Efficient Address Generation for Affine Subscripts in Data-Parallel Programs

Kuei-Ping Shih
Department of Computer Science and Information Engineering
National Central University
Chung-Li 32054, Taiwan
Email: steven@axp1.csie.ncu.edu.tw

Jang-Ping Sheu
Department of Computer Science and Information Engineering
National Central University
Chung-Li 32054, Taiwan
Email: sheujp@csie.ncu.edu.tw

Chih-Yung Chang
Department of Information Science
Tamsui Oxford University College
Tamsui, Taipei, Taiwan
E-mail: changcy@jupiter.touc.edu.tw

Abstract

This paper presents an efficient compilation technique to generate the local memory access sequences for block-cyclically distributed array references with affine subscripts in data-parallel programs. For the memory accesses of an array reference with affine subscript within a two-nested loop, there exist repetitive patterns both at the outer and inner loops. We use tables to record the memory accesses of repetitive patterns. According to these tables, a new start-computation algorithm is proposed to compute the starting elements on a processor for each outer loop iteration. The complexities of the table constructions are $O(k+s_2)$, where k is the distribution block size and s_2 is the access stride for the inner loop. After tables are constructed, generating each starting element for each outer loop iteration can run in $O(1)$ time. Moreover, we also show that the repetitive iterations for outer loop are $Pk/\gcd(Pk, s_1)$, where P is the number of processors and s_1 is the access stride for the outer loop. Therefore, the total complexity to generate the local memory access sequences for a block-cyclically distributed array with affine subscript in a two-nested loop is $O(\frac{Pk}{\gcd(Pk,s_1)} + k + s_2)$.

1 Introduction

Distributed-memory multicomputers are widely used for applications in scientific and engineering fields. However, programming on multicomputers is a vital disadvantage to multicomputers owing to the absence of a global shared memory. Fortunately, data-parallel languages, such as Fortran D, Vienna Fortran and High Performance Fortran (HPF), provide a global name space and data distribution directives for programmers to specify the data placement on distributed-memory multicomputers. Although data-parallel languages make programming on distributed-memory multicomputers much easier, the tasks to distribute computation and data onto processors and to manage communication among processors are left to parallelizing compilers. Hence, the efficiency of parallelizing compilers is the key factor affecting the performance on distributed-memory multicomputers.

Generally speaking, data-parallel languages support three regular data distributions: *block*, *cyclic*, and *block-cyclic* data distributions. The address generation problems for compiling array references with *block* or *cyclic* distributions have been studied thoroughly [5]. The more general problems for compiling array references with *block-cyclic* distribution also have been studied extensively [2, 4, 6, 10, 11]. Recently, several efforts on compiling array references with affine array subscripts are proposed [1, 3, 4, 7, 11]. Affine array subscript means the array subscript is a linear combination of multiple induction variables (MIVs). In [1], the authors use a linear algebra framework to generate communication sets for affine array subscripts. Complex loop bounds and local array subscripts of the generated code will incur significant overhead. A table-based approach is proposed in [11]. The authors

classify all blocks into classes and use a class table to record the memory accesses of the first repetitive pattern. By using the class table, they derived the communication sets for non-local accessed data among processors. Both [1] and [11] are addressing the compilation of array references with affine subscripts within a multi-nested loop. However, these methods are not efficient enough, especial for dealing with the case within a two-nested loop.

In [8], they have made an empirical study of program characteristics that are important to parallelizing compiler writers. The report shows that one-dimensional array references account for 56 percent among array references examined and 60 percent are affine subscripts for one-dimensional array references checked. Moreover, two-nested loops are also very common in real programs. Therefore, in a two-nested loop, one-dimensional array references with affine subscripts should be paid more attention.

