The Journal of Supercomputing, 17, 23–46, 2000

Efficient Methods for Multi-Dimensional Array Redistribution

CHING-HSIEN HSU, YEH-CHING CHUNG, AND CHYI-REN DOW

chhsu@fcu.edu.tw
ychung@fcu.edu.tw
crdow@fcu.edu.tw

Department of Information Engineering, Feng Chia University, Taichung, Taiwan 407, ROC

Final version accepted January 8, 1999

Abstract. In many scientific applications, array redistribution is usually required to enhance data locality and reduce remote memory access on distributed memory multicomputers. Since the redistribution is performed at run-time, there is a performance tradeoff between the efficiency of the new data decomposition for a subsequent phase of an algorithm and the cost of redistributing data among processors. 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 also 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 array size is large. When implemented these two techniques on an IBM SP2 parallel machine along with the *PITFALLS* method and the *Prylli's* method, 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.

Keywords: array redistribution, distributed memory multicomputers, the basic-block calculation technique, the complete-dimension calculation technique

1. Introduction

The data parallel programming model has become a widely accepted paradigm for programming distributed memory multicomputers. To efficiently execute a data parallel program on a distributed memory multicomputer, an appropriate data decomposition is critical. The data decomposition involves *data distribution* and *data alignment*. The data distribution deals with how data arrays should be distributed. The data alignment deals with how data arrays should be aligned with respect to one another. The purpose of data decomposition is to balance the computational load and minimize the communication overheads.

Many data parallel programming languages such as High Performance Fortran (HPF) [9], Fortran D [6], Vienna Fortran [33], and High Performance C (HPC) [28] provide compiler directives for programmers to specify array distribution. The array distribution provided by those languages, in general, can be classified into two categories, *regular* and *irregular*. The regular array distribution, in general, has three types, `BLOCK`, `CYCLIC`, and `BLOCK-CYCLIC`(c). The `BLOCK-CYCLIC`(c) is the

most general regular array distribution among them. Dongarra *et al*. [5] have shown that these distribution are essential for many dense matrix algorithms design in distributed memory machines. The irregular array distribution uses user-defined array distribution functions to specify array distribution.

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. 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 tradeoff 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.

In this paper, based on the *basic-cycle calculation technique* [4], we present a *basic-block calculation* (*BBC*) and a *complete-dimension calculation* (*CDC*) technique 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. In this paper, we also developed a theoretical model to analyze the tradeoff between these two techniques. The two techniques can be easily implemented in a parallelizing compiler, run-time systems, or parallel programs.

This paper is organized as follows. In Section 2, a brief survey of related work will be presented. In Section 3, we will introduce notations and terminology used in this paper. Section 4 presents the basic-block calculation and the complete-dimension calculation techniques for multi-dimensional array redistribution. The theoretical model to analyze the performance tradeoff of these two methods will also be presented in this section. In Section 5, the experimental results of the basic-block calculation technique, the complete-dimension calculation technique, the *PITFALLS* method, and the *Prylli's* method will be given.

2. Related work

Many methods for performing array redistribution have been presented in the literature. Since techniques of redistribution can be performed either by using the multicomputer compiler technique [27] or using the runtime support technique, we briefly describe the related research in these two approaches.

Gupta *et al*. [7] derived closed form expressions to efficiently determine the send/receive processor/data sets. They also provided a virtual processor approach [8]

for addressing the problem of reference index-set identification for array statements with BLOCK-CYCLIC(c) distribution and formulated active processor sets as closed forms. A recent work in [16] extended the virtual processor approach to address the problem of memory allocation and index-sets identification. By using their method, closed form expressions for index-sets of arrays that were mapped to processors using one-level mapping can be translated to closed form expressions for index-sets of arrays that were mapped to processors using two-level mapping and vice versa. A similar approach that addressed the problems of the index set and the communication sets identification for array statements with BLOCK-CYCLIC(c) distribution was presented in [24]. In [24], the CYCLIC(k) distribution was viewed as an union of k CYCLIC distribution. Since the communication sets for CYCLIC distribution is easy to determine, communication sets for CYCLIC(k) distribution can be generated in terms of unions and intersections of some CYCLIC distributions.

In [3], Chatterjee *et al*. enumerated the local memory access sequence of communication sets for array statements with BLOCK-CYCLIC(c) distribution based on a finite-state machine. In this approach, the local memory access sequence can be characterized by a FSM at most c states. In [17], Kennedy *et al*. also presented algorithms to compute the local memory access sequence for array statements with BLOCK-CYCLIC(c) distribution. Lee *et al*. [18] derived communication sets for statements of arrays which were distributed in arbitrary BLOCK-CYCLIC(c) fashion. They also presented closed form expressions of communication sets for restricted block size.

Thakur *et al.* [25, 26] presented algorithms for run-time array redistribution in HPF programs. For BLOCK-CYCLIC(kr) to BLOCK-CYCLIC(r) redistribution (or vice versa), in most cases, a processor scanned its local array elements once to determine the destination (source) processor for each block of array elements of size r in the local array. In [10], an approach for generating communication sets by computing the intersections of index sets corresponding to the LHS and RHS of array statements was presented. The intersections are computed by a scanning approach that exploits the repetitive pattern of the intersection of two index sets. In [22, 23], Ramaswamy and Banerjee used a mathematical representation, *PITFALLS*, for regular data redistribution. The basic idea of *PITFALLS* is to find all intersections between source and destination distributions. Based on the intersections, the send/receive processor/data sets can be determined and general redistribution algorithms can be devised. Prylli *et al*. [21] proposed runtime scan algorithm for BLOCK-CYCLIC array redistribution. Their approach has the same time complexity as that proposed in [23], but has simple basic operation compared to that proposed in [23]. The disadvantage of these approaches is that, when the number of processors is large, iterations of the out-most loop in intersection algorithms increased as well. This leads to high indexing overheads and degrades the performance of a redistribution algorithm.

In [32], a spiral mapping technique was proposed. The main idea of this approach was to map formal processors onto actual processors such that the global communication can be translated to the local communication in a certain processor group. Since the communication is local to a processor group, one can reduce communication conflicts when performing a redistribution. Kalns and Ni [12, 13]

proposed a processor mapping technique to minimize the amount of data exchange for BLOCK to BLOCK-CYCLIC(c) redistribution and vice versa. Using the data to logical processors mapping, they show that the technique can achieve the maximum ratio between data retained locally and the total amount of data exchanged. Walker *et al*. [30] used the standardized message passing interface, MPI, to express the redistribution operations. They implemented the BLOCK-CYCLIC array redistribution algorithms in a synchronous and an asynchronous scheme. Since the excessive synchronization overheads incurred from the synchronous scheme, they also presented the random and optimal scheduling algorithms for BLOCK-CYCLIC array redistribution.

