A Generalized Basic Cycle Calculation Method for Efficient Array Redistribution

Yeh-Ching Chung[†], *Sheng-Wen Bai*[†], *Ching-Hsien Hsu*[†], *and Chu-Sing Yang*[††]

[†] Department of Information Engineering, Feng Chia University, Taiwan, ROC
Email: ychung, swbai, chhsu@pine.iecs.fcu.edu.tw

[††] Institute of Computer and Information Engineering, National Sun Yat-sen University, Taiwan, ROC
Email : csyang@cie.nsysu.edu.tw

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)×P) in the source distribution and lcm(sP, tQ)/(gcd(s,t)×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. 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 multi-dimensional array redistributions is also presented.*

Keywords: 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), Fortran D and High Performance C (HPC) provide compiler directives for programmers to specify regular array distribution, namely, `BLOCK`, `CYCLIC`, and `BLOCK-CYCLIC`. In many scientific programs, it is necessary to change distribution fashion of a program at different phases in order to achieve better performance. Examples are multidimensional fast Fourier transform, the Alternative Direction Implicit (ADI) method for solving two-dimensional diffusion equations, linear algebra solvers, etc. 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. Many methods for performing array redistribution were proposed in the literature [2, 4, 6, 10-15]. Due to the page limitation, we will not describe these methods here. The detail information about these works can be found in [2].

In [2], we proposed a basic-cycle calculation technique to efficiently perform a `BLOCK-CYCLIC(`s`)` to `BLOCK-CYCLIC(`t`)` redistribution on the same processor set. In HPF, it supports array redistribution with arbitrary source and destination processor sets. Based on the spirit of the basic-cycle calculation technique, 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. 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. Both theoretical and experimental performance analysis were conducted for these three methods. The theoretical performance analysis shows that the indexing cost of the *GBCC* method is less than those of the *PITFALLS* method and the *ScaLAPACK* method. The packing/unpacking cost of the *GBCC* method is less than or equal to those of the *PITFALLS* method and the *ScaLAPACK* method. 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 multi-dimensional array redistributions is also presented.

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 $P-1$.

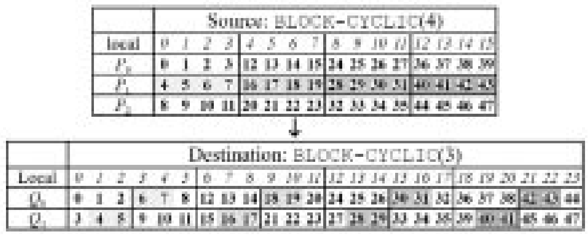
Definition 4: Given a $(s, P)\rightarrow(t, Q)$ redistribution on a one-dimensional array $A[0{:}N-1]$, the generalized basic-cycle (GBC) is defined as $GBC = \frac{lcm(s\times P, t\times Q)}{gcd(s,t)\times P}$ in the source distribution and $GBC = \frac{lcm(s\times P, t\times Q)}{gcd(s,t)\times Q}$ 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), and so on.

Definition 5: 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.

3. The *GBCC* method for Array Redistribution

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, Figure 1 shows a $(4, 3)\rightarrow(3, 2)$ redistribution on a one-dimensional array with 48 elements. According to Definition 4, the generalized basic-cycle in the source distribution and the destination distribution of the redistribution is 4 and 6, respectively. In Figure 1, 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 kth 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.

Figure 1: A $(4, 3)\rightarrow(3, 2)$ redistribution on a one-dimensional array with N=48 elements.



	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

Another example of a $(6, 4)\rightarrow(4, 3)$ redistribution on $A[0{:}95]$ is shown in Figure 2(a). The generalized basic-cycle in the source distribution and the destination distribution of the redistribution is 3 and 4, respectively. However, the observation that we obtained from Figure 1 (each generalized basic-cycle of a local array has the same communication pattern) cannot be applied to the case shown in Figure 2(a) 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, which 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 Figure 1 can be held if we use array B for the redistribution. An example of using meta-array for the array redistribution of Figure 2(a) is shown in Figure 2(b).

In the following discussion, we assume that a (s, P)→(t, Q) redistribution on $A[0{:}N-1]$ is given. We also assume that $gcd(s, t)$ is equal to 1. If $gcd(s, t)$ is not equal to 1, we use $s/gcd(s, t)$ and $t/gcd(s, t)$ as the source and destination distribution factors of the redistribution, respectively.

