Efficient Methods for Multi-Dimensional Array Redistribution

Yeh-Ching Chung and Ching-Hsien Hsu

Department of Information Engineering
Feng Chia University, Taichung, Taiwan 407, ROC
Tel : 886-4-4517250 x3746
Fax : 886-4-4516101
Email: ychung, chhsu@pine.iecs.fcu.edu.tw

Abstract

In this paper, we present efficient methods for multi-dimensional array redistribution. Based on the previous work, the basic-cycle calculation technique, we present a basic-block calculation (BBC) and a complete-dimension calculation (CDC) techniques. We have developed a theoretical model to analyze the computation costs of these two techniques. The theoretical model shows that the BBC method has smaller indexing costs and performs well for the redistribution with small array size. The CDC method has smaller packing/unpacking costs and performs well when the array size is large. We also have implemented these two techniques along with the PITFALLS method and the Prylli's method on an IBM SP2 parallel machine. The experimental results show that the BBC method has the smallest execution time of these four algorithms when the array size is small. The CDC method has the smallest execution time of these four algorithms when the array size is large. Furthermore, the BBC method outperforms the PITFALLS method and the Prylli's method for all test samples.

Keywords: array redistribution, the basic-block calculation technique, the complete-dimension calculation technique.

1. Introduction

In some algorithms, such as multi-dimensional fast Fourier transform, the Alternative Direction Implicit (ADI) method for solving two-dimensional diffusion equations, and linear algebra solvers, an array distribution that is well-suited for one phase may not be good for a subsequent phase in terms of performance. Array redistribution is required for those algorithms during run-time to enhance algorithm performance. Therefore, many data parallel programming languages support run-time primitives for array redistribution. Since array redistribution is performed at run-time, there is a performance trade-off between the efficiency of new data decomposition 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 those languages. Many methods for performing array redistribution were proposed in the literature [1-2, 5-7, 9-14, 16-19]. Due to the page limitation, we will not describe these methods here. The details of these methods can be found in [2].

In this paper, based on the *basic-cycle calculation technique* [2], we present a *basic-block calculation* (*BBC*) and a *complete-dimension calculation* (*CDC*) techniques for multi-dimensional array redistribution. The main idea of the basic-block calculation technique is first to use the basic-cycle calculation technique to determine source/destination processors of some specific array elements in a basic-block. From the source/destination processor/data sets of a basic-block, we can efficiently perform a redistribution. The complete-dimension calculation technique also uses the basic-cycle calculation technique to generate the communication sets of a redistribution. However, it generates the communication sets for array elements in the first row of each dimension of a local array. This will result in a high indexing overheads. But the packing/unpacking overheads can be greatly reduced. These two techniques can be easily implemented in a parallelizing compiler, run-time systems, or parallel programs. In this paper, we also developed a theoretical model to analyze the tradeoff between these two techniques.

2. Preliminaries

To simplify the presentation, we use $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ to represent the $(\texttt{CYCLIC}(s_0), \texttt{CYCLIC}(s_1), \ldots, \texttt{CYCLIC}(s_{n-1}))$ to $(\texttt{CYCLIC}(t_0), \texttt{CYCLIC}(t_1), \ldots, \texttt{CYCLIC}(t_{n-1}))$ redistribution for the rest of the paper.

Definition 1: An n-dimensional array is defined as the set of array elements $A^{(n)} = A[1:n_0, 1:n_1, ..., 1:n_{n-1}] = \{ a_{d_0,d_1,\ldots,d_{n-1}} \mid 0 \le d_\ell \le n_\ell - 1, 0 \le \ell \le n-1 \}$. The size of array $A^{(n)}$, denoted by $|A^{(n)}|$, is equal to $n_0 \times n_1 \times \ldots \times n_{n-1}$. In this paper, we assume that array elements are stored in a memory by a row-major manner.

Definition 2: An n-dimensional processor grid is defined as the set of processors $M^{(n)} =$

$M[m_0, m_1, ..., m_{n-1}]$ = { $\tilde{p}_{d_0,d_1,...,d_{n-1}} \mid 0 \le d_\ell \le m_\ell - 1$, $0 \le \ell \le n-1$ }. The number of processors of $M^{(n)}$, denoted by $|M^{(n)}|$, is equal to $m_0 \times m_1 \times ... \times m_{n-1}$.

Definition 3: Given an n-dimensional processor grid $M^{(n)}$, the rank of processor $\tilde{p}_{d_0,d_1,...,d_{n-1}}$ is equal to $i = \sum_{k=0}^{n-1}(d_k \times \prod_{\ell=k+1}^{n-1} m_\ell)$, where $0 \le d_\ell \le m_\ell - 1$, $0 \le \ell \le n-1$. To simplify the presentation, we also use processor P_i to denote $\tilde{p}_{d_0,d_1,...,d_{n-1}}$ in this paper, where $0 \le i \le |M^{(n)}| - 1$.

Definition 4: Given a BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution, BC(s_0, s_1, ..., s_{n-1}), BC(t_0, t_1, ..., t_{n-1}), s_ℓ and t_ℓ are called the *source distribution*, the *destination distribution*, the *source distribution factors*, and the *destination distribution factors* of the redistribution, respectively, where $0 \le \ell \le n$-1.

Definition 5: Given a BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution on $A^{(n)}$ over $M^{(n)}$, the *source* (*destination*) *local array* of processor P_i, denoted by $SLA_i^{(n)}$ ($DLA_j^{(n)}$)[1:$\frac{n_0}{m_0}$, 1:$\frac{n_1}{m_1}$, ..., 1:$\frac{n_{n-1}}{m_{n-1}}$], is defined as the set of array elements that are distributed to processor P_i (P_j) in the source (destination) distribution, i.e., $|SLA_i^{(n)}| = \prod_{b=0}^{n-1}\lceil \frac{n_b}{m_b} \rceil$, where $0 \le i \le |M^{(n)}| - 1$.

