JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 14, 695-723 (1998)

Compiling Array References with Affine Functions for Data-Parallel Programs

WEN-HSING WEI, KUEI-PING SHIH AND JANG-PING SHEU
Department of Computer Science and Information Engineering
National Central University
Chungli, Taiwan 320, R.O.C.
E-mail: sheujp@csie.ncu.edu.tw

An important research topic is parallelizing of compilers to generate local memory access sequences and communication sets while compiling a data-parallel language into an SPMD (Single Program Multiple Data) program. In this paper, we present a scheme to efficiently enumerate local memory access sequences and to evaluate communication sets. We use a *class table* to store information that is extracted from array sections and data distribution patterns. Given array references and data distributions, we can utilize the *class table* to generate communication sets in closed forms. Furthermore, we derive the algorithms for sending and receiving necessary data between processors. An algorithm for generating the *class table* is presented, and the time complexity of this algorithm is $O(s)$, where s is the array section stride. The technique of generating communication sets for one index variable has been implemented on a DEC Alpha 3000 workstation. The experimental results confirm the advantage of our scheme, especially when the array section stride is larger than the block size. Finally, we adapt our approach to handle array references with multiple index variables. The time complexity for constructing the whole *class table is* $O(s^2)$.

Keywords: communication set, data-parallel language, distributed memory multicomputers, HPF, parallelizing compilers, SPMD.

1. INTRODUCTION

Data-parallel languages, such as High Performance Fortran (HPF) [1] and Fortran D [3, 7], support global name spaces and provide directives for programmers to specify distributions of arrays at the language level. For a data-parallel program in which the distribution can be specified by programmers or by means of a separate compilation phase, the compiler must automatically partition the arrays according to the distributions and generate the SPMD (Single Program Multiple Data) code. Actually, while it distributes arrays over processors, the compiler also partitions the computation among processors. In general, the compiler uses the owner-computes rule to partition the computation. By means of this rule, as a computation is executed on a processor, data movement between processors is needed if a processor references an array element which is allocated on another

Received April 16, 1996; accepted December 22, 1996
Communicated by Shing-Tsaan Huang

processor. Therefore, the source processor has to determine the destination processors and find out to which array elements will be sent. The destination processor has to determine the source processors and find out which array elements will be received. The sequence of the local memory addresses where these referenced elements are allocated is called the *local memory access sequence*. The sets of data that have to be sent or received between source and destination processors are *communication sets* [2, 6, 9].

Consider an array reference of the form $A(l_1 : u_1 : s_1) = f(X(l_2 : u_2 : s_2))$ with arrays A and X both distributed in block-cyclic distributions, where f is a function of array reference $X(l_2 : u_2 : s_2)$. Paalvast *et al.* [11] proposed a scheme based on scanning the indices of referenced elements to determine the elements which need to be communicated. Chatterjee *et al.* [2] proposed an approach to solve the local memory access sequence problem in terms of a finite state machine. Hiranandani *et al.* [8] provided a method that works in $\Omega(t)$ time, but some special conditions ($s \bmod (p*t) < t$) must hold, where t is the block size, s is the array section stride, and p is the number of processors. Kennedy *et al.* [9] described an improved algorithm that computes the memory access sequence for the general case in $O(t + \min(\log s, \log p))$ time. In their approaches, when the communication set is evaluated, an explicit local-to-global translation corresponding to the *r.h.s.* section and a global-to-local translation corresponding to the *l.h.s.* section need to be performed for each referenced element on the *r.h.s.* section. Stichnoth *et al.* [12, 13] addressed communication set identification and local memory access determination for referenced array elements using a block-cyclic distribution. However, their method does not attempt to formulate active processor sets with respect to each source/destination processor. Gupta *et al.* [6] provided a *virtual processor scheme* to address the problem of referenced index-set identification for array statements with block-cyclic distributions and to formulate active processor sets as closed forms. During evaluation of communication sets, evaluation of the first and last iterations within each block is required.

In this paper, we address the problems of finding local memory access sequences and evaluating communication sets. During evaluation of communication sets, however, our method does not require explicit local-to-global or global-to-local transformation for each referenced array element. To efficiently generate communication sets, we classify all the blocks of array elements into several classes according to the offsets of their first array elements referenced within blocks. We use a *class table* to store the information that is extracted from these classes. In a *class table*, we record four items for each class, including the offsets of the first and last referenced elements, and the first and last iterations of the first block in a class. We can get the first and last iterations within each block easily from the *class table* and avoid some computation overhead. Moreover, in case the array section stride is larger than the block size, we can use the *class table* to enumerate the communication sets systematically and avoid runtime address resolution [6]. In addition, previous researches almost all focused on the array reference with one index variable. One advantage of our work over others is that it relaxes the restriction of an array reference with one index variable into affine functions of loop indices. According to the different boundary values of the array section, we can determine the corresponding *class table* to enumerate the communication sets. The time complexity of the algorithm used to construct class tables for the case of multiple index variables is $O(s^2)$, where s is the array section stride. The strategy for compiling the array reference with multiple index variables is easy, intuitive, and efficient. We believe that this method is feasible for compiling an array reference with multiple index variables.

Closed form characterization of index sets and processor sets would reduce the overhead needed for packing data into messages on source processors and unpacking data at destination processors. For an array reference with one index variable, if the array has only block or cyclic distribution, then the data index sets and the processor sets can be characterized with regular sections for closed forms [4, 5, 10]. However, for general block-cyclic distribution, closed form characterization of these sets with simple regular sections is impossible [6]. However, if we change our viewpoint from processors to blocks, closed form characterizations of communication sets and local memory access sequence are the same as cyclic distribution. Therefore, when evaluating a communication set, the following index sets and block sets for a given block b have to be determined.

- The set of blocks to which b has to send data.
- The set of indices of the array elements which are allocated on b but are needed by block k.
- The set of blocks from which b has to receive data.
- The set of indices of the array elements which are needed by b but are resident on block k.

For a general block-cyclic distribution, since a message from processor p to processor q consists of all the data which have to be sent from the blocks on p to the blocks on q, we can obtain communication sets by the intersection of the index sets for all blocks b and k, where blocks b are on p and blocks k are on q. We can get the processor sets for processor p in the same way. However, in order to efficiently compute the index sets and processor sets for a given processor p, we use the *class table* to generate these sets. We have stated that the *class table* records the offsets of the first and last referenced elements and the first and last iterations of the first block in a class. The first two elements can help us to generate the local memory access sequence, and the last two elements are useful for evaluating the communication sets. The time complexity of the algorithm used to construct the *class table* is $O(s/gcd(s, t))$, and $O(s)$ is the worst case. For array references with multiple index variables, the evaluation of communication sets is more complicated than it is for array references with one index variable. We still have a general method for compiling array references with multiple index variables, and the overhead for constructing a *class table* with an array reference is $O(s^2)$.

The rest of the paper is organized as follows. In Section 2, we introduce some notations and terminology used throughout the paper. The closed forms of the communication sets for array references with one index variable and multiple index variables derived using *class tables* are described in Section 3. Experimental results and a comparison of our technique with the technique proposed in [5, 6] are given in Section 4. Section 5 concludes the paper.

2. PRELIMINARIES

In this section, we will introduce the array statement model used in the paper, formulate the closed form of the indices of array elements with block, cyclic, block-cyclic distributions, and show the basic compilation phases of array references, including the sending, receiving, and computation phases. The outlines of the algorithms for these three phases will also be provided in this section.

2.1 Assignment Statement Model

In this paper, we consider a simple FORALL loop in a data-parallel program:

$$\text{FORALL}(I_1 = L_1{:}U_1,\ I_2 = L_2{:}U_2,\ \ldots,\ I_n = L_n{:}U_n)$$

$$A(f_A(I_1, I_2, \ldots, I_n)) = F(X(f_X(I_1, I_2, \ldots, I_n))).$$

This loop has the following characteristics.

- $F(X(f_X))$ denotes the function of $X(f_X)$. F can contain only one array variable, and the array variable's reference function is fixed.
- The array references f_A and f_X are a linear combination of index variables with integer constant coefficients:

$$f_A(I_1, I_2, \ldots, I_n) = a_0 + a_1I_1 + a_2I_2 + \ldots + a_nI_n$$

$$f_X(I_1, I_2, \ldots, I_n) = x_0 + x_1I_1 + x_2I_2 + \ldots + x_nI_n\ .$$

- The loops' lower and upper bound expressions, L_1, U_1, L_2, U_2, …, L_n, U_n, can be constants or a linear combination of outer index variables.

For simplicity, here, we assume that the iteration strides are one. If the iteration stride S_i is not one for some i, we can rewrite the loop as follows in order to preserve the iteration stride with one step:

$$\text{FORALL}(I_1 = L_1{:}U_1,\ \ldots,\ I_i = 0 : \lfloor (U_i - L_i)/S_i \rfloor,\ \ldots,\ I_n = L_n : U_n)$$

$$A(f_A(I_1, I_2, \ldots, I_n)) = F(X(f_X(I_1, I_2, \ldots, I_n))),$$

where

$$f_A(I_1, I_2, \ldots, I_n) = a_0 + a_1I_1 + \ldots + (a_i * S_i * I_i + a_i * L_i) + \ldots + a_nI_n$$

$$f_X(I_1, I_2, \ldots, I_n) = x_0 + x_1I_1 + \ldots + (x_i * S_i * I_i + x_i * L_i) + \ldots + x_nI_n.$$

Each array in such an assignment statement will be specified as a distribution with directives provided by data-parallel languages. We will now introduce common distributions provided by general data-parallel languages.