For compiling array references with affine subscripts, some researchers pay their attention on the array reference enclosed within a two-nested loop to find a better result [3, 4, 7]. Based on FSM approach [2], Kennedy et al. proposed another approach to solving the compilation of array references with affine subscripts within a two-nested loop [3, 4]. They proposed an $O(Pk)$ algorithm to find the local starting element on a processor, where P is the number of processors and k is the distribution block size. For the global starting element, they found that the repetitive iterations for the outer loop are Pk iterations. Hence, the total complexity to generate the local memory access sequence for an array reference with affine subscript within a two-nested loop is $O(P^2k^2)$. On the other hand, Ramanujam et al. proposed an improved work to find the local starting elements on each processor [7]. Since a traverse step is incurred, the complexity of their proposed algorithm is $O(k)$. Thus the total complexity of Ramanujam's algorithm is turned out to be $O(Pk^2)$.

In this paper, we propose a new and more efficient algorithm to find the local starting element. A preprocessing step is required before we compute the starting elements. The complexity of the preprocessing step is $O(k + s_2)$, where s_2 is the access stride for the inner loop. After preprocessing step is done, the time complexity to generate each starting element on a processor just needs $O(1)$. In addition, we also find that the outer loop repetitive iterations are $Pk/\gcd(Pk, s_1)$ iterations, where s_1 is the access stride for the outer loop. Therefore, the total complexity of our proposed approach is $O(Pk/\gcd(Pk, s_1) + k + s_2)$, which is asymptotical to $O(Pk + s_2)$. Strictly speaking, our proposed approach is better than the existing methods when

```
!HPF$ PROCESSORS PROC(P)
!HPF$ DISTRIBUTE A(cyclic(k)) ONTO PROC
...
do i1 = 0, n1
   do i2 = 0, n2
      A(s1*i1 + s2*i2 + o) = ...
   enddo
enddo
```

Fig. 1: HPF-like program model considered in the paper.

$s_2 < Pk^2$. In general, the inner loop access stride s_2 is much smaller than the value of Pk. Hence, the term s_2 can be omitted. Thus, we may say that the proposed algorithm is an $O(Pk)$ algorithm. As a result, the proposed approach is much efficient against the existing methods.

The rest of the paper is organized as follows. Section 2 formulates the problem and describes the conventional techniques to generate local memory access sequences for compiling the array references with affine subscripts within a two-nested loop. An efficient approach to finding the starting elements from a given global start is proposed in Section 3. The performance analyses and comparisons with the existing work are demonstrated as well. Section 4 concludes the paper.

2 Address Generation for Affine Subscripts

Compiling array references with block-cyclic distributions to generate an efficient SPMD (Single Program Multiple Data) code is one important and necessary phase in a parallelizing compiler. The address generation problem is quite complex especially when array references involve multiple induction variables (MIVs). In this section, we deal with the problem of generating local memory access sequences for compiling array references with multiple induction variables. We first describe the problem and then propose an efficient technique to solve the problem.

2.1 Problem Formulation

Specifically, Fig. 1 illustrates the program model considered in this paper. Array A is distributed onto P processors with $cyclic(k)$ distribution. The array reference contains two induction variables i_1 and i_2. The access strides of the array reference with respect to i_1 and i_2 are s_1 and s_2, respectively. The access offset of

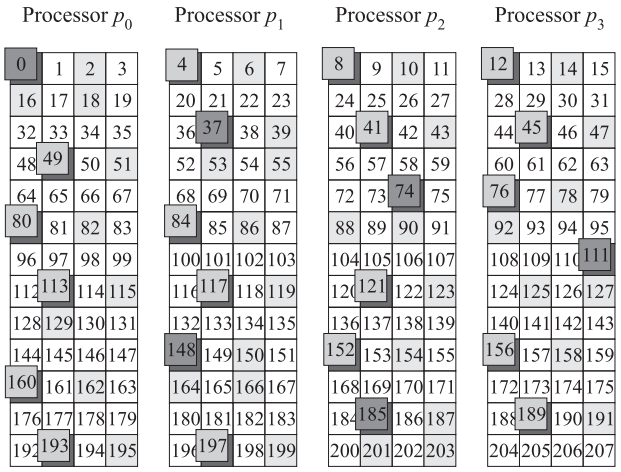

Fig. 2: An MIV address generation example, where $P = 4$, $k = 4$, $s_1 = 37$, $s_2 = 2$, $o = 0$, and $n_2 = 9$.

the array reference is o. Fig. 2 is an example amenable to the program model shown in Fig. 1, where $P = 4$, $k = 4$, $s_1 = 37$, $s_2 = 2$, $o = 0$, and $n_2 = 9$. The gray-colored elements are the array elements accessed by the array reference in the two-nested loop. The MIV address generation problem is to generate the local addresses of these gray-colored elements for some processor.