Definition 6: We define $SLA_{i,\ell}^{(n)}$ as the set of array elements in the first row of the ℓ th dimension of $SLA_i^{(n)}$, i.e., $SLA_{i,\ell}^{(n)} = SLA_i^{(n)}$ [1,...,1, 1:$\frac{n_\ell}{m_\ell}$, 1,...,1], where $0 \le i \le |M^{(n)}| - 1$ and $0 \le \ell \le n-1$. The number of array elements in $SLA_{i,\ell}^{(n)}$ is equal to $\frac{n_\ell}{m_\ell}$. $SLA_{i,\ell}^{(n)}[r]$ is defined as the rth array element of $SLA_{i,\ell}^{(n)}$.

Definition 7: Given a BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution on $A^{(n)}$ over $M^{(n)}$, a *basic-cycle* of the ℓ th dimension of $SLA_i^{(n)}$ (or $DLA_j^{(n)}$), denoted by BC_ℓ, is defined as $BC_\ell = lcm(s_\ell, t_\ell)/gcd(s_\ell, t_\ell)$, where $0 \le \ell \le n-1$.

Definition 8: Given a BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution on $A^{(n)}$ over $M^{(n)}$, a *basic-block* of $SLA_i^{(n)}$ (or $DLA_j^{(n)}$) is defined as the multiplication of the basic-cycles in each dimension. The size of a basic-block is equal to $BC_0 \times BC_1 \times \ldots \times BC_{n-1}$.

3. Multi-dimensional Array Redistribution

To perform a BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution, in general, a processor needs to compute the communication sets. Based on the characteristics of a redistribution, we have the following lemmas. Due to the page limitation, we will not present the proofs of lemmas in this paper.

Lemma 1: Given a BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution on $A^{(n)}$ over $M^{(n)}$, for a source (destination) processor P_i, if the rank of the destination (source) processor of $SLA_{i,k}^{(n)}[r_k]$ ($DLA_{i,k}^{(n)}[r_k]$) is $\tilde{p}_{0,...0,j_k,0,...,0}$, where $0 \le i \le |M^{(n)}| - 1$, k = 0 to $n-1$, $0 \le j_k \le m_k - 1$, and $1 \le r_k \le \lceil \frac{n_k}{m_k} \rceil$, then the destination (source) processor of $SLA_i^{(n)}[r_0, r_1, ..., r_{n-1}]$ ($DLA_i^{(n)}[r_0, r_1, ..., r_{n-1}]$) is P_j, where $j = \sum_{k=0}^{n-1}(j_k \times \prod_{\ell=k+1}^{n-1} m_\ell)$. ■

According to Lemma 1, the destination (source) processor of $SLA_i^{(n)}[r_0, r_1, ..., r_{n-1}]$ ($DLA_j^{(n)}[r_0, r_1, ..., r_{n-1}]$) can be determined by the rank of destination (source) processors of $SLA_{i,0}^{(n)}[r_0]$, $SLA_{i,1}^{(n)}[r_1]$,..., and $SLA_{i,n-1}^{(n)}[r_{n-1}]$ ($DLA_{j,0}^{(n)}[r_0]$, $DLA_{j,1}^{(n)}[r_1]$,..., and $DLA_{j,n-1}^{(n)}[r_{n-1}]$). Therefore, how to efficiently determine the communication sets of these array elements is important. The basic-block calculation technique and the complete-dimension calculation technique are based on the basic-cycle calculation technique proposed in [2]. The main idea of the basic-cycle calculation technique is based on the following lemma.

Lemma 2: Given a BC(s)→BC(t) and a BC($s/gcd(s,t)$)→BC($t/gcd(s,t)$) redistribution on a one-dimensional array A[1:N] over M processors, for a source (destination) processor P_i (P_j), if the destination (source) processor of $SLA_i[k]$ ($DLA_j[k]$) in BC($s/gcd(s, t)$)→BC($t/gcd(s, t)$) redistribution is P_j (P_i), then the destination (source) processors of $SLA_i[(k-1)\times gcd(s,t)+1: k\times gcd(s,t)]$ ($DLA_j[(k-1)\times gcd(s,t)+1: k\times gcd(s,t)]$) in BC($s$) → BC($t$) redistribution will also be P_j (P_i), where $1 \le k \le \lceil N/(M\times gcd(s,t)) \rceil$. ■

Given a BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution, according to Lemma 2, we know that the communication sets of BC($s_0/gcd(s_0,t_0)$, $s_1/gcd(s_1,t_1)$, ..., $s_{n-1}/gcd(s_{n-1},t_{n-1})$) → BC($t_0/gcd(s_0,t_0)$, $t_1/gcd(s_1,t_1)$, ..., $t_{n-1}/gcd(s_{n-1},t_{n-1})$) redistribution can be used to generate the communication sets of BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution. Therefore, in the following discussion, for a BC(s_0, s_1, ..., s_{n-1})→BC(t_0, t_1, ..., t_{n-1}) redistribution, we assume that $gcd(s_i, t_i)$ is equal to 1, where $1 \le i \le n-1$. If $gcd(s_i, t_i)$ is not equal to 1, we use $s_i/gcd(s_i,t_i)$ and $t_i/gcd(s_i,t_i)$ as the source and destination distribution factors of the redistribution, respectively.

