A Generalized Basic-Cycle Calculation Method for Efficient Array Redistribution

Ching-Hsien Hsu, Sheng-Wen Bai,
Yeh-Ching Chung, *Member*, *IEEE Computer Society*, and Chu-Sing Yang

Abstract—In many scientific applications, dynamic array redistribution is usually required to enhance the performance of an algorithm. In this paper, we present a *generalized basic-cycle calculation* (*GBCC*) method to efficiently perform a BLOCK-CYCLIC(*s*) over *P* processors to BLOCK-CYCLIC(*t*) over *Q* processors array redistribution. In the *GBCC* method, a processor first computes the source/destination processor/data sets of array elements in the first generalized basic-cycle of the local array it owns. A generalized basic-cycle is defined as $lcm(sP,\ tQ)/(gcd(s,t) \times P)$ in the source distribution and $lcm(sP,\ tQ)/(gcd(s,t) \times Q)$ in the destination distribution. From the source/destination processor/data sets of array elements in the first generalized basic-cycle, we can construct packing/unpacking pattern tables to minimize the data-movement operations. Since each generalized basic-cycle has the same communication pattern, based on the packing/unpacking pattern tables, a processor can pack/unpack array elements efficiently. To evaluate the performance of the *GBCC* method, we have implemented this method on an IBM SP2 parallel machine, along with the *PITFALLS* method and the *ScaLAPACK* method. The cost models for these three methods are also presented. The experimental results show that the *GBCC* method outperforms the *PITFALLS* method and the *ScaLAPACK* method for all test samples. A brief description of the extension of the *GBCC* method to multidimensional array redistributions is also presented.

Index Terms—Redistribution, generalized basic-cycle calculation method, distributed memory multicomputers.

1 Introduction

The data-parallel programming model has become a widely accepted paradigm for programming distributed-memory parallel computers. To efficiently execute a data-parallel program on a distributed memory multicomputer, appropriate data decomposition is necessary. Many data-parallel programming languages such as High Performance Fortran (HPF) [7], Fortran D [2], and High Performance C (HPC) [27] provide compiler directives for programmers to specify regular array distribution, namely, BLOCK, CYCLIC, and BLOCK-CYCLIC. Fig. 1 shows examples of these three array distributions.

Dongarra et al. [5] have shown that the above distributions are essential for many dense matrix algorithms design in distributed memory machines. Many methods were proposed to address the problems of the communication sets identification for array statements with BLOCK-CYCLIC(*c*) distribution [1], [5], [7], [12], [13], [14], [15], [21], [24], [25]. However, in many scientific programs, such as multidimensional Fast Fourier Transform [28], the Alternative Direction Implicit (ADI) method for solving two-dimensional diffusion equations, linear algebra solvers [19], etc., it is necessary to change distribution fashion of a program at different phases in order to achieve a better performance. Since array redistribution is performed at run-time, there is a performance trade-off between the efficiency of the new data distribution for a subsequent phase of an algorithm and the cost of redistributing array among processors. Thus, efficient methods for performing array redistribution are of great importance for the development of distributed memory compilers for data-parallel programming languages.

Given a redistribution of BLOCK-CYCLIC(*s*) over *P* processors to BLOCK-CYCLIC(*t*) over *Q* processors on a one-dimensional array with *N* elements, in general, the redistribution can be performed in two phases, the send phase and the receive phase. In the send phase, a processor P_i has to determine all the data sets that it needs to send to other processors (destination processors), pack those data sets into messages, and send messages to their destination processors. In the receive phase, a processor P_j has to determine all the data sets that it needs to receive from other processors (source processors), receive messages from source processors, and unpack elements in messages to their corresponding local array positions. We called these three steps in the send/receive phase the indexing, the packing/unpacking, and the communication issues of a redistribution, respectively.

Many methods for performing array redistribution have been presented in the literature. In general, they can be classified into three categories according to the redistribution type that they solved.

- General Case Solutions. Methods in this category provide algorithms to perform the redistribution of BLOCK-CYCLIC(*s*) over *P* processors to BLOCK-CYCLIC(*t*) over *Q* processors, where *s*, *t*, *P*, *Q* are positive integers and *P* may not be equal to *Q*. The

- C.-H. Hsu and Y.-C. Chung are with the Department of Information Engineering, Feng Chia University, Taichung, Taiwan 407, ROC. E-mail: chhsu@fhk.edu.tw, ychung@fcu.edu.tw.
- S.-W. Bai and C.-S. Yang are with the Institute of Computer and Information Engineering, National Sun-Yet-San University, Kaohsiung 804, ROC. E-mail: {swbai, csyang}@cie.nsysu.edu.tw.

Manuscript received 01 June 1998; revised 15 Aug. 2000; accepted 17 Aug. 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 106936.

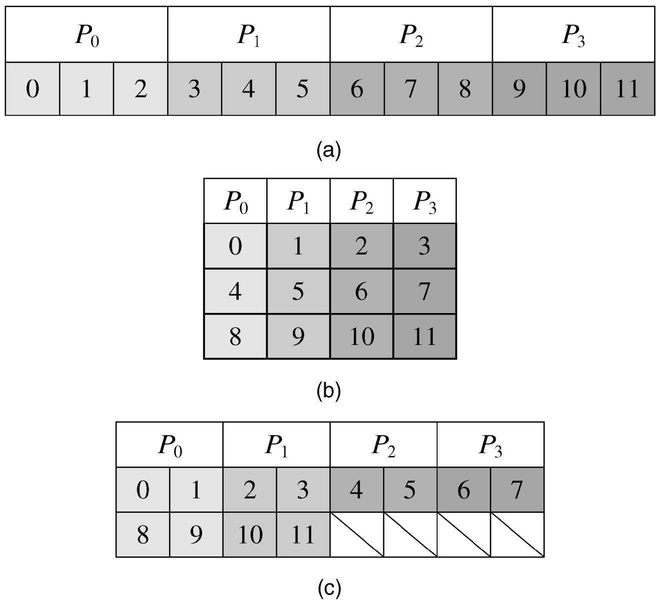

Fig. 1. Examples of regular array distributions. (a) A BLOCK distribution, (b) a CYCLIC distribution, and (c) a BLOCK-CYCLIC(2) distribution on an array with 12 elements over four processors.

PITFALLS [20], [21] and the *ScaLAPACK* [19] methods are two examples. They pay more attention on the indexing and the packing/unpacking issues.

- Special Case Solutions. Methods in this category assume that the redistribution of an array is under the same source/destination processor set, $P = Q$. In general, they provide algorithms to generate the communication sets for some specific type of redistribution, such as BLOCK to CYCLIC redistribution [3], BLOCK-CYCLIC(kr) to BLOCK-CYCLIC(r) redistribution [23], [24], and BLOCK-CYCLIC(s) to BLOCK-CYCLIC(t) redistribution [4], where k, r, s, t are positive integers. The BLOCK-CYCLIC(s) to BLOCK-CYCLIC(t) redistribution is the most general case in this category. Methods in this category pay more attention on the indexing and the packing/unpacking issues.
- Communication Optimization Solutions. In general, methods in this category provide different approaches to reduce the communication overheads in a redistribution. Examples are the processor mapping technique [9], [10], the multiphase redistribution technique [11], [12], the communication scheduling approaches [17], [18], [29], the strip mining approach [30], and the spiral mapping method [31]. Methods in this category pay more attention on the communication issue.

In this paper, we want to provide an efficient method for array redistributions in the category of General Case Solutions. For the *PITFALLS* method, the main idea is to find all intersections between source and target distributions. Based on the intersections, the send/receive processor/data sets can be determined and general redistribution algorithms can be devised. It uses the repetitive pattern in communication sets calculation. The disadvantage of this approach is that the number of iterations of the outermost loop in the FALLS intersection algorithm depends on the number of processors. When the number of processor is large, it may lead to high indexing overheads and degrades the performance of a redistribution algorithm. The *ScaLAPACK* method is similar to the *PITFALLS* method but has simpler indexing calculation than that of the *PITFALLS* method. In addition, both methods did not minimize the data-movement operations when packing/unpacking array elements. This also leads to high packing/unpacking costs for some cases.

To overcome the drawbacks of the *PITFALLS* method and the *ScaLAPACK* method, we propose a *generalized basic-cycle calculation* (*GBCC*) method. The *GBCC* method provides a fast indexing technique in which a processor first computes the source/destination processor/data sets of array elements in the first generalized basic-cycle of the local array it owns. A generalized basic-cycle is defined as $lcm(sP,\ tQ)/(gcd(s,\ t) \times P)$ in the source distribution and $lcm(sP,\ tQ)/(gcd(s,\ t) \times Q)$ in the destination distribution. From the source/destination processor/data sets of array elements in the first generalized basic-cycle, the *GBCC* method constructs packing/unpacking pattern tables that can optimize the data-movement operations. Based on the packing/unpacking pattern tables, a processor can pack/unpack array elements efficiently. The generalized basic-cycle calculation (*GBCC*) technique has the following characteristics:

- It is a simple method to perform the general BLOCK-CYCLIC(s) over P processors to BLOCK-CYCLIC(t) over Q processors array redistribution.
- The indexing overhead of the generalized basic-cycle calculation technique is very small and independent of the array size involved in a redistribution.
- It minimizes the data-movement operations when packing/unpacking array elements.
- The generalized basic-cycle calculation technique uses an asynchronous communication scheme to overlap the computation and the communication. This leads to a better performance for a redistribution.
- It can be easily extended to handle multidimensional array redistributions.

To evaluate the performance of the *GBCC* method, we have implemented this method on an IBM SP2 parallel machine, along with the *PITFALLS* and the *ScaLAPACK* methods. Both theoretical analysis and experimental results were conducted for these three methods. The theoretical analysis shows that the indexing cost of the *GBCC* method is less than that of the *PITFALLS* and the *ScaLAPACK* methods. The packing/unpacking cost of the *GBCC* method is less than or equal to that of the *PITFALLS* and the *ScaLAPACK* methods. The experimental results show that the *GBCC* method outperforms the *PITFALLS* method and the *ScaLAPACK* method for all test samples.

