The Journal of Supercomputing, 17, 205–227, 2000

Efficient Address Generation for Affine Subscripts in Data-Parallel Programs

KUEI-PING SHIH kpshih@tkvr.tku.edu.tw

Department of Computer Science and Information Engineering, Tamkang University, Tamsui, Taipei, Taiwan

JANG-PING SHEU sheujp@csie.ncu.edu.tw

Department of Computer Science and Information Engineering, National Central University, Chung-Li 32054, Taiwan

CHIH-YUNG CHANG changcy@email.au.edu.tw

Department of Computer and Information Science, Aletheia University, Tamsui, Taipei, Taiwan

Final version accepted February 29, 2000

Abstract. Address generation for compiling programs, written in HPF, to executable SPMD code is an important and necessary phase in a parallelizing compiler. 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 $O(Pk/\gcd(Pk, s_1) + k + s_2)$.

Keywords: address generation, affine subscripts, data distribution, distributed-memory multicomputers, data-parallel languages, multiple induction variables (MIVs), single program multiple data (SPMD)

1. Introduction

Distributed-memory multicomputers are widely used for applications in scientific and engineering fields. However, programming on multicomputers is a vital disadvantage to such platforms owing to the absence of a global shared memory. Fortunately, data-parallel languages, such as Fortran D [5, 22], Vienna Fortran [2, 3] and High Performance Fortran (HPF) [9, 14], 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 computa-

tion 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, 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 [6, 13, 15, 22]. The more general problems for compiling array references with *block-cyclic* distribution also have been studied extensively [4, 8, 10, 12, 16, 20, 21, 23]. Recently, several efforts on compiling array references with affine array subscripts are proposed [1, 11, 12, 17, 18, 23]. 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 [23]. 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 [23] are addressing the compilation of array references with affine subscripts within a multi-nested loop. However, these methods are not very efficient, in particular for dealing with the case within a two-nested loop.

In [19], they have made an empirical study of program characteristics that are important to parallelizing compiler writers. From their report, one-dimensional array references with affine subscripts occur quite often in real programs. 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. It means that one-dimensional array references are very usual and affine subscripts in one-dimensional array references occur quite frequently. 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. The generation of local memory access sequence for one-dimensional array references with affine subscripts within a two-nested loop is very important.

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 [11, 12, 18]. Based on FSM (Finite State Machine) approach [4], Kennedy et al. proposed another approach to solving the compilation of array references with affine subscripts within a two-nested loop [11, 12]. 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 [18]. Since a traverse step is incurred, the complexity of their proposed algorithm is $O(k)$. Thus the total complexity of Ramanujam's algorithm 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 the 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, our proposed approach is better than the existing methods when $s_2 < Pk^2$. 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 more efficient than the existing methods. The approach can find the starting element, if any; otherwise, the approach can also report that there exists no starting element for that iteration on that processor. The technique proposed in this paper can be applied in compiler design for generating executable SPMD code.

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 generations of tables used in finding the starting elements are presented in Section 4. The performance analyses and comparisons with the existing work are demonstrated in Section 5. Section 6 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, Figure 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 the array reference is o. Figure 2 is an example amenable to the program model shown in Figure 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

```
!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
```

Figure 1. HPF-like program model considered in the paper.

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 [11, 18]. Actually, the values of s_1 and s_2 are not proportional to the difficulty of the problem.

2.2. *Table-based address generation for affine subscripts*

Consider the program model shown in Figure 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 [4] 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 Figure 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

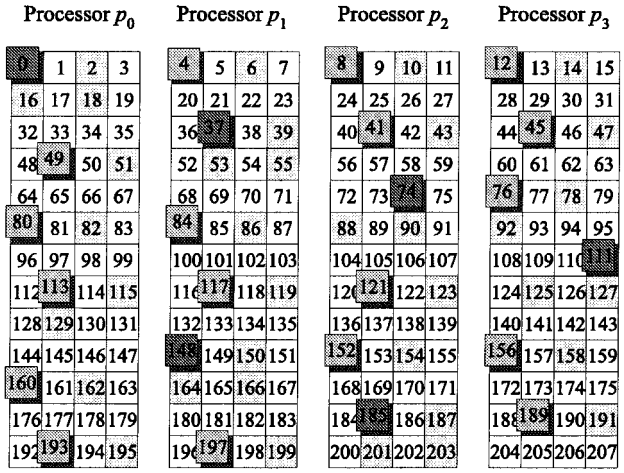


Figure 2. An MIV address generation example, where $P = 4$, $k = 4$, $s_1 = 37$, $s_2 = 2$, $o = 0$, and $n_2 = 9$.

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 $i_1 = 1, 2, 3, \ldots, n_1$.

In fact, there is no need to iterate all of the outer loop iterations from 0 to n_1. We have found that iterating $Pk/\gcd(Pk, s_1)$ outer loop iterations is sufficient 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.

Theorem 1. *For the program model shown in Figure* 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*.