3.1 The Basic-Block Calculation Technique

Given a BC(s_0, s_1)→BC(t_0, t_1) redistribution on a two-dimensional array A[1:n_0,1:n_1] over M [m_0, m_1], to perform the redistribution, we have to first construct the communication sets. According to Lemma 1, a source processor P_i only needs to determine the destination processor sets for $SLA_{i,0}^{(2)}$[1:BC_0] and $SLA_{i,1}^{(2)}$[1:BC_1].

Then it can generate the destination processor sets for $SLA_i^{(2)}[1{:}BC_0,\ 1{:}BC_1]$. For example, if the destination processors of $SLA_{i,0}^{(2)}[r_0]$ and $SLA_{i,1}^{(2)}[r_1]$ are $\tilde{p}_{j_0,0}$ and $\tilde{p}_{0,j_1}$, respectively, the destination processor $P_j = \tilde{p}_{j_0,j_1}$ of $SLA_i^{(2)}[r_0][r_1]$ can be determined by the following equation,

$$Rank(P_j) = j_0 \times m_1 + j_1, \quad (1)$$

For a source processor P_i, if $P_i = \tilde{p}_{i_0,i_1}$, according to Definition 3, we have $i_0 = \lfloor i/m_1 \rfloor$ and $i_1 = mod(i, m_1)$, where $0 \le i_0 \le m_0 - 1$ and $0 \le i_1 \le m_1 - 1$. Since the values of i_0 and i_1 are known, the destination processors of $SLA_{i,0}^{(2)}[1{:}BC_0]$ and $SLA_{i,1}^{(2)}[1{:}BC_1]$ can be determined by the following equation,

$$DP_{(\ell)} = F(x) \times \begin{bmatrix} 1 \\ 2 \\ \vdots \\ BC_\ell \end{bmatrix}_{BC_\ell \times 1} \quad (2)$$

Where $\ell = 0$ and 1. The function $F(x)$ is defined as

$$F(x) = \left\lfloor \frac{mod(\beta \times s_\ell, m_\ell \times t_\ell)}{t_\ell} \right\rfloor, \quad (3)$$

where $x = 1$ to BC_ℓ and β is defined as

$$\beta = i_\ell + m_\ell \times \left\lfloor \frac{x}{s_\ell} \right\rfloor, \quad (4)$$

For a two-dimensional array redistribution, from Equation 2, we can obtain $DP_{(0)}$ and $DP_{(1)}$ that represent the destination processors of $SLA_{i,0}^{(2)}[1{:}BC_0]$ and $SLA_{i,1}^{(2)}[1{:}BC_1]$, respectively. According to $DP_{(0)}$, $DP_{(1)}$, and Equation 1, a source processor P_i can determine the destination processor of array elements in the first basic-block of $SLA_i^{(2)}$, i.e., $SLA_i^{(2)}[1{:}\ BC_0, 1{:}BC_1]$.

For a multi-dimensional array redistribution, each basic-block of a local array has the same communication patterns. The following lemma states this characteristic.

Lemma 3: Given a $\mathsf{BC}(s_0, s_1) \rightarrow \mathsf{BC}(t_0, t_1)$ redistribution on a two-dimensional array $A[1{:}n_0, 1{:}n_1]$ over M $[m_0, m_1]$, $SLA_i^{(2)}[x, y]$, $SLA_i^{(2)}[x+k_0 \times BC_0, y]$, $SLA_i^{(2)}[x, y+k_1 \times BC_1]$, $SLA_i^{(2)}[x+k_0 \times BC_0, y+k_1 \times BC_1]$ have the same destination processor, where $0 \le i \le m_0 \times m_1 - 1$, $1 \le x \le lcm(s_0, t_0)$, $1 \le y \le lcm(s_1, t_1)$, $1 \le k_0 \le \lfloor n_0 / (lcm(s_0, t_0) \times m_0) \rfloor$ and $1 \le k_1 \le \lfloor n_1 / (lcm(s_1, t_1) \times m_1) \rfloor$. ■

Since each basic-block has the same communication patterns, we can pack local array elements to messages according to the destination processors of array elements in $SLA_i^{(2)}[1{:}\ BC_0, 1{:}BC_1]$. However, if the value of $BC_0 \times BC_1$ is large, it may take a lot of time to compute the destination processors of array elements in a basic-block by using Equation 1. In the basic-block calculation technique, instead of using the destination processors of array elements in the first basic-block, it uses a table lookup method to pack array elements. Given a $\mathsf{BC}(s_0, s_1) \rightarrow \mathsf{BC}(t_0, t_1)$ redistribution on a two-dimensional array $A[1{:}n_0, 1{:}n_1]$ over M $[m_0, m_1]$, since the destination processors of $SLA_i^{(2)}[1{:}BC_0, 1{:}BC_1]$ can be determined by $DP_{(0)}$ and $DP_{(1)}$, if we gather the indices of array elements in $SLA_{i,\ell}^{(2)}[1{:}\ BC_\ell]$ that have the same destination processor to tables, *Send_Tables*, we can also determine the destination processors of $SLA_i^{(2)}[1{:}\ BC_0, 1{:}BC_1]$ from *Send_Tables*.

