to wait for others before they can receive any data, or hot spots can arise if several processors attempt to send messages to the same processor at the same step.

`MPI_Alltoallv` *communication*. Using `MPI_Alltoallv` message passing routine in MPI can also perform array redistribution.

All communication schemes have certain advantages over the other. In this research, both synchronous communication and asynchronous communication are used in different places to reduce communication times within a data redistribution. Furthermore, we also compare our approaches with `MPI_Alltoallv` implementation.

2.1. Local data descriptor, communication table, and scheduling table

In [4,5] we proposed an approach to generate the redistribution algorithm, which is based on the representation called local data descriptor (LDD). An LDD expresses the set of the array elements partitioned into a local distributed memory. We also defined some operations on LDDs and referred to the fact that the data being redistributed between two processors is indicated by the intersection of LDDs of the processors. Because we will use some concepts of LDD in the following sections, in this section we give an overview of the redistribution algorithm based on LDD described in our earlier papers [4,5]. Further details can be found there.

Definition of one-dimensional LDD. A 4-tuple $d = (o, b, s, n)$ is called an LDD 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 the length of the block; s the stride between two consecutive blocks; and n is the number of blocks distributed onto 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 corresponds to a set of the global array index defined as follows:

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

We also use $|S[d]|$ to represent the number of element of set $S[d]$ within the slice L.

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 sets are $S[d_1]$ and $S[d_2]$ (namely LDD set), respectively. The intersection of $S[d_1]$ and $S[d_2]$ is defined as follows:

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

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

$$\begin{aligned} &S[d_1] \cap S[d_2] \neq \emptyset \\ &\quad \Longleftrightarrow \forall u_1, u_2 (0 \leqslant u_1 < n_1 \wedge 0 \leqslant u_2 < n_2), \\ &\qquad \max(o_1 + s_1 * u_1, o_2 + s_2 * u_2) \\ &\qquad < \min(o_1 + b_1 + s_1 * u_1, o_2 + b_2 + s_2 * u_2), \end{aligned} \quad (1)$$

and

$$\begin{aligned} &|S[d_1] \cap S[d_2]| \\ &\quad = \sum_{u_1, u_2 = 0}^{\frac{L}{xp}, \frac{L}{yq}} (\min(o_1 + b_1 + s_1 * u_1, o_2 + b_2 + s_2 * u_2) \\ &\quad - \max(o_1 + s_1 * u_1, o_2 + s_2 * u_2)). \end{aligned} \quad (2)$$

Communication table and scheduling table: We construct a communication matrix (table) M for a redistribution, to describe the communication pattern between the source and target processor sets. An (i, j) entry, which is not equal to 0, represents the fact that a sending processor P_i needs to communicate to a receiving processor Q_j with the message size $M(i, j)$. That is, $M(i, j) = m$ if and only if sending processor P_i sends a data item with size m (in a slice L) to receiving processor Q_j. According to the usage of LDDs, let d_i and d_j be the LDDs of processors P_i and P_j, respectively, then

$$M(i, j) = \begin{cases} |S[d_i] \cap S[d_j]|, & d_i \cap d_j \neq \emptyset, \\ 0, & d_i \cap d_j = \emptyset. \end{cases} \quad (3)$$

A communication scheduling table CS expresses the scheduling result where $CS(i, k) = j$ means that a sending processor P_i sends a message to a receiving processor Q_j at a communication step k. (A communication step is defined as the time during which all the sending and receiving processor pairs complete a communication. A redistribution routine needs several communication steps.) At a communication step k, if a sequence $CS(0, k), CS(1, k), \ldots, CS(p-1, k)$ is a permutation of the set of receiving processor indices, we say the communication is contention-free.

2.2. Motivating examples

Example 1. Consider the first example with $x = 4, y = 3, p = q = 5$. The redistribution routine needs all-to-all communication. From Formulas (1)–(3), the Communication table M can be obtained as shown in Fig. 1(a). If we only need to avoid the contention in all communication steps, we can add a statement *scheduled_dest* $= ($*me* $+$ *dest*$) \bmod p$ into the sending loop of a redistribution routine, and schedule the send as *send*(*buffer*[*scheduled_dest*], *scheduled_dest*). Thus, from this scheduling, the CS table may be shown as Fig. 1(b). Although it can improve the communication performance because node contention is avoided, there still exists communication waiting times. The reason is that the cost of a step is likely to be dictated by the length of the longest message exchanged during the step, and at each communication step, processors exchange messages with the different lengths. Fig. 1(d) shows the communication scheduling in this pattern. A horizontal bar consisting of several differently numbered boxes (or blank boxes) represents that a sending processor sends a message with different lengths (different number of blocks) to the receiving processors. A box marks as one-unit message and the number in the box is the target processor number. That is, the differently numbered boxes mark the messages which are sent to different

cyclic (4) to cyclic (3), p = q = 5

	Q_0	Q_1	Q_2	Q_3	Q_4
P_0	3	2	3	2	2
P_1	3	2	2	3	2
P_2	2	3	2	3	2
P_3	2	3	2	2	3
P_4	2	2	3	2	3

Communication table

(a)

k	1	2	3	4	5
P_0	0	1	2	3	4
P_1	1	2	3	4	0
P_2	2	3	4	0	1
P_3	3	4	0	1	2
P_4	4	0	1	2	3

Scheduling table

(b)

k	1	2	3	4	5
P_0	0	1	2	3	4
P_1	3	4	0	1	2
P_2	1	2	3	4	0
P_3	4	0	1	2	3
P_4	2	3	4	0	1

Scheduling table

(c)

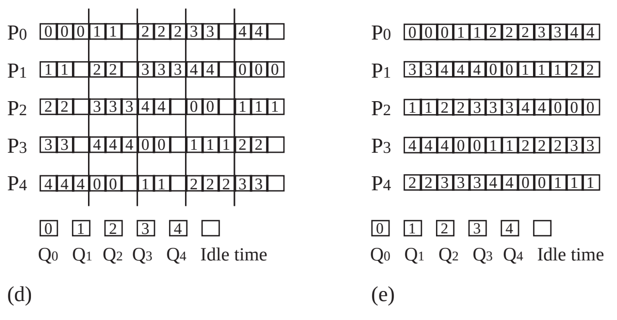


Fig. 1. Motivating example 1.

