Journal of Parallel and Distributed Computing **60**, 189–216 (2000)
doi:10.1006/jpdc.1999.1601, available online at http://www.idealibrary.com on IDEAL®

Efficient Index Generation for Compiling Two-Level Mappings in Data-Parallel Programs[1]

Kuei-Ping Shih and Jang-Ping Sheu

Department of Computer Science and Information Engineering, *National Central University*, *Chung-Li 32054*, *Taiwan*
E-mail: sheujp@csie.ncu.edu.tw

Chua-Huang Huang

Department of Computer Science and Information Engineering, *National Dong-Hua University*, *Hualien*, *Taiwan*

and

Chih-Yung Chang

Department of Computer Information Science, *Tamsui Oxford University College*, *Tamsui*, *Taipei*, *Taiwan*

Received October 16, 1998; revised August 30, 1999; accepted September 30, 1999

This paper presents compilation techniques used to compress holes, which are caused by the nonunit alignment stride in a two-level data-processor mapping. Holes are the memory locations mapped by useless template cells. To fully utilize the memory space, memory holes should be removed. In a two-level data-processor mapping, there is a repetitive pattern for array elements mapped onto processors. We classify blocks into classes and use a *class table* to record the distribution of each class in the first repetitive data distribution pattern. Similarly, data distribution on a processor also has a repetitive pattern. We use a *compression table* to record the distribution of each block in the first repetitive data distribution pattern on a processor. By using a *class table* and a *compression table*, hole compression can be easily and efficiently achieved. Compressing holes can save memory usage, improve spatial locality and further improve system performance. The proposed method is efficient, stable, and easy to implement. The experimental results do confirm the advantages of our proposed method over existing methods.

Key Words: communication set; distributed-memory multicomputers; high performance Fortran; hole compression; two-level data-processor mapping.

[1] A preliminary version of the paper appeared in "LCPC '97" [22].

0743-7315/00 $35.00

1. INTRODUCTION

Distributed-memory multicomputers are superior to shared-memory multiprocessors in cost and scalability. These advantages have led to extensive use of distributed-memory multicomputers in scientific and engineering computation. However, the absence of a global shared address makes it difficult to program distributed-memory multicomputers. Programmers must not only distribute the computation and data onto processors, but also manage communication among processors. Currently, parallelizing compilers try to fill the gap between users and machines in order to relieve the burden of such tedious and error-prone work from users. Many researchers have investigated the extension of existing languages to provide users with more convenient ways to manage data distribution, such as using Fortran D [6, 26], High Performance Fortran (HPF) [8, 16], and Vienna Fortran [2, 3]. The major characteristic of these languages is that they provide users with a global addressing space and directives to specify data distribution at the language level.

Generally speaking, data parallel languages support two-level data-processor mapping. A two-level data-processor mapping enables the user to specify data-processor mapping by *aligning* related array objects with a template, an abstract index space, and then *distributing* the template onto the user-declared abstract processors. In the distribution phase, three regular data distributions, *block*, *cyclic*, and *block-cyclic* data distributions, are provided by these languages. Distributing array elements contiguously and evenly onto processors is the process of *block* distribution. *Cyclic* distribution distributes each array element onto processors one at a time and in a round-robin fashion. The distribution in which blocks of contiguous array elements of size x are distributed onto processors in a round-robin fashion is *block-cyclic* distribution and is denoted as *cyclic*(x). *Block-cyclic* distribution is known to be the most general data distribution. The *block* and *cyclic* distributions can be, respectively, represented by *block-cyclic* distribution as *cyclic*($\lceil N_A/P \rceil$) and *cyclic*(1), where N_A is the number of array elements and P is the number of processors.

For completeness, this paper considers a two-level data-processor mapping in which an array object is aligned with a template and the template is *block-cyclic* distributed onto processors. The program model considered in this paper is shown in Fig. 1. Figure 2 shows an example of a two-level data-processor mapping, which assumes that array $A(i)$ is aligned with a template T at $(3 * i + 1)$, and that the template is then distributed onto 4 processors with a *cyclic*(5) distribution. The white squares represent the array elements of A, and the number in the square is the global index of that array element. The gradations of the gray squares represent different template cells on different processors, and the number in the square is the

```
!HPF$ PROCESSORS PROC(P)
!HPF$ ALIGN A(i) WITH T(s * i + o)
!HPF$ DISTRIBUTE T(cyclic(x)) ONTO PROC
...
```

FIG. 1. HPF-like program model considered in this paper.

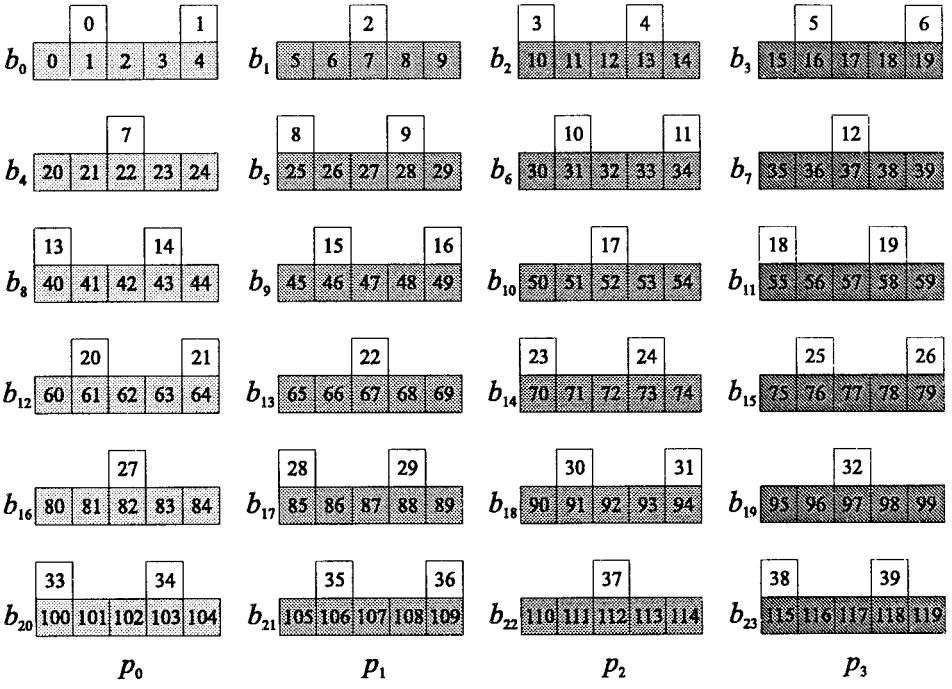
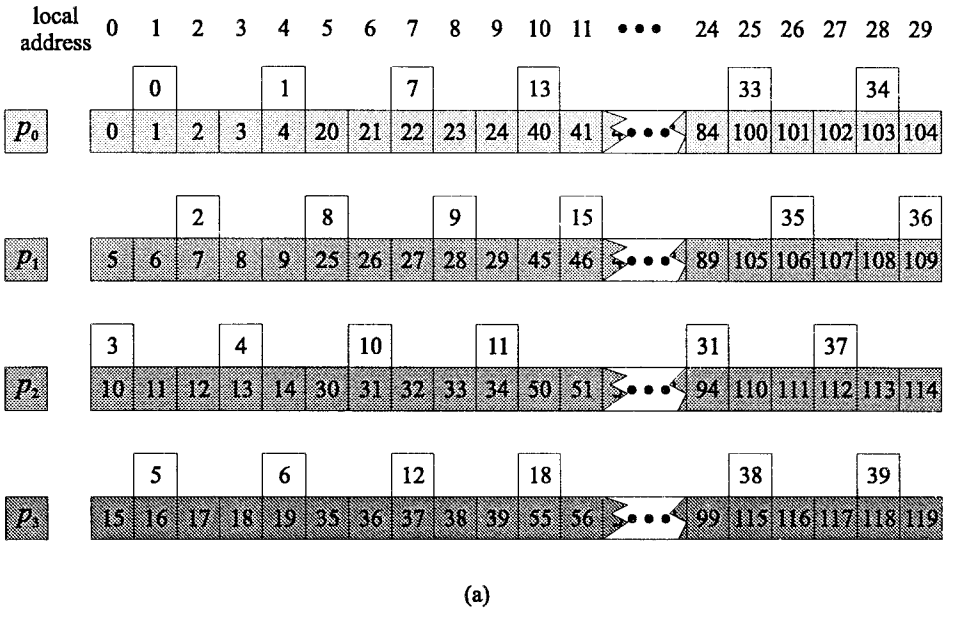


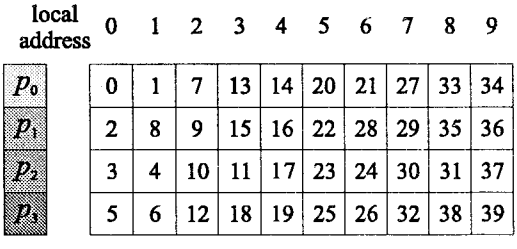
FIG. 2. Two-level data-processor mapping. Array $A(i)$ is aligned with template $T(3 * i + 1)$, and the template is then distributed onto 4 processors with a *cyclic*(5) distribution.

global index of that template cell. Basically, a template is an abstract index space whose elements have no content and occupy no storage [8]. Therefore, in a two-level data-processor mapping, if the alignment stride is nonunit, the mapping will lead to many *holes*. That is, there will be many template cells with which no array element is aligned. These holes imply that the memory locations will not be used. As a result, to fully utilize the memory space, only the template cells aligned by array elements need to be mapped to memory locations, and all the holes should be removed.

Memory holes caused by the nonunit alignment stride will result in a large amount of memory wastage, even for a small alignment stride. Suppose that the number of template cells is N_T, and that s is the alignment stride. Accordingly, only N_T/s template cells are aligned by array elements. The percentage of memory usage is $((N_T/s)/N_T) \times 100\%$, which equals $1/s \times 100\%$. In other words, the percentage of memory wastage is $((s-1)/s) \times 100\%$. The larger the alignment stride, the more memory is wasted. The least positive nonunit alignment stride of 2 still wastes 50% of the memory space. For the example shown in Fig. 2, Fig. 3 shows the distributions of data elements onto processors without and with hole compression. Figure 3a illustrates the distribution of data elements onto processors without hole compression. Meanwhile, Fig. 3b illustrates that with hole compression. Obviously, each processor should allocate 30 memory spaces if hole compression is not performed. However, among these spaces, only a few template cells are aligned by array elements. The rest of the template cells, which are aligned with no array elements, cause *holes* in the memory space. As a result, 10 memory spaces are enough for



(a)



(b)

FIG. 3. The distributions of data elements onto processors without and with hole compression: (a) without hole compression, (b) with hole compression.

each processor after hole compression is performed. Therefore, compressing holes is quite necessary and important for compiling a two-level data-processor mapping in data-parallel programs. The process that maps the aligned template cells to processors and eliminate unused holes is called *hole compression*. In addition to increasing memory usage, removing *holes* can also improve spatial locality and, furthermore, achieve higher performance.