In the receive phase, according to Lemma 2, a destination processor P_j only needs to determine the source processor sets for $DLA_{j,0}^{(2)}[1{:}BC_0]$ and $DLA_{j,0}^{(2)}[1{:}BC_1]$. Then it can generate the source processor sets for $DLA_j^{(2)}[1{:}\ BC_0, 1{:}BC_1]$. For example, if the source processors of $DLA_{j,0}^{(2)}[r_0]$ and $DLA_{j,0}^{(2)}[r_1]$ are $\tilde{p}_{i_0,0}$ and $\tilde{p}_{0,i_1}$, respectively, the source processor P_i of $DLA_j^{(2)}[r_0][r_1]$ can be determined by the following equation,

$$Rank(P_i) = i_0 \times m_1 + i_1, \quad (5)$$

For a destination processor P_j, if $P_j = \tilde{p}_{j_0,j_1}$, according to Definition 3, we have $j_0 = \lfloor j/m_1 \rfloor$ and $j_1 = j \bmod m_1$, where $0 \le j_0 \le m_0 - 1$ and $0 \le j_1 \le m_1 - 1$. Since the value of j_0 and j_1 are known, the source processors of $DLA_{j,0}^{(2)}[1{:}BC_0]$ and $DLA_{j,1}^{(2)}[1{:}BC_1]$ can be determined by the following equation,

$$SP_{(\ell)} = G(x) \times \begin{bmatrix} 1 \\ 2 \\ \vdots \\ BC_\ell \end{bmatrix}_{BC_\ell \times 1} \quad (6)$$

Where $\ell = 0, 1$. The function $G(x)$ is defined as

$$G(x) = \left\lfloor \frac{mod(\gamma \times t_\ell, m_\ell \times s_\ell)}{s_\ell} \right\rfloor \quad (7)$$

where $x = 1$ to BC_ℓ and γ is defined as

$$\gamma = j_\ell + m_\ell \times \left\lfloor \frac{x}{t_\ell} \right\rfloor, \quad (8)$$

For a two-dimensional array redistribution, from Equation 6, we can obtain $SP_{(0)}$ and $SP_{(1)}$ that represent

the source processors of $DLA^{(2)}_{j,0}[1{:}BC_0]$ and $DLA^{(2)}_{j,1}[1{:}BC_1]$, respectively. According to $SP_{(0)}$ and $SP_{(1)}$, we can also construct the *Receive_Tables* for the destination processor P_j as we construct the *Send_Tables* in the send phase. Based on the *Receive_Tables*, we can unpack array elements from the received messages to their appropriate destination local array positions.

The algorithm of the basic-block calculation technique is given as follows.

Algorithm Basic_Block_Calculation(s_0, ..., s_{n-1}, t_0, ..., t_{n-1}, n_0, ..., n_{n-1}, m_0, ..., m_{n-1})

1. *Construct Send_Tables*;
2. **For** (*j = myrank*, *z*=0 ; *z* < |*M*| ; *j*++, *z*++)
3. *j = mod*(*j*, |*M*|);
4. *Pack the message for destination processor P_j to out_buffer according to the STs*;
5. **If** (*out_buffer* != *NULL*)
6. **Send** *out_buffer* to *destination processor P_j*;
7. *Construct Receive_Tables*;
8. *x = the number of messages to be received*;
9. **For** (*z*=0 ; *z* < *x* ; *z*++)
10. **Receive** data sets *in_buffer* from ***any*** source processor;
11. *Unpack the received messages according to the RTs*;

end_of_ Basic_Block_Calculation

3.2 The Complete-Dimension Calculation Technique

In section 3.1, we stated that each basic-block has the same communication patterns. Therefore, a processor only needs to construct the *Send_Tables* and the *Receive_Tables* for the first basic-cycle in each dimension of its local array. Then it can perform a multi-dimensional array redistribution. In this section, we will present a *complete-dimension calculation* (*CDC*) technique. In the complete-dimension calculation technique, a processor constructs the *Send_Tables* and the *Receive_Tables* not only for array elements of the first basic-cycle in each dimension of its local array, but also for array elements in the first row of each dimension of its local array, i.e., $SLA^{(n)}_{i,\ell}[1{:}n_\ell]$, where $\ell = 0$ n–1. In the following, we will describe the complete-dimension calculation technique in details.

Assume that a $\mathsf{BC}(s_0, s_1, ..., s_{n-1}) \rightarrow \mathsf{BC}(t_0, t_1, ..., t_{n-1})$ redistribution on an n-dimensional array $A^{(n)} = A[1{:}n_0, 1{:}n_1, ..., 1{:}n_{n-1}]$ over $M^{(n)} = M[m_0, m_1, ..., m_{n-1}]$ is given. For the complete-dimension calculation technique, in the send phase, a source processor P_i computes the destination processors for array elements in $SLA^{(n)}_{i,0}[1{:}L_0]$, $SLA^{(n)}_{i,1}[1{:}L_1]$, …, $SLA^{(n)}_{i,n-1}[1{:}L_{n-1}]$, where L_k is the local array size in each dimension, i.e., $L_k = \frac{n_k}{m_k}$, k = 0 to n–1. The destination processors of $SLA^{(n)}_{i,0}[1{:}L_0]$, $SLA^{(n)}_{i,1}[1{:}L_1]$, …, $SLA^{(n)}_{i,n-1}[1{:}L_{n-1}]$ can be determined by the following Equation:

$$DP_{(\ell)} = F(x) \times \begin{bmatrix} 1 \\ 2 \\ \vdots \\ L_\ell \end{bmatrix}_{L_\ell \times 1} \qquad (9)$$

Where ℓ =1 to n–1. The function $F(x)$ is defined in Equation 3.