cyclic (2) to cyclic (3), p = q = 6

	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5
P_0	2	0	2	0	2	0
P_1	1	1	1	1	1	1
P_2	0	2	0	2	0	2
P_3	2	0	2	0	2	0
P_4	1	1	1	1	1	1
P_5	0	2	0	2	0	2

Communication table

(a)

k	1	2	3	4	5	6
P_0	0		2		4	
P_1	4	5	0	1	2	3
P_2	1		3		5	
P_3	2		4		0	
P_4	5	4	1	0	3	2
P_5	3		5		1	

Scheduling table

(b)

K	1	2	3
P_0	0	2	4
P_1	45	01	23
P_2	1	3	5
P_3	2	4	0
P_4	54	10	32
P_5	3	5	1

Scheduling table

(c)

Fig. 2. Motivating example 2.

receiving processors. The vertical lines separate the communication steps. Obviously, there exists waiting time (blank boxes) in each step. As the message length increases, the waiting time may increase considerably as well.

On the other hand, if we schedule the communication according to another *CS* table (Fig. 1(c)), the corresponding scheduling pattern is optimized as shown in Fig. 1(e). We observe that the communications are not only contention-free, but the message length in each step is equal to each other. In other words, each processor starts to send the message and finishes communication simultaneously in a step. Hence, the minimum waiting time can be achieved.

Example 2. Consider the following example with $x = 2$, $y = 3$, $p = q = 6$. The communication matrix is shown in Fig. 2(a). Obviously, as the communication patterns are different among some sending processors (e.g., processor P_0 and P_3 need not send message to processor Q_1, Q_3 and Q_5), we can extend the contention-free scheduling algorithm and obtain the scheduling result as shown in Fig. 2(b). Because the barrier synchronization is carried out at each communication step, the total cost is estimated as 9 unit lengths (2 units in 3 steps $k = 1, 3, 5$ and 1 unit in 3 steps $k = 2, 4, 6$). On the other hand, if the asynchronization (non-blocking) communication strategy is applied and the barrier is carried out in a "large" step (two communication steps), the communication can be optimized to the cost of 6 unit lengths(Fig. 2(c)). This is because, with respect to processors P_1 and P_4, the sum of the message lengths in two "small" steps is just same as the message lengths of one step for processor P_0, P_2, P_3, and P_5. The schedule generated not only results in contention-free but also minimizes the communication waiting time. In fact, there is no need to actually insert barrier when implementing these algorithms using MPI, because in MPI implementation, if the communication link is busy, the communication system delays the communication.

From the above observation, we need to consider communication scheduling techniques for some redistribution schemes such that the communication becomes contention-free and the difference of message lengths in each communication step is minimized. In this way, the communication idle time is reduced in redistribution routines.

We divide the redistribution schemes into two categories: one is that processors send and/or receive the same number of messages, as well as if a message of length *m* occurs in one of the sending/receiving processor pair, it also occurs in the other pairs. Another is unbalanced pattern, in which processors may have a different number of messages to send and/or receive.

3. Related work

Recently, several algorithms have been reported that handle general block-cyclic redistribution. However, most of the work involves message generation only. For instance, Thakur et al. [26,27] 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 pair-wise 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 multidimensional 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 employed. Further, the approach used for multidimensional array redistribution involves a series of one-dimensional redistributions, which can be costly.

Chung et al. [2] presented a basic-cycle calculation technique to efficiently perform $cyclic(x)$ to $cyclic(y)$ redistribution. 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 redistribution from $cyclic(x)$ over P processors to $cyclic(y)$ over Q processors [7]. 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 table to minimize the data-movement operations was constructed.

Many researchers have concentrated mainly on the efficient index computation for generating the communication messages to be exchanged by the processors involved in the redistribution [1,11,12,21,22,25]. However, the question of how to efficiently schedule the messages has received little attention. The followings are the researches concerned with the communication optimization in redistribution.

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. Hsu et al. [8] extended their idea and proposed a generalized processor mapping technique for *cyclic*(kx) to *cyclic*(x) redistribution or vice versa. The main idea of these methods is to develop mapping functions for computing a new rank of each target processor. Based on the mapping functions, a new logical sequence of target processors can be derived.

A multi-phase redistribution approach is suggested in [9,26]. 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 so that the total cost of the sequence is less than that of direct redistribution.

Lim et al. [14,15] 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 circular 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. They stated that their algorithms generate a schedule that minimizes the number of communication steps and eliminates node contention in each communication step. Following their work [20], 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.

Desprez et al. [3] 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, their algorithm did not provide the overlap between communication steps and also may cause communication contention.

All the existing work either deals with the special redistribution scheme (*cyclic*$(kx) \rightarrow cyclic(x)$ or vice versa), or only avoids the node contention. Our technique, however, can deal with all redistribution schemes (*cyclic*(x) on p processors to *cyclic*(y) on q processors); especially for some “irregular” cases—processors may have a different number of messages to send and/or receive—our method can achieve better performance as evidenced in our experimental results.

4. Communication scheduling algorithms

Given the global address g of an array element, we can easily determine the processor P that owns this element and the local address Loc of the element on that processor using the following formulas:

$$P = (g \, div \, x) \bmod p, \tag{4}$$

$$Loc = x * (g \, div \, px) + g \bmod x. \tag{5}$$

Without loss of generality, we can assume $gcd(x, y) = 1$, because if $gcd(x, y) = z \neq 1$, let $x = x'z$, $y = y'z$, and M' be the communication table of redistribution from *cyclic*(x') to *cyclic*(y'), we can prove $M = z * M'$. Hereby there is no influence for M's pattern with assumption $gcd(x, y) = 1$. This fact can be validated by the following lemma.

Lemma 2. *For an arbitrary processor pair* (P_i, Q_j), *if there is* $M'(i, j) = m$ *under redistribution cyclic*(x) *to cyclic*(y), *then* $M(i, j) = z * m$ *is true under redistribution cyclic*(zx) *to cyclic*(zy), *where z is a positive integer*.

Proof. According to Formula (1), an element, whose global subscript is g, redistributed from P_i to Q_j satisfies $P_i = (g \, div \, x) \bmod p$, and $Q_j = (g \, div \, y) \bmod q$. Because $M'(i, j) = m$ means that there are m such elements in a slice L, we assume that these elements' global subscripts are $g_1, \ldots, g_m$.