Although the example is very uncommon, for comparison, we use the same example with [3, 7]. Actually, the values of s_1 and s_2 make no difference with the difficulty of the problem. There may exist output dependences in the program model, the proposed method can generate the local memory access sequence in order without breaking the execution ordering.

2.2 Table-Based Address Generation for Affine Subscripts

Consider the program model shown in Fig. 1. For each outer loop iteration, the MIV address generation problem is reduced to an SIV address generation problem. Thus we can utilize the FSM approach [2] to generate the local memory access sequence for that SIV problem. Generating the local memory access sequence for an MIV problem can, therefore, be easily solved by enumerating the local memory access sequence for each outer loop iteration until reaching the outer loop bound.

For example, consider the example illustrated in Fig. 2. Let $i_1 = 0$. Thus, we can just focus our attention only on the inner loop. The MIV address generation problem is reduced to the SIV problem, i.e., to generate the local addresses of the accessed elements for the array reference $A(2i_2)$. Thus a finite state machine (FSM) can be built to enumerate the local memory access sequences for the SIV problem. The initial state of the FSM depends on the position of the starting array element in a block. For instance, when $i_1 = 0$, the starting element on processor p_0 is 0 and its position in a block is 0, thus the initial state of the FSM for the case when $i_1 = 0$ is at state 0. In addition to the initial state of the FSM, we also need to know the local address of the starting element since FSM only records the local memory gaps between successive array elements allocated on the processor. FSM has no enough information to show where to start in terms of local address. For example, when $i_1 = 0$, the local address of the starting element 0 on processor p_0 is 0. It means that, to use the FSM to generate the local memory access sequence for the case of $i_1 = 0$, the initial state of the FSM is at state 0 and the beginning of the sequence starts from 0. Therefore, when $i_1 = 0$, the local memory access sequence for processor p_0 is $0, 2, 4$, and so on. Similarly, it is done likewise for each outer loop iteration $i_1 = 1, 2, 3, \cdots, n_1$.

In fact, there is no need to iterate all of the outer loop iterations from 0 to n_1. We have found out that iterating $Pk/\gcd(Pk, s_1)$ outer loop iterations is enough because there is a repetitive pattern for the outer loop. Having this discovery can save a lot of time due to the avoidance of recomputation for repetitive patterns. Moreover, it can also reduce the table size which is used for recording the starting elements for outer loop iterations. The following theorem demonstrates that the repetitive period of the outer loop is $Pk/\gcd(Pk, s_1)$ iterations. For the sake of space limitation, we omit the proofs. The details please refer to [9].

Theorem 1 *For the program model shown in Fig. 1, the memory accesses of the array reference have a repetitive pattern for the outer loop and its repetitive period is* $Pk/\gcd(Pk, s_1)$ *iterations.* □

According to the above description, evidently, determining the local address of the starting element for each outer loop iteration is the primary step to solve the MIV address generation problem. The problem to find the local address of a starting element for each outer loop iteration will be described in the next section. A new approach to generating the local addresses of the starting elements will be presented in the next section as well.