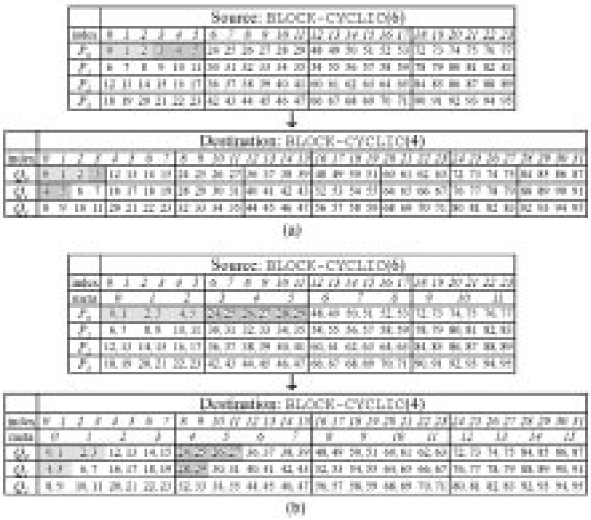


Figure 2: (a) A (6, 4)→(4, 3) redistribution with N = 96. (b) An example of using a grouped meta-array for the redistribution in (a).

3.1 Send Phase

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)→(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,

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

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

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 Equations (1) and (2). Since array elements in a source section have consecutive global array indices, for a source processors P_i, if the destination processor of $SLA_i[0{:}r-1]$ is Q_j, then the destination processors of $SLA_i[r{:}r+t-1]$, $SLA_i[r+t{:}r+2t-1]$, …, and $SLA_i[r+\lfloor (s-r)/t \rfloor \times t{:}s-1]$ are $Q_{mod(j+1,Q)}$, $Q_{mod(j+2,Q)}$, …, and $Q_{mod(j+\lfloor (s-r)/t \rfloor, Q)}$, respectively, where $1 \le r \le t$. For example, Figure 3 shows the send processor/data sets of the first generalized basic-cycle of source processors for a (10,3)→(3,4) redistribution shown in Figure 2. In Figure 3, 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, Equation (1) can be simplified as follow,

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

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,

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

Since a generalized basic-cycle has GBC/s source sections, Equations (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.

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	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
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	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
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	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Figure 3: The send processor/data sets of the first generalized basic-cycle for a (10,3)→(3,4) redistribution.

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 processors P_i, if the destination processor of $SLA_i[0{:}r-1]$ is Q_j, then the destination processors of $SLA_i[r{:}r+t-1]$, $SLA_i[r+t{:}r+2t-1]$, …, and $SLA_i[r+\lfloor(s-r)/t\rfloor\times t{:}s-1]$ are $Q_{mod(j+1,Q)}$, $Q_{mod(j+2,Q)}$, …, and $Q_{mod(j+\lfloor(s-r)/t\rfloor,Q)}$, respectively, where $1 \le r \le t$. 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 Figure 3, for source processor P_1, its corresponding packing pattern table is give 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, and so on. 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.

3.2 Receive Phase

Similar to 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:

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

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

$$rgindex_{ji}(k) = k \times Q + j \times t \quad (7)$$

where k is the local array index of the first array element of a source section. The algorithm of the *GBCC* method is given as follows.

Algorithm GBCC (s, P, t, Q)
/* Send Phase */
i = *get_myrank_of_source_processors*();
call PPT_construction(i, s, P, t, Q);
for j = 0 to $Q-1$
    if $c_j > 0$ then
        pack data from source local array to a message according to $PPT_i[j]$;
        send message to Q_j;
    endif
endfor
/* Receive Phase */
j = *get_myrank_of_destination_processors*();
call UPT_construction(j, s, P, t, Q);
for i = 0 to $P-1$
    if $c_i > 0$ then
        receive message from P_i;
        unpack received message to destination local array according to $UPT_j[i]$;
    endif
endfor
wait for all communication;
End_of_GBCC

3.3 The *GBCC* method for Multi-Dimensional Array Redistribution

The *GBCC* method can be easily extended to perform multi-dimensional 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 tables, array elements that will be sent to the same destination processor are packed dimension by dimension starting from the first (last) dimension if 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 array is in column-major (row-major).

