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 \( \text{lcm}(sP, tQ)/(\text{gcd}(s, t) \times Q) \) in the source distribution and \( \text{lcm}(sP, \ Q)/(\text{gcd}(s, t) \times Q) \) 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 runtime, 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 \( Q \).
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 \( \text{lcm}(sP, tQ)/(\gcd(s, t) \times P) \) in the source distribution and \( \text{lcm}(sP, tQ)/(\gcd(s, t) \times Q) \) 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

<table>
<thead>
<tr>
<th>( P_0 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

(a)
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 local source 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 \(SLA_i\) of source processor \(P_i\) is \(SLA_i[0 : 19] = \{A[10], \ldots, A[19], A[40], \ldots, A[49]\}\).
SLA\(_{1}[0:19]\) can be divided into two section sizes (size = 10), SLA\(_{1}[0:9]\) and SLA\(_{1}[0:19]\). The second generalized basic-cycle in SLA\(_{1}\) of source processor \(P_i\) 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 of destination processor \(Q_1\) is

\[
DLA_{1}[0:14] = \{A[3], \ldots, A[5], A[15], \ldots, A[17], A[27], \ldots, A[29], A[39], \ldots, A[41], A[51], \ldots, A[53]\}.
\]

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

\[
DLA_{1}[15:29] = \{A[63], \ldots, A[65], A[75], \ldots, A[77], A[87], \ldots, A[89], A[99], \ldots, A[101], A[111], \ldots, A[113]\}.
\]

### 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, element indices 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 \(SLA[k]\) (\(DLA[k]\)) is \(Q_j\) (\(P_i\)), then the destination (source) processors of \(SLA[k+GBC]\), \(SLA[k+2GBC]\), \(\ldots\), \(SLA[k+N/P-GBC]\) (\(DLA[k+GBC]\), \(DLA[k+2GBC]\), \(\ldots\), \(DLA[k+N/Q-GBC]\) \(\ldots\)) 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,

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

For a source processor \(P_i\), if the global array index of \(SLA[k]\) is \(\alpha\), then the global array indices of \(SLA[k+GBC]\), \(SLA[k+2GBC]\), \(\ldots\), and \(SLA[k+N/P-GBC]\) are \(\alpha + \text{lcm}(s \times P, t \times Q), \alpha + 2 \times \text{lcm}(s \times P, t \times Q), \ldots\), and \(\alpha + (N - \text{lcm}(s \times P, t \times Q))\), respectively, where \(0 \leq i \leq P-1, 0 \leq k \leq GBC-1\) and \(0 \leq \alpha \leq \text{lcm}(s \times P, t \times Q)\) – 1. Since \(\text{lcm}(s \times P, t \times Q)\) is a multiple of \(t \times Q\) in the destination distribution, if \(A[\alpha]\) is distributed to the
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 \( SLA_i[k \times gcd(s, t) : (k + 1) \times gcd(s, t) - 1] \) in \( (s, P_i) \rightarrow (t, Q_j) \) redistribution is \( \alpha \times gcd(s, t) \), then the global array index of \( SLA_i[k \times gcd(s, t) : (k + 1) \times gcd(s, t) - 1] \) in \( (s, P_i) \rightarrow (t, Q_j) \) 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 \((t/gcd(s, t), Q_j)\) redistribution, \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \). If \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor \( P_j \), so are \( A[\alpha + lcm(s \times P, t \times Q)], A[\alpha + lcm(s \times P, t \times Q)], \ldots, A[\alpha + (N - lcm(s \times P, t \times Q))] \), where \( 0 \leq j \leq Q - 1 \).

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

\[(DLA_i[k \times gcd(s, t) : (k + 1) \times gcd(s, t) - 1])\]

in \((s, P) \rightarrow (t, Q)\) redistribution will also be \( Q_j(P_i) \), where \( 0 \leq k \leq \lfloor N/(P \times gcd(s, t)) \rfloor \) and \( 0 \leq k < \lfloor N/(Q \times gcd(s, t)) \rfloor \).