2.2 Data Distribution

Data-parallel languages, such as High Performance Fortran (HPF) [1] and Fortran D [3], provide regular distributions, including block, cyclic, block-cyclic distributions. When the array elements are distributed over processors, we need to store these array elements in local memory. The index of an array element in the original program is called a *global index*, and the index of an array element in some processor's local memory is called a *local index*. There exists a mapping between local indices and global indices.

Consider an array $A(0 : n - 1)$ distributed onto P processors. Let $B = [n/t]$ denote the total number of blocks, where t is the block size and $POB(p)$ denotes the set of blocks owned by processor p. Thus, we obtain $POB(p) = \{b \mid b = p + i * P, 0 \le i \le \lceil B/P \rceil - 1, 0 \le b \le B - 1\}$. Actually, block and cyclic distributions are both special cases of the block-cyclic distribution. Obviously, a cyclic distribution is equivalent to cyclic (1), so each element forms a block, and $POB(p)$ is equal to the global index set of the array elements which are owned by processor p. A block distribution is equivalent to cyclic ($\lceil n/P \rceil$), so each processor has only one block. Thus, we have $POB(p) = \{p\}$ Hence, the general functions for mapping the global to local indices and local to global indices can be formulated as follows:

global-to-local: $loc = \lfloor gl/(t * P) \rfloor * t + (gl \bmod t)$,

local-to-global: $gl = (\lfloor loc/t \rfloor * P + p) * t + (loc \bmod t)$,

where loc indicates the local index and gl is the global index.

Let the array reference $A(l : u : s)$ contain the access sequence as follows: $(l, l + s, l + 2s, \ldots, u)$, where l, u, and s are all integers. $(l : u : s)$ indicates a regular section with a fixed memory stride, s. If we can represent the local memory access sequence of each processor in a regular section, it will be easy to enumerate the access sequence in closed forms. However, the local memory access sequence for each processor may not be presented as a regular section when array elements are distributed in a block-cyclic distribution.

Now, we can define a set of iterations, denoted by $BOI(b)$, which includes the iterations that reference the array elements owned by block b. Let $POI(p)$ denote the set of iterations that references the array elements allocated on processor p, that is, $POI(p) = \cup_{b \in POB(p)} BOI(b)$. For an array reference $A(l : u : s)$ and a array A in block-cyclic distribution, the local memory access sequence in a processor maybe have not constant memory stride, but it must be a constant within each block. Thus, if we know the first and last active elements in each block, we will have a regular memory access sequence. Following this idea, we can try to classify the blocks to enumerate the local memory access sequence more efficiently. The way to classify blocks will be introduced in the next subsection.

2.3 Classification of Blocks

First of all, blocks are classified into three types. If all the elements of a block are less than l or greater than u, then we call it a *blank block*. Those two blocks which have the elements $A(l)$ or $A(u)$ are called *boundary blocks*. The rest of the blocks are called *active blocks*. Since no elements of a *blank block* will be referenced and only two blocks are *boundary blocks*, we will first concentrate our attention on active blocks, and the exception will be discussed in the next section. For the sake of simplicity, hereinafter, a block means an *active block* except additional specification.

Two blocks belong to the same class if and only if the offsets of the first and last active elements within these two blocks are the same. Therefore, for a given block b, the class number which it belongs to is equal to b mod *class*, where *class* means the number of classes. Since those blocks belonging to the same class have the same offsets of the first and last active elements, we can denote the offsets of the first and last active elements in class i as *index_low*(i) and *index_up*(i), respectively. The value of *index_low*(i) will be set to *null* if the active blocks in class i contain no active element.

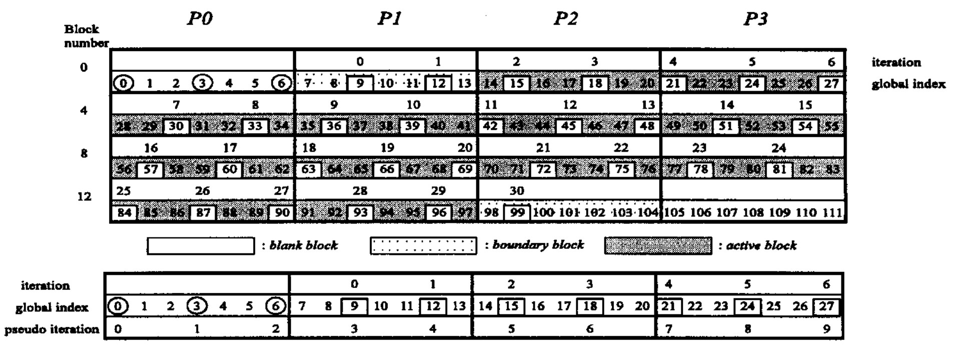
As previously described, those blocks belonging to the same class have the same offsets of the first and last active elements. Thus, if we view blocks in cyclic manner from class 0 to class *class* – 1 as a cycle, the number of active elements in each cycle is the same. This property is very helpful when we are evaluating the corresponding iteration of an active element. Let *iter_low*(i) and *iter_up*(i) denote the first and last iterations of class i in a cycle, respectively. Those data elements in regular section $((l \bmod s) : (l \bmod s) + (\lfloor \frac{l}{s} \rfloor - 1) \times s : s)$ are not active elements. However, when we evaluate *iter_low*(i) and *iter_up*(i), these data elements should be taken into consideration. To do so, evaluation of *iter_low*(i) and *iter_up*(i) can begin with block 0 instead of the first active block of a cycle. We term these data elements pseudo active elements. Those iterations which access pseudo active elements are termed pseudo active iterations. By definition, a *class table* keeps the information about the classification of blocks.

Fig. 1 illustrates the notions of *index_low*, *index_up*, *iter_low*, and *iter_up*. Blocks 3, 6, 9, and 12 belong to class 0. The offsets of the first and last active elements of these blocks are 0 and 6, respectively. Therefore, the values of *index_low*(0) and *index_up*(0) are set to be 0 and 6, respectively. On the other hand, the pseudo active elements are (0 : 6 : 3). When we evaluate *iter_low* and *iter_up*, these three pseudo active elements have to be counted. The first and last iterations of class 0 in a cycle are 0 and 2, respectively. Thus, *iter_low*(0) = 0 and *iter_up*(0) = 2. In the same way, *iter_low*(1), *iter_up*(1), *iter_low*(2), and *iter_up*(2) can be obtained.

We can prove that the classification of blocks is true for any array reference of the form $A(l : u : s)$ and array A with a cyclic(t) distribution using the following theorem.

Theorem 1: *Given an array A(0 : n-1) in a cyclic(t) distribution, for any array reference A(l : u : s), all the active blocks can be classified into s/gcd(s , t) classes.*
proof.

Suppose the offset of the first active element in block b is i; we want to prove that the offset of the first active element in block $b + s/gcd(s, t)$ is also i. By definition, we obtain the index of the first active element of block $b + s/gcd(s, t)$ as follows:



class number	0	1	2
index_low	0	2	1
index_up	6	5	4
iter_low	0	3	5
iter_up	2	4	6

class number	0	1	2
block number	3,6,9,12	4,7,10,13	2,5,8,11

Fig. 1. Block classes and the *class table* for $A(3 * I + 9)$, where $0 \le I \le 30$, $A(0 : 111)$ is distributed in cyclic (7). Rectangles indicate the active elements. Circles indicate the pseudo active elements.

$$\lceil ((b + s/gcd(s, t)) * t - (b * t + i))/s \rceil * s + b * t + i$$

$$= \lceil (lcm(s, t) - i)/s \rceil * s + b * t + i.$$

Because $i < s$ holds, $\lceil (lcm(s, t) - i)/s \rceil = lcm(s, t)/s$. Therefore, we have the solution of index $b * t + i + lcm(s, t)$ such that the offset index of the first active element of block $b + s/gcd(s, t)$ is i. □

The algorithm for constructing the *class table* is shown in Fig. 2.

```
$\upsilon = l \bmod s$
$class = s/gcd(s, t)$
Do $i = 0$ TO $class - 1$
    $iter\_low(i) = \lceil max(i * t - \upsilon, 0)/s \rceil$
    $iter\_up(i) = \lfloor ((i + 1) * t - 1 - \upsilon)/s \rfloor$
    IF $iter\_low(i) \le iter\_up(i)$ THEN
        $index\_low(i) = iter\_low(i) * s + \upsilon - i * t$
        $index\_up(i) = iter\_up(i) * s + \upsilon - i * t$
    ELSE
        $index\_low(i) = null$
    ENDIF
ENDDO
```

Fig. 2. Compute_Class_Table algorithm.

In Fig. 2, *class* denotes the number of classes. From Fig. 2, the *class table* will be changed according to different values of υ, s, and t. υ is changed according to different values of l and s. Therefore, the factors that really affect the construction of *class tables* are the values of l, s, and t. Hence, each array may have several *class tables* corresponding to respective array references and distributions. For each array reference, in order to construct the respective *class table*, the algorithm has to be executed at the beginning of an assignment statement. This algorithm takes $O(s/gcd(s, t))$ time because it has only one loop, and the time complexity is subjected to the bound of the loop. Hence, the time complexity of the worst case for this algorithm is $O(s)$ as s is relatively prime to t.

In the next section, we will discuss how the *class table* can be used to compute communication sets and to list local memory access sequences.