This paper presents compilation techniques used to efficiently remove holes for compiling a two-level data-processor mapping in data-parallel programs. Observing the two-level mapping shown in Fig. 2, one can find that the distribution patterns for every three blocks are identical. Based on this observation, we can classify all blocks into classes. We design a table, called a *class table*, to record the distribution in each class for the first repetitive data distribution pattern. On the other hand, from the processor's viewpoint, the above observation is true as well. Therefore, another table called a *compression table* is established to record the distribution of

each block in the first repetitive data distribution pattern on a processor. The *compression table* can be established based on the *class table*. We design systematic methods to construct these tables, and the time complexity of each construction is $O(s)$ in the worst case, where s is the alignment stride. Hole compression can be easily achieved by using a *compression table*. The proposed approach can eliminate a large number of redundant computations since the computations for the repetitive patterns can be obtained by means of table lookup instead of recomputation. Experimental results verify the advantages of the proposed approach. Moreover, the proposed approach has high stability against existing methods. The execution time does not vary much with the alignment stride and/or the distribution block size. In addition, the proposed approach can be easily implemented. The technique presented here could be integrated into the existing address generation algorithms or communication set generation algorithms which either run in compiler time or run time under the owner computes rule policy.

This paper is organized as follows. In compiling a two-level data-processor mapping, *class table* is used to record the distribution of the first repetitive data distribution pattern in the two-level mapping. Therefore, the structure and characteristics of the *class table* are presented in Section 2. Section 3 introduces how the *class table* can be used to construct the *compression table*. Based on the *compression table*, the process of compressing holes and generating the compressed local array are also described. Experimental results to verify the advantages of our method over existing methods are provided in Section 4. Section 5 discusses related work. Section 6 concludes the paper and points out a possible direction for future research.

2. CHARACTERISTICS OF CLASS TABLE

For the purpose of compiling an array statement in a one-level data-processor mapping, we have designed a useful structure to summarize the characteristics of access patterns [29]. Based on a similar concept, a structure called a *class table* is designed in this paper to record the distribution of the first repetitive data distribution pattern in a two-level data-processor mapping. In this section, we briefly describe the basic components of a *class table* and how a *class table* is constructed. Without loss of generality, we assume that every numbering system starts from zero, such as numbered array elements, template cells, and processors, except where otherwise noted.

Suppose the array element $A(i)$ is aligned with $T(s * i + o)$, where s is the *alignment stride* and o is the *alignment offset*. In order to make the *class table* reusable, we define a term called *isomorphic alignment*. All isomorphic alignments can use the same *class table*. The definition of isomorphic alignments is stated as follows.

DEFINITION 1 (Isomorphic Alignments). Suppose $A(i)$ is aligned with $T(s_1 * i + o_1)$ and $A'(i)$ is aligned with $T(s_2 * i + o_2)$. The two alignments are *isomorphic* if and only if

- $s_1 = s_2$,
- $o_1 \equiv o_2 \pmod{s}$,

where $s = s_1 = s_2$.

Consequently, for an alignment in which array $A(i)$ is aligned with $T(s * i + o)$, we use the isomorphic alignment offset r to construct the *class table*, where $r = (o \bmod s)$. A template cell t is aligned by an array element if and only if $t \equiv r \pmod{s}$. For example, suppose array $A(i)$ is aligned with a template T at $(3i + 10)$. The alignment is illustrated in Fig. 4. We construct a *class table* by using the isomorphic alignment offset 1 instead of the original alignment offset 10, where $1 = (10 \bmod 3)$. For those template cells where $t \equiv r \pmod{s}$, there must exist array elements aligned with them. A template cell is *active* if it is aligned with an array element and its index is within the range $o \leqslant t \leqslant s * (N_A - 1) + o$, where N_A is the number of array elements. Otherwise, it is termed *pseudo active*. For this example, the active template cells start from 10, each then strides 3 and the template cells 1, 4, 7 are pseudo active elements. Figure 4 also illustrates the active elements and pseudo active elements for this example. The numbers in boldface are active elements, and the numbers in italics are pseudo active elements. Note that the pseudo active elements are viewed as being the same as active elements when we generate a *class table* and a *compression table*. However, the pseudo active elements will not be counted when we generate the compressed local array.

For a *cyclic*(x) distribution, every x template cells form a block. A block is numbered according to the occurrence of the block in the data-processor mapping. Suppose the number of array elements and template cells is N_A and N_T, respectively. Let N_b be the number of blocks. Thus, $N_b = \lceil N_T / x \rceil$. Those blocks within which active elements have the same positions are classified into the same class. For example, consider the two-level data-processor mapping shown in Fig. 2. Blocks 0, 3, 6, 9, ... have active elements with the same positions. These blocks are classified into the same class. Similarly, blocks 1, 4, 7, 10, ... can be classified into the same class. The following theorem demonstrates that, for a two-level data-processor mapping, according to the positions of the active elements within the blocks, all the blocks can be classified into different classes. Moreover, the number of classes is equal to $s/\gcd(s, x)$, where $\gcd(a, b)$ is the greatest common divisor of a and b.

THEOREM 1. *For any two-level data-processor mapping in which array A is aligned with T at a stride s and an offset o and template T is distributed onto processors using cyclic*(x) *distribution, all the template blocks can be classified into* $s/\gcd(s, x)$ *classes.*

Proof. Since each block is of the same size and the alignment stride is s, if the position of the first active element for some block is the same as that of another

A		√			√			√			0			1			2		···
T	0	*1*	2	3	*4*	5	6	*7*	8	9	**10**	11	12	**13**	14	15	**16**	17	···

FIG. 4. The *active* elements and *pseudo active* elements. The numbers in boldface are *active* elements, and the numbers in italics are *pseudo active* elements.

one, the positions of the rest of the active elements within the two blocks are the same. Suppose the position of the first active element within block b is δ, $0 \leqslant \delta < s$. We would like to show that the position of the first active element is still δ for every period of $s/\gcd(s, x)$ blocks.

The position in a block for any active element before or after δ can be formulated as $((\delta + k \cdot s) \bmod x)$, where $k \in \mathbf{Z}$ and $\mathbf{Z}$ is the set of integers. We claim that $(\delta + k' \cdot s) \bmod x = \delta$, for some k'. Therefore, the number of blocks between b and the block in which the position of the first active element is the same with b is $(k' \cdot s)/x$. Since $(\delta + k' \cdot s) \bmod x = \delta$, it follows that $\delta + k' \cdot s = x \cdot q + \delta$, for some $q \in \mathbf{Z}$. Thus, $k' \cdot s = x \cdot q$. Because k' and q are integers, the unique solution to $k' \cdot s = x \cdot q$ is

$$k' = t \cdot \frac{\text{lcm}(s, x)}{s} \qquad \text{and} \qquad q = t \cdot \frac{\text{lcm}(s, x)}{x},$$

where $t \in \mathbf{Z}$ and $\text{lcm}(a, b)$ is the least common multiple of a and b. Therefore

$$\frac{k' \cdot s}{x} = q = t \cdot \frac{\text{lcm}(s, x)}{x} = t \cdot \frac{s}{\gcd(s, x)}.$$

In other words, for every period of $s/\gcd(s, x)$ blocks, the first active elements of these blocks are the same. Thus, for arbitrary two-level data-processor mapping, blocks can be classified into $s/\gcd(s, x)$ classes. The theorem is, therefore, obtained. ▮

Let N_c be the number of classes. By Theorem 1, $N_c = s/\gcd(s, x)$. Since all blocks are classified into N_c classes, we arrange the class numbers of the block in lexicographical order and in a round-robin fashion. Formally, the class number of block b is $(b \bmod N_c)$. Furthermore, blocks b_1 and b_2 belong to the same class if and only if $b_1 \equiv b_2 \pmod{N_c}$. Accordingly, blocks $\{b, (b+1), (b+2), ..., (b + N_c - 1) \mid b \equiv 0 \pmod{N_c}\}$ are a period in terms of classes, and we term such a period a *class cycle*. A *class table* is used to record the first, the last and the number of active elements on a class for each class in a class cycle. For a class c, $\mathscr{A}_f(c)$ represents the order of occurrence in a class cycle for the *first* active element in c, $\mathscr{A}_l(c)$ the order of occurrence in a class cycle for the *last* active element in c, and $\mathscr{A}_n(c)$ the number of active elements within c. In addition to the characteristics described above, a *class table* also contains implicitly a lot of additional information implicitly. For a class c, $\mathscr{A}_f(c)$ also indicates the number of active elements contained from class 0 to class $(c-1)$, and $(\mathscr{A}_l(c)+1)$ indicates the number of active elements contained from 0 up to class c. Therefore, the number of active elements contained in a class cycle is equal to $\mathscr{A}_l(N_c - 1) + 1$. Let $\mathscr{A}_c$ be the number of active elements contained in a class cycle. Thus, $\mathscr{A}_c = \mathscr{A}_l(N_c - 1) + 1$.

Figure 5 shows the *class table* for the example shown in Fig. 2. In this example, blocks can be classified into 3 classes. The blocks 0, 3, 6, 9, ..., belong to class 0, the blocks 1, 4, 7, 10, ..., belong to class 1, and the blocks 2, 5, 8, 11, ..., belong to class 2. Blocks 0, 1, 2 form a class cycle, blocks 3, 4, 5 form another class cycle, and

c	$\mathscr{A}_f$	$\mathscr{A}_l$	$\mathscr{A}_n$
0	0	1	2
1	2	2	1
2	3	4	2

FIG. 5. Class table for the example shown in Fig. 2.

so on. For the first class in a class cycle, the number of occurrences of the first and the last active elements is 0 and 1, respectively, and the number of active elements in this class is 2. Thus, $\mathscr{A}_f(0)=0$, $\mathscr{A}_l(0)=1$, and $\mathscr{A}_n(0)=2$. Likewise, $\mathscr{A}_f(1)=2$, $\mathscr{A}_l(1)=2$, and $\mathscr{A}_n(1)=1$ for class 1, and $\mathscr{A}_f(2)=3$, $\mathscr{A}_l(2)=4$, and $\mathscr{A}_n(2)=2$ for class 2. Hence, the number of active elements within a class cycle, A_c, is $5(=\mathscr{A}_l(2)+1)$. Figure 6 shows the algorithm used to generate a *class table*, which is termed the Class_Table_Generation algorithm. The major factors affecting the construction of a *class table* are the alignment stride s, alignment offset o, and distribution block size x. The time complexity of the Class_Table_Generation algorithm is $O(s/\gcd(s, x))$.

Note that, if the block size x is larger than the alignment stride s, each block contains at least one active element. Thus, $\mathscr{A}_l(c)$ is always larger than or equal to $\mathscr{A}_f(c)$, where c is the class number of that block. However, if the block size is smaller than the alignment stride, each block contains at most one active element, and there are blocks that contain no active elements at all. For any block that contains no active element, $\mathscr{A}_f(c)$ will be one larger than $\mathscr{A}_l(c)$. Lemma 1 demonstrates this phenomenon.

LEMMA 1. *For any block b,*

$$\mathscr{A}_f(c)=\mathscr{A}_l(c)+1 \Leftrightarrow \mathscr{A}_n(c)=0,$$

where c is the class number of b.