For a $\mathsf{BC}(s_0, s_1, ..., s_{n-1}) \rightarrow \mathsf{BC}(t_0, t_1, ..., t_{n-1})$ redistribution, from Equation 9, we can obtain $DP_{(0)}$, $DP_{(1)}$, …, $DP_{(n-1)}$ that represent destination processors of $SLA^{(n)}_{i,0}[1{:}L_0]$, $SLA^{(n)}_{i,1}[1{:}L_1]$, …, $SLA^{(n)}_{i,n-1}[1{:}L_{n-1}]$, respectively. Since the destination processors of $SLA^{(n)}_{i,0}[1{:}L_0]$, $SLA^{(n)}_{i,1}[1{:}L_1]$, …, $SLA^{(n)}_{i,n-1}[1{:}L_{n-1}]$ are known, we can construct the *Send_Tables* for $SLA^{(n)}_{i,0}[1{:}L_0]$, $SLA^{(n)}_{i,1}[1{:}L_1]$, …, $SLA^{(n)}_{i,n-1}[1{:}L_{n-1}]$.

In the receive phase, a destination processor P_j computes the source processors for array elements in $DLA^{(n)}_{j,0}[1{:}L_0]$, $DLA^{(n)}_{j,1}[1{:}L_1]$, …, $DLA^{(n)}_{j,n-1}[1{:}L_{n-1}]$, where L_k is the local array size in each dimension, i.e., $L_k = \frac{n_k}{m_k}$, k = 0 to n–1. The source processors of $DLA^{(n)}_{j,0}[1{:}L_0]$, $DLA^{(n)}_{j,1}[1{:}L_1]$, …, $DLA^{(n)}_{j,n-1}[1{:}L_{n-1}]$ can be determined by the following Equation,

$$SP_{(\ell)} = G(x) \times \begin{bmatrix} 1 \\ 2 \\ \vdots \\ L_\ell \end{bmatrix}_{L_\ell \times 1} \qquad (10)$$

Where ℓ =1 to n–1. The function $G(x)$ is defined in Equation 7.

For a $\mathsf{BC}(s_0, s_1, ..., s_{n-1}) \rightarrow \mathsf{BC}(t_0, t_1, ..., t_{n-1})$ redistribution, from Equation 10, we can obtain $SP_{(0)}$, $SP_{(1)}$, …, $SP_{(n-1)}$ that represent the source processors of $SLA^{(n)}_{i,0}[1{:}L_0]$, $SLA^{(n)}_{i,1}[1{:}L_1]$, …, $SLA^{(n)}_{i,n-1}[1{:}L_{n-1}]$, respectively. Since the source processors of $DLA^{(n)}_{j,0}[1{:}L_0]$, $DLA^{(n)}_{j,1}[1{:}L_1]$, …, $DLA^{(n)}_{j,n-1}[1{:}L_{n-1}]$ are known, we can construct the *Receive_Tables* for $DLA^{(n)}_{j,0}[1{:}L_0]$, $DLA^{(n)}_{j,1}[1{:}L_1]$, …, $DLA^{(n)}_{j,n-1}[1{:}L_{n-1}]$. Based on the *Receive_Tables*, we can unpack array elements in the received messages to their appropriate local array positions.

3.3 Theoretical Performance Comparisons of BBC and CDC

The complete-dimension calculation technique has higher indexing cost than that of the basic-block calculation technique because it constructs larger *Send_Tables* and *Receive_Tables*. However, the complete-dimension calculation technique provides more packing/unpacking information than the basic-block calculation technique. It may have lower packing/unpacking cost than that of the basic-block calculation technique. In this section, we derive a theoretical model to analyze the tradeoff between these two methods.

Given a $\mathsf{BC}(s_0, s_1, ..., s_{n-1}) \rightarrow \mathsf{BC}(t_0, t_1, ..., t_{n-1})$ redistribution on an n-dimensional array $A^{(n)}$ over $M^{(n)}$, the computation cost for an algorithm to perform the redistribution, in general, can be modeled as follows:

$$T_{comp} = T_{indexing} + T_{(un)packing} \qquad (11)$$

We first construct a model for two-dimensional array redistribution. Then, extend the model to multi-dimensional array redistribution.

Given a $\mathsf{BC}(s_0, s_1) \rightarrow \mathsf{BC}(t_0, t_1)$ redistribution on a two-dimensional array $A[1:n_0, 1:n_1]$ over $M[m_0, m_1]$, the indexing time of the basic-block calculation technique and the complete-dimension calculation technique can be modeled as follows,

$$T_{indexing}(BBC) = O(BC_0) + O(BC_1) \qquad (12)$$

$$T_{indexing}(CDC) = O(L_0) + O(L_1) \qquad (13)$$

where BC_k is the size of basic-cycle in each dimension; L_k is the local array size in each dimension, $L_k = \frac{n_k}{m_k}$, k = 0,1.

In the basic-block calculation technique, the *Send_Tables* only store the indices of local array elements in the first basic-cycle. A processor needs to calculate the stride distance when it packs local array elements that are in the rest of basic-cycles into messages. The time for a processor to pack array elements to messages in each row is $O(\frac{L_1}{BC_1})$, where $\frac{L_1}{BC_1}$ is the number of basic-cycles in dimension 1. There are L_0 rows in a local array. The time for a processor to pack array elements in dimension 1 to messages is $O(\frac{L_0 \times L_1}{BC_1})$. Since a processor packs local array elements to messages in a row-major manner, the time for a processor to pack array elements in dimension zero to messages is $O(\frac{L_0}{BC_0})$. Therefore, the time for a processor to pack array elements to messages can be modeled as follows,

$$T_{(un)packing}(BBC) = O(\frac{L_0 \times L_1}{BC_1}) + O(\frac{L_0}{BC_0}) \qquad (14)$$

In the complete-dimension calculation technique, the *Send_Tables* store the indices of local array elements in $SLA^{(n)}_{i,0}[1:L_0]$ and $SLA^{(n)}_{i,1}[1:L_1]$. According to the *Send_Tables*, a processor can pack local array elements into messages directly. It does not need to calculate the stride distance when it packs array elements that are not in the first basic-cycle.