Proof. Consider the program model shown in Figure 1. As well known, the memory accesses of the array reference have a repetitive pattern. Although the access offset can affect the shape of the repetitive pattern, the repetitive property does not change with the access offset. Therefore, without loss of generality, the inner loop access stride s_2 and the access offset o can be ignored when we consider the memory accesses for the outer loop iterations.

Since array A is block-cyclically distributed onto P processors with block size k, intuitively, an accessed element can be at any position of Pk. Here, we prove that only $Pk/\gcd(Pk, s_1)$ positions the accessed elements can be at, not every Pk position. In other words, the positions of the accessed elements would be repeated after $Pk/\gcd(Pk, s_1)$ iterations.

The position of an accessed element can be represented as $(s_1 i \bmod Pk)$. Suppose i and i' are two iterations that their accessed elements have the same position. We would like to show that $i - i' = tPk/\gcd(Pk, s_1)$, for some $t \in \mathbf{Z}$, where $\mathbf{Z}$ is the set of integers. As the accessed elements of i and i' have the same position, we have $(s_1 i \bmod Pk) = (s_1 i' \bmod Pk)$. That is, $s_1 i \equiv s_1 i' \pmod{Pk}$. According to [7, Equation (4.39)], we have $i \equiv i' \pmod{Pk/\gcd(PK, s_1)}$. As a result, we have $i - i' = tPk/\gcd(Pk, s_1)$, for some $t \in \mathbf{Z}$. The theorem is therefore obtained. ∎

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 can be found in [11, 18]. In case of $s \leq k$, complexity for finding starting elements in [11, 18] is $O(1)$. However, in case of $s > k$, complexities for finding starting elements in [11] and [18] 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. *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. 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 have given an overall description of the MIV address generation problem in Section 2.2. Finding the starting element on a processor from a given outer loop iteration plays an important role in dealing with the MIV address generation problem. 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 Figure 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.

For simplicity, we first describe a special case, which assumes that the access stride s is relatively prime to the distribution block size k. That is, $\gcd(s, k) = 1$. The extension to general case is presented later.

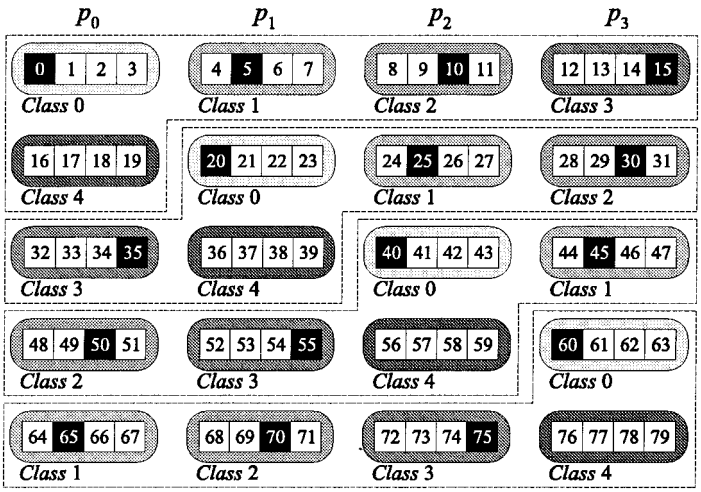


Figure 3. An SIV example assuming that array elements are distributed onto 4 processors with *cyclic*(4) distribution and the access stride of the array reference is 5.

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 pre-compute 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 will be introduced in Section 4.

3.2.1. C2P and P2C tables. As is well-known, there is a repetitive pattern for the accessed elements on blocks. By [23], all blocks can be classified into $s/\gcd(s, k)$ 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 $C = s/\gcd(s, k)$. 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.

Consider Figure 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 Figure 4(a).

(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

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

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 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, 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 such table be named P2C table. To illustrate, we explain the P2C table for the example shown in Figure 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 Figure 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 a processor's viewpoint, blocks on a processor have a repetitive pattern in terms of classes. It is important to have such a remark 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 Figure 3, consider 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 Figure 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. Figure 5 illustrates the algorithm to evaluate the starting element from a given global start. We term the algorithm Start_Computation algorithm.

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* [4]. 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. Figure 6 illustrates the basic idea of the Start_Computation algorithm. Let pos_g denote the position of $\mathcal{G}$ in a block on processor p and pos_s denote the position of $\mathcal{S}_q$ in a block on processor q. The element denoted by gray-colored $\mathcal{G}$ on processor q is the corresponding entry to $\mathcal{G}$ on the same course. Thus, the two elements have the same local index, that is, $\mathcal{G}$. As a result, if the distance between $\mathcal{S}_q$ and the gray-colored $\mathcal{G}$ is figured out, $\mathcal{S}_q$ can be obtained accordingly. Let b denote the block on processor q which is immediately greater than the block which $\mathcal{G}$ is located on. Figure 6(a) is the case that $q > p$, $pos_g < pos_s$, and b is the *active* block. Since $q > p$, b must be at the same course with the block where $\mathcal{G}$ is located. Moreover, as b is an active block, $\mathcal{S}_q$ must be on b. By the tables described in Section 3.2, we can easily obtain pos_g and pos_s. Therefore, $\mathcal{S}_q = \mathcal{G} + pos_s - pos_g$. It is similar for the other cases.