Proof. ($\Rightarrow$): Since $\mathscr{A}_n(c)=\mathscr{A}_l(c)-\mathscr{A}_f(c)+1$ and $\mathscr{A}_f(c)=\mathscr{A}_l(c)+1$; obviously, $\mathscr{A}_n(c)=0$.

($\Leftarrow$): Since $\mathscr{A}_n(c)=\mathscr{A}_l(c)-\mathscr{A}_f(c)+1$ and $\mathscr{A}_n(c)=0$; clearly, $\mathscr{A}_f(c)=\mathscr{A}_l(c)+1$. ∎

In the following section, we will explain how the class table is used to construct a *compression table* to extract information from the first repetitive data distribution pattern on a processor. Next, we will describe how the compressed local array for

$N_c = s/\gcd(s,x)$
$r = o \mod s$
DO $c = 0$ **TO** $(N_c - 1)$
  $\mathscr{A}_f(c) = \lceil (c*x - r)/s \rceil$
  $\mathscr{A}_l(c) = \lfloor ((c+1)*x - r - 1)/s \rfloor$
  $\mathscr{A}_n(c) = \mathscr{A}_l(c) - \mathscr{A}_f(c) + 1$
ENDDO

FIG. 6. Class_Table_Generation algorithm.

a processor is generating by using the *compression table*. The transformations between global and local indices will be addressed as well.

3. HOLE COMPRESSION

A two-level data-processor mapping may cause many useless holes if the alignment stride is nonunit. Memory holes result in memory wastage and degrade system performance. Therefore, this section proposes compilation techniques to totally eliminate these useless holes and systematically generate the sequence of array elements exactly obtained by each processor. The transformations between the global and the local addresses are also described.

3.1. Construction of a Compression Table

Suppose a two-level data-processor mapping is of the form shown in Fig. 1. By Theorem 1, for an arbitrary two-level data-processor mapping, all the blocks can be classified into N_c classes, where $N_c = s/\gcd(s, x)$. Consequently, $\text{lcm}(N_c, P)$ blocks will form a data distribution pattern. Blocks within a data distribution pattern can be viewed as different but are identical for every different data distribution patterns. Hence, we only have to consider the first data distribution pattern, and the rest of data distribution patterns can be processed likewise. Therefore, we propose a *compression table* used to record information about the generation of a compressed local array for a processor on the first data distribution pattern. Similar to a *class table*, a *compression table* also regards an alignment offset o as the isomorphic alignment offset r, where $r = (o \bmod s)$.

Let the number of blocks in a data distribution pattern be N_{pb}. Thus, $N_{pb} = \text{lcm}(N_c, P)$. The number of blocks on each processor within a data distribution pattern is, thus, equal to N_{pb}/P, which can also be written as $N_c/\gcd(N_c, P)$. Let the number of blocks on a processor within a data distribution pattern be N_{pb}^p; then, $N_{pb}^p = N_c/\gcd(N_c, P)$. We number a block on a processor within a data distribution pattern according to the order of occurrence of that block on the processor. For the example shown in Fig. 2, a data distribution pattern contains $12 (= N_{pb})$ blocks. Blocks 0 to 11 form the first data distribution pattern, and blocks 12 to 23 are within the second data distribution pattern. Blocks 0 and 4 are the first and the second blocks occurring on processors 0, blocks 1 and 5 are the first and the second blocks occurring on processor 1 within the first data distribution pattern, blocks 12 and 13 are the first blocks occurring, respectively, on processor 0 and 1 within the second data distribution pattern, and so on. The representations of the notations *class*, *class cycle*, *data distribution pattern*, and *occurrence* of a block within a data distribution pattern are shown in Fig. 7.

As mentioned above, data distributions among different data distribution patterns are identical. The only difference between the first data distribution pattern and other data distribution patterns is the indices of every pair of corresponding cells. However, for every corresponding cell, the indices are different in only a fixed offset. Hence, for a processor, we only need to consider how to compress holes for the first data distribution pattern and for the rest of the data distribution patterns,

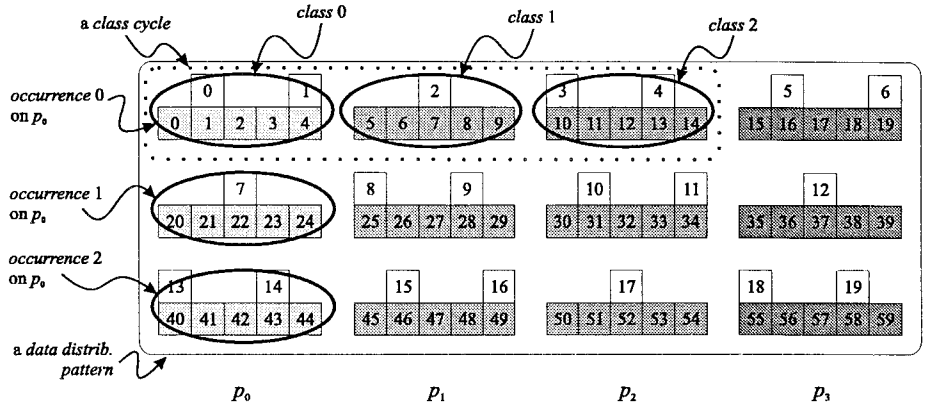


FIG. 7. The representations of the notations *class*, *class cycle*, *data distribution pattern*, and *occurrence* of a block within a data distribution pattern.

only the fixed offset needs to be evaluated. To facilitate hole compression, we design a structure, termed a *compression table*, used to characterize blocks in the first data distribution pattern on a processor. For a block of occurrence $\mathcal{O}$ on processor p, a *compression table* records $\mathscr{C}_g^p$, $\mathscr{C}_n^p$, and $\mathscr{C}_l^p$, where

- $\mathscr{C}_g^p(\mathcal{O})$ is the *global* index of the first array element in $\mathcal{O}$ on processor p,
- $\mathscr{C}_n^p(\mathcal{O})$ is the *number* of array elements in $\mathcal{O}$ on processor p, and
- $\mathscr{C}_l^p(\mathcal{O})$ is the *local* index in the compressed local array for the first array element in $\mathcal{O}$ on processor p.

Take the *compression table* of processor p_0 as an example. Block b_0 is the block of occurrence 0 within the first data distribution pattern on processor p_0. The global index of the first array element in b_0 is 0, the number of array elements in the block is 2, and the local index of array element $A(0)$ in the compressed local array is 0. Thus, $\mathscr{C}_g^0(0)=0$, $\mathscr{C}_n^0(0)=2$, and $\mathscr{C}_l^0(0)=0$. The second occurrence of a block within the first data distribution pattern on processor p_0 is b_4. The global index of the first array element in b_4 is 7, the number of array elements in the block is 1, and the local index of $A(7)$ in the compressed local array is 2. Therefore, $\mathscr{C}_g^0(1)=7$, $\mathscr{C}_n^0(1)=1$, and $\mathscr{C}_l^0(1)=2$. Similarly, $\mathscr{C}_g^0(2)=13$, $\mathscr{C}_n^0(2)=2$, and $\mathscr{C}_l^0(2)=3$. The *compression tables* for processor p_0 and p_1 in the example shown in Fig. 2 are illustrated in Figs. 8a and 8b, respectively.

$\mathcal{O}$	$\mathcal{C}_g^0$	$\mathcal{C}_n^0$	$\mathcal{C}_l^0$
0	0	2	0
1	7	1	2
2	13	2	3

(a)

$\mathcal{O}$	$\mathcal{C}_g^1$	$\mathcal{C}_n^1$	$\mathcal{C}_l^1$
0	2	1	0
1	8	2	1
2	15	2	3

(b)

FIG. 8. Compression tables for processor p_0 and p_1 in the example shown in Fig. 2. (a) The *compression table* for processor p_0. (b) The *compression table* for processor p_1.

```
b = p;  nelem = 0;  addr = 0;   /* initialization */
N_pb^p = N_c / gcd(N_c, P)
DO O = 0 TO (N_pb^p - 1)
    c = b  mod N_c   /* the class number of block b */
    C_g^p(O) = ⌊b/N_c⌋ * A_c + A_f(c)
    C_n^p(O) = A_n(c)
    C_l^p(O) = addr + nelem
    b = b + P   /* the next block on processor p */
    nelem = C_n^p(O) /* keep the current value for the next C_l^p */
    addr = C_l^p(O)   /* keep the current value for the next C_l^p */
ENDDO
```

FIG. 9. Compression_Table_Generation algorithm.

The construction of a *compression table* for processor p is described as follows. Since all the blocks are classified into N_c classes, every N_c blocks forms a class cycle. For any block b, there are $\lfloor b/N_c \rfloor$ class cycles which appear before b. Furthermore, each class cycle contains $\mathscr{A}_c$ active elements. Hence, there are $(\lfloor b/N_c \rfloor * \mathscr{A}_c)$ active elements in these class cycles. From Section 2, $\mathscr{A}_f(c)$ implicitly indicates the number of active elements from class 0 to class $(c-1)$ in a class cycle. As a result, for any block b, the global index of the first array element in the block can be obtained by $\lfloor b/N_c \rfloor * \mathscr{A}_c + \mathscr{A}_f(c)$, where c is the class number of b. Therefore, $\mathscr{C}_g^p$ can be obtained by $\lfloor b/N_c \rfloor * \mathscr{A}_c + \mathscr{A}_f(c)$. On the other hand, $\mathscr{C}_n^p$ can be obtained by $\mathscr{A}_n(c)$, and $\mathscr{C}_l^p$ can be evaluated by summing the last evaluated values of $\mathscr{C}_n^p$ and $\mathscr{C}_l^p$. Initially, $\mathscr{C}_l^p = 0$. Note that $\mathscr{C}_l^p$ can also represent the number of active elements occurring before the block in a data distribution pattern. The algorithm to generate *compression table* is shown in Fig. 9. The time complexity of the Compression_Table_Generation algorithm is $O(N_c/\gcd(N_c, P))$, where N_c is the number of classes and P is the number of processors. From Section 2, $N_c = s/\gcd(s, x)$. Hence, the worst case of the Compression_Table_Generation algorithm is $O(s)$, as much as that of Class_Table_Generation algorithm.

Note that if the alignment stride is smaller than or equal to the distribution block size ($s \leqslant x$), then each block contains at least one active element. In this case, the first array element in a block obtained by the above calculation is exactly the first array element on that block. On the other hand, if the alignment stride is larger than the distribution block size ($s > x$), then a block may contain no active element. The first array element of the empty block, a block which contains no active element, obtained by the above calculation is a *pseudo* array element. However, pseudo array elements do not affect the correctness of our proposed approach. We shall verify this in the next section.

3.2. Generation of Compressed Local Arrays

Using the *compression table* to generate the compressed local array on a processor is proposed in this subsection. Since the isomorphic alignments differ only in the alignment offset, to simplify our discussion, we will first consider a two-level

data-processor mapping without considering the pseudo active elements. The general case will be discussed after that.