3 Generating Starting Elements for $s > k$

It is obvious that for a given outer loop iteration the memory accesses just depend on the inner loop access stride s_2. Therefore, in this section, we use s to indicate the inner loop access stride s_2 except otherwise notified. The method to find the starting elements in case of $s \leq k$ can be found in [3, 7]. Both of them are $O(1)$ in complexity. However, their methods to find the starting elements in case of $s > k$ are $O(Pk)$ and $O(k)$, respectively. We propose a new method to find the starting elements in case of $s > k$ and the time complexity of the algorithm is $O(1)$. The case of $s > k$ occurs very often. In many real programs, *cyclic* distribution is usually used for load-balance consideration. However, *cyclic* distribution is a special case of a block-cyclic distribution (*cyclic*(1) distribution) and the distribution block size is 1. Therefore, the access strides are always larger than the distribution block size (1). Consequently, the problem in case of $s > k$ deserves to be paid more attention. The problem and its solution are described as follows.

3.1 Problem Description

We formally describe the induced problem as follows. Let the initial accessed element for some fixed outer loop iteration be a global start and $\mathcal{G}$ denote the local address of the global start. Specifically, given a global start $\mathcal{G}$, the processor p where $\mathcal{G}$ is allocated and the processor q which we would like to find its starting element, the problem is to figure out $\mathcal{S}_q$, the local address of the starting element, for processor q. For example, consider the example shown in Fig. 2. The gray-colored elements are the elements accessed by the array reference, in which the deep-colored shaded elements are the global starts corresponding to every outer loop iteration and the light-colored shaded elements on each processor are the starting elements corresponding to every global start. Suppose a given global start is 37 whose local address is 9 on processor p_1. The starting elements on processors p_0, p_2, and p_3 are 49, 41, and 45, respectively, in terms of global addresses. The problem is to figure out the local addresses of these starting elements. That is, 13, 9, and 9, respectively. Finally, we want to build a table to record the local addresses for those shaded elements on processor q.

Due to the space limitation, in this paper, we only present the case that the access stride s is relatively prime to the distribution block size k. That is, $\gcd(s, k) = 1$. The approach can be easily extended to the general case by slightly modification. For the general solution, please refer to [9].

3.2 Preprocessing

Given a global start $\mathcal{G}$, we propose a new approach to find the local address of the starting element $\mathcal{S}_q$ for processor q in case of $s > k$. Since the proposed approach is a table-based approach, it is necessary to precompute a few tables in order to evaluate the starting elements for a given global start. In this section, we describe the characteristics of these tables and how they are used in the proposed approach. The constructions of these tables are omitted in the paper. The details please refer to [9]. The complexities in time and space to construct and store the tables will be analyzed in Section 3.4.

3.2.1 C2P and P2C Tables

As well-known, there is a repetitive pattern for the accessed elements on blocks. By [11], all blocks can be classified into $\frac{s}{\gcd(s,k)}$ classes according to the positions of the accessed elements on a block[1]. Note that blocks of the same class have the same format. Let $C = \frac{s}{\gcd(s,k)}$. A repetitive pattern contains blocks from class 0 to class $C - 1$. In addition, since $s > k$, there is at most one accessed element on a block. Therefore, we can use a table to record the position of the only accessed element for every class. The blocks with no accessed element are recorded by "–". We denote the table C2P table. With the table we can easily and efficiently get the position of an accessed element on a block from the class number of the block.

Take Fig. 3 as an example, in which it assumes that array elements are distributed over 4 processors with *cyclic*(4) distribution and the access stride is 5. Without loss of generality, the access offset is set to 0 for simplifying discussion. The accessed elements on classes 0, 1, 2, and 3 are at positions 0, 1, 2, and 3, respectively. Therefore, the values of C2P(0), (1), (2), and (3) are 0, 1, 2, and 3, respectively. Moreover, there is no accessed element in class 4. So, C2P(4)="–". Thus we can obtain the C2P table for this example and it has been shown in Fig. 4(a).

We can get the position of an accessed element on a block according to the class number of a block by using C2P table. By contrast, if the position of the accessed element on a block is given, can we get the class number of that block efficiently? Intuitively, we can get the class number of a block according to the position of the

[1] All blocks can be numbered in terms of class according to the rule: $b \mod C$, where b is the block number of that block and C is the number of classes.