The algorithm for the *GBCC* method to perform multi-dimensional array redistribution is given as follows:

Algorithm GBCC_MD (s[], P[], t[], Q[])
/* Send Phase */
i[]=*ranks_of_each_dimension*();
for d = 0 **to** *number_of_dimension*
    call PPT_construction($i[d]$, $s[d]$, $P[d]$, $t[d]$, $Q[d]$);
endfor

```
    for j[] = 0 to Q[] − 1
      if c_{j[]} > 0 then
        pack data from source local array to a message
          according to PPT_{i[]}[j[]];
        send message to Q_{j[]};
      endif
    endfor
   /* Receive Phase */
    j[]=ranks_of_each_dimension();
    for d = 0 to number_of_dimension
      call UPT_construction(j[d], s[d], P[d], t[d], Q[d]);
    endfor
    for i[] = 0 to P[]−1
      if c_{i[]} > 0 then
        receive message from P_{i[]};
        unpack received message to destination local
          array according to UPT_{j[]} [i[]];
      endif
    endfor
    wait for all communication;
End_of_GBCC_MD
```

4. 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 and experimental performance evaluations were conducted. We first develop 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 [13] and [12], 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 $(s, P)\rightarrow(t, Q)$ 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 follow:

$$T = T_{comp.} + T_{comm.} \tag{8}$$

For the same redistribution, the total number of messages and the size of messages sent and received by each processor are the same for these three methods. Although they all use asynchronous communication schemes, we assume that the communication costs of these three methods are the same in our theoretical model. Therefore, we will focus on the analysis of the computation costs of these 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,

$$T_{comp} = T_{index} + T_{(un)pack}, \tag{9}$$

where T_{index} and $T_{(un)pack}$ are the indexing cost and the packing/unpacking cost of a redistribution, respectively. In our analysis, the packing/unpacking cost is represented in terms of the number of data-movement operations. For the *PITFALLS* method, the indexing cost for a processor to perform the efficient FALLS intersection algorithm [13] is

$$T_{index}(PITFALLS)= O\left(\frac{lcm(s\times P, t\times Q)}{min(s, t\times Q)\times P}\times Q + \frac{lcm(s\times P, t\times Q)}{min(t, s\times P)\times Q}\times P\right) \tag{10}$$

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

$$T_{(un)pack}(PITFALLS) = O\left(\frac{N/P + N/Q}{min(s,t)}\right) \tag{11}$$

For the *ScaLAPACK* method [12], the indexing cost for a processor to determinate the send processor/data sets is

$$T_{index}(ScaLAPACK)= O\left(\frac{lcm(s\times P, t\times Q)}{min(s, t\times Q)\times P}\times Q + \frac{lcm(s\times P, t\times Q)}{min(t, s\times P)\times Q}\times P\right) \tag{12}$$

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

$$T_{(un)pack}(ScaLAPACK) = O\left(\frac{N/P + N/Q}{min(s,t)}\right) \tag{13}$$

From Equations (10) to (13), we can see that the *ScaLAPACK* method and the *PITFALLS* method have the same indexing and packing/unpacking time complexities.

For the *GBCC* method, according to the algorithm presented in Sections 3.1 and 3.2, the indexing cost is

$$T_{index}(GBCC) = O\left(\frac{lcm(s\times P, t\times Q)}{min(s,t)\times P} + \frac{lcm(s\times P, t\times Q)}{min(s,t)\times Q}\right) \tag{14}$$

According to Sections 3.1 and 3.2, the packing/unpacking cost of the generalized basic calculation method can be classify 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. Figure 4 gives an example to show this behavior. For the second class $t > s{\times}P$, there are similar behaviors in the packing/unpacking.

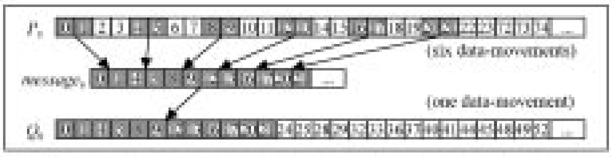


Figure 4: Given a $(24, 3){\rightarrow}(2, 2)$ redistribution, the shadowed array elements in a source section of SLA_0 will be sent from P_0 to Q_0. There are six data-moving operations and one data-moving operation in the sending phase and the receiving phase, respectively.