3.2.1. *Special Case of a Two-Level Data-Processor Mapping*

Generally speaking, the main factors affecting a two-level data-processor mapping are the alignment stride s, alignment offset o, distribution block size x, and the number of processors P. However, by Definition 1, for the same s, x, and P, two different alignment offsets o_1 and o_2 lead to *isomorphic* data-processor mappings if and only if $o_1 \equiv o_2 \pmod{s}$. Hence, the constructions of *class table* and *compression table* use the isomorphic alignment offset r, where $r = (o \bmod s)$. In addition, to simplify our discussion, we neglect the pseudo active elements in this subsection. By doing so, we can avoid evaluating the pseudo active elements occurring in front and in back of the first and the last active elements. Figure 2 shows an example of such a two-level data-processor mapping.

As previously stated, the data distributions among different data distribution patterns are identical except for the indices of the corresponding elements. Let the difference between the global indices of two corresponding array elements on two contiguous data distribution patterns be the *global indexing offset*. Therefore, if the global index of an array element is known, the corresponding array element on the previous(next) data distribution pattern can be obtained by subtracting(adding) the *global indexing offset* from(to) the global index of that array element. Hence, the simplest approach to generating the compressed local array for processor p is to first generate the compressed local array for the first data distribution pattern according to the *compression table*. Then, for the following data distribution patterns, the compressed local array can be generated according to the previously generated compressed local array and the *global indexing offset*. Consider the two-level data-processor mapping shown in Fig. 2. Take the generation of the compressed local array for processor p_0 as an example. The compressed local array for the first data distribution pattern is $A(0, 1, 7, 13, 14)$. Accordingly, the compressed local array for the second data distribution pattern is $A(20, 21, 27, 33, 34)$ since the *global indexing offset* is 20. Thus, the compressed local array of processor p_0 is $A^{p_0}(0, 1, 7, 13, 14, 20, 21, 27, 33, 34)$. The Hole_Compression algorithm for the special case of two-level data-processor mapping is shown in Fig. 10.

Algorithm: Hole_Compression algorithm for the special case of two-level data-processor mapping.
Input: *Compression table* and *global indexing offset*.
Output: A^p, the compressed local array on processor p.
Procedure:
Step 1. Generate the compressed local array for the first data distribution pattern on processor p by using the *compression table*.
Step 2. Generate A^p for the rest of data distribution patterns according to the *global indexing offset*.

FIG. 10. Hole_Compression algorithm to generate the compressed local array for processor p in the special case of two-level data-processor mapping.

The algorithm shown in Fig. 10 is straightforward but less efficient since it incurs indirect memory accesses. Based on a similar idea, we propose a more efficient algorithm to generate the compressed local array for the special case of two-level data-processor mapping. Let the difference between the local indices in a compressed local array for two corresponding array elements on two contiguous data distribution patterns be the *local indexing offset*. As the differences between the global and the local indices of two corresponding array elements on two contiguous data distribution patterns are fixed, and are equal to the *global* and the *local indexing offsets*, respectively, the global indices of the array elements on the latter data distribution pattern can be obtained by adding the *global indexing offset* to the global indices of the corresponding array elements on the former data distribution pattern. Similarly, the local indices in the compressed local array for the compressed array elements on the latter data distribution pattern can be obtained by adding the *local indexing offset* to the local indices of the corresponding array elements on the former data distribution pattern. Therefore, once the *global* and *local indexing offsets* for two corresponding array elements on two contiguous data distribution patterns are determined, we can determine the global indices of the corresponding array elements on contiguous data distribution patterns and the local indices in the compressed local array for the corresponding array elements. The compressed local array for processor p can, thus, be generated as follows. For the blocks of the same occurrence in every data distribution pattern, the global index of the first array element in the first data distribution pattern can be obtained by the *compression table*, and for the array elements in the remaining data distribution pattern, the index can be obtained by adding the *global indexing offset* to the global index of the previous corresponding array element. Similarly, the local index in compressed local array of the first array element in the first data distribution pattern can also be obtained by the *compression table*. For the other corresponding array elements, the local indices can be obtained by adding the local indexing offset to the local index of the previous corresponding array element.

Again, take the generation of the compressed local array for processor p_0 in Fig. 2 as an example. The global index of the first array element in block b_0 in the first data distribution pattern is 0, which can be obtained by looking up the *compression table* of processor p_0. The local index of $A(0)$ is 0, which can also be obtained by the *compression table*. For the corresponding array elements in the second data distribution pattern in this example, the global index of that array element is 20, which is equal to $20+0$, where 20 is the *global indexing offset* and 0 is the global index of the previous corresponding array element. The local index of $A(20)$ is 5, which equals $5+0$, where 5 is the *local indexing offset* and 0 is the local index of the previous corresponding array element. That is, $A(20)$ is mapped to $A^{p_0}(5)$. Likewise, we generate the compressed local array elements according to the sequence $A(0)$, $A(20)$, $A(1)$, $A(21)$, ... As a result, we can also obtain the compressed local array $A^{p_0}=(0, 1, 7, 13, 14, 20, 21, 27, 33, 34)$ for processor p_0. Figure 11 illustrates the modified algorithm, which can generate the compressed local array for processor p in the special case of a two-level data-processor mapping more efficiently. In the algorithm, N_{ptn} is the number of data distribution patterns in the two-level mapping, $\mathscr{A}_{ptn}$ is the *global indexing offset*,

```
DO O = 0 TO (N_pb^p - 1)  /* iterate the number of occurrences in sequence */
    gl_ini = C_g^p(O)  /* initialize to the global index of the first array
                          element in the block of occurrence O */
    loc_ini = C_l^p(O)  /* initialize to the local index in the compressed local array
                           for the first array element in the block of occurrence O */
    DO elem = 0 TO (C_n^p(O) - 1)  /* iterate the number of array elements
                                      in the block of occurrence O */
        gl = gl_ini + elem  /* shift to the next element */
        loc = loc_ini + elem  /* shift to the next element */
        DO ptn = 1 TO N_ptn  /* iterate the number of data distribution
                                patterns in a sequence */
            A^p(loc) = A(gl)  /* generate the compressed local array element */
            gl = gl + A_ptn  /* add the global indexing offset */
            loc = loc + A_ptn^p  /* add the local indexing offset */
        ENDDO
    ENDDO
ENDDO
```

FIG. 11. The modified Hole_Compression algorithm for generating the compressed local array of processor p in the special case of a two-level data-processor mapping.

and $\mathscr{A}_{ptn}^{p}$ is the *local indexing offset*, which will be described in the following subsection.

Implementation issues. The main concept used to generate the compressed local array for the special case of a two-level data-processor mapping has been described. However, some implementation issues should be addressed. Since the data distributions among different data distribution patterns are identical except for the indices of the corresponding elements, the key to generating the compressed local array is to evaluate the *global* and *local indexing offset*. Evidently, the difference between the global indices of two corresponding array elements in two contiguous data distribution patterns is the number of active elements in a data distribution pattern. Let $\mathscr{A}_{ptn}$ be the number of active elements in a data distribution pattern. Thus, $\mathscr{A}_{ptn}$ is the *global indexing offset*. The number of active elements in a data distribution pattern can be evaluated by $\mathscr{A}_{ptn} = \lceil (N_{pb} * x)/s \rceil$, where N_{pb} is the number of blocks in a data distribution pattern. For instance, in the example shown in Fig. 2, $\mathscr{A}_{ptn} = 20$. The first blocks in the first and the second data distribution patterns are blocks 0 and 12, respectively. The first array elements in blocks 0 and 12 are $A(0)$ and $A(20)$, respectively. The difference between the global indices of the two corresponding array elements is 20, which is equal to $\mathscr{A}_{ptn}$.

The global indices of two corresponding array elements in two contiguous data distribution patterns have a fixed difference. Similarly, the local indices in a compressed local array for two corresponding array elements in two contiguous data distribution patterns also have a fixed difference. We term this fixed difference the *local indexing offset*. Clearly, the difference between the local indices in a compressed local array for two corresponding array elements in two contiguous data distribution patterns equals the number of active elements in a data distribution pattern allocated to that processor. Let $\mathscr{A}_{ptn}^{p}$ be the number of active elements

allocated to processor p in a data distribution pattern. Obviously, $\mathscr{A}^p_{ptn}$ is the *local indexing offset*. From the *compression table*,

$$\mathscr{A}^p_{ptn} = \mathscr{C}^p_n(N^p_{pb} - 1) + \mathscr{C}^p_l(N^p_{pb} - 1).$$

For processor p_0 in the example shown in Fig. 2, the number of active elements in a data distribution pattern in p_0 is 5. The first array elements in the first and the second data distribution patterns allocated to processor p_0 are 0 and 20, respectively. The local index of array element 0 in compressed local array is 0, and that for array element 20 is 5. The difference between the local indices in the compressed local array of p_0 for these two corresponding array elements is 5, which is equal to $\mathscr{A}^p_{ptn}$.

Suppose there are N_{ptn} data distribution patterns in the special case of a two-level data-processor mapping. The number of data distribution patterns N_{ptn} can be obtained by $N_{ptn} = N_b/N_{pb}$, where N_b is the number of blocks and N_{pb} is the number of blocks in a data distribution pattern. Since the array elements are distributed into multiple of data distribution patterns, the number of compressed local array elements on processor p can be figured out as follows. Let the number of compressed local array elements on processor p be N^p_A. Then,

$$N^p_A = N_{ptn} * \mathscr{A}^p_{ptn}.$$

In this subsection, the generation of a compressed local array for the special case of a two-level data-processor mapping has been described. The next subsection will discuss the generation of a compressed local array for a general two-level mapping, which takes the pseudo active elements into consideration.

3.2.2. General Case of a Two-Level Data-Processor Mapping

The concept used in the general case of a two-level data-processor mapping is similar to the special case of a two-level mapping. However, the pseudo active elements should be taken into consideration for the general case of a two-level data-processor mapping. For example, Fig. 12 shows a general two-level data-processor mapping. In this data-processor mapping, array A has 30 elements indexed from 0 to 29. Array element $A(i)$ is aligned with a template T with stride 3 and offset 28. Template T is distributed onto 4 processors using *cyclic*(5) distribution. The two data-processor mappings shown in Figs. 2 and 12 are isomorphic since the alignment strides, the distribution block sizes and the numbers of processors of the two mappings are equal, and the two alignment offsets $1 \equiv 28 \pmod 3$. The *class table* and *compression table* used by the data-processor mapping shown in Fig. 2 are the same as those for the mapping shown in Fig. 12. The compressed local array of processor p_0 in Fig. 2 is $A^{p_0} = (0, 1, 7, 13, 14, 20, 21, 27, 33, 34)$, which has been introduced in the previous subsection. However, in Fig. 12, taking the pseudo active elements into consideration, the compressed local array becomes $A^{p_0} = (4, 5, 11, 12, 18, 24, 25)$.

