EFFICIENT ADDRESS AND COMMUNICATION GENERATION FOR DATA-PARALLEL PROGRAMS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Arun T. Venkatachar
B.E. in Electronics, Bangalore University, India, 1993
December 1996

Acknowledgments

I would like to express my sincere gratitude to Dr. Ramanujam for his guidance throughout this work. I am also grateful to him for having supported me financially throughout my stay here at LSU. His patience and constant encouragement gave me the drive to complete this work successfully. It was really a pleasure working with him.

I would also like to thank Dr. Subhash C. Kak and Dr. R. Vaidyanthan for serving on my committee.

Last but not the least, I would like to thank my parents for having stood by my side throughout my life. Their trust, love and constant encouragement in every stage of my life has made me the person I am today. I also would like to thank my friends here at LSU for making my stay here at LSU memorable.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Programming distributed memory machines has remained a difficult task in spite of their great potential. As a result, languages such as High Performance Fortran (HPF) have been designed to ease programming while preserving high levels of performance. A compiler for such a language is responsible for distributing data and computation onto the processors based on directives from the user and for generating node code along with sends and receives for each processor. This thesis addresses issues in code generation for programs with HPF-style data mappings.

Algorithms for deriving the communication sets for each processor are presented; these algorithms exploit a repeating communication pattern. In addition, local storage schemes using the idea of generalized overlap regions are presented for storing the received data on each processor; this results in increased data locality. In addition, communication optimizations such as offset communication, message coalescing and aggregation are discussed. Experiments show an average of 30% improvement in execution time due to overlap regions.

Address generation for two-level mapping resulting from non-unit alignment strides is handled in this thesis using simple ideas such as the use of tables and search techniques. These techniques are shown to be six to ten times faster in practice than existing methods. The search technique works extremely well for small strides and block sizes, while the table-based method works well for larger block sizes.

Code generation for programs with array references involving multiple induction variables (MIV) where the arrays are block-cyclically distributed is also addressed in this thesis. A collection of runtime techniques with varying space and time overheads are presented, and their impact on execution time is clearly demonstrated through experiments.

CHAPTER 1

INTRODUCTION

The ever-increasing need for performance has led researchers to design better and faster machines for the high performance computing community. Current parallel machines can be classified in many ways. One such classification based on memory leads to shared memory machines and distributed memory architectures. The shared memory models have a global memory which is accessible by every processor. Programming such machines is relatively easy as the programmer encounters a single address space as in the case of a sequential program. However these type of machines do not scale to a large number of processors. On the other hand processors in a distributed-memory system have their own memory and have no direct access to the memories on other processors; the processors are connected through a network. These machines are easily scalable and offer high levels of flexibility. The low level programming resulting from the need to explicitly partition data and computation, and explicit detailed message passing renders the task of programming these machines very difficult.

In order to ease the difficulty in programming for distributed memory machines, languages such as High Performance Fortran (HPF) [9] aim to raise the level of programming on distributed memory machines while retaining the object code efficiency derived from message passing. These languages include directives such as `ALIGN` and `DISTRIBUTE` that describe how data is distributed among processors. These directives act as hints to the compiler, which then takes care of performing the mapping of data and computations onto the distributed-memory multiprocessor and for generating the required message passing code. The programmer hence need not worry about the low-level details involved in the mapping. This eases the level of programming. These languages are designed independent of the underlying architecture and are hence portable.

1.1 DATA MAPPING

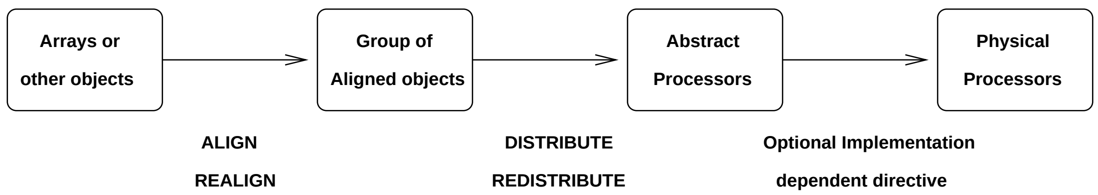


Figure 1.1: Model used for data mapping in HPF

Template	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Array	0		1		2		3		4		5		6		7	

Figure 1.2: Alignment of an array with the template

As mentioned earlier, a novel feature of HPF is that users (and perhaps tools) are allowed to specify the partitioning of data onto processors through compiler directives. Using these annotations compilers for distributed memory machines can easily generate node code for data parallel or SPMD (Single Program Multiple Data) programs, where each processor executes the same program text but calculates its own share of the global work on the distributed data structure. In the HPF model, there is a two-level mapping of data to processor memories referred to as *abstract* processors. Data objects (primarily arrays) are first aligned to an abstract discrete Cartesian grid called a *template*. This group of arrays is then distributed onto a rectilinear arrangement of abstract processors. Then there is an optional explicit mapping of abstract processors onto the physical processors. The mapping model is shown in Figure 1.1.

Example 1.1

```
        REAL A(20)
!HPF$   TEMPLATE T(40)
!HPF$   PROCESSORS PROCS(5)
!HPF$   ALIGN A(j) WITH T(2*j)
!HPF$   DISTRIBUTE T(CYCLIC) ONTO PROCS
        do i = 0, 15
             A(i) = ···
        enddo
```

The ALIGN directive specifies the alignment between the data objects and the *template*. The first step in HPF style data mapping involves aligning the given array to the *template*. In Example 1.1 an element of array $A(j)$ is aligned with $T(2 * j)$. This is an example of stride alignment. Figure 1.2 illustrates the alignment of array A with the *template*. Note that array elements $A(0), A(1), A(2), \cdots$ are aligned to the *template* with an alignment stride of 2. For cases when the alignment stride is unity, the mapping is referred to as *one-level* mapping. This corresponds to the case of identical alignment of the array to the template. Mappings with alignment strides whose magnitudes exceed 1 are referred to as *two-level* mappings.

The DISTRIBUTE directive specifies a mapping of data objects to abstract processors in a processor arrangement. This forms the next step after alignment in HPF style data mappings. Technically the distribution step of the HPF model applies to the *template*

Processor 0	Processor 1	Processor 2	Processor 3	Processor 4
0		1		2
	3		4	
5		6		7
	8		9	
10		11		12
	13		14	
15				

Figure 1.3: Distribution of the template cell to the processor arrangement

to which the array is ultimately aligned. A template may be distributed in either *block*, *cyclic*, or *block-cyclic* fashion. The *block* distribution specifies that the template should be distributed across a set of abstract processors by slicing it uniformly into blocks of contiguous elements. In the case of *cyclic* distribution, template elements are assigned to processors in a round-robin fashion. With block-cyclic distribution with block size k, referred to as $\texttt{CYCLIC}(k)$, blocks of the template of size k are allocated to processors in a wrap-around fashion. In Example 1.1 the template is assumed to be distributed `CYCLIC` onto a processor arrangement with five processors. Figure 1.3 shows this distribution for Example 1.1

The HPF compiler is responsible for generating an SPMD node code for the given HPF program. The node-code should specify the sequence of local memory addresses for each array accessed by each processor. Consider Processor 0 in Figure 1.3; the node-code should access local addresses of elements 0, 5, 10, 15 due to the *cyclic* distribution. If the distribution of elements uses the *block-cyclic* distributions local address generation becomes a difficult task. Many methods have been proposed to solve this problem of generating the address sequence in the case of *block-cyclic* distribution with one-level mapping. Communication may be required in order to provide access to nonlocal data. If the distribution of computation follows the *owner-computes* rule, *i.e.,* the processor performs only those computations (or assignments) for which it owns the left hand side variable, then access to nonlocal right hand side variables is provided by inserting sends and receives. In order to generate node code we need to compute the sequence of sends and receives for a given processor.

While the problem of address generation for one-level mapping has received much attention, problems in communication generation, address generation for non-unit alignment stride (two-level mapping) and loops with array references that contain multiple induction variables have not been dealt with extensively. This thesis aims at addressing

Table 1.1: Symbols used in this thesis.

A	a distributed array
T	the template to which array A is aligned
t	a template cell of template T
a	stride of alignment of A to the template T
b	offset of alignment of A to the template T
k	block size of distribution of the template
p	number of processors to which the template is distributed
ℓ	lower bound of the loop containing array A
h	upper bound of the loop containing array A
s	stride of access of array A
m	processor number
c	course number for the array layout
e	offset of the element in the course
course	course on which the array element lies
offset	offset of the element from the beginning of the course
column	offset of the element from the beginning of the course

issues in communication generation and optimizations along with presenting efficient techniques to address issues in address generation for two-level mappings and for the case when array statements contain multiple induction variables.

1.2 REVIEW OF LATTICE-BASED ADDRESS GENERATION

Consider the HPF program where an array A has been distributed *block-cyclic* onto 4 processors with a block size of 4 shown in Example 1.2

Example 1.2

```
REAL A(200)
```

Table 1.2: Definitions used in this thesis.

$(i \operatorname{div} k) \bmod p$	processor holding array element i
$(i \bmod pk) \operatorname{div} k$	processor holding array element i
$(i \operatorname{div} pk)$	course on which the array element lies
$(i \bmod pk)$	global offset of the element from the start of the course
$(i \bmod k)$	local offset of the element from the start of the course on the processor holding element i

Processor 0				Processor 1				Processor 2				Processor 3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143

Figure 1.4: Block-cyclic distribution of elements of array A onto $p = 4$ processors with $k = 4$ and $l = 0$

```
!HPF$   TEMPLATE T(200)
!HPF$   PROCESSORS PROCS(4)
!HPF$   ALIGN A(j) WITH T(j)
!HPF$   DISTRIBUTE T(CYCLIC(4)) ONTO PROCS
        do i = 0, 47
             A(3*i) = ···
        enddo
```

The array layout for Example 1.2 is shown in Figure 1.4. Chatterjee *et al.* [4] identified a repeating access pattern and characterized it as a finite state machine. They have shown that both the starting location and the table of local memory gaps can be found by solving a set of k linear diophantine equations. Let m be the processor which holds the array element $A(i)$; $A(i)$ belongs to a processor m if the offset in the course to which it belongs lies in the range $[km, k(m+1)-1]$. Each value of the offset corresponds to a column in the layout. If h and l are the lower and upper bounds of the loop containing array A and s is the access stride then every element that is accessed in the loop can be written as $(l+is)$ where i is a non-negative integer not exceeding $\left\lfloor \frac{h-l}{s} \right\rfloor$. The global offset of this element from the beginning of this course is given by $(l + is)$ mod pk. The course to which this element belongs is $c = (l + is) \operatorname{div} pk$. The element can now be represented using a course number and an offset within the course which correspond to rows and columns in a two-dimensional coordinate system. Hence the element can now be represented as

$$l + is = cpk + (l + is) \bmod pk$$

Hence

$$is - cpk = (l + is) \bmod pk - l$$

where

$$km \leq (l + is) \bmod pk \leq k(m + 1) - 1.$$

This is equivalent to finding an integer c such that

$$is - cpk = \lambda \quad where \quad mk - l \leq \lambda \leq mk - l + k - 1$$

which corresponds to solving k linear diophantine equations. Integer solutions exists if $\gcd(s, pk)$ divides λ. Chatterjee *et al.* [4] record the smallest non-negative solution for i for each value of λ. The minimum of all these solutions will correspond to the first array element $A(l + is)$ on processor m. Sorting of all the solutions will determine the local memory gaps between consecutive accesses. The complexity involved in solving the memory address problem using this method is found to be $O(k \log k + \min(\log s, \log pk))$. The first term is due to the sorting involved in order to find the memory gaps.

Consider Figure 1.4 which shows the distribution of array A onto 4 processors for the code in Example 1.2. The elements in boxes indicate the elements accessed due to stride $s = 3$. The alignment stride a is 1 for this example. If the method by Chatterjee *et al.* [4] is used to access elements for processor 0, their solution for each of the columns for which we get a nonnegative i is found to be 0, 33, 18, 3. The minimum for this set is 0 which give us the starting element for this processor. If we sort this set we get lexicographic set of accesses which help in building the finite state machine for this processor.

In order to reduce the complexity of this method Kennedy, Nedeljkovic and Sethi [12] and Thirumalai and Ramanujam [24, 25, 26, 27] view the address generation problem as an integer lattice and present algorithms to generate the set of accessed elements. Their method does not perform any sorting. They choose a lattice basis that allows for lexicographic enumeration of array accesses. Both the methods mainly differ in the way they construct the basis vectors for the lattice. They exploit the repeating nature of pattern accesses and construct pattern tables which generate the addresses of accessed elements during runtime.