For a $g \in \{g_1, \ldots, g_m\}$, $P_i = (g \, div \, x) \bmod p \Rightarrow$ existing z numbers of the elements $(gz, gz + x, \ldots, gz + (z-1) * x))$ satisfy $P_i = (gz \, div \, zx) \bmod p, \ldots, P_i = ((gz + (z-1) * x) div \, zx) \bmod p$. The similar result occurs in $Q_j = (gz \, div \, zy) \bmod q, \ldots$. Because there are m numbers of such g, the amount of such elements is $z * m$. This means $M(i, j) = z * m$ under redistribution *cyclic*(zx) to *cyclic*(zy). $\square$

We divide the following discussions into two cases according to whether $gcd(x, q) = 1 \wedge gcd(y, p) = 1$ or not, because this condition indicates that if a source processor sends a message with size m to a destination processor, the other source processors certainly must send a message with the same length to a corresponding destination processor.

4.1. Scheduling algorithm when $gcd(x, q) = 1 \wedge gcd(y, p) = 1$

For the case of scheduling when $gcd(x, q) = 1 \wedge gcd(y, p) = 1$, we have $L = lcm(xp, yq) = \Delta * lcm(x, y) = \Delta * x * y$, where $\Delta > 1$ is a constant. The algorithm is based on the following theorem.

Theorem 1. *Given redistribution parameters, x, y, p, q, and let M represent the communication table from source processors to target processors, then*

$$M(i, j) = m \Rightarrow M((i + y * k) \bmod p, (j + x * k) \bmod q) = m, \quad (0 \leqslant k < \Delta).$$

Proof. $M(i, j) = m$ means that, m elements of the array A in a slice L, with global addresses $g_1, g_2, \ldots, g_m$, are distributed onto P_i and Q_j before and after redistribution, respectively. Let $A[g]$ be the one of these m elements, then $A[g + x * y]$ is located on source processor $P_{(i+y) \bmod p}$ (because the block length of source distribution pattern is x), and on target processor $Q_{(j+x) \bmod q}$, respectively. Similarly, $A[(g+x*y*k) \bmod L]$ $(0 \leqslant k < \Delta)$ is located on source processor $P_{(i+y*k) \bmod p}$ and on target processor $Q_{(j+x*k) \bmod q}$. Because $A[g] \in \{A[g_1], A[g_2], \ldots, A[g_m]\}$, $g' \neq g'' \Rightarrow (g' + x * y * k) \bmod L \neq (g'' + x * y * k) \bmod L$. Thereby $A[(g + x * y * k) \bmod L]$ can also contain m different elements in a slice. This means $M((i + y * k) \bmod p, (j + x * k) \bmod q) = m$, which is required for our proof. □

The values $(i+y*k) \bmod p$ are different each other when $0 \leqslant k < \frac{L}{x*y}$, because $gcd(y, p) = 1$. The same conclusion can be obtained with $(j + x * k) \bmod q$. In other words, according to Theorem 1, each source processor can send message with the same size to distinct target processor in a step. The source processors $P_0, P_1, \ldots, P_{p-1}$ send messages of size m to the target processors $(Q_{j_0}, Q_{j_1}, \ldots, Q_{j_{q-1}})$, where $j_0, j_1, \ldots, j_{q-1}$ is a permutation of target processor numbers $(0, 1, \ldots, q - 1)$. Thus the scheduling algorithm can be as follows (CS_i indicates the ith row of CS and $\mathcal{K}_i$ is the number of communication steps of P_i):

Algorithm 1. (Scheduling algorithm when $gcd(x, q) = 1 \wedge gcd(y, p) = 1$).

Step 1: For a source processor P_{i_0}(e.g., P_0), its CS vector is calculated as $CS_{i_0} = \langle Q_{j_0}, Q_{j_1}, \ldots, Q_{j_{\mathcal{K}_{i_0}-1}} \rangle$, where $M(i_0, j_k) \neq 0$ (in principle, $j_0, j_1, \ldots, j_{\mathcal{K}_{i_0}-1}$ can be any permutation of target processor numbers $(0, 1, \ldots, q - 1)$).

Step 2: For any other source processors $P_i(i \neq i_0 \wedge 0 \leqslant i < p - 1)$, *if* $\exists k.i = (i_0 + y * k) \bmod p$, then the CS vectors of P_i are calculated as
$CS_i = \langle Q_{j'_0}, Q_{j'_1}, \ldots, Q_{j'_{\mathcal{K}_i-1}} \rangle$, where $j'_l = (j_l + x * k) \bmod q (0 \leqslant l < \mathcal{K}_i)$. □

Back to the motivating example 1. First we can specify the scheduling vector for P_0 as $CS_0 = \langle 0, 1, 2, 3, 4 \rangle$. Then for P_1, $i = (i_0 + y * k) \bmod p \Rightarrow 1 = (0 + 3 * 2) \bmod 5 \Rightarrow k = 2 \Rightarrow j'_l = (j_l + 4*2) \bmod 5 = (j_l + 3) \bmod 5 (0 \leqslant l < 5)$, that is $CS_1 = \langle 3, 4, 0, 1, 2 \rangle$. The other CS_i can be obtained by using the similar computations. The resulting CS is shown as Fig. 1(c).

4.2. Scheduling algorithm when $gcd(x, q) \neq 1 \vee gcd(y, p) \neq 1$

Algorithm 1 is based on the fact that the receiving processors form a permutation of the target processors in a communication step. This is guaranteed by the values $(j + x * k) \bmod q (0 \leqslant k < \frac{L}{x*y})$ are different from each other, as the value k is different. The similar result occurs in the formula $(i + y * k) \bmod p$. However, if $gcd(x, q) = t \neq 1 \vee gcd(y, p) = s \neq 1$, then for $k = 0, 1, \ldots, p-1$, because y is divided by s, $(y * k) \bmod p = (y * k + y * \frac{p}{s}) \bmod p = \cdots = (y * k + y * \frac{p}{s} * s) \bmod p$. Simplifying this equality, $\exists n (0 \leqslant n \leqslant s)$. $(y * k) \bmod p = (y * (n * \frac{p}{s} + k)) \bmod p$. That is, only if $0 \leqslant k < \frac{p}{s}$, $(y * k) \bmod p$ have different values. A similar result can be obtained for $0 \leqslant k < \frac{q}{t}$, $(x*k) \bmod q$. In other words, there certainly exist $i_1, \ldots, i_{\frac{p}{s}}$ and $j_1, \ldots, j_{\frac{q}{t}}$ such that $M(i_1, j_1) = \cdots = M(i_{\frac{p}{s}}, j_1) = \cdots = M(i_{\frac{p}{s}}, j_{\frac{q}{t}})$, where $i_k = i_1 + k * s$ and $j_k = j_1 + k * t$. Algorithm 1 cannot be applied in this case. Therefore, through the above observations, we have the following theorem.