The paper is organized as follows: In Section 2, we introduce notations and terminology used in this paper. Section 3 presents the *GBCC* method in details. A brief description of the extension of the *GBCC* method to multidimensional array redistributions is also presented in this section. The cost models and performance comparisons

	BLOCK-CYCLIC(10), $P = 3$																			
Local	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
SLA_0	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
SLA_1	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**
SLA_2	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**
Local	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*	*33*	*34*	*35*	*36*	*37*	*38*	*39*
SLA_0	**60**	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**	**90**	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**
SLA_1	**70**	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**	**100**	**101**	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**
SLA_2	**80**	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**	**110**	**111**	**112**	**113**	**114**	**115**	**116**	**117**	**118**	**119**

$\downarrow$

	BLOCK-CYCLIC(3), $Q = 4$														
Local	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
DLA_0	**0**	**1**	**2**	**12**	**13**	**14**	**24**	**25**	**26**	**36**	**37**	**38**	**48**	**49**	**50**
DLA_1	**3**	**4**	**5**	**15**	**16**	**17**	**27**	**28**	**29**	**39**	**40**	**41**	**51**	**52**	**53**
DLA_2	**6**	**7**	**8**	**18**	**19**	**20**	**30**	**31**	**32**	**42**	**43**	**44**	**54**	**55**	**56**
DLA_3	**9**	**10**	**11**	**21**	**22**	**23**	**33**	**34**	**35**	**45**	**46**	**47**	**57**	**58**	**59**
Local	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*
DLA_0	**60**	**61**	**62**	**72**	**73**	**74**	**84**	**85**	**86**	**96**	**97**	**98**	**108**	**109**	**110**
DLA_1	**63**	**64**	**65**	**75**	**76**	**77**	**87**	**88**	**89**	**99**	**100**	**101**	**111**	**112**	**113**
DLA_2	**66**	**67**	**68**	**78**	**79**	**80**	**90**	**91**	**92**	**102**	**103**	**104**	**114**	**115**	**116**
DLA_3	**69**	**70**	**71**	**81**	**82**	**83**	**93**	**94**	**95**	**105**	**106**	**107**	**117**	**118**	**119**

Fig. 2. A $(10, 3) \rightarrow (3, 4)$ redistribution on a one-dimensional array with N = 120 elements.

of the *GBCC* method, the *PITFALLS* method, and the *ScaLAPACK* method are given in Section 4.

2 Preliminaries

To simplify the presentation, we use $(s, P) \rightarrow (t, Q)$ to represent the redistribution of BLOCK-CYCLIC(s) over P processors to BLOCK-CYCLIC(t) over Q processors and N denotes the global array size for the rest of the paper. We also assume that all array elements and processors are indexed starting from 0.

Definition 1. *Given a $(s, P) \rightarrow (t, Q)$ redistribution, BLOCK-CYCLIC(s), BLOCK-CYCLIC(t), s, t, P, and Q are called the* source distribution, *the* destination distribution, *the* source distribution factor, *the* destination distribution factor, *the* number of source processors, *and the* number of destination processors *of the redistribution, respectively.*

Definition 2. *Given a $(s, P) \rightarrow (t, Q)$ redistribution on a one-dimensional array $A[0 : N - 1]$, the* source local array *of processor P_i, denoted by $SLA_i[0 : N/P - 1]$, is defined as the set of array elements that are distributed to processor P_i in the source distribution, where $i = 0$ to $P - 1$. The* destination local array *of processor Q_j, denoted by $DLA_j[0 : N/Q - 1]$, is defined as the set of array elements that are distributed to processor Q_j in the destination distribution, where $j = 0$ to $Q - 1$.*

Definition 3. *Given a $(s, P) \rightarrow (t, Q)$ redistribution on a one-dimensional array $A[0 : N - 1]$, the* source processor *of an array element in $A[0 : N - 1]$ or $DLA_j[0 : N/Q - 1]$ is defined as the processor that owns the array element in the source distribution, where $j = 0$ to $Q - 1$. The* destination processor *of an array element in $A[0 : N - 1]$ or $SLA_i[0 : N/P - 1]$ is defined as the processor that owns the array element in the destination distribution, where $i = 0$ to $P - 1$.*

Definition 4. *Given integers a and b, their least common multiple and greatest common divisor are denoted as lcm(a, b) and gcd(a, b), respectively.*

Definition 5. *Given a $(s, P) \rightarrow (t, Q)$ redistribution on a one-dimensional array $A[0 : N - 1]$, the generalized basic-cycle (GBC) is defined as*

$$GBC = \frac{lcm(s \times P, t \times Q)}{gcd(s, t) \times P}$$

in the source distribution and

$$GBC = \frac{lcm(s \times P, t \times Q)}{gcd(s, t) \times Q}$$

in the destination distribution. We define $SLA_i[0 : GBC - 1]$ ($DLA_j[0 : GBC - 1]$) as the first generalized basic-cycle of a source (destination) local array of processor P_i (Q_j), $SLA_i[GBC : 2 \times GBC - 1]$ ($DLA_j[GBC : 2 \times GBC - 1]$) as the second basic-cycle of a source (destination) local array of processor P_i (Q_j), etc.

Definition 6. *Given a $(s, P) \rightarrow (t, Q)$ redistribution, a generalized basic-cycle of a source (destination) local array can be divided into GBC/s (GBC/t) blocks. We define those blocks as the* source (destination) sections *of a generalized basic-cycle of a source (destination) local array.*

We now give an example to clarify the above definitions. Fig. 2 shows a $(10, 3) \rightarrow (3, 4)$ redistribution on a one-dimensional array with N = 120 elements, $A[0 : 119]$. The local array indices are represented as italic numbers while the global array indices are represented as bold numbers. According to Definition 5, we know that the generalized basic-cycle in the source distribution is 20. The generalized basic-cycle in the destination distribution is 15. The first generalized basic-cycle in SLA_1 of source processor P_1 is $SLA_1[0 : 19] = \{A[10], \ldots, A[19], A[40], \ldots, A[49]\}$.

	Source: BLOCK-CYCLIC(4)															
local	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
P_0	**0**	**1**	**2**	**3**	**12**	**13**	**14**	**15**	**24**	**25**	**26**	**27**	**36**	**37**	**38**	**39**
P_1	**4**	**5**	**6**	**7**	**16**	**17**	**18**	**19**	**28**	**29**	**30**	**31**	**40**	**41**	**42**	**43**
P_2	**8**	**9**	**10**	**11**	**20**	**21**	**22**	**23**	**32**	**33**	**34**	**35**	**44**	**45**	**46**	**47**

↓

	Destination: BLOCK-CYCLIC(3)																							
Local	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*
Q_0	**0**	**1**	**2**	**6**	**7**	**8**	**12**	**13**	**14**	**18**	**19**	**20**	**24**	**25**	**26**	**30**	**31**	**32**	**36**	**37**	**38**	**42**	**43**	**44**
Q_1	**3**	**4**	**5**	**9**	**10**	**11**	**15**	**16**	**17**	**21**	**22**	**23**	**27**	**28**	**29**	**33**	**34**	**35**	**39**	**40**	**41**	**45**	**46**	**47**

Fig. 3. A $(4,3) \rightarrow (3,2)$ redistribution on a one-dimensional array with $N = 48$ elements.

$SLA_1[0:19]$ can be divided into two source sections (size = 10), $SLA_1[0:9]$ and $SLA_1[0:19]$. The second generalized basic-cycle in SLA_1 of source processor P_1 is $SLA_1[20:39] = \{A[70], \ldots, A[79], A[100], \ldots, A[109]\}$. In the destination distribution, the first generalized basic-cycle in DLA_1 of destination processor Q_1 is

$$\begin{aligned} &DLA_1[0:14] \\ &\quad = \{A[3], \ldots, A[5], A[15], \ldots, A[17], A[27], \ldots, \\ &\qquad A[29], A[39], \ldots, A[41], A[51], \ldots, A[53]\}. \end{aligned}$$

$DLA_1[0:14]$ can be divided into five destination sections (size= 3): $DLA_1[0:2], DLA_1[3:5], DLA_1[6:8], DLA_1[9:11]$, and $DLA_1[12:14]$. The second generalized basic-cycle of destination processor Q_1 is

$$\begin{aligned} &DLA_1[15:29] \\ &\quad = \{A[63], \ldots, A[65], A[75], \ldots, A[77], A[87], \ldots, \\ &\qquad A[89], A[99], \ldots, A[101], A[111], \ldots, A[113]\}. \end{aligned}$$

3 The *GBCC* Method for Array Redistribution

In the following, we will describe how the indexing and packing/unpacking operations can be performed efficiently by the *GBCC* method.

The main idea of the *GBCC* method is based on that every generalized basic-cycle of a local array has the same communication pattern. For example, Fig. 3 shows a $(4,3) \rightarrow (3,2)$ redistribution on a one-dimensional array with 48 elements. According to Definition 5, the generalized basic-cycle in the source distribution and the destination distribution of the redistribution is four and six, respectively. In Fig. 3, the local array indices are represented as italic numbers while the global array indices are represented as normal numbers. There are four generalized basic-cycles in each source/destination local array. For each source (destination) local array, array elements in the *k*th position of each generalized basic-cycle have the same destination (source) processor, i.e., all of them will be sent to (received from) the same destination (source) processor during the redistribution, where $k = 0$ to $GBC - 1$. This observation shows that each generalized basic-cycle of a local array has the same communication pattern.

Another example of a $(6,4) \rightarrow (4,3)$ redistribution on $A[0:95]$ is shown in Fig. 4a. The generalized basic-cycle in the source distribution and the destination distribution of the redistribution is three and four, respectively. However, the observation that we obtained from Fig. 3 (each generalized basic-cycle of a local array has the same communication pattern) cannot be applied to the case shown in Fig. 4a directly. For example, the destination processors of the second array elements in the first and the second generalized basic-cycles of the source local array of processor P_0 are Q_0 and Q_1, respectively. The reason the observation cannot be applied directly is that the value of $gcd(6,4)$ is not equal to one. By grouping every $gcd(6,4)$ global array indices of array A to a meta-index, array $A[0:N-1]$ can be transformed to a meta-array $B[0:N/gcd(6,4)-1]$, where $B[k] = \{A[k \times gcd(6,4)], \ldots, A[(k+1) \times gcd(6,4) - 1]\}$ and $k = 0$ to $N/gcd(6,4) - 1$. Then, the observation that we obtained from Fig. 3 can be held if we use array *B* for the redistribution. An example of using meta-array for the array redistribution of Fig. 4a is shown in Fig. 4b.

According to the above analysis, we have the following lemmas.

Lemma 1. *Given a $(s,P) \rightarrow (t,Q)$ redistribution on a one-dimensional array $A[0:N-1]$ and $gcd(s,t) = 1$, for a source (destination) processor $P_i(Q_j)$, if the destination (source) processor of $SLA_i[k](DLA_j[k])$ is Q_j (P_i), then the destination (source) processors of $SLA_i[k+GBC], SLA_i[k+2 \times GBC], \ldots, SLA_i[k+N/P-GBC]$ $(DLA_i[k+GBC], DLA_i[k+2 \times GBC], \ldots, DLA_i[k+N/Q-GBC])$ will also be Q_j (P_i), where $0 \leq k < GBC$ and N/P (N/Q) is a multiple of GBC.*

Proof. We only prove the source processor part. The proof of the destination processor part is similar. In the source distribution,

$$GBC = \frac{lcm(s \times P, t \times Q)}{gcd(s,t) \times P} = \frac{lcm(s \times P, t \times Q)}{P}.$$

For a source processor P_i, if the global array index of $SLA_i[k]$ is α, then the global array indices of $SLA_i[k+GBC], SLA_i[k+2GBC], \ldots,$ and $SLA_i[k+N/P-GBC]$ are $\alpha + lcm(s \times P, t \times Q), \alpha + 2 \times lcm(s \times P, t \times Q), \ldots,$ and $\alpha + (N - lcm(s \times P, t \times Q))$, respectively, where $0 \leq i \leq P-1, 0 \leq k \leq GBC - 1$ and $0 \leq \alpha \leq lcm(s \times P, t \times Q) - 1$. Since $lcm(s \times P, t \times Q)$ is a multiple of $t \times Q$, in the destination distribution, if $A[\alpha]$ is distributed to the