The 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

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. $dist = pos_s - pos_g + \text{JUMP}(c) * k$
13. **if** $q < p$ **then**
14. $dist = dist + k$
15. **endif**
16. $\mathcal{S}_q = \mathcal{G} + dist$
17. **return** $\mathcal{S}_q$

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

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 directly 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 $q > p$, it is the case shown in Figure 6(a). 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 this is the case shown in Figure 6(b). 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, that is, the cases illustrated in Figure 6(c) and 6(d). 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, which means that we can find an active block on the processor, we can get the number of blocks needed to skip over the current block to

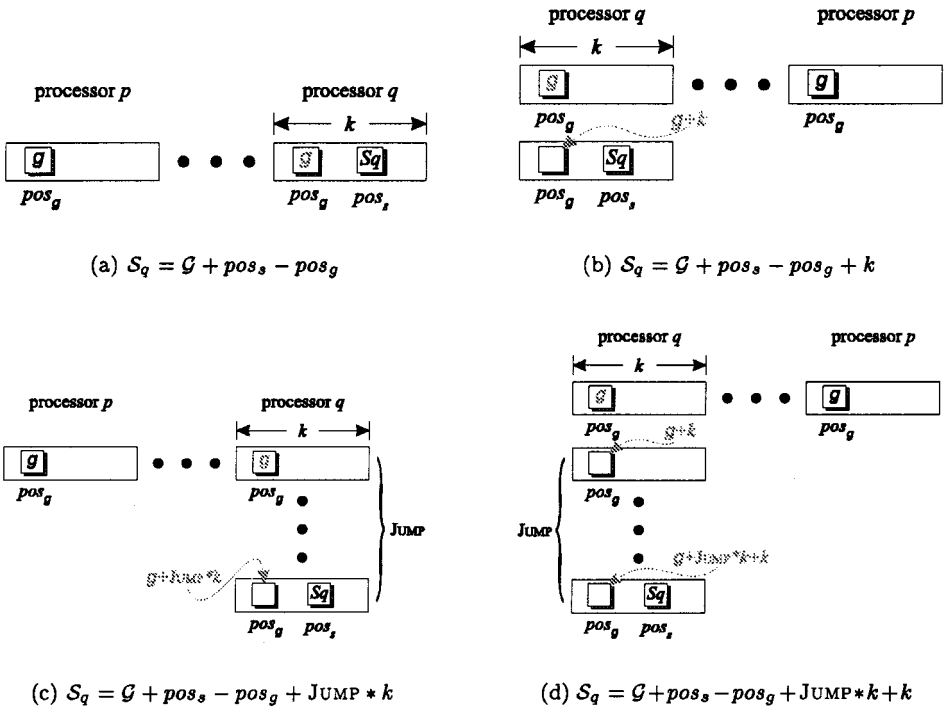


Figure 6. The basic concept of the Start_Computation algorithm. (a) The case that $q > p$, $pos_g < pos_s$, and the current block is an *active* block. (b) The case that $q < p$, $pos_g < pos_s$, and the current block is an *active* block. (c) The case that $q > p$, $pos_g < pos_s$, and the current block is an *empty* block. (b) The case that $q < p$, $pos_g < pos_s$, and the current block is an *empty* block.

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. It makes no difference for the cases shown in Figures 6(a) and 6(b) since the entry in JUMP table is 0 if the current block is an active block.

Let us consider Figure 7 as an example, where it assumes that $P = 4$, $k = 4$, $s_1 = 37$, $s_2 = 5$, $o = 0$, and $n_2 = 7$. Consider a global start 37, whose local address 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 $C = 5(= s/\gcd(s, k))$. The tables used for the example are the same as shown in Figure 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 directly to Step 12 and we obtain that $dist = 1$. Due to the invalidation of the condition in Step 13, we go directly 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 Step 4, we have $pos_g = 1$, $pdist = 3$, $c = 4$, and $pos_s =$ "$-$." Since pos_s equals "$-$," which means that the block

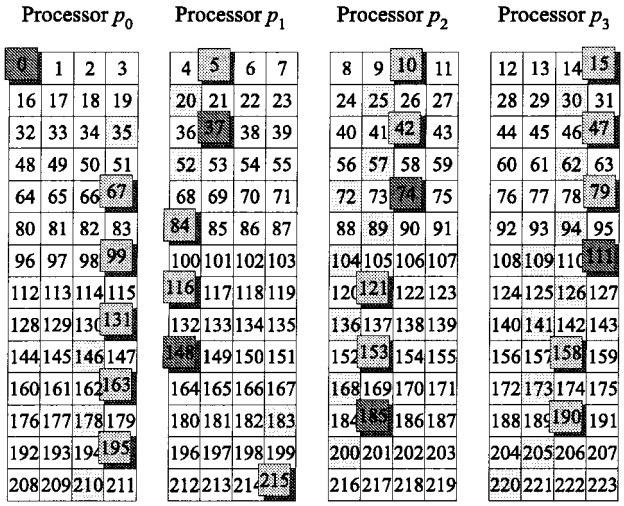


Figure 7. Layout of array elements on processors for the case of $s_2 > k$, another MIV example, where $P = 4$, $k = 4$, $s_1 = 37$, $s_2 = 5$, $o = 0$, and $n_2 = 7$.

