Table-Lookup Approach for Compiling Two-Level Data-Processor Mappings in HPF*

Kuei-Ping Shih[1], Jang-Ping Sheu[1], and Chua-Huang Huang[2]

[1] Department of Computer Science and Information Engineering
National Central University
Chung-Li 32054, Taiwan
Email: steven@axp1.csie.ncu.edu.tw
sheujp@csie.ncu.edu.tw

[2] Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
E-mail: chh@cis.ohio-state.edu

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

1 Introduction

Generally speaking, data parallel languages such as Fortran D[4], HPF (High Performance Fortran)[6], and Vienna Fortran[2] support two-level data-processor mapping. A two-level data-processor mapping provides 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 distribution phase, three regular data distributions, *block*, *cyclic*, and *block-cyclic* data distributions, are provided by these languages. However, the *block-cyclic* distribution is known to be the most general data distribution since both the *block* and the *cyclic* distributions can be represented by the *block-cyclic* distribution. The program model considered in this paper is demonstrated

* This work was supported in part by the National Science Council of the Republic of China under Grant #NSC86-2213-E-008-020.

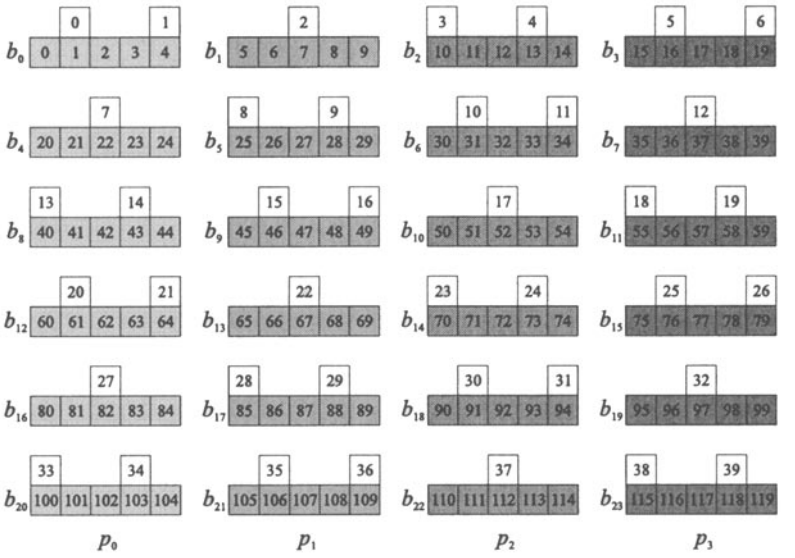
!HPF$ PROCESSORS $PROC(P)$
!HPF$ ALIGN $A(i)$ WITH $T(s*i+o)$
!HPF$ DISTRIBUTE $T(cyclic(x))$ ONTO $PROC$
...

Fig.1. HPF-like program model.

in Fig. 1, where P is the number of processors, s is the alignment stride, o is the alignment offset and $cyclic(x)$ is the representation of a $block - cyclic$ distribution and x is the distribution block size. Fig. 2(a) illustrates an example in this model, where $P = 4$, $s = 3$, $o = 1$, and $x = 5$. 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 gray squares represent different template cells mapped to different processors and the number in the square is the global index of that template cell.

In a two-level data-processor mapping, if the alignment stride is non-unit, the mapping will cause lots of *holes*. Memory holes caused by the non-unit alignment stride will result in a large amount of memory wastage, even for a small alignment stride. Therefore, only the template cells aligned by array elements need to be mapped to memory locations and all the holes should be removed. Suppose the number of template cells is N_T and s is the alignment stride. Thus, only N_T/s template cells are aligned by array elements. The percentage of memory usage is $\frac{N_T/s}{N_T} \times 100\%$ and equals $\frac{1}{s} \times 100\%$. In other words, the percentage of memory wastage is $\frac{s-1}{s} \times 100\%$. The larger the alignment stride is, the more the memory usage wastes. Even for the least positive non-unit alignment stride 2, it still has 50% waste of memory space. Fig. 2(b) shows the distributions of array elements onto processors with hole compression. Obviously, there are 30 memory spaces that should be allocated by each processor if hole compression is not performed. However, only a few template cells are aligned by array elements. The rest of template cells, which have no array elements aligned with, are never used and are *holes*. As a result, only 10 memory spaces are needed by each processor after hole compression is performed. Therefore, compressing holes is quite necessary and important. The processes that map the useful template cells to processors and eliminate useless holes are called *hole compression*. In addition to increase memory usage, removing *holes* can also improve spatial locality and, furthermore, achieve higher performance.