	Source: BLOCK-CYCLIC(6)																							
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
P_0	0	1	2	3	4	5	24	25	26	27	28	29	48	49	50	51	52	53	72	73	74	75	76	77
P_1	6	7	8	9	10	11	30	31	32	33	34	35	54	55	56	57	58	59	78	79	80	81	82	83
P_2	12	13	14	15	16	17	36	37	38	39	40	41	60	61	62	63	64	65	84	85	86	87	88	89
P_3	18	19	20	21	22	23	42	43	44	45	46	47	66	67	68	69	70	71	90	91	92	93	94	95

$\downarrow$

	Destination: BLOCK-CYCLIC(4)																															
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Q_0	0	1	2	3	12	13	14	15	24	25	26	27	36	37	38	39	48	49	50	51	60	61	62	63	72	73	74	75	84	85	86	87
Q_1	4	5	6	7	16	17	18	19	28	29	30	31	40	41	42	43	52	53	54	55	64	65	66	67	76	77	78	79	88	89	90	91
Q_2	8	9	10	11	20	21	22	23	32	33	34	35	44	45	46	47	56	57	58	59	68	69	70	71	80	81	82	83	92	93	94	95

(a)

	Source: BLOCK-CYCLIC(6)											
index	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15	16 17	18 19	20 21	22 23
meta	0	1	2	3	4	5	6	7	8	9	10	11
P_0	0, 1	2, 3	4, 5	24, 25	26, 27	28, 29	48, 49	50, 51	52, 53	72. 73	74, 75	76, 77
P_1	6, 7	8, 9	10, 11	30, 31	32, 33	34, 35	54, 55	56, 57	58, 59	78, 79	80, 81	82, 83
P_2	12, 13	14, 15	16, 17	36, 37	38, 39	40, 41	60, 61	62, 63	64, 65	84, 85	86, 87	88, 89
P_3	18, 19	20, 21	22, 23	42, 43	44, 45	46, 47	66, 67	68, 69	70, 71	90, 91	92, 93	94, 95

$\downarrow$

	Destination: BLOCK-CYCLIC(4)															
index	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15	16 17	18 19	20 21	22 23	24 25	26 27	28 29	30 31
meta	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Q_0	0, 1	2, 3	12, 13	14, 15	24, 25	26, 27	36, 37	38, 39	48, 49	50, 51	60, 61	62, 63	72, 73	74, 75	84, 85	86, 87
Q_1	4, 5	6, 7	16, 17	18, 19	28, 29	30, 31	40, 41	42, 43	52, 53	54, 55	64, 65	66, 67	76, 77	78, 79	88, 89	90, 91
Q_2	8, 9	10, 11	20, 21	22, 23	32, 33	34, 35	44, 45	46, 47	56, 57	58, 59	68, 69	70, 71	80, 81	82, 83	92, 93	94, 95

(b)

Fig. 4. (a) A $(6,4) \rightarrow (4,3)$ redistribution with $N = 96$. (b) An example of using a grouped meta-array for the redistribution in (a).

destination processor P_j, so are $A[\alpha + lcm(s \times P, t \times Q)]$, $A[\alpha + 2 \times lcm(s \times P, t \times Q)], \ldots,$ and $A[\alpha + (N - lcm(s \times P,\ t \times Q))]$, where $0 \leq j \leq Q - 1$. □

Lemma 2. *Given a $(s, P) \rightarrow (t, Q)$ and a $(s/gcd(s,t), P) \rightarrow (t/gcd(s,t), Q)$ redistribution on a one-dimensional array $A[0 : N - 1]$, for a source (destination) processor $P_i(Q_j)$, if the destination (source) processor of $SLA_i[k](DLA_j[k])$ in $(s/gcd(s,t), P) \rightarrow (t/gcd(s,t)$ redistribution is $Q_j(P_i)$, then the destination (source) processors of*

$$SLA_i[k \times gcd(s,t) : (k+1) \times gcd(s,t) - 1]$$
$$(DLA_j[k \times gcd(s,t) : (k+1) \times gcd(s,t) - 1])$$

in $(s, P) \rightarrow (t, Q)$ redistribution will also be $Q_j(P_i)$, where

$$0 \leq k \leq \lceil N/(P \times gcd(s,t))\rceil (0 \leq k < \lceil N/(Q \times gcd(s,t))\rceil).$$

Proof. We only prove the source processor part. The proof of the destination processor part is similar. For a source processor P_i, if the global array index of $SLA_i[k]$ in $(s/gcd(s,t), P) \rightarrow (t/gcd(s,t), Q)$ redistribution is α, then the global array indices of $SLA_i[k \times gcd(s,t) : (k+1) \times gcd(s,t) - 1]$ in $(s,\ P) \rightarrow (t,\ Q)$ redistribution are

$$\alpha \times gcd(s,t), \alpha \times gcd(s,t) + 1, \ldots, (\alpha + 1) \times gcd(s,t) - 1.$$

If $A[0 : N - 1]$ is distributed by BLOCK-CYCLIC $(t/gcd(s,t))$ distribution, then $A[\alpha]$ is in the $\lceil(\alpha \times gcd(s,t))/t\rceil$th block of size $t/gcd(s,t)$. If $A[0 : N - 1]$ is distributed by BLOCK-CYCLIC(t) distribution, then $A[\alpha \times gcd(s,t)], A[\alpha \times gcd(s,t) + 1], \ldots,$ and $A[(\alpha + 1) \times gcd(s,t) - 1]$ are in the $\lceil \alpha \times gcd(s,t)/t\rceil$th, the $\lceil(\alpha \times gcd(s,t) + 1)/t\rceil$th, ..., and the $\lceil((\alpha + 1) \times gcd(s,t) - 1)/t\rceil$th block of size t, respectively. Since

$$\lceil \alpha \times gcd(s,t)/t\rceil = \lceil(\alpha \times gcd(s,t) + 1)/t\rceil = \ldots$$
$$= \lceil((\alpha + 1) \times gcd(s,t) - 1)/t\rceil,$$

if the destination processor of $A[\alpha]$ is Q_j in $(s/gcd(s,t), P) \rightarrow (t/gcd(s,t), Q)$ redistribution, then the destination processors of $A[\alpha \times gcd(s,t)], A[\alpha \times gcd(s,t) + 1], \ldots,$ and $A[(\alpha + 1) \times gcd(s,t) - 1]$ are Q_j in $(s, P) \rightarrow (t, Q)$ redistribution. Therefore, if the destination processor of $SLA_i[k]$ in $(s/gcd(s,t), P) \rightarrow (t/gcd(s,t), Q)$ redistribution is Q_j, then the destination processors of $SLA_i[k \times gcd(s,t) : (k+1) \times gcd(s,t) - 1]$ in $(s, P) \rightarrow (t, Q)$ redistribution will also be Q_j, where

$$0 \leq i \leq P - 1, 0 \leq j \leq Q - 1$$
$$\text{and } 0 \leq k < \lceil N/(P \times gcd(s,t))\rceil.$$ □

In the following discussion, we assume that a $(s, P) \rightarrow (t, Q)$ redistribution on $A[0 : N - 1]$ is given. We also assume

SLA_0	Local index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
	Global index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
	Destination processor	Q_0			Q_1			Q_2			Q_3	Q_2			Q_3			Q_0			Q_1
SLA_1	Local index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
	Global index	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**
	Destination processor	Q_3		Q_0			Q_1			Q_2		Q_1		Q_2			Q_3			Q_0	
SLA_2	Local index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
	Global index	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**
	Destination processor	Q_2	Q_3			Q_0			Q_1			Q_0	Q_1			Q_2			Q_3		

Fig. 5. The send processor/data sets of the first generalized basic-cycle for a $(10, 3) \rightarrow (3, 4)$ redistribution shown in Fig. 2.

that $gcd(s, t)$ is equal to one. If $gcd(s, t)$ is not equal to one, we use $s/gcd(s, t)$ and $t/gcd(s, t)$ as the source and destination distribution factors of the redistribution, respectively.

3.1 The Send Phase

According to Lemma 1, each generalized basic-cycle of a local array has the same communication pattern. Therefore, each source processor only needs to compute the send processor/data sets on the first generalized basic-cycle of the local array that it owns. Then, based on the send processor/data sets of the first generalized basic-cycle, it can pack array elements into messages and send messages to their corresponding destination processors.

Given a $(s, P) \rightarrow (t, Q)$ redistribution on $A[0 : N-1]$, the destination processor of array element $SLA_i[k]$ in $SLA_i[0 : GBC-1]$ of source processor P_i can be determined by the following equations,

$$sgindex_i(k) = \lfloor k/s \rfloor \times s \times P + i \times s + mod(k, s), \qquad (1)$$

$$dp_i(sgindex_i(k)) = mod(\lfloor sgindex_i(k)/t \rfloor, Q), \qquad (2)$$

where $k = 0$ to $GBC - 1$. The function $sgindex_i(k)$ converts the local array index of an array element in a source local array to its corresponding global array index, i.e., $SLA_i[k] = A[sgindex_i(k)]$. The function $dp_i(sgindex_i(k))$ is used to determine the destination processor of the global array element $A[sgindex_i(k)]$.

If the value of GBC is large, it may take a lot of time to compute the destination processor of every array element in a generalized basic-cycle by using (1) and (2). Since array elements in a source section have consecutive global array indices, for a source processor P_i, if the destination processor of $SLA_i[0 : r-1]$ is Q_j, then the destination processors of $SLA_i[r : r+t-1]$, $SLA_i[r+t : r+2t-1], \ldots,$ and $SLA_i[r + \lfloor (s-r)/t \rfloor \times t : s-1]$ are $Q_{mod(j+1,Q)}$, $Q_{mod(j+2,Q)}, \ldots,$ and $Q_{mod(j+\lfloor (s-r)/t \rfloor,Q)}$, respectively, where $1 \leq r \leq t$. For example, Fig. 5 shows the send processor/data sets of the first generalized basic-cycle of source processors for a $(10, 3) \rightarrow (3, 4)$ redistribution shown in Fig. 2. In Fig. 5, for source processor P_1, the destination processor of $SLA_1[0 : r-1] = SLA_1[0 : 1]$ is $Q_j = Q_3$, where $r = 2$ and $j = 3$. The destination processors of $SLA_1[r : r + \ t-1] = SLA_1[2 : 4]$, $SLA_1[r + t : r+2t-1] = SLA_1[5 : 7]$, and $SLA_1[r + \lfloor (s-r)/t \rfloor \times t : s-1] = SLA_1[8 : 9]$ are $Q_{mod(j+1,Q)} = Q_0$, $Q_{mod(j+2,Q)} = Q_1$, and $Q_{mod(j+\lfloor (s-r)/t \rfloor,Q)} = Q_2$, respectively. Therefore, if we know the destination processor of the first array element of a source section and the value of r, we can determine the send processors/data sets in a source section. To determine the global array index of the first array element of a source section, (1) can be simplified as follows:

$$sgindex_i(k) = k \times P + i \times s, \qquad (3)$$

where k is the local array index of the first array element of a source section. The value of r can be determined by the following equation,

$$r = (\lfloor sgindex_i(k)/t \rfloor + 1) \times t - sgindex_i(k). \qquad (4)$$

Since a generalized basic-cycle has GBC/s source sections, (2), (3), and (4) only need to be performed GBC/s times. Then the send processor/data sets of a generalized basic-cycle can be obtained.