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 \( SLA_i[k \times gcd(s, t) : (k + 1) \times gcd(s, t) - 1] \) in \((s/gcd(s, t), P) \rightarrow (t/gcd(s, t), Q)\) redistribution is \( \alpha \), 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 \((t/gcd(s, t), Q_j)\) redistribution, then \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor of \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \). If \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor of \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \). If \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor of \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \). If \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor of \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \). If \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor of \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \). If \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor of \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \). If \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor of \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \). If \( A[0 : N-1] \) is distributed by BLOCK-CYCLIC \((t/gcd(s, t), Q_j)\) redistribution, then the destination processor of \( A[0 : N-1] \) is in the \([\alpha \times gcd(s, t), \alpha + 1 \times gcd(s, t)]\) block of size \( t/gcd(s, t) \).
and send processor/data sets of the first generalized basic-cycle, respectively, where

g = \gcd(s, t)

use 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 \( SLA[k] \) in \( SLA[0 : GBC - 1] \) of source processor \( P \) can be determined by the following equations,

\[
sgindex(i, k) = \lfloor k/s \rfloor \times s \times P + i \times s + \text{mod}(k, s),
\]

\[
d_p(sgindex(i, k)) = \text{mod}(\lfloor sgindex(i, k)/t \rfloor, Q),
\]

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

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 \( P \), if the destination processor of \( SLA[0 : r - 1] \) is \( Q \), then the destination processors of \( SLA[r : r + t - 1], SLA[r + t : r + 2t - 1], \ldots \), and \( SLA[r + \lfloor (s - r)/t \rfloor \times t : s - 1] \) are \( Q_{mod(j+1,Q)}, Q_{mod(j+2,Q)}, \ldots, Q_{mod(j+\lfloor (s-r)/t \rfloor,Q)} \), 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 \( P \), the destination processor of \( SLA[0 : r - 1] = SLA[0 : 1] \) is \( Q_1 = Q_{10} \), where \( r = 2 \) and \( j = 3 \). The destination processors of \( SLA[r : r + t - 1] = SLA[2 : 4], SLA[r + t : r + 2t - 1] = SLA[5 : 7], \) and \( SLA[r + \lfloor (s - r)/t \rfloor \times t : s - 1] = SLA[8 : 9] \) are \( Q_{mod(j+1,Q)}, Q_{mod(j+2,Q)}, \) \( Q_1 \), and \( Q_{mod(j+\lfloor (s-r)/t \rfloor,Q)} = Q_{10} \), 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:

\[
sgindex(k) = k \times P + i \times s,
\]

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,

\[
r = (\lfloor sgindex(k)/t \rfloor + 1) \times t - sgindex(k),
\]

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 \( P \), if the destination processor of \( SLA[0 : r - 1] = Q_j \), then the destination processors of \( SLA[r : r + t - 1], SLA[r + t : r + 2t - 1], \ldots, \) and \( SLA[r + \lfloor (s - r)/t \rfloor \times t : s - 1] \) 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 \( P \), we can construct a packing pattern table \( PPT[0 : Q - 1] \) 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 \( P \)’s corresponding packing pattern table is given as follows:

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

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 \(PPT[0 : 3]\), when packing array elements whose destination processor is \(Q_0\) into \(message_0\), the entry \(PPT[0][0] = [2, 3, [18, 2]]\) will be used. According to \(PPT[0][0] = [2, 3, [18, 2]]\), source processor \(P_1\) will pack array elements \(SLA[2 : 4]\) and \(SLA[18 : 19]\) in the first generalized basic-cycle of \(SLA\) into \(message_0[0 : 2]\) (descriptor \([2, 3]\)) and \(message_0[3 : 4]\) (descriptor \([18, 2]\)), respectively. Array elements \(SLA[2 + GBC : 4 + GBC]\) and \(SLA[18 + GBC : 19 + GBC]\) in the second generalized basic-cycle of \(SLA\) 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 \(P_j\) is equal to (the number of descriptors in \(PPT[0 : Q - 1]\)) \times (the number of generalized basic-cycles in \(SLA\)), 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. \(gcdst = gcds(t, s); \ t = t/gcdst;\)
2. calculate the GBC for the sending phase; \(lastp = -1;\)
3. for \(m = 0\) to \(GBC/1\)
4. \(k = m \times s; \ \text{gcdx} = k \times P + i \times s; \ \text{second} = \text{gcdx} + s;\)
5. \(j = \text{mod}(\text{gcdx}/t, Q); \ \text{link} = (\min((\text{gcdx}/t + 1) \times t, \text{second}) - \text{gcdx});\)
6. if \(j = \text{lastp}\) then
7. \(PPT[i][j][c - 1].len = \text{gcdx} + s;\)
8. \(k += 1; \ \text{gcdx} += 1; \ t = t; \ j = \text{mod}(j + 1, Q);\)
9. endif
10. while \(\text{gcdx} < \text{send}\)
11. \(l = \min(l, \text{second} - \text{gcdx});\)
12. \(PPT[i][j][c].pos = k \times \text{gcdst}; \ PPT[i][j][c].len = l \times \text{gcdst};\)
13. \(c += 1; \ k += 1; \ \text{gcdx} += 1; \ t = t;\)
14. \(lastp = j; \ j = \text{mod}(j + 1, Q);\)
15. endwhile
16. endfor
\(End\_of\_PPT\)-construction