This paper presents compilation techniques to efficiently remove holes for two-level data-processor mappings. Observing the two-level mapping shown in Fig. 2(a), one can find out that the distribution patterns for every three blocks are identical. By the observation, we can classify all blocks into classes and design a table, named *class table*, to record the attributes of the first repeated pattern. On the other hand, from processor viewpoint, the above observation is true as well. Therefore, another table named *compression table* are established to



(a)

local address	0	1	2	3	4	5	6	7	8	9
p_0	0	1	7	13	14	20	21	27	33	34
p_1	2	8	9	15	16	22	28	29	35	36
p_2	3	4	10	11	17	23	24	30	31	37
p_3	5	6	12	18	19	25	26	32	38	39

(b)

Fig.2. A Two-level data-processor mapping. (a) Array $A(i)$ is aligned with template $T(3*i+1)$ and the template is then distributed onto 4 processors with *cyclic*(5) distribution. (b) The distribution of array elements onto processors with hole compression.

record the attributes of the first repeated data distribution pattern on that processor. *Compression table* is established according to the *class table*. We design systematic methods to construct these tables and the time complexity of each construction is $O(s)$ in worst case, where s is the alignment stride. Hole compression can be easily achieved by using *compression table* accordingly. Table-based approach can save a lot of redundant computations since the computations of repetition of fixed patterns can be obtained by table lookup instead of recomputations. Experimental results verify the advantages of the proposed approach. Moreover, the proposed approach has high stability against existing methods. The execution time varies a little with the alignment stride and the distribution

block size. In addition, from implementation viewpoint, the proposed approach can be easily implemented as well.

This paper is organized as follows. On compiling two-level data-processor mapping, *class table* is useful for summarizing the two-level mapping. Therefore, the structure and characteristics of *class table* are presented in Section 2. Section 3 introduces how to utilize *class table* to construct *compression table*. By using *compression table*, compressing holes and generating compressed local array are also described. Experimental results to show the advantages of our method over existing methods are provided in Section 4. Section 5 discusses the related work. Section 6 concludes the paper and points out the possible direction of future research as well.

2 Characteristics of Class Table

As compiling array statements, we have designed a useful structure to summarize the characteristics of access patterns for an array statement, while the data distribution is block-cyclic distribution [15]. Based on the similar concept, a structure named *class table* is designed in this paper to record the attributes of a two-level data-processor mapping. In this section we briefly describe the basic components of *class table*. Without loss of generality, we assume every numbering system is starting from zero, such as numbering array elements, template cells, and processors, etc.

Suppose array element $A(i)$ is aligned with template T at $(s * i + o)$, where s is the *alignment stride* and o is the *alignment offset*. Two different alignment offsets o_1 and o_2 lead to *isomorphic* data-processor mapping if and only if $o_1 \equiv o_2 \pmod{s}$. In order to reuse the same *class table* for isomorphic data-processor mappings, the alignment offset o is reduced to r as we are generating *class table*, where $r = (o \mod s)$. For example, suppose array $A(i)$ is aligned with a template T at $(3i + 10)$. The alignment phase is illustrated in Fig. 3. In this example, $s = 3$ and $o = 10$. Since $10 > 3$, the alignment offset is reduced to 1, which is equal to $(10 \mod 3)$. For the reduced alignment, a template cell t has an array element aligned with if and only if $t \equiv r \pmod{s}$, such as the template cells $1, 4, 7, 10, 13, 16, \cdots$, in this example. For these template cells with which have array elements aligned, a template cell t is *active* if $o \leq t \leq (s * (N_A - 1) + o)$; otherwise, it is termed *pseudo active*, where N_A is the number of array elements. For this example, the active template cells are starting from 10 and then each strides 3 and the template cells 1, 4, 7 are pseudo active elements. Fig. 3 also illustrates the active elements and pseudo active elements for this example. The boldfaced numbers are active elements and the italic numbers are pseudo active elements. Note that the pseudo active elements are viewed as the same with active elements when we are generating *class table* and *compression table*. However, the pseudo active elements will not be counted when we are generating the compressed local array.