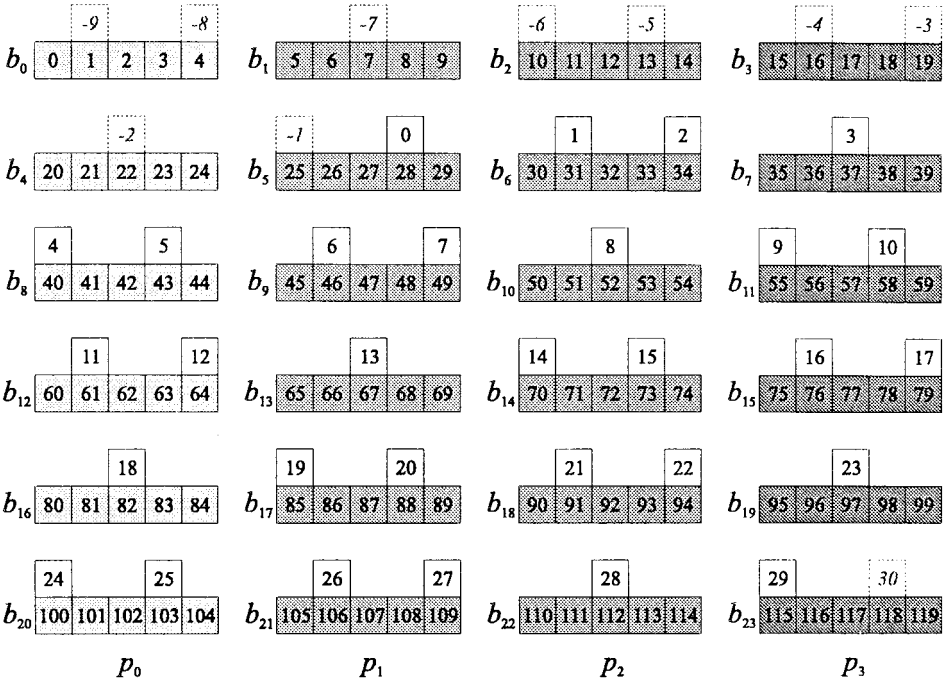


FIG. 12. A general two-level data-processor mapping. Array $A(i)$ is aligned with template $T(3 * i + 28)$, and the template is then distributed onto 4 processors with *cyclic*(5) distribution, assuming $N_A = 30$.

In the general case of two-level data-processor mapping, the first and the last data distribution patterns may have pseudo active elements. These two data distribution patterns should be considered separately. As for the rest of the data distribution patterns between the first and the last data distribution patterns, there is no pseudo active element in these data distribution patterns. Hence, the generation of a compressed local array for these data distribution patterns can adopt the concept used for the special case of a data-processor mapping. The evaluations of the *global* and *local indexing offsets* in the special case of a two-level data-processor mapping are also important for general two-level data-processor mapping, where the *global* and *local indexing offsets* are, respectively, the differences between the global indices and the local indices in a compressed local array for two corresponding array elements in two contiguous data distribution patterns. In addition to the evaluations of *global* and *local indexing offsets*, an evaluation of the number of pseudo active elements is required as well. We have to count the number of pseudo active elements mapped onto p before the first active element $T(o)$. Moreover, the number of pseudo active elements mapped onto p after the last active element is also determined. Here, determining the number of pseudo active elements in the last active block is enough.

Consider the generation of a compressed local array for processor p_0 in the general case of a two-level data-processor mapping shown in Fig. 12. Since the data-processor mapping shown in Fig. 12 is isomorphic to that shown in Fig. 2, the *compression tables* used by the two data-processor mappings are identical and are

shown in Fig. 8. The *global* and *local indexing offsets* are 20 and 5, respectively, which are the same as those used in the special case of a two-level data-processor mapping shown in Fig. 2. The compressed local array of processor p_0 without considering the pseudo active elements is $A^{p_0} = (0, 1, 7, 13, 14, 20, 21, 27, 33, 34)$, as discussed in Section 3.2.1. Consider the pseudo active elements. There are totally 9 pseudo active elements which appear before $T(o)$ in the two-level data-processor mapping, which are indexed as 1, 4, 7, 10, 13, 16, 19, 22, and 25. This implies that all the array elements are shifted 9 elements. That is, all the global indices of array elements have to subtract 9 from their original indices in the special case of a two-level data-processor mapping to get their real indices in the general case. Thus, the compressed local array of processor p_0 becomes $A^{p_0} = (-9, -8, -2, 4, 5, 11, 12, 18, 24, 25)$. In addition, there are three pseudo active elements on processor p_0 (i.e., $T(1, 4, 22)$). As a result, the previous three array elements should be eliminated. Actually, the compressed local array of processor p_0 for the general two-level data-processor mapping shown in Fig. 12 is $A^{p_0} = (4, 5, 11, 12, 18, 24, 25)$.

However, if the number of compressed local array elements on processor p is known in advance, the compressed local array of processor p can be generated more efficiently. Let the number of compressed local array elements on processor p be N_A^p. Once the number of pseudo active elements allocated on processor p is calculated, N_A^p can be obtained easily. Let $\mathscr{A}_{ptn}$ be the number of active elements in a data distribution pattern, and let $\mathscr{A}^p_{ptn}$ be the number of active elements allocated to processor p in a data distribution pattern. Obviously, $\mathscr{A}_{ptn}$ and $\mathscr{A}^p_{ptn}$ are the *global* and the *local indexing offsets*, respectively. As mentioned above, if an array element $A(gl)$ is mapped to the compressed local array at $A^p(loc)$, the array element $A(gl + \mathscr{A}_{ptn})$ will be mapped to the compressed local array at $A^p(loc + \mathscr{A}^p_{ptn})$. Thus, we can generate the compressed local array for the first $\mathscr{A}^p_{ptn}$ elements by using the *compression table*. After that, we can generate the next $\mathscr{A}^p_{ptn}$ elements according to the previous $\mathscr{A}^p_{ptn}$ elements and the *global indexing offset* $\mathscr{A}_{ptn}$. Finally, we can generate the last (N_A^p mod $\mathscr{A}^p_{ptn}$) elements accordingly. The

Algorithm: Hole_Compression algorithm for a general two-level data-processor mapping.
Input: N_A^p, *compression table*, *global* and *local indexing offsets*.
Output: A^p, the compressed local array on processor p.
Procedure:
Step 1. Generate the first $\mathcal{A}^p_{ptn}$ compressed local array elements according to the *compression table* and the number of pseudo active elements.
Step 2. Generate the remaining $(\lfloor N_A^p / \mathcal{A}^p_{ptn} \rfloor - 1)$ $\mathcal{A}^p_{ptn}$ compressed local array elements. Every $\mathcal{A}^p_{ptn}$ elements are obtained by adding the *global indexing offset* to the previous $\mathcal{A}^p_{ptn}$ elements.
Step 3. Generate the last (N_A^p mod $\mathcal{A}^p_{ptn}$) compressed local array elements according to the previous $\mathcal{A}^p_{ptn}$ compressed local array elements and the *global indexing offset*.

FIG. 13. Hole_Compression algorithm for generating the compressed local array of processor p in a general two-level data-processor mapping.

algorithm used to generate the compressed local array for the general case of a two-level data-processor mapping is shown in Fig. 13.

Implementation issues. For a two-level data-processor mapping, we only need to be concerned about the blocks which contain active elements. Hence, a block which contains active elements is termed an *active block*. Let b_f^p and b_l^p, respectively, denote the first and the last active blocks contained by processor p. Thus, b_f^p and b_l^p can be determined as follows. The block number of a block contained by processor p can be written in the form $(p + k * P)$, where $k \in \mathbf{Z}^+$ and $\mathbf{Z}^+$ is the set of positive integers. Since the first and the last active elements are $T(o)$ and $T(s * (N_A - 1) + o)$, they are allocated to blocks $\lfloor o/x \rfloor$ and $\lfloor (s * (N_A - 1) + o)/x \rfloor$, respectively. Therefore, b_f^p should be greater than or equal to $\lfloor o/x \rfloor$ and b_l^p should be smaller than or equal to $\lfloor (s * (N_A - 1) + o)/x \rfloor$. Suppose $b_f^p = (p + k_f * P)$ and $b_l^p = (p + k_l * P)$, for some k_f and $k_l \in \mathbf{Z}^+$. Thus, $(p + k_f * P) \geqslant \lfloor o/x \rfloor$ and $(p + k_l * P) \leqslant \lfloor (s * (N_A - 1) + o)/x \rfloor$. We can obtain that $k_f = \lceil (\lfloor o/x \rfloor - p)/P \rceil$ and $k_l = \lfloor (\lfloor (s * (N_A - 1) + o)/x \rfloor - p)/P \rfloor$. As a result,

$$b_f^p = p + \lceil (\lfloor o/x \rfloor - p)/P \rceil * P,$$

$$b_l^p = p + \lfloor (\lfloor (s * (N_A - 1) + o)/x \rfloor - p)/P \rfloor * P.$$

Any block in processor p will be in a unique data distribution pattern and will occur in that data distribution pattern. Therefore, for any block b in processor p, b can be decomposed into two factors α and β, which are the data distribution pattern number and the occurrence in the distribution pattern where b belongs, respectively. α and β can be, respectively, obtained by $\lfloor b/N_{pb} \rfloor$ and $\lfloor (b \bmod N_{pb})/P \rfloor$, where N_{pb} is the number of blocks in a data distribution pattern and $N_{pb} = \mathrm{lcm}(N_c, P)$, which has been discussed in Section 3.1. Given $(\alpha, \beta)^p$, a unique block number b can be obtained via

$$b = p + \alpha * N_{pb} + \beta * P.$$

Thus, we call $(\alpha, \beta)^p$ an *addressing pair*. Accordingly, the addressing pairs for the first and the last active blocks that are contained by processor p, b_f^p and b_l^p, are represented by $(\alpha_f, \beta_f)^p$ and $(\alpha_l, \beta_l)^p$, respectively.

In order to exactly evaluate the number of compressed local array elements allocated on a processor, the number of pseudo active elements has to be determined first. Let $\mathscr{PA}_h^p$ denote the number of pseudo active elements allocated to processor p and occurring before the first active element $T(o)$, and let $\mathscr{PA}_t^p$ denote the number of pseudo active elements occurring in the last active block b_l^p. The values of $\mathscr{PA}_h^p$ and $\mathscr{PA}_t^p$ can be calculated as follows.