Kaushik *et al*. [14, 15] proposed a multi-phase redistribution approach for BLOCK-CYCLIC(s) to BLOCK-CYCLIC(t) redistribution. The main idea of multi-phase redistribution is to perform a redistribution as a sequence of redistribution such that the communication cost of data movement among processors in the sequence is less than that of direct redistribution. Instead of redistributing the entry array at one time, a strip mining approach was presented in [31]. In this approach, portions of array elements were redistributed in sequence in order to overlap the communication and computation. In [19], a generalized circulant matrix formalism was proposed to reduce the communication overheads for BLOCK-CYCLIC(r) to BLOCK-CYCLIC(kr) redistribution. Using the generalized circulant matrix formalism, the authors derived direct, indirect, and hybrid communication schedules for the cyclic redistribution with the block size changed by an integer factor k. They also extended this technique to solve some multi-dimensional redistribution problem [20]. However, as the array size increased, the above methods will have a large amount of extra transmission costs and degrades the performance of a redistribution algorithm.

3. Preliminaries

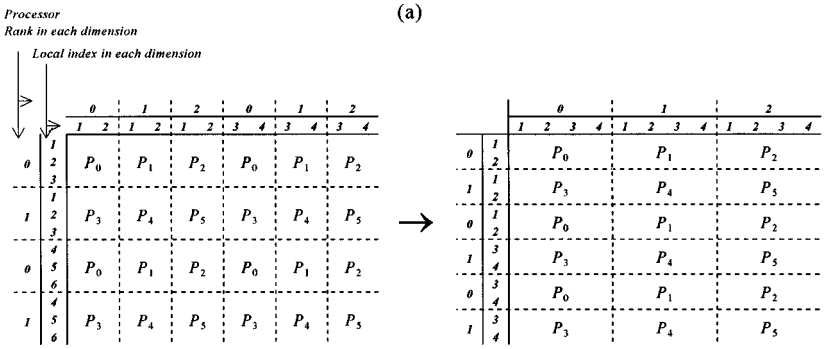
In this section, we will present the notations and terminology used in this paper. To simplify the presentation, we use $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ to represent the $(\texttt{CYCLIC}(s_0), \texttt{CYCLIC}(s_1), \ldots, \texttt{CYCLIC}(s_{n-1}))$ 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 $A^{(n)} = A[1{:}\,n_0, 1{:}\,n_1, \ldots, 1{:}\,n_{n-1}] = \{a_{d_0, d_1, \ldots, d_{n-1}} | 0 \leq d_\ell \leq n_\ell - 1, 0 \leq \ell \leq n-1\}$. The size of array $A^{(n)}$, denoted by $|A^{(n)}|$, is equal to $n_0 \times n_1 \times \cdots \times n_{n-1}$. In this paper, we assume that array elements are stored in a memory by a row-major manner.

Figure 1(a) shows a two-dimensional array $A^{(2)} = A[1{:}12, 1{:}12]$. There are 12×12 array elements in $A[1{:}12, 1{:}12]$, i.e., $|A^{(2)}| = 144$. In Figure 1(a), we use the italic fonts and the normal fonts to represent the indices of each dimension of array $A[1{:}12, 1{:}12]$ and the global array indices of array $A[1{:}12, 1{:}12]$, respectively. We assume that array elements were stored in a row major fashion and the array index starts from 1.

	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	13	14	15	16	17	18	19	20	21	22	23	24
3	25	26	27	28	29	30	31	32	33	34	35	36
4	37	38	39	40	41	42	43	44	45	46	47	48
5	49	50	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70	71	72
7	73	74	75	76	77	78	79	80	81	82	83	84
8	85	86	87	88	89	90	91	92	93	94	95	96
9	97	98	99	100	101	102	103	104	105	106	107	108
10	109	110	111	112	113	114	115	116	117	118	119	120
11	121	122	123	124	125	126	127	128	129	130	131	132
12	133	134	135	136	137	138	139	140	141	142	143	144

(a)



(b)

$$SLA_0^{(2)} = \begin{pmatrix} 1 & 2 & 7 & 8 \\ 13 & 14 & 19 & 20 \\ 25 & 26 & 31 & 32 \\ 73 & 74 & 79 & 80 \\ 85 & 86 & 91 & 92 \\ 97 & 98 & 103 & 104 \end{pmatrix}, \quad SLA_{0,0}^{(2)} = \begin{pmatrix} 1 \\ 13 \\ 25 \\ 73 \\ 85 \\ 97 \end{pmatrix}, \quad SLA_{0,1}^{(2)} = \begin{pmatrix} 1 \\ 2 \\ 7 \\ 8 \end{pmatrix},$$

$$SLA_{0,0}^{(2)}[2] = \{13\}, \quad SLA_{0,0}^{(2)}[4] = \{73\}, \quad SLA_{0,1}^{(2)}[2] = \{2\}, \quad SLA_{0,1}^{(2)}[4] = \{8\}$$

(c)

Figure 1. (a) A global array $A[12, 12]$. (b) A BC(3, 2) → BC(2, 4) redistribution on $A[1{:}12, 1{:}12]$ over $M[2, 3]$. (c) Examples of $SLA_i^{(n)}$, $SLA_{i,\ell}^{(n)}$, and $SLA_{i,\ell}^{(n)}[r]$.

Definition 2. An n-dimensional processor grid is defined as the set of processors $M^{(n)} = M[m_0, m_1, \ldots, m_{n-1}] = \{\tilde{p}_{d_0, d_1, \ldots, d_{n-1}} | 0 \le d_\ell \le m_\ell - 1,\ 0 \le \ell \le n-1\}$. The number of processors of $M^{(n)}$, denoted by $|M^{(n)}|$, is equal to $m_0 \times m_1 \times \cdots \times m_{n-1}$.

Figure 1(b) shows a BC(3, 2) to BC(2, 4) redistribution on $A[1{:}12, 1{:}12]$ over a processor grid $M[2, 3]$ with six processors. The shadow portions represent the array elements distributed to processor P_0 before and after the redistribution.

Definition 3. Given an n-dimensional processor grid $M^{(n)}$, the rank of processor $\tilde{p}_{d_0, d_1, \ldots, d_{n-1}}$ 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 $\tilde{p}_{d_0, d_1, \ldots, d_{n-1}}$ in this paper, where $0 \le i \le |M^{(n)}| - 1$.