3. COMPILATION OF ARRAY REFERENCES

In this section, we will discuss how to generate communication sets and enumerate the local memory access sequences for array references with one index variable. Finally, we will adapt our approach to array references with multiple index variables.

3.1 Compilation Phases

We will give the details of the three compilation phases, *sending*, *receiving*, and *computation* phases, for compiling an assignment statement $A(f_A) = F(X(f_X))$ using the notations of sets. The following definitions of these sets are similar to those of Gupta *et al*. [6]. For integral description in this paper, we modify some notations and enumerate them as follows.

In the sending phase, we have to evaluate $PSD(p, q)$, the set of the local indices of the array elements needed by processor q but owned by processor p, and $SP(p)$, the set of the processors that processor p has to send data to. However, the elements in these sets are naturally discontinuous when array elements are block-cyclically distributed across processors. Thus, we define the following sets which have definitions corresponding to the above sets from the viewpoint of blocks to help us represent the sets $SP(p)$ and $PSD(p, q)$:

- $BSD(b, k)$: the set of the local indices of the array elements required by block k but allocated on block b. We have to send this set from block b to block k.
- $SB(b)$: the set of blocks that block b has to send data to.

We can represent the above sets as closed forms. Therefore, $SP(p)$ and $PSD(p, q)$ can be represented as closed forms by $SB(b)$ and $BSD(b, k)$ as follows:

$$SP(p) = \cup_{b \in POB(p)}(\cup_{k \in SB(b)} proc(k))$$

$$PSD(p, q) = \cup_{b \in POB(p)}(\cup_{k \in SB(b) \cap POB(q)} BSD(b, k)).$$

proc(k) is used to evaluate the processor number that block k resides on. By means of these equations, we can use a two-nested loop to calculate the sets $SP(p)$ and $PSD(p, q)$. We illustrate the basic steps in the sending phase as follows:

```
( 1) /*algorithm for sending phase */
( 2) DO $b \in POB(p)$
( 3)       DO $k \in SB(b)$
( 4)             $q = proc(k)$
( 5)             $SP(p) = SP(p) \cup \{q\}$
( 6)             $PSD(p, q) = PSD(p, q) \cup BSD(b, k)$
( 7)       ENDDO
( 8) ENDDO
( 9) DO $q \in SP(p)$
(10)       $send(X(PSD(p, q)))$
(11) ENDDO
```

The loop from line 2 to line 8 is used to compute the set of processors $SP(p)$ and the set $PSD(p, q)$ for each processor q in $SP(p)$. Then, in lines 9-11, we pack the messages which consist of $X(PSD(p, q))$ and send it to each processor q in $SP(p)$. In the receiving phase, we use the following sets, which have the dual definitions of those sets used in the sending phase, to show the basic steps of the receiving phase:

- $BRD(b, k)$: the set of the local indices of the array elements required by block b but allocated on block k.
- $RB(b)$: the set of blocks that block b has to receive data from.
- $PRD(p, q)$: the set of the local indices of the array elements needed by processor p but owned by processor q. Processor p has to receive this set from processor q.
- $RP(p)$: the set of processors that processor p has to receive data from.

Similarly, the algorithm for receiving phase is stated as follows:

```
( 1) /*algorithm for receiving phase */
( 2) DO $b \in POB(p)$
( 3)       DO $k \in RB(b)$
( 4)             $q = proc(k)$
( 5)             $RP(p) = RP(p) \cup \{q\}$
( 6)             $PRD(p, q) = PRD(p, q) \cup BRD(b, k)$
( 7)       ENDDO
( 8) ENDDO
( 9) DO $q \in RP(p)$
(10)       $receive(tmp\_input)$
(11)       $tmp\_X(PRD(p, q)) = tmp\_input$
(12) ENDDO
```

The loop in lines 2-8 is parallel to the loop for the sending phase in the same line. Processor p sends the elements of the indices in $PSD(p, q)$ to processor q in the sending phase and receives the elements of the indices in $PRD(p, q)$ from q in the receiving phase. Received messages are stored in temporary array *temp_X*, which has the same size as array A. Then, we align the receiving elements with the elements of array A on each processor. The algorithm for the computation phase is shown as follows:

```
( 1) /* algorithm for computation phase */
( 2) DO $b \in POB(p)$
( 3)      DO $i \in BOI(b)$
( 4)           $d = global\text{-}to\text{-}local(f_A(i))$
( 5)           $A(d) = F(tmp\_X(d))$
( 6)      ENDDO
( 7) ENDDO
```

Based on the above ideas, the algorithms for the three compilation phases will be discussed in more detail in the next subsection.

3.2 Array References with One Index Variable

Let arrays $A(0 : n_1 - 1)$ and $X(0 : n_2 - 1)$ be distributed on P_1 and P_2 processors with cyclic(t_1) and cyclic(t_2) distribution, respectively. For each processor p, the formulations of the sets which have been introduced in previous subsection are described as follows:

$$B_1 = \lceil n_1/t_1 \rceil$$

$$POB^A(p) = \{b \mid b = p + i * P_1,\ 0 \le i \le \lceil B_1/P_1 \rceil - 1,\ 0 \le b \le B_1 - 1\}$$

$$B_2 = \lceil n_2/t_2 \rceil$$

$$POB^X(p) = \{b \mid b = p + i * P_2,\ 0 \le i \le \lceil B_2/P_2 \rceil - 1,\ 0 \le b \le B_2 - 1\}.$$

In this subsection, the array references with one index variable are considered. Without loss of generality, let the array reference be in the form $A(s_1 * I + c_1) = F(X(s_2 * \mathrm{I} + c_2))$, where $L \le I \le U$. In the sending phase, we have to evaluate $PSD(p, q)$, the set of the local indices of the array elements needed by processor q and owned by processor p, and $SP(p)$, the set of processors that processor p has to send data to. From the block point of view, before evaluating these two sets, we first have to determine the index set, the set of the indices of the array elements which the source block has to send to the destination block, and the block set, the set of the destination blocks that some source block has to send data to, i.e., $BSD(b, k)$ and $SB(b)$, respectively. Hence, we first solve the set $SB(b)$. Because most of the blocks belong to the active blocks, only the active blocks with respect to array X are considered to evaluate the set $SB(b)$; i.e, block b needs to be an active block of array X. However, blocks with respect to array A are all considered; that is, the set $BSD(b, k)$ is the index set that block b must send to block k, where b has to be some active block of array X, but k can be either a boundary block or an active block of array A. Regarding the boundary blocks of array X, this case will be discussed at the end of this subsection.

For simplicity, we use the subscript 1 to denote those terms with respect to array A and the subscript 2 to denote the terms related to array X except additional specification. Let $A(l_1)$ and $A(u_1)$ be the first and last active elements of array A; we obtain $l_1 = s_1 * L + c_1$ and $u_1 = s_1 * U + c_1$. Similarly, let $X(l_2)$ and $X(u_2)$ denote the first and last active elements of array X; we have $l_2 = s_2 * L + c_2$ and $u_2 = s_2 * U + c_2$. Let $X(s_2 * \tau_2 + c_2)$ and $X(s_2 * \rho_2 + c_2)$ be the first and last active elements within block b. We obtain τ_2 and ρ_2 as follows:

$$\tau_2 = \lceil (b * t_2 - l_2)/s_2) \rceil, \quad \sigma_2 = \lfloor ((b + 1) * t_2 - 1 - l_2)/s_2 \rfloor.$$

Then, the iterations that reference the array elements X on block b, denoted by the set $BOI^X(b)$, are $BOI^X(b)=\{i \mid \tau_2 \le i \le \rho_2\}$. Therefore, block b has to send data to each block k which owns the elements of the set $f_A(BOI^X(b))$. If stride s_1 is less than the block size t_1, each block has at least one active element. This implies that every block between $\lfloor f_A(\tau_2)/t_1 \rfloor$ and $\lfloor f_A(\sigma_2)/t_1 \rfloor$ will receive at least one array element from block b. In another case where $s_1 \ge t_1$, the contiguous active elements will be located on discontinuous blocks, and each block will contain at most one active element. Thus, we have

$$SB(b) = \begin{cases} \{k \mid \lfloor f_A(\tau_2) / t_1 \rfloor \le k \le \lfloor f_A(\sigma_2) / t_1 \rfloor\} & \text{if } s_1 < t_1 \\ \{k \mid k = \lfloor f_A(i) / t_1 \rfloor,\ \tau_2 \le i \le \sigma_2\} & \text{if } s_1 \ge t_1 . \end{cases}$$

Figs. 3 and 4 illustrate the above two cases.