From the send processor/data sets, we can pack array elements into messages and send messages to their corresponding destination processors. The naive way to pack array elements into messages is to copy them to messages one element at a time according to the send processor/data sets. We define the operation of moving a block of data between a local array and a message as a data-movement operation. Since packing is a sequence of data-movement operations, if the local array size is large, this naive method may produce high packing cost. If we can reduce the number of data-movement operations, the packing cost can be reduced. From the indexing method described above, for a source processor P_i, if the destination processor of $SLA_i[0 : r-1]$ is Q_j, then the destination processors of $SLA_i[r : r+t-1], SLA_i[r+t : r+2t-1], \ldots,$ and $SLA_i[r + \lfloor (s-r)/t \rfloor \times t : s-1]$ are $Q_{mod(j+1,Q)}$, $Q_{mod(j+2,Q)}, \ldots,$ and $Q_{mod(j+\lfloor (s-r)/t \rfloor,Q)}$, respectively, where $1 \leq r \leq t$. For each source processor P_i, we can construct a *packing pattern table* $PPT_i[0 : Q-1]$ to describe the above send processor/data sets. For example, for the send processor/data sets of the first generalized basic-cycle shown in Fig. 5, source processor P_1's corresponding packing pattern table is given as follows:

$PPT_1[0]$ = {{2, 3}, {18, 2}},
$PPT_1[1]$ = {{5, 3}, {10, 2}},
$PPT_1[2]$ = {{8, 2}, {12, 3}},
$PPT_1[3]$ = {{0, 2}, {15, 3}}.

Each entry of a packing pattern table contains a list of descriptors. Each descriptor stores information of the start position and the number of array elements to be packed when performing a data-movement operation. A descriptor is of the form {*pos*, *len*}, where *pos* denotes the start position and *len* is the number of array elements to be packed. It is possible that the last array element of source section m and the first array element of source section $m+1$ have the same destination processor. In our implementation, we will combine the descriptors corresponding to these two array elements to a descriptor. Based on the above packing pattern table $PPT_1[0:3]$, when packing array elements whose destination processor is Q_0 into $message_0$, the entry $PPT_1[0]$ = {{2, 3}, {18, 2}} will be used. According to $PPT_1[0]$ = {{2, 3}, {18, 2}}, source processor P_1 will pack array elements $SLA_1[2:4]$ and $SLA_1[18:19]$ in the first generalized basic-cycle of SLA_1 into $message_0[0:2]$ (descriptor {2,3}) and $message_0[3:4]$ (descriptor {18,2}), respectively. Array elements $SLA_1[2+GBC:4+GBC]$ and $SLA_1[18+GBC:19+GBC]$ in the second generalized basic-cycle of SLA_1 will be packed into $message_0[5:7]$ (descriptor {2,3}) and $message_0[8:9]$ (descriptor {18,2}), respectively, etc. Based on the packing pattern table, the total number of data-movement operations performed by each source processor P_i is equal to (the number of descriptors in $PPT_i[0:Q-1]$) × (the number of generalized basic-cycles in SLA_i), which is much less than that of the naive method. The algorithm to construct the packing pattern table in the send phase is given as follows:

```
Algorithm PPT_construction (i, s, P, t, Q)
1. gcdst = gcd(s, t);   s = s/gcdst;   t = t/gcdst;
2. calculate the GBC for the sending phase;   lastp = -1;
3. for m = 0 to GBC/s-1
4.   k = m×s;   gidx = k×P + i×s;   secend = gidx + s;
5.   j = mod(⌊gidx/t⌋, Q);   l = (min((⌊gidx/t⌋ + 1) ×t,
         secend) -gidx);
6.   if j = lastp then
7.     $PPT_i[j][c_j-1].len += l×gcdst$;
8.     k += l;   gidx += l;   l = t;   j = mod(j + 1, Q);
9.   endif
10.  while gidx < secend
11.    l = min(l, secend-gidx);
12.    $PPT_i[j][c_j].pos = k × gcdst$;   $PPT_i[j][c_j].len = l × gcdst$;
13.    $c_j$++;   k += l;   gidx += l;   l = t;
14.    lastp = j;   j = mod(j + 1, Q);
15.  endwhile
16. endfor
End_of_PPT_construction
```

3.2 The Receive Phase

In the receive phase, techniques for the indexing and the packing/unpacking issues are similar to those in the send phase. We only state the key points of the techniques and ignore the details of examples as we did in the send phase. Given a $(s,P) \to (t,Q)$ redistribution on $A[0:N-1]$, for destination processor Q_j, the source processor of array element $DLA_j[k]$ in $DLA_j[0:GBC-1]$ can be determined by the following equations:

$$rgindex_j(k) = \lfloor k/t \rfloor \times t \times Q + j \times t + mod(k,t) \tag{5}$$

$$sp_j(rgindex_j(k)) = mod(\lfloor rgindex_j(k)/s \rfloor, P) \tag{6}$$

where $k = 0$ to $GBC-1$. The function $rgindex_i(k)$ converts the local array index of an array element in a destination local array to its corresponding global array index, i.e., $DLA_j[k] = A[rgindex_j(k)]$. The function $sp_j(rgindex_j(k))$ is used to determine the source processor of the global array element $A[rgindex_j(k)]$.

Since array elements in a destination section have consecutive global array indices, for a destination processor Q_j, if the source processor of $DLA_j[0:u-1]$ is P_i, then the source processors of $DLA_j[u:u+s-1]$, $DLA_j[u+s:u+2s-1], \ldots$, and $DLA_j[u+\lfloor (t-u)/s \rfloor \times s:t-1]$ are $P_{mod(i+1,P)}$, $P_{mod(i+2,P)}, \ldots$, and $P_{mod(i+\lfloor (t-u)/s \rfloor,P)}$, respectively, where $1 \leq u \leq s$. If we know the source processor of the first array element of a destination section and the value of *u*, we can determine the receive processors/data sets in a destination section. To determine the global array index of the first array element of a destination section, (5) can be simplified as follows:

$$rgindex_j(k) = k \times Q + j \times t, \tag{7}$$

where *k* is the local array index of the first array element of a destination section. The value of *u* can be determined by the following equation:

$$u = (\lfloor rgindex_j(k)/s \rfloor + 1) \times s - rgindex_j(k). \tag{8}$$

According to the indexing method described above, for a destination processors Q_j, if the source processor of $DLA_j[0:u-1]$ is P_i, then the source processors of $DLA_j[u:u+s-1]$, $DLA_j[u+s:u+2s-1], \ldots$, and $DLA_j[u+\lfloor (t-u)/s \rfloor \times s:t-1]$ are $P_{mod(i+1,P)}$, $P_{mod(i+2,P)}, \ldots$, and $P_{mod(i+\lfloor (t-u)/s \rfloor,P)}$, respectively, where $1 \leq u \leq s$. For each destination processor Q_j, we can construct an *unpacking pattern table* $UPT_j[0:P-1]$ to describe the above receive processor/data sets. Based on the unpacking pattern table, a destination processor can unpack array elements from received messages efficiently. The algorithm to construct the unpacking pattern table is given as follows:

```
Algorithm UPT_construction (j, s, P, t, Q)
1. gcdst = gcd(s, t);   s = s/gcdst;   t = t/gcdst;
2. calculate the GBC for the receive phase;   lastp = −1;
3. for m = 0 to GBC/t − 1
4.   k = m×t;   gidx = k×Q + j×t;   secend = gidx + t;
5.   i = mod(⌊gidx/s⌋, P); l = (min((⌊gidx/s⌋ + 1) × s,
         secend) −gidx);
6.   if i = lastp then
7.     $UPT_j[i][c_i-1].len += l×gcdst$;
8.     k += l;   gidx += l;   l = s;   i = mod(i + 1,P);
9.   endif
10.  while gidx < secend
11.    l = min(l, secend − gidx);
12.    $UPT_j[i][c_i].pos = k×gcdst$;   $UPT_j[i][c_i].len = l×gcdst$;
13.    $c_i$++; k += l;   gidx += l;   l = s;
14.    lastp = i;   i = mod(i + 1,P);
15.  endwhile
16. endfor
End_of_UPT_construction
```

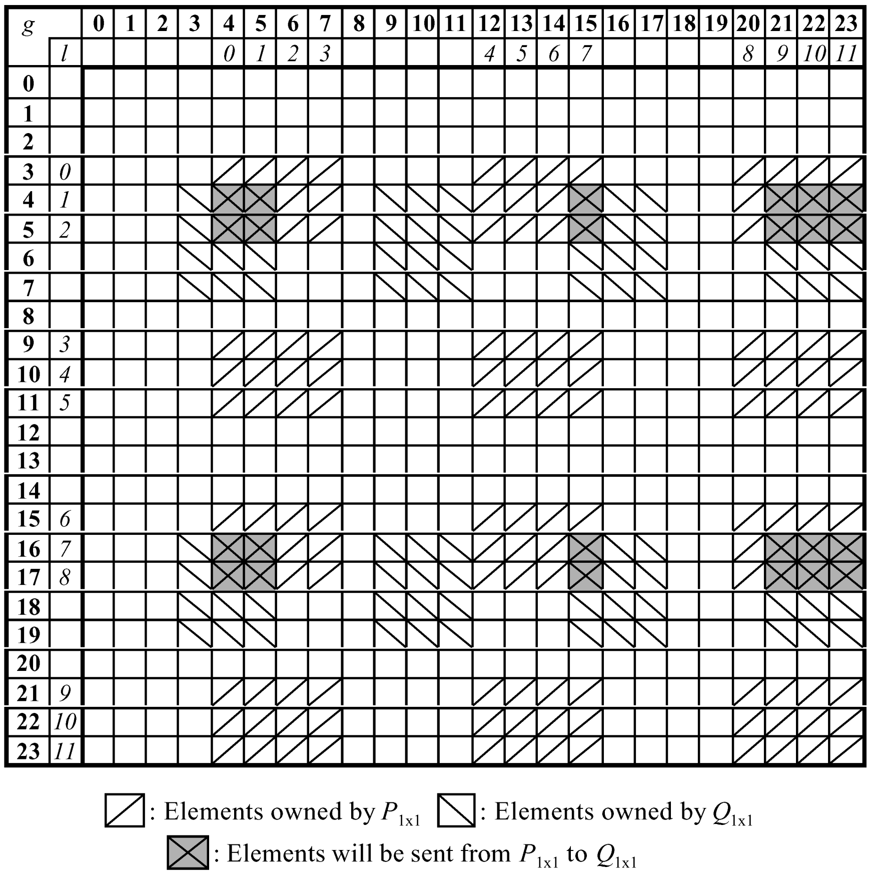
g
l
: Elements owned by P_{1x1}
: Elements owned by Q_{1x1}
: Elements will be sent from P_{1x1} to Q_{1x1}

Fig. 6. An example of a $(3 \times 4, 2 \times 2) \rightarrow (4 \times 3, 3 \times 2)$ redistribution, where *g* is the global array index and *l* is the source local array index of the source processor P_{1x1} for each dimension.