The indexing costs of these three classes are given as follows:

$$T_{(un)pack}(GBCC) = \begin{cases} O\left(\dfrac{N/P}{t} + \dfrac{N/Q}{s/Q}\right) & \text{if } s > t \times Q \\ O\left(\dfrac{N/P}{t/P} + \dfrac{N/Q}{s}\right) & t > s \times Q \\ O\left(\dfrac{N/P + N/Q}{min(s,t)}\right) & \text{otherwise} \end{cases} \quad (15)$$

From the above analysis, the indexing cost of the *GBCC* method is less than that of the *PITFALLS* method and the *ScaLAPACK* method. The packing/unpacking cost of the *GBCC* method is less than or equal to that of the *PITFALLS* method and the *ScaLAPACK* method. Table 1 summarizes the indexing costs and the packing/unpacking costs of these three 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 the single program multiple data (SPMD) programming paradigm with C+MPI codes. 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: $s \le t{\times}Q$ and $t \le s{\times}P$,
case 2: $s > t{\times}Q$ or $t > s{\times}P$,
case 3: $P = kP'$, $Q = kQ'$ where $gcd(P', Q') = 1$ and $k \ge 1$,

For each case, 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 experimental results for two-dimensional array redistributions.

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 one-dimensional array with N array elements.

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

Case 1: $s \le t{\times}Q$ and $t \le s{\times}P$

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 = 80000 and N = 20000000. From Table 2, we can see that the indexing costs of the *GBCC* method are less than those of the *ScaLAPACK* method and the *PITFALLS* method for all test samples. We also observe that the indexing costs of these three methods are independent of the array size. These phenomena match the indexing cost models presented in Section 4.1.

According to Table 1, in this case, these three methods have the same packing/unpacking costs. However, from Table 2, we can see that the packing/unpacking costs of the *GBCC* method are less than those of the *ScaLAPACK* method which are less than those of the *PITFALLS* method for all test samples. The reason of this situation is that the *GBCC* method uses a simpler computation approach than that of the *ScaLAPACK* method which uses a simpler computation approach than that of the *PITFALLS* method when packing/unpacking array elements.

For the communication costs, these three methods use asynchronous communication schemes. There has no clear winner in the communication cost part for all test samples due to the character of an asynchronous communication scheme. These three methods have approximately the same communication costs for all test samples.

Table 2: 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 = 80000 and N = 20000000.

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}
(3, 8)→(2, 5)	0.229	11.95	5.72	17.9	0.155	11.02	5.13	16.3	0.029	9.19	4.88	14.1
(50, 8)→(20, 3)	0.228	2.25	5.42	7.9	0.148	2.20	5.23	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
(3, 5)→(2, 8)	0.816	8.85	4.80	14.5	0.807	8.25	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	5.68	4.36	10.0
(50, 10)→(20, 10)	0.358	1.40	5.34	7.1	0.308	1.27	5.22	6.9	0.057	1.28	3.98	5.3
(6, 10)→(5, 10)	0.421	4.21	4.17	8.8	0.386	3.98	4.03	8.4	0.052	3.45	4.10	7.6
(5, 50)→(2, 50)	1.625	1.32	4.86	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	3.51	5.4	0.039	0.38	3.20	3.6
(6, 50)→(5, 50)	1.795	0.95	3.16	5.9	1.831	0.95	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}
(3, 8)→(2, 5)	0.238	3081	886	3967	0.160	2844	858	3702	0.030	2426	824	3250
(50, 8)→(20, 3)	0.233	737	938	1675	0.156	691	981	1672	0.030	632	950	1602
(4, 8)→(5, 5)	1.159	2065	873	2939	1.112	1965	936	2900	0.243	1910	965	2875
(3, 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
(6, 10)→(5, 10)	0.446	1243	797	2040	0.391	1175	839	2014	0.053	1045	795	1840
(5, 50)→(2, 50)	1.632	375	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
(6, 50)→(5, 50)	1.867	249	216	467	1.831	234	216	452	0.055	210	210	420

Time (ms)

Case 2: $s > t \times Q$ or $t > s \times P$

Table 3 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 = 80000 and N = 20000000. From Table 3, for the indexing costs, we have similar observations as those described for Case 1.