Thirumalai and Ramanujam [24, 25, 26, 27] propose both runtime and compile time solutions to the address generation problem. Their runtime [27, 24, 25] technique constructs the basis vectors during runtime. They view the global address space as a two dimensional space. If B is the basis matrix of the two dimensional space and if α_1 and α_2 are the two dimensional components of the left vector of the lattice and β_1 and β_2 are the two dimensional components of the right vector of the lattice then we have

$$B = \begin{bmatrix} \alpha_1 & \beta_1 \\ \alpha_2 & \beta_2 \end{bmatrix}$$

It was shown [27] that the determinant of B is the access stride s itself. Given that the access pattern for the array repeats after every $\frac{s}{\text{gcd}\,(s,pk)}$ courses of elements, the pattern of accesses in the first $\frac{s}{\text{gcd}\,(s,pk)}$ courses can be used to determine the rest of the local addresses accessed. Using a transformation $T = B^{-1}$ and Fourier-Motzkin elimination [20, 31], we derive the code shown below for address pattern generation. The code determines the local and global addresses of the elements accessed on processor m for the first pattern of repetition. These elements are stored in pattern tables. The local addresses are stored in a table called *loc_pattern*$_m$ where as the global addresses are stored in *glob_pattern*$_m$. Here len gives the length of the pattern table for the first pattern of repetition. This code assumes that the basis vectors are already calculated. It uses these basis vectors to scan the lattice and to build the pattern tables for processor m.

$len = 0$
do $u = \left\lceil \frac{(-mk-k+1)\beta_1}{s_1} \right\rceil, \left\lfloor \frac{(s_1-1)\beta_2 - mk\beta_1}{s_1} \right\rfloor$
  if $\beta_1 = 0$
    $lower = \left\lceil \frac{mk-u\alpha_2}{\beta_2} \right\rceil; \; upper = \left\lfloor \frac{mk+k-1-u\alpha_2}{\beta_2} \right\rfloor$
  else
    $lower = \left\lceil \max\left(\frac{mk-u\alpha_2}{\beta_2}, \frac{-u\alpha_1}{\beta_1}\right) \right\rceil; \; upper = \left\lfloor \min\left(\frac{mk+k-1-u\alpha_2}{\beta_2}, \frac{s_1-1-u\alpha_1}{\beta_1}\right) \right\rfloor$
  endif
  do $v = lower\ ,\ upper$
    loc_pattern$_m[len] = (\alpha_1 u + \beta_1 v)k + \alpha_2 u + \beta_2 v - mk$
    glob_pattern$_m[len] = (\alpha_1 u + \beta_1 v)pk + \alpha_2 u + \beta_2 v$
    $len = len + 1$
  enddo
enddo

These pattern tables are constructed only once and are later used to generate the remaining set of elements accessed during execution.

1.3 ORGANIZATION OF THIS THESIS

This thesis is organized as follows. In Chapter 2 we discuss issues arising in communication generation. We present techniques to generate the communication sets. We present several local storage schemes for handling nonlocal elements. In addition, techniques to optimize communication are addressed. We demonstrate the efficacy of our approach using experimental results. Chapter 3 proposes methods to enumerate the set of accesses for the case of two-level mapping. Our model does not incur any memory wastage and results show that the methods presented in this thesis results in the best times for address generation. In order to generate array accesses for array statements which contain multiple induction variables we present efficient runtime techniques in Chapter 4. This also proposes a new method of computing the first element accessed on each processor for a given global starting element. Chapter 5 concludes by summarizing all the above discussed methods and also leading to immediate future work.

CHAPTER 2

COMMUNICATION GENERATION AND OPTIMIZATION

In this chapter we address several issues in communication generation and optimization for data parallel languages. Communication generation involves determining the elements to be sent from the source processor to the destination processor. It also involves computing the set of elements a processor needs to receive. Generating this *send* and *receive* sets forms the first part of our discussion. We then present two novel methods namely *course* and *column* padding to provide storage for received elements in the local space. These methods incur very small overheads due to unpacking but help in enhancing the data locality. The performance of these methods are compared to the traditional buffer-based approach. Several message optimizations such as offset communication, message aggregation and coalescing are also discussed.

2.1 PROBLEM STATEMENT

Consider the HPF code in the following example

Example 2.1

```
        REAL A(N), B(N)
!HPF$   PROCESSORS PROCS(p)
!HPF$   ALIGN A AND B WITH T
!HPF$   DISTRIBUTE T(CYCLIC(k)) ONTO PROCS
```

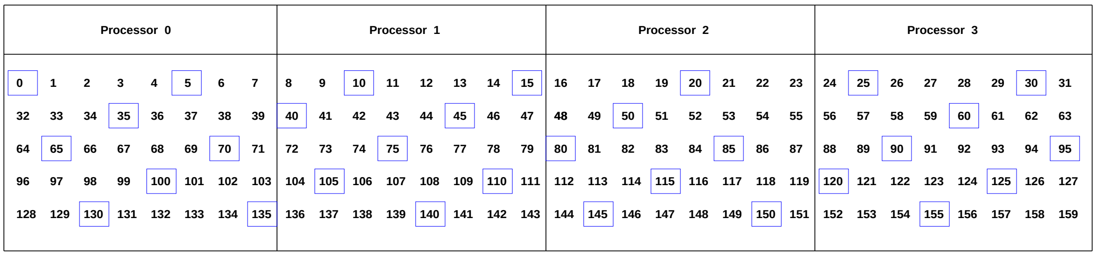
do $i = 0, \left\lfloor \frac{h_1 - l_1}{s_1} \right\rfloor$

$$A(l_1 + is_1) = \mathcal{F}\left[B(l_2 + is_2)\right]$$

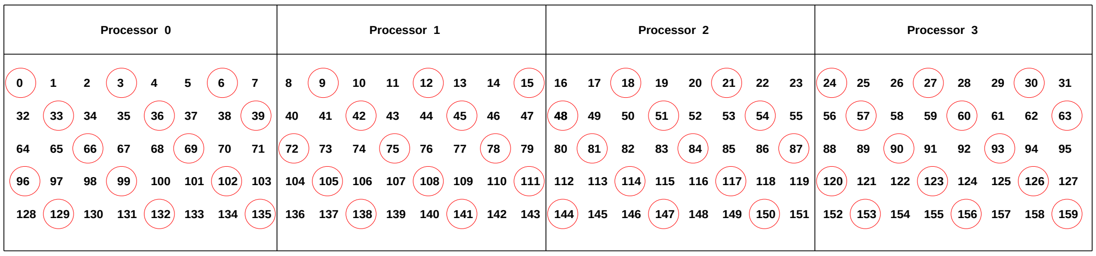
enddo

where s_1 and s_2 are the access strides of arrays A and B respectively. If the computation follows the *owner-computes* rule, access to nonlocal right hand side variables is achieved by inserting sends and receives. In order to generate node code we need to compute the sequence of sends and receives of elements of B for a given processor to access nonlocal data in addition to computing the sequence of local memory addresses accessed by each processor. In addition, the compiler has to manage storage for nonlocal references and exploit data locality in code execution whenever possible.

Figures 2.1(a) and (b) show an example of two arrays A and B distributed CYCLIC(8) onto 4 processors for the given loop. Elements in boxes are the accessed elements due to strides s_1 = 5 and s_2 = 3. It is our aim now to determine the set of elements involved in communication. If we consider processor 0 for the LHS array as shown in Figure 2.1(a) we see that global elements 0, 5, 35, 65, 70, 100, 130, 135 are accessed in iterations 0, 1, 7, 13, 14, 20, 16 before the pattern starts repeating. The corresponding array elements for the RHS for the above iteration set are 0, 3, 21, 39, 42, 60, 78, 81. We see that not all



(a): Global addresses of array A accessed for $p = 4$, $k = 8$, and $s_1 = 5$.



(b): Global addresses of array B accessed for $p = 4$, $k = 8$, and $s_2 = 3$.

Figure 2.1: Global addresses of arrays A and B for $A(5i) = B(3i)$ when $p = 4$, $k = 8$, $s_1 = 5$, $s_2 = 3$, $l_1 = 0$ and $l_2 = 0$

elements of the RHS set are located on processor 0. Hence we need to determine this set of elements to be received in order to execute code locally. The same argument holds true for the array on the RHS, which needs to determine the set of elements which it needs to send in order to complete the communication cycle.

Further issues have to addressed with regard to treatment of storage for nonlocal elements. This chapter deals with presenting novel techniques to generate the communication sets and new methods for storage of nonlocal data. The method of choice adopted to store the nonlocal references leads to increased data locality and thus faster execution. The generation of communication sets is explained first and the following sections deal with the concept of a generalized overlap region for storing nonlocal references. Several optimizations such as offset communication, message coalescing and aggregation are also presented.

2.2 Related work on communication generation

Several solutions have been proposed to address the code generation problem for distributed memory machines [1, 2, 4, 8, 12, 14, 15, 21, 22, 23, 24, 26, 27, 29, 31]. Many of these are good schemes for address generation but issues in communication optimization have not received much attention. Chatterjee *et al.* [4] describe the set of accesses as a finite state machine (FSM) and use the FSM to generate communication. Their model of communication is based on packing elements using an address-value pair before sending. The receiving processor does not have to unpack messages; they use message buffers for the combined receive-execute phase. The code uses expensive conditionals and may suffer from locality when a processor communicates with a large number of processors. Extensions to multiple right hand side arrays and accesses have not been dealt with. Stichnoth [22] and Gupta *et al.* [8] discuss similar schemes for communication.

Kennedy, Nedeljkovic and Sethi [14] view the set of accesses as an integer lattice and make use of the repeating set of accesses. Their communication scheme performs packing of elements before communication. On the receive side, each processor allocates overflow regions at the end of local storage (for each array) for storing nonlocal accesses. They then construct a table of accesses to index into the local as well as the overlap region. Their code does not require unpacking of messages but requires the added overhead of recomputing table entries; of the two tables of length k that they use, the entries in one are updated after every array access (*i.e.,* in each iteration). In effect a processor that performs N local computations computes a total of N addresses for each right hand side reference, computed using the two tables of length k.

Communication generation issues are also dealt with by Thirumalai [24] and by Thirumalai, Ramanujam and Venkatachar [29]. The work in this chapter builds on the work proposed by these methods. In contrast to these methods, we present storage schemes for nonlocal references that help maintain data locality while reducing runtime overhead with `CYCLIC`(k) distributions. In particular, re-computation of the address pattern for references is eliminated. Our approach of providing a storage area for nonlocal references was inspired by Gerndt's work [6] on overlap regions in the context of shift communication with `BLOCK` distribution. Also, we present adaptations of these storage schemes for several communication optimizations.

2.3 Communication set generation

Communication set generation for each processor involves computing the set of elements to be sent, the processors to which these elements are sent, the set of elements received and the processors from which these are received. We show that there is a pattern of communication which can be exploited in communication set generation. We then present algorithms for deriving these communication patterns. The send communication pattern stores the indices of the destination processor and the set of elements to be sent before the communication pattern starts repeating. The receive communication pattern on the other hand stores the indices of the processor sending the data and the addresses of the element to be received. These communication patterns are used to derive both the send and receive communication sets for each processor. In the next sections, we use these pattern to pack and unpack messages.

2.3.1 Receive pattern generation

$RIndPattern_0$	0	3	21	39	42	60	78	81
$RProcPattern_0$	0	0	2	0	1	3	1	2
$RPattern_0$	0	3		15				

Figure 2.2: Receive pattern generation tables for Processor 0 when $p = 4$, $k = 8$, $s_1 = 5$, $s_2 = 3$, $l_1 = 0$ and $l_2 = 0$ for $A(5i) = B(3i)$

In order to exploit the communication pattern between a pair of processors, we need to determine the complete set of elements involved in communication which repeats itself in the global space. The access pattern of the LHS array in the global address space repeats itself after pk elements are accessed, *i.e.,* after s_1 rows, where s_1 is the access stride of the LHS array. In order to generate the receive set, we run the code generated as described in Section 1.2 to obtain the global address of the first pk elements accessed on the LHS. For each of these elements we calculate the processor that owns the corresponding RHS element. The pattern of elements which are needed to perform computation and are local to the processor are also computed.

Let *RIndPattern*$_m$ and *RPattern*$_m$ tables store the global and local indices of the elements on RHS which are needed by the LHS before the communication pattern starts repeating for processor m. Let *RProcPattern*$_m$ store the processors which hold the RHS element needed by the LHS. Let len_l be the length of the communication pattern. The receive pattern generation code is shown below.

do $j = 0, len_l - 1$
    $index_l = glob\_LHS\_pattern_m[j]$
    $RIndPattern_m[j] = index_r = \frac{s_2}{s_1}(index_l - l_1) + l_2$
    $RProcPattern_m[j] = (index_r \text{ div } k) \text{ mod } p$
    if $m = RProcPattern_m[j]$
        $RPattern_m[j] = k * (index_r \text{ div } pk) + index_r \text{ mod } k$
    endif
enddo

For each accessed element in the LHS we first determine the corresponding global index of the RHS element. If $index_l$ is the global index of the LHS element we first find the global iteration corresponding to this element by finding $\frac{index_l - l_1}{s_1}$. We then multiply this entity with the RHS stride in order to obtain the corresponding RHS element. We store these entries in the *RIndPattern* table. The next step in the algorithm is to find the processor which holds this RHS element. If $index_r$ is the index of the *rhs* element $(index_r \text{ div } k) \text{ mod } p$ is the processor which holds this element. We store all these entries in a separate table

$SIndPattern_0$	0	3	6	33	36	39	66	69
$SProcPattern_0$	0	0	1	2	3	0	1	2
$SPattern_0$			6	9	12		18	21

Figure 2.3: Send pattern generation tables for Processor 0 when $p = 4$, $k = 8$, $s_1 = 5$, $s_2 = 3$, $l_1 = 0$ and $l_2 = 0$ for $A(5i) = B(3i)$

called the *RProcPattern* table. If the processor holding this RHS element is m, only then we update the *RPattern* table. This table holds only the local indices of RHS elements which are present on processor m. This is due to the fact that, during execution we lookup a separate table which holds the nonlocal elements. We need the *RPattern* table only to provide the indices of local elements.