For a *cyclic*(x) distribution, every x template cells forms a block. A block is numbered according to the occurrence of the block in the data-processor map-

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

Fig.3. The illustration of *active* elements and *pseudo active* elements. The boldfaced numbers are *active* elements and the italic numbers are *pseudo active* elements.

ping. Suppose the numbers of array elements and template cells are N_A and N_T, respectively. Let N_b be the number of blocks. Thus $N_b = \lceil N_T/x \rceil$. According to the alignment stride and offset, blocks of the same format can be classified into the same *class*. In other words, those blocks within which have the same positions of active elements are classified into the same class. For example, consider the two-level data-processor mapping shown in Fig. 2(a). Blocks 0, 3, 6, 9, $\cdots$ have the same positions of active elements. These blocks are classified into the same class. Similarly, blocks 1, 4, 7, 10, $\cdots$ can be classified into a class and blocks 2, 5, 8, 11, $\cdots$ is another class. The following theorem demonstrates that, for a two-level data-processor mapping, according to the positions of active elements within the blocks, all blocks can be classified into different classes and the number of classes is equal to $s/\gcd(s,x)$, where $\gcd(a,b)$ is the greatest common divisor of a and b. Due to the space limitation, all proofs in the paper are omitted. Whoever is interested in details can refer to [10].

Theorem 1. *For any two-level data-processor mapping that 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 template blocks can be classified into $s/\gcd(s,x)$ classes.* □

Let N_c be the number of classes. By Theorem 1, $N_c = s/\gcd(s,x)$. Since all blocks are classified into $s/\gcd(s,x)$ classes, we number the class number of each block in lexicographical order and every $s/\gcd(s,x)$ blocks repeats again. Formally, block b belongs to class ($b \mod N_c$). 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 $\{b, (b+1), (b+2), \ldots, (b+N_c-1) \mid b \equiv 0 \pmod{N_c}\}$ are a period in terms of classes and we term such a period a *class cycle*. For a class cycle, a *class table* is used for recording the first, the last and the number of active elements on a class. For a class c, $\mathcal{A}_f(c)$ represents the order of occurrence in a class cycle for the *first* active element in c, $\mathcal{A}_l(c)$ the order of occurrence in a class cycle for the *last* active element in c, and $\mathcal{A}_n(c)$ the number of active elements within c. In addition to the original meanings described above, *class table* also contains a lot of additional information implicitly. For a class c, $\mathcal{A}_f(c)$ implies total number of active elements contained from class 0 to class $(c-1)$ and $(\mathcal{A}_l(c)+1)$ implies total 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 $\mathcal{A}_l(N_c-1)+1$. Let $\mathcal{A}_c$ be the number of active elements contained in a class cycle. Thus, $\mathcal{A}_c = \mathcal{A}_l(N_c-1)+1$.

For the example shown in Fig. 2(a), the *class table* is shown in Fig. 4. In this example, blocks can be classified into 3 classes. The blocks 0, 3, 6, 9, $\cdots$, are

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

Fig.4. Class table for the example shown in Fig. 2(a).

$N_c = s/\gcd(s,x)$ /* the number of classes */
$r = o \mod s$ /* reduced alignment offset */
DO $c = 0$ **TO** $(N_c - 1)$
  $\mathcal{A}_f(c) = \lceil \max(c * x - r, 0)/s \rceil$
  $\mathcal{A}_l(c) = \lfloor ((c+1) * x - r - 1)/s \rfloor$
  $\mathcal{A}_n(c) = \mathcal{A}_l(c) - \mathcal{A}_f(c) + 1$
ENDDO

Fig.5. Class_Table_Generation algorithm.