To calculate $\mathscr{PA}_h^p$, we must decide whether the first active element $T(o)$ is allocated to processor p or not. If the first active element $T(o)$ is not allocated to p, the pseudo active elements can only occur in the blocks before the first active block b_f^p. Therefore, $\mathscr{PA}_h^p = \alpha_f * \mathscr{A}_{ptn}^p + \mathscr{C}_l^p(\beta_f)$, where $\mathscr{A}_{ptn}^p$ is the number of active elements allocated to processor p in a data distribution pattern. Otherwise, the first active element $T(o)$ is allocated to processor p; thus, in addition to the blocks

before the first active block b_f^p, the pseudo active elements can also occur in the first active block and need to be considered as well. Hence, $\mathscr{PA}_h^p = \alpha_f * \mathscr{A}_{ptn}^p + \mathscr{C}_l^p(\beta_f) + \lfloor (o \bmod x)/s \rfloor$. As a result, $\mathscr{PA}_h^p$ can be obtained by

$$\mathscr{PA}_h^p = \begin{cases} \alpha_f * \mathscr{A}_{ptn}^p + \mathscr{C}_l^p(\beta_f) & \text{if} \quad T(o) \notin p, \\ \alpha_f * \mathscr{A}_{ptn}^p + \mathscr{C}_l^p(\beta_f) + \lfloor (o \bmod x)/s \rfloor & \text{if} \quad T(o) \in p. \end{cases}$$

For the example shown in Fig. 12, $\mathscr{PA}_h^p$ for processor p_0 can be evaluated as follows. The first active block in p_0 is b_8, and the addressing pair for the block is $(0, 2)^{p_0}$. Since the first active element $T(28)$ is not allocated to processor p_0, the pseudo active elements in p_0 only occur in the blocks before the first active block b_8, that is, blocks b_0 and b_4. Accordingly, $\mathscr{PA}_h^{p_0} = 0 * 5 + \mathscr{C}_l^{p_0}(2) = 3$, where $\mathscr{A}_{ptn}^p = 5$. With regard to $\mathscr{PA}_h^p$ for processor p_1, the first active block in p_1 is b_5, and the addressing pair of this block is $(0, 1)^{p_1}$. Since the first active element $T(28)$ is allocated on p_1, the pseudo active elements occur not only in the blocks before the first active block, but also in the first active block. Thus, $\mathscr{PA}_h^{p_1} = 0 * 5 + \mathscr{C}_l^{p_1}(1) + 1 = 2$.

To evaluate $\mathscr{PA}_t^p$, we also must decide whether the last active element $T(s * (N_A - 1) + o)$ is allocated on processor p or not. If the last active element $T(s * (N_A - 1) + o)$ is not allocated on p, there is no pseudo active element in the last active block. Therefore, $\mathscr{PA}_t^p = 0$. For the example shown in Fig. 12, the values of $\mathscr{PA}_t^p$ for processors p_0, p_1 and p_2 are 0 because the last active element $T(115)$ is not allocated on p_0, p_1 and p_2. On the other hand, if the last active element is allocated on p, the number of pseudo active elements in the last active block b_l^p can be calculated by $\mathscr{PA}_t^p = \mathscr{C}_n^p(\beta_l) - \lfloor ((s * (N_A - 1) + o) \bmod x)/s \rfloor - 1$. For the case of p_3 shown in Fig. 12, since the last active element $T(115)$ is allocated on p_3, $\mathscr{PA}_t^{p_3} = \mathscr{C}_n^{p_3}(2) - 0 - 1 = 1$, where the last active block $b_l^{p_3}$ on p_3 is b_{23} and its addressing pair is $(1, 2)^{p_3}$. Consequently, $\mathscr{PA}_t^p$ can be obtained by

$$\mathscr{PA}_t^p = \begin{cases} 0 & \text{if} \quad T(s * (N_A - 1) + o) \notin p, \\ \mathscr{C}_n^p(\beta_l) - \lfloor ((s * (N_A - 1) + o) \bmod x)/s \rfloor - 1 & T(s * (N_A - 1) + o) \in p. \end{cases}$$

Since the first and the last active blocks on processor p and the corresponding addressing pairs can be evaluated, the number of active elements on processor p can be obtained accordingly. For processor p, the number of active elements which occur from the first block up to the last active block is $\alpha_l * \mathscr{A}_{ptn}^p + \mathscr{C}_l^p(\beta_l) + \mathscr{C}_n^p(\beta_l)$. However, the pseudo active elements are also counted. Therefore, the number of pseudo active elements should be deducted from the total. As a result, the number of active elements allocated on processor p, N_A^p can be obtained by

$$N_A^p = \alpha_l * \mathscr{A}_{ptn}^p + \mathscr{C}_l^p(\beta_l) + \mathscr{C}_n^p(\beta_l) - \mathscr{PA}_h^p - \mathscr{PA}_t^p.$$

We will explain the ideas behind the algorithm shown in Fig. 13 by using the following example. Consider the general two-level data-processor mapping shown in Fig. 12. Take the generation of the compressed local array of processor p_0 as an example. We have to evaluate the number of active elements allocated on processor

p_0 first. The first and the last active blocks on processor p_0 are b_8 and b_{20}, respectively. That is, $b_f^{p_0} = b_8$ and $b_l^{p_0} = b_{20}$. The addressing pairs $(\alpha_f, \beta_f)^{p_0}$ and $(\alpha_l, \beta_l)^{p_0}$ are $(0, 2)^{p_0}$ and $(1, 2)^{p_0}$, respectively. The calculations of $\mathscr{PA}_h^p$ and $\mathscr{PA}_t^p$ on processor p_0, described in previous paragraphs, produce values of 3 and 0, respectively. The number of active elements on processor p_0 is $N_A^{p_0} = 1 * 5 + \mathscr{C}_l^{p_0}(2) + \mathscr{C}_n^{p_0}(2) - 3 - 0 = 7$, where $\mathscr{A}_{ptn}^{p_0} = 5$. Since $\mathscr{A}_{ptn}^{p_0} = 5$, according to the Hole_Compression algorithm shown in Fig. 13, the first step is to generate the first 5 compressed local array elements. The *compression table* records the information used to generate $\mathscr{A}_{ptn}^p$ compressed local array elements without considering the pseudo active elements. Let $\mathscr{PA}$ be the number of pseudo active elements occurring before the first active element $T(o)$, which can be obtained by $\mathscr{PA} = \lfloor o/s \rfloor$. Obviously, $\mathscr{PA} = 9$ in this example. That is, 9 pseudo active elements occur before the first active element $T(o)$ in this example. Hence, the number of global indices in the two-level data-processor mapping should be reduced by 9 to obtain the number of real global indices in the mapping shown in Fig. 12. Therefore, according to the *compression table*, the first $\mathscr{A}_{ptn}^{p_0}$ compressed local array elements are $A(-9, -8, -2, 4, 5)$. Because $\mathscr{PA}_h^{p_0} = 3$, we discard three pseudo active elements and regenerate three more active elements to obtain the first real 5 compressed local array elements $A(4, 5, 11, 12, 18)$ on processor p_0. Since only 2 compressed local array elements need to be generated, we skip the second step of the Hole_Compression algorithm shown in Fig. 13 and go directly to step 3 to generate the last 2 compressed local array elements. As a result, the compressed local array elements on processor p_0 are $A^{p_0} = A(4, 5, 11, 12, 18, 24, 25)$.

3.3. $A(gl) \leftrightarrow A^p(loc)$ Transformations

Another important problem with hole compression is the transformations between the global indices of array elements and the local indices of the corresponding compressed local array elements. Suppose that gl is a global index of an array element, and that loc is the local index of the corresponding compressed local array element. The transformation between $A(gl) \leftrightarrow A^p(loc)$ is important for compiling array statements and data redistribution. In other words, given a gl, we have to find the values of p and loc, where p and loc are the processor number and the local index in A^p mapped by gl, respectively. On the other hand, given p and loc, we have to find the corresponding value of gl.

Given a gl, $(s * gl + o)$ is the global index of the aligned template cell and $\lfloor (s * gl + o)/x \rfloor$ is the block number mapped by gl. Let $b = \lfloor (s * gl + o)/x \rfloor$. The processor to which b belongs is $p = (b \bmod P)$. That is, p is the processor number mapped by gl. Let $(\alpha_b, \beta_b)^p$ be the addressing pair of b. The local index in A^p where gl is mapped can be determined as follows. The number of array elements occurring in front of b on processor p is $\alpha_b * \mathscr{A}_{ptn}^p + \mathscr{C}_l^p(\beta_b) - \mathscr{PA}_h^p$, where $\mathscr{A}_{ptn}^p$ is the number of active elements on processor p in a data distribution pattern and $\mathscr{PA}_h^p$ is the number of pseudo active elements allocated to processor p and occurring before the first active element $T(o)$. The number of array elements appeared before gl on block b is $\lfloor ((s * gl + o) \bmod x)/s \rfloor$. Therefore, the corresponding local index in the compressed local array A^p is $\alpha_b * \mathscr{A}_{ptn}^p + \mathscr{C}_l^p(\beta_b) - \mathscr{PA}_h^p + \lfloor ((s * gl + o) \bmod x)/s \rfloor$.

Thus, given a global index gl of array A, the processor number p and the local index loc in the compressed local array A^p mapped by gl can be obtained as follows.

$$p = b \bmod P,$$
$$loc = \alpha_b * \mathscr{A}^p_{ptn} + \mathscr{C}^p_l(\beta_b) - \mathscr{P}\mathscr{A}^p_h + \lfloor ((s * gl + o) \bmod x)/s \rfloor.$$

Consider the general two-level data-processor mapping shown in Fig. 12. Given $gl = 25$, we would like to find the corresponding processor number and the local index in the compressed local array. Based on an alignment stride $s = 3$ and offset $o = 28$, the global index of the aligned template cell is $103 (= 3 * 25 + 28)$. The corresponding block number of gl is $20 (= \lfloor 103/5 \rfloor)$. Hence, the corresponding processor number p is $0 (= 20 \bmod 4)$. The addressing pair of b_{20} is $(1, 2)^{p_0}$. The corresponding local index in the compressed local array A^{P_0} is $loc = 1 * 5 + \mathscr{C}^{p_0}_l(2) - 3 + 1 = 6$. As a result, $A(25)$ is mapped to $A^{p_0}(6)$.

On the other hand, given a processor number p and a local index in the compressed local array A^p, we would like to find the corresponding global index gl of array A. According to p and loc, we can restore the corresponding addressing pair of the block where gl is allocated and then obtain the corresponding global index gl. Let $(\alpha, \beta)^p$ be the addressing pair of the block mapped by gl. Since $\mathscr{P}\mathscr{A}^p_h$ is the number of pseudo active elements which are allocated to processor p and appear before the first active element $T(o)$, and $\mathscr{A}^p_{ptn}$ is the number of active elements on processor p within a data distribution pattern without considering the pseudo active elements, it follows that $\alpha = \lfloor (loc + \mathscr{P}\mathscr{A}^p_h)/\mathscr{A}^p_{ptn} \rfloor$. On the other hand, β is the occurrence number of the block where gl is allocated. The value of β can be obtained by checking the *compression table* to make sure that the condition that $\mathscr{C}^p_l(\beta)$ is the largest value but is no greater than the value of $((loc + \mathscr{P}\mathscr{A}^p_h) \bmod \mathscr{A}^p_{ptn})$ is satisfied. Since the values of $\mathscr{C}^p_l$ in a *compression table* are sorted, the process for determining an appropriate β can adopt either a simple linear search or an efficient binary search on the values of $\mathscr{C}^p_l$ in the *compression table*. According to the addressing pair, the global index gl can be obtained as follows:

$$gl = \alpha * \mathscr{A}_{ptn} + \mathscr{C}^p_g(\beta) + ((loc + \mathscr{P}\mathscr{A}^p_h) \bmod \mathscr{A}^p_{ptn}) - \mathscr{C}^p_l(\beta) - \mathscr{P}\mathscr{A},$$

where $\mathscr{P}\mathscr{A}$ is the number of pseudo active elements occurring before the first active element $T(o)$ and $\mathscr{P}\mathscr{A} = \lfloor o/s \rfloor$.