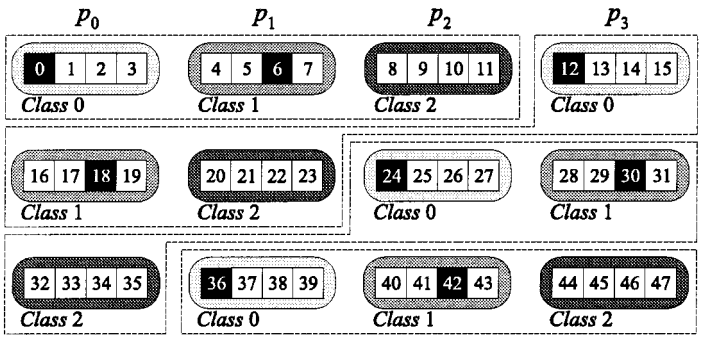
contains no accessed element, we go to Step 6. According to ACT and JUMP tables, there is an active block at one block after the current empty block on processor p_0. By Step 9, we have $pos_s = 3$. After Step 12, we have $dist = 6$. As $q < p$, $dist$ still needs to add 4, a block size. It turns out that $dist = 10$. Thus, $\mathscr{S}_0 = 19$, 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 5.

3.4. *Extension to the general case*

In previous discussions, we have described the starting element findings for the cases of $s > k$ and $\gcd(s, k) = 1$. Actually, the assumption of $\gcd(s, k) = 1$ can be removed by easily extension of the proposed approach. The extension is described as follows.

In Section 3.2.1, we have described the C2P and P2C tables. In P2C table description, we have mentioned that if $\gcd(s, k) = 1$, each position must have an accessed element to map to. However, if $\gcd(s, k) \neq 1$, each position may have no accessed element to map to. Let's take Figure 8 as an example for illustration, where Figure 8 assumes that array elements are block-cyclically distributed onto 4 processors with *cyclic*(4) distribution and the access stride s for the array reference equals 6. Figure 8(a) shows the layout of array elements on processors and Figure 8(b) illustrates the tables used for the starting element findings. In this example, all blocks can be classified into 3 classes. C2P= $(0, 2, -)$. The accessed elements only occurs at positions 0 and 2, and they are corresponding to classes 0 and 2, respectively. That is, P2C(0) = 0 and P2C(2) = 2. As for the positions 1



(a)

C2P	0	2	–	
P2C	0	0	1	1
OFFSET	0	1	0	1
NEXTACT	0	1	0	
JUMP	0	0	1	

(b)

Figure 8. 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 6. (a) Layout of array elements on processors. (b) Tables used for starting element findings.

and 3, there is no accessed element at these positions. Therefore, there is no proper class number to map to for these positions.

Actually, in our proposed approach, we let the array element $A(0)$ be a base point. The accessed elements are assumed to start from the base point and then each strides s. It implies that a block with an accessed element at position 0 is always classified into class 0, and vice versa. Therefore, the tables described above can be reused. In other words, if the block size k and the access stride s are unchanged, the classification of blocks is also unchanged. As a result, for the positions which no accessed elements is at, we let their values in P2C table be the values of their previous entries. Nevertheless, according to C2P table, each class number has its real position to correspond to. Consequently, there is a difference between the real position and the assumed position. Therefore we use another table to record the difference in order to make use of C2P table for every case. The table is denoted as OFFSET table.

Take the example shown in Figure 8 for illustration. The C2P and P2C tables for this example are $(0, 2, -)$ and $(0, 0, 1, 1)$, respectively. Since position 1 has no suitable class number to correspond to, we assign the class number corresponded by position 0 to position 1, i.e., class 0. Although position 1 corresponds to class 0, the real position of the accessed element on the block of class 0 is at position 0

according to C2P table. Thus there is a 1-difference between the assumed position and the real position. As a result, OFFSET(1) = 1. Similarly, OFFSET(3) = 1. There is no problem on positions 0 and 2 since they have their suitable class numbers to correspond to. Consequently, OFFSET table for this example is (0, 1, 0, 1).

In addition to the above modifications, the algorithm described in Section 3.3 also needed to be modified. The modification is very slight. We only need to modify the line 12 of the algorithm shown in Figure 5. It is modified to

$$12. \quad dist = pos_s - pos_g + \text{JUMP}(c) * k + \text{OFFSET}(pos_g).$$

The modification is to add the offset between the real position and the assumed position to obtain the correct distance.