class 0, the blocks 1, 4, 7, 10, $\cdots$, are class 1, and the blocks 2, 5, 8, 11, $\cdots$, are class 2. Blocks 0, 1, 2 form a class cycle and blocks 3, 4, 5 form another class cycle, and so on. For the first class in a class cycle, the occurrences of the first and the last active elements are 0 and 1 respectively, and the number of active elements in this class is 2. Thus, $\mathcal{A}_f(0) = 0$, $\mathcal{A}_l(0) = 1$, and $\mathcal{A}_n(0) = 2$. Likewise, $\mathcal{A}_f(1) = 2$, $\mathcal{A}_l(1) = 2$, and $\mathcal{A}_n(1) = 1$ for class 1 and $\mathcal{A}_f(2) = 3$, $\mathcal{A}_l(2) = 4$, and $\mathcal{A}_n(2) = 2$ for class 2. Hence, the number of active elements within a class cycle, $\mathcal{A}_c$, is $5 (= \mathcal{A}_l(2) + 1)$. Fig. 5 shows the algorithm to generate a *class table*, which is termed Class_Table_Generation algorithm. The major factors affecting the construction of a *class table* are the alignment stride s, alignment offset o, and the distribution block size x. The time complexity of Class_Table_Generation algorithm is $O(s/\gcd(s,x))$.

3 Hole Compression

In the section, how to use the *class table* to construct a *compression table* to extract information from a repeated data distribution pattern on a processor will be proposed. The generation of the compressed local array for a processor by using the *compression table* will be described as well.

3.1 Construction of Compression Table

Suppose a two-level data-processor mapping is of the form shown in Fig. 1. By Theorem 1, for arbitrary two-level data-processor mapping, all blocks can be classified into N_c classes, where $N_c = s/\gcd(s,x)$. Therefore, for every

$\text{lcm}(N_c, P)$ blocks, the same data distribution pattern will repeat again. In other words, blocks within a data distribution pattern can be viewed as different, but are identical for every different data distribution pattern. Hence, we only have to consider the first data distribution pattern and the rest of data distribution patterns can be done likewise. As a result, a *compression table* to record the information of the generation of compressed local array for a processor on the first data distribution pattern is proposed. Similar to the *class table*, *compression table* also regards an alignment offset o as the reduced alignment offset r if $o \geq s$, where $r = (o \mod s)$.

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 $N_{pb}^p = N_c/\gcd(N_c, P)$. That is, on each processor, each data distribution pattern contains N_{pb}^p blocks. We number a block according to the order of occurrence of the block on a processor within a data distribution pattern. For the example shown in Fig. 2(a), 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 occurred on processors 0, blocks 1 and 5 are the first and the second blocks occurred on processor 1 within the first data distribution pattern, and blocks 12 and 13 are the first blocks occurred respectively on processor 0 and 1 within the second data distribution pattern, and so on.

From processor's viewpoint, 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 cells, their indices are different in only a fixed offset. Hence, for a processor, only how to compress holes for the first data distribution pattern needs to be considered and, for the rest of data distribution patterns, only the fixed offset needs to be evaluated. To facilitate hole compression, we design a structure, termed *compression table*, to characterize blocks in the first data distribution pattern on a processor. For a block in processor p, the *compression table* records the following three items: C_g^p, C_n^p, and C_l^p.

- C_g^p, the global index of the first array element in a block on processor p,
- C_n^p, the number of array elements in a block on processor p,
- C_l^p, the local index in the compressed local array for the first array element in a block on processor p.

Take the *compression table* of processor p_0 as an example. Block b_0 is the first block within the first data distribution pattern occurred on processor p_0. The global index of the first array element in block 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, $C_g^0(0) = 0$, $C_n^0(0) = 2$, and $C_l^0(0) = 0$. The second block within the first data distribution pattern occurred on processor

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

(a)

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

(b)

Fig.6. Compression tables for processor p_0 and p_1 in the example shown in Fig. 2(a). (a) The *compression table* of processor p_0. (b) The *compression table* of processor p_1.

p_0 is block b_4. The global index of the first array element in block 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, $\mathcal{C}_g^0(1) = 7$, $\mathcal{C}_n^0(1) = 1$, and $\mathcal{C}_l^0(1) = 2$. Similarly, $\mathcal{C}_g^0(2) = 13$, $\mathcal{C}_n^0(2) = 2$, and $\mathcal{C}_l^0(2) = 3$. The *compression tables* of processor p_0 and p_1 for the example shown in Fig. 2(a) are illustrated in Fig. 6 (a) and (b), respectively.