Figure 2.2 shows the entries in the receive pattern tables needed in generating the receive set for processor 0. We see that the entries in $RIndPattern_0$ correspond to global indices of the RHS elements involved in communication. The entries in $RProcPattern_0$ indicate the processors which hold these RHS entries. We also see that the $RPattern_0$ table holds only the local addresses of elements which belong to processor 0.

2.3.2 SEND PATTERN GENERATION

Our approach to the send set generation is very similar to the receive set generation code in Section 2.3.1. It should be obvious that there is a one to one correspondence between the LHS array element and the RHS array elements. So the send pattern for a processor is the same as the receive pattern calculated assuming that the RHS element is actually the LHS element. Here again we notice that the send pattern should repeat after every s_2 rows.

Let *SIndPattern*$_m$ and *SPattern*$_m$ tables store the global and local indices of the elements on RHS which are accessed before the communication pattern starts repeating for processor m. Let *SProcPattern*$_m$ store the indices of processors to which processor m sends data. Let len_r be the length of the communication pattern. The pattern generation code is shown below.

do $j = 0, len_r - 1$
  $index_r = glob\_RHS\_pattern_m[j]$
  $SIndPattern_m[j] = index_l = \frac{s_1}{s_2}(index_r - l_2) + l_1$
  $SProcPattern_m[j] = (index_l \text{ div } k) \bmod p$
  if $m \neq SProcPattern_m[j]$
    $SPattern_m[j] = loc\_RHS\_pattern_m[j]$

endif
enddo

The algorithm in this case, determines the global index of the RHS element on processor m. For each such element we determine the corresponding LHS element by first determining the iteration that accesses the RHS element and then multiplying this by the LHS stride. We maintain a table called *SIndPattern* which stores the indexes of these LHS elements. We maintain a *SProcPattern* table which holds the index of the processor which holds the LHS element. A separate table called the *SPattern* table is now maintained which stores only the local indices of the RHS element to be sent from processor m to other processors. There is no need to store the information for local elements.

Figure 2.3 shows the required data structures to maintain the information of the send set of processor 0. The first table $SIndPattern_0$ contains global accessed elements of array B for processor 0. The corresponding processors which receive these elements are stored in $SProcPattern_0$ table. The $SPattern_0$ table holds only the local addresses of elements which needs to be sent from processor 0 to all other processors.

2.4 GENERALIZED OVERLAP REGIONS FOR BLOCK-CYCLIC DISTRIBUTIONS

In order to perform computations, a processor needs to receive all the necessary nonlocal data using interprocessor communication. we assume that the compiler uses the owner-computes rule in generating code to perform all the sends and receives. Communication between processors requires packing of data into buffers before being sent from one processor to another. At the receive end, the processor can either unpack the nonlocal elements in the buffers into its local array address space before execution or can read directly from the receive buffers during execution. Unpacking elements into the local address space involves providing additional space in the local memory for the processor which receives nonlocal data.

Reading data directly out of buffers incurs no unpacking overhead but requires expensive conditionals to be executed in each iteration. In order to overcome this drawback, one can unpack the elements into overlap areas before execution. This unpacking of messages will not result in expensive conditionals and lead to improved locality of reference. In this section, we discuss two such strategies for unpacking the received elements into overlap regions.

Our algorithm unpacks the data from buffers into the local address space in order to exploit spatial locality. Since we provide storage for overlap regions, the notion of the local array has been extended; this results in a need to recompute the *RPattern*$_m$ and *SPattern*$_m$ tables. Data locality during execution is enhanced because of the way we allocate storage for nonlocal accesses.

2.4.1 Course padding for overlap regions

As noted earlier, the communication pattern for an array assignment statement repeats after pk elements are accessed over the p processors. Hence the access pattern for the LHS array repeats after s_1 courses while that of the RHS array repeats after s_2 courses. For the example in Figure 2.1 the LHS pattern repeats after 5 courses, whereas the RHS pattern repeats after 3 courses. On each processor we execute at most k local iterations before the communication pattern repeats; some RHS elements are local and others nonlocal. For each set of s_2k RHS elements, we allocate space equal to the number of nonlocal elements (referred to as *num_nonloc* in the following pages) in the receive index pattern. In the worst case every one of the k local iterations may require nonlocal data; thus, we might have to allocate space for at most k elements for each block of s_2 RHS courses. With course padding, after each set of s_2k local elements we allocate storage for holding nonlocal elements. Consider processor 0 in the example shown in Figure 2.1. Figure 2.4 shows padding an array with courses for this example. Elements 0, 3, 39 in Figure 2.4(a) are the global addresses of local elements accessed on RHS for processor 0. Elements 21, 42, 60, 78, 81 in Figure 2.4(a) are the global addresses of nonlocal elements received due to stride s_1 of the LHS for processor 0. Figure 2.4(b) shows the local addresses of access patterns of both local and nonlocal elements for the same example. The reason for allocating extra storage after each block of s_2 courses on the RHS is to increase locality during execution. Note that padding courses in the RHS array results in a change in the addresses of the locally allocated portion of an array on a processor.

The algorithms for packing, unpacking and execute phases for lexicographic execution are shown below. We assume that the layout of array on the RHS takes into account the overlap regions. We also assume that the *RPattern*$_m$ and *SPattern*$_m$ tables have been recomputed for this type of storage; *num_nonloc* is the number of nonlocal elements (for each block of the communication pattern) to be padded in the new storage space before the communication pattern starts repeating. The value of *num_nonloc* is computed based on the information in the *RProcPattern*$_m$ table. Array elements received from processor q are in *ReceiveBuffer*$_q$ for each processor q where $0 \leq q \leq p-1$. *Apattern* stores the local access pattern of array A. Note that max_A is the local address of the last element of A accessed on processor m. In order to keep the presentation simple, we only present the algorithms for a full set of blocks; the code for the last incomplete block is simple and is not discussed here.

Let us consider Processor 0 from Figure 2.1. The algorithm needs to pack the elements in the array on the RHS into the *ReceiveBuffers*$_q$ where q is the index into the *SProcPattern*$_0$ table. Let max_B be the local address of the last element of B accessed on processor m. This constitutes the packing phase (Step 3), where, processor 0 scans its send patterns and for each nonlocal element i of *SPattern*$_0$, processor 0 packs all elements in the array section $(B[\text{SPattern}_0[i]] : B[max_B] : B[s_2k + \text{num\_nonloc}])$ into contiguous locations in the *SendBuffer* corresponding to the entry in *SProcPattern*$_0[i]$. This necessarily means that we make as many scans as the number of nonlocal elements in *SPattern*$_0$ before we pack all elements. Figure 2.5 shows the set of elements which are packed into *SendBuffers*. The

Processor 0

0	1	2	3	4	5	6	7
32	33	34	35	36	37	38	39
64	65	66	67	68	69	70	71
21	42	60	78	81			

(a): Access pattern using global addresses.

Processor 0

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28			

(b): Access pattern using local addresses.

Figure 2.4: RHS elements on processor 0 (using padded courses) for $A(5i) = B(3i)$ when $p = 4$ and $k = 8$.

SendBuffer$_1$	6	35	64	93	122	18	47	76	105	134
SendBuffer$_2$	9	38	67	96	127	21	50	79	108	137
SendBuffer$_3$	12	41	70	99	128					

BPattern	0	3	24	15	25	26	27	28

B(24:144:29) = *ReceiveBuffer*$_2$(0 : 4)
B(25:144:29) = *ReceiveBuffer*$_1$(0 : 4)
B(26:144:29) = *ReceiveBuffer*$_3$(0 : 4)
B(27:144:29) = *ReceiveBuffer*$_1$(5 : 9)
B(28:144:29) = *ReceiveBuffer*$_2$(5 : 9)

LHS	0	5	11	17	22	28	34	39	40	45	51	57	62	68	74	79	...
RHS	0	3	24	15	25	26	27	28	29	32	53	44	54	55	56	57	...

Figure 2.5: Illustration of packing, unpacking and execution for $A(5i) = B(3i)$ on processor 0 when $h_1 = 795$, $h_2 = 477$, $p = 4$ and $k = 8$.

array section involved in packing the first element to be sent to Processor 1 is (6 : 477 : 29), where 29 is the total number of elements including the space allocated for the 5 nonlocal elements before the pattern starts repeating as shown in Figure 2.4(b).

Once all the elements to be sent have been packed, communication takes place. We do not show the communication code here. At the receiving end each processor receives all the elements from other processors in the corresponding *ReceiveBuffers*. For each access on the LHS for the first repeating set we compute and record (Step 2) the corresponding entry from the new storage space of the RHS into a table of access called the *BPattern*$_m$ table. This table contains the sequence of accesses of both local and received data for the first repeating set; these entries are in lexicographic order. This table is constructed only once before the unpacking stage and does not require updates such as those required by the algorithm due to Kennedy, Nedeljkovic and Sethi [14]. For the above example the entries in the *BPattern*$_m$ table are shown in Figure 2.5. Entries 0, 3, and 15 correspond to local elements whereas 24, 25, 26, 27, and 28 correspond to nonlocal elements. These entries correspond to the first repeating set of the communication pattern. Step 2 illustrates the code to compute the table. We record consecutive locations into *BPattern*$_m$ table for nonlocal elements, since course padding schemes allocates consecutive storage for these elements.

In the unpacking stage, (Step 6) entries in the *BPattern*$_m$ table are used to determine where the incoming elements are to be unpacked. For each entry in the $RProcPattern_m$ table we obtain the index of the corresponding *ReceiveBuffer* from which the received elements are used to unpack. With each entry of the $BPattern_m$ table, we now obtain the address to which we need to unpack the received element. For each received element i, the array section into which we unpack the elements is given by $(B[\text{BPattern}_m[i]] : B[max_B] : B[s_2k + \text{num\_nonloc}])$. Figure 2.5 shows the array sections of memory into which the elements in the *ReceiveBuffers* have been unpacked. Here again we make as many scans of the local address space as the number of nonlocal elements. Figure 2.5 shows the array sections into which consecutive elements from the *ReceiveBuffers* are unpacked.

Once all the elements are unpacked into the local address space, the execution stage (Step 7) involves the *APattern*$_m$ and *BPattern*$_m$ tables whose entries will determine the location of elements on LHS and RHS needed for execution. Figure 2.5 illustrates local addresses of LHS and RHS which are accessed during execution using course padding scheme. All the addresses are in lexicographic order. The complete algorithm which performs packing, unpacking and execution on processor m is shown below.

Step 1: The entire local address space is assumed to be reorganized in order to account for the overlap regions. We allocate extra storage equal to the total number of nonlocal elements to be received.

Step 2: Code to compute a table to store the locations of both local and nonlocal accesses on RHS. This table will be used during both unpacking and execution.

index $= s_2k$

do $j = 0, \mathit{sizeof}(\mathit{RProcPattern}_m) - 1$
$dest = \mathit{RecvProcPattern}_m[j]$
if $(m = dest)$
$\mathit{BPattern}[j] = \mathit{RPattern}_m[j]$
else
$\mathit{BPattern}[j] = \mathit{index}$
$\mathit{index} = \mathit{index} + 1$
enddo

Step 3: This stage is executed by processor m to pack all the elements which it needs to send to other processors. The code involves tables *SPattern* and *SProcPattern* which determine the local address of the element to be sent as well as the processor to which it needs to be sent as discussed in Section 2.3.2. For each element to be sent it first determines the *SendBuffer* corresponding to the processor which receives it. It then packs this element into the *SendBuffer*. This procedure is followed until all the elements to be sent are packed into the respective *SendBuffers*.

$\mathit{Scount}_q = 0$, where $0 \leq q \leq p - 1$
$\mathit{baseadd} = 0$
$\mathit{blksize}_B = s_2 k + \mathit{num\_nonloc}$
do $i = 0, \mathit{sizeof}(\mathit{SProcPattern}_m) - 1$
$q = \mathit{SProcpattern}_m[i]$
if $m \neq q$
$loc_B = \mathit{SPattern}_m[i]$
do $j = 0, \frac{max_B}{\mathit{blksize}_B}$
$\mathit{address} = \mathit{baseadd} + loc_B$
$\mathit{SendBuffer}_q[\mathit{Scount}_q] = B_{loc}[\mathit{address}]$
$\mathit{Scount}_q = \mathit{Scount}_q + 1$
$\mathit{baseadd} = \mathit{baseadd} + \mathit{blksize}_B$
enddo
endif
enddo

Step 4: Send data in the *SendBuffers* to the respective processors.

Step 5: Receive data from the processors into the appropriate *ReceiveBuffers*

Step 6: This stage involves receiving all the nonlocal data into the overlap regions allocated in the local address space. The first step involves reading an entry from the *RProcPattern* which provides the index to the corresponding *ReceiveBuffer*. The next step involves reading an entry from the *BPattern* table to determine the location to which we unpack the received element. This table as explained before keeps track of the locations where the nonlocal data are to be placed. For each entry in the *ReceiveBuffer* we unpack the element into the overlap regions. This procedure is followed until all the elements are unpacked into the overlap areas.