According to Definition 3, we know that $\tilde{p}_{0,0} = P_0$, $\tilde{p}_{0,1} = P_1$, $\tilde{p}_{0,2} = P_2$, $\tilde{p}_{1,0} = P_3$, $\tilde{p}_{1,1} = P_4$, $\tilde{p}_{1,2} = P_5$.

Definition 4. Given an $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution, $\texttt{BC}(s_0, s_1, \ldots, s_{n-1})$, $\texttt{BC}(t_0, t_1, \ldots, t_{n-1})$, 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 $0 \le \ell \le n-1$.

In Figure 1(b), the source distribution is $\texttt{BC}(3, 2)$. The destination distribution is $\texttt{BC}(2, 4)$. The source distribution factors in the first and the second dimension are equal to three and two, respectively. The destination distribution factors in the first and the second dimension are equal to two and four respectively.

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

Definition 6. We define $SLA_{i,\ell}^{(n)}$ as the set of array elements in the first row of the ℓth dimension of $SLA_i^{(n)}$, i.e., $SLA_{i,\ell}^{(n)} = SLA_i^{(n)}[1, \ldots, 1, 1{:}(n_\ell/m_\ell), 1, \ldots, 1]$, 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 n_ℓ/m_ℓ. $SLA_{i,\ell}^{(n)}[r]$ is defined as the rth array element of $SLA_{i,\ell}^{(n)}$.

Figure 1(c) shows examples of notations that were defined in Definitions 5 and 6. In Figure 1(c), there are 24 array elements in $SLA_0^{(2)}$. The sets of array elements in $SLA_{0,0}^{(2)}$ and $SLA_{0,1}^{(2)}$ are $\{1, 13, 25, 73, 85, 97\}$ and $\{1, 2, 7, 8\}$, respectively. The second and the fourth elements in $SLA_{0,0}^{(2)}$ are $SLA_{0,0}^{(2)}[2] = \{13\}$ and $SLA_{0,0}^{(2)}[4] = \{73\}$, respectively. The second and the fourth elements in $SLA_{0,1}^{(2)}$ are $SLA_{0,1}^{(2)}[2] = \{2\}$ and $SLA_{0,1}^{(2)}[4] = \{8\}$, respectively.

Definition 7. Given a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution on $A^{(n)}$ over $M^{(n)}$, a *basic-cycle* of the ℓth dimension of $SLA_i^{(n)}$ (or $DLA_j^{(n)}$), denoted by BC_ℓ, is defined as the quotient of the least common multiple of s_ℓ and t_ℓ to the greatest common divisor of s_ℓ and t_ℓ, i.e., $BC_\ell = lcm(s_\ell, t_\ell)/gcd(s_\ell, t_\ell)$. We

define $SLA^{(n)}_{i,\ell}[1{:}BC_\ell]$ ($DLA^{(n)}_{j,\ell}[1{:}BC_\ell]$) as the first basic-cycle of $SLA^{(n)}_{i,\ell}$ ($DLA^{(n)}_{j,\ell}$) of processor P_i (P_j), $SLA^{(n)}_{i,\ell}[BC_\ell + 1{:}2 \times BC_\ell]$ ($DLA^{(n)}_{i,\ell}[BC_\ell + 1{:}2 \times BC_\ell]$) as the second basic-cycle of $SLA^{(n)}_{i,\ell}$ ($DLA^{(n)}_{j,\ell}$) of processor P_i (P_j), and so on, where $0 \leq \ell \leq n-1$.

In the BC(3, 2) to BC(2, 4) redistribution shown in Figure 1(b), in the first dimension, the source and the destination distribution factor are equal to three and two, respectively. According to the above definition, the basic-cycle of the first dimension is $BC_0 = lcm(3,2)/gcd(3,2) = 6$. In the second dimension, the source and the destination distribution factor are equal to two and four, respectively. According to the above definition, the basic-cycle of the first dimension is $BC_1 = lcm(2,4)/gcd(2,4) = 2$.

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

In Figure 1(b), $BC_0 = 6$ and $BC_1 = 2$. According to Definition 8, the basic-block is equal to $BC_0 \times BC_1 = 12$.

4. Multi-dimensional array redistribution

To perform a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1})$ to $\texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution, in general, a processor needs to compute the communication sets. Based on the characteristics of a redistribution, we have the following lemmas.

Lemma 1. *Given a* $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ *redistribution on* $A^{(n)}$ *over* $M^{(n)}$, *for a source* (*destination*) *processor* P_i, *if the rank of the destination* (*source*) *processor of* $SLA^{(n)}_{i,k}[r_k]$ ($DLA^{(n)}_{i,k}[r_k]$) *is* $\tilde{p}_{0,\ldots 0,\, j_k,\, 0,\ldots,0}$, *where* $0 \leq i \leq |M^{(n)}| - 1$, $k = 0$ *to* $n-1$, $0 \leq j_k \leq m_k - 1$, *and* $1 \leq r_k \leq \lceil n_k/m_k \rceil$, *then the destination* (*source*) *processor of* $SLA^{(n)}_i[r_0, r_1, \ldots, r_{n-1}]$ ($DLA^{(n)}_i[r_0, r_1, \ldots, r_{n-1}]$) *is* $P_j = \tilde{p}_{j_0,\, j_1, \ldots, j_{n-1}}$, *where* $j = \sum_{k=0}^{n-1}(j_k \times \prod_{\ell=k+1}^{n-1} m_\ell)$.