### 3.2 The Receive Phase

In the receive phase, techniques for the indexing and the 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 \((s, P) \rightarrow (t, Q)\) redistribution on \(A[0 : N - 1]\), for destination processor \(Q_j\), the source processor of array element \(DLA[k]\) in \(DLA[0 : GBC - 1]\) can be determined by the following equations:

\[
rgindex_j(k) = \left\lfloor \frac{k/t} \times t + Q + j \times t + \text{mod}(k, t) \right\rfloor
\]

\[
s_p(j) = \text{mod}(\text{rgindex}_j(k)/s, P)
\]

where \(k = 0\) to \(GBC - 1\). The function \(rgindex_j(k)\) converts the local array index of an array element in a destination local array to its corresponding global array index, i.e., \(DLA[k] = A[\text{rgindex}_j(k)]\). The function \(s_p(j)\) is used to determine the source processor of the global array element \(A[\text{rgindex}_j(k)]\).

Since array elements in a destination section have consecutive global array indices, for a destination processor \(Q_j\), if the source processor of \(DLA[0 : u - 1]\) is \(P_i\), then the source processors of \(DLA[0 : u + s - 1]\), \(DLA[0 : u + 2s - 1]\), \(DLA[0 : u + t - u)/s \times s - s - 1]\), \(DLA[u + \ldots]\) are \(P_{mod(i+1,P)}, P_{mod(i+2,P)}, \ldots, P_{mod(i+([t-u)/s,P)},\) 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:

\[
rgindex_j(k) = k \times Q + j \times t,
\]

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:

\[
u = \left\lfloor \frac{\text{rgindex}_j(z)}{s} + 1 \right\rfloor \times s - \text{rgindex}_j(z).
\]
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
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 Multidimensional 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...
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