$count_q = 0$ for $0 \leq q \leq p-1$
do $i = 0, sizeof(RProcPattern_m) - 1$
  $q = RProcPattern_m[i]$
  $loc_B = BPattern_m[i]$
  do $j = 0, \frac{max_A}{s_1 k}$
    $Bloc[loc_B] = ReceiveBuffer_q[count_q]$
    $count_q = count_q + 1$
    $loc_B = loc_B + s_2 k + num\_nonloc$
  enddo
enddo

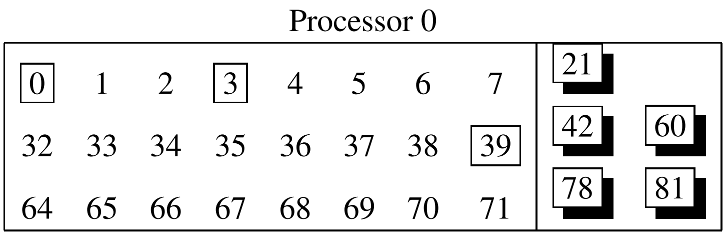
Step 7: This stage performs lexicographic execution of elements on processor m. Since all the received elements are already unpacked into local regions, this stage involves the tables *APattern* and *BPattern* whose entries determine the addresses of the LHS and RHS array elements accessed in the computation and then performing the computation. This procedure is followed until all the elements are executed.

$baseadd_A = baseadd_B = 0$
do $j = 0, \frac{max_A}{s_1 k}$
  do $i = 0, sizeof(RProcPattern_m) - 1$
    $loc_A = APattern_m[j] + baseadd_A$
    $loc_B = BPattern_m[j] + baseadd_B$
    $Aloc[loc_A] = \mathcal{F}\ (Bloc[loc_B])$
  enddo
  $baseadd_A = baseadd_A + s_1 k$
  $baseadd_B = baseadd_B + s_2 k + num\_nonloc$
enddo

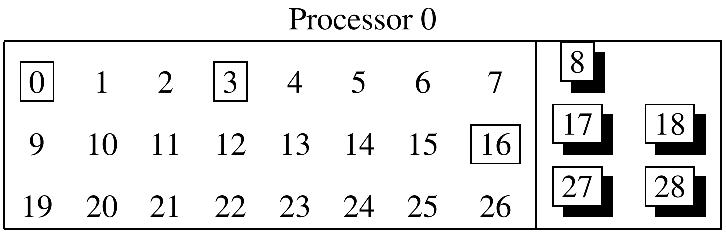
Figure 2.5 shows the three main stages of the algorithm, packing, unpacking and execution for processor 0. The *SendBuffer* tables shows the order in which elements are packed. Elements 6, 35, 64, 93, $\cdots$ are the set of elements which are being sent to processor 1. Similarly the second table *BPattern* holds addresses of locations where the received elements are to be stored. The next set of tables indicate the process of unpacking. As mentioned earlier the unpacking is performed into the overlap region. The last stage shows the local addresses of the LHS and RHS on processor 0 which are involved in the execution stage.

2.4.2 COLUMN PADDING FOR OVERLAP REGIONS

With the column padding scheme we again allocate local storage for nonlocal RHS elements that are accessed. Unlike the previous unpacking strategy where the nonlocal elements



(a): Access pattern using global addresses.



(b): Access pattern using local addresses.

Figure 2.6: RHS elements on processor 0 (using padded columns) for $A(5i) = B(3i)$ when $p = 4$ and $k = 8$.

were unpacked into overlap regions after each set of s_2k elements, the overlap region is in the form of extra local storage within each block of s_2k elements of the local address space. The total number of extra storage space that are padded to the RHS array is equal to the number of nonlocal elements being accessed. There can be storage for at most k extra elements before the pattern starts repeating. The addresses where the received element is to be unpacked depends on the address of the element being received. Figures 2.6(a) and (b) shows an example of padding an array with columns.

We assume that the layout of array on the RHS takes into account the overlap regions. We also assume that the *RPattern*$_m$ and *SPattern*$_m$ tables have been recomputed for this type of storage; *num_nonloc* is the number of nonlocal elements (for each block of the communication pattern, *i.e.,* the number of nonlocal elements in $RPattern_m$) to be padded in the new storage space before the communication pattern starts repeating. Array elements received from processor q are in *ReceiveBuffer*$_q$ for each processor q where $0 \leq q \leq p - 1$. *Apattern* stores the local access pattern of array A. Note that max_A is the local address of the last element of A accessed on processor m. Note that, in order to keep the presentation simple, we only present the algorithms for a full set of blocks; the code for the last incomplete block is simple and is not discussed here.

The algorithm for packing (Step 3) is exactly similar to the one discussed in Section 2.4.1. It is only the unpacking and execution stages that differ from those of course padding scheme. Figure 2.7 shows the elements padded into the *SendBuffers*.

SendBuffer$_1$	6	35	64	93	122	21	50	79	108	137
SendBuffer$_2$	10	39	68	97	126	24	53	82	111	140
SendBuffer$_3$	13	42	71	100	129					

BPattern	0	3	8	16	17	18	27	28

B(8:144:29) = *ReceiveBuffer*$_2(0:4)$
B(17:144:29) = *ReceiveBuffer*$_1(0:4)$
B(18:144:29) = *ReceiveBuffer*$_3(0:4)$
B(27:144:29) = *ReceiveBuffer*$_1(5:9)$
B(28:144:29) = *ReceiveBuffer*$_2(5:9)$

LHS	0	5	11	17	22	28	34	39	40	45	51	57	62	68	74	79	...
RHS	0	3	8	16	17	18	27	28	29	32	37	45	46	47	56	57	...

Figure 2.7: Illustration of packing, unpacking and execution for $A(5i) = B(3i)$ on processor 0 when $h_1 = 795$, $h_2 = 477$, $p = 4$ and $k = 8$.

At the receiving end however, we compute a table of access (Step 2) similar to the *BPattern* table as discussed in the previous section for the RHS to record the sequence of accesses to both local and received data for the first repeating set; these entries are in lexicographic order. The addresses computed depends on the element to be received; whose information can be found from the pattern tables discussed earlier. Figure 2.7 shows the entries held by the *BPattern* table.

In the unpacking stage (Step 6) for each entry in the $RProcPattern_m$ we first determine the corresponding *ReceiveBuffer* from which we obtain the received elements. With each entry from the *BPattern*$_m$ table, we now determine the location to which we need to unpack the received element. For each received element i we unpack all the received elements to locations in the local address space using the array section $(B[\textit{BPattern}_m[i]] : B[max_B] : B[s_2k + \textit{num\_nonloc}])$. The unpacking strategy is similar to the one discussed in the previous section. However the array sections involved in unpacking here are different from that of course padding. Figure 2.7 shows the array sections into which the received elements are unpacked. The entries in the *ReceiveBuffers* are exactly the same as in course padding.

The final stage (Step 7) involves execution with tables *APattern*$_m$ and *BPattern*$_m$ whose entries determine the addresses of accessed elements on LHS and RHS needed for computation. Figure 2.7 illustrates local addresses of LHS and RHS which are accessed during execution using column padding scheme. The complete algorithm which performs packing, unpacking and execution on processor m is shown below.

Step 1: The entire local address space is assumed to be reorganized in order to account for the overlap regions. We allocate extra storage equal to the total number of nonlocal elements to be received.

Step 2: Compute *BPattern* which is the table that stores local addresses of the RHS elements needed including elements received from other processors.

Step 3: This stage is executed by processor m to pack all the elements which it needs to send to other processors. The code involves tables *SPattern* and *SProcPattern* which determine the local address of the element to be sent and the processor to which it needs to be sent as discussed in Section 2.3.2. For each element to be sent it first determines the *SendBuffer* corresponding to the processor which receives it. It then packs this element into the *SendBuffer*. This procedure is followed until all the elements to be sent are packed into the respective *SendBuffers*.

$\mathit{Scount}_q = 0$, where $0 \leq q \leq p-1$
$\mathit{baseadd} = 0$
$\mathit{blksize}_B = s_2 k + \mathit{num\_nonloc}$
do $i = 0, \mathit{sizeof}(\mathit{SProcPattern}_m) - 1$
  $q = \mathit{SProcpattern}_m[i]$
  if $m \neq q$
    $\mathit{loc}_B = \mathit{SPattern}_m[i]$
    do $j = 0, \frac{max_B}{\mathit{blksize}_B}$
      $\mathit{address} = \mathit{baseadd} + \mathit{loc}_B$
      $\mathit{SendBuffer}_q[\mathit{Scount}_q] = B_{loc}[\mathit{address}]$
      $\mathit{Scount}_q = \mathit{Scount}_q + 1$
      $\mathit{baseadd} = \mathit{baseadd} + \mathit{blksize}_B$
    enddo
  endif
enddo

Step 3: Send data in the *SendBuffers* to the respective processors.

Step 4: Receive data from the processors into the appropriate *ReceiveBuffers*

Step 5: This stage involves receiving all the nonlocal data into the overlap regions allocated in the local address space. The first step involves the *RProcPattern* table whose entries determine the corresponding *ReceiveBuffer* which stores the received elements. The next step involves the *BPattern* table whose entries determine the address location to which we unpack the received element. This table as explained before keeps track of the locations where the nonlocal data are to be placed. For each entry in the *ReceiveBuffer* we unpack the element into the overlap regions. This procedure is followed until all the elements are unpacked into the overlap areas.

$count_q = 0$ for $0 \leq q \leq p-1$
do $i = 0, sizeof(RProcPattern_m) - 1$
  $q = RProcPattern_m[i]$
  $loc_B = BPattern_m[i]$
  do $j = 0, \frac{max_A}{s_1 k}$
    $Bloc[loc_B] = ReceiveBuffer_q[count_q]$
    $count_q = count_q + 1$
    $loc_B = loc_B + s_2 k + num\_nonloc$
  enddo
enddo

Step 6: This stage performs lexicographic execution of elements on processor m. Since all the received elements are already unpacked into local regions, this stage involves the *APattern* and *BPattern* tables whose entries determine the addresses of elements accessed of the LHS and RHS arrays involved in computation. The computation is then performed on these elements. This procedure is followed until all the elements are executed.

$baseadd_A = baseadd_B = 0$
do $j = 0, \frac{max_A}{s_1 k}$
  do $i = 0, sizeof(RProcPattern_m) - 1$
    $loc_A = APattern_m[j] + baseadd_A$
    $loc_B = BPattern_m[j] + baseadd_B$
    $Aloc[loc_A] = \mathcal{F}\ (Bloc[loc_B])$
  enddo
  $baseadd_A = baseadd_A + s_1 k$
  $baseadd_B = baseadd_B + s_2 k + num\_nonloc$
enddo

Figure 2.7 shows the three main stages of the algorithm, packing, unpacking and execution for processor 0. The *SendBuffer* tables shows the order in which elements are packed. This method is exactly similar to the one discussed in *course* padding. Similarly the second table *BPattern* holds entries of locations where the received elements are to be stored. The next set of tables indicate the process of unpacking. This stage is different from the one in *course* padding due to the different set of locations which hold the nonlocal elements. As mentioned earlier the unpacking is performed into the overlap region. The last stage shows the local addresses of both LHS and RHS on processor 0 which are involved in the execution stage. Note that the local addresses of these elements are different from the local addresses in the case of course padding.

2.5 MESSAGE OPTIMIZATIONS

In this section, we discuss the use of message optimizations in code generation. We discuss three optimizations namely offset communication, message coalescing and aggregation.

In case of multiple right hand side references to the same array, we can perform message coalescing by sending only one copy of an element. We can also reduce the overhead of startup cost in communication by combining several messages (called message aggregation) possibly due to different arrays referenced on the RHS to be communicated between a sender and a receiver pair. We also discuss the idea of offset communication optimization which is when both LHS and RHS have the same distribution and access strides, using the concept of overlap areas.

2.5.1 OFFSET COMMUNICATION WITH OPTIMIZED OVERLAP REGIONS

If the mapping and access strides of the LHS and RHS arrays are identical and the access offsets differ (while being less than or comparable to the block size), the communication is required only between neighboring processors. This type of communication is termed as *offset communication*. Offset communication is implemented using the idea of overlap areas and can be implemented by either course padding or column padding. Extra storage is allocated for only nonlocal elements and this leads to a change in the local access order (similar to that with course and column padding in the general case). As offset communication involves communication between only the neighboring virtual processors, the resulting code is very efficient.