Proof. We only prove the source processor part. The proof of the destination processor part is similar. In a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution, we assume that the destination processor of $SLA^{(n)}_i[r_0][r_1]\ldots[r_{n-1}]$, is $\tilde{p}_{d_0,\, d_1,\, \ldots,\, d_{n-1}}$, where $0 \leq d_\ell \leq m_\ell - 1$ and $0 \leq \ell \leq n-1$. If the destination processor of $SLA^{(n)}_{i,0}[r_0]$ is $\tilde{p}_{j_0,\, 0,\, \ldots,\, 0}$, then $d_0 = j_0$, where $0 \leq j_0 \leq m_0 - 1$ and $1 \leq r_0 \leq \lceil n_0/m_0 \rceil$. For the same reason, if the destination processors of $SLA^{(n)}_{i,1}[r_1], SLA^{(n)}_{i,2}[r_2], \ldots, SLA^{(n)}_{i,n-1}[r_{n-1}]$ are $\tilde{p}_{0,\, j_1,\, 0,\, \ldots,\, 0}$, $\tilde{p}_{0,\, 0,\, j_2,\, 0,\, \ldots,\, 0}$, …, and $\tilde{p}_{0,\, \ldots,\, 0,\, j_{n-1}}$, respectively, we have $d_1 = j_1, d_2 = j_2, \ldots$, and $d_{n-1} = j_{n-1}$. Therefore, according to Definition 3, if the rank of the destination processor of $SLA^{(n)}_{i,k}[r_k]$ is $\tilde{p}_{0,\ldots 0,\, j_k,\, 0,\, \ldots,\, 0}$, where $0 \leq i \leq |M^{(n)}| - 1$, $k = 0$

to $n-1$, $0 \le j_k \le m_k - 1$, and $1 \le r_k \le \lceil n_k/m_k \rceil$, then the destination processor of $SLA_i^{(n)}[r_0][r_1]\ldots[r_{n-1}]$ is $P_j = \tilde{p}_{j_0, j_1, \ldots, j_{n-1}}$, 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 $SLA_i^{(n)}[r_0, r_1, \ldots, r_{n-1}]$ ($DLA_j^{(n)}[r_0, r_1, \ldots, r_{n-1}]$) can be determined by the ranks of the 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. In this section, we present two efficient techniques, the basic-block calculation technique and the complete-dimension calculation technique, to deal with this problem. Both techniques are based on the basic-cycle calculation technique proposed in [4]. The main idea of the basic-cycle calculation technique is based on the following lemma.

Lemma 2. *Given a* $\texttt{BC}(s) \rightarrow \texttt{BC}(t)$ *and a* $\texttt{BC}(s/gcd(s,t)) \rightarrow \texttt{BC}(t/gcd(s,t))$ *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* $\texttt{BC}(s/gcd(s,t)) \rightarrow \texttt{BC}(t/gcd(s,t))$ *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* $\texttt{BC}(s) \rightarrow \texttt{BC}(t)$ *redistribution will also be* P_j (P_i), *where* $1 \le k \le \lceil N/(M \times gcd(s,t)) \rceil$.

Proof. We only prove the source processor part. The proof of the destination processor part is similar. For a source processor P_i, if the global array index of $SLA_i[k]$ in $\texttt{BC}(s/gcd(s,t)) \rightarrow \texttt{BC}(t/gcd(s,t))$ redistribution is α, then the global array indices of $SLA_i[(k-1) \times gcd(s,t) + 1{:}k \times gcd(s,t)]$ in $\texttt{BC}(s) \rightarrow \texttt{BC}(t)$ redistribution are $(\alpha-1) \times gcd(s,t) + 1, (\alpha-1) \times gcd(s,t) + 2, \ldots, \alpha \times gcd(s,t)$. If $A[1{:}N]$ is distributed by $\texttt{BC}(t/gcd(s,t))$ distribution, then $A[\alpha]$ is in the $\lceil \alpha \times gcd(s,t)/t \rceil$th block of size $t/gcd(s,t)$. If $A[1{:}N]$ is distributed by $\texttt{BC}(t)$ distribution, then $A[(\alpha-1) \times gcd(s,t) + 1]$, $A[(\alpha-1) \times gcd(s,t) + 2]$, ..., and $A[\alpha \times gcd(s,t)]$ are in the $\lceil (\alpha-1) \times gcd(s,t) + 1/t \rceil$th, the $\lceil (\alpha-1) \times gcd(s,t) + 2/t \rceil$th, ..., and the $\lceil (\alpha \times gcd(s,t)/t \rceil$th block of size t, respectively. Since $\lceil (\alpha-1) \times gcd(s,t) + 1/t \rceil = \lceil (\alpha-1) \times gcd(s,t) + 2/t \rceil = \ldots = \lceil \alpha \times gcd(s,t)/t \rceil$, if the destination processor of $A[\alpha]$ is P_j in $\texttt{BC}(s/gcd(s,t)) \rightarrow \texttt{BC}(t/gcd(s,t))$ redistribution, then the destination processors of $A[(\alpha-1) \times gcd(s,t) + 1]$, $A[(\alpha-1) \times gcd(s,t) + 2]$, ..., and $A[\alpha \times gcd(s,t)]$ are P_j in $\texttt{BC}(s) \rightarrow \texttt{BC}(t)$ redistribution. Therefore, if the destination processor of $SLA_i[k]$ in $\texttt{BC}(s/gcd(s,t)) \rightarrow \texttt{BC}(t/gcd(s,t))$ redistribution is P_j, then the destination processors of $SLA_i[(k-1) \times gcd(s,t) + 1{:}k \times gcd(s,t)]$ in $\texttt{BC}(s) \rightarrow \texttt{BC}(t)$ redistribution will also be P_j, where $0 \le i, j \le M-1$ and $1 \le k \le \lceil N/(M \times gcd(s,t)) \rceil$. ∎

Given a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution, according to Lemmas 1, and 2, we know that the communication sets of $\texttt{BC}(s_0/gcd(s_0, t_0), s_1/gcd(s_1, t_1), \ldots, s_{n-1}/gcd(s_{n-1}, t_{n-1})) \rightarrow \texttt{BC}(t_0/gcd(s_0, t_0), t_1/gcd(s_1, t_1), \ldots, t_{n-1}/gcd(s_{n-1}, t_{n-1}))$ redistribution can be used to generate the communication

sets of $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution. Therefore, in the following discussion, for a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution, we assume that $gcd(s_i, t_i)$ is equal to 1, where $1 \leq i \leq n-1$. 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.

4.1. The basic-block calculation technique

Given a $\texttt{BC}(s_0, s_1) \rightarrow \texttt{BC}(t_0, t_1)$ 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 the two basic cycles, $SLA^{(2)}_{i,0}[1{:}\,BC_0]$ and $SLA^{(2)}_{i,1}[1{:}\,BC_1]$. Then it can generate the destination processor sets for the basic block, $SLA^{(2)}_{i}[1{:}\,BC_0, 1{:}\,BC_1]$. For example, if the destination processors of $SLA^{(2)}_{i,0}[r_0]$ and $SLA^{(2)}_{i,1}[r_1]$ are $\tilde{p}_{j_0,0}$ and $\tilde{p}_{0,j_1}$, respectively, the destination processor P_j of $SLA^{(2)}_{i}[r_0][r_1]$ can be determined by the following equation,

$$Rank(P_j) = j_0 m_1 + j_1, \tag{1}$$

where $rank(P_j)$ is the rank of destination processor P_j.