In the case $s_1 \ge t_1$, due to the discontinuity of block set $SB(b)$, the trivial way to enumerate these blocks is to list them one by one. To avoid runtime address resolution, we can use the characteristics of the *class table* to calculate the set $SB(b)$ when $s_1 \ge t_1$. In the *class table*, those classes that the value of *index_low* equals *null* imply that the blocks which belong to these classes contain no active element. These blocks can be precluded when the blocks in $SB(b)$ are enumerated. Therefore, when the set $SB(b)$ is enumerated, the blocks between $\lfloor f_A(\tau_2)/t_1 \rfloor$ and $\lfloor f_A(\sigma_2)/t_1 \rfloor$ except for the null blocks are all needed. We can rewrite the set $SB(b)$ in the case $s_1 \ge t_1$ as follows:

$$SB(b) = \{k \mid k = i + j * class_1,\ 0 \le j \le \left\lfloor \left(\left\lfloor \frac{f_A(\sigma_2)}{t_1} \right\rfloor - i \right) / class_1 \right\rfloor,$$

$$\left\lfloor \frac{f_A(\tau_2)}{t_1} \right\rfloor \le i \le \left\lfloor \frac{f_A(\tau_2)}{t_1} \right\rfloor + (class_1 - 1) \text{ and } index\_low(i \bmod class_1) \ne null\} .$$

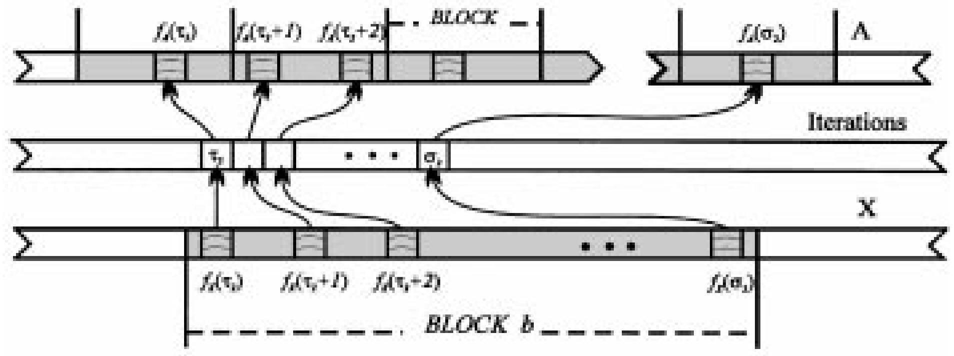


Fig. 3. Determination of the block set $SB(b)$ in case $s_1 < t_1$.

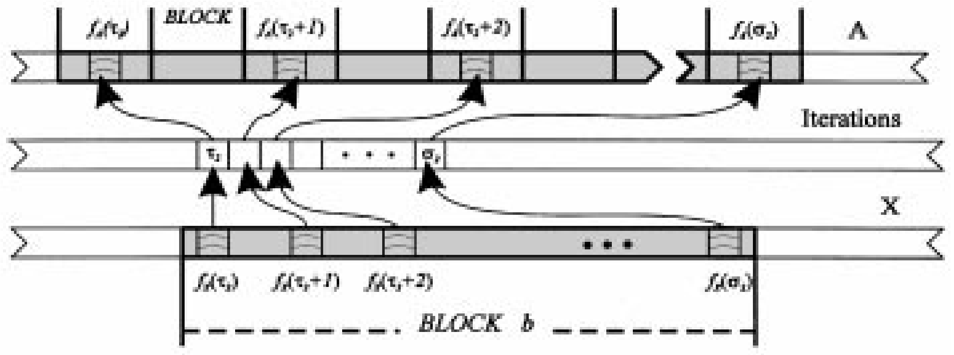


Fig. 4. Determination of the block set $SB(b)$ in case $s_1 < t_1$.

After solving the block set $SB(b)$, we can now consider the index set $BSD(b, k)$. To solve the set $BSD(b, k)$, we have to determine which iterations on block k require the data on block b, where block k can be either a boundary block or an active block of array A and block b has to be an active block of array X. The iterations which reference the data of X on block b are in $BOI^X(b)$. The iterations which are executed on block k are denoted by $BOI^A(k)$, where $BOI^A(k) = \{i \mid \tau_1 \le i \le s_1\}$ and

$$\tau_1 = \lceil max(k * t_1 - l_1, 0)/s_1 \rceil,\ \sigma_1 = \lfloor min((k + 1) * t_1 - 1 - l_1, u_1 - l_1)/s_1 \rfloor.$$

From $BOI^X(b)$ and $BOI^A(k)$, the global index set of the array elements of X, which have to be sent from block b to block k, can be formulated as follows:

$$f_X(BOI^A(k) \cap BOI^X(b)) = \{f_X(i) \mid max(\tau_2, \tau_1) \le i \le min(\sigma_2, \sigma_1)\}.$$

The set $BSD(b, k)$ is defined as the set of the local indices of the array elements required by block k but allocated on block b. From the above formulation, we already have evaluated the global index set of the array elements which block b needs to send to block k. We can then transform the global index into local index by means of the function of *global-to-local*. The set $BSD(b, k)$ can, therefore, be obtained as follows:

$$BSD(b, k) = \{loc \mid loc = gl \ mod \ t_2 + \lfloor b/P_2 \rfloor * t_2, gl \in f_X(BOI^A(k) \cap BOI^X(b))\}.$$

In the sending phase, what we really want to evaluate is the sets $PSD(p, q)$ and $SP(p)$, that is, the set of the local indices of the array elements needed by processor q but owned by processor p, and the set of the processors that processor p has to send data to, respectively. Evaluation of the sets $BSD(b, k)$ and $SB(b)$ is a preprocessing step when we evaluate the sets $PSD(p, q)$ and $SP(p)$. By the above description, we already have the sets $BSD(b, k)$ and $SB(b)$. Now, we will describe how to evaluate the sets $PSD(p, q)$ and $SP(p)$ using the sets $BSD(b, k)$ and $SB(b)$.

For a given processor p, only the active blocks owned by p need to evaluate $SB(b)$ and $BSD(b, k)$. Let *head_b* and *tail_b* denote the first and last active blocks in p, respectively. We can illustrate the formulations of these two variables as follows:

$$head\_b(p) = \begin{cases} p + \lfloor l/(P*t) \rfloor * P & \text{if } p > \lfloor l/t \rfloor \bmod P \\ p + (\lfloor l/(P*t) \rfloor + 1) * P & \text{if } p \le \lfloor l/t \rfloor \bmod P \end{cases}$$

$$tail\_b(p) = \begin{cases} p + \lfloor u/(P*t) \rfloor * P & \text{if } p < \lfloor u/f \rfloor \bmod P \\ p + (\lfloor u/(P*t) \rfloor - 1) * P & \text{if } p \ge \lfloor u/f \rfloor \bmod P. \end{cases}$$

It is possible that *head_b* may be greater than *tail_b*. In this case, it is implied that there is no active block on this processor. For array A, $head\_b_1$ and $tail\_b_1$ are the first and last active blocks in some processor p. Similarly, for array X, $head\_b_2$ and $tail\_b_2$ are the first and last active blocks in some processor q. In the sending phase, we have to visit all the active blocks with respect to array X to evaluate the sending set for each processor. We can represent all the active blocks in p as a regular section $(head\_b_2 : tail\_b_2 : P_2)$. For each block b in this regular section, the first and last iterations are required for evaluation of $BOI^X(b)$; that is, τ_2 and σ_2 need to be evaluated. Although the formulations of these two variables already have been described, we can get them in another way by using the *class table* instead of straightforward computation and save some computation overhead. First, we evaluate the following four variables for a given block b:

$c_2 = b \bmod class_2$
$r = b \text{ div } class_2$
$per\_iter = iter\_up_2(class_2 - 1) + 1$
$iter\_dist = r * per\_iter - \lfloor l_2/s_2 \rfloor.$

c_2 is the class number that block b belongs to, r stands for the number of cycles before block b appears, *per_iter* indicates the total number of iterations that a cycle contains, and *iter_dist* is the number of iterations before block b appears. From the above, we obtain τ_2 and σ_2 as follows:

$$\tau_2 = iter\_dist + iter\_low_2(c_2)$$

$$\sigma_2 = iter\_dist + iter\_up_2(c_2).$$

Fig. 1 shows an example with *class* = 3, *per_iter* = 7, l = 9, and s = 3. Let b = 8; then, we obtain c_2 = 2, r = 2, *iter_dist* = 2 * 7 – 3 = 11. Therefore, we have τ = 11 + 5 = 16 and σ = 11 + 6 =17. The sending algorithm is shown in Fig. 5. The initialization of the variables *class*, *head_b* and *tail_b* is not shown in Fig. 5. We assume that the values of these variables and the *class table* are evaluated before the algorithm is executed. The Do loops in lines 10-21 and lines 23-36 are executed for the set $SB(b)$. The Do loop in lines 17-20 and lines 29-32 are for evaluation of $BSD(b, k)$. In order to reduce the communication overhead, we collect all the messages together that need to be sent to the same destination processor but may belong to different blocks.

In the sending phase, we travel over the active blocks of array X which are owned by processor p and determine which elements of array A need the elements of array X. In contrast, we travel over the active blocks of array A which are owned by p and determine which elements of array X need the elements of array A in the receiving phase. Hence, the evaluation part of the receiving phase is different from the evaluation part of the sending phase. The difference between these two phases is that the messages received have to be unpacked and stored in temporary array *tmp_X*. Fig. 6 shows the receiving phase algorithm.