Offset communication is highly effective if the offsets on the right hand side arrays are all less than k. If the offsets are greater than k, we have communication among virtual processors that are not neighbors. The amount of extra storage depends mainly on the values of the offsets in the RHS arrays and the stride of the LHS array. If the offsets are all less than k and s, there are no repeating elements in the nonlocal accesses. On the other hand if any of the offsets is less than k but greater than s, there might be some repeating elements in the nonlocal accesses due to different terms on the RHS, and message coalescing may be useful here. Hence in this case the storage space is allocated based on elements received after performing message coalescing.
Consider the following loop.

$$\begin{array}{l} \textbf{do } i = 0, \left\lfloor \frac{h_1 - l_1}{s} \right\rfloor \\ \qquad A(l_1 + is) = \mathcal{F}\left(B(l_2 + \theta_1 + is), B(l_2 + \theta_2 + is), \cdots, B(l_2 + \theta_r + is)\right) \\ \textbf{enddo} \end{array}$$

where $|\theta_i| \leq k$ and $|\theta_i| \leq s$ (for $i = 1, \cdots, r$) and arrays A and B have identical mapping with CYCLIC(k) distribution on p processors. With course padding, the number of extra storage needed for every s courses of array B is $\sum_{i=1}^{r} |\theta_i|$. In the case of column padding, local storage allocated depends on the signs of the offsets. If $\theta_i < 0$ the padding column is to the left of the storage for the course; if $\theta_i > 0$ the padding column is to the right. For each s courses of local storage for the array B, we need $\sum_{i=1}^{r} |\theta_i|$ units of extra storage for nonlocal references. Figures 2.8 and 2.9 illustrate course and column padding respectively for the following loop which benefits from offset communication.

$$\textbf{do } i = 0, \left\lfloor \frac{N}{5} \right\rfloor$$

Processor 1

8	9	10	11	$\boxed{12}$	$\boxed{13}$	$\boxed{14}$	15	
40	41	$\boxed{42}$	$\boxed{43}$	$\boxed{44}$	45	46	$\boxed{47}$	
72	73	74	75	76	$\boxed{77}$	$\boxed{78}$	$\boxed{79}$	
104	105	106	$\boxed{107}$	$\boxed{108}$	$\boxed{109}$	110	111	
136	137	138	139	140	141	$\boxed{142}$	$\boxed{143}$	
$\boxed{17}$	$\boxed{18}$	$\boxed{19}$	$\boxed{48}$	$\boxed{49}$	$\boxed{112}$	$\boxed{113}$	$\boxed{114}$	$\boxed{144}$

(a): Offset communication with padded courses: global address pattern.

Processor 1

0	1	2	3	$\boxed{4}$	$\boxed{5}$	$\boxed{6}$	7	
8	9	$\boxed{10}$	$\boxed{11}$	$\boxed{12}$	13	14	$\boxed{15}$	
16	17	18	19	20	$\boxed{21}$	$\boxed{22}$	$\boxed{23}$	
24	25	26	$\boxed{27}$	$\boxed{28}$	$\boxed{29}$	30	31	
32	33	34	35	36	37	$\boxed{38}$	$\boxed{39}$	
$\boxed{40}$	$\boxed{41}$	$\boxed{42}$	$\boxed{43}$	$\boxed{44}$	$\boxed{45}$	$\boxed{46}$	$\boxed{47}$	$\boxed{48}$

(b): Offset communication with padded courses: local address pattern.

Figure 2.8: Access patterns for RHS elements on processor 1 (using offset communication using padded courses) when $p = 4$ and $k = 8$.

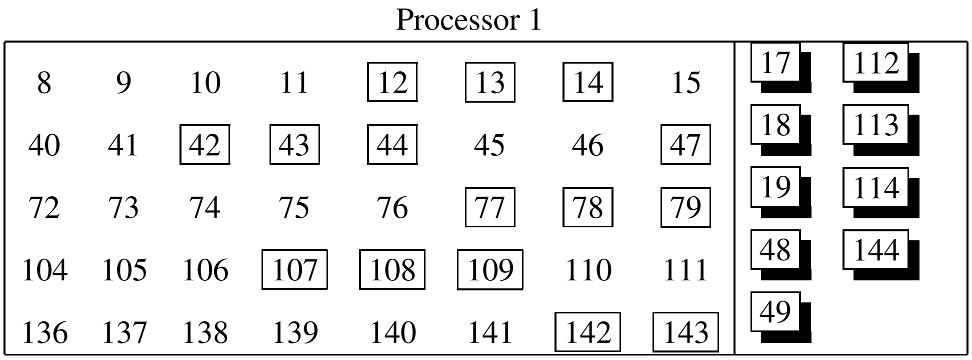
$$A(5i) = \mathcal{F}\,(B(2+5i), B(3+5i), B(4+5i))$$

enddo

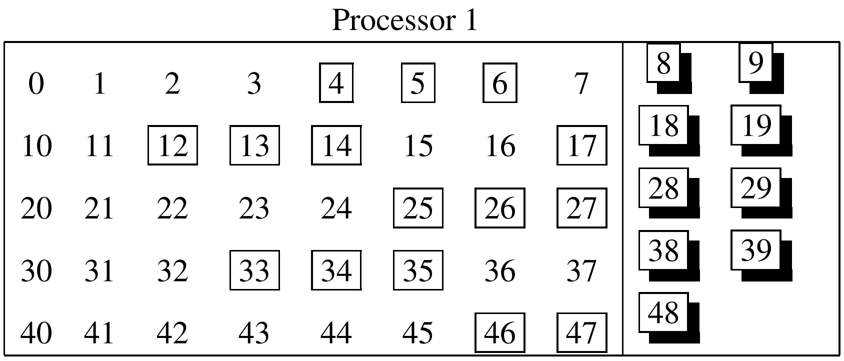
2.5.2 MESSAGE AGGREGATION AND COALESCING

If there are references to multiple RHS arrays in a loop, it is better to combine all the messages going from one processor to the same destination into a single message and then send it. This optimization is called *message aggregation* and helps in reducing communication overhead. Message aggregation is performed by aggregating the *SendBuffer*$[]_q$ (q is the receiving processor) for all the r RHS arrays. This results in a single message buffer being generated to be sent to a processor. In this case, the message header includes additional information as necessary.

If there are multiple references to the same element of an array, then we can again perform message coalescing and just send a single copy of the element; this optimization results in reduced communication volume. While performing message coalescing we have to keep track of the RHS array involved in sending or receiving the common element.



(a): Offset communication with padded columns: global address pattern.



(b): Offset communication with padded columns: local address pattern.

Figure 2.9: Access patterns for RHS elements on processor 1 (using offset communication using padded columns) when $p = 4$ and $k = 8$.

Special message handling has to be provided at the receiving end in order to perform correct unpacking and execution.

2.6 EXPERIMENTS

In order to verify the usefulness of overlap regions, we implemented all the schemes discussed above and tested them for varying values of s_1, s_2 and k. Our execution model used the DAXPY operation

$$\begin{aligned}&\mathbf{do}\ i = 0, \left\lfloor \frac{h_1 - l_1}{s_1} \right\rfloor\\ &\qquad A(l_1 + is_1) = A(l_1 + is_1) + \mathit{constant} * B(l_2 + is_2)\\ &\mathbf{enddo}\end{aligned}$$

In all our experiments we fixed the number of iterations per processor to be 4000. This allows us to measure the effect of the strides and block sizes for a fixed amount of computation. We used 32 processors in our experiments. All the experiments were conducted on a Sun ultrasparc. The reported times are in microseconds. We do not discuss the actual communication time for performing interprocessor communication as it varies widely across configurations and since all the three methods incur the same interprocessor communication cost.

Tables 2.1–2.6 lists the time it takes for packing, unpacking+local loop execution(pure execution), and total time for all the methods mentioned in this chapter. The model where we read directly from the buffers during execution is referred to as *buffer*. The methods using course padding and column padding schemes are referred to as *course* and *column* respectively. Since the *buffer* scheme does not perform unpacking, the entries for unpacking+execution are purely times for pure buffer execution. The total execution time in the table corresponds to the sum of packing, unpacking and pure execution for both *course* and *column* whereas it corresponds to the sum of packing and pure execution for *buffer*.

For each fixed value of s_1 parameters k and s_2 were varied. The low times for packing and unpacking is due to the fact that we reduced the number of lookups in the final SPMD code and also due to the choice of building a *BPattern* table which reduces expensive runtime conditionals and recalculation of tables. Table 2.1 lists the times when $s_1 = 1$. This case indicates that the overhead in execution is purely due to the effect of s_2. The LHS stride being unity, there is not much table lookup overhead during execution due to array A. Even for this case we observe that the overall total time for the *course* and *column* method outperforms the *buffer* method. Varying s_1 as shown in the remaining tables also conclude that, there is a benefit in unpacking messages into overlap regions instead of reading elements from the *buffer*.

We conclude that the *course* and *column* methods perform well when compared to reading data out of buffers directly. Based on the results we see a 30% improvement in the total time when compared to the *buffer* method. All these methods indicate that the overhead in unpacking of elements is not that expensive when compared to the benefit we get due to cache locality. Therefore, using overlap regions appears to be beneficial in executing data parallel programs on distributed-memory message passing machines. In addition, unpacking of data into overlap regions improves data locality.

2.7 Chapter summary

In this chapter, we have dealt with communication issues in code generation for data-parallel programs with block-cyclic distribution executing on distributed-memory message passing architectures. We presented a new approach for the management of communication sets and strategies for local storage of remote references. We derived algorithms for determining communication patterns. We then presented two schemes that extend the notion of a local array by providing storage for nonlocal elements (called *overlap regions*) interspersed throughout the storage for the local portion. The two schemes, namely *course padding* and *column padding* enhance locality of reference significantly at the cost of a small overhead due to unpacking of messages. The performance of these schemes was found to be up to 30% better than the traditional buffer-based approach. In addition, several message optimizations such as offset communication, message aggregation and coalescing were discussed. Extension of these schemes to two-level mapping is rather straightforward.

Table 2.1: Communication packing, unpacking and execution time results (in microseconds) of our algorithms on an ultrasparc for the case $p = 32$, $s_1 = 1$ and number of iterations = 4000

		Packing			Unpacking+Exec			Total		
k	s_2	buffer	column	course	buffer	column	course	buffer	column	course
4	3	665	347	328	1392	1155	1100	2057	1502	1467
	5	676	346	318	1418	1146	1094	2096	1495	1426
	7	673	346	337	1383	1118	1108	2056	1464	1453
	9	679	345	340	1360	1128	1107	2041	1473	1459
	11	697	344	343	1365	1116	1113	2067	1460	1470
8	3	608	291	291	1325	1070	1041	1935	1361	1333
	5	630	336	334	1317	1038	1015	1946	1374	1349
	7	628	342	341	1354	1010	1008	1984	1352	1350
	9	625	340	340	1362	1033	1000	1989	1376	1341
	11	632	330	330	1352	1002	990	1987	1332	1325
16	3	603	293	293	1302	1052	1119	1905	1346	1413
	5	612	321	321	1299	1107	1022	1911	1428	1343
	7	601	333	333	1301	1080	1076	1903	1413	1409
	9	635	333	333	1311	1013	1095	1948	1346	1430
	11	632	333	333	1295	1092	1037	1927	1425	1370
32	3	572	292	295	1290	956	972	1862	1249	1266
	5	594	342	343	1282	962	970	1876	1306	1313
	7	595	341	329	1266	951	949	1861	1292	1280
	9	607	341	342	1276	933	933	1884	1275	1275
	11	597	333	341	1293	921	928	1890	1254	1269
64	3	595	301	301	1270	1687	964	1865	1989	1266
	5	592	343	345	1271	955	965	1863	1299	1309
	7	605	331	332	1259	936	947	1864	1268	1280
	9	597	342	342	1264	951	958	1861	1294	1301
	11	597	344	344	1257	931	970	1855	1275	1314
128	3	570	311	313	1271	975	977	1840	1288	1291
	5	613	353	353	1254	1022	992	1867	1375	1348
	7	603	342	343	1263	958	960	1866	1301	1303
	9	614	356	357	1262	1001	967	1876	1357	1324
	11	605	361	361	1245	960	1003	1850	1321	1364
256	3	578	342	342	1300	1012	1027	1878	1354	1372
	5	609	375	380	1283	1034	1028	1892	1409	1408
	7	635	381	383	1259	1033	1186	1894	1414	1570
	9	629	384	386	1262	1017	1134	1891	1404	1520
	11	647	385	386	1271	1013	1011	1918	1400	1399

Table 2.2: Communication packing, unpacking and execution time results (in microseconds) of our algorithms on an ultrasparc for the case $p = 32$, $s_1 = 3$ and number of iterations = 4000

k	s_2	Packing			Unpacking+Exec			Total		
		buffer	column	course	buffer	column	course	buffer	column	course
4	5	626	340	336	1459	1242	1160	2090	1583	1505
	7	587	346	349	1463	1274	1165	2050	1620	1538
	9	609	392	366	1446	1294	1182	2056	1686	1564
	11	696	362	344	1532	1266	1158	2229	1629	1504
8	5	576	283	282	1453	1181	1128	2032	1464	1411
	7	625	340	340	1441	1173	1131	2067	1512	1471
	9	664	399	388	1428	1227	1136	2094	1625	1525
	11	621	335	335	1461	1165	1121	2083	1499	1457
16	5	623	349	348	1444	1158	1218	2066	1507	1566
	7	611	339	326	1446	1264	1145	2058	1603	1471
	9	668	402	403	1396	1228	1194	2065	1630	1598
	11	605	343	330	1447	1167	1219	2055	1511	1550
32	5	591	352	328	1437	1075	1064	2028	1427	1392
	7	604	327	327	1476	1066	1054	2082	1394	1381
	9	639	404	388	1390	1071	1054	2031	1476	1442
	11	609	348	348	1446	1064	1052	2055	1412	1400
64	5	586	333	382	1413	1075	1847	1999	1408	2229
	7	606	346	346	1419	1075	1069	2025	1422	1415
	9	643	405	406	1402	1064	1083	2045	1470	1489
	11	603	343	343	1409	1072	1073	2014	1416	1416
128	5	618	356	357	1430	1106	1100	2048	1465	1458
	7	600	350	352	1431	1111	1105	2031	1461	1456
	9	664	412	413	1407	1104	1093	2072	1517	1507
	11	612	349	349	1406	1107	1101	2018	1456	1450
256	5	598	382	384	1431	1126	1124	2029	1511	1508
	7	620	377	377	1414	1160	1132	2035	1539	1513
	9	721	417	418	1411	1124	1134	2132	1541	1552
	11	645	387	389	1405	1152	1143	2049	1542	1535