In this section, we have described how to find the starting element from a given global start. The approach is table-based. Table generations are described in the following section.

4. Tables constructions

In the previous section we have described the technique to find the starting element $\mathcal{S}_q$ from a given global start $\mathcal{G}$ by means of five tables, C2P, P2C, OFFSET, NEXTACT, and JUMP tables. In this section we will present the methods to construct these tables.

4.1. C2P, P2C, and OFFSET *tables constructions*

C2P table is a table to record the position of the accessed element in every class. Figure 9 illustrates the algorithm to construct the C2P table, which is termed C2P_Construction algorithm. In this algorithm, Step 1 sets C2P(0) to 0. We use a variable p to denote the position of an accessed element on a block. Initially p is set to s, the access stride, to denote the next accessed element. Then we scan the blocks of all classes first from the block of class 1. For each block, p subtracts a block size k. If $p < k$, it implies that the accessed element is on this block and the position of this accessed element is at p. The value of this entry in C2P table is set to p. Afterward p will add another access stride s to denote the next accessed element. Otherwise ($p \geq k$), it implies that the block is an empty block. Thus the value of the entry in C2P table is set to "−." In this fashion we can obtain the C2P table. The complexity to generate the C2P table is $O(C)$, where C is the number of classes.

P2C table records the class number according to the position of the accessed element on a block. OFFSET table records the difference between the real position and the assumed position. P2C table is constructed by means of C2P table. OFFSET table is constructed by means of C2P table. However, OFFSET and P2C tables can be constructed simultaneously. Figure 10 shows the methods to construct the P2C and OFFSET tables. The algorithm is termed P2C_and_OFFSET_Constructions algorithm.

Algorithm: C2P_Construction algorithm.
Input: s, the access stride,
k, the distribution block size,
C, the number of classes, where $C = \frac{s}{\gcd(s,k)}$.
Output: C2P table.
Steps:

```
1. C2P(0) = 0
2. p = s
3. do i = 1, C − 1
4.     p = p − k
5.     if p < k then
6.         C2P(i) = p
7.         p = p + s
8.     else
9.         C2P(i) = "–"
10.    endif
11. enddo
```

Figure 9. C2P_Construction algorithm.

Algorithm: P2C_and_OFFSET_Constructions algorithm.
Input: k, the distribution block size,
C, the number of classes, where $C = \frac{s}{\gcd(s,k)}$,
C2P table.
Output: P2C and OFFSET tables.
Steps:

```
1. initialize all entries in P2C table to "–"
2. P2C(0) = 0
3. OFFSET(0) = 0
4. do i = 1, C − 1
5.     if C2P(i) ≠ "–" then
6.         P2C(C2P(i)) = i
7.     endif
8. enddo
9. count = 0
10. do i = 1, k − 1
11.     if P2C(i) ≠ "–" then
12.         OFFSET(i) = 0
13.         count = 0
14.     else
15.         P2C(i) = P2C(i − 1)
16.         count = count + 1
17.         OFFSET(i) = count
18.     endif
19. enddo
```

Figure 10. P2C_and_OFFSET_Constructions algorithm.

In this algorithm Step 1 first initializes P2C table to "–." Steps 2 and 3 set the initial values of P2C and OFFSET tables to 0. Steps 4–8 are to set the P2C table for those entries that have values in C2P table. For those entries that have not been set in previous steps, it implies that these entries have no suitable class numbers to correspond to. These entries should be set to the values of their previous entries. Therefore we scan P2C table from the beginning. We use a counter to count the distance between the real position and the assumed position. If the scanned entry

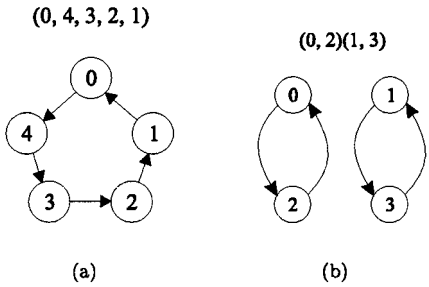


Figure 11. (a) A one-group circular sequence. (b) A multi-groups circular sequence.

has a value, it implies the entry has a corresponding class number. Thus the entry in OFFSET table is set to 0 and the counter is reset to 0. Otherwise, it implies that the position has no corresponding class number. We reset the entry to the value of its previous entry and the entry in OFFSET table is set to the value of the counter. Steps 9–19 do what we have described above. The complexity to generate the P2C and OFFSET tables is $O(C + k)$.