After completion of the communication phase, all the data referenced by the computation have been allocated in the processor's local memory. Hence, the most important work in the computation phase is to enumerate the local memory access sequence of array A. Though it can not be presented as a closed form, the sequence within a block is a regular section. Hence, we first scan all the blocks in a regular section ($head\_b_1 : tail\_b_1 : P_1$) and then find the indices of the first and last active elements within a given block b. The indices, just like the first and last iterations of blocks, can be obtained easily using information from the *class table*. We can evaluate the following two variables which can assist us in finding these two elements:

$$c_1 = b \bmod class_1$$

$$block\_dist = \left\lfloor \frac{b}{P_1} \right\rfloor * t.$$

Then, we can obtain the local indices of the first and last active elements for block b as follows:

$$first\_index = block\_dist + index\_low_1(c_1)$$

$$last\_index = block\_dist + index\_up_1(c_1).$$

```
( 1) /* Sending_Evaluation algorithm */
( 2) $per\_iter = iter\_up_2(class_2 - 1) + 1$
( 3) $c_2 = head\_b_2$ mod $class_2$
( 4) $iter\_dist = (head\_b_2$ div $class_2) * per\_iter - \lfloor l_2/s_2 \rfloor$
( 5) DO $b = head\_b_2$ TO $tail\_b_2$ STEP $P_2$
( 6)     IF $index\_low_2(c_2) \neq null$ THEN
( 7)         $\tau_2 = iter\_dist + iter\_low_2(c_2)$
( 8)         $\sigma_2 = iter\_dist + iter\_up_2(c_2)$
( 9)         IF $s_1 < t_1$ THEN
(10)             DO $k = \lfloor f_A(\tau_2)/t_1 \rfloor$ To $\lfloor f_A(\sigma_2)/t_1 \rfloor$
(11)                 $q = proc(k)$
(12)                 $SP(p) = SP(p) \cup \{q\}$
(13)                 $\tau_1 = \lceil max(k * t_1 - l_1, 0)/s_1 \rceil$
(14)                 $\sigma_1 = \lfloor min((k + 1) * t_1 - 1 - l_1, u_1 - l_1)/s_1 \rfloor$
(15)                 $first = f_X(max(\tau_1, \tau_2))$
(16)                 $last = f_X(min(\sigma_1, \sigma_2))$
(17)                 DO $loc = first$ mod $t_2 + \lfloor b/P_2 \rfloor * t_2$ TO $last$ mod $t_2 + \lfloor b/P_2 \rfloor * t_2$ STEP $s_2$
(18)                     $tmp\_output_q(cnt_q) = X(loc)$
(19)                     $cnt_q = cnt_q + 1$
(20)                 ENDDO
(21)             ENDDO
(22)         ELSE /* $s_1 \geq t_1$ */
(23)             DO $i = \lfloor f_A(\tau_2)/t_1 \rfloor$ TO $\lfloor f_A(\tau_2)/t_1 \rfloor + class_1 - 1$
(24)                 IF $(index\_low_1(i$ mod $class_1) \neq null)$ THEN
(25)                     $u = \tau_2$
(26)                     DO $j = i$ To $\lfloor f_A(\sigma_2)/t_1 \rfloor$ STEP $class_1$
(27)                         $q = proc(j)$
(28)                         $SP(p) = SP(p) \cup \{q\}$
(29)                         $loc = f_X(u)$ mod $t_2 + \lfloor b/P_2 \rfloor * t_2$
(30)                         $tmp\_output_q(cnt_q) = X(loc)$
(31)                         $cnt_q = cnt_q + 1$
(32)                         $u = u + per\_iter$
(33)                     ENDDO
(34)                 ENDIF
(35)                 $\tau_2 = \tau_2 + 1$
(36)             ENDDO
(37)         ENDIF
(38)     ENDIF
(39)     $r = (c_2 + P_2)$ div $class_2$
(40)     $c_2 = (c_2 + P_2)$ mod $class_2$
(41)     $iter\_dist = per\_iter * r + iter\_dist$
(42) ENDDO
(43) /* Sending_Packing Algorithm */
(44) DO $q \in SP(p)$
(45)     $send(q, tmp\_output_q, cnt_q)$
(46) ENDDO
```

Fig. 5. Sending algorithm for array references with one index variable.

```
( 1) /* Receiving_Evaluation Algorithm */
( 2) $per\_iter = iter\_up_1(class_1 - 1) + 1$
( 3) $c_1 = head\_b_1$ mod $class_1$
( 4) $iter\_dist = (head\_b_1$ div $class_1) * per\_iter - \lfloor l_1/s_1 \rfloor$
( 5) DO $b = head\_b_1$ TO $tail\_b_1$ STEP $P_1$
( 6)     IF $index\_low_1(c_1) \neq null$ THEN
( 7)         $\tau_1 = iter\_dist + iter\_low_1(c_1)$
( 8)         $\sigma_1 = iter\_dist + iter\_up_1(c_1)$
( 9)         IF $s_2 < t_2$ THEN
(10)             DO $k = \lfloor f_X(\tau_1)/t_2 \rfloor$ TO $\lfloor f_X(\sigma_1)/t_2 \rfloor$
(11)                 $q = proc(k)$
(12)                 $RP(p) = RP(p) \cup \{q\}$
(13)                 $\tau_2 = \lceil max(k * t_2 - l_2, 0)/s_2 \rceil$
(14)                 $\sigma_2 = \lfloor min((k + 1) * t_2 - 1 - l_2, u_2 - l_2)/s_2 \rfloor$
(15)                 $first = f_A(max(\tau_2, \tau_2))$
(16)                 $last = f_A(min(\sigma_1, \sigma_2))$
(17)                 DO $loc = first$ $mod$ $t_1 + \lfloor b/P_1 \rfloor * t_1$ TO $last$ $mod$ $t_1 + \lfloor b/P_1 \rfloor * t_1$ STEP $s_1$
(18)                     $tmp\_loc_q(cnt_q) = loc$
(19)                     $cnt_q = cnt_q + 1$
(20)                 ENDDO
(21)             ENDDO
(22)         ELSE /* $s_2 \geq t_2$ */
(23)             DO $i = \lfloor f_X(\tau_1)/t_2 \rfloor$ TO $\lfloor f_X(\tau_1)/t_2 \rfloor + class_2 - 1$
(24)                 IF ($index\_low_2(i$ mod $class_2) \neq null$) THEN
(25)                     $u = \tau_1$
(26)                     DO $j = i$ TO $\lfloor f_X(\sigma_1)/t_2 \rfloor$ STEP $class_2$
(27)                         $q = proc(j)$
(28)                         $RP(p) = RP(p) \cup \{q\}$
(29)                         $loc = f_A(u)$ mod $t_1 + \lfloor b/P_1 \rfloor * t_1$
(30)                         $tmp\_loc_q(cnt_q) = loc$
(31)                         $cnt_q = cnt_q + 1$
(32)                         $u = u + per\_iter$
(33)                     ENDDO
(34)                 ENDIF
(35)                 $\tau_1 = \tau_1 + 1$
(36)             ENDDO
(37)         ENDIF
(38)     ENDIF
(39)     $r = (c_1 + P_1)$ div $class_1$
(40)     $c_1 = (c_1 + P_1)$ mod $class_1$
(41)     $iter\_dist = per\_iter * r + iter\_dist$
(42) ENDDO
(43) /* Receiving_Unpacking Algorithm */
(44) DO $q \in RP(p)$
(45)     $receive(q, tmp\_input_q, cnt_q)$
(46)     DO $i = 0$ TO $cnt_q - 1$
(47)         $tmp\_X(tmp\_loc_q(i)) = tmp\_input(i)$
(48)     ENDDO
(49) ENDDO
```

Fig. 6. Receiving algorithm for array references with one index variable.

Hence, the local indices of the active elements in block b are represented as a regular section (*first_index* : *last_index* : s_1). The algorithm shown in Fig. 7 is a two-nested loop. The outer loop is used to enumerate the active blocks that processor owns, and the inner loop executes the elements of the regular section, which consists of the active elements in the active block. The *class table* avoids translating global indices to local indices or local indices to global indices and assists in generating the next active element.

```
( 1) /* Computation_Enumeration algorithm */
( 2) $block\_distance = \lfloor head\_b_1/P_1 \rfloor * t_1$
( 3) DO $b = head\_b_1$ TO $tail\_b_1$ STEP $P_1$
( 4)      $c_1 = b \bmod class_1$
( 5)      IF $index\_low_1(c_1) \neq null$ THEN
( 6)          DO $d = index\_low_1(c_1)$ TO $index\_up_1(c_1)$ STEP $s_1$
( 7)              $i = block\_distance + d$
( 8)              $A(i) = F(tmp\_X(i))$
( 9)          ENDDO
(10)      ENDIF
(11)      $block\_distance = block\_distance + t_1$
(12) ENDDO
```

Fig. 7. Computation algorithm for array references with one index variable.