Table 2.3: Communication packing, unpacking and execution time results (in microseconds) of our algorithms on an ultrasparc for the case $p = 32$, $s_1 = 5$ and number of iterations = 4000

k	s_2	Packing			Unpacking+Exec			Total		
		buffer	column	course	buffer	column	course	buffer	column	course
4	3	609	308	320	1517	1238	1195	2126	1547	1533
	7	690	370	346	1572	1316	1200	2262	1688	1568
	9	685	366	346	1576	1309	1194	2262	1675	1543
	11	697	362	385	1552	1300	1181	2252	1664	1568
8	3	586	264	264	1450	1145	1120	2036	1409	1384
	7	628	351	325	1472	1211	1124	2101	1562	1451
	9	621	335	339	1445	1214	1108	2066	1549	1447
	11	636	361	356	1451	1202	1108	2087	1566	1464
16	3	581	261	291	1516	1149	1130	2097	1410	1420
	7	637	342	337	1500	1171	1121	2137	1515	1459
	9	606	349	340	1614	1173	1124	2220	1522	1463
	11	611	345	340	1689	1162	1117	2300	1508	1457
32	3	598	264	288	1501	1100	1105	2099	1365	1394
	7	604	326	328	1497	1128	1112	2101	1454	1439
	9	599	347	347	1496	1115	1102	2095	1462	1449
	11	610	352	352	1537	1118	1099	2148	1472	1451
64	3	591	295	295	1475	1153	1114	2066	1449	1409
	7	594	352	352	1466	1159	1154	2060	1511	1507
	9	582	335	335	1465	1149	1148	2048	1484	1483
	11	596	357	358	1475	1150	1143	2072	1508	1501
128	3	560	288	288	1477	1156	1153	2037	1445	1442
	7	607	351	351	1463	1173	1161	2070	1525	1515
	9	616	360	363	1456	1158	1176	2073	1520	1541
	11	609	377	377	1456	1548	1155	2065	1925	1532
256	3	602	341	343	1477	1335	1325	2079	1677	1672
	7	629	397	398	1466	2282	2269	2095	2679	2667
	9	614	381	383	1467	2782	2775	2081	3165	3159
	11	660	405	406	1459	2772	2765	2119	3177	3172

Table 2.4: Communication packing, unpacking and execution time results (in microseconds) of our algorithms on an ultrasparc for the case $p = 32$, $s_1 = 7$ and number of iterations = 4000

k	s_2	Packing			Unpacking+Exec			Total		
		buffer	column	course	buffer	column	course	buffer	column	course
4	3	589	346	319	1465	1205	1150	2059	1552	1468
	5	636	350	318	1462	1250	1163	2102	1600	1497
	9	591	340	355	1457	1241	1167	2052	1581	1522
	11	695	362	371	1443	1261	1148	2138	1623	1527
8	3	574	292	277	1494	1203	1147	2069	1495	1425
	5	586	299	285	1484	1244	1151	2070	1543	1437
	9	636	347	343	1492	1278	1133	2131	1625	1476
	11	636	335	348	1500	1302	1128	2136	1638	1476
16	3	553	270	268	1463	1229	1189	2016	1499	1458
	5	604	339	339	1477	1243	1180	2081	1583	1519
	9	603	330	342	1459	1263	1178	2063	1592	1520
	11	612	341	341	1475	1249	1183	2088	1591	1525
32	3	549	293	288	1454	1141	1130	2004	1435	1418
	5	582	337	325	1449	1155	1126	2031	1492	1451
	9	602	340	340	1479	1166	1130	2081	1506	1470
	11	611	338	338	1456	1169	1141	2067	1507	1479
64	3	550	289	289	1549	1073	1072	2099	1362	1362
	5	584	340	340	1536	1082	1080	2122	1423	1420
	9	602	343	344	1539	1086	1074	2141	1429	1418
	11	609	339	339	1540	1083	1073	2150	1423	1414
128	3	576	302	302	1455	1129	1131	2031	1431	1437
	5	592	356	357	1462	1148	1142	2054	1505	1499
	9	612	362	363	1457	1146	1141	2070	1508	1507
	11	618	354	355	1453	1142	1139	2072	1498	1497
256	3	569	331	333	1473	1142	1146	2042	1476	1479
	5	616	373	374	1464	1149	1144	2081	1523	1519
	9	637	377	381	1457	1146	1134	2094	1524	1516
	11	637	381	382	1478	1147	1350	2115	1529	1732

Table 2.5: Communication packing, unpacking and execution time results (in microseconds) of our algorithms on an ultrasparc for the case $p = 32$, $s_1 = 9$ and number of iterations = 4000

k	s_2	Packing			Unpacking+Exec			Total		
		buffer	column	course	buffer	column	course	buffer	column	course
4	3	641	359	338	1466	1217	1189	2107	1576	1537
	5	665	330	359	1453	1250	1160	2122	1579	1519
	7	685	356	356	1467	1260	1166	2154	1616	1522
	11	693	355	369	1490	1266	1178	2193	1624	1550
8	3	586	297	291	1516	1219	1191	2104	1515	1482
	5	606	315	335	1486	1212	1146	2097	1528	1481
	7	627	344	357	1497	1275	1142	2126	1619	1500
	11	622	321	329	1484	1271	1139	2109	1592	1469
16	3	550	295	294	1498	1155	1146	2049	1451	1441
	5	592	334	320	1493	1141	1101	2085	1476	1425
	7	596	324	334	1489	1170	1090	2087	1493	1427
	11	614	344	345	1514	1187	1098	2129	1534	1445
32	3	540	292	259	1500	1178	1158	2041	1471	1418
	5	584	323	323	1511	1165	1140	2096	1488	1463
	7	593	337	337	1527	1179	1145	2120	1515	1486
	11	595	337	337	1533	1190	1142	2131	1528	1481
64	3	563	298	298	1473	1149	1139	2036	1448	1437
	5	585	337	337	1471	1126	1122	2056	1465	1462
	7	593	341	342	1471	1135	1119	2064	1479	1463
	11	594	344	344	1481	1145	1131	2075	1490	1476
128	3	544	343	344	1466	1157	1148	2025	1501	1493
	5	604	337	338	1463	1145	1135	2066	1485	1473
	7	617	346	352	1464	1145	1144	2081	1492	1500
	11	623	349	349	1464	1144	1137	2089	1498	1488
256	3	598	307	310	1500	1366	1363	2098	1687	1676
	5	586	373	378	1491	1342	1338	2076	1722	1717
	7	615	378	376	1475	1189	1164	2090	1570	1543
	11	626	391	389	1478	1350	1363	2104	1743	1755

Table 2.6: Communication packing, unpacking and execution time results (in microseconds) of our algorithms on an ultrasparc for the case $p = 32$, $s_1 = 11$ and number of iterations = 4000

k	s_2	Packing			Unpacking+Exec			Total		
		buffer	column	course	buffer	column	course	buffer	column	course
4	3	579	336	318	1552	1326	1231	2132	1663	1562
	5	629	339	346	1570	1347	1287	2203	1687	1633
	7	590	343	344	1544	1352	1270	2138	1695	1618
	9	603	368	342	1555	1337	1242	2158	1705	1584
8	3	571	284	267	1495	1328	1283	2066	1614	1554
	5	579	281	280	1504	1568	1372	2088	1849	1651
	7	625	347	334	1491	1398	1381	2117	1745	1715
	9	643	361	358	1478	1517	1450	2124	1880	1811
16	3	544	294	267	1504	1120	1085	2048	1414	1353
	5	591	330	328	1489	1131	1079	2080	1461	1410
	7	589	322	328	1495	1171	1086	2087	1497	1414
	9	606	345	345	1485	1178	1077	2093	1524	1424
32	3	542	270	270	1468	1159	1139	2022	1429	1412
	5	574	317	317	1459	1156	1134	2033	1474	1454
	7	578	312	312	1459	1172	1114	2038	1485	1427
	9	595	344	344	1468	1185	1120	2065	1529	1468
64	3	542	276	276	1475	1099	1097	2017	1376	1373
	5	574	330	335	1464	1124	1101	2038	1458	1437
	7	587	330	337	1479	1128	1108	2068	1460	1448
	9	594	341	341	1471	1124	1101	2064	1465	1449
128	3	550	287	289	1468	1140	1135	2018	1428	1430
	5	580	335	338	1460	1145	1147	2046	1482	1487
	7	590	351	350	1472	1150	1131	2062	1501	1484
	9	602	353	356	1473	1146	1135	2079	1503	1501
256	3	562	315	319	1484	1161	1153	2046	1478	1472
	5	594	366	367	1487	1150	1150	2081	1517	1525
	7	611	362	367	1481	1152	1152	2091	1517	1527
	9	630	377	389	1485	1169	1259	2115	1550	1648

CHAPTER 3

ADDRESS SEQUENCE GENERATION FOR TWO-LEVEL MAPPING

Address sequence generation for non-unit alignment strides can be difficult. This chapter presents solution to the problem of address generation for such a case when the data objects are mapped to processor memories using CYCLIC(k) distribution. We present several methods of generating the address sequence. Our approach involves construction of pattern tables which do not incur runtime overheads as compared to other existing solutions for this problem. We view this problem as an integer lattice and use two applications of the method developed by Thirumalai and Ramanujam [27] to generate the pattern of accesses.

3.1 PROBLEM STATEMENT

Consider the following HPF code

```
        REAL A(N)
!HPF$   TEMPLATE T(a*N + b)
!HPF$   PROCESSORS PROCS(p)
!HPF$   ALIGN A(j) WITH T(a*j + b)
!HPF$   DISTRIBUTE T(CYCLIC(k)) ONTO PROCS
        do i = 0, ⌊(h-l)/s⌋
              A(l + is) = ···
        enddo
```

The compiler which generates the node code for the above HPF program has to generate the set of local elements *accessed* by array A on processor m.

As discussed before in Chapter 1, when $a > 1$, the mapping is called two-level mapping. This mapping results in having many template cells which do not have any array elements mapped to them. These empty template cells are termed as *holes*. We need not allocate memory for these *holes* in the local address space during mapping. The challenge now is to generate the set of accessed elements in this local address space.

Figure 3.1(a) shows the distribution of the template cells onto a processor arrangement. For this example, the alignment stride a and access stride s are both equal to 3. The number of processors p is 4 and we assume a CYCLIC(4) distribution. This example is from Chatterjee *et al.* [4]. The set of global indices of array elements that are aligned to some template cell on this processor is called the set of *allocated* elements. The set of global indices of accessed array elements that are aligned to some template cell on this processor is called the set of *accessed* elements. These *accessed* elements are however

Processor 0				Processor 1				Processor 2				Processor 3			
0			1			2			3			4			5
		6			7			8			9			10	
	11			12			13			14			15		
16			17			18			19			20			21
		22			23			24			25			26	
	27			28			29			30			31		
32			33			34			35			36			37
		38			39			40			41			42	
	43			44			45			46			47		

(a): Global layout of template cells on $p = 4$ processors.

Processor 0	0			1			6			11		
Processor 1			2			7			12			13
Processor 2		3			8			9			14	
Processor 3	4			5			10			15		

(b): Local memory layout for template cells

Processor 0	0	1	6	11	16	17	22	27	32	33	38	43
Processor 1	2	7	12	13	18	23	28	29	34	39	44	45
Processor 2	3	8	9	14	19	24	25	30	35	40	41	46
Processor 3	4	5	10	15	20	21	26	31	36	37	42	47

(c): Local memory layout for array cells.

Figure 3.1: Two-level mapping of array A when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

a subset of *allocated* elements. For the given example 0, 1, 6, 11, 16, $\cdots$ is the set of *allocated* elements and 0, 6, 27, 33 ,48, $\cdots$ is the set of *accessed* elements for processor 0. Figure 3.1(b) shows the local address space of template cells on all the processors. The problem of generating *accessed* elements for this template space is similar to a one-level mapping problem where the stride s is replaced with $a * s$. However using this method we incur huge memory wastage and suffer from data locality resulting in higher execution times.

However if one eliminates holes in this layout, we can have a significant savings in memory usage. This can be achieved by viewing the local address space as shown in Figure 3.1(c). This local address space does not have any memory wastage. However additional work at address generation time has to be done to switch from the template space to the local space. Due to the absence of these *holes* we can expect improved data locality and thus leading to faster execution times. The address generation problem now is to generate the set of elements *accessed* in this local address space. The algorithms discussed in this chapter aims at solving this problem using efficient runtime techniques.