4.2. ACT *and* JUMP *tables constructions*

In Section 3.2.2 we have mentioned that, from a processor's viewpoint, blocks on processors have a repetitive pattern in terms of classes. Moreover, the repetitive pattern of the blocks on processors will be the same except the initial block for all processors. To explain specifically, let us take a look at the example shown in Figure 3. The blocks on processor p_0 are in classes $0, 4, 3, 2, 1$, and then repeat again from class 0. A similar situation also happens on processors p_1, p_2, and p_3. The sequence of class numbers on p_1 is $1, 0, 4, 3, 2$, that for p_2 is $2, 1, 0, 4, 3$, and that for p_3 is $3, 2, 1, 0, 4$. It is interesting that the sequence of class numbers on each processor is the same except the initial class number on each processor. That is, the sequence of class numbers on each processor can be viewed as the sequence $0, 4, 3, 2, 1$ and the initial class numbers for p_0, p_1, p_2, and p_3 are $0, 1, 2$, and 3, respectively. We use the notation $(0, 4, 3, 2, 1)$ to denote the circular sequence. The circular sequence contains only one *group*. We call the circular sequence one-group circular sequence. Figure 11(a) illustrates the one-group circular sequence for this example. By the way, the circular sequence for the example shown in Figure 8 is $(0, 1, 2)$. It is also a one-group circular sequence.

It should be addressed that it is possible that the sequence of class numbers on each processor may be different and there may be more than one group in a circular sequence. Nevertheless, all class numbers have appeared in the circular sequence. Moreover, groups are mutually disjoint and a processor can belong to one and only one group. We use the example shown in Figure 12 to illustrate the phenomenon. There are 4 classes in this example. The sequence of class numbers for p_0, p_2, and p_4 is 0, 2, and that for p_1, p_3, and p_5 is 1, 3. The circular sequence can be represented as $(0, 2)(1, 3)$. Obviously, the circular sequence is a two-groups circular sequence. One group is $(0, 2)$ and another is $(1, 3)$. $(0, 2)$ and $(1, 3)$ are

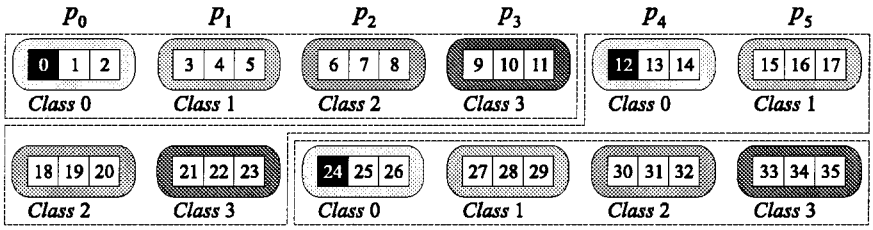


Figure 12. An SIV example assuming that array elements are distributed onto 6 processors with *cyclic*(3) distribution and the access stride of the array reference is 12.

Algorithm: Circular_Sequence_Generation algorithm.
Input: P, the number of processors,
C, the number of classes, where $C = \frac{s}{\gcd(s,k)}$.
Output: CS, a circular sequence.
Steps:

```
1.  S = {0, 1, 2, ..., C - 1}
2.  while S ≠ ∅ do
3.      c = select one element form S
4.      start a new group G beginning from c
5.      S = S - {c}
6.      c = (c + P) mod C
7.      while c ∈ S do
8.          add c to G
9.          S = S - {c}
10.         c = (c + P) mod C
11.     endwhile
12.     CS = CS + G
13. endwhile
```

Figure 13. Circular_Sequence_Generation algorithm.

mutually disjoint. Processors p_0, p_2, and p_4 belong to the group $(0, 2)$ and p_1, p_3, and p_5 belong to the group $(1, 3)$. Figure 11(b) illustrates the multi-groups circular sequence for this example.

Since the constructions of ACT and JUMP tables are based on the circular sequence, we first demonstrate the generation of a circular sequence. Figure 13 demonstrates the algorithm to generate the circular sequence. The algorithm is termed Circular_Sequence_Generation algorithm. In this algorithm we use a set to represent the classes that have not been visited. Initially, all classes are in the set. It first chooses one class to start a new group and delete the class from the set. It then finds the next class in the group by $(c + P) \bmod C$. If the class has not been visited, that is, the class belongs to the set, it adds the class to the group and deletes the class from the set. It repeats until the class has been visited. If the class has been visited, it implies that the group is finished. It adds the group to the circular sequence and then restarts from the very beginning by choosing another class in the set to begin another new group. The algorithm terminates while the set is empty. The complexity of the Circular_Sequence_Generation algorithm is $O(C)$.

On a processor, ACT table is to record the class number of the active block from the current block. JUMP is to record the number of empty blocks which we have

```
Algorithm: ACT_and_JUMP_Constructions algorithm.
Input: CS, a circular sequence,
       C2P table.
Output: ACT and JUMP tables.
Steps:
1. initialize all entries in ACT and JUMP tables to "–"
2. initialize a stack
3. for each group G in CS do
4.     repeat
5.         c = select one element from G orderly
6.         if C2P(c) ≠ "–" then
7.           ACT(c) = c
8.           JUMP(c) = 0
9.           count = 0
10.          while stack is not empty do
11.                c' = pop one element from the stack
12.                ACT(c') = c
13.                count = count + 1
14.                JUMP(c') = count
15.          endwhile
16.        else
17.          push c into the stack
18.        endif
19.    until c is the last unselected element in G
20.    if C2P(c) = "–" THEN
21.      first = the first element in G
22.      if ACT(first) ≠ "–" then
23.        count = 0
24.        while stack is not empty do
25.              c' = pop one element from the stack
26.              ACT(c') = ACT(first)
27.              count = count + 1
28.              JUMP(c') = JUMP(first)+count
29.        endwhile
30.      else
31.        empty the stack
32.      endif
33.    endif
34. endfor
```