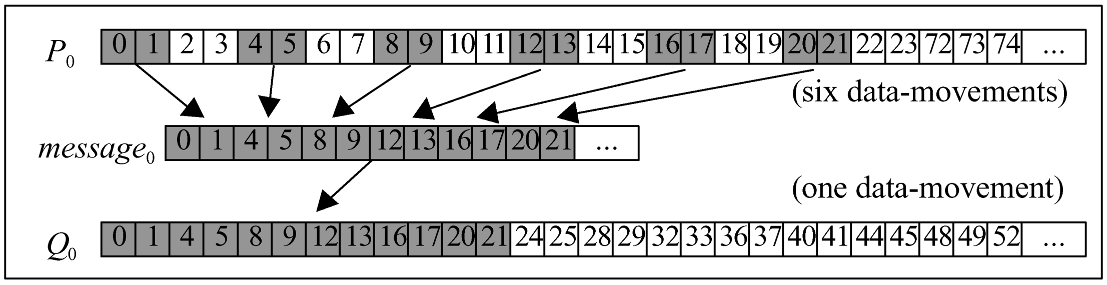

Fig. 7. Given a $(24, 3) \rightarrow (2, 2)$ redistribution, the shadowed array elements in a source section of SLA_0 will be sent from P_0 to Q_0. There are six data-movement operations and one data-movement operation in the send phase and the receive phase, respectively.

The algorithm of the *GBCC* method is given as follows:

```
Algorithm GBCC (s, P, t, Q)
   /* Sending Phase */
1.  i = get_myrank_of_source_processors();
2.  call PPT_construction(i, s, P, t, Q);
3.  for j = 0 to $Q - 1$
4.      if $c_j$ > 0 then
5.          pack data from source local array to a message according
            to $PPT_i[j]$;
6.          send message to $Q_j$;
7.      endif
8.  endfor
   /* Receiving Phase */
9.  j = get_myrank_of_destination_processors();
10. call UPT_construction(j, s, P, t, Q);
11. for i = 0 to $P - 1$
12.     if $c_i$ > 0 then
13.         receive message from $P_i$;
14.         unpack received message to destination local array
            according to $UPT_j[i]$;
15.     endif
16. endfor
17. wait for all communication;
End_of_GBCC
```

3.3 The *GBCC* Method for Multiimensional Array Redistribution

The *GBCC* method can be extended easily to perform multidimensional array redistributions. In the send phase, the packing pattern table for each dimension is calculated by using the *GBCC* method. Based on the packing pattern

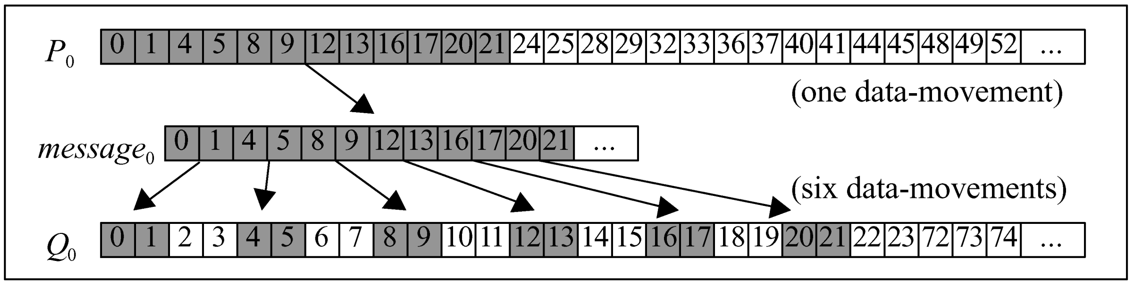

Fig. 8. Given a $(2,2) \rightarrow (24,3)$ redistribution, the shadowed array elements in a source section of SLA_0 will be sent from P_0 to Q_0. There are one data-movement operation and six data-movement operations in the send phase and the receive phase, respectively.

TABLE 1
The Indexing Costs and the Packing/Unpacking Costs of the *PITFALLS* Method, the *ScaLAPACK* Method, and the *GBCC* Method for a $(s,P) \rightarrow (t,Q)$ Redistribution on a 1D Array with *N* Array Elements

Algorithms	**Indexing costs**		
PITFALLS / *ScaLAPACK*	$O\left(\frac{lcm(s\times P,t\times Q)}{min(s,t\times Q)\times P}\times Q+\frac{lcm(s\times P,t\times Q)}{min(t,s\times P)\times Q}\times P\right)$		
GBCC	$O\left(\frac{lcm(s\times P,t\times Q)}{min(s,t)\times P}+\frac{lcm(s\times P,t\times Q)}{min(s,t)\times Q}\right)$		
	Packing/unpacking costs		
PITFALLS / *ScaLAPACK*	$O\left(\frac{N/P+N/Q}{min(s,t)}\right)$		
GBCC	$s > t\times Q$	$t > s\times P$	otherwise
	$O\left(\frac{N/P}{t}+\frac{N/Q}{t\times Q}\right)$	$O\left(\frac{N/P}{s\times P}+\frac{N/Q}{s}\right)$	$O\left(\frac{N/P+N/Q}{min(s,t)}\right)$

tables, array elements that will be sent to the same destination processor are packed dimension by dimension starting from the first (last) dimension if the array is in column-major (row-major). In the receive phase, the unpacking pattern table for each dimension is calculated by using the *GBCC* method. Based on the unpacking pattern tables, elements in a message that was received from a source processor are unpacked to their corresponding positions dimension by dimension starting from the first (last) dimension if the array is in column-major (row-major).

We now give an example to explain how to use the *GBCC* method to perform a multidimensional array redistribution. Fig. 6 shows the array elements that will be sent from P_{1x1} to Q_{1x1} in a $(3\times 4, 2\times 2) \rightarrow (4\times 3, 3\times 2)$ redistribution with $N = 24\times 24$ array elements. For the first dimension (P_{1x} to Q_{1x}), the packing pattern table for destination processor Q_{1x} is $PPT_{1x}[1] = \{\{1,2\}\}$. For the second dimension (P_{1x} to Q_{1x}), the packing pattern table for destination processor Q_{1x} is $PPT_{x1}[1] = \{\{0,2\},\{7,1\},\{9,3\}\}$. Assume that array elements are stored in memory in a row-major manner. From Fig. 6, for the source processor $P_{1\times 1}$, we can see that the array elements in SLA_o that have consecutive local array indices in the second dimension (the last dimension) will be stored in consecutive positions in memory. But it is not the case for other dimensions. Based on the observation, $PPT_{1x}[1]$, and $PPT_{x1}[1]$, source processor P_{1x1} can pack array elements $SLA_{1x1}[1,0]$, $SLA_{1x1}[1,1]$, $SLA_{1x1}[1,7]$, $SLA_{1x1}[1,9]$, $SLA_{1x1}[1,10]$, $SLA_{1x1}[1,11]$, $SLA_{1x1}[2,0]$, $SLA_{1x1}[2,1]$, $SLA_{1x1}[2,7]$, $SLA_{1x1}[2,9]$, $SLA_{1x1}[2,10]$, and $SLA_{1x1}[2,11]$ into $message_{1x1}[0:11]$ according to $PPT_{1x}[1]$ and $PPT_{x1}[1]$. For the second generalized basic-cycle of SLA_{1x1} in the first dimension, array elements $SLA_{1x1}[7,0]$, $SLA_{1x1}[7,1]$, $SLA_{1x1}[7,7]$, $SLA_{1x1}[7,9]$, $SLA_{1x1}[7,10]$, $SLA_{1x1}[7,11]$, $SLA_{1x1}[7,0]$, $SLA_{1x1}[7,1]$, $SLA_{1x1}[7,7]$, $SLA_{1x1}[7,9]$, $SLA_{1x1}[7,10]$, and $SLA_{1x1}[7,11]$ will be packed into $message_{1x1}[12:23]$ according to $PPT_{1x}[1]$ and $PPT_{x1}[1]$. For each destination processor, the received messages can be unpacked in a similar manner.

4 Performance Evaluation and Experimental Results

To evaluate the performance of the *GBCC* method, we compare the proposed method with the *PITFALLS* method and the *ScaLAPACK* method. Both theoretical analysis and experimental evaluation were conducted. We first develop

TABLE 2
The Indexing Costs, the Packing/Unpacking Costs, the Communication Costs, and the Total Costs for These Three Methods to Perform Test Samples on Arrays with N = 80,000 and N = 20,000,000

methods / cases	*PITFALLS*				*ScaLAPACK*				*GBCC*			
	$N = 80000$											
	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}
(5, 8)→(2, 5)	0.229	11.95	5.72	17.9	0.155	11.02	5.13	16.3	0.029	9.39	4.68	14.1
(50, 8)→(20, 5)	0.228	2.25	5.42	7.9	0.148	2.20	5.25	7.6	0.029	2.06	5.41	7.5
(4, 8)→(5, 5)	1.143	6.18	5.48	12.8	1.092	5.73	5.08	11.9	0.242	4.84	5.62	10.7
(5, 5)→(2, 8)	0.816	8.85	4.83	14.5	0.807	8.35	5.14	14.3	0.142	7.25	4.71	12.1
(50, 5)→(20, 8)	0.816	2.02	5.06	7.9	0.806	1.92	5.17	7.9	0.142	1.88	5.58	7.6
(4, 5)→(5, 8)	0.169	6.86	4.57	11.6	0.123	6.45	4.03	10.6	0.028	5.60	3.87	9.5
(5, 10)→(2, 10)	0.361	7.06	6.48	13.9	0.312	6.56	6.13	13.0	0.036	5.60	4.36	10.0
(50, 10)→(20, 10)	0.358	1.40	5.34	7.1	0.308	1.37	5.22	6.9	0.037	1.28	3.98	5.3
(4, 10)→(5, 10)	0.421	4.21	4.17	8.8	0.389	3.98	4.03	8.4	0.052	3.45	4.10	7.6
(5, 50)→(2, 50)	1.625	1.52	4.66	7.8	1.500	1.42	3.88	6.8	0.038	1.23	3.33	4.6
(50, 50)→(20, 50)	1.611	0.38	4.11	6.1	1.498	0.37	3.53	5.4	0.039	0.36	3.20	3.6
(4, 50)→(5, 50)	1.795	0.95	3.16	5.9	1.831	0.93	2.74	5.5	0.053	0.80	2.35	3.2
	$N = 20000000$											
	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}
(5, 8)→(2, 5)	0.238	3081	886	3967	0.160	2844	858	3702	0.030	2426	824	3250
(50, 8)→(20, 5)	0.233	717	938	1655	0.156	691	941	1632	0.030	652	950	1602
(4, 8)→(5, 5)	1.159	2063	875	2939	1.112	1963	936	2900	0.243	1910	965	2875
(5, 5)→(2, 8)	0.832	2935	879	3815	0.816	2799	820	3620	0.143	2771	793	3564
(50, 5)→(20, 8)	0.831	629	1541	2171	0.815	618	1475	2094	0.143	576	1547	2123
(4, 5)→(5, 8)	0.174	1828	826	2654	0.129	1718	830	2548	0.028	1495	742	2237
(5, 10)→(2, 10)	0.368	1854	508	2362	0.321	1723	525	2248	0.037	1482	588	2070
(50, 10)→(20, 10)	0.367	427	763	1190	0.310	412	751	1163	0.037	390	727	1117
(4, 10)→(5, 10)	0.446	1243	797	2040	0.391	1175	839	2014	0.053	1045	795	1840
(5, 50)→(2, 50)	1.632	373	166	541	1.495	344	173	518	0.040	297	185	482
(50, 50)→(20, 50)	1.616	86	204	292	1.520	83	199	284	0.040	79	195	274
(4, 50)→(5, 50)	1.867	249	216	467	1.831	234	216	452	0.055	210	210	420

Time (ms)

cost models for these three methods and analyze their performance in terms of the indexing and the packing/unpacking costs. The cost models developed for the *PITFALLS* method and the *ScaLAPACK* method are based on algorithms proposed in [20], [21], and [19], respectively. We then execute these three methods on an IBM SP2 parallel machine and use the cost models to analyze the experimental results.