Take $p = 0$ and $loc = 6$ in Fig. 12 as an example. Since, in this data-processor mapping, $\mathscr{P}\mathscr{A}^{p_0}_h = 3$ and $\mathscr{A}^{p_0}_{ptn} = 5$, $\alpha = \lfloor (6 + 3)/5 \rfloor = 1$ and $\beta = 2$, it follows that $gl = 1 * 20 + 13 + ((6 + 3) \bmod 5) - 3 - 9 = 25$, where $\mathscr{A}_{ptn} = 20$ and $\mathscr{P}\mathscr{A} = \lfloor 28/3 \rfloor = 9$. Thus, given $p = 0$ and $loc = 6$, $A^{p_0}(6)$ is mapped by $A(25)$.

4. EXPERIMENTAL RESULTS

This section presents experimental results obtained to evaluate the performance of our proposed scheme and the work proposed in [12]. In our experiments, three methods are compared. Two are virtual processor schemes [12], and the last is our

proposed scheme. The virtual processor scheme includes *virtual block* and *virtual cyclic* approaches. Generally speaking, a block-cyclic distribution can be viewed as either a block distribution on a set of virtual processors, which are then cyclically mapped to processors, or a cyclic distribution on a set of virtual processors, which are block-wise mapped to processors. The former is termed a virtual block scheme and is denoted as *v-block* here. The latter is termed a virtual cyclic scheme and is denoted as *v-cyclic*. As for our proposed scheme, we denote it as *ours* in the experiment.

Specifically, suppose that array A has N_A elements, that $A(i)$ is aligned with $T(s * i + o)$, and that T is block-cyclically distributed onto P processors with block size x. The experiment measures the execution time of each method. Actually, our method includes two phases. One is done at compile-time and the other is done at runtime. The generations of the *class table* and *compression tables* are done at compile-time if the parameters are known at compile-time. The generation of the compressed local array is done at runtime. Hence, in most cases, the times to generate the *class table* and *compression tables* can be ignored when we generate the compressed local array. However, in this experiment, the time we measured for our scheme also includes the table generation time.

In this experiment, we fix all the parameters except for s and x since the major factors affecting hole compression are s and x. Thus, the number of array elements N_A, the number of processors P, and the alignment offset o are set to 50000, 16, and 0, respectively. In real programs, the values of s and x are very correlated with program behavior. However, we do not take the code part into consideration since the generation of the compressed local array is irrelevant with the code. We test various values of s and x to observe the variation of our scheme against virtual processor schemes. The experiments are performed on a DEC Alpha 3000/400 workstation. In the experiment, times are measured in terms of CPU time, and the time unit used is one microsecond. We evaluate the execution time needed to generate the compressed local array for one processor. For each case, we run 100 times, each for the processor generated by a random number generator. Each experimental result is the total time of the 100 executions.

Figure 14a shows the performance comparisons of the three methods when the alignment stride is fixed at 12 and the block size varies from 1 to 24. In Fig. 14a,

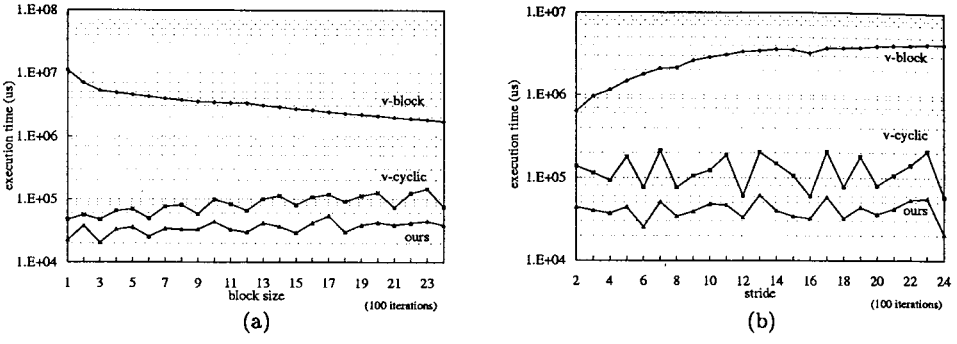


FIG. 14. Performance comparisons of the three methods. (a) The alignment stride s is fixed at 12, and the block size varied from 1 to 24. (b) The block size x is fixed at 12, and the alignment stride varied from 2 to 24.

the x-axis is the block size, and, the y-axis is the accumulated execution time. The proposed method outperforms the two virtual processor approaches, especially the virtual block approach. In Fig. 14a, the execution time of the virtual block approach describes as the block size increases. This is because the execution time of the virtual block approach is proportional to the number of virtual processors. Therefore, as the block size increases, the number of virtual processors contained by a processor decreases. Thus, the execution time of the virtual block approach decreases accordingly. Similarly, the execution time of the virtual cyclic approach is also proportional to the number of active virtual processors. The number of active virtual processors is inversely proportional to $\gcd(P * x, s)$ [12]. As a result, the execution time of the virtual cyclic approach is inversely proportional to $\gcd(P * x, s)$. The experiments also verifies this phenomenon. Therefore, there is no regular pattern for either the block size or the alignment stride. For our proposed method, the execution time is closely related to the number of occurrences, N_{pb}^{p}, which is obtained by $N_{pb}^{p} = N_c/\gcd(N_c, P)$, where $N_c = s/\gcd(s, x)$. The larger the number of occurrences, the more time our method takes. Hence, the execution time of our method is proportional to the number of occurrences, just as Fig. 14a shows.

On the other hand, Fig. 14b shows performance comparisons of the three methods when the block size is fixed at 12 and the alignment stride varies from 2 to 24. As is well known, template cells should span the entire range of array elements. In this situation, since the number of array elements is fixed at 50000, the number of template cells will increase as the alignment stride increases. Hence, for the virtual block approach, the number of virtual processors will increase when the alignment stride increases. As a result, the execution time of the virtual block approach increases if the alignment stride increases. Similar to Fig. 14a, the execution time of the virtual cyclic approach is inversely proportional to $\gcd(P * x, s)$, and that of our method is directly proportional to the number of occurrences, N_{pb}^{p}. Obviously, our proposed scheme also outperforms the two virtual processor approaches when the block size is fixed and the alignment stride varies.

In Figs. 14a and 14b, the virtual cyclic approach outperforms the virtual block approach in every case. However, the virtual cyclic approach does not always outperform the virtual block approach. Since the block sizes shown in Figs. 14a and 14b are too small, the number of virtual processors in the virtual block approach is much larger than that in the virtual cyclic approach. Furthermore, the execution times of the two virtual processor approaches are proportional to the number of virtual processors. Thus, the execution time of the virtual block approach is much longer than that of the virtual cyclic approach. Nevertheless, as long as the block size is large enough, the execution time of the virtual block approach can be shorter than that of the virtual cyclic approach. Figs. 15 and 16 show this phenomenon.

We experimented using various alignment strides ranging from 2 to 100 and various block size ranging from 50 to 10000. The alignment strides studied are $s = 2, 8, 16, 25, 32, 48, 60, 72, 80,$ and 100, and the block sizes studied are $b = 50, 100, 250, 500, 750, 1000, 2500, 5000, 7500,$ and 10000. Figure 15 shows performance comparisons of the three methods for all the tested block sizes and two selected alignment strides, $s = 2$ and 100. On the other hand, Fig. 16 shows

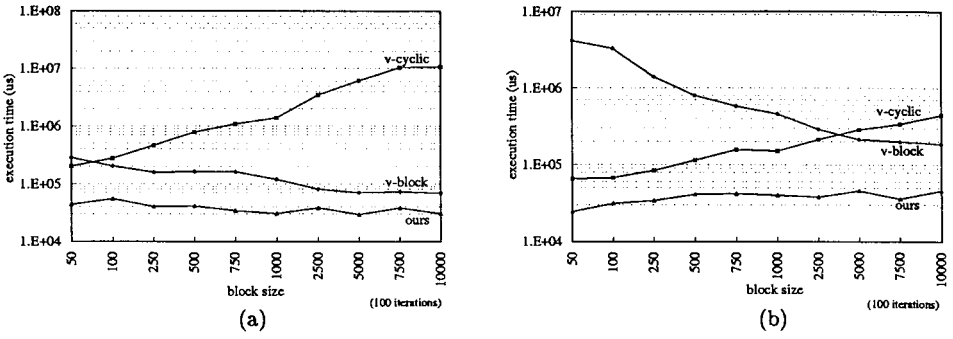


FIG. 15. Performance comparisons of the three methods. (a) The alignment stride is fixed at 2 ($s = 2$), and the block size varies between the selected values. (b) The alignment stride is fixed at 100 ($s = 100$), and the block size varies between the selected values.

performance comparisons of the three methods for all the tested alignment strides and two selected block sizes $x = 50$ and 10000. In Fig. 15, as the block size grows, the number of virtual processors increases. Therefore, the execution time used by the virtual cyclic approach increases. However, the execution time of the virtual block approach decreases as the block size increases. On the other hand, in Fig. 16, the execution time of the virtual cyclic approach decreases and that of the virtual block approach increases when the alignment stride increases. In Fig. 16a, the execution time of the virtual cyclic approach is lower than that of the virtual block approach. However, when the block size is large, as shown in Fig. 16b, the execution time of the virtual cyclic approach turns out to be longer than that of the virtual block approach.

One more significant result of our approach over the two virtual processor approaches, in addition to better performance, is the stability of the execution time. Obviously, there is a tradeoff between the virtual block approach and the virtual cyclic approach. We have to decide which approach is appropriate, the virtual block approach or the virtual cyclic approach, when either the block size or the alignment stride is changed. Nevertheless, from Figs. 15 and 16, the execution time of our approach is very stable when the alignment stride or the block size is changed from a small value to a large value.

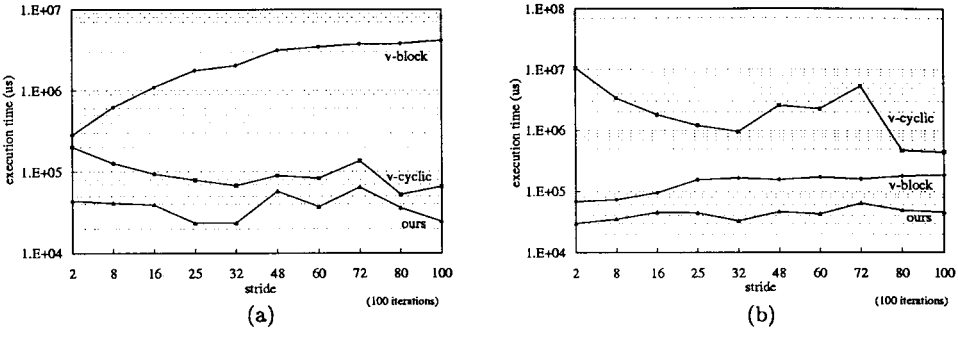


FIG. 16. Performance comparisons of the three methods. (a) The bock size is fixed at 50 ($b = 50$), and the alignment stride varies between the selected values. (b) The block size is fixed at 10000 ($b = 10000$), and the alignment stride varies between the selected values.