The construction of *compression table* for processor p is described as follows. Since all blocks are classified into N_c classes, every N_c blocks forms a class cycle. For any block b, there are $\lfloor \frac{b}{N_c} \rfloor$ class cycles appeared before b. Furthermore, each class cycle contains $\mathcal{A}_c$ active elements. Hence, there are $(\lfloor \frac{b}{N_c} \rfloor * \mathcal{A}_c)$ active elements in these class cycles. From Section 2, $\mathcal{A}_f(c)$ implicitly implies 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 $\lfloor \frac{b}{N_c} \rfloor * \mathcal{A}_c + \mathcal{A}_f(c)$, where c is the class number of b. Therefore, $\mathcal{C}_g^p = \lfloor \frac{b}{N_c} \rfloor * \mathcal{A}_c + \mathcal{A}_f(c)$. On the other hand, $\mathcal{C}_n^p$ can be obtained by $\mathcal{A}_n(c)$ and $\mathcal{C}_l^p$ can be evaluated by the local index of the first array element in the previous block plus the number of active elements contained by the previous block. Of course, the local index of the first array element in the first block on the first data distribution pattern is 0. Note that $\mathcal{C}_l^p$ can also represent the number of active elements occurred before the block in a data distribution pattern. The algorithm to generate *compression table* is demonstrated in Fig. 7. The time complexity of 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. By Section 2, $N_c = s/\gcd(s, x)$. Hence, the worst case of 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, $(s \leq x)$, each block contains at least one active element. In such a condition, the first array element in a block obtained by the above calculation is exactly the first array element of that block. On the contrary, if the alignment stride is larger than the distribution block size, $(s > x)$, a block may contain no active element. The first array element of the empty block, a block contains no active element, obtained by the above calculation is a *pseudo* array element. However, the pseudo array element does not affect the correctness of our proposed approach. We shall show the fact in the following section.

$b = p$; $nelem = 0$; $addr = 0$; /* initialization */
$N_{pb}^{p} = \frac{N_c}{\gcd(N_c, P)}$
DO $occurrence = 0$ **TO** $(N_{pb}^{p} - 1)$
  $c = b \mod N_c$ /* the class number of block b */
  $C_g^p(occurrence) = \lfloor \frac{b}{N_c} \rfloor * \mathcal{A}_c + \mathcal{A}_f(c)$
  $C_n^p(occurrence) = \mathcal{A}_n(c)$
  $C_l^p(occurrence) = addr + nelem$
  $b = b + P$ /*the next block on processor p */
  $nelem = C_n^p(occurrence)$ /* keep the current value for the next C_l^p */
  $addr = C_l^p(occurrence)$ /* keep the current value for the next C_l^p */
ENDDO

Fig.7. Compression_Table_Generation algorithm.

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 data-processor mappings differ only in the alignment offset, to simplify discussion, we first consider a two-level data-processor mapping without considering the pseudo active elements. That is, we assume $o < s$. General case is discussed next.

Special Case of Two-Level Data-Processor Mapping. As previously stated, the data distribution among different data distribution patterns are identical except the indices of the corresponding elements. Let the difference of the global indices of two corresponding array elements on two contiguous data distribution patterns be *global indexing offset*. Therefore, if one 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 most efficient 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*, and then, for the following data distribution patterns, the compressed local array can be generated according to previously generated compressed local array and the *global indexing offset*. Consider the two-level data-processor mapping shown in Fig. 2(a) and 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 $A^{p_0}(0, 1, 7, 13, 14, 20, 21, 27, 33, 34)$.

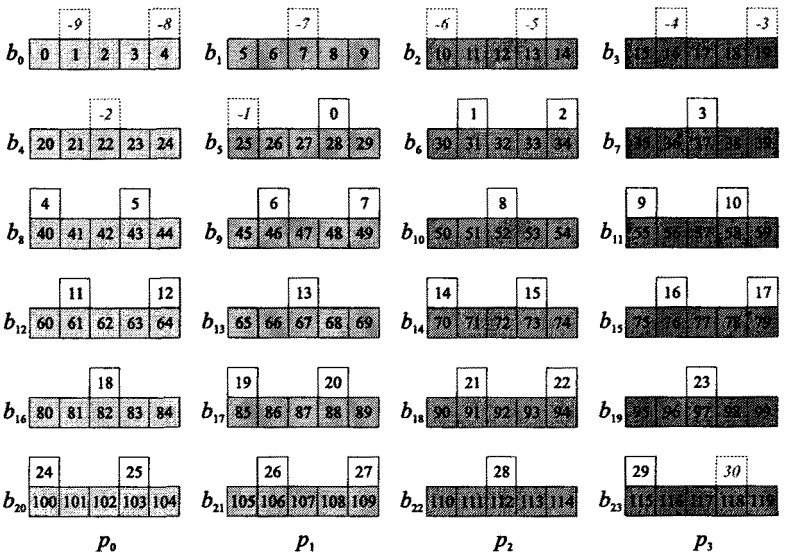


Fig.8. A general two-level data-processor mapping. Array $A(i)$ is aligned with template $T(3 * i + 28)$ and the template is then distributed onto 4 processors with $cyclic(5)$ distribution, assuming $N_A = 30$.