In the previous discussion, we were only concerned with the *active blocks* of array X. For completeness, we will discuss how to deal with the boundary blocks of array X. The differences between active blocks and boundary blocks are described in the following two instances. In active blocks, the offset of the first active element must be less than the array reference stride. However, in boundary blocks, the offset of the first active element may be larger than the array section stride. It all depends on the value of l. On the other hand, the distance between the last active element and the last element (which may not be an active element) within an active block must be less than the array section stride; however, within a boundary block, this distance may be greater than the memory stride. Therefore, the above formulations are not suitable for boundary blocks. Hence, we generate another piece of code to handle them. Adding this code to the sending algorithm, we obtain the complete algorithm, which is called the One_Variable_Sending algorithm. Similarly, we have the One_Variable_Receiving algorithm and One_Variable_Computation algorithm for the receiving phase and computation phase, respectively. In these algorithms, there is a range behind some statement. This shows where the statement should be replaced by the range of codes in the specified algorithm. For instance, the fourth statement of the One_Variable_Sending algorithm means that here we should insert the codes from the ninth statement to the thirty-seventh statement in the Sending_Evaluation algorithm. We show these algorithms in Figs. 8, 9, 10 and the entire algorithm in Fig. 11.

```
( 1) IF $p = \lfloor l_2/t_2 \rfloor \bmod P_2$ THEN
( 2)     $\tau_2 = 0$
( 3)     $\sigma_2 = \lfloor (t_2 - 1 - (l_2 \bmod t_2))/s_2 \rfloor$
( 4)     Sending_Evaluation algorithm (9-37)
( 5) ENDIF
( 6) Sending_Evaluation algorithm
( 7) IF $p = \lfloor u_2/t_2 \rfloor \bmod P_2$ THEN
( 8)     $\sigma_2 = \lfloor (u_2 - l_2)/s_2 \rfloor$
( 9)     $\tau_2 = \sigma_2 - \lfloor (u_2 \bmod t_2)/s_2 \rfloor$
(10)     Sending_Evaluation algorithm (9-37)
(11) ENDIF
```

Fig. 8. One_Variable_Sending algorithm.

```
( 1) IF $p = \lfloor l_1/t_1 \rfloor \bmod P_1$ THEN
( 2)     $\tau_1 = 0$
( 3)     $\sigma_1 = \lfloor (t_1 - 1 - (l_1 \bmod t_1))/s_1 \rfloor$
( 4)     Receiving_Evaluation algorithm (9-37)
( 5) ENDIF
( 6) Receiving_Evaluation algorithm
( 7) IF $p = \lfloor u_1/t_1 \rfloor \bmod P_1$ THEN
( 8)     $\sigma_1 = \lfloor (u_1 - l_1)/s_1 \rfloor$
( 9)     $\tau_1 = \sigma_1 - \lfloor (u_1 \bmod t_1)/s_1 \rfloor$
(10)     Receiving_Evaluation algorithm (9-37)
(11) ENDIF
```

Fig. 9. One_Variable_Receiving algorithm.

```
( 1) IF $p = \lfloor l_1/t_1 \rfloor \bmod P_1$ THEN
( 2)     $low = global - to - local(l_1)$
( 3)     $up = global - to - local((\lfloor l_1/t_1 \rfloor + 1) * t - 1)$
( 4)     DO $i = low$ TO $up$ STEP $s_1$
( 5)         $A(i) = F(tmp\_X(i))$
( 6)     ENDDO
( 7) ENDIF
( 8) Computation_Enumeration
( 9) IF $p = \lfloor u_1/t_1 \rfloor \bmod P_1$ THEN
(10)     $low = index\_low_1(c_1) + block\_distance$
(11)     $up = global - to - local(u_1)$
(12)     DO $i = low$ TO $up$ STEP $s_1$
(13)         $A(i) = F(tmp\_X(i))$
(14)     ENDDO
(15) ENDIF
```

Fig. 10. One_Variable_Computation algorithm.

```
(1) Compute_Class_Table algorithm for array A
(2) Computd_Class_Table algorithm for array X
(3) Initialization of $head\_b_1$, $head\_b_2$, $tail\_b_1$, $tail\_b_2$
(4) One_Variable_Sending algorithm
(5) Sending_Packing algorithm
(6) One_Variable_Receiving algorithm
(7) Receiving_Unpacking algorithm
(8) One_Variable_Computation algorithm
```

Fig. 11. One_Variable algorithm.

3.3 Array References with Multiple Index Variables

The local memory access sequence in each block must be a regular section though the local memory access sequence on each processor may not be a regular section when the array reference involves only one index variable. However, the local memory access sequence is no longer a regular section, even in a block where the array reference involves multiple index variables. Therefore, we have to change our viewpoint from array elements to iterations and reduce the problem to the case of array reference with one index variable. If we substitute index variables in outer loops for values of iterations and leave the index variable in the innermost loop to vary, we reduce the problem to one index variable, and we can apply the approach developed for array references with one index variable. Let IS denote the iteration space, i.e., $IS = \{(I_1, I_2, \ldots, I_n) \mid L_i \le I_i \le U_i, 1 \le i \le n\}$. If we ignore the last dimension of the iteration space, a reduced iteration space, denoted by IS_R, $IS_R = \{(I_1, I_2, \ldots, I_{n-1}) \mid L_i \le I_i \le U_i, 1 \le i \le (n-1)\}$ is formed. Let $(I_1, I_2, \ldots, I_{n-1})$ be the reduced iteration instance. We traverse the reduced iteration space in lexicographical order and substitute each reduced iteration instance into the array reference; then, the array reference with multiple index variables will be reduced to another array reference involving only the innermost index variable. For example, the array reference in the following 2-nested loop involves 2 index variables, I_1 and I_2:

```
FORALL($I_1 = L_1 : U_1$, $I_2 = L_2 : U_2$)
    $A(f_A(I_1, I_2)) = F(X(f_X(I_1, I_2)))$.
```

Applying the above strategy, the array reference can be reduced such that it involves just one index variable, I_2:

```
$I_1 = L_1$       : FORALL($I_2 = L_2 : U_2$)
                        $A(f_A(L_2, I_2)) = F(X(f_X(L_1, I_2)))$,
$I_1 = L_1 + 1$   : FORALL($I_2 = L_2 : U_2$)
                        $A(f_A(L_1 + 1, I_2)) = F(X(f_X(L_1 + 1, I_2)))$,
                  ⋮
$I_1 = U_1$       : FORALL($I_2 = L_2 : U_2$)
                        $A(f_A(U_1, I_2)) = F(X(f_X(U_1, I_2)))$.
```

Obviously, each reduced array reference is independent. The One_ Variable_Sending algorithm requires that each reduced array reference be performed once. In a different reduced array reference, the class table has to be recomputed due to the changed boundary values. However, computing the *class table* for each reduced iteration instance incurs additional overhead. To reduce this kind of overhead, we analyze the construction of the *class table* and derive the following theorem:

Theorem 2: *Given an array A(0 : n-1) in a cyclic(t) distribution, for any array reference A(l : u : s), active blocks can be divided into s/gcd(s,t) classes. Moreover, for different l, we have different class tables. All the class tables for all l can be divided into s distinct class tables.*
proof.

By Fig. 2, we know that different values of υ, s, and t will result in different kinds of *class tables*. If the array distribution and the stride of the reference patterns are fixed, i.e., s and t are fixed, then the factor that can affect the *class tables* is simply the value of υ. From Fig. 2, $\upsilon = l \bmod s$. Therefore, for different l, υ can have at most s kinds of values. Hence, all the class tables for all l can be divided into s distinct *class tables*. □

The benefit of the Multiple_Variable algorithm from Theorem 2 is that evaluation of class tables is performed just once; then, for different l, we can determine which class table fits the array reference $A(l : u : s)$ in constant time. We can rewrite the Compute_Class_Table algorithm into the Compute_Whole_ Class_Table algorithm as shown in Fig. 12. The variable m denotes the table number, and we can determine m as follows:

```
class = s/gcd(s, t)
DO m = 0 TO s – 1
    DO i = 0 TO class – 1
        iter_low(m, i) = ⌈max((i * t – m), 0)/s⌉
        iter_up(m, i) = ⌊((i + 1) * t – 1 – m)/s⌋
        IF iter_low(m, i) ≤ iter_up(m, i) THEN
            index_low(m, i) = iter_low(m, i) * s + m – i * t
            index_up(m, i) = iter_up(m, i) * s + m – i * t
        ELSE
            index_low(m, i) = null
        ENDIF
    ENDDO
ENDDO
```

Fig. 12. Compute_Whole_Class_Table algorithm.

$m = l \bmod s$.

We show the Multiple_Variables_Sending algorithm in Fig. 13. In this algorithm, we apply the One_Variable algorithm in the inner loop. In fact, it is not correct to place One_Variable algorithm there without any modification of the code, but all we have to do is to add the parameter m to specify the *class table*. Following the same idea, we can obtain the Multiple_Variable_Receiving algorithm and Multiple_Variable_Computation algorithm as shown in Fig. 14 and Fig. 15, respectively.

```
DO $I_1 = L_1$ TO $U_1$
    DO $I_2 = L_2$ TO $U_2$
            ⋮
        DO $I_{n-1} = L_{n-1}$ TO $U_{n-1}$
            $l_2 = x_0 + x_1 * I_1 + x_2 * I_2 + \ldots + x_{n-1} * I_{n-1} + x_n * L_n$
            $u_2 = x_0 + x_1 * I_1 + x_2 * I_2 + \ldots + x_{n-1} * I_{n-1} + x_n * U_n$
            Initialization of $head\_b_2$, $tail\_b_2$
            IF $head\_b_2 \le tail\_b_2$ THEN
                $m = l_2 \bmod s_2$
                One_Variable_Sending algorithm
            ENDIF
        ENDDO
            ⋮
    ENDDO
ENDDO
Sending_Packing algorithm
```

Fig. 13. Multiple_Variables_Sending algorithm.

```
DO $I_1 = L_1$ TO $U_1$
    DO $I_2 = L_2$ TO $U_2$
            ⋮
        DO $I_{n-1} = L_{n-1}$ TO $U_{n-1}$
            $l_1 = a_0 + a_1 * I_1 + a_2 * I_2 + \ldots + a_{n-1} * I_{n-1} + a_n * L_n$
            $u_1 = a_0 + a_1 * I_1 + a_2 * I_2 + \ldots + a_{n-1} * I_{n-1} + a_n * U_n$
            Initialization of $head\_b_1$, $tail\_b_1$
            IF $head\_b_1 \le tail\_b_1$ THEN
                $m = l_1 \bmod s_1$
                One_Variable_Receiving algorithm
            ENDIF
        ENDDO
            ⋮
    ENDDO
ENDDO
Receiving_Unpacking algorithm
```

Fig. 14. Multiple_Variables_Receiving algorithm.

```
DO $I_1 = L_1$ TO $U_1$
    DO $I_2 = L_2$ TO $U_2$
            ⋮
        DO $I_{n-1} = L_{n-1}$ TO $U_{n-1}$
            $l_1 = a_0 + a_1 * I_1 + a_2 * I_2 + \ldots + a_{n-1} * I_{n-1} + a_n * L_n$
            $u_1 = a_0 + a_1 * I_1 + a_2 * I_2 + \ldots + a_{n-1} * I_{n-1} + a_n * U_n$
            IF $head\_b_1 \le tail\_b_1$ THEN
                $m = l_1 \bmod s_1$
                One_Variable_Computation algorithm
            ENDIF
        ENDDO
            ⋮
    ENDDO
ENDDO
```

Fig. 15. Multiple_Variables_Computation algorithm.