According to Equations 11 to 14, the computation time of the complete-dimension calculation is less than that of the basic-block calculation technique if and only if the following equation is true.

$$T_{comp}(CDC) < T_{comp}(BBC) \Leftrightarrow$$
$$O(L_0 + L_1) < O(BC_0 + BC_1 + \frac{L_0 \times L_1}{BC_1} + \frac{L_0}{BC_0}) \qquad (15)$$

By truncating BC_0, BC_1 and $\frac{L_0}{BC_0}$ from $T_{comp}(BBC)$, we obtain the following equation:

$$T_{comp}(CDC) < T_{comp}(BBC) \Leftrightarrow$$
$$O(L_0 + L_1) < O(\frac{L_0 \times L_1}{BC_1}) \qquad (16)$$

Given a $\mathsf{BC}(s_0, s_1, ..., s_{n-1}) \rightarrow \mathsf{BC}(t_0, t_1, ..., t_{n-1})$ redistribution on an n-dimensional array $A^{(n)} = A[1:n_0, 1:n_1, ..., 1:n_{n-1}]$ over $M^{(n)} = M[m_0, m_1, ..., m_{n-1}]$, according to Equation 16, the computation time of the complete-dimension calculation is less than that of the basic-block calculation technique if and only if the following equation is true.

$$T_{comp}(CDC) < T_{comp}(BBC) \Leftrightarrow O(L_0 + L_1 + \cdots + L_{n-1}) <$$
$$O(\frac{L_{n-1}}{BC_{n-1}} \times L_{n-2} \times \cdots \times L_0) \qquad (17)$$

4. Experimental Results

To evaluate the performance of the basic-block calculation and the complete-dimension calculation techniques, we have implemented these two techniques along with the *PITFALLS* method [14] and the *Prylli*'s method [13] on an IBM SP2 parallel machine. All algorithms were written in the single program multiple data (SPMD) programming paradigm with C+MPI codes. To get the experimental results, we used different redistribution as test samples. For these redistribution samples, we roughly classify them into the following three types:

- *Dimension Shift redistribution*:
  Ex: $\mathsf{BC}(x, y)$ to $\mathsf{BC}(y, x)$ of two-dimensional arrays, and $\mathsf{BC}(x, y, z)$ to $\mathsf{BC}(y, z, x)$ of three-dimensional arrays, where x, y and z are positive integers.
- *Refinement redistribution*:

Ex: BC(x, y) to BC($\frac{x}{p}$, $\frac{y}{q}$) of two-dimensional arrays, and BC(x, y, z) to BC($\frac{x}{p}$, $\frac{y}{q}$, $\frac{z}{r}$) of three-dimensional arrays, where p, q and r are factors of x, y and z, respectively.

- *Block Cyclic redistribution*:
  Ex: (BLOCK, BLOCK) to (CYCLIC, CYCLIC) of two-dimensional arrays, and (BLOCK, BLOCK, BLOCK) to (CYCLIC, CYCLIC, CYCLIC) of three-dimensional arrays.

Table 1 shows the execution time of these four algorithms to perform a BC(5, 8) to BC(8, 5) (i.e., dimension shift) redistribution with fixed array size on different numbers of processors. From Table 1, we can see that the indexing time of the basic-block calculation technique is independent of the number of processors. The indexing time of the *Prylli's* method and the *PITFALLS* method depends on the number of processors. When the number of processors increases, the indexing time of the *Prylli's* method and the *PITFALLS* method increases as well. The indexing time of the complete-dimension calculation technique decreases when the number of processors increases. The reason is that when the array size is fixed and the number of processors is increased, the number of array elements that will be processed by the complete-dimension calculation technique decreases.

For the same test sample, the complete-dimension calculation technique has smaller packing/unpacking time than that of other methods. The reason is that the complete-dimension calculation technique provides more packing/unpacking information than other methods. This packing/unpacking information allows the complete-dimension calculation technique to pack/unpack elements directly. Other methods need to spend time to calculation stride distance of array elements when packing/unpacking array elements. The packing/unpacking time of the basic-block calculation technique, the *PITFALLS* method and the *Prylli's* method are similar.

All of these four methods use asynchronous communication schemes. Therefore, the computation and the communication overheads can be overlapped. However, the basic-block calculation and the complete-dimension calculation techniques unpack any received messages in the receive phase while the *PITFALLS* and the *Prylli's* methods unpack messages in a specific order. Therefore, in general, we can expect that the communication time of the basic-block calculation and the complete-dimension calculation techniques is less than or equal to that of the *PITFALLS* and the *Prylli's* methods.

From Table 1, we can see that the complete-dimension calculation technique has the smallest execution time when the number of processors is less than or equal to 24 (8×3). The basic-block calculation technique has the smallest execution time when the number of processors is greater than or equal to 32 (8×4). These phenomena match the theoretical analysis given in Equation 17. We also observe that the execution time of the basic-block calculation technique is smaller than that of the *PITFALLS* and the *Prylli's* methods for all test samples.

Table 2 shows the performance of these four algorithms to execute a BC(10, 20) to BC(5, 10) (i.e., Refinement) redistribution with fixed array size on different numbers of processors. From Table 2, we have similar observations as those described for Table 1.