General Case of Two-Level Data-Processor Mapping. The concept used by the special case of two-level data-processor mapping is the same for the general case of two-level data-processor mapping. However, the pseudo active elements are taken into consideration for the general case of two-level data-processor mapping. For example, Fig. 8 is 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 at a stride 3 and an offset 28. Template T is distributed onto 4 processors using $cyclic(5)$ distribution. The two data-processor mappings shown in Figs. 2(a) and 8 are isomorphic since the alignment strides, distribution block sizes and the numbers of processors of two mappings are identical, and the alignment offsets $1 \equiv 28 \pmod 3$. The *class table* and *compression table* used by the data-processor mapping shown in Fig. 2(a) are the same for that in Fig. 8. The compressed local array of processor p_0 in Fig. 2(a) is $A^{p_0} = A(0, 1, 7, 13, 14, 20, 21, 27, 33, 34)$, which has been introduced in previous section. However, in Fig. 8, taking the pseudo active elements into consideration, the compressed local array is turned to $A^{p_0} = A(4, 5, 11, 12, 18, 24, 25)$.

In general case, we first evaluate the number of array elements allocated on processor p. Let the number of compressed local array elements on processor p be N_A^p. To evaluate N_A^p, the number of pseudo active elements allocated on processor p is calculated. The evaluations of the *global* and *local* indexing offsets are also important for general two-level data-processor mapping, where the *global* and *local indexing offsets* are respectively the differences of the global indices and the local indices in compressed local array for two corresponding array

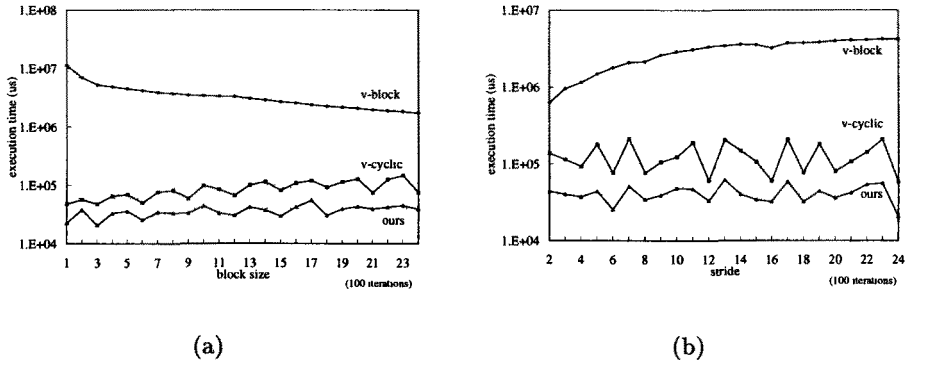
elements on two contiguous data distribution patterns. Let $\mathcal{A}_{ptn}$ be the number of active elements in a data distribution pattern and $\mathcal{A}^p_{ptn}$ be the number of active elements allocated onto processor p in a data distribution pattern. Obviously, $\mathcal{A}_{ptn}$ and $\mathcal{A}^p_{ptn}$ are the *global* and the *local* indexing offsets, respectively. For example, in the two-level data-processor mapping shown in Fig. 8, the *global* and the *local* indexing offsets are 20 and 5, respectively. The *global* and *local* indexing offsets can be used for generating the compressed local array elements for the repeated patterns as follows. If an array element gl is mapped to the compressed local array at loc, the array element $gl + \mathcal{A}_{ptn}$ will be mapped to the compressed local array at $loc + \mathcal{A}^p_{ptn}$. Thus we can generate the compressed local array for the first $\mathcal{A}^p_{ptn}$ elements by using the *compression table* and the number of pseudo active elements. After that, we can generate the next $\mathcal{A}^p_{ptn}$ elements according to the previous $\mathcal{A}^p_{ptn}$ elements and the *global indexing offset*, $\mathcal{A}_{ptn}$, until the last ($N^p_A \mod \mathcal{A}^p_{ptn}$) elements. Finally, we can generate the last ($N^p_A \mod \mathcal{A}^p_{ptn}$) elements accordingly. Detailed algorithms and implementation issues for the special and the general cases please refer to [10].