Theorem 2. *Given redistribution parameters, x, y, p, and q, the communication matrix M can be divided into $\frac{p}{s} \times \frac{q}{t}$ equivalent sub-matrices. Each sub-matrix has the size $s \times t$:*

$$M = \begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1,\frac{q}{t}} \\ C_{21} & C_{22} & \ldots & C_{2,\frac{q}{t}} \\ \ldots & \ldots & \ldots & \ldots \\ C_{\frac{p}{s},1} & C_{\frac{p}{s},2} & \ldots & C_{\frac{p}{s},\frac{q}{t}} \end{bmatrix},$$

$$C_{ij} = \begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1,t} \\ c_{21} & c_{22} & \ldots & c_{2,t} \\ \ldots & \ldots & \ldots & \ldots \\ c_{s,1} & c_{s,2} & \ldots & c_{s,t} \end{bmatrix}.$$

Notice that C_{ij} are identical to each other. $\Sigma_{j=1}^{t} c_{1j} = \cdots = \Sigma_{j=1}^{t} c_{sj}$ in a sub-matrix(block) is the total message length in this block. Therefore, we can extend the communication step to a "large" step in which sending processors send $\frac{\mathcal{K}_i t}{q}$ messages to several receiving processors in one block. That is, the barrier is not inserted in every communication step but between two blocks. Asynchronous communication strategy (non-blocking communication in MPI) can be used in the large step. For each sending processor, because the sum of message sizes are the same in a large step, communication can be simultaneously completed at the end of the step, if there is no node contention.

Let $\kappa_i = \frac{\mathcal{K}_i t}{q}$, be the number of small steps in a block, and n be the large step number for P_i (we will omit the subscript i of κ_i for P_i if there is no confusion). For a sending

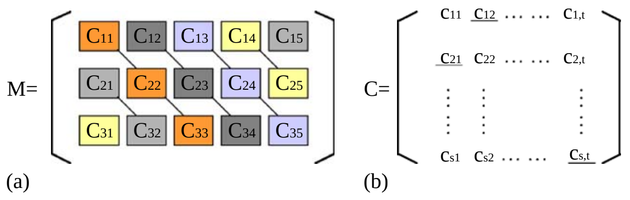


Fig. 3. Large step and small step scheduling when $gcd(x, q) \neq 1 \vee gcd(y, p) \neq 1$. (a) The communication table M is composed of $\frac{p}{s} \times \frac{q}{t}$ sub-matrix. The scheduling in a large communication step is along with diagonal subblocks. (b) All C_{ij} are identical to each other. Schedule the communication with the same size in a small step.

processor P_i, if the scheduling of the first large step is determined (denoted as $Q_{j_1}, Q_{j_2}, \ldots, Q_{j_\kappa}$), the processor numbers in the following large steps are $((Q_{j_1}, Q_{j_2}, \ldots, Q_{j_\kappa}) + t * (n-1)) \bmod q$. In order to schedule the first large communication step optimally, we select the diagonal sub-matrices $C_{11}, C_{22}, \ldots, C_{\frac{p}{s},\frac{p}{s}}, C_{1,2}, \ldots, C_{\frac{p}{s},\frac{p}{s}+1}, \ldots,$ and $C_{1,h}, \ldots, C_{\frac{p}{s},(\frac{p}{s}+h-1) \bmod \frac{q}{t}}$ as the scheduling lines in the first large step, which guarantees that all sub-matrices have different receiving processor numbers (assume $\frac{p}{s} < \frac{q}{t}$, without loss of generality). Then we schedule the small step (communicate with one processor) in these sub-matrixes. We can schedule the communication with the same message size in a small step, according to the descending order of message sizes, because the messages with the same size are sent to the different receiving processors in a block (Theorem 1). The details are shown in the Algorithm 2. Thus the node contention can be avoided as much as possible (Fig. 3). The scheduling algorithm for this case is shown below.

Algorithm 2. (Scheduling algorithm when $gcd(x, q) \neq 1 \vee gcd(y, p) \neq 1$).

```
Step 1. Divide the $M$ into $\frac{p}{s} \times \frac{q}{t}$ equivalent sub-matrices
        $C_{ij} (1 \leqslant i \leqslant \frac{p}{s}, 1 \leqslant j \leqslant \frac{q}{t})$.
Step 2. Do the following sub-steps:
    2.1. Put sub-matrices $C_{1,h}, C_{2,h+1}, \ldots, C_{\frac{p}{s},(\frac{p}{s}+h-1) \bmod \frac{q}{t}}$ as the first large communication step. The first $\kappa$ receiving processor numbers in $CS_i$ of $P_i (0 \leqslant i < p-1)$ are denoted as $\langle Q_{j_1}, \ldots, Q_{j_\kappa} \rangle$, where $M(i, j_k) \neq 0 (1 \leqslant k \leqslant \kappa)$.
    2.2 List the message sizes appearing in $M$ in the descending order as $m_1 \geqslant m_2 \geqslant \cdots \geqslant m_l$.
    2.3 Initialization: $S_1(i) = \perp, S_2(j) = \perp (0 \leqslant i < p-1, 0 \leqslant j < q-1)$, $\gamma \leftarrow 0, k \leftarrow 1$.
Step 3. For each processor $P_i$ that satisfies $k < \kappa$ do the following steps.
Step 4. $n \leftarrow 1$.
Step 5. $\gamma \leftarrow \gamma + 1$;
        for $h = 1, 2, \ldots, \frac{p}{s}$ do
        $\forall c_{\alpha\beta} = m_\gamma \in C_{h,(n+h-1) \bmod \frac{q}{t}}$,
        $i \leftarrow \alpha + (h-1) * s$, $j \leftarrow \beta + (h-1) * t$;
        if $S_1(i) \neq \top \wedge S_2(j) \neq \top$ then $Q_{j_k} \leftarrow j$;
        $S_1(i) \leftarrow \top, S_2(j) \leftarrow \top$.
Step 6. If $\gamma < l$ go to step 5.
Step 7. $n \leftarrow n + 1, \gamma \leftarrow 0$.
Step 8. Repeat the steps 5, 6, and 7 until $n \geqslant \frac{p}{s}$.
Step 9. If $k < \kappa, k \leftarrow k + 1$, go to step 3.
Step 10. The $CS_i$ of the $n$th large step is $(\langle Q_{i,1}, \ldots, Q_{i,k} \rangle + t * (n-1)) \bmod q$.
```