3.2 RELATED WORK

A few methods have been proposed to address the code generation problem for two-level mapping. The solution by Chatterjee *et al.* [4] involves two applications of the one-level algorithm, where the input strides are a for the first and $a * s$ for the second. They build two finite state machines which will generate the access sequence for both *allocated* and *accessed* elements. The next step involves using the finite state machines for the template space to rebuild a new finite state machine for the local address space. This fix-up involves computing expensive division and mod operations to determine the addresses of *accessed* elements in the local memory space. One other drawback of their technique when compared to our technique is the fact that their one-level pattern tables contain local memory gaps and not actual addresses. However their execution preserves lexicographic ordering and they do not incur any memory wastage.

Ancourt *et al.* [1] presented a linear algebraic framework to solve the problem of address generation for distributed memory machines. Their framework to solve two-level mapping involves a change of basis which leads to compression of holes but still incur some memory wastage. The node-code generated by this framework has complicated loops and incurs high execution overhead. However their execution order does not preserve lexicographic ordering.

Kaushik [11] uses processor virtualization to handle block-cyclic distributions. His method involves generation of addresses for both hole-compression and for the one without hole-compression. He first determines the amount of memory to be allocated using regular section characterization for *block* and *cyclic* distributions. He then extends this technique to virtual processor approach for handling *block-cyclic* distributions. However he cannot ensure lexicographic execution in the case of virtual cyclic approach. He also cannot guarantee a 100% compression and hence incur small amount of memory wastage.

3.3 ALGORITHMS FOR TWO-LEVEL MAPPING

The algorithms proposed in this section solve the problem of generating addresses for a compressed space for two-level mapping. These algorithms exploit the repetitive pattern of accesses by constructing pattern tables for the local address space. These pattern tables are then used to generate the complete set of accesses for the array layout just like in the case of one-level mapping [27].

INDEX	0	1	2	3	4	5	6	7	8	9	10	11
ALLOCATED	0	3	6	9	12	15	18	21	24	27	30	33

(a)

ACCESSED	0	6	21	27

(b)

PATTERN	0	2	7	9

(c)

Figure 3.2: Local addresses of *accessed* and *allocated* elements along with *two-level* access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

The main idea behind these algorithms is to construct tables which store indices to switch from the template space to the local space. Since we do not allocate memory for *holes*, we have no memory wastage. We also do not incur high costs for generating access function to switch from the template space to local address space, this leads to faster execution times. This coupled with the fact that no memory is wasted proves that these methods are superior to any other existing methods which executes elements lexicographically.

The algorithms discussed in this chapter can be broadly classified into two different methods. These algorithms differ mainly in the manner in which the tables are constructed in order to switch from the template space to the local address space. The first algorithm constructs a table of offsets whereas the algorithms in the second method uses different

Y1	0	0	1	1
Y2	0	3	2	1

(a)

X1	0	1	5	6
X2	0	2	1	3

(b)

ITABLE	0	3	2	1

(c)

PATTERN	0	2	7	9

(d)

Figure 3.3: Two dimensional coordinates of *allocated* and *accessed* elements along with *Itable* and *two-level* access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

search techniques to locate *accessed* elements in the set of *allocated* elements in the compressed space.

All algorithms first view the address space as an integer lattice and use basis vectors to generate the access sequence of both *allocated* and *accessed* elements. The basis vectors are generated using the one-level address generation algorithm [27]. Two application of the one-level algorithm [27] with input strides being a and $a * s$ in each case, generates the set of accesses for both *allocated* and *accessed* elements. Figure 3.2(a) shows the first set of repetition pattern of local addresses of the set of accesses for *allocated* elements. The numbers in boxes are the set of elements *accessed*, and the pattern of repetition of these elements is shown in Figure 3.2(b). In a compressed space we need to locate the position of these *accessed* elements, in the a set of *allocated* elements. So we record the positions of these entries in a separate table as shown in Figure 3.2(c). The main objective of these algorithms is to generate this switching table which helps in switching from the template space to the local compressed space. The construction of these switching tables is discussed in the following sections.

3.3.1 ALGORITHM WHICH CONSTRUCTS A TABLE OF OFFSETS

The main idea behind this algorithm is to construct a table of offsets, which is used to help switch from the non-compressed space to the compressed space. The algorithm exploits the repetition of accesses of both *allocated* and *accessed* elements. This algorithm first generates a two dimensional view of the set of accesses of both *allocated* and *accessed* elements for the non-compressed space. This is done by the application of the one-level algorithm with input strides being a and $a * s$ respectively. Recording the two-dimensional

Input: Layout parameters (p, k), loop limits (ℓ, h), access stride s, alignment stride a for array A, processor m

Output: Two_level

Method:

$d_1 \longleftarrow \gcd(a, pk)$
$d_2 \longleftarrow \gcd(a * s, pk)$
$(length_a, Y_1, Y_2) \longleftarrow one\_level(p, k, a, m, d_1)$
$(length_{as}, X_1, X_2) \longleftarrow one\_level(p, k, a * s, m, d_2)$
for $i = 0, length_a - 1$ **do**
    $Itable[Y_2[i]] \longleftarrow i$
enddo
$a_{cou} \longleftarrow \frac{a}{d_1}$
for $i = 0, length_{as} - 1$ **do**
    $Two\_Level[i] \longleftarrow \frac{X_1[i]}{a_{cou}} * length_a + Itable[X_2[i]]$
enddo
return Two_level

Figure 3.4: Algorithm which constructs the *Itable* for determining the two-level access pattern table

view of these sets does not incur any extra overhead such as expensive division and modular operations due to the way we generate the set of accesses using the one-level algorithm.

Figures 3.3(a) and (b) lists the two-dimensional coordinates of both *accessed* and *allocated* elements for the first set of pattern repetition for the example discussed previously. The second coordinates of both these sets indicate the offsets of the elements from the beginning of each course. A quick glance clearly indicates that the *accessed* elements are a subset of *allocated* elements. Using this information the algorithm first builds a table of offsets called the *Itable* for the first repetition pattern of allocated elements. The allocated access pattern repeats itself after every $\frac{a}{\gcd(a,pk)}$ courses. This table records the order in which the offsets of *allocated* elements are accessed in lexicographic order.

The next stage involves using this table to determine the location of the *accessed* element in the compressed space. The problem now is to find two things. Firstly we need to determine the repetition block in which the *accessed* element is located. Secondly we need to find its position among a set of *allocated* elements in that particular repetition block. Finding the repetition block in which the element is located is straight forward, as we know the number of courses after which the set of *allocated* elements repeat and the length of this set. In order to find the position of the element in a list of *allocated* elements in a particular repetition block we need to index into the *Itable* which gives the position of the *accessed* element based on its offset from the beginning of the course. Hence by finding the repetition block in which the element exists and the position of the element in that block we can determine the local address of the element.

A detailed listing of the algorithm is as shown in the Figure 3.4. Lines 1–4 generate the pattern tables for the case when stride is a and $a * s$. These tables record the two dimensional indices of elements accessed. Y_1 and Y_2 hold the two dimensional coordinates of the *allocated* elements while X_1 and X_2 hold the two dimensional coordinates for the *accessed* elements. Lines 5–7 construct the *Itable* which records the positions of offsets of *allocated* elements accessed in lexicographic order. The length of this table is always of size k. Lines 9–11 involves generation of the two-level pattern table. For each element in the *accessed* element set, a corresponding entry in the *Itable* will help determine the location of this element in the *allocated* set.

Let us consider the example in Figure 3.1. We see that elements 0, 1, 6, 11 have offsets 0, 3, 2, 1 respectively from the beginning of the course. The two-dimensional coordinates for both *allocated* and *accessed* elements are as shown in Figures 3.3(a) and (b). Based on the entries in the Y_2 table, the *Itable* is constructed and is as shown in Figure 3.3(c). In this case the second coordinates of the *allocated* elements are same as that of the *Itable*, but in general the entries in the *Itable* depends on the value of $\gcd(a, pk)$. The *Itable* is always of size k, as there could be a maximum of k elements for each pattern of repetition. In order to construct the *Two-Level* pattern table, let us consider the third entry $(5, 1)$ from the table of *accessed elements* as shown in Figure 3.3(b). This means that the accessed element lies in course number 5 and hence falls in the second repetition block of *allocated* elements. The value 1 in the second coordinate corresponds to the offset of the *accessed* element from the beginning of the course. This serves as an index into the *Itable*. Hence the value at position 1 of the *Itable* will yield 3 as shown in Figure 3.3(c). This value gives the position of the *accessed* element in that particular repetition block. Since we know the number of elements present in a single block (which corresponds to 4 in this example), we can simply evaluate 4*1 + 3 = 7, which gives us the position of the third element accessed in the compressed local space. Figure 3.3(d) shows the positions of *accessed* elements among a set of *allocated* elements for the first set of pattern repetition. The next sections deal with generating the set of *accessed* elements using search techniques.

3.3.2 Algorithms which uses search techniques

The main idea behind determining the local addresses of *accessed elements*, is to find the location of *accessed* elements in a list of *allocated* elements (expanded to accommodate the longest element in the accessed set), since *accessed* elements are a subset of *allocated* elements. This can be achieved by using a naive approach by searching for the index of the *accessed* element in the list of *allocated* elements. In this section we propose new search methods which exploit the property that the list of elements are in sorted order.

The first step in performing these methods is to run the one-level algorithm to obtain the local addresses of the set of accesses for the first pattern of repetition for both *allocated* and *accessed* elements. These entries are in lexicographic order and are assumed to be stored in $pattern_a$ and $pattern_{as}$ tables respectively. These tables are as shown in Figures 3.5(a) and (b) for the example in Figure 3.1. The entries in these table correspond to the local

(a)

ALLOCATED	**0**	**3**	**6**	**9**

(b)

ACCESSED	**0**	**6**	**21**	**27**

(c)

INDEX	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
ALLOCATED	**0**	**3**	**6**	**9**	**12**	**15**	**18**	**21**	**24**	**27**	**30**

(d)

PATTERN	**0**	**2**	**7**	**9**

Figure 3.5: Local addresses of *allocated* and *accessed* elements, the replicated *allocated* table, along with two-level access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

addresses in a non-compressed template space. The table size of the former depends on $\frac{k}{\gcd(a,pk)}$, whereas that of the latter table depends on $\frac{k}{\gcd(as,pk)}$.

Here we see that not all elements in the *accessed* set are present in the *allocated* set for the first pattern of repetition. This is due to the fact that the *accessed* elements lie in different repetition blocks of the local address space. Hence we need to expand the set of *allocated* elements in order to represent all the elements in the *accessed* table. The total number of elements in the first repetition block of *allocated* table in the uncompressed space is $\frac{ak}{\gcd(a,pk)}$ whereas the total number of elements for the *accessed* table is $\frac{ask}{\gcd(as,pk)}$. Hence the factor needed to expand the *allocated* elements table is $\frac{\frac{ask}{\gcd(as,pk)}}{\frac{ak}{\gcd(a,pk)}} = s\frac{\gcd(a,pk)}{\gcd(as,pk)}$.

Technique to perform this expansion is straight forward. It involves replicating the first set of *allocated* elements as many times depending on the factor of replication. This is accomplished by copying elements one at a time from the first pattern of *allocated* elements to the extended memory space with a suitable increment. One another technique can be implemented where replication is performed using a demand driven approach.

A search now has to be performed to locate the position of an *accessed* element in this new replicated table. Figure 3.5(c) shows the replicated table after expansion for the example discussed previously. The factor for replication in this case was found to be 3. Since the length of the table which holds the addresses of the *accessed* elements is never

Input: Layout parameters (p, k), loop limits (ℓ, h), access stride s, alignment stride a for array A, processor m
Output: Two_level
Method:

1 $d_1 \longleftarrow$ gcd (a, pk)
2 $d_2 \longleftarrow$ gcd (as, pk)
3 $(length_a, \textit{pattern}_a) \longleftarrow one\_level(p, k, a, m, d_1)$
4 $(length_{as}, \textit{pattern}_{as}) \longleftarrow one\_level(p, k, a * s, m, d_2)$
5 Factor of replication $f \longleftarrow \frac{\frac{ask}{d_2}}{\frac{ak}{d_1}}$
6 Replicate $\textit{pattern}_a$ by factor f
7 $i \longleftarrow 0; j \longleftarrow 0$
8 **while** $(j < length_{as})$ **do**
9   **if** $(\textit{pattern}_{as}[j] = \textit{pattern}_a[i])$ **then**
10   *Two_level*$[j] \longleftarrow i$
11       $j \longleftarrow j + 1$
12   **endif**
13    $i \longleftarrow i + 1$
14 **enddo**
15 **return** *Two_level*

Figure 3.6: Algorithm which constructs a two-level access pattern table using regular search method

greater than the length of the table which holds the *allocated* elements, the algorithm needs to find the locations of common elements from two tables of different size. Several search algorithms can be implemented for finding the locations of *accessed* elements. We discuss several such algorithms in the following sections. These algorithms differ mainly in their complexities and the speed of execution. We also propose a search technique to find the set of *accessed* elements by not having to perform replication.