4.1 Cost Models

Given a (s, P)→(t, Q) redistribution on a one-dimensional array A[0:N-1], the time for an algorithm to perform the redistribution, in general, can be modeled as follows:

$$T = T_{comp} + T_{comm}, \tag{9}$$

where T_{comp} is the time for an algorithm to compute the source/destination processors of local array elements, pack source local array elements that have the same destination processors to the same message, and unpack array elements in messages that received from source processors to their corresponding destination local array positions; and T_{comm} is the communication time for an algorithm to send and receive data among processors. We said that T_{comp} and T_{comm} are the computation and communication time of an algorithm to perform a redistribution, respectively. For the communication cost, the number of send and receive operations required by a processor in a redistribution are the same for different methods. Therefore, we assume that the communication costs of these three methods are the same in our theoretical model. In the following, we will focus on the analysis of the computation costs of the three methods.

The computation cost consists of the indexing cost and the packing/unpacking cost. The indexing cost is the time to construct the send/receive processor/data sets for a redistribution. The packing/unpacking cost is the time to pack and unpack array elements. We have the following equation,

$$T_{comp} = T_{index} + T_{(un)pack}, \tag{10}$$

where T_{index} and $T_{(un)pack}$ are the indexing cost and the packing/unpacking cost of a redistribution, respectively. In the cost model analysis, the packing/unpacking cost is represented in terms of the number of data-movement operations. For the *PITFALLS* method, the indexing cost for

TABLE 3
The Indexing Costs, the Packing/Unpacking Costs, the Communication Costs, and the Total Costs for These Three Methods to Perform Test Samples on Arrays with *N* = 80,000 and *N* = 20,000,000

methods / cases	*PITFALLS*				*ScaLAPACK*				*GBCC*			
	$N = 80000$											
	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}
(500, 8)→(3, 5)	16.2	6.7	6.1	29.0	12.2	4.6	5.2	22.0	4.3	3.1	2.9	10.3
(3, 8)→(500, 5)	10.2	7.3	5.9	23.4	10.9	6.0	4.9	21.8	3.8	4.0	4.9	12.7
(500, 8)→(1, 5)	15.9	15.7	5.7	37.3	7.3	13.5	5.9	26.7	1.4	5.6	4.9	11.9
(1, 8)→(500, 5)	10.1	16.2	7.5	33.8	9.3	13.5	6.1	28.9	1.6	8.0	5.0	14.6
(500, 5)→(3, 8)	10.2	7.4	5.3	22.9	10.9	5.1	5.2	21.2	3.8	4.1	5.7	13.6
(3, 5)→(500, 8)	16.2	6.9	4.9	28.0	12.1	4.8	4.8	21.7	3.1	2.8	5.2	11.1
(500, 5)→(1, 8)	10.1	16.5	5.6	32.2	9.3	13.9	5.3	28.5	1.6	8.7	5.4	15.7
(1, 5)→(500, 8)	15.9	15.9	5.1	36.9	7.3	13.7	5.0	26.0	1.3	5.2	6.1	12.6
(500, 10)→(3, 10)	12.9	4.6	2.6	20.1	12.9	3.2	2.5	18.6	2.3	2.2	2.9	7.4
(3, 10)→(500, 10)	12.9	4.6	2.6	20.1	12.8	3.2	2.5	18.5	2.4	2.2	2.8	7.4
(500, 10)→(1, 10)	12.6	10.1	3.1	25.8	10.3	8.5	2.6	21.4	1.0	4.5	2.7	8.2
(1, 10)→(500, 10)	12.5	10.2	2.6	25.3	10.3	8.5	2.6	21.4	1.0	4.3	4.4	9.7
	$N = 20000000$											
	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}
(500, 8)→(3, 5)	16.3	2521	799	3336	12.1	2436	829	3277	3.1	1395	800	2198
(3, 8)→(500, 5)	10.3	2581	798	3389	10.9	2487	805	3303	3.8	1503	795	2302
(500, 8)→(1, 5)	16.0	5084	1203	6303	7.3	4843	957	5807	1.4	2580	1126	3707
(1, 8)→(500, 5)	10.1	5308	934	6252	9.4	5086	923	6018	1.7	3335	921	4258
(500, 5)→(3, 8)	10.4	2544	803	3357	10.9	2476	867	3354	3.8	2182	850	3036
(3, 5)→(500, 8)	16.3	2500	857	3373	12.2	2419	850	3281	3.1	1341	838	2182
(500, 5)→(1, 8)	10.1	5611	1050	6671	9.4	5431	958	6398	1.6	3335	1124	4461
(1, 5)→(500, 8)	16.0	5381	876	6273	7.3	5179	1003	6189	1.4	2572	939	3512
(500, 10)→(3, 10)	12.9	1482	831	2326	12.9	1436	744	2193	2.4	1063	821	1886
(3, 10)→(500, 10)	12.9	1482	830	2325	12.9	1426	819	2258	2.4	1024	795	1821
(500, 10)→(1, 10)	12.6	3083	931	4027	10.4	2938	974	3922	1.0	2014	892	2907
(1, 10)→(500, 10)	12.6	3108	988	4109	10.3	2951	1073	4034	1.0	2018	1143	3162

Time (ms)

a processor to perform the FALLS intersection algorithm [20], [21] is

$$T_{index}(PITFALLS) = O\left(\frac{lcm(s \times P, t \times Q)}{min(s, t \times Q) \times P} \times Q + \frac{lcm(s \times P, t \times Q)}{min(t, s \times P) \times Q} \times P\right). \quad (11)$$

The packing/unpacking cost of the *PITFALLS* method is

$$T_{(un)pack}(PITFALLS) = O\left(\frac{N/P + N/Q}{min(s, t)}\right). \quad (12)$$

For the *ScaLAPACK* method [19], the indexing and packing/unpacking costs are the same as the *PITFALLS* method.

For the *GBCC* method, according to the algorithm presented in Section 3, the indexing cost is

$$T_{index}(GBCC) = O\left(\frac{lcm(s \times P, t \times Q)}{min(s, t) \times P} + \frac{lcm(s \times P, t \times Q)}{min(s, t) \times Q}\right). \quad (13)$$

The packing/unpacking cost of the generalized basic calculation method can be classified into three classes, $s > t \times Q$, $t > s \times P$, and otherwise. For the first class $s > t \times Q$, array elements that have the same destination processors in the same source section will have consecutive local array indices in its corresponding destination local array. Therefore, $\frac{s}{t \times Q}$ data-movement operations are needed to pack those array elements to a message and one data-movement operation is needed to unpack those array elements to their corresponding local array positions. For example, given a $(24, 3) \rightarrow (2, 2)$ redistribution, Fig. 7 shows that there are $\frac{s}{t \times Q} = 6$ data-movement operations that must be performed to pack 12 array elements in a source section of SLA_0 to $messages_0$ by source processor P_0 in the send phase. In the receive phase, only one data-movement operation is needed to unpack these 12 elements from the received message to their corresponding local array positions.

For the second class $t > s \times P$, array elements that have the same source processors in the same destination section will have consecutive local array indices in its corresponding source local array. Therefore, only one data-movement operation is needed to pack these array elements into a message and $\frac{t}{s \times P}$ data-movement operations are needed to

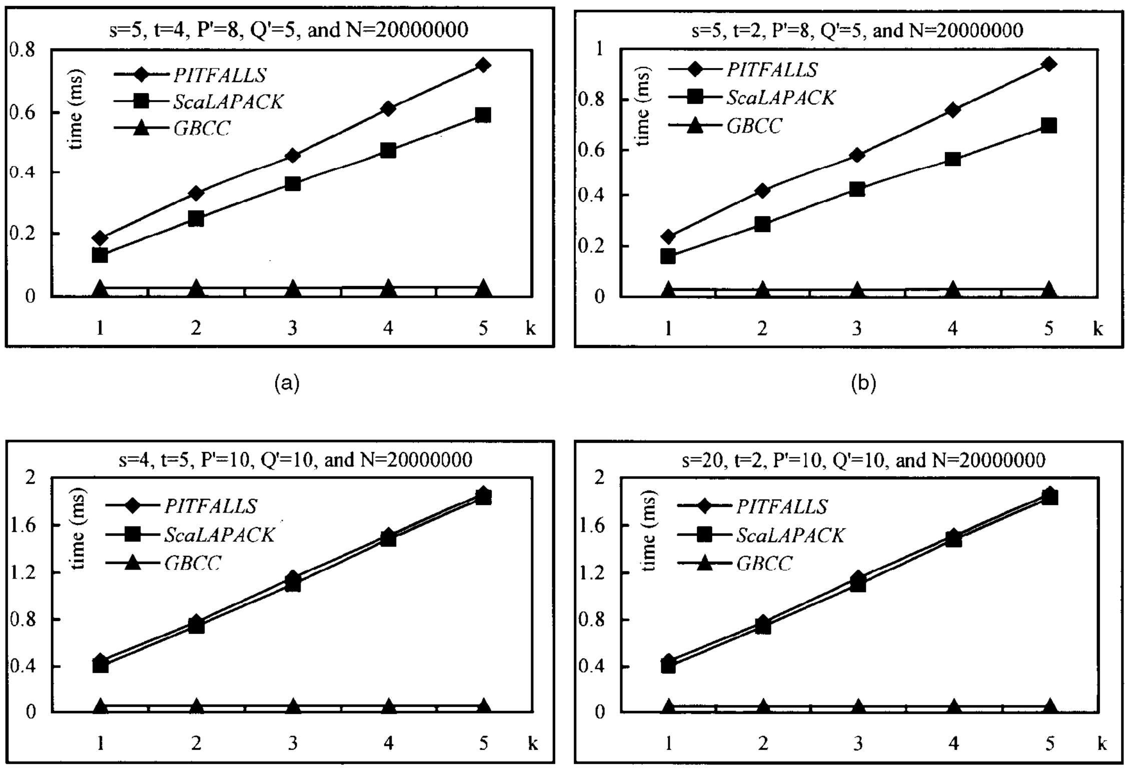

Fig. 9. The indexing costs of the $(s, kP') \rightarrow (t, kQ')$ redistribution where $k = 1, 2, 3, 4,$ and 5.

unpack those array elements to their corresponding local array positions. For example, given a $(2,2) \rightarrow (24,3)$ redistribution, Fig. 8 shows that only one data-movement operation is needed to pack 12 array elements into $message_0$ by source processor P_0 in the send phase. There are $\frac{t}{s \times P} = 6$ data-movement operations that must be performed by destination processor Q_0 to unpack those 12 elements from $messages_0$ to their corresponding local array positions in the receive phase.