Table 3 shows the execution time of these four algorithms to perform a (BLOCK, BLOCK) to (CYCLIC, CYCLIC) redistribution. In this case, the *Send_Tables* and *Receive_Tables* constructed by the basic-block calculation technique and the complete-dimension calculation technique are the same. Therefore, they have almost the same execution time. The execution time of both methods for this redistribution is less than that of the *PITFALLS* method and the *Prylli's* method.

Table 4 shows the performance of these four algorithms to execute these three redistribution with various array size on a processor grid M[8,7]. From Table 4, for the BC(5, 8) to BC(8, 5) and BC(10, 20) to BC(5, 10) redistribution, we can see that the execution time of the complete-dimension calculation technique is less than that of the basic-block calculation technique for all test samples. The reason can be explained by Equation 17. Moreover, the execution time of both methods is less than that of the *PITFALLS* method and the *Prylli's* method for all test samples.

For the (BLOCK, BLOCK) to (CYCLIC, CYCLIC) redistribution, the execution time of these four algorithms has the order $T_{exec}(CDC) \approx T_{exec}(BBC) << T_{exec}(Prylli's) < T_{exec}(PITFALLS)$. In this case, the *PITFALLS* method and the *Prylli's* method have very large execution time compared to that of the *BBC* method and the *CDC* method. The reason is that each processor needs to find out all intersections between source and destination distribution with all other processors in the *PITFALLS* and the *Prylli's* methods. The computation time of the *PITFALLS* and the *Prylli's* methods depends on the number of intersections. In this case, there are $N_0/m_0 + N_1/m_1$ intersections between each source and destination processor. Therefore, a processor needs to compute $\lfloor N_0/m_0 \rfloor \times m_0 + \lfloor N_1/m_1 \rfloor \times m_1$ intersections which demands a lot of computation time when N_0 and N_1 are large.

Table 5 shows the performance of these four algorithms to execute different redistribution on three-dimensional arrays. Each redistribution with various array size on a processor grid M[2,4,7] with 56 processors were tested. From Table 5, we have similar observations as those described for Table 4.

5. Conclusions

In this paper, we have presented efficient algorithms for performing multi-dimensional array redistribution. Based on the basic-cycle calculation technique, we presented a basic-block calculation technique and a complete-dimension calculation technique. In these two methods, the *Send_Tables* and the *Receive_Tables* are used to store the packing/unpacking information of a redistribution. From the information of *Send_Tables* and *Receive_Tables*, we can efficiently perform the redistribution of multidimensional arrays. The theoretical model shows that the *BBC* method has smaller indexing costs and performs well for the redistribution with small array size. The *CDC* method has smaller packing/unpacking costs and performs well when the array size is large. The experimental results also show that our algorithms can provide better performance than the *PITFALLS* method and the *Prylli's* method.

References

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

[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] Frederic Desprez, Jack Dongarra, and Antoine Petitet," Scheduling Block-Cyclic Array Redistribution," *IEEE Trans. on PDS*, vol. 9, no. 2, Feb. 1998.

[4] 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.

[5] S. Hiranandani, K. Kennedy, J. Mellor-Crammey, and A. Sethi," Compilation technique for block-cyclic distribution," In *Proc. ACM Intl. Conf. on Supercomputing*, pp. 392-403, July 1994.

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

[7] S. D. Kaushik, C. H. Huang, J. Ramanujam, and P. Sadayappan, "Multiphase array redistribution: Modeling and evaluation," In *Proc. of IPPS*, pp. 441-445, 1995.

[8] 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.

[9] K. Kennedy, N. Nedeljkovic, and A. Sethi, "Efficient address generation for block-cyclic distribution," In *Proc. of Intl Conf. on Supercomputing*, pp. 180-184, July 1995.

[10] P-Z. Lee and W. Y. Chen, "Compiler techniques 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.

[11] Young Won Lim, Prashanth B. Bhat, and Viktor, K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," *Proc. of the Eighth IEEE SPDP*, pp. 74-83, 1996.

[12] 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.

[13] L. Prylli and B. Touranchean, "Fast runtime block cyclic data redistribution on multiprocessors," *JPDC*, Vol. 45, pp. 63-72, Aug. 1997.

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

[15] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross," Generating communication for array statements: Design, implementation, and evaluation," *JPDC*, Vol. 21, pp. 150-159, 1994.

[16] Rajeev. Thakur, Alok. Choudhary, and J. Ramanujam, "Efficient Algorithms for Array Redistribution, " *IEEE Trans. on PDS*, vol. 7, no. 6 , June, 1996.

[17] 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.

[18] Akiyoshi Wakatani and Michael Wolfe, "A New Approach to Array Redistribution: Strip Mining Redistribution," In *Proc. of Parallel Architectures and Languages*, July 1994.

[19] Akiyoshi Wakatani and Michael Wolfe, "Optimization of Array Redistribution for Distributed Memory Multicomputers, " short communication, *Parallel Computing*, Vol. 21, Number 9, pp. 1485-1490, September 1995.

Table 1: The time of four algorithms to execute a BC(5,8) to BC(8,5) redistribution on different number of processors with fixed array size (N_0, N_1) = (400, 640).