From Table 3, we can see the packing/unpacking costs of these three methods depend on the array size. Therefore, when the local array size is large, the performance of a packing/unpacking method plays an important role in a redistribution. From Table 3, for the same test sample with array size N = 20000000, we can see that the packing/unpacking cost of the *GBCC* method is much less than those of the *PITFALLS* method and the *ScaLAPACK* method. These phenomena match the theoretical performance analysis presented in Section 4.1. Therefore, the packing/unpacking method provided in the *GBCC* method outperforms those of provided in the *PITFALLS* method and the *ScaLAPACK* method for this case.

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

Figure 5 shows the indexing costs of $(s, kP') \rightarrow (t, kQ')$ redistributions with array size N = 20000000, where k = 1 to 5. From Figure 5, 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 Equations (10) and (12) are approximately to $\frac{t \times Q^2 + s \times P^2}{gcd(s \times P, t \times Q)}$ while $T_{index}(GBCC)$ shown in Equation (14) is approximately to $\frac{t \times Q + s \times P}{gcd(s \times P, t \times Q)}$. In this case, both $T_{index}(PITFALLS)$ and $T_{index}(ScaLAPACK)$ are approximately to $\frac{k(t \times Q'^2 + s \times P'^2)}{gcd(s \times P', t \times Q')}$ which depends on the value of k. $T_{index}(GBCC)$ is approximately to $\frac{t \times Q' + s \times P'}{gcd(s \times P', t \times Q')}$ which is independent of the value of k. Therefore, the experimental results match the theoretical analysis for this case.

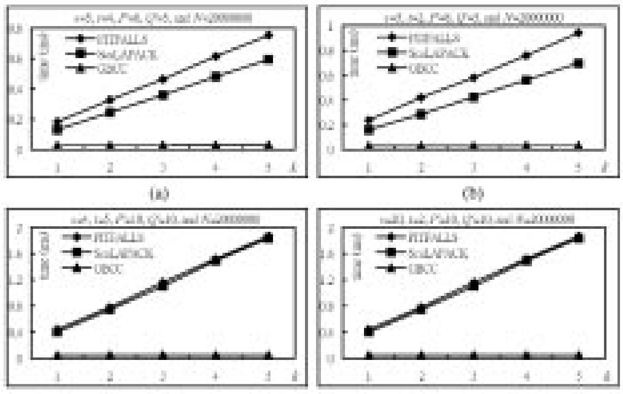


Figure 5: The indexing costs of the $(s, kP') \rightarrow (t, kQ')$ redistribution where k = 1, 2, 3, 4, and 5.

Experimental Results for Two-dimensional array redistributions

Table 4 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 960x960 and 4800x4800. From Table 4, we can see that the proposed method outperforms the *PITFALLS* method and the *ScaLAPACK* method for all test samples.

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. To evaluate the performance of the *GBCC* method, we compare it with the *PITFALLS* method and the *ScaLAPACK* method. Both theoretical and experimental performance analysis were conducted for these three methods. The theoretical performance analysis shows that the indexing cost of the *GBCC* method is less than that of the *PITFALLS* method and the *ScaLAPACK* method. The packing/unpacking cost of the *GBCC* method is less than or equal to those of the *PITFALLS* method and the *ScaLAPACK* method. The experimental results demonstrate that the *GBCC* method outperforms the *PITFALLS* method and the *ScaLAPACK* method for all test samples.

References

[1] 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. of Intl. Computing Symposium,* pp. 504-513, 1992.

[2] Y.-C. Chung, C.-H. Hsu, and S.-W. Bai, "A Basic-Cycle Calculation Technique for Efficient Dynamic Data Redistribution," *IEEE Trans. on PDS*, Vol. 9, No. 4, pp. 359-377, April 1998.

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

[4] Edgar T. Kalns, and Lionel M. Ni, "Processor Mapping Method Toward Efficient Data Redistribution, " *IEEE Trans. on PDS*, vol. 6, no. 12, December 1995.

[5] E. T. Kalns and L. M. Ni, "DaReL: A portable data redistribution library for distributed-memory machines," in *Proc. of the 1994 Scalable Parallel Libraries Conference II*, Oct. 1994.

[6] S. D. Kaushik, C. H. Huang, J. Ramanujam, and P. Sadayappan, "Multi-phase array redistribution: Modeling and evaluation," In *Proc. of International Parallel processing Symposium*, pp. 441-445, 1995.