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 $0 \leq i_0 \leq m_0 - 1$ and $0 \leq i_1 \leq m_1 - 1$. Since the values of i_0 and i_1 are known, the destination processors of $SLA^{(2)}_{i,0}[1{:}\,BC_0]$ and $SLA^{(2)}_{i,1}[1{:}\,BC_1]$ can be determined by the following equation,

$$DP_{(\ell)} = Destination\_Processors(SLA^{(2)}_{i,\ell}[1{:}\,BC_\ell]) = \begin{bmatrix} F(1) \\ F(2) \\ \vdots \\ F(BC_\ell) \end{bmatrix}_{BC_\ell \times 1} \tag{2}$$

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

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

where $x = 1$ to BC_ℓ, i_ℓ is the rank of source processor P_i in the ℓth dimension, $\ell = 0, 1$.

For a two-dimensional array redistribution, from Equation 2, we can obtain $DP_{(0)}$ and $DP_{(1)}$ that represent the destination processors of $SLA^{(2)}_{i,0}[1{:}\,BC_0]$ and $SLA^{(2)}_{i,1}[1{:}\,BC_1]$, 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

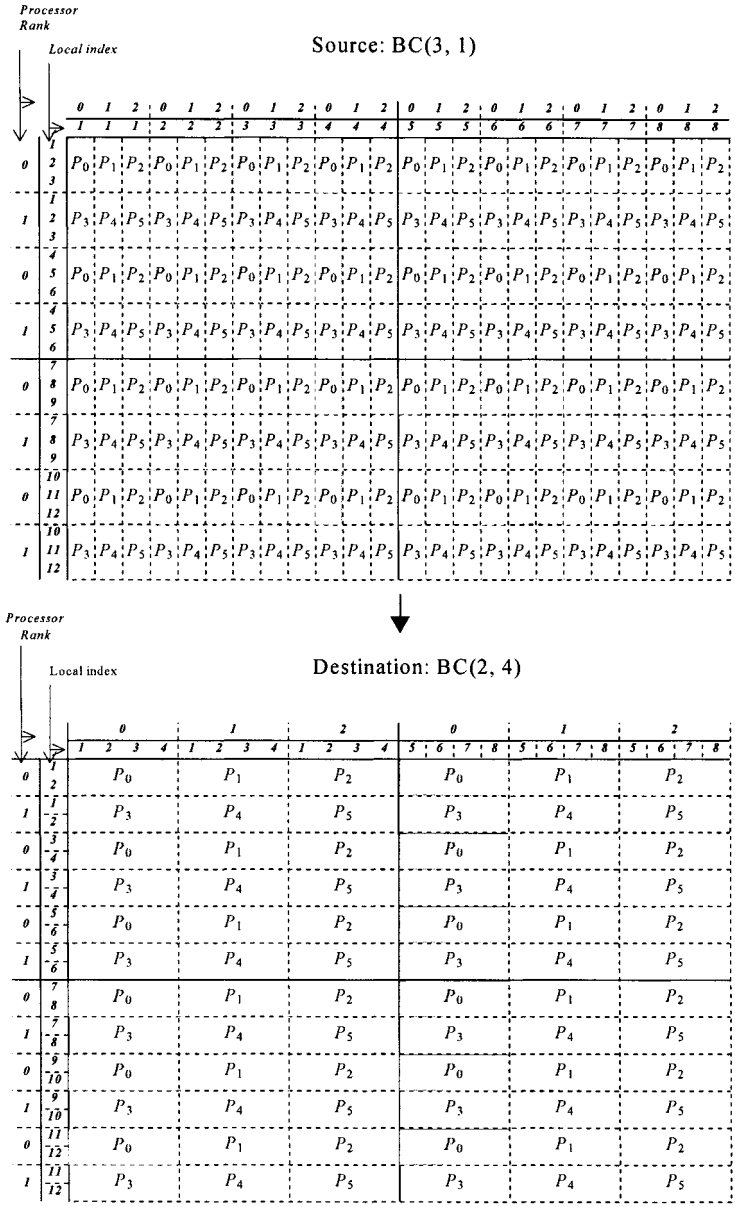


Figure 2. A BC(3, 1) → BC(2, 4) redistribution on $A[1{:}24, 1{:}24]$ over $M[2, 3]$.

first basic-block of $SLA_i^{(2)}$, i.e., $SLA_i^{(2)}[1{:}BC_0, 1{:}BC_1]$. Figure 2 shows an example of a BC(3, 1) → BC(2, 4) redistribution on $A[1{:}24, 1{:}24]$ over $M[2, 3]$. In this example, $BC_0 = 6$ and $BC_1 = 4$. For source processor P_0 ($= \tilde{p}_{0,0}$), according to Equation 2, the destination processors of $SLA_{0,0}^{(2)}[1{:}BC_0]$ and $SLA_{0,1}^{(2)}[1{:}BC_1]$ are $DP_{(0)} = [0, 0, 1, 1, 1, 0]$ and $DP_{(1)} = [0, 0, 1, 2]$, respectively. Based on $DP_{(0)}$, $DP_{(1)}$,

and Equation 1, the destination processors of $SLA_0^{(2)}[1{:}6, 1{:}4]$ are

$$\begin{bmatrix} 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 2 \\ 3 & 3 & 4 & 5 \\ 3 & 3 & 4 & 5 \\ 3 & 3 & 4 & 5 \\ 0 & 0 & 1 & 2 \end{bmatrix}_{6\times 4}.$$

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

Lemma 3. *Given a* $\mathtt{BC}(s) \rightarrow \mathtt{BC}(t)$ *redistribution on a one-dimensional array* $A[1{:}N]$ *over M processors*, $SLA_i[m]$, $SLA_i[m + lcm(s,t)]$, $SLA_i[m + 2 \times lcm(s,t)]$, ..., *and* $SLA_i[m + (\lfloor N/lcm(s,t) \times M \rfloor - 1) \times lcm(s,t)]$ *have the same destination processor, where* $0 \le i \le M-1$ *and* $1 \le m \le lcm(s,t)$.

Proof. The proof of this lemma can be found in [11]. ∎

Lemma 4. *Given a* $\mathtt{BC}(s_0, s_1) \rightarrow \mathtt{BC}(t_0, t_1)$ *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* $1 \le k_1 \le \lfloor n_1/(lcm(s_1,t_1) \times m_1) \rfloor$.