Steps 1 and 2 are the initialization of the algorithm. Step 3 to step 9 form the core of the algorithm. Step 5 determines the scheduling in the small steps of a block. n is the count for large steps and γ is the count for the message sizes. $S_1(i)$ and $S_2(j)$ are used for deciding whether processor Q_j is served as a receiving processor for P_i.

Back to the motivating Example 2. The communication table M can be divided into 6 sub-blocks:

$$M = \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \end{bmatrix}, \quad C_{ij} = \begin{bmatrix} 2 & 0 \\ 1 & 1 \\ 0 & 2 \end{bmatrix}.$$

Furthermore, we have $m_1 = 2, m_2 = 1, s = 3, t = 2$. Using Algorithm 2, we obtain $CS_0 = \langle 0, 2, 4 \rangle, CS_1 = \langle (45), (01), (23) \rangle, CS_2 = \langle 1, 3, 5 \rangle, CS_3 = \langle 2, 4, 0 \rangle, CS_4 = \langle (54), (10), (32) \rangle, CS_5 = \langle 3, 5, 1 \rangle$. The scheduling result is represented as Fig. 2(c).

5. Experimental results

For purpose of performance evaluation of our optimized communication scheduling algorithms, we present the experimental evaluation for these techniques. All the experiments are implemented on two platforms: one is CP-PACS Pilot3 [18], a 64-processor subset of an MIMD distributed memory parallel computer developed at the University of Tsukuba, and another is a workstation cluster. The node programs are written in C++, using MPI communication library and system call `MPI_Wtime()` for measuring execution time. The single-precision array was used for the test.

5.1. Communication cost model

Modeling the communication cost for the all-to-many personalized communication for array redistribution requires accounting for the message startup and transmission costs and the overhead arising due to node and link contention. If the message startup cost is T_s, the network bandwidth is $\frac{1}{T_d}$, then the communication cost in one step transmitting m data items can be modeled as

$$T_{\text{comm}} = T_s + m * T_d.$$

Notice $T_s \gg T_d$ for most parallel architectures. If the waiting time invoked by node contention is considered, the communication cost for one step may change to $T_{\text{comm}} = N * (T_s + m * T_d)$, where N is the maximum number of sending processors which send messages to the same processor. On the other hand, if the communication steps in redistribution routines try to avoid the node contention, but do not consider the scheduling involved in message lengths, all communication cost may estimated as

$$T_{\text{comm}} = K * (T_s + \max(m_1, m_2, \ldots, m_K) * T_d),$$

where K is the communication steps and $m_1, \ldots, m_K$ are the message size for each step, respectively. If the algorithms proposed in this paper are used, the cost is approximately as

$$T_{\text{comm}} = K * T_s + (m_1 + m_2 + \cdots + m_K) * T_d.$$

From the above two formulas for communication cost estimation , theoretically we can estimate our algorithms proposed in this paper should be better than contention-free algorithms since $(m_1 + m_2 + \cdots + m_K) \leqslant K * \max(m_1, m_2, \ldots, m_K)$. The experiments in the following subsections are used to confirm this observation.

5.2. Experiments for comparison with naive redistribution routine and contention-free algorithm

CP-PACS (Computational Physics by Parallel Array Computer System) is a massively parallel processor developed and in full operation at the Center for Computational Physics at the University of Tsukuba. It is an MIMD machine with a distributed memory, equipped with 2048 processing units and 128 GB of main memory. The theoretical peak performance of CP-PACS is 614.4 Gflops. CP-PACS achieved 368.2 Gflops with the Linpack benchmark in 1996, which at that time was the fastest Gflops rating in the world. CP-PACS has two remarkable features. Pseudo Vector Processing feature (PVP-SW) on each node processor, which can perform high-speed vector processing on a single chip superscalar microprocessor; and a three-dimensional Hyper-Crossbar (3-D HXB) interconnection network, which provides high-speed and flexible communication among node processors.

Fig. 4 shows the communication time (ms) of redistribution from *cyclic*(4) to *cyclic*(3), with different local sizes on 5 processors (Example 1). The reason that we selected this scheme as our experimental example is to show how our algorithm can improve communication performance for the case of inter-processor communication with different message lengths in a communication step. Three versions of redistribution are compared in our experiments. The curve marked with “general” is a naive redistribution algorithm [4] which is a simple algorithm as shown in the beginning of Section 2. It is implemented in a runtime library of an optimized C++ compiler version 02-06-C installed in CP-PACS (Pilot3); the curve marked with “align-message” is our scheduling algorithm proposed in this paper, and the curve marked with “contention-free” is the scheduling algorithm only avoiding the node contention, proposed in [5]. Since we are mainly concerned with the communication cost, the communication times were only measured in the redistribution routines. In this experimental example, the algorithm

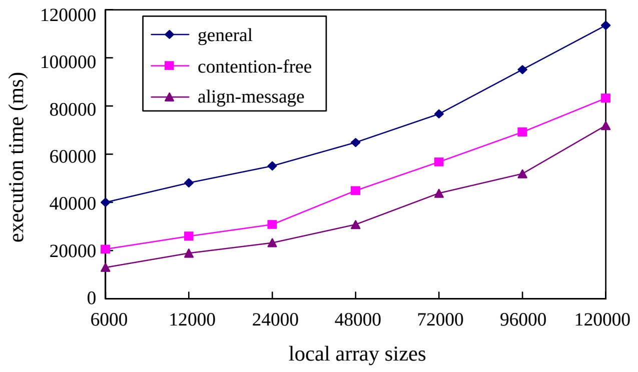


Fig. 4. Comparing communication time on the CP-PACS Pilot3 for *cyclic*(4) to *cyclic*(3),$p = q = 5$.