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

[8] C. Koelbel," Compiler-time generation of communication for scientific programs," In *Supercomputing'91*, pp. 101-110, Nov. 1991.

[9] P-Z. Lee and W. Y. Chen, "Compiler methods for determining data distribution and generating communication sets on distributed-memory multicomputers," *29th IEEE Hawaii Intl. Conf. on System Sciences*, Maui, Hawaii, pp.537-546, Jan 1996.

[10] Young Won Lim, Prashanth B. Bhat, and Viktor, K. Prasanna, "Efficient Algorithms for BLOCK-CYCLIC Redistribution of Arrays," *Proc. of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pp. 74-83, 1996.

[11] Y. W. Lim, N. Park, and V. K. Prasanna, "Efficient Algorithms for Multi-Dimensional Block-Cyclic Redistribution of Arrays," *Proc. of the 26th ICPP*, pp. 234-241, 1997.

[12] L. Prylli and B. Tourancheau, "Fast Runtime Block Cyclic Data Redistribution on Multiprocessors," *JPDC*, Vol. 45, pp. 63-72, Aug. 1997.

[13] S. Ramaswamy, B. Simons, and P. Banerjee, "Optimization for Efficient Array Redistribution on Distributed Memory Multicomputers," *JPDC,* Vol. 38, pp. 217-228, 1996.

[14] Rajeev. Thakur, Alok. Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution, " *IEEE Trans. On PDS*, vol. 7, no. 6, pp. 587-594, JUNE 1996.

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

Table 3: 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 = 80000 and N = 20000000.

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}
(300, 8)→(3, 5)	16.2	6.7	6.1	29.0	12.2	4.6	5.2	21.9	3.1	2.7	4.5	10.3
(3, 8)→(300, 5)	16.2	7.3	3.9	23.4	10.9	6.0	4.9	21.8	3.8	4.8	4.9	12.7
(300, 8)→(1, 5)	13.9	15.7	3.7	33.3	7.3	13.5	5.9	26.7	1.4	5.6	4.9	11.9
(1, 8)→(300, 5)	16.1	16.2	7.5	39.8	9.3	13.5	6.1	28.9	1.6	8.8	3.9	14.6
(300, 3)→(3, 8)	16.2	7.4	3.3	22.9	10.8	5.1	5.2	21.2	3.8	4.1	3.7	13.6
(3, 3)→(300, 8)	16.2	6.9	4.9	28.0	12.1	4.9	4.8	21.7	3.1	2.8	3.2	11.1
(300, 3)→(1, 8)	16.1	16.5	3.6	32.2	9.3	13.9	5.3	28.5	1.6	8.7	3.4	15.7
(1, 3)→(300, 8)	13.9	15.9	3.1	36.9	7.3	13.7	5.0	26.0	1.3	3.2	8.1	12.6
(300, 10)→(3, 10)	12.9	4.6	2.6	28.1	12.9	3.2	2.9	18.6	2.3	2.2	2.9	7.4
(3, 10)→(300, 10)	12.9	4.6	2.6	28.1	12.8	3.2	2.5	18.5	2.4	2.2	2.8	7.4
(300, 10)→(1, 10)	12.6	10.1	3.1	25.8	10.3	8.5	2.6	21.4	1.0	4.3	2.7	8.2
(1, 10)→(300, 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}
(300, 8)→(3, 5)	16.3	2521	799	3336	12.1	2436	829	3277	3.1	1395	800	2198
(3, 8)→(300, 5)	16.3	2581	798	3389	10.9	2467	885	3363	3.8	1501	785	2382
(300, 8)→(1, 5)	16.0	5084	1203	6303	7.3	4845	957	5807	1.4	2598	1126	3787
(1, 8)→(300, 5)	16.1	5398	914	6293	9.4	5886	923	6818	1.7	3359	531	4278
(300, 3)→(3, 8)	16.6	2544	809	3357	10.9	2476	887	3354	3.8	2182	859	3036
(3, 3)→(300, 8)	16.5	2500	857	3373	12.2	2419	820	3252	3.1	1363	818	2182
(300, 3)→(1, 8)	16.1	5601	1050	6671	9.4	5431	959	6398	1.6	3335	1124	4461
(1, 3)→(300, 8)	16.0	5381	876	6273	7.3	5079	1080	6189	1.4	2572	939	3512
(300, 10)→(3, 10)	12.9	1482	811	2326	12.9	1436	744	2193	2.4	1063	821	1886
(3, 10)→(300, 10)	12.9	1482	834	2325	12.9	1426	819	2258	2.4	1024	795	1821
(300, 10)→(1, 10)	12.6	2080	931	4027	10.4	2938	974	3922	1.0	2014	882	2897
(1, 10)→(300, 10)	12.6	3188	988	4189	10.3	2951	1053	4014	1.0	2018	1143	3162