We have adapted the approach for array references with one index variable to array references with multiple index variables. Since the problems related to listing local memory access sequences and generating communication sets for multiple index variables are much more complicated than are those for one index variable, almost all the existing methods target the problems related to one index variable. Therefore, the intuitive method for solving this problem is to directly scan over the all of the active elements and determine which elements need to be communicated. Note that the active elements can not be represented by just one variable. However, this method involves too many local-to-global and global-to-local computations. It is the most inefficient way to generate communication sets. Our method is an alternative choice. Although the cost of calculating all of the class tables is $O(s^2)$, once these class tables are available, the algorithm for one index variable, the One_Variable algorithm, can be directly used. In addition, our method is more efficient than other methods for one index variable. Therefore, we believe that this method is easier to apply to an array reference with multiple index variables than the other methods.

4. EXPERIMENTAL RESULTS

Regular data distributions include block distribution, cyclic distribution and block-cyclic distribution. Block and cyclic distributions are special cases of block-cyclic distribution. Given data distribution, parallelizing compilers have to partition computations and data across processors according to the programmer specified data distribution and *owner-computes* rule for execution on distributed-memory multicomputers. To do so, compilers have to evaluate local memory access sequences and generate communication sets. This paper has focused on general data distribution, block-cyclic distribution, and generation of necessary communication sets. To verify the advantages of our scheme, the techniques for generating communication sets for one index variable proposed in this paper and [5, 6] have been implemented on a DEC Alpha 3000 workstation. The viewpoint of block classification is similar to that of the virtual-block to virtual-block scheme proposed in [5, 6]. Therefore, we implemented the virtual-block to virtual-block scheme and denote this scheme as Gupta's method. A comparison of our technique and Gupta's [5, 6] has been extensively made. In this experiment, the times were measured by means of CPU time, and the time unit used was a microsecond.

Given two processors, $p_1 \in P_1$ and $p_2 \in P_2$, the time needed to generate $RP(p_1)$ and $PRD(p_1, q_1)$ for p_1 and $SP(p_2)$ and $PSD(p_2, q_2)$ for p_2 was measured. The time needed to generate the *class table* was also included in the measurement. What we were interested in was the impact on performance incurred by different values of the block size and the array section stride. Therefore, only the block size and the array section stride were varied; other values were fixed except additional specification. For simplicity, the block size (t), the array section stride (s), the number of processors (P), the number of array elements (n), and the lower bound (l) of the array section for both arrays were the same. Suppose the number of array elements was 30000 and suppose they were distributed across 3 processors using *cyclic*(t) distribution, where t is the block size. The array section was from 0, and the stride was s. Since the upper bound, u, was irrelevant to our experiment, we omitted the specification of u.

When the block size was fixed and equaled 12, the execution times varied with the array section stride as shown in Fig. 16. When the array section stride was smaller than or equal to the block size, the execution time of both ours and Gupta's scheme

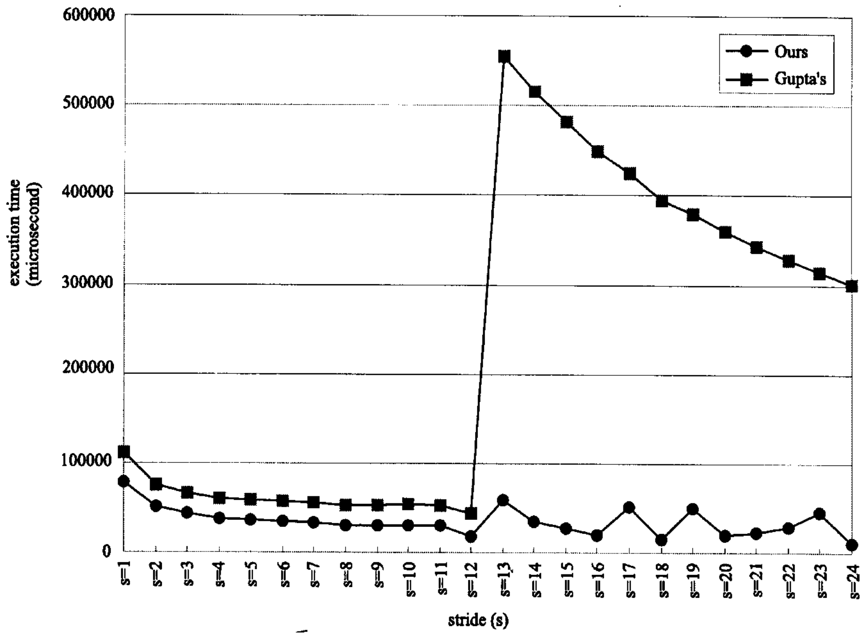


Fig. 16. The execution time of our technique and Gupta's when the block size was 12 and the array section stride varied from 1 to 24.

decreased with the increase of the array section stride. This is because the number of active elements decreased as the array section stride increased. However, a sudden change occurred in Gupta's scheme when the array section stride was larger than the block size. The execution times were much higher than those measured when the array section stride was smaller than or equal to the block size. Since a block contained at most one active element and continuous blocks did not necessarily contain one active element, as illustrated in Fig. 4, the technique proposed in [5, 6] directly calculated each element of the communication sets one by one. A large amount of computation was involved using this straightforward method. Fig. 16 also shows the fact that Gupta's method was very inefficient when the array section stride was larger than the block size. For our technique, since the class table gives enough information to generate communication sets no matter whether the array section stride is larger than, equal to or smaller than the block size, the phenomenon which occurred in Gupta's method did not occur in our scheme. However, when the array section stride was larger than the block size, the calculation of communication set was dependent of the number of classes in our method. That is why our method had slight vibrations when the array section stride was larger than the block size. Comparing the two methods, our method was always superior to Gupta's scheme, especially when the array section stride was larger than the block size.

On the other hand, we were curious about the influence of the block size on performance when the array section stride was fixed. Fig. 17 illustrates the variation of the execution time when the array section stride was fixed and the block size varied. For both methods, the bigger the block size was, the shorter was the execution time. This again confirmed the fact that when the array section stride was larger than the block size, the execution time of Gupta's scheme was much longer than that of our method. It is worth mentioning that when the block size equaled 1, in other words, the arrays were distributed using *cyclic* distribution, we observed that the execution time of Gupta's method was much longer than others. This means that Gupta's method is inefficient when the array is distributed using *cyclic* distribution.

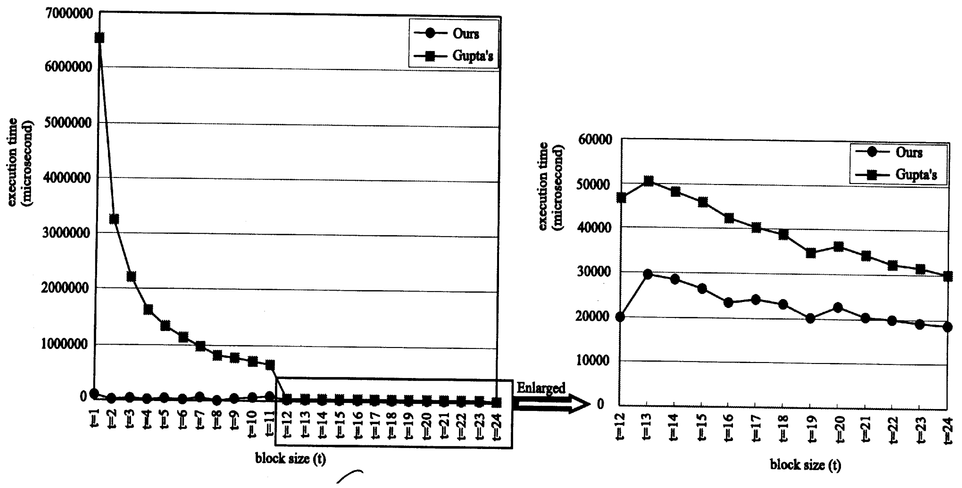


Fig. 17. The execution time of our technique and Gupta's when the array section stride was 12 and the block size varied from 1 to 24.

Previously, we have experimented on fixing either the array section stride on the block size. To better understand the impact on performance caused by different values of the array section stride and the block size, we also experimented using different values of *t* and *s*. Table 1 gives the *speedups* of our method against Gupta's for *t* = 1, 10, 50, 100, 500, 1000, 5000, 10000 and *s* = 1, 2, 3, 5, 7, 8, 10, 12, 15, 18, 20, 24, 25, where *speedups* were obtained from $\frac{Gupta's}{ours}$. From the experimental result, the bigger the block size was, the smaller was the speedup. Since the number of active elements contained in a block was larger if the block size was bigger, the overhead incurred by Gupta's method was smaller.