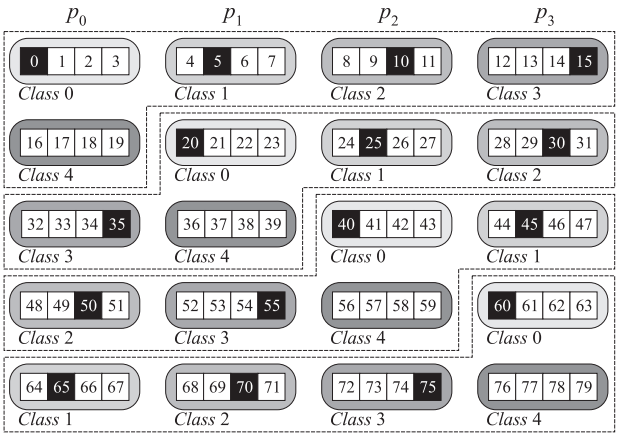

Fig. 3: An SIV example assuming that array elements are distributed onto 4 processors with $cyclic(4)$ distribution and the access stride of the array reference is 5.

(a)	C2P	0	1	2	3	–
(b)	P2C	0	1	2	3	
(c)	ACT	0	1	2	3	3
(d)	JUMP	0	0	0	0	1

Fig. 4: Tables used in starting elements findings for the example shown in Fig. 3.

accessed element on the block by means of C2P table. However, it requires a search operation. Thus, we use a table to record the class number according to the position of the accessed element on a block. With the table we can get the class number of a block according to the position of the accessed element on that block directly and efficiently.

Since a block can have at most one accessed element in case of $s > k$ and the blocks with the same position of the accessed element are classified into the same class, thus a position can have at most one class number to correspond to. As a result, it is feasible to use a table to record the corresponding class number by a given position of an accessed element. Let the table recording the class number according to the position of the accessed element on a block be P2C table. For example, we explain the P2C table for the example shown in Fig. 3. Obviously, for positions 0, 1, 2, and 3, the corresponding class numbers are 0, 1, 2, and 3, respectively. Consequently, P2C=(0, 1, 2, 3), which is shown in Fig. 4(b). It is worth mentioning that since we have assumed that $\gcd(s, k) = 1$, it is sure that each position has an accessed element to map to. Thus, each position has a class number to correspond to. However, it is not true any more if $\gcd(s, k) \neq 1$. It should be paid more attention when we are dealing with the general problem.

3.2.2 ACT and JUMP Tables

As previously described, a block contains at most one accessed element when the access stride is larger than the block size. Thus, we name a block which has an accessed element to map to as an *active* block; otherwise, it is termed an *empty* block. On a processor, the tables ACT and JUMP that we would like to introduce below are used for skipping over the empty blocks to an active block. One important observation here is that, from processor's viewpoint, blocks on a processor have a repetitive pattern in terms of classes. It is important to have such a discovery since we can obtain the class number of the next block on a processor from current block if the class number of the current block is known. Based on the discovery, we can use one table to record the class number of the next active block from the current block on a processor and another to record the number of empty blocks which we have to skip over to get the next active block if the current block is an empty block. The two tables are named ACT and JUMP, respectively. The rules to construct the two tables are as follows. If the current block is not an empty block, we do not need to skip any block. Thus, the value in ACT table for that block is recorded by its class number and that in JUMP table is recorded by 0. Otherwise, it implies that the current block is an empty block. Then the value in ACT table for that block is recorded by the class number of the next active block on the processor and that in JUMP table is recorded by the number of blocks that we have to skip over. If we can not find any active block, both the values in ACT and JUMP tables are recorded by "–". It is worth mentioning that the repetitive pattern of the blocks on processors will be the same except the initial block for all processors. Therefore, although ACT and JUMP tables are constructed from viewpoint of processors, these two tables do not change with different processors.