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Indexing costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PITFALLS</td>
<td>(O\left( \frac{\text{lcm}(s \times P, t \times Q)}{\min(s, t \times Q)} \times P \right) + \frac{\text{lcm}(s \times P, t \times Q)}{\min(t, s \times Q)} \times Q )}</td>
</tr>
<tr>
<td>ScaLAPACK</td>
<td>(O\left( \frac{\text{lcm}(s \times P, t \times Q)}{\min(s, t \times P)} \right) + \frac{\text{lcm}(s \times P, t \times Q)}{\min(t, s \times t)} \times Q )}</td>
</tr>
<tr>
<td>GBCC</td>
<td>(O\left( \frac{N/P + N/Q}{\min(s, t)} \right) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Packing/unpacking costs</th>
</tr>
</thead>
</table>
| PITFALLS    | \(s > t \times Q\) \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad... 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 \((P_{1x} \rightarrow Q_{1x})\), the packing pattern table for destination processor \(Q_{1x}\) is \(PPT_{1x}[1] = \{1, 2\}\). For the second dimension \((P_{1x} \rightarrow Q_{1x})\), the packing pattern table for destination processor \(Q_{1x}\) is \(PPT_{1x}[2] = \{0, 1, 7, 1, 9, 3\}\). Assume that array elements are stored in memory in a row-major manner. From Fig. 6, for the source processor \(P_{1x1}\), we can see that the array elements in \(SLA_0\) 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 \(PPT_{1x}[2]\), source processor \(P_{1x1}\) can pack array elements \(SLA_{1x1}[1, 0], SLA_{1x1}[1, 1], SLA_{1x1}[1, 7], SLA_{1x1}[1, 9], SLA_{1x1}[1, 10], SLA_{1x1}[1, 11]\), \(SLA_{1x2}[2, 0], SLA_{1x2}[2, 1], SLA_{1x2}[2, 7], SLA_{1x2}[2, 9], SLA_{1x2}[2, 10]\), and \(SLA_{1x2}[2, 11]\) into \(message_{1x1}[0 : 11]\) according to \(PPT_{1x}[1]\) and \(PPT_{1x}[2]\). For the second generalized basic-cycle of \(SLA_{1x2}\) in the first dimension, array elements \(SLA_{1x2}[1, 7], SLA_{1x2}[1, 7], SLA_{1x2}[1, 7], SLA_{1x2}[1, 7], SLA_{1x2}[1, 7], SLA_{1x2}[1, 7], SLA_{1x2}[1, 7], SLA_{1x2}[1, 7], SLA_{1x2}[1, 7], and SLA_{1x2}[1, 7]\) will be packed into \(message_{1x2}[12 : 23]\) according to \(PPT_{1x}[1]\) and \(PPT_{1x}[2]\). 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...
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 \((s, P) \rightarrow (t, Q)\) 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 follows:

\[
T = T_{\text{comp}} + T_{\text{comm}},
\]

where \(T_{\text{comp}}\) 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 \(T_{\text{comm}}\) is the communication time for an algorithm to send and receive data among processors. We said that \(T_{\text{comp}}\) and \(T_{\text{comm}}\) 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,

\[
T_{\text{comp}} = T_{\text{index}} + T_{\text{(un)pack}},
\]

where \(T_{\text{index}}\) and \(T_{\text{(un)pack}}\) 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>
<thead>
<tr>
<th>methods cases</th>
<th>PITFALLS</th>
<th>ScaLAPACK</th>
<th>GBCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>((5, 8) \rightarrow (2, 5))</td>
<td>0.229</td>
<td>11.95</td>
<td>5.72</td>
</tr>
<tr>
<td>((50, 5) \rightarrow (20, 8))</td>
<td>0.816</td>
<td>2.02</td>
<td>5.06</td>
</tr>
<tr>
<td>((4, 5) \rightarrow (5, 8))</td>
<td>0.169</td>
<td>6.86</td>
<td>4.57</td>
</tr>
<tr>
<td>((5, 10) \rightarrow (2, 10))</td>
<td>0.361</td>
<td>7.06</td>
<td>6.48</td>
</tr>
<tr>
<td>((50, 10) \rightarrow (20, 10))</td>
<td>0.358</td>
<td>1.40</td>
<td>5.34</td>
</tr>
<tr>
<td>((4, 10) \rightarrow (5, 10))</td>
<td>0.421</td>
<td>4.21</td>
<td>4.17</td>
</tr>
<tr>
<td>((5, 50) \rightarrow (2, 50))</td>
<td>1.625</td>
<td>1.52</td>
<td>4.66</td>
</tr>
<tr>
<td>((5, 50) \rightarrow (20, 50))</td>
<td>1.611</td>
<td>0.38</td>
<td>4.11</td>
</tr>
<tr>
<td>((4, 50) \rightarrow (5, 50))</td>
<td>1.795</td>
<td>0.95</td>
<td>3.16</td>
</tr>
</tbody>
</table>

\(N = 80,000\)