Table 1. Speedups of our technique against Gupta's for different values of block sizes and array section strides.

	t = 1	t = 10	t = 50	t = 100	t = 500	t = 1000	t = 5000	t = 10000
s = 1	2.42	1.33	1.20	1.22	1.11	1.09	1.15	1.13
s = 2	280.72	1.48	1.29	1.21	1.15	1.14	1.10	1.17
s = 3	158.86	1.53	1.30	1.24	1.18	1.12	1.24	1.52
s = 5	153.41	1.66	1.33	1.27	1.19	1.14	1.13	1.18
s = 7	124.17	1.72	1.37	1.32	1.14	1.13	1.11	1.48
s = 8	132.63	1.77	1.44	1.36	1.19	1.16	1.00	1.19
s = 10	97.71	2.31	1.48	1.32	1.23	1.21	1.10	1.00
s = 12	186.36	13.83	1.48	1.35	1.19	1.21	1.09	1.17
s = 15	154.84	16.88	1.52	1.40	1.19	1.11	1.19	1.18
s = 18	133.10	11.58	1.59	1.43	1.26	1.18	1.05	1.15
s = 20	57.51	27.46	1.58	1.45	1.28	1.18	1.16	1.16
s = 24	102.24	9.79	1.62	1.47	1.27	1.18	1.09	1.17
s = 25	47.37	18.04	1.64	1.49	1.31	1.16	1.06	1.23

We have analyzed the performance results when all the parameters for two arrays were the same. Table 2 gives the performance of our technique and Gupta's for different parameters to two arrays. The number of array elements was 30000, the number of processors was 3, and the lower bound of the array section was 0 for two arrays. We give not only the *speedup* but also the *improvement* to verify the advantages of our method against Gupta's, where the *improvement* was obtained from $\frac{Gupta's - ours}{Gupta's} * 100\%$. Although the parameters for the two arrays were different, the experimental results also display the same phenomena. The bigger the block size was, the smaller was the improvement or speedup obtained. When the array section stride was larger than the block size, the performance of our method was much better than that of Gupta's.

Table 2. Performance results of our technique and Gupta's and a comparison of ours with Gupta's for variations of *s* and *t*.

Parameters				Execution Time (μs)		Improvement	Speedup
t_1	s_1	t_2	s_2	Ours (O)	Gupta's (G)	$(\frac{G-O}{G} * 100\%)$	$(\frac{G}{O})$
1	1	1	1	188084.95	498482.25	62.27%	2.65
5	2	5	3	82657.44	165107.69	38.82%	1.64
5	4	7	5	63156.96	107926.08	41.48%	1.71
10	6	12	4	44085.92	68017.44	35.18%	1.54
15	10	20	5	25893.28	40279.52	35.72%	1.56
50	10	80	5	12600.16	16904.32	25.46%	1.34
100	12	120	10	9106.08	12258.56	25.72%	1.35
200	4	250	5	12483.04	15098.72	17.32%	1.21
300	3	500	5	11799.84	14640.00	19.40%	1.24
1000	10	800	8	5953.60	7388.32	19.42%	1.24
5000	7	4000	9	6080.48	6627.04	8.25%	1.09
10000	3	8000	7	5465.60	5934.08	7.89%	1.09
1	5	1	3	56432.32	15570363.00	99.64%	275.91
5	7	5	9	47170.08	1782283.38	97.35%	37.78
5	7	7	12	45110.72	1159546.50	96.11%	25.70
10	15	12	15	30753.76	528055.06	94.18%	17.17
15	18	20	25	18797.76	209713.13	91.04%	11.16

These experimental results confirm that our technique outperforms Gupta's in all cases. When the array section stride is smaller than or equals the block size, the amount of improvement obtained using our technique against Gupta's varied with different parameters, but our technique always outperformed Gupta's. In addition, our technique always exhibited superior advantages against Gupta's when the array section stride was larger than the block size.

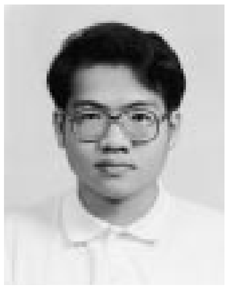
5. CONCLUSIONS

Automatic generation of efficient SPMD code for implementation of data-parallel programs on multicomputers is a very important issue for parallelizing compilers. In this paper, we have proposed a strategy to generate efficient SPMD codes for assignment statements in a data-parallel program. Given an array reference and an array in a block-cyclic distribution, we construct the *class table* to keep necessary information to assist both the evaluation of communication sets and the enumeration of local memory access sequences. The information in the *class table* is useful in reducing the overhead of translation between local indices and global indices when local memory access sequences are enumerated. In addition, calculation of the first and the last iterations in a block for evaluation of communication sets can also be handled by the *class table*. We have presented an algorithm for computing the essential information of the *class table*. The time complexity of this algorithm is $O(s/gcd(s, t))$. We have also implemented our technique and the virtual-block to virtual-block scheme proposed in [5, 6]. Extensive experiments were performed, and performance analyses have been given. The experimental results show the advantages of our technique against Gupta's, especially when the array section stride is larger than the block size. Furthermore, compiling of array references with multiple index variables also can be done by applying our method. The construction of *class tables* for compilation of array references with multiple index variables takes $O(s^2)$ time. The strategy for compiling array references with multiple index variables is easier and more efficient than those of existing methods. We believe that this method is feasible for compiling array references with multiple index variables.

REFERENCES

1. High Performance Fortran Forum, "High performance fortran language specification (version 1.0)," Technical Report CRPC-TR92225, Department of Computer Science, Rice University, 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*, Vol. 26, No. 1, 1995, pp. 72-84.
3. 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, 1991.
4. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang and P. Sadayappan, "On the generation of efficient data communication for distributed-memory machines," in *Proceedings of International Computing Symposium*, 1992, pp. 504-513.
5. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang and P. Sadayappan, "On compiling array expressions for efficient execution on distributed-memory machines," in *Proceedings of International Conference on Parallel Processing*, Vol. II, 1993, pp. 301-305.
6. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang and P. Sadayappan, "On compiling array expressions for efficient execution on distributed-memory machines," Technical Report OSE- CISRC-4/94-TR19, Department of Computer and Information Science, The Ohio State University, 1994.
7. S. Hiranandani, K. Kennedy and C.-W. Tseng, "Compiling fortran D for MIMD distributed-memory machines," *Communication of the ACM*, Vol. 35, No. 8, 1992, pp. 66-80.
8. S. Hiranandani, K. Kennedy, J. Mellor-Crummey and A. Sethi, "Compilation techniques for block-cyclic distributions," in *Conference Proceedings: 1994 International Conference on Supercomputing*, 1994, pp. 392-403.
9. K. Kennedy, N. Nedeljkovic and A. Sethi, "A linear time algorithm for computing the memory access sequence in data-parallel programs," Technical Report CRPC-TR94485-S, Center for Research on Parallel Computation, Rice

University, 1994.
10. C. Koelbel, “Compile-time generation of communication for scientific programs,” in *Conference Proceedings: 1991 International Conference on Supercomputing*, 1991, pp. 101-110.
11. E. M. Paalvast, H. J. Sips and A. J. van Gemun, “Automatic parallel program generation and optimization from data decompositions,” in *Proceedings of International Conference on Parallel Processing,* Vol. II, 1991, pp. 124-131.
12. J. M. Stichnoth, “Efficient compilation of array statements for private memory multicomputers,” Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, 1993.
13. J. Stichnoth, D. O’Hallaron and T. Gross, “Generating communication for array statements: Design, implementation, and evaluation,” *Journal of Parallel and Distributed Computing*, Vol. 21, No. 1, 1994, pp. 150-159.



Wen-Hsing Wei (韋文祥) received the B. S. degree in mathematics from National Taiwan University, Taiwan, the Republic of China, in 1993, and the M. S. degree in computer science and information engineering from National Central University, Taiwan, the Republic of China, in 1995.

His research interests include parallelizing compilers and parallel algorithms.

J.-P. Sheu (許健平) received the B.S. degree in computer science from Tamkang University, Taiwan, the Republic of China, in 1981, and the M. S. and Ph. D. degrees in computer science from the National Tsing Hua University, Taiwan, the Republic of China, in 1983 and 1987, respectively.

He joined the faculty of the Department of Electrical Engineering, National Central University, Taiwan, the Republic of China, as an Associate Professor in 1987. Since 1992, he has been a Full Professor in the Department of Computer Science and Information Engineering, National Central University. From March to June of 1995, he was a visiting researcher at the IBM Thomas J. Watson Research Center, New York. His current research interests include parallelizing compilers, and interconnection networks, and mobile computing.

Dr. Sheu is a senior member of the IEEE Computer Society, a member of the ACM, and the Phi Tau Phi Society. He is an associate editor of the Journal of Information Science and Engineering and the Journal of the Chinese Institute of Electrical Engineering, and the Journal of the Chinese Institute of Engineers. He received the Distinguished Research Awards of the National Science Council of the Republic of China in 1993-1994, 1995-1996, and 1997-1998.

Kuei-Ping Shih（石貴平）was born in Taiwan, the Republic of China, in 1969. He received the B.S. degree in mathematics from Fu-Jen University, Taiwan, the Republic of China, in 1991 and the Ph. D. degree in computer science from the Department of Computer Science and Information Engineering, National Central University, Taiwan, the Republic of China, in 1998.

His research interests include parallelizing compilers and parallel algorithms.