Proof. The proof of this lemma can be easily established according to Lemma 3. ∎

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 $\mathtt{BC}(s_0, s_1) \rightarrow \mathtt{BC}(t_0, t_1)$ 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, that have the same destination processor, in $SLA_{i,l}^{(2)}[1{:}BC_\ell]$ to tables, called *Send_Tables*, we can also determine the destination processors of $SLA_i^{(2)}[1{:}BC_0, 1{:}BC_1]$ from *Send_Tables*. For example, considering the $BC(3,1) \rightarrow BC(2,4)$ redistribution shown in Figure 2. For the source processor P_0, since $DP_{(0)}$= [0, 0, 1, 1, 1, 0], $SLA_{0,0}^{(2)}[1]$, $SLA_{0,0}^{(2)}[2]$ and $SLA_{0,0}^{(2)}[6]$ have the same destination processor $\tilde{p}_{0,0}$. $SLA_{0,0}^{(2)}$ [3], $SLA_{0,0}^{2}$ [4] and $SLA_{0,0}^{(2)}$ [5] have the same destination processor $\tilde{p}_{1,0}$. Therefore, the indices of array elements in $SLA_{0,0}^{(2)}[1{:}6]$ can be classified into two sets as shown in Figure 3(a). Since $DP_{(1)}$=

Send_Tables ST_0	*DP*	*Indices set*
	$\tilde{p}_{0,0}$	1, 2, 6
	$\tilde{p}_{1,0}$	3, 4, 5

(a)

ST_1	*DP*	*Indices set*
	$\tilde{p}_{0,0}$	1, 2
	$\tilde{p}_{0,1}$	3
	$\tilde{p}_{0,2}$	4

(b)

Figure 3. The *Send_Tables* for $SLA^{(2)}_{0,0}[1{:}BC_0]$ and $SLA^{(2)}_{0,1}[1{:}BC_1]$.

[0, 0, 1, 2], $SLA^{(2)}_{0,1}[1]$ and $SLA^{(2)}_{0,1}[2]$ have the same destination processor $\tilde{p}_{0,0}$. The destination processors of $SLA^{(2)}_{0,1}[3]$ and $SLA^{(2)}_{0,1}[4]$ are $\tilde{p}_{0,1}$ and $\tilde{p}_{0,2}$, respectively. Therefore, the indices of array elements in $SLA^{(2)}_{0,1}[1{:}4]$ can be classified into three sets as shown in Figure 3(b).

Based on the *Send_Tables*, we can pack array elements in a source local array to messages, considering the message that processor P_0 will send to the destination processor P_4 in the example shown in Figure 2. Since the destination processor $P_4 = \tilde{p}_{1,1}$, according to Figure 3 and Lemma 1, the destination processor of $SLA^{(2)}_0[3][3]$, $SLA^{(2)}_0[4][3]$ and $SLA^{(2)}_0[5][3]$ is P_4. According to Lemma 4, each basic-block has the same communication patterns. Processor P_0 will also send $SLA^{(2)}_0[3][3+4]$, $SLA^{(2)}_0[4][3+4]$, and $SLA^{(2)}_0[5][3+4]$ in the second basic-block, $SLA^{(2)}_0[3+6][3]$, $SLA^{(2)}_0[4+6][3]$, and $SLA^{(2)}_0[5+6][3]$ in the third basic-block, and $SLA^{(2)}_0[3+6][3+4]$, $SLA^{(2)}_0[4+6][3+4]$, and $SLA^{(2)}_0[5+6][3+4]$ in the fourth basic-block to the destination processor P_4. Note that array elements are packed in a row-major manner for the techniques presented in this paper.

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

$$Rank(P_i) = i_0 m_1 + i_1, \tag{4}$$

where $rank(P_i)$ is the rank of source processor P_i.

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 $DLA^{(2)}_{j,0}[1{:}BC_0]$ and $DLA^{(2)}_{j,1}[1{:}BC_1]$ can be determined by the following equation,

$$SP_{(\ell)} = Source\_Processors\left(DLA^{(2)}_{j,\ell}[1{:}BC_\ell]\right) = \begin{bmatrix} G(1) \\ G(2) \\ \vdots \\ G(BC_\ell) \end{bmatrix}_{BC_\ell \times 1} \tag{5}$$

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

$$G(x) = \left\lfloor \frac{mod\left(\left(j_\ell + m_\ell \times \left\lfloor \frac{x}{t_\ell} \right\rfloor\right) \times t_\ell, m_\ell \times s_\ell\right)}{s_\ell} \right\rfloor \tag{6}$$

where $x = 1$ to BC_ℓ, j_ℓ is the rank of destination processor P_j in the ℓth dimension, $\ell = 0, 1$.

For a two-dimensional array redistribution, from Equation 5, 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($s_0, \ldots, s_{n-1}, t_0, \ldots, t_{n-1}, n_0, \ldots, n_{n-1}, m_0, \ldots, m_{n-1}$)

1. *Construct Send_Tables* (*STs*);
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* (*RTs*);
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

4.2. The complete-dimension calculation technique

In Section 4.1, 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 that are in the first basic-cycle of 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 $\ell = 0$ to $n - 1$. In the following, we will describe the complete-dimension calculation technique in details.

Assume that a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution on an n-dimensional array $A^{(n)} = A[1{:}\,n_0, 1{:}\,n_1, \ldots, 1{:}\,n_{n-1}]$ over $M^{(n)} = M[m_0, m_1, \ldots, m_{n-1}]$ 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 $SLA^{(n)}_{i,0}[1{:}\,L_0]$, $SLA^{(n)}_{i,1}[1{:}\,L_1], \ldots, SLA^{(n)}_{i,n-1}[1{:}\,L_{n-1}]$, where L_k is the local array size in each dimension, i.e., $L_k = (n_k/m_k)$, $k = 0$ to $n-1$. The destination processors of $SLA^{(n)}_{i,0}[1{:}\,L_0]$, $SLA^{(n)}_{i,1}[1{:}\,L_1], \ldots, SLA^{(n)}_{i,n-1}[1{:}\,L_{n-1}]$ can be determined by the following equation:

$$DP_{(\ell)} = Destination\_Processors(SLA^{(n)}_{i,\ell}[1{:}\,L_\ell]) = \begin{bmatrix} F(1) \\ F(2) \\ \vdots \\ F(L_\ell) \end{bmatrix}_{L_\ell \times 1} \tag{7}$$

where $\ell = 0$ to $n-1$. The function $F(x)$ is defined in Equation 3.