<table>
<thead>
<tr>
<th>methods cases</th>
<th>PITFALLS</th>
<th>ScaLAPACK</th>
<th>GBCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>((5, 8) \rightarrow (2, 5))</td>
<td>0.238</td>
<td>3081</td>
<td>886</td>
</tr>
<tr>
<td>((50, 8) \rightarrow (20, 5))</td>
<td>0.233</td>
<td>717</td>
<td>938</td>
</tr>
<tr>
<td>((4, 8) \rightarrow (5, 5))</td>
<td>1.159</td>
<td>2063</td>
<td>875</td>
</tr>
<tr>
<td>((5, 5) \rightarrow (2, 8))</td>
<td>0.832</td>
<td>2935</td>
<td>879</td>
</tr>
<tr>
<td>((50, 5) \rightarrow (20, 8))</td>
<td>0.831</td>
<td>629</td>
<td>1541</td>
</tr>
<tr>
<td>((4, 10) \rightarrow (5, 8))</td>
<td>0.974</td>
<td>1828</td>
<td>826</td>
</tr>
<tr>
<td>((5, 10) \rightarrow (2, 10))</td>
<td>0.368</td>
<td>1854</td>
<td>808</td>
</tr>
<tr>
<td>((50, 10) \rightarrow (20, 10))</td>
<td>0.567</td>
<td>427</td>
<td>763</td>
</tr>
<tr>
<td>((4, 10) \rightarrow (5, 10))</td>
<td>0.446</td>
<td>1243</td>
<td>797</td>
</tr>
<tr>
<td>((50, 50) \rightarrow (2, 50))</td>
<td>1.632</td>
<td>373</td>
<td>166</td>
</tr>
<tr>
<td>((50, 50) \rightarrow (20, 50))</td>
<td>1.616</td>
<td>86</td>
<td>204</td>
</tr>
<tr>
<td>((4, 50) \rightarrow (5, 50))</td>
<td>1.867</td>
<td>249</td>
<td>216</td>
</tr>
</tbody>
</table>

\(N = 20,000,000\)

Time (ms)
a processor to perform the FALLS intersection algorithm [20, 21] is

\[
T_{\text{index}}(\text{PITFALLS}) = O\left(\frac{\text{lcm}(s \times P, t \times Q)}{\text{min}(s, t \times Q)} \times P + \frac{\text{lcm}(s \times P, t \times Q)}{\text{min}(t, s \times P)} \times Q\right).
\]

(11)

The packing/unpacking cost of the PITFALLS method is

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

(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

\[
T_{\text{index}}(\text{GBCC}) = O\left(\frac{\text{lcm}(s \times P, t \times Q)}{\text{min}(s, t) \times P} + \frac{\text{lcm}(s \times P, t \times Q)}{\text{min}(s, t) \times Q}\right).
\]

(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 \(message_{0}\) 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