4 Experimental Results

In this section, experimental results to evaluate the performance of our proposed scheme and the work proposed in [8] are presented. Performing hole compression for two-level data-processor mapping is experimented. In the experiment, three methods are compared. Two are virtual processor schemes proposed in [8] and the last is the scheme proposed in this paper. The virtual processor scheme includes *virtual block* and *virtual cyclic* approaches. The former is termed virtual block scheme and is denoted as *v-block* in the experiment. The latter is termed virtual cyclic scheme and is denoted as *v-cyclic* in the experiment. As for our proposed scheme, we denote it as *ours* in the experiment.

The experiments are performed on a DEC Alpha 3000/400 workstation. Since the major factors affecting hole compression are the block size and the alignment stride, we fix all parameters except the alignment stride and the block size while we measure the execution times for each methods. In this experiment, the number of array elements N_A, the number of processors P, and the alignment offset o are fixed on 50000, 16, and 0, respectively. The experiments estimate the execution time of generating the compressed local array for some processor. The tested processor number is randomly generated by a random number generator. In the experiment, times are measured by CPU time and the time unit used is microsecond. The execution time is the accumulated execution time of 100 iterations.

Fig. 9(a) illustrates the performance comparisons of the three methods when the alignment stride is fixed on 12 and the block size varies from 1 to 24. In Fig. 9(a), x-axis is the block size and y-axis is the accumulated execution time. The proposed method outperforms the two virtual processor approaches, especially over the virtual block approach. In Fig. 9(a), the execution time of the virtual block approach is decreasing as the block size is increasing. It is be-



(a) (b)

Fig.9. Performance comparisons of the three methods. (a) Performance comparisons of the three methods when the alignment stride $s = 12$ and the block size varies from 1 to 24. (b) Performance comparisons of the three methods when the block size $x = 12$ and the alignment stride varies from 2 to 24.

cause 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 $\gcd(P * x, s)$ [8]. As a result, the execution time of the virtual cyclic approach is inversely proportional to $\gcd(P * x, s)$. The experiments also verifies the phenomenon. This fact revealed in Fig. 9(a) is that the execution time vibrates according to the value of $\gcd(P * x, s)$. There is no regular pattern with 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 is, the more time it takes. Hence, the execution time of ours is proportional to the number of occurrences, just as Fig. 9(a) shows.

On the other hand, Fig. 9(b) shows the performance comparisons of the three methods when the block size is fixed on 12 and the alignment stride varies from 2 to 24. In such a situation, since the number of array elements is fixed on 50000 and the template cells will cover all of array elements, thus the template cells will increase as the alignment stride increases. Hence, for 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. 9(a), the execution time of the virtual cyclic approach is inversely proportional to $\gcd(P * x, s)$ and that of ours is directly proportional to the number of occurrences, N_{pb}^{p}.

Obviously, our proposed scheme also outperforms over the two virtual processor approaches when the block size is fixed and the alignment stride varies.

One more significant result of ours over the two virtual processor approaches, in addition to the 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, the execution time of ours is very stable even though the alignment stride or the block size is changed from small value to large value. More experimental results please refer to [10].

5 Related Works

In recent years, numbers of researchers paying their attention on compiling array statements or array redistribution take only one-level mapping into consideration [5, 7, 11, 12, 13, 15]. However, a complete parallelizing compiler should take affine alignment into consideration as well. Nevertheless, affine alignment wastes a lot of memory space if the alignment stride is non-unit. Such a wastage of memory usage is unacceptable for limited local spaces of processors on distributed-memory multicomputers. Allocating spaces only for useful template cells is, therefore, of critical importance for distributed-memory multicomputers.

Gradually, a number of researchers have been aware of this fact and propose methods to compress holes for compiling two-level data-processor mapping with non-unit 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 are considered in [3]. Both identical alignment and affine alignment with hole compression are addressed. A finite state machine(FSM) approach is adopted to traverse the local index space of each processor. The construction of state table involves solving k linear Diophantine equations and a sorting operation. Moreover, the FSM approach is a runtime technique. High runtime overhead to enumerate local memory access sequences will be involved.