For the example shown in Fig. 3, take processor p_0 for illustration. Since the blocks of classes 0, 1, 2, and 3 are active blocks, the values of these entries in ACT table are the class numbers of their own and those entries in JUMP table records 0. On the other hand, the block of class 4 is an empty block. It needs to skip one block to the next active block, i.e., the block of class

3. Thus the fourth entry in ACT table is 3, the class number of the next active block and that in JUMP table is 1 as we need to skip one block to the next active block. As a result, for this example, ACT=(0, 1, 2, 3, 3) and JUMP=(0, 0, 0, 0, 1), which have been shown in Fig. 4(c) and (d), respectively.

3.3 The Algorithm

With these tables we can evaluate the starting element $\mathcal{S}_q$ from a given global start $\mathcal{G}$ in $O(1)$ time complexity. Fig. 5 illustrates the algorithm to evaluate the starting element from a given global start. We term the algorithm Start_Computation algorithm.

Algorithm: Start_Computation algorithm for the case of $s > k$.
Input: $\mathcal{G}$, a global start,
p, the processor where the global start is allocated,
q, the processor that we would like to find its starting element, where $q \neq p$
k, the distribution block size,
P, the number of processors,
s, the access stride,
C, the number of classes, where $C = \frac{s}{\gcd(s,k)}$,
C2P, P2C, ACT, and JUMP tables.
Output: $\mathcal{S}_q$, the starting element on processor q.
Assumption: $\gcd(s, k) = 1$.
Steps:

1. $pos_g = \mathcal{G} \mod k$
2. $pdist = (q - p) \mod P$
3. $c = (\text{P2C}(pos_g) + pdist) \mod C$
4. $pos_s = \text{C2P}(c)$
5. **if** pos_s = "–" **then**
6. **if** ACT(c) = "–" **then**
7. **return** no starting element on processor q
8. **else**
9. $pos_s = \text{C2P}(\text{ACT}(c))$
10. **endif**
11. **endif**
12. $dist = pos_s$ - pos_g + JUMP$(c)*k$
13. **if** $q < p$ **then**
14. $dist = dist + k$
15. **endif**
16. $\mathcal{S}_q = \mathcal{G} + dist$
17. **return** $\mathcal{S}_q$

Fig. 5: Start_Computation algorithm for the case of $s > k$.

The basic concept of the Start_Computation algorithm is as follows. The continuous blocks from processor 0 to $P - 1$ are said to be on the same *course* [2]. The fact that the corresponding entries on the blocks at the same course have the same local index is very important in Start_Computation algorithm. From the viewpoint of the global start $\mathcal{G}$, we try to figure out the distance between the starting element $\mathcal{S}_q$ and $\mathcal{G}$. With the distance we can, therefore, get the local address of the starting element by adding the distance to $\mathcal{G}$.

Start_Computation algorithm is based on the concept described above. Let's go back to the algorithm. The details of the algorithm is explained as follows. Given $\mathcal{G}$, the local address of a global start, and p where $\mathcal{G}$ is allocated, Step 1 is to calculate the position of $\mathcal{G}$ on a block, that is, pos_g. Step 2 is to measure the distance between processors p and q, which is then stored in $pdist$. In Step 3, P2C(pos_g) can get the class number of the block which the global start $\mathcal{G}$ is on. Since the blocks mapped onto processors are in a round-robin fashion in terms of classes, thus, Step 3 can get the class number of the block on processor q, which is denoted as c. According to C2P table, C2P(c) can get the position of the accessed element on the block of class c, if ever. Therefore, Step 4 can obtain the position of the starting element $\mathcal{S}_q$ on a block if it exists, i.e., pos_s. If pos_s does not equal "–", it means that the current block is an active block and pos_s denotes the position of the starting element. We can go direct to Step 12 to evaluate the distance between the starting element $\mathcal{S}_q$ and the global start $\mathcal{G}$. The distance between $\mathcal{S}_q$ and $\mathcal{G}$ is denoted as $dist$. If $q > p$, the local address of the starting element on processor q, $\mathcal{S}_q$, is equal to $\mathcal{G}$ plus $dist$, just as Step 16 shows. Otherwise, it implies that $q < p$ and we still need to add one block size to the distance since the starting element must be at one more course than the course where the global start is located. Those are what Steps 13–15 do. As a result, the local address of the starting element can be obtained, just as Step 16 shows.