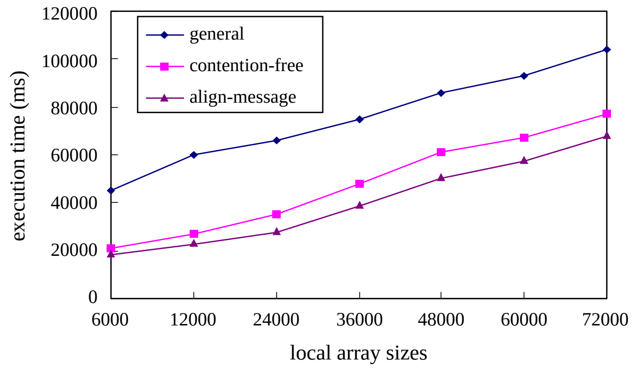


Fig. 5. Comparing communication time on the CP-PACS Pilot3 for *cyclic*(3) to *cyclic*(2), $p = q = 6$.

“align-message” minimizes the waiting time. The waiting time is approximately equal to 1/5 of the communication time used to transfer a message in each step. We can observe that about 15% speed-up can be achieved due to the existence of the waiting time in the old algorithm. As the message length increases, the speed-up also increases because most of the time is consumed in data transmission.

The second experiment corresponds to motivating Example 2. In this experiment, the communication primitive `MPI_Isend` and `MPI_Irecv` are used for communication between two large steps. The results are shown in Fig. 5. Compared to the contention-free algorithm, because the message unit is reduced from 9 to 6, about 1/3 of the waiting time can be cut down.

The third experiment tries to demonstrate the efficacy and scalability of our proposed algorithm when the numbers of source and destination processors are different. We selected a global array size of 120,000, a redistribution from *cyclic*(8) to *cyclic*(6), and the number of source processors ranging from 5, 10, 20, 30, 40, 50, to 60 and the number of destination processors is immutably 5. From Fig. 6, we observe that if there is no scheduling, the execution time is not improved although the local array size is decreased as the number of source processors is increased. Using the proposed

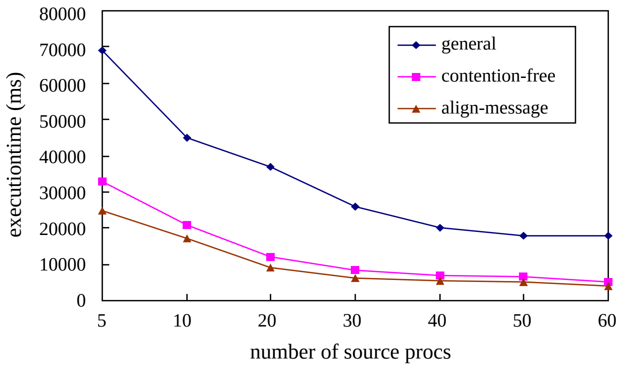


Fig. 6. Comparing communication time on the CP-PACS Pilot3 for *cyclic*(8) to *cyclic*(6), global array size is 120, 000, and different number of source processors.

algorithm in this paper, the performance is better in all the cases of different number of source processors.

These figures show that the algorithms proposed in this paper achieve better performance than some previously existing redistribution algorithms. The performance improvement becomes more appreciable as the message length increases. This means that it is vital to use communication scheduling with aligning at message sizes in each communication step of redistribution routines.

5.3. *Experiments for comparison with* `MPI_Alltoallv`

In principle, as mentioned above, all-to-all redistribution with the same length of messages can be implemented using `MPI_Alltoall`, and all-to-all redistribution with the different length of messages can be implemented using `MPI_Alltoallv`, where parameters `send_count[]`, `send_displacements[]`, `recv_count[]`, and `recv_displacements[]` can specify the different message lengths and displacements in the sending and receiving phases, respectively.

In order to measure the performance in different platforms and compare our algorithms with MPI library routines, we also implemented our approach in a homogenous workstation (WS) cluster. This WS cluster system consists of 24 SUN workstations with 400 Hz CPU, and with 128 M memory on each node. All nodes are connected by a 100Base-T Ethernet via a switching hub. The OS is Solaris 8 and MPI library is MPICH Ver.1.1.2 developed by the Argonne National Laboratory.

We evaluated Algorithm 1 and Algorithm 2 described in Sections 4.1 and 4.2, contention-free algorithm, and `MPI_Alltoallv` for the same local array sizes. Fig. 7 presents the results for *cyclic*(6) to cyclic(8) on 5 source and target processors. In this figure, Algorithm 1 (marked as “align-message”), the contention-free algorithm (marked as “contention-free”) and `MPI_Alltoallv` (marked as “MPI_Alltoallv”) were measured. The measured execution time of Algorithm 1 and the contention-free al-

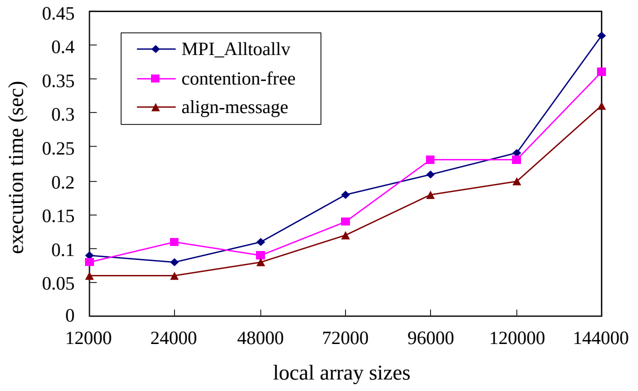


Fig. 7. Comparing communication time on the Sun Workstation cluster for *cyclic*(6) to *cyclic*(8), $p = q = 5$.