The work improving the FSM approach [3] is proposed in [9]. Efficient FSM table generations are proposed. The improved work enumerates the local memory access sequences by viewing the accessed elements an integer lattice. The sorting step in [3] is avoidable in the improved work. However, runtime resolution of Diophantine equations is also required.

In [8], the authors proposed the virtual processor approaches. They proposed hole compressions for block and cyclic distributions. Accordingly, hole compression for block-cyclic distribution can also be derived. However, in addition to the disadvantages mentioned in Section 4, holes can not be totally eliminated by the two virtual processor approaches. Moreover, the virtual cyclic approach can not preserve the order of compressed local array elements.

The work similar to the virtual processor approach is presented in [14]. In [14], row-wise and column-wise scanning the index space are proposed. One

corresponds to the virtual block approach and the other to the virtual cyclic approach. They can also apply to affine alignment with hole compression. Based on scanning polyhedra, an approach for enumerating local memory access sequences and generating communication sets is proposed in [1]. The method will cause a significant overhead.

In this paper, a new approach is proposed to compress holes for compiling two-level data processor mappings. The proposed approach is also a table-based approach. However, the approach need not solve k linear Diophantine equations and has no sorting operation. Furthermore, the proposed approach has less runtime overhead. In Section 4, we have compared our method with the method proposed in [8] extensively. Experimental results also verify the advantages of our proposed approach. Moreover, the proposed approach has high stability against existing methods. The execution time varies a little with the alignment stride and the distribution block size. In addition, from implement viewpoint, the proposed approach can be easily implemented as well.

6 Conclusions

Data-parallel languages support two-level data-processor mapping for user to specify data distribution. However, non-unit alignment stride always causes a lot of memory holes, even for a small alignment stride. Holes result in not only memory wastage but also performance degradation. Therefore, this paper presents compilation techniques to do with the problem. The paper uses *class table* and *compression table* to facilitate the generation of compressed local array for each processor. *Class table* is used for recording the attributes of blocks in a class cycle and *compression table* is used for recording the attributes of blocks in a data distribution pattern on a processor. The time complexities of the constructions of these two tables are $O(s)$ in 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 of our approach varies a little when the alignment stride or the distribution block size are increasing. As for the implementation issue, the proposed method is easy-implement. Experimental results do confirm the advantages of our methods over the existing methods.

On the other hand, the compilations of array statements and data redistribution are very important for parallelizing compilers on distributed-memory multicomputers. However, performing array statements or data redistribution incurs indexing overhead and communication overhead. How to alleviate the overheads resulted from performing array statements or data redistribution becomes critical important for distributed-memory multicomputers. Hence, the future focuses are on how to efficiently generate communication sets for performing 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*, pages 117–132, Delft, The Netherlands, December 1993.
2. B. M. Chapman, P. Mehrotra, and H. P. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1), August 1992.
3. 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. *Journal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.
4. 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.
5. 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(2):155–172, February 1996.
6. High Performance Fortran Forum. *High Performance Fortran Language Specification*, November 1994. (Version 1.1).
7. S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase redistribution: A communication-efficient approach to array redistribution. Technical Report OSU-CISRC-9/94-52, Department of Computer and Information Science, The Ohio State University, 1994.
8. 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(2):237–247, November 1996.
9. K. Kennedy, N. Nedeljković, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proceedings of ACM International Conference on Supercomputing*, pages 180–184, July 1995.
10. K.-P. Shih, J.-P. Sheu, and C.-H. Huang. Table-lookup approach for compiling two-level data-processor mappings in HPF. Technical Report NCU-PPCTL-1997-05, Department of Computer Science and Information Engineering, National Central University, Taiwan, 1997.
11. J. M. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21:150–159, 1994.
12. 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.
13. A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing*, 38(2):188–203, November 1996.
14. 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. Technical Report CP-96-001, Computational Physics Section, Faculty of Applied Physics, Delft University of Technology, 1996.
15. W.-H. Wei, K.-P. Shih, and J.-P. Sheu. Compiling array references with affine functions for data-parallel programs. To be appeared in *Journal of Infromation Science and Engineering*, 1997.