Time (ms)

Table 4: 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 960x960 and 4800x4800.

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}
(2x4, 4x3)→(4x5, 4x3)	1.80	48.1	86.8	135.9	0.79	41.1	85.3	127.1	0.08	34.4	82.8	117.3
(4x5, 4x3)→(8x2, 4x3)	0.66	65.8	50.9	117.1	0.45	59.7	50.3	110.4	0.04	53.2	49.2	102.4
(40x40, 4x3)→(1x1, 4x3)	3.31	124.6	89.4	217.3	2.49	97.2	83.0	181.7	0.18	56.2	79.3	135.9
(1x1, 4x3)→(40x40, 4x3)	3.37	123.9	91.1	218.6	2.49	99.7	80.5	182.7	0.18	32.4	82.7	115.3
(2x4, 8x6)→(4x5, 6x4)	2.87	19.8	26.6	49.3	2.50	16.1	24.8	43.2	0.09	13.6	21.3	35.2
(4x5, 8x6)→(8x2, 6x4)	3.43	28.5	21.4	53.1	2.60	22.7	21.6	46.9	0.07	21.4	21.2	42.7
(40x40, 8x6)→(1x1, 6x4)	26.40	57.9	33.8	118.3	18.17	39.2	27.3	84.7	0.42	19.8	24.8	45.4
(1x1, 8x6)→(40x40, 6x4)	11.88	78.0	38.6	117.7	10.78	42.7	26.4	79.9	0.29	35.3	28.2	64.8
(2x4, 6x4)→(4x5, 8x6)	6.60	24.5	28.1	59.2	5.80	17.2	23.7	46.7	0.22	17.2	25.8	43.2
(4x5, 6x4)→(8x2, 8x6)	2.80	28.0	27.1	55.9	2.25	22.6	25.7	50.5	0.06	20.9	26.9	47.9
(40x40, 6x4)→(1x1, 8x6)	30.69	76.3	32.5	119.5	10.64	45.3	25.7	81.6	0.29	38.3	22.2	61.8
(1x1, 6x4)→(40x40, 8x6)	26.38	58.9	29.6	114.8	18.23	40.7	28.6	87.7	0.49	21.8	33.3	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}
(2x4, 4x3)→(4x5, 4x3)	1.88	1101	1797	2899	0.77	1021	1771	2793	0.08	826	1698	2524
(4x5, 4x3)→(8x2, 4x3)	0.66	1535	1128	2662	0.46	1449	1112	2561	0.04	1261	1052	2313
(40x40, 4x3)→(1x1, 4x3)	3.36	2590	1825	4419	2.51	2346	1795	4145	0.18	1290	1822	3112
(1x1, 4x3)→(40x40, 4x3)	3.39	2596	1860	4460	2.51	2339	1808	4150	0.18	1198	1805	3000
(2x4, 8x6)→(4x5, 6x4)	3.00	443	424	870	2.33	406	448	856	0.09	335	398	729
(4x5, 8x6)→(8x2, 6x4)	3.34	617	466	1087	2.65	364	467	1031	0.07	311	449	958
(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
(2x4, 6x4)→(4x5, 8x6)	6.81	461	685	1163	5.84	412	739	1157	0.22	382	672	1054
(4x5, 6x4)→(8x2, 8x6)	2.94	617	564	1184	2.29	378	522	1002	0.06	313	368	1082
(40x40, 6x4)→(1x1, 8x6)	11.83	1144	830	1985	10.69	943	740	1694	0.29	682	673	1353
(1x1, 6x4)→(40x40, 8x6)	28.60	1042	800	1871	18.15	911	735	1664	0.42	399	818	1199

Time (ms)