For a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution, from Equation 7, we can obtain $DP_{(0)}, DP_{(1)}, \ldots, DP_{(n-1)}$ that represent destination processors of $SLA^{(n)}_{i,0}[1{:}\,L_0], SLA^{(n)}_{i,1}[1{:}\,L_1], \ldots, SLA^{(n)}_{i,n-1}[1{:}\,L_{n-1}]$, respectively. Since the destination processors of $SLA^{(n)}_{i,0}[1{:}\,L_0], SLA^{(n)}_{i,1}[1{:}\,L_1], \ldots, SLA^{(n)}_{i,n-1}[1{:}\,L_{n-1}]$ are known, we can construct the *Send_Tables* for $SLA^{(n)}_{i,0}[1{:}\,L_0], SLA^{(n)}_{i,1}[1{:}\,L_1], \ldots, SLA^{(n)}_{i,n-1}[1{:}\,L_{n-1}]$. For example, consider the redistribution shown in Figure 2. The *Send_Tables* constructed by the source processor P_0 are shown in Figure 4. Based on the *Send_Tables*, one can pack array elements in source local arrays to messages.

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

$$SP_{(\ell)} = Source\_Processors(DLA^{(n)}_{j,\ell}[1{:}\,L_\ell]) = \begin{bmatrix} G(1) \\ G(2) \\ \vdots \\ G(L_\ell) \end{bmatrix}_{L_\ell \times 1} \tag{8}$$

where $\ell = 1$ to $n-1$. The function $G(x)$ is defined in Equation 6.

For a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution, from Equation 8, we can obtain $SP_{(0)}, SP_{(1)}, \ldots, SP_{(n-1)}$ that represent the source processors

Send_Tables ST_0	*DP*	*Indices set*	ST_1	*DP*	*Indices set*
	$\tilde{p}_{0,0}$	1,2,6,7,8,12		$\tilde{p}_{0,0}$	1,2,5,6
	$\tilde{p}_{1,0}$	3,4,5,9,10,11		$\tilde{p}_{0,1}$	3,7
				$\tilde{p}_{0,2}$	4,8

Figure 4. The *Send_Tables* for $SLA^{(2)}_{0,0}[1{:}\,L_0]$ and $SLA^{(2)}_{0,1}[1{:}\,L_1]$.

of $SLA_{i,0}^{(n)}[1{:}L_0], SLA_{i,1}^{(n)}[1{:}L_1], \ldots, SLA_{i,n-1}^{(n)}[1{:}L_{n-1}]$, respectively. Since the source processors of $DLA_{j,0}^{(n)}[1{:}L_0], DLA_{j,1}^{(n)}[1{:}L_1], \ldots, DLA_{j,n-1}^{(n)}[1{:}L_{n-1}]$ are known, we can construct the *Receive_Tables* for $DLA_{j,0}^{(n)}[1{:}L_0], DLA_{j,1}^{(n)}[1{:}L_1], \ldots, DLA_{j,n-1}^{(n)}[1{:}L_{n-1}]$. Based on the *Receive_Tables*, we can unpack array elements in the received messages to their appropriate local array positions.

4.3. 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 costs. Therefore, there is a performance tradeoff between the indexing and packing/unpacking overheads of these two techniques. In this section, we derive a theoretical model to analyze the tradeoff between these two methods.

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

$$T_{comp} = T_{indexing} + T_{(un)packing} \tag{9}$$

where $T_{indexing}$ is the time for an algorithm to compute the source/destination processors of local array elements. $T_{(un)packing}$ is the time to pack/unpack array elements. We said that $T_{indexing}$ and $T_{(un)packing}$ is the indexing and packing/unpacking time of an algorithm to perform a redistribution, respectively. In the following discussion, since the sending phase and the receiving phase have the same time complexity, we only construct a model for the send phase. We will first construct a model for two-dimensional array redistribution, then, extend the model to multi-dimensional array redistribution.

Given a $\texttt{BC}(s_0, s_1) \rightarrow \texttt{BC}(t_0, t_1)$ 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 and the complete-dimension calculation can be modeled as follows,

$$T_{indexing}(BBC) = O(BC_0) + O(BC_1) \tag{10}$$

$$T_{indexing}(CDC) = O(L_0) + O(L_1) \tag{11}$$

where BC_k is the size of basic-cycle in each dimension; L_k is the local array size in each dimension, $L_k = (n_k/m_k), k = 0, 1$.

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. Assume that array elements were packed in a row-major manner. The time for a processor to pack array elements to messages in each row is $O(L_1/BC_1)$,

where L_1/BC_1 is the number of basic-cycles in dimension 1. Since there are L_0 rows in a local array, the time for a processor to pack array elements in dimension 1 to messages is $O((L_0 \times L_1)/BC_1)$. 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 0 to messages is $O(L_0/BC_0)$. Therefore, the time for a processor to pack array elements to messages can be modeled as follows,

$$T_{(un)packing}(BBC) = O\left(\frac{L_0 \times L_1}{BC_1}\right) + O\left(\frac{L_0}{BC_0}\right) \tag{12}$$

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 9 to 12, 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.

$$\begin{aligned} T_{comp}(CDC) < T_{comp}(BBC) &\Leftrightarrow O(L_0 + L_1) \\ &< O\left(BC_0 + BC_1 + \frac{L_0 \times L_1}{BC_1} + \frac{L_0}{BC_0}\right) \end{aligned} \tag{13}$$

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

$$T_{comp}(CDC) < T_{comp}(BBC) \Leftrightarrow O(L_0 + L_1) < O\left(\frac{L_0 \times L_1}{BC_1}\right) \tag{14}$$

Given a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1}) \rightarrow \texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ redistribution on an n-dimensional array $A^{(n)} = A[1{:}n_0, 1{:}n_1, \ldots, 1{:}n_{n-1}]$ over $M^{(n)} = M[m_0, m_1, \ldots, m_{n-1}]$, according to Equation 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.

$$\begin{aligned} T_{comp}(CDC) < T_{comp}(BBC) &\Leftrightarrow O(L_0 + L_1 + \cdots + L_{n-1}) \\ &< O\left(\frac{L_{n-1}}{BC_{n-1}} \times L_{n-2} \times \cdots \times L_0\right) \end{aligned} \tag{15}$$

From Equation 15, we can evaluate the tradeoff between the indexing and the packing/unpacking overheads. The performance of the basic-block calculation and the complete-dimension calculation techniques can be also predicted by Equation 15.

5. Performance evaluation and 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 [23] and the *Prylli's* method [21] 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 three types.