<table>
<thead>
<tr>
<th>methods cases</th>
<th>PITFALLS</th>
<th>ScaLAPACK</th>
<th>GBCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>((500, 8)\rightarrow(3, 5))</td>
<td>16.2</td>
<td>6.7</td>
<td>6.1</td>
</tr>
<tr>
<td>((3, 8)\rightarrow(500, 5))</td>
<td>10.2</td>
<td>7.3</td>
<td>5.9</td>
</tr>
<tr>
<td>((500, 8)\rightarrow(1, 5))</td>
<td>15.9</td>
<td>15.7</td>
<td>5.7</td>
</tr>
<tr>
<td>((1, 8)\rightarrow(500, 5))</td>
<td>10.1</td>
<td>16.2</td>
<td>7.5</td>
</tr>
<tr>
<td>((500, 5)\rightarrow(3, 8))</td>
<td>10.2</td>
<td>7.4</td>
<td>5.3</td>
</tr>
<tr>
<td>((3, 5)\rightarrow(500, 8))</td>
<td>16.2</td>
<td>6.9</td>
<td>4.9</td>
</tr>
<tr>
<td>((500, 5)\rightarrow(1, 8))</td>
<td>10.1</td>
<td>16.5</td>
<td>5.6</td>
</tr>
<tr>
<td>((1, 5)\rightarrow(500, 8))</td>
<td>15.9</td>
<td>15.9</td>
<td>5.1</td>
</tr>
<tr>
<td>((500, 10)\rightarrow(3, 10))</td>
<td>12.9</td>
<td>4.6</td>
<td>2.6</td>
</tr>
<tr>
<td>((3, 10)\rightarrow(500, 10))</td>
<td>12.9</td>
<td>4.6</td>
<td>2.6</td>
</tr>
<tr>
<td>((500, 10)\rightarrow(1, 10))</td>
<td>12.6</td>
<td>10.1</td>
<td>3.1</td>
</tr>
<tr>
<td>((1, 10)\rightarrow(500, 10))</td>
<td>12.5</td>
<td>10.2</td>
<td>2.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>methods cases</th>
<th>PITFALLS</th>
<th>ScaLAPACK</th>
<th>GBCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>((500, 8)\rightarrow(3, 5))</td>
<td>16.3</td>
<td>2521</td>
<td>799</td>
</tr>
<tr>
<td>((3, 8)\rightarrow(500, 5))</td>
<td>10.3</td>
<td>2581</td>
<td>798</td>
</tr>
<tr>
<td>((500, 8)\rightarrow(1, 5))</td>
<td>16.0</td>
<td>5084</td>
<td>1203</td>
</tr>
<tr>
<td>((1, 8)\rightarrow(500, 5))</td>
<td>10.1</td>
<td>5308</td>
<td>934</td>
</tr>
<tr>
<td>((500, 5)\rightarrow(3, 8))</td>
<td>10.4</td>
<td>2544</td>
<td>803</td>
</tr>
<tr>
<td>((3, 5)\rightarrow(500, 8))</td>
<td>16.3</td>
<td>2500</td>
<td>857</td>
</tr>
<tr>
<td>((500, 5)\rightarrow(1, 8))</td>
<td>10.1</td>
<td>5611</td>
<td>1050</td>
</tr>
<tr>
<td>((1, 5)\rightarrow(500, 8))</td>
<td>16.0</td>
<td>5381</td>
<td>876</td>
</tr>
<tr>
<td>((500, 10)\rightarrow(3, 10))</td>
<td>12.9</td>
<td>1482</td>
<td>831</td>
</tr>
<tr>
<td>((3, 10)\rightarrow(500, 10))</td>
<td>12.9</td>
<td>1482</td>
<td>830</td>
</tr>
<tr>
<td>((500, 10)\rightarrow(1, 10))</td>
<td>12.6</td>
<td>3083</td>
<td>931</td>
</tr>
<tr>
<td>((1, 10)\rightarrow(500, 10))</td>
<td>12.6</td>
<td>3108</td>
<td>988</td>
</tr>
</tbody>
</table>
unpack those array elements to their corresponding local array positions. For example, given a 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 data-movement operations that must be performed by destination processor $Q_0$ to unpack those 12 elements from messages to their corresponding local array positions in the receive phase.

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

$$T_{(un)pack}(GBCC) = O\left(\frac{N}{P} + \frac{N}{Q} \right)$$  \hspace{1cm} \text{if } s > t \times Q,$$  \hspace{1cm} \text{or } O\left(\frac{N}{s \times P} + \frac{N}{Q} \right) \hspace{1cm} \text{if } t > s \times P,$$  \hspace{1cm} \text{or } O\left(\frac{N}{P} + \frac{N}{s \times Q} \right) \hspace{1cm} \text{otherwise.}$$

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.
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 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 \( T_{\text{pack}}^{\text{GBCC}} < T_{\text{pack}}^{\text{ScaLAPACK}} < T_{\text{pack}}^{\text{PITFALLS}} \). 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 \( T_{\text{pack}}^{\text{PITFALLS}} < T_{\text{pack}}^{\text{GBCC}} < T_{\text{pack}}^{\text{ScaLAPACK}} \). 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 \( (5.8k) \to (4.5k) \) 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 \( T_{\text{index}}^{\text{PITFALLS}} \) and \( T_{\text{index}}^{\text{ScaLAPACK}} \) shown in (11) is approximately

\[
\frac{t \times Q^2 + s \times P^2}{\gcd(s \times P, t \times Q)}
\]
while $T_{\text{index}}(GBCC)$ shown in (13) is approximately

$$T_{\text{index}}(GBCC) = \frac{t \times Q + s \times P}{\gcd(s \times P, t \times Q)}.$$ 

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

$$k(t \times Q^2 + s \times P^2) / \gcd(s \times P^2, t \times Q^2),$$

which depends on the value of $k$. $T_{\text{index}}(GBCC)$ 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 $960 \times 960$ 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**


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-Yat-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, 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.