On the other hand, if pos_s = "–", it means that the current block is an empty block, we can use ACT table to obtain the class number of the next active block. If ACT(c) = "–", it implies that there exists no active block on the processor. Certainly, there is no starting element on the processor. Otherwise, we can find an active block on the processor. We can get the number of blocks needed to skip over the current block to the next active block and the position of the accessed element on that active block from JUMP and ACT tables, respectively. Thus, we have Steps 5–11. For simplicity, in Step 12 the operation JUMP$*k$ is executed for all cases.

Let us take Fig. 6 as an example, where it assumes that $P = 4$, $k = 4$, $s_1 = 37$, $s_2 = 5$, $o = 0$, and $n_2 = 7$. Given an global start 37, whose local address

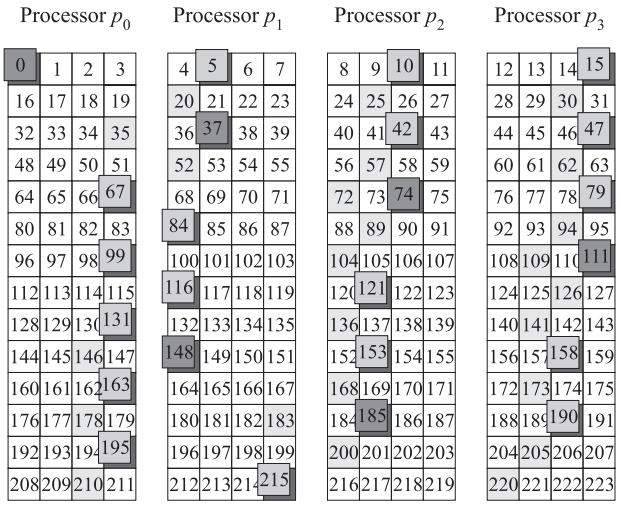

Fig. 6: Layout of array elements on processors for the case of $s_2 > k$, another MIV example, where $P = 4$, $k = 4$, $s_1 = 37$, $s_2 = 5$, $o = 0$, and $n_2 = 7$.

is 9 on processor p_1, we first find the starting element for processor p_2. The input of the Start_Computation algorithm is $\mathcal{G} = 9$, $p = 1$, $q = 2$, $k = 4$, $P = 4$, $s = 5(= s_2)$, and $C = 5(= \frac{s}{\gcd(s,k)})$. The tables used for the example are the same as shown in Fig. 4. Following the Steps from 1 to 4 in the algorithm we can obtain that $pos_g = 1$, $pdist = 1$, $c = 2$, and $pos_s = 2$. Since pos_s does not equal "–", we go direct to Step 12 and we obtain that $dist = 1$. Due to the invalidation of the condition in Step 13, we go direct to Step 16 and we have $\mathcal{S}_2 = 10$, which corresponds to the array element 42 in terms of global address.

On the same input except $q = 0$, we take the finding of the starting element on processor p_0 as another example. After executing the Step 4, we have $pos_g = 1$, $pdist = 3$, $c = 4$, and $pos_s =$"–". Since pos_s equals "–", which means that the block contains no accessed element, we go to Step 6. According to ACT and JUMP tables, there is an active block at one block after the current empty block on processor p_0. By Step 9, we have $pos_s = 3$. After Step 12, we have $dist = 6$. As $q < p$, $dist$ still needs to add 4, a block size. It turns out that $dist = 10$. Thus, $\mathcal{S}_0 = 19$, which corresponds to the array element 67 in terms of global address.

Clearly, the time complexity of Start_Computation algorithm is $O(1)$. The complexity analyses of the tables used in the algorithm and the performance comparisons against the existing methods will be discussed in Section 3.4.

Table 1: Time and space complexities analyses for tables constructions.

TABLE	Complexity	
	TIME	SPACE
C2P	$O(C)$	C
P2C	$O(C+k)$	k
ACT	$O(C)$	C
JUMP		C

Table 2: Performance comparisons of our method against the existing methods.