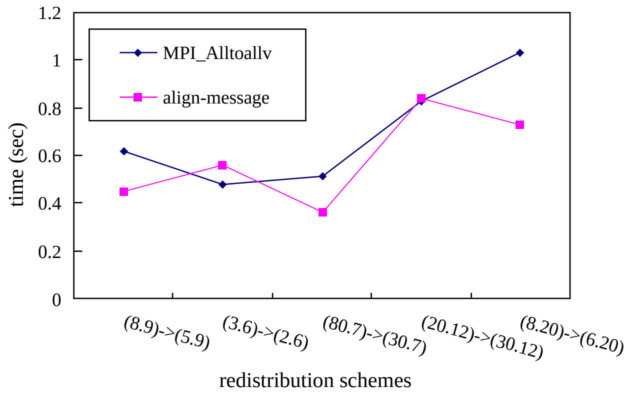


Fig. 8. Comparing communication time on the Sun Workstation cluster for the different redistribution schemes with local array size $N = 120, 000$.

gorithm include dynamic displacement calculation and packing and unpacking for each communication step. Despite of this overhead, our methods are still better than `MPI_Alltoallv` in most cases, especially Algorithm 1 can approximately get 20% performance improvement. This reason is in our experimental environment of MPI, the implementation of `MPI_Alltoallv` do not avoid communication bottleneck. It should be noted that our algorithms can be tuned further on packing/unpacking and displacement calculation.

Fig. 8 shows another comparison for our methods with `MPI_Alltoallv`. In this experiment, the different redistribution schemes are implemented. Strictly, `MPI_Alltoallv` cannot be invoked if the number of source and target processors are different (although some data length between source and target processors can be set to 0 so that the real communication does not occur among the part of source and target processors, the start-up time are still wasted. Thus it influences the total communication overhead). We only measured some schemes with the same source and target processors ($(x, p) \to (y, p)$ marked in x-axis means that redistribute from *cyclic*(x) to *cyclic*(y) on processor p). (We also experimented the schemes with different number of source and target processors using our algorithms which are not shown in the paper.) Furthermore, the contention-free algorithm cannot be adapted in the schemes listed in the figure. Thus, we only measured Algorithm 1, Algorithm 2 and `MPI_Alltoallv` for local array size $N = 120,000$, where $(8, 9) \to (5, 9)$, $(80, 7) \to (30, 7)$, and $(8, 20) \to (6, 20)$ are implemented with Algorithm 1, while $(3, 6) \to (2, 6)$ and $(20, 12) \to (30, 12)$ are implemented with Algorithm 2. From the figure we observe that Algorithm 2 is a little worse than `MPI_Alltoallv` due to more complicated displacement calculation used in the algorithm. However, in all schemes which is suitable for Algorithm 1, better performance is achieved in Algorithm 1 than in `MPI_Alltoallv`.

6. Conclusions

In this paper, we have shown an efficient approach for communication scheduling in redistribution routines. Our scheduling algorithms not only consider avoiding the node contention when data are received from various sending processors, but also arrange messages of the same lengths as much as possible in a communication step. The communication scheduling results in a permutation of the destination processors in each communication step. Furthermore, if the messages of same length cannot be put into a communication step no matter how to schedule, the barriers are inserted between two "large" steps. Because the sum of message lengths in each large step is identical to each other, and through using non-blocking communication in each large step, the minimum communication idle time can be achieved. The algorithms proposed in this paper can guarantee a lower communication overhead in a redistribution process. These methods are also useful in the implementation of MPI routine `MPI_Alltoallv`.

However, our algorithms only consider improving communication performance on software aspect and do not schedule communication to fit the hardware communication architecture. We have noticed the related work with respect to optimization of multicast, all-to-all communication in hypercube, mesh, torus and other network topologies [16,17]. There are different scheduling strategies for different multicomputer architectures. In the future, we will extend our algorithms to consider the aspect of hardware support, such as various communication topologies.

Acknowledgments

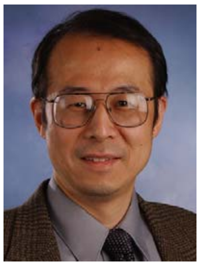
This research was supported in part by the Japanese Okawa Foundation for Information and Telecommunications under Grant Program 01-12, and by the US National Science Foundation under Grant ECS-0196569. The authors thank the four anonymous reviewers for their careful reading of the manuscript and many insightful comments and suggestions.

References

[1] R. Bixby, K. Kennedy, U. Kremer, Automatic data layout using 0-1 integer programming, Proceedings of the 1994 International Conference on Parallel Archs. and Compilation Techniques, Montreal, Canada, August 1994.
[2] Y. Chung, C. Hsu, S. Bai, A basic-cycle calculation technique for efficient dynamic data redistribution, IEEE Trans. Parallel Distrib. Systems 9 (4) (1998) 359–377.
[3] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, Y. Robert, Scheduling block-cyclic array redistribution, IEEE Trans. Parallel Distrib. Systems 9 (2) (1998) 192–205.
[4] M. Guo, Y. Yamashita, I. Nakata, Efficient implementation of multi-dimensional array redistribution, IEICE Trans. Inform. Systems E81-D (11) (1998) 1195–1204.
[5] M. Guo, I. Nakata, Y. Yamashita, Contention-free communication scheduling for array redistribution, Parallel Comput. 25 (3) (2000).
[6] HPF Forum, High Performance Fortran Language Specification, version 2.0 edition, Rice University, Houston, Texas, November 1996.
[7] C. Hsu, S. Bai, Y. Chung, C. Yang, A generalized basic-cycle calculation method for efficient array redistribution, IEEE Trans. Parallel Distrib. Systems 11 (12) (2000) 1201–1216.
[8] C. Hsu, Y. Chung, D. Yang, C. Dow, A generalized processor mapping technique for array redistribution, IEEE Trans. Parallel Distrib. Systems 12 (7) (2001) 743–757.
[9] S.D. Kaushik, C.-H. Huang, R.W. Johmson, P. Sadayappan, An approach to communication-efficient data redistribution, Proceedings of the Eighth ACM International Conference on Supercomputing, Manchester, UK, July 1994.
[10] S.D. Kaushik, C.-H. Huang, J. Ramanujam, P. Sadayappan, Multi-phase redistribution: a communication-efficient approach to array redistribution, Technical report, The Ohio Sate University, 1995.
[11] K. Kennedy, U. Kremer, Automatic data layout for high performance fortran, Proceedings of Supercomputing'95, San Diego, CA, December 1995.
[12] U. Kremer, NP-completeness of dynamic remapping, Proceedings of the Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands, December 1993.
[13] E.T. Kalns, L.M. Ni, Processor mapping techniques toward efficient data redistribution, IEEE Trans. Parallel Distrib. Systems 6 (12) (1995) 1234–1247.
[14] Y.W. Lim, P.B. Bhat, V. Prasanna, Efficient algorithms for block-cyclic redistribution of arrays, IEEE Symposium on Parallel and Distributed Processing, October 1996.
[15] Y.W. Lim, N. Park, V. Prasanna, Efficient algorithms for multi-dimensional block-cyclic redistribution of arrays, Proceedings of the 26th International Conference on Parallel Processing, Bloomingdale, IL, August 1997.
[16] P.K. McKinley, H. Xu, A.-H. Esfahanian, L.M. Ni, Unicast-based multicast communication in wormhole-routed networks, IEEE Trans. Parallel Distrib. Systems 5 (12) (1994) 1252–1265.
[17] D.F. Robinson, D. Judd, P.K. McKinley, B.H.C. Cheng, Efficient multicase in all-port wormhole-routed hypercubes, J. Parallel Distrib. Comput. 31 (1995) 126–140.
[18] K. Nakazawa, H. Nakamura, T. Boku, I. Nakata, Y. Yamashita, CP-PACS: a massively parallel processor at the University of Tsukuba, Parallel Comput. 25 (13–14) (1999) 1635–1661.
[19] Y. Pan, J. Shang, M. Guo, A scalable HPF implementation of a finite volume CEM application on a CRAY T3E parallel system, Concurrency Comput.: Pract. Exp. 15 (6) (2003) 607–621.
[20] N. Park, V.K. Prasanna, C.S. Raghavendra, Efficient algorithms for block-cyclic array redistribution between processor sets, IEEE Trans. Parallel Distrib. Systems 10 (12) (1999) 1217–1239.