5. RELATED WORKS

Two-level data-processor mappings involving first *aligning* related array objects with a template and then *distributing* the template onto processors have been well-used with data-parallel languages in distributing data onto processors. In recent years, researchers who have focused their attention on compiling array statements or array redistribution have taken only identical alignment into consideration. That is, the alignment stride is forced to be 1, and the alignment offset is forced to be 0. As for compiling array statements, enumerating local memory access sequences and generating communication sets for compiling array statements with unit access stride were considered in [17] and for nonunit access stride were considered in [15]. Both works considered the distribution to be either a block or a cyclic distribution. With regard to the block-cyclic distribution, the enumerations of the local memory access sequences and the generation of communication sets for compiling array statements have been extensively discussed recently in the literature [7, 9, 18, 19, 23, 25, 28, 29]. On the other hand, researchers who have studied the generation of communication sets for compiling array redistributions have seldom taken arbitrary affine alignment into consideration [5, 10, 11, 20, 21, 24]. However, affine alignment wastes a lot of memory space if the alignment stride is non unit. Such a wastage of memory usage is unacceptable considering the limited local space in processors on distributed-memory multicomputers. Allocating space only for useful template cells is, therefore, of critical importance for distributed-memory multicomputers.

Gradually, a number of researchers have become aware of this fact and propose methods for compressing holes for compiling two-level data-processor mapping with nonunit alignment stride. For a two-level data-processor mapping with affine alignment and block-cyclic distribution, the enumeration of local memory access sequences for compiling array statements was considered in [4]. Both identical alignment and affine alignment with hole compression were addressed. A finite state machine (FSM) approach was adopted to traverse the local index space of each processor. The construction of a state table involves solving k linear Diophantine equations and performing a sorting operation. Moreover, the FSM approach is a runtime technique. High runtime overhead to enumerate local memory access sequences is the result.

The work improving the FSM approach [4] was proposed in [13, 14]. Efficient FSM table generation was proposed. The improved method enumerates the local memory access sequences by viewing the accessed elements an integer lattice. The sorting step in [4] is avoided in the improved method. However, runtime resolution of Diophantine equations is also required.

In [12], the authors proposed virtual processor approaches. From the different viewpoint of a block-cyclic distribution, the virtual processor approach actually contains two approaches, one termed the virtual block approach and the other the virtual cyclic approach. The virtual block approach views a block-cyclic distribution as a block distribution on a set of virtual processors, which are then cyclically mapped onto processors. On the other hand, the virtual cyclic approach views a block-cyclic distribution as a cyclic distribution on a set of virtual processors, which

are then block-wise mapped onto processors. Therefore, if hole compression for block and cyclic distributions are derived, hole compression for block-cyclic distribution can be obtained accordingly. However, in addition to the disadvantages mentioned in Section 4, holes cannot be totally eliminated by using the two virtual processor approaches. Moreover, the virtual cyclic approach can not preserve the order of compressed local array elements.

An approach similar to the virtual processor approach was presented in [27]. In [27], row-wise and column-wise scanning of the index space were proposed. One corresponds to the virtual block approach and the other to the virtual cyclic approach. They can also be applied to affine alignment with hole compression. Based on scanning polyhedra, an approach to enumerating local memory access sequences and generating communication sets was proposed in [1]. Complex loop bounds and local array subscripts of the generated code will incur significant overhead.

In this paper, a new approach has been proposed to compress holes for compiling two-level data processor mappings. The proposed approach is also a table-based approach. However, our approach does not need to solve k linear Diophantine equations and has no sorting operation. Furthermore, the proposed approach has less runtime overhead. In Section 4, we extensively compared our method with the method proposed in [12]. Experimental results also verify the advantages of our proposed approach. Moreover, the proposed approach has higher stability than existing methods. The execution time varies a little with the alignment stride and the distribution block size. In addition, the proposed approach can be easily implemented.

6. CONCLUSIONS

Data-parallel languages support two-level data-processor mappings which users can use to specify data distributions. However, a non-unit alignment stride always results in a lot of memory holes, even for a small alignment stride. Holes result in not only memory wastage, but also performance degradation. Eliminating holes is of critical importance for compiling two-level data-processor mappings. Therefore, this paper presents compilation techniques for solving this problem. Our approach uses a *class table* and a *compression table* to facilitate the generation of a compressed local array for each processor. The *class table* is used to record the distribution of blocks in a class cycle, and the *compression table* is used to record the distribution of blocks in a data distribution pattern on a processor. The time complexities of the constructions of these two tables are $O(s)$ in the worst case, where s is the alignment stride. The approach proposed in this paper is straightforward but efficient. Moreover, one significant advantage of our approach is its stability. The execution time required by our approach varies a little when the alignment stride or the distribution block size increase. As for implementation, the proposed method is easy to implement. Experimental results do confirm the advantages of our proposed method over the existing methods.

On the other hand, the compilations of array statements and data redistribution are very important for compiling data-parallel languages. However, compiling array

statements or data redistribution incurs an indexing overhead and a communication overhead. Alleviating the overheads that result from compiling array statements or data distribution has become very important for distributed-memory multicomputers. Hence, based on the idea of hole compression, future work will focus on efficiently generating communication sets for compiling array statements and data redistribution in order to reduce the indexing and communication overheads.

REFERENCES

1. C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell, A linear algebra framework for static HPF code distribution, *in* "The Fourth International Workshop on Compilers for Parallel Computers," pp. 117–132, Delft, The Netherlands, December 1993.
2. B. M. Chapman, P. Mehrotra, and H. P. Zima, Programming in Vienna Fortran, *Sc. Prog.* **1**, 1 (August 1992).
3. B. M. Chapman, P. Mehrotra, and H. P. Zima, Vienna Fortran—A Fortran language extension for distributed memory multiprocessors, *in* "Language, Compilers and Runtime Environments for Distributed Memory Machines" (J. Saltz and P. Mehrotra, Eds.), pp. 39–62, 1992.
4. S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng, Generating local addresses and communication sets for data parallel programs, *J. Parallel Distrib. Comput.* **26**, 1 (April 1995), 72–84.
5. F. Coelho and C. Ancourt, Optimal compilation of HPF remappings, *J. Parallel Distrib. Comput.* **38**, 2 (Nov. 1996), 229–236.
6. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. W. Tseng, and M. Wu, "Fortran D Language Specificiation," Tech. Rep. 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 compiling array expressions for efficient execution on distributed-memory machines, *J. Parallel Distrib. Comput.* **32**, 2 (Feb. 1996), 155–172.
8. High Performance Fortran Forum, "High Performance Fortran Language Specification," Version 1.1, (November 1994).
9. S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi, Compilation techniques for block-cyclic distributions, *in* "proceedings of ACM International Conference on Supercomputing," pp. 392–403, July 1994.
10. S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan, "Multi-phase Redistribution: A Communication-Efficient Approach to Array Redistribution," Tech. Rep. OSU-CISRC-9/94-52 Department of Computer and Information Science, Ohio State University, 1994.
11. S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan, Multiphase array redistribution: Modeling and evaluation, *in* "Proceedings of International Parallel Processing Symposium," pp. 441–445, April 1995.
12. S. D. Kaushik, C.-H. Huang, and P. Sadayappan, Efficient index set generation for compiling HPF array statements on distributed-memory machines, *J. Parallel Distrib. Comput.* **38**, 2 (Nov. 1996), 237–247.
13. K. Kennedy, N. Nedeljković, and A. Sethi, Efficient address generation for block-cyclic distributions, *in* "Proceedings of ACM International Conference on Supercomputing," pp. 180–184, July 1995.
14. K. Kennedy, N. Nedeljković, and A. Sethi, A linear-time algorithm for computing the memory access sequence in data-parallel programs, *in* "Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming," pp. 102–111, July 1995.
15. C. Koelbel, Compile-time generation of communication for scientific programs, *in* "Supercomputing 91," pp. 101–110, Nov. 1991.

16. C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, "The High Performance Fortran Handbook," MIT Press, Cambridge, MA, 1994.

17. C. Koelbel and P. Mehrotra, Compiling global name-space parallel loops for distributed execution, *IEEE Trans. Parallel Distrib. Systems* **2**, 4 (Oct. 1991), 440–451.

18. S. P. Midkiff, Local iteration set computation for block-cyclic distributions, *in* "Proceedings of International Conference on Parallel Processing," Vol. II, pp. 77–84, Aug. 1995.

19. S. P. Midkiff, Optimizing the representation of local iteration sets and access sequences for block-cyclic distributions, *in* "Languages and Compilers for Parallel Computing," 1996. [Also available in D. Sehr *et al.* (Eds.), "Lecture Notes in Computer Science," Vol. 1239, pp. 420–434, Springer-Verlag, Berlin, 1997.]

20. D. J. Palermo, E. W. Hodges IV, and P. Banerjee, Interprocedural array redistribution data-flow analysis, *in* "Languages and Compilers for Parallel Computing," 1996. [Also available in D. Sehr *et al.* (Eds.), "Lecture Notes in Computer Science," Vol. 1239, pp. 435–449, Springer-Verlag, Berlin, 1997.]

21. S. Ramaswamy, B. Simons, and P. Banerjee, Optimizations for efficient array redistribution on distributed memory multicomputers, *J. Parallel Distrib. Comput.* **38**, 2 (Nov. 1996), 217–228.

22. K.-P. Shih, J.-P. Sheu, and C.-H. Huang, Table-lookup approach for compiling two-level data-processor mappings in HPF, *in* "Proceedings of the Tenth International Workshop on Languages and Compilers for Parallel Computing," Minneapolis, MN, Aug. 1997.

23. J. M. Stichnoth, D. O'Hallaron, and T. Gross, Generating communication for array statements: Design implementation, and evaluation, *J. Parallel Distrib. Comput.* **21** (1994), 150–159.

24. R. Thakur, A. Choudhary, and J. Ramanujam, Efficient algorithms for array redistribution, *IEEE Trans. Parallel Distrib. Systems* **7**, 6 (June 1996), 587–594.

25. A. Thirumalai and J. Ramanujam, Efficient computation of address sequences in data parallel programs using closed forms for basis vectors, *J. Parallel Distrib. Comput.* **38**, 2 (Nov. 1996), 188–203.

26. C. W. Tseng, "An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines," Ph.D. Thesis, Rice University, 1993.

27. C. van Reeuwijk, W. Denissen, H. J. Sips, and E. M. Paalvast, "An implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems," Tech. Rep. CP-96-001 Computational Physics Section, Faculty of Applied Physics, Delft University of Technology, 1996.

28. A. Venkatachar, J. Ramanujam, and A. Thirumalai, Generalized overlap regions for communication optimization in data-parallel programs, *in* "Languages and Compilers for Parallel Computing," 1996. [Also available in D. Sehr *et al.* (Eds.), "Lecture Notes in Computer Science," Vol. 1239, pp. 404–419, Springer-Verlag, Berlin, 1997.]

29. W.-H Wei, K.-P. Shih, and J.-P. Sheu, Compiling array references with affine functions for data-parallel programs, *J. Inform. Sc. Eng.* **14**, 4 (Dec. 1998), 695–723.