Figure 14. ACT_and_JUMP_Constructions algorithm.

to skip over to reach an active block from the current block. The two tables can be constructed together. The constructions of the two tables are based on the circular sequence and C2P table. Figure 14 illustrates the algorithm to construct ACT and JUMP tables, which is termed ACT_and_JUMP_Constructions algorithm. We use a stack to help construct these two tables. The stack is used for storing the class numbers of the empty blocks. First of all, we initialize all the entries in ACT and JUMP tables to "−" and also initialize a stack. Afterward, we scan the circular sequence one group by one group and for each group one class by one class. If the scanned class corresponds to an empty block, the class is pushed into the stack. Otherwise, it implies the block is an active block. By the construction rules in Section 3.2.2, the corresponding entry in ACT table is set to the class number of its own and the entry in JUMP table is set to 0. Moreover, the previous classes pushed in the stack

are popped out from the stack under this condition. In other words, once we find an active block, we pop the stack one by one until the stack is empty, set each entry in ACT table for each popped class to the class number of the active block, and set each entry in JUMP table for each popped class to the order of the popped class in the stack.

The algorithm can work correctly except the last scanned class in a group corresponds to an empty block. Otherwise, it needs more consideration. If the last scanned class in a group corresponds to an empty block, we use the values in the ACT and JUMP tables of the first class in the group for the unpopped classes. If the value in ACT table of the first class is not set, it implies that we can not find an active block in the group. By the construction rules, all the entries in ACT and JUMP tables should be recorded by "$-$." In this algorithm, we just need to empty the stack since we have set them to "$-$" at the initial step. On the other hand, if the value in ACT table of the first class has ever been set, then we pop the stack also one by one until the stack is empty. For each popped class, the entry in ACT table for the popped class is set to that for the first class in the group and the entry in JUMP is set to that for the first class plus the order of the popped class in the stack.

5. Performance analyses and comparisons

In Section 3 we have described the algorithm to find the starting element from a given global start. Simultaneously, the tables used to facilitate the starting element findings are also presented. In Section 4 we have also demonstrated the methods to construct these tables. In this section we analyze the tables construction algorithms in theoretical. The comparisons of our proposed method against the existing methods are presented as well.

5.1. Performance analyses

C2P_Construction algorithm shown in Figure 9 is to construct a C2P table. Let C be the number of classes, where $C = s/\gcd(s, k)$ [23]. The algorithm terminates when all classes are visited once. Obviously, the time complexity of this algorithm is $O(C)$. As for the space, a C2P table requires C entries to store the position of the accessed element for every class.

P2C and OFFSET tables are constructed together. The algorithm to construct these two tables is shown in Figure 10. In Step 3 of this algorithm, the initial step requires $O(k)$. Steps 4–8 require $O(C)$ and Steps 10–19 require another $O(k)$. Thus the algorithm to construct P2C and OFFSET tables runs $O(C + k)$ in complexity. On the other hand, the spaces used by P2C and OFFSET are both k.

JUMP table can be constructed in conjunction with the construction of ACT table. However, to construct ACT and JUMP tables, we need first to generate a circular sequence, which has been described in Section 4.2. Circular_Sequence_Generation algorithm is to generate a circular sequence. In this algorithm we use a set structure to maintain the classes which have not been visited. The algorithm terminates

Table 1. Time and space complexities analyses for tables constructions

	Complexity	
Table	Time	Space
C2P	$O(C)$	C
P2C	$O(C+k)$	k
OFFSET		k
ACT	$O(C)$	C
JUMP		C

when the set is empty. It visits each class one and only one time. Therefore, the time complexity of Circular_Sequence_Generation algorithm is $O(C)$. Even though we use a set structure in this algorithm, it does not increase the time complexity. However, it is an implementation issue. We do not discuss the details in this paper.

ACT and JUMP tables are constructed by ACT_and_JUMP_Constructions algorithm shown in Figure 14. The algorithm scans the circular sequence one group by one group and for each group one class by one class. We have mentioned that a circular sequence contains all classes. The algorithm visits each class at most twice. As a result, the time complexity of the algorithm is $O(C)$. Both ACT and JUMP tables need C space.

Table 1 summarizes the complexities in time and space for constructing these tables.

5.2. Performance comparisons

We compare our method with two existing methods. One is proposed by Kennedy et al. [11] and another is proposed by Ramanujam et al. [18]. The method proposed by Kennedy et al. is denoted *Kennedy's*, the one proposed by Ramanujam et al. is denoted *Ramanujam's*, and our proposed method is denoted *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 affine subscripts within a two-nested loop.