[21] D.J. Palermo, P. Banerjee, Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers, Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing, August 1995.
[22] D.J. Palermo, E.W. Hodges IV, P. Banerjee, Dynamic data partitioning for distributed-memory multicomputers, J. Parallel Distrib. Comput. (38) (1996) 158–175.
[23] S. Ramaswamy, B. Simons, P. Banerjee, Optimizations for efficient array redistribution on distributed memory multicomputers, J. Parallel Distrib. Comput. 38 (1996) 217–228.
[24] S. Ranka, J-C. Wang, G. Fox, Static and run-time algorithms for all-to-many personalized communication on permutation networks, IEEE Trans. Parallel Distrib. Systems 5 (12) (1994) 1266–1274.
[25] L. Prylli, B. Tourancheau, Efficient block-cyclic data redistribution, in: EuroPar'96, Volume of Lectures Notes in Computer Science, Springer, Berlin, 1996. pp. 155–164.
[26] R. Thakur, A. Choudhary, J. Ramanujam, Efficient algorithms for array redistribution, IEEE Trans. Parallel Distrib. Systems 7 (6) (1996) 587–593.
[27] R. Thakur, A. Choudhary, G. Fox, Runtime array redistribution in HPF programs, in: Proceedings of the Scalable High Performance Computing Conference, May 1994, pp. 309–316.

Dr. Minyi Guo received his Ph.D. degree in information science from University of Tsukuba, Japan in 1998. From 1998 to 2000, Dr. Guo had been a research scientist of NEC Soft, Ltd. Japan. He is currently a professor at the Department of Computer Software, The University of Aizu, Japan. From 2001 to 2004, he was a visiting professor of Georgia State University, USA, Hong Kong Polytechnic University, and University of New South Wales, Australia. Dr. Guo has served as general chair, program committee or organizing committee chair for many international conferences. He is the editor-in-chief of the International Journal of Embedded Systems. He is also in editorial board of International Journal of High Performance Computing and Networking, Journal of Embedded Computing, Journal of Parallel and Distributed Scientific and Engineering Computing, and International Journal of Computer and Applications. Dr. Guo's research interests include parallel and distributed processing, parallelizing compilers, data parallel languages, data mining, molecular computing and software engineering. He is a member of the ACM, IEEE, IEEE Computer Society, and IEICE. He is listed in Marquis Who's Who in Science and Engineering.



Dr. Yi Pan entered Tsinghua University in March 1978 with the highest college entrance examination score among all 1977 high school graduates in Jiangsu Province. Dr. Pan received his B.Eng. and M.Eng. degrees in computer engineering from Tsinghua University, China, in 1982 and 1984, respectively, and his Ph.D. degree in computer science from the University of Pittsburgh, USA, in 1991. Currently, he is a Yamacraw professor in the Department of Computer Science at Georgia State University.
Dr. Pan's research interests include parallel and distributed computing, optical networks, wireless networks, and bioinformatics. Dr. Pan has published more than 80 journal papers with 27 papers published in various IEEE journals. In addition, he has published over 90 papers in refereed conferences (including IPDPS, ICPP, ICDCS, INFOCOM, and GLOBECOM). He has also co-edited 13 books (including proceedings) and contributed several book chapters. His pioneer work on computing using reconfigurable optical buses has inspired extensive subsequent work by many researchers, and his research results have been cited by more than 100 researchers worldwide in books, theses, journal and conference papers. He is a co-inventor of three U.S. patents (pending) and 5 provisional patents, and has received many awards from agencies such as NSF, AFOSR, JSPS, IISF and Mellon Foundation. His recent research has been supported by NSF, NIH, NSFC, AFOSR, AFRL, JSPS, IISF and the states of Georgia and Ohio. He has served as a reviewer/panelist for many research foundations/agencies such as the U.S. National Science Foundation, the Natural Sciences and Engineering Research Council of Canada, the Australian Research Council, and the Hong Kong Research Grants Council. Dr. Pan has served as an editor-in-chief or editorial board member for 8 journals including 3 IEEE Transactions and a guest editor for 7 special issues. He has organized several international conferences and workshops and has also served as a program committee member for several major international conferences such as INFOCOM, GLOBECOM, ICC, IPDPS, and ICPP.
Dr. Pan has delivered over 50 invited talks, including keynote speeches and colloquium talks, at conferences and universities worldwide. Dr. Pan is an IEEE Distinguished Speaker (2000-2002), a Yamacraw Distinguished Speaker (2002), a Shell Oil Colloquium Speaker (2002), and a senior member of IEEE. He is listed in Men of Achievement, Who's Who in Midwest, Who's Who in America, Who's Who in American Education, Who's Who in Computational Science and Engineering, and Who's Who of Asian Americans.