The packing/unpacking costs of the three classes are given as follows:

$$T_{(un)pack}(GBCC) = O\left(\frac{N/P}{t} + \frac{N/Q}{t \times Q}\right) \quad \text{if } s > t \times Q, \quad (14)$$

$$\text{or} \quad O\left(\frac{N/P}{s \times P} + \frac{N/Q}{s}\right) \quad \text{if } t > s \times P, \quad (15)$$

$$\text{or} \quad O\left(\frac{N/P + N/Q}{min(s,t)}\right) \quad \text{otherwise.} \quad (16)$$

From the above analysis, we observe that the indexing cost of the *GBCC* method is less than that of the *PITFALLS* and the *ScaLAPACK* methods. The packing/unpacking cost of the *GBCC* method is less than or equal to that of the *PITFALLS* and the *ScaLAPACK* methods. We summarize the indexing costs and the packing/unpacking costs of these three methods in Table 1. According to Table 1, we use the example given in Fig. 7 to show the advantages of the *GBCC* method. For the $(2,2) \rightarrow (24,3)$ redistribution in Fig. 7, the indexing costs of the *GBCC*, the *PITFALLS*, and the *ScaLAPACK* methods are equal to 30, 66, and 66, respectively. The packing/unpacking costs of the *GBCC*, the *PITFALLS*, and the *ScaLAPACK* methods are equal to $7N/24$, $5N/12$, and $5N/12$, respectively, where N is the array size. The *GBCC* method has smaller indexing and packing/unpacking costs than those of the *PITFALLS* and the *ScaLAPACK* methods.

4.2 Experimental Results

To verify the performance analysis presented in Section 4.1, the *GBCC* method, the *PITFALLS* method, and the *ScaLAPACK* method were implemented on an IBM SP2 parallel machine. All algorithms were written in C+MPI codes with the single program multiple data (SPMD) programming paradigm. Based on the values of s, t, P, and Q in a $(s,P) \rightarrow (t,Q)$ redistribution, we have the following three cases:

Case 1. $s \leq t \times Q$ and $t \leq s \times P$,

Case 2. $s > t \times Q$ or $t > s \times P$,

Case 3. $P = kP', Q = kQ'$ where $gcd(P', Q') = 1$ and $k \geq 1$,

For each case, at least 10 different redistributions were used as test samples. Each test sample was executed 10 times. The mean time for the 10 tests was used as the time of a test sample. We also give some experimental results for two-dimensional array distributions.

TABLE 4
The Indexing Costs, the Packing/Unpacking Costs, the Communication Costs, and the Total Costs for These Three Methods to Perform Test Samples

methods / cases	*PITFALLS*				*ScaLAPACK*				*GBCC*			
	$N = 20000000$											
	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}
(5, 8)→(4, 5)	0.185	1797	813	2610	0.131	1693	807	2500	0.028	1469	804	2273
(5, 16)→(4, 10)	0.327	895	369	1264	0.246	839	357	1196	0.029	724	405	1129
(5, 24)→(4, 15)	0.463	598	327	925	0.359	559	291	850	0.028	485	344	829
(5, 32)→(4, 20)	0.613	451	299	751	0.480	420	304	724	0.030	362	307	669
(5, 40)→(4, 25)	0.753	361	196	558	0.593	342	192	535	0.030	294	165	459
(5, 8)→(2, 5)	0.238	3081	886	3967	0.160	2844	858	3702	0.030	2426	824	3250
(5, 16)→(2, 10)	0.419	1540	413	1953	0.286	1421	448	1869	0.031	1210	392	1602
(5, 24)→(2, 15)	0.580	1025	305	1331	0.425	941	313	1254	0.030	803	324	1127
(5, 32)→(2, 20)	0.759	764	282	1047	0.562	705	280	986	0.032	602	268	870
(5, 40)→(2, 25)	0.940	621	207	829	0.698	568	218	787	0.032	491	207	698
(4, 10)→(5, 10)	0.446	1243	797	2040	0.401	1175	739	1914	0.053	1045	795	1840
(4, 20)→(5, 20)	0.778	630	540	1171	0.745	597	524	1122	0.053	528	474	1002
(4, 30)→(5, 30)	1.160	500	392	893	1.103	469	361	831	0.053	392	378	770
(4, 40)→(5, 40)	1.655	348	380	730	1.431	334	318	653	0.053	292	334	626
(4, 50)→(5, 50)	1.867	249	216	467	1.831	234	216	452	0.055	210	210	420
(20, 10)→(2, 10)	0.490	1383	884	2267	0.405	1305	877	2182	0.040	1159	1009	2168
(20, 20)→(2, 20)	0.863	694	510	1205	0.744	657	488	1146	0.041	582	551	1133
(20, 30)→(2, 30)	1.254	549	411	961	1.101	464	435	900	0.042	366	615	981
(20, 40)→(2, 40)	1.655	348	380	730	1.431	335	317	653	0.043	292	334	626
(20, 50)→(2, 50)	2.049	279	293	574	1.779	265	301	568	0.042	235	305	540

Time (ms)

Case 1. $s \leq t \times Q$ **and** $t \leq s \times P$ Table 2 shows the indexing costs, the packing/unpacking costs, the communication costs, and the total costs for these three methods to perform test samples in this case on arrays with N = 80,000 and N = 20,000,000. From Table 2, we can see that the indexing costs of the *GBCC* method are less than that of the *ScaLAPACK* and the *PITFALLS* methods for all test samples. We also observed that the indexing costs are independent of the array size in these three methods. These phenomena match the indexing cost models presented in Section 4.1.

For the packing/unpacking part, the execution time of the three methods has the order $T_{(un)pack}(GBCC) < T_{(un)pack}(ScaLAPACK) < T_{(un)pack}(PITFALLS)$. This result is better than the analysis that given in Table 1. The reason is that the *GBCC* method uses a simpler computation approach than that of the *ScaLAPACK* and the *PITFALLS* methods when packing/unpacking array elements.

For the communication part, these three methods use asynchronous communication schemes. There is no clear winner in the communication cost for all test samples due to the characteristics of the asynchronous communication schemes. However, these three methods have approximately the same communication costs for all test samples.

Case 2. $s > t \times Q$ **or** $t > s \times P$**.** Table 3 shows the experimental results for the redistributions in Case 2. According to Table 1, the packing/unpacking costs of array redistribution depend on the array size. Therefore, when array size is large, the performance of packing/unpacking technique plays an important role in a redistribution. From Table 3, for test samples with array size N = 20,000,000, the packing/unpacking costs of the three methods has the order $T_{(un)pack}(GBCC) << T_{(un)pack}(ScaLAPACK) < T_{(un)pack}(PITFALLS)$. The packing/unpacking technique of the *GBCC* method outperforms those provided in the *PITFALLS* and the *ScaLAPACK* methods. The phenomenon matches the theoretical analysis presented in Section 4.1. For the communication costs, we have similar observations as those described for Case 1.

Case 3. $P = kP', Q = kQ'$ **where** $gcd(P', Q') = 1$ **and** $k \geq 1$ Fig. 9 shows the indexing costs of $(5, 8k) \rightarrow (4, 5k)$ redistributions with array size N = 20,000,000, where $k = 1$ to 5 . From Fig. 9, we can see that the indexing costs of the *PITFALLS* method and the *ScaLAPACK* method increase when the value of k increases. The indexing costs of the *GBCC* method are independent of the value of k. As described in Section 4.1, both $T_{index}(PITFALLS)$ and $T_{index}(ScaLAPACK)$ shown in (11) is approximately

$$\frac{t \times Q^2 + s \times P^2}{gcd(s \times P, t \times Q)},$$

TABLE 5
The Indexing Costs, the Packing/Unpacking Costs, the Communication Costs, and the Total Costs of These Three Methods to Perform 2D Array Redistributions on Arrays with Size 960×960 and $4,800 \times 4,800$

methods / cases	*PITFALLS*				*ScaLAPACK*				*GBCC*			
	$N = 960x960$											
	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}
(5x4, 4x3)→(4x5, 4x3)	1.00	48.1	86.8	135.9	0.74	41.1	85.3	127.1	0.08	34.4	83.0	117.5
(4x5, 4x3)→(8x2, 4x3)	0.66	65.4	51.0	117.1	0.45	59.7	50.3	110.4	0.04	53.2	49.2	102.4
(40x40, 4x3)→(1x1, 4x3)	3.53	124.6	89.4	217.5	2.49	97.2	85.0	184.7	0.18	56.2	79.5	135.9
(1x1, 4x3)→(40x40, 4x3)	3.57	123.9	91.1	218.6	2.49	99.7	80.5	182.7	0.18	52.4	82.7	135.3
(5x4, 8x6)→(4x5, 6x4)	2.87	19.8	26.6	49.3	2.30	16.1	24.8	43.2	0.09	13.6	25.5	39.2
(4x5, 8x6)→(8x2, 6x4)	3.43	28.3	21.4	53.1	2.63	22.7	21.6	46.9	0.07	21.4	21.2	42.7
(40x40, 8x6)→(1x1, 6x4)	26.60	57.9	33.8	118.3	18.17	39.2	27.3	84.7	0.42	19.0	24.0	43.4
(1x1, 8x6)→(40x40, 6x4)	11.08	76.0	30.6	117.7	10.78	42.7	26.4	79.9	0.29	35.5	28.2	64.0
(5x4, 6x4)→(4x5, 8x6)	6.60	24.5	25.1	56.2	5.80	17.2	23.7	46.7	0.22	17.2	25.8	43.2
(4x5, 6x4)→(8x2, 8x6)	2.80	26.0	27.1	55.9	2.23	22.6	25.7	50.5	0.06	20.9	26.9	47.9
(40x40, 6x4)→(1x1, 8x6)	10.69	76.3	32.5	119.5	10.64	45.3	25.7	81.6	0.29	38.5	22.2	61.0
(1x1, 6x4)→(40x40, 8x6)	26.28	58.9	29.6	114.8	18.23	40.7	28.8	87.7	0.49	21.0	33.1	54.6
	$N = 4800x4800$											
	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}	T_{index}	$T_{(un)pack}$	T_{comm}	T_{total}
(5x4, 4x3)→(4x5, 4x3)	1.04	1101	1797	2899	0.77	1021	1771	2793	0.08	826	1698	2524
(4x5, 4x3)→(8x2, 4x3)	0.68	1553	1128	2682	0.46	1449	1112	2561	0.04	1281	1032	2313
(40x40, 4x3)→(1x1, 4x3)	3.56	2590	1825	4419	2.51	2346	1796	4145	0.18	1290	1822	3112
(1x1, 4x3)→(40x40, 4x3)	3.59	2596	1860	4460	2.51	2339	1808	4150	0.18	1198	1805	3003
(5x4, 8x6)→(4x5, 6x4)	3.00	443	424	870	2.33	406	448	856	0.09	335	390	725
(4x5, 8x6)→(8x2, 6x4)	3.54	617	466	1087	2.65	561	467	1031	0.07	511	449	960
(40x40, 8x6)→(1x1, 6x4)	28.09	1014	571	1613	18.44	883	576	1477	0.42	409	482	891
(1x1, 8x6)→(40x40, 6x4)	10.90	1115	613	1739	10.71	905	581	1497	0.29	623	621	1244
(5x4, 6x4)→(4x5, 8x6)	6.81	461	695	1163	5.84	412	739	1157	0.22	382	672	1054
(4x5, 6x4)→(8x2, 8x6)	2.94	617	564	1184	2.29	578	522	1102	0.06	513	569	1082
(40x40, 6x4)→(1x1, 8x6)	11.03	1144	830	1985	10.69	943	740	1694	0.29	682	671	1353
(1x1, 6x4)→(40x40, 8x6)	28.68	1042	800	1871	18.15	911	735	1664	0.42	389	810	1199

Time (ms)

while $T_{index}(GBCC)$ shown in (13) is approximately

$$\frac{t \times Q + s \times P}{gcd(s \times P, t \times Q)}.$$

In this case, both $T_{index}(PITFALLS)$ and $T_{index}(ScaLAPACK)$ are approximately

$$\frac{k(t \times Q'^2 + s \times P'^2)}{gcd(s \times P', t \times Q')},$$

which depends on the value of k. $T_{index}(GBCC)$ is approximately

$$\frac{t \times Q' + s \times P'}{gcd(s \times P', t \times Q')},$$

which is independent of the value of k. Therefore, the experimental results match the theoretical analysis for this case.