- *Dimension shift redistribution:*

  Ex: BC(x, y) to BC(y, x) of two-dimensional arrays, and BC(x, y, z) to BC(y, z, x) of three-dimensional arrays, where x, y and z are positive integers.

- *Refinement redistribution:*

  Ex: BC(x, y) to BC$(x/p, y/q)$ of two-dimensional arrays, and BC(x, y, z) to BC$(x/p, y/q, z/r)$ 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 array 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.

Table 1. The time (ms) 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

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 receiving 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 15. 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 15. 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 $T_{exec}(CDC) \approx T_{exec}(BBC) \ll T_{exec}(Prylli's) < T_{exec}(PITFALLS)$. 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

Table 2. The time (ms) 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

Table 3. The time (ms) 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

Table 4. The time (ms) 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

Table 5. The time (ms) 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

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.

6. Conclusions and future work

In many scientific applications, array redistribution is usually required to enhance data locality and reduce remote memory access on distributed memory multicomputers. Since the redistribution is performed at run-time, there is a performance tradeoff between the efficiency of the new data decomposition for a subsequent phase of an algorithm and the cost of redistributing data among processors. 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 a $\texttt{BC}(s_0, s_1, \ldots, s_{n-1})$ to $\texttt{BC}(t_0, t_1, \ldots, t_{n-1})$ 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.

Our techniques can only handle dense arrays and In-core programs. There are some possible extensions could be made. One of the issues would be to consider out-of-core external array redistribution. Another important future research direction would be to investigate the redistribution techniques in irregular scientific computation programs. it would also be interesting to consider the array redistribution of sparse arrays.

Acknowledgments

The work of this paper was partially supported by NSC of R.O.C. under contract NSC-87-2213-E035-011.

References

1. S. Benkner. Handling block-cyclic distribution arrays in Vienna Fortran 90. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, pp. 224–253, June 1995.
2. B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic data distribution in Vienna Fortran. In *Proceedings of Supercomputing '93*, pp. 284–293, November 1993.
3. 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. *Journal of Parallel and Distributed Computing*, 26:72–84, 1995.

4. Y.-C. Chung, C.-H. Hsu, and S.-W. Bai. A basic-cycle calculation technique for efficient dynamic data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):359–377, April 1998.
5. F. Desprez, J. Dongarra, and A. Petitet. Scheduling block-cyclic array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):192–205, February 1998.
6. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran-D language specification. Technical Report TR-91-170, Department of Computer Science, Rice University, December 1991.
7. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On the generation of efficient data communication for distributed-memory machines. *Proceedings of International Computing Symposium*, pp. 504–513, 1992.
8. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32:155–172, 1996.
9. High Performance Fortran Forum. High performance Fortran language specification, version 1.1, Rice University, November 1994.
10. S. Hiranandani, K. Kennedy, J. Mellor-Crammey, and A. Sethi. Compilation technique for block-cyclic distribution. In *Proceedings of the ACM International Conference on Supercomputing*, pp. 392–403, July 1994.
11. C.-H. Hsu and Y.-C. Chung. Efficient methods for $kr \rightarrow r$ and $r \rightarrow kr$ array redistribution. *The Journal of Supercomputing*, 12(2):253–276, May 1998.
12. E. T. Kalns and L. M. Ni. Processor mapping technique toward efficient data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1234–1247, December 1995.
13. L. M. Ni, H. Xu, and E. T. Kalns. Issues in scalable library design for massively parallel computers. In *Supercomputing '93*, pp. 181–190, November 1993.
14. S. D. Kaushik, C. H. Huang, R. W. Johnson, and P. Sadayappan. An approach to communication efficient data redistribution. In *Proceedings of the International Conference on Supercomputing*, pp. 364–373, July 1994.
15. S. D. Kaushik, C. H. Huang, J. Ramanujam, and P. Sadayappan. Multiphase array redistribution: modeling and evaluation. In *Proceedings of the International Parallel Processing Symposium*, pp. 441–445, 1995.
16. S. D. Kaushik, C. H. Huang, and P. Sadayappan. Efficient index set generation for compiling HPF array statements on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 38:237–247, 1996.
17. K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distribution. In *Proceedings of the International Conference on Supercomputing*, Barcelona, pp. 180–184, July 1995.
18. P.-Z. Lee and W. Y. Chen. Compiler techniques for determining data distribution and generating communication sets on distributed-memory multicomputers. In *29th IEEE Hawaii International Conference on System Sciences*, Maui, Hawaii, pp. 537–546, January 1996.
19. Y. W. Lim, P. B. Bhat, and V. K. Prasanna. Efficient algorithms for block-cyclic redistribution of arrays. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pp. 74–83, 1996.
20. Y. W. Lim, N. Park, and V. K. Prasanna. Efficient algorithms for multi-dimensional block-cyclic redistribution of arrays. In *Proceedings of the 26th International Conference on Parallel Processing*, pp. 234–241, 1997.
21. L. Prylli and B. Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *Journal of Parallel and Distributed Computing*, 45:63–72, August 1997.
22. S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Frontier '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean, Va., pp. 342–349, February 1995.
23. S. Ramaswamy, B. Simons, and P. Banerjee. Optimization for efficient array redistribution on distributed memory multicomputers. *Journal of Parallel and Distributed Computing*, 38:217–228, 1996.
24. J. M. Stichnoth, D. O'Hallaron, and T. R. Gross. Generating communication for array statements: design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21:150–159, 1994.

25. R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pp. 309–316, May 1994.
26. R. Thakur, A. Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587–594, June 1996.
27. A. Thirumalai and J. Ramanujam. Efficient computation of address sequence in data parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing*, 38:188–203, 1996.
28. V. Van Dongen, C. Bonello, and C. Freehill. High performance C—language specification version 0.8.9. Technical Report CRIM-EPPP-94/04-12, 1994.
29. C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, Philadelphia, Pa., 1992.
30. D. W. Walker and S. W. Otto. Redistribution of `BLOCK-CYCLIC` data distributions using MPI. *Concurrency: Practice and Experience*, 8(9):707–728, November 1996.
31. A. Wakatani and M. Wolfe. A new approach to array redistribution: strip mining redistribution. In *Proceeding of Parallel Architectures and Languages Europe*, July 1994.
32. A. Wakatani and M. Wolfe. Optimization of array redistribution for distributed memory multicomputers (short communication). *Parallel Computing*, 21(9):1485–1490, September 1995.
33. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran—a language specification version 1.1. ICASE Interim Report 21, ICASE NASA Langley Research Center, Hampton, Va., March 1992.