As described in Section 2.2, the solution to the MIV address generation problem contains two phases. The first phase is to iterate outer loop iterations. Once the outer loop iteration is fixed, the MIV address generation problem is reduced to an SIV address generation problem. Therefore, the second phase is to solve the reduced SIV problem. In the second phase, the FSM approach [4] is adopted. However, before using the FSM approach, we need to solve another induced problem—starting element findings. We term the findings as start computation.

In Kennedy's method, the first phase needs to iterate Pk outer loop iterations. In the second phase the start computation requires $O(1)$ complexity for the case of $s_2 \leq k$. If $s_2 > k$, it needs $O(Pk)$ in time complexity. In this case they have to pre-compute two vectors (l- and r-vectors). It requires $O(1)$. Therefore, the total complexity to solve the MIV address generation problem is $O(P^2k^2)$. In

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

	Kennedy's	Ramanujam's	Ours
Start computation in case of $s \leq k$	$O(1)$	$O(1)$	$O(1)$
Preprocessing	$O(1)$	$O(1)$	$O(s_2 + k)$
Start computation in case of $s > k$	$O(Pk)$	$O(k)$	$O(1)$
Number of iterations needed for outer loop	Pk	Pk	$\frac{Pk}{\gcd(Pk, s_1)}$
Total	$O(P^2k^2)$	$O(Pk^2)$	$O\left(\frac{Pk}{\gcd(Pk, s_1)} + s_2 + k\right)$

Ramanujam's method, the first phase also needs to iterate Pk outer loop iterations. In the second phase the start computation is $O(1)$ if $s_2 \leq k$. Otherwise, $s_2 > k$, the start computation is $O(k)$ in complexity. Similarly, they also need to pre-compute the l- and r-vectors. As a result, the total complexity is $O(Pk^2)$.

In our method, by Theorem 1, the first phase needs to iterate only $Pk/\gcd(Pk, s_1)$ outer loop iterations. In the second phase, we do not propose any method to deal with the start computation while $s_2 \leq k$. However, it can be solved by adopting either Kennedy's or Ramanujam's method. Therefore, it requires $O(1)$ in complexity. In the case of $s_2 > k$, we propose a new start computation technique. The technique needs only $O(1)$ in complexity for each starting element finding. However, we need to construct some tables to facilitate the starting element findings. The tables constructions requires $O(C + k)$, where C is the number of classes and $C = s_2/\gcd(s_2, k)$. Hence the worst case of the tables constructions is $O(s_2 + k)$ in complexity. Consequently, the total complexity of our method is $O(Pk/\gcd(Pk, s_1) + s_2 + k)$.

Table 2 summarizes the performance comparisons of our method against the existing methods. 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 more efficient against the existing methods.

6. 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 complexity of the best approach for this problem is $O(Pk^2)$ [18] 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.

Acknowledgments

This work was supported by National Science Council of the Republic of China under grants NSC 89-2213-E-008-023 and NSC 89-2213-E-156-001.

Note

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

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. *Scientific Programming*, 1(1), 1992.
3. B. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortran—a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, ed., *Language, Compilers and Runtime Environments for Distributed Memory Machines*, 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. *Journal of Parallel and Distributed Computing*, 26(1):72–84, 1995.
5. 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.
6. M. Gerndt. Automatic parallelization for distributed-memory multiprocessing systems. Ph.D. thesis, University of Bonn, December 1989.
7. R. L. Graham, D. E. Knuth, and O. Patashink. *Concrete Mathematics*. Addison Wesley, Reading, Mass., 1989.
8. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32(2):155–172, 1996.

9. High Performance Fortran Forum. *High Performance Fortran Language Specification*, November 1994. (Version 1.1).
10. 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.
11. 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.
12. 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.
13. C. Koelbel. Compile-time generation of regular communication patterns. In *Proceedings of Supercomputing'91*, pp. 101–110, Albuquerque, NM, November 1991.
14. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, Mass., 1994.
15. C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, 1991.
16. S. P. Midkiff. Local iteration set computation for block-cyclic distributions. In *Proceedings of International Conference on Parallel Processing*, Vol. II, pp. 77–84, August 1995.
17. S. P. Midkiff. Computing the local iteration set of a block-cyclically distributed reference with affine subscripts. In *Proceedings of the sixth Workshop on Compilers for Parallel Computers*, Aachen, Germany, December 1996.
18. J. Ramanujam, S. Dutta, and A. Venkatachar. Code generation for complex subscripts in data-parallel programs. In *Languages and Compilers for Parallel Computing*, Minneapolis, MN, August 1997.
19. 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, 1992.
20. 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.
21. 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, 1996.
22. C. W. Tseng. An optimizing Fortran D compiler for MIMD distributed-memory machines. Ph.D. thesis, Rice University, 1993.
23. W.-H. Wei, K.-P. Shih, and J.-P. Sheu. Compiling array references with affine functions for data-parallel programs. *Journal of Information Science and Engineering*, 14(4):695–723, 1997.