Table 4 shows the indexing costs, the packing/unpacking costs, the communication costs, and the total costs for these three methods to perform test samples in Case 3. For the packing/unpacking costs and the communication costs, we have similar observations as those described in Case 1.

4.3 Experimental Results for Multidimensional Array Redistributions

All three methods can be applied to multidimensional array redistribution. Due to the page limitation, we only show experimental results for two-dimensional array. Table 5 shows the indexing costs, the packing/unpacking costs, the communication costs, and the total costs of these three methods to perform two-dimensional array redistributions on arrays with size 960×960 and $4,800 \times 4,800$. From Table 5, we can see that the proposed method outperforms the *PITFALLS* method and the *ScaLAPACK* method for all test samples. For higher dimensional array redistributions, we have similar observations as those described above.

5 Conclusions

In this paper, we have presented a generalized basic-cycle calculation method to efficiently perform a general array redistribution of BLOCK-CYCLIC(s) over P processors to BLOCK-CYCLIC(t) over Q processors. The basic idea of the *GBCC* method is to construct the packing (unpacking) pattern table for array elements in the first generalized basic-cycle of a source (destination) local array. Based on the packing (unpacking) pattern table, a source (destination) processor can pack (unpack) array elements efficiently. To evaluate the performance of the *GBCC* method, we compare it with the *PITFALLS* method and the *ScaLAPACK* method. Both theoretical analysis and experimental results were conducted for these three methods. The theoretical analysis shows that the indexing cost of the *GBCC* method is less than that of the *PITFALLS* and the *ScaLAPACK* methods. The packing/unpacking cost of the *GBCC* method is less than or equal to that of the *PITFALLS* and the *ScaLAPACK* methods. The experimental results demonstrate that the *GBCC* method outperforms the *PITFALLS* method and the *ScaLAPACK* method for all test samples.

ACKNOWLEDGMENTS

The work of this paper was partially supported by National Science Council of the Republic of China under contract NSC87-2213-E035-011.

REFERENCES

[1] S. Chatterjee, J.R. Gilbert, F.J.E. Long, R. Schreiber, and S.-H. Teng, "Generating Local Address and Communication Sets for Data Parallel Programs," *J. Parallel and Distributed Computing,* vol. 26, pp. 72-84, 1995.

[2] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu, "Fortran-D Language Specification," Technical Report TR-91-170, Dept. of Computer Science, Rice Univ., Dec. 1991.

[3] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan, "On the Generation of Efficient Data Communication for Distributed-Memory Machines," *Proc. Int'l Computing Symp.,* pp. 504-513, 1992.

[4] Y.-C. Chung, C.-H. Hsu, and S.-W. Bai, "A Basic-Cycle Calculation Technique for Efficient Dynamic Data Redistribution," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 4, pp. 359-377, Apr. 1998.

[5] J.J. Dongarra, R. Van De Geijn, and D.W. Walker, "A Look at Scalable Dense Linear Algebra Libraries," Technical Report ORNL/TM-12126 from Oak Ridge Nat'l Laboratory, Apr. 1992.

[6] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan, "On Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines," *J. Parallel and Distributed Computing,* vol. 32, pp. 155-172, 1996.

[7] High Performance Fortran Forum, "High Performance Fortran Language Specification (version 1.1)," Rice Univ., Nov. 1994.

[8] S. Hiranandani, K. Kennedy, J. Mellor-Crammey, and A. Sethi, "Compilation Method for BLOCK-CYCLIC Distribution," *Proc. ACM Int'l Conf. Supercomputing,* pp. 392-403, July 1994.

[9] E.T. Kalns and L.M. Ni, "Processor Mapping Method Toward Efficient Data Redistribution," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 12, Dec. 1995.

[10] E.T. Kalns and L.M. Ni, "DaReL: A Portable Data Redistribution Library for Distributed-Memory Machines," *Proc. Scalable Parallel Libraries Conference II,* Oct. 1994.

[11] S.D. Kaushik, C.H. Huang, R.W. Johnson, and P. Sadayappan, "An Approach to Communication Efficient Data Redistribution," *Proc. Int'l Conf. Supercomputing,* pp. 364-373, July 1994.

[12] S.D. Kaushik, C.H. Huang, J. Ramanujam, and P. Sadayappan, "Multi-Phase Array Redistribution: Modeling and Evaluation," *Proc. Int'l Parallel Processing Symp.,* pp. 441-445, 1995.

[13] S.D. Kaushik, C.H. Huang, and P. Sadayappan, "Efficient Index Set Generation for Compiling HPF Array Statements on Distributed-Memory Machines," *J. Parallel and Distributed Computing,* vol. 38, pp. 237-247, 1996.

[14] K. Kennedy, N. Nedeljkovic, and A. Sethi, "Efficient Address Generation for BLOCK-CYCLIC Distribution," *Proc. Int'l Conf. Supercomputing,* pp. 180-184, July 1995.

[15] C. Koelbel, "Compiler-Time Generation of Communication for Scientific Programs," *Supercomputing '91,* pp. 101-110, Nov. 1991.

[16] P-Z. Lee and W.Y. Chen, "Compiler Methods for Determining Data Distribution and Generating Communication Sets on Distributed-Memory Multicomputers," *Proc. 29th Hawaii Int'l Conf. System Sciences,* pp. 537-546, Jan. 1996.

[17] Y.W. Lim, P.B. Bhat, and V.K. Prasanna, "Efficient Algorithms for BLOCK-CYCLIC Redistribution of Arrays," *Proc. Eighth Symp. Parallel and Distributed Processing,* pp. 74-83, 1996.

[18] Y.W. Lim, N. Park, and V.K. Prasanna, "Efficient Algorithms for Multi-Dimensional Block-Cyclic Redistribution of Arrays," *Proc. 26th Int'l Conf. Parallel Processing,* pp. 234-241, 1997.

[19] L. Prylli and B. Tourancheau, "Fast Runtime Block Cyclic Data Redistribution on Multiprocessors," *J. Parallel and Distributed Computing,* vol. 45, pp. 63-72, Aug. 1997.

[20] S. Ramaswamy and P. Banerjee, "Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers," *Proc. Frontier '95: Fifth Symp. Frontiers of Massively Parallel Computation,* pp. 342-349, Feb. 1995.

[21] S. Ramaswamy, B. Simons, and P. Banerjee, "Optimization for Efficient Array Redistribution on Distributed Memory Multicomputers," *J. Parallel and Distributed Computing,* vol. 38, pp. 217-228, 1996.

[22] J.M. Stichnoth, D. O'Hallaron, and T.R. Gross, "Generating Communication for Array Statements: Design, Implementation, and Evaluation," *J. Parallel and Distributed Computing,* vol. 21, pp. 150-159, 1994.

[23] R. Thakur, A. Choudhary, and G. Fox, "Runtime Array Redistribution in HPF Programs," *Proc. Scalable High Performance Computing Conf.,* pp. 309-316, May 1994.

[24] R. Thakur, A. Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 6, pp. 587-594, June 1996.

[25] A. Thirumalai and J. Ramanujam, "HPF Array Statements: Communication Generation and Optimization," *Proc. Third Workshop Languages, Compilers and Run-time system for Scalable Computers,* May 1995.

[26] A. Thirumalai and J. Ramanujam, "Efficient Computation of Address Sequences in Data Parallel Programs Using Closed Forms for Basis Vectors," *J. Parallel and Distributed Computing,* vol. 38, pp. 188-203, 1996.

[27] V. Van Dongen, C. Bonello, and C. Freehill, " High Performance C–Language Specification Version 0.8.9," Technical Report CRIM-EPPP-94/04-12, 1994.

[28] C. Van Loan, "Computational Frameworks for the Fast Fourier Transform," *SIAM,* 1992.

[29] D.W. Walker and S.W. Otto, "Redistribution of BLOCK-CYCLIC Data Distributions Using MPI," *Concurrency: Practice and Experience,* vol. 8, no. 9, pp. 707-728, Nov. 1996.

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

[31] A. Wakatani and M. Wolfe, "Optimization of Array Redistribution for Distributed Memory Multicomputers," *Parallel Computing,* vol. 21, no. 9, 1995.

Ching-Hsien Hsu received the BS degree in computer science from Tung Hai University in 1995, and the PhD degree in Information Engineering from Feng Chia University in 1999, respectively. He is currently a teaching instructor in the Information & Multimedia Education Center at Fu Hsing Kang College. His research interests are in the areas of parallel and distributed computing, parallel algorithms, and high performance compilers for data parallel programming languages.

Sheng-Wen Bai received the BS and the MS degrees in Information Engineering from Feng Chia University in 1996 and 1998, respectively. He is currently a PhD student in the Department of Computer and Information Engineering at National Sun-Yet-San University. His research interests are in the areas of parallel and distributed computing, performance analysis, and high performance compilers for data parallel programming languages.

Yeh-Ching Chung received the BS degree in computer science from Chung Yuan Christian University in 1983 and the MS and PhD degrees in computer and information science from Syracuse University in 1988 and 1992, respectively. Currently, he is a professor and the chair with the Department of Information Engineering at Feng Chia University, where he directs the parallel and distributed processing laboratory. His research interests include parallel compilers, parallel programming tools, mapping, scheduling, load balancing, embedded systems, and virtual reality. He is a member of the IEEE Computer Society.

Chu-Sing Yang received the BS degree in engineering science from National Cheng Kung University in 1976, and the MS and PhD degrees in electrical engineering and institute of microelectronics from National Cheng Kung University in 1984 and 1987, respectively. Since 1993, he has been a professor in the Department of Computer Science and Engineering at National Sun Yat-Sen University. His research interests include parallel and distributed systems, mobile computing systems, and Web servers.