	Prylli's			*PITFALLS*			*BBC*			*CDC*		
Processor grid	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total
8×2	1.498	25.617	70.930	2.236	26.423	72.63	1.396	25.120	68.408	3.648	20.954	66.675
8×3	2.038	18.604	59.882	3.110	18.442	60.226	1.386	18.781	56.085	3.489	15.894	55.856
8×4	2.381	12.558	50.664	4.166	12.893	49.955	1.403	11.077	46.21	3.013	10.171	48.053
8×5	2.445	11.346	41.245	4.585	11.637	43.96	1.383	9.771	38.43	2.607	8.945	42.022
8×6	3.748	8.226	31.417	5.175	8.259	34.378	1.388	7.179	28.538	2.241	6.542	31.824
8×7	4.492	6.389	22.372	5.421	6.351	23.221	1.394	5.304	18.914	2.152	4.869	21.555

Time(ms)

Table 2: The time of four algorithms to execute a BC(10, 20) to BC(5, 10) redistribution on different number of processors with fixed array size (N_0, N_1) = (400, 640).

	Prylli's			*PITFALLS*			*BBC*			*CDC*		
Processor grid	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total
8×2	1.414	24.140	69.143	2.138	26.089	71.119	1.126	23.137	66.149	3.128	20.100	64.112
8×3	2.142	17.233	58.265	3.252	18.162	60.207	1.165	16.229	54.290	2.545	14.178	53.210
8×4	2.241	12.583	49.585	4.148	12.345	49.438	1.141	11.263	44.584	2.453	10.350	47.426
8×5	2.409	11.136	40.191	4.523	10.556	43.100	1.178	8.131	37.932	2.120	8.553	41.105
8×6	3.125	8.148	30.465	5.122	8.140	34.133	1.162	6.551	27.079	1.710	6.305	30.405
8×7	4.220	6.302	21.868	5.508	6.216	23.317	1.156	5.004	17.156	1.413	4.868	20.733

Time(ms)

Table 3: The time of four algorithms to execute a (`BLOCK`, `BLOCK`) to (`CYCLIC`, `CYCLIC`) redistribution on different number of processors with fixed array size (N_0, N_1) = (400, 640).

	Prylli's			*PITFALLS*			*BBC*			*CDC*		
Processor grid	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total	Indexing	Packing/ Unpacking	Total
8×2	5.093	26.105	74.802	6.112	26.860	76.758	3.242	19.479	63.530	3.673	19.516	63.572
8×3	5.121	19.113	62.119	6.126	19.907	63.766	3.170	13.511	52.643	3.132	13.593	52.557
8×4	5.149	13.138	53.171	6.134	13.105	52.115	2.114	9.636	43.857	2.174	9.621	43.121
8×5	5.231	11.310	44.248	6.301	11.152	46.208	2.184	7.250	35.190	2.272	7.166	35.201
8×6	5.363	8.551	33.348	6.515	8.237	36.399	1.396	5.482	25.416	1.510	5.336	25.328
8×7	5.903	7.108	24.943	6.603	7.685	25.729	1.804	3.984	16.894	1.934	3.962	16.644

Time(ms)

Table 4: The time of different algorithms to execute different redistribution on a two-dimensional array with various array size on a 56-node SP2, (N_0, N_1) = (1200, 1600).

	Prylli's	*PITFALLS*	*BBC*	*CDC*
Array size	BC(5, 8) to BC(8, 5)			
(N_0,N_1)	28.367	35.202	27.771	26.763
($2N_0$,$2N_1$)	144.630	150.013	130.002	123.407
($3N_0$,$3N_1$)	321.218	335.736	317.212	297.565
($4N_0$,$4N_1$)	511.111	534.277	503.035	489.143
	BC(10, 20) to BC(5, 10)			
(N_0,N_1)	27.326	33.454	27.277	25.968
($2N_0$,$2N_1$)	144.408	168.581	134.247	120.319
($3N_0$,$3N_1$)	327.077	342.153	305.005	291.011
($4N_0$,$4N_1$)	518.172	539.914	508.474	484.268
	(BLOCK, BLOCK) to (CYCLIC, CYCLIC)			
(N_0,N_1)	29.545	32.238	24.565	24.192
($2N_0$,$2N_1$)	150.153	153.357	135.406	135.497
($3N_0$,$3N_1$)	451.118	491.118	402.924	402.799
($4N_0$,$4N_1$)	931.347	1045.838	566.802	565.324

Time(ms)

Table 5: The time of different algorithms to execute different redistribution on a three-dimensional array with various array size on a 56-node SP2, (N_0, N_1, N_2) = (120, 180, 160).

	Prylli's	*PITFALLS*	*BBC*	*CDC*
Array size	BC(5, 10, 20) to BC(10, 20, 5)			
(N_0,N_1,N_2)	50.961	52.100	45.476	44.423
($2N_0$,$2N_1$,$2N_2$)	236.156	240.086	229.271	225.303
($3N_0$,$3N_1$,$3N_2$)	409.062	427.057	361.258	343.309
($4N_0$,$4N_1$,$4N_2$)	910.413	973.718	869.111	807.249
	BC(10, 20, 30) to BC(1, 2, 3)			
(N_0,N_1,N_2)	51.319	51.292	49.134	43.952
($2N_0$,$2N_1$,$2N_2$)	244.283	255.721	238.697	227.676
($3N_0$,$3N_1$,$3N_2$)	445.187	469.731	410.987	368.073
($4N_0$,$4N_1$,$4N_2$)	812.320	873.900	750.708	631.445
	(BLOCK, BLOCK, BLOCK) to (CYCLIC, CYCLIC, CYCLIC)			
(N_0,N_1,N_2)	61.545	77.990	37.964	37.414
($2N_0$,$2N_1$,$2N_2$)	363.723	383.345	250.983	249.725
($3N_0$,$3N_1$,$3N_2$)	552.444	623.724	326.750	326.750
($4N_0$,$4N_1$,$4N_2$)	1411.378	1493.714	918.662	918.226

Time(ms)