	Ken.'s	Ram.'s	Ours
$Comp_{s\le k}$	$O(1)$	$O(1)$	$O(1)$
$Prep.$	$O(1)$	$O(1)$	$O(s_2+k)$
$Comp_{s>k}$	$O(Pk)$	$O(k)$	$O(1)$
$Iters.$	Pk	Pk	$\frac{Pk}{\gcd(Pk,s_1)}$
TOTAL	$O(P^2k^2)$	$O(Pk^2)$	$O(\frac{Pk}{\gcd(Pk,s_1)}+s_2+k)$

3.4 Performance Analyses and Comparisons

Since the space is limited, we only give the time complexity for each table construction. Table 1 summarizes the complexities in time and space for constructing these tables. To compare with the existing methods, we denote the method proposed by Kennedy et al. as *Ken.'s*, the one proposed by Ramanujam et al. as *Ram.'s*, and our proposed one as *ours*. All the three methods (Kennedy's, Ramanujam's, and ours) are to generate the local memory access sequence for an array reference in one-level mapping with affin subscripts within a two-nested loop. Table 2 summarizes the performance comparisons of our method against the existing methods. The comparisons are made in four different time costs: the start computation time in case of $s \le k$, the preprocessing, the start computation time in case of $s > k$, and the number of iterations needed for outer loop, which are denoted respectively by $Comp_{s\le k}$, $prep.$, $Comp_{s>k}$, and $Iters.$ Clearly, our proposed approach is better than the existing methods when $s_2 < Pk^2$. However, the inner loop access stride s_2 is, in general, much smaller than the value of Pk. Hence, the dominated term would be the value of Pk. Thus, we can say that the proposed algorithm is an $O(Pk)$ algorithm. As a result, the proposed approach is much efficient against the existing methods.

4 Conclusions

In this paper, we have presented an efficient approach to the evaluation of the starting element for some processor from a given global start, which is a key step to solve the MIV address generation problem in data-parallel programs, assuming array is block-cyclically distributed and its access subscript is affine. The approach is a table-based approach. The constructions of these tables require $O(s_2 + k)$ in time complexity, where k is the distribution block size and s_2 is the access stride of the inner loop. With these tables, the Start_Computation algorithm can run in $O(1)$ time. In addition, we have shown that there exists a repetitive pattern for every $Pk/\gcd(Pk, s_1)$ outer loop iterations. Therefore, the MIV address generation problem can be solved in $O(Pk/\gcd(Pk, s_1) + k + s_2)$ time, where P is the number of processors and s_1 is the access stride of the outer loop. Currently, the best approach we ever know for this problem is $O(Pk^2)$ [7] in the literature. Hence, the proposed approach is better than the known methods if $s_2 < Pk^2$. In general, s_2 is much smaller than Pk in real applications. Thus, the dominated term would be Pk. As a result, our proposed approach is much better than the existing methods.

Since the problem model considered in the paper is focused on one-level mapping, in the near future, we would like to extend the approach to two-level mapping. Moreover, we also hope to apply the address generation approach to evaluate communication sets. It is a challenge problem since it would incur data dependences. The preservation of execution order needs the utmost care and attention. The address generation and communication sets evaluation for general affine subscripts are also under investigation.

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] 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.

[3] 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.

[4] 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*, pages 102–111, July 1995.

[5] C. Koelbel. Compile-time generation of regular communication patterns. In *Proceedings of Supercomputing'91*, pages 101–110, Albuquerque, NM, November 1991.

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

[7] J. Ramanujam, S. Dutta, and A. Venkatachar. Code generation for complex subscripts in data-parallel programs. In *Proceedings of Languages and Compilers for Parallel Computing*, Minneapolis, MN, August 1997.

[8] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1992.

[9] K.-P. Shih, J.-P. Sheu, and C.-Y. Chang. Efficient address generation for affine subscripts in data-parallel programs. Technical report, Department of Computer Science and Information Engineering, National Central University, 1998.

[10] 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.

[11] 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*, December 1998.