3.3.3 REGULAR SEARCH

The algorithm for regular search builds the two-level pattern table needed to switch from the template space to the local address space. Figure 3.6 lists the complete algorithm. Lines 1–4 discusses the build up of the *accessed* and *allocated* table. The next step is to find the factor for replication and is as shown in Line 5. This factor is used to replicate the *allocated* table. Once replication is performed, we now need to perform a simple search in order to locate the positions of each *accessed* element in the *allocated* table. Lines 8–14 shows the search algorithm. For each entry in the *accessed* table, it determines the location of this element in the replicated *allocated* set. As and when the location is determined, the position is recorded into a *Two-Level* pattern table. The entries in this table reflects the local address of the *accessed* element in the compressed space.

Figure 3.5 can be used to explain the functioning of this algorithm. Let us consider the element 21 from the *accessed* set as shown in Figure 3.5(b). The search involves finding the position of this element in the replicated set as shown in Figure 3.5(c). This element can be found at location 7. This entry is then stored in the *Two-level* pattern table. The complete pattern table for the example is as shown in Figure 3.5(d). Since the search is performed on a sorted table of length $f * len_a$ and no element of this table is accessed more than once, the complexity of the algorithm is $O(f * len_a)$.

3.3.4 BINARY SEARCH

If the factor of replication becomes very large, then the previous algorithm suffers from the time taken to perform the search. In this case it is better to resort to faster techniques for searching. Hence this algorithm uses the binary search method in order to search for elements. Figure 3.7 lists the complete algorithm. Lines 1–4 in the algorithm construct the pattern table by applying the one-level algorithm. The next step involves determining the factor for replication. Once the necessary replication has been performed the algorithm then enters the binary search phase. Lines 9–24 shows the procedure involved in the binary search. For each entry in the *accessed* set, a binary search is performed to determine the corresponding location of this element in the replicated *allocated* set. Since the table is sorted, the search for the next entry begins at the location where the last entry was found. As and when an index of each element is found, the algorithm records this index into the *Two-level* pattern table. This algorithm has a complexity of $O(len_{as} * \log(f * len_a))$ which includes the complexity involved in the binary search.

3.3.5 MODIFIED-BINARY SEARCH

The search technique discussed in the previous section can be improved by reducing the amount of search, each time we encounter a common element. Exploiting the fact that the elements in the replicated table are in sorted order and are stored in different repetition blocks of length $length_a$ (length of the $pattern_a$ table), we first compute the repetition block in which the element we are searching exists. Secondly knowing the repetition block, we know that the element will surely lie within the next len_a elements. We then perform a binary search within this block in order to find the location of the element. This leads to faster search techniques and reduced complexity. Since the search is limited only to the length of the $pattern_a$ table, the complexity of this algorithm is $O(len_{as} * \log len_a)$.

The complete algorithm is shown in Figure 3.8. Line 11 in the algorithm shows the process of obtaining the repetition block in which the element exists. If val is the value we are searching, the repetition block in which this element exists is given by $\frac{\frac{val}{k}}{acou}$ where $acou$ is $\frac{a}{\gcd(a,pk)}$. The remaining parts of the algorithm is exactly similar to the one discussed in the previous section.

Input: Layout parameters (p, k), loop limits (ℓ, h), access stride s, alignment stride a for array A, processor m
Output: Two_level
Method:

$d_1 \leftarrow \gcd(a, pk)$
$d_2 \leftarrow \gcd(as, pk)$
$(length_a, \textit{pattern}_a) \leftarrow one\_level(p, k, a, m, d_1)$
$(length_{as}, \textit{pattern}_{as}) \leftarrow one\_level(p, k, a * s, m, d_2)$
Factor of replication $f \leftarrow \frac{s}{d_2} * d_1$
Replicate $\textit{pattern}_a$ by a factor f
$low \leftarrow 0$
$high \leftarrow f * len_a$
for $j = 0, length_{as} - 1$ **do**
    while $(low \leq high)$ **do**
        $mid \leftarrow \frac{high+low}{2}$
        $midval \leftarrow \textit{pattern}_a[mid]$
        if $(val < midval)$ **then**
            $high \leftarrow mid - 1$
        else if $(val > midval)$ **then**
            $low \leftarrow mid + 1$
        else
            $\textit{Two\_level}[j] \leftarrow mid$
        endif
    enddo
    $low \leftarrow mid$
    $high \leftarrow f * len_a$
enddo
return Two_level

Figure 3.7: Algorithm which constructs a two-level access pattern table using the binary search method

Input: Layout parameters (p, k), loop limits (ℓ, h), access stride s, alignment stride a for array A, processor m
Output: Two_level
Method:

$d_1 \leftarrow \gcd(a, pk)$
$d_2 \leftarrow \gcd(as, pk)$
$(length_a, \textit{pattern}_a) \leftarrow one\_level(p, k, a, m, d_1)$
$(length_{as}, \textit{pattern}_{as}) \leftarrow one\_level(p, k, a * s, m, d_2)$
Factor of replication $f \leftarrow \frac{s}{d_2} * d_1$
Replicate $\textit{pattern}_a$ by a factor f
$acou \leftarrow \frac{a}{d_1}$
$low \leftarrow 0$
for $j = 0, length_{as} - 1$ **do**
  $val \leftarrow \textit{pattern}_{as}[j]$
  $low \leftarrow \frac{\frac{val}{k}}{acou} \times len_a$
  $high \leftarrow low + len_a - 1$
  while $(low \leq high)$ **do**
    $mid \leftarrow \frac{high+low}{2}$
    $midval \leftarrow \textit{pattern}_a[mid]$
    if $(val < midval)$ **then**
      $high \leftarrow mid - 1$
    else if $(val > midval)$ **then**
      $low \leftarrow mid + 1$
    else
      $\textit{Two\_level}[j] \leftarrow mid$
    endif
  enddo
enddo
return Two_level

Figure 3.8: Algorithm to construct a pattern table using the modified binary search method

3.3.6 SEARCH ALGORITHM WITH NO REPLICATION

As discussed before, all the above mentioned search algorithms require the set of *allocated* elements to be replicated in order to determine the position of the *accessed* element. Replication in general depends on the individual values of $\gcd(a, pk)$ and $\gcd(a * s, pk)$. If the factor of replication is large, than the time for replication can be more than the time for search rendering the search algorithms inefficient.

Given an element from the *accessed* set we can determine the repetition block in which it exists. We can also find the offset (memory gap) of this element from the beginning of this repetition block. The offset we obtain will definitely match an entry in the pattern table for the first block of repetition of *allocated* elements. Hence knowing the address of an *accessed* element we need to just search for the location of the offset of this element in the pattern table of *allocated* elements. This completely avoids replication of the *allocated* set and hence saves a lot on runtime overheads. The search within this pattern table can be accomplished using a simple or a binary search.

Figure 3.9 lists the complete algorithm. Line 9 shows the factor to determine the repetition block in which the element exists. Line 10 provides the offset of the *accessed* element within the repetition block. Line 11 computes the memory gap between the *accessed* element and the corresponding entry in the first pattern of repetition of the *allocated* set. Consider element 27 in Figure 3.5(b). If we need to find its location in the local template space, we first need to map this element into the first block of repetition of the *allocated* set as shown in Figure 3.5(a). This is done by first finding the repetition block in which this element exists. In this case the number of the repetition block is equal to 2. The offset of this element within this block is 3 which matches the second entry in the *allocated* set. We now need to search for the position of this element in the *allocated* table. we use binary search within the first block of repetition to search for element 2. Once the location of 3 which is 1 is recorded, we now need to determine the actual location of the element. Since we know the repetition block (2) and the number of elements in each block we obtain the actual location by performing $4 * 2 + 1 = 9$ which gives the local address of the *accessed element*.

3.4 EXPERIMENTS AND RESULTS

The major issue in case of two-level mapping is to reduce the amount of storage. The advantage in saving memory leads to increase in data locality which in effect leads to faster execution of codes. But if we compress the local space in order to allocate memory for only the non-empty template cells, we incur an additional overhead in generating the access functions which takes us from the uncompressed space to the compressed space. We have been successful in completely eliminating the wastage due to memory. Our choices of approach to solve the problem of generating the access functions which takes us from the uncompressed space to the compressed space does not incur huge overheads and result in less runtime overheads. In order to test the above mentioned issues, we implemented all of our techniques discussed in this chapter and compared this with the

Input: Layout parameters (p, k), loop limits (ℓ, h), access stride s, alignment stride a for array A, processor m
Output: Two_level
Method:

```
d_1 ← gcd(a, pk)
d_2 ← gcd(as, pk)
(length_a, pattern_a) ← one_level(p, k, a, m, d_1)
(length_as, pattern_as) ← one_level(p, k, a * s, m, d_2)
acou ← a / d_1
incr ← ak / d_1
for j = 0, length_as − 1 do
   val ← pattern_as[j]
   factor ← (val / k) / acou
   val ← val − factor * incr
   offset ← factor * length_a
   low ← 0; high ← len_a − 1
   while (low ≤ high) do
      mid ← (high + low) / 2
      midval ← pattern_a[mid]
      if (val < midval) then
         high ← mid − 1
      else if (val > midval) then
         low ← mid + 1
      else
         Two_level[j] ← mid + offset
      endif
   enddo
enddo
return Two_level
```

Figure 3.9: Algorithm which constructs a two-level access pattern table using binary search without performing replication

work of Chatterjee *et al.* [4]. Their method also does not incur any memory wastage, but the difference comes from the method used to construct the switching tables which takes us from the template space to the local address space.

In order to contrast all the above mentioned methods, we ran our experiments on a varying number of problem parameters. We fixed the number of processors to $p = 32$ in all our experiments. All of our numbers were run on a Sun ultrasparc. For each value of alignment stride a, we varied both the block-size k and the access stride s. The method which builds a table of offset is termed as ITABLE. The best of the search algorithms which performs replication was chosen for the results. It is termed as SEARCH in the tables. The search algorithm which does not perform replication is termed as NOREP in the results. The method due to Chatterjee *et al.* [4] is termed as RIACS. Tables 3.1–3.15 show the time it takes to build the two-level pattern tables.

The results indicate that the times taken by all the above mentioned methods depend on the value of k, s and a. If the access and alignment strides are small, the SEARCH technique does better. This is because the time taken for replication and the overhead in performing a search is very minimal. But as s and k increases we notice that the SEARCH starts performing worse. This is because as s and k start increasing the time for replication in the SEARCH dominates over search and renders this method inefficient.

The construction of the ITABLE forms the major part of time taken for two-level pattern build up. This construction is purely a function of a and k and not of s. Hence as s increases the times for ITABLE does not vary widely.

The method by RIACS suffers with large block sizes due to the expensive runtime overheads. The NOREP method performs better than the SEARCH method as s starts increasing. This is due to the fact that we do not pay the overhead due to replication. But for large k we see that the times for search increases rendering NOREP inefficient.

3.5 CHAPTER SUMMARY

Having the advantage of saving on memory wastage and faster construction of table we have shown in this chapter that the method which constructs a table of offsets performs very efficiently for large values of block size and access strides. We have also proposed new search techniques which performs very well when the block sizes and access strides are comparatively small. We have contrasted our techniques with other existing methods to solve the address generation problem for two-level mapping. Our results indicate that the proposed methods perform very efficiently for varying values of alignment and access stride along with varying block sizes. We also get the added benefit of having good data locality while executing code due to savings in memory.

Table 3.1: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 2$, $s = 3$ and for $s = 5$

k	S=3				S=5			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	1	1	2	13	1	1	2	13
4	2	1	3	18	2	1	3	18
8	3	2	6	26	3	2	6	26
16	6	2	12	43	6	4	12	43
32	11	4	24	81	11	6	25	79
64	21	7	51	153	21	12	51	150
128	42	14	107	297	42	23	107	299
256	85	28	222	589	83	46	219	586
512	170	54	453	1171	169	90	463	1165

Table 3.2: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 2$, $s = 7$ and for $s = 11$

k	S=7				S=11			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	1	1	1	13	1	1	2	13
4	2	2	3	18	2	2	3	18
8	3	3	6	26	3	3	6	26
16	6	4	12	43	6	6	12	44
32	11	8	25	79	11	12	25	79
64	21	16	51	150	21	23	52	153
128	42	32	107	299	42	49	107	299
256	85	64	223	588	85	99	223	589
512	169	128	462	1163	169	203	464	1167

Table 3.3: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 2$, $s = 23$ and for $s = 99$

k	S=23				S=99			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	1	1	1	13	1	3	2	13
4	2	2	3	18	2	6	3	18
8	3	5	6	26	3	12	6	26
16	6	9	12	44	6	23	12	45
32	11	19	25	81	11	47	25	80
64	22	40	51	154	22	97	52	154
128	43	85	108	299	43	210	108	300
256	85	183	224	590	86	446	226	594
512	170	409	457	1176	170	935	467	1180

Table 3.4: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 3$, $s = 3$ and for $s = 5$

k	S=3				S=5			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	1	3	17	2	1	3	17
4	3	2	6	28	3	2	6	28
8	6	3	12	44	6	4	12	43
16	11	4	24	81	11	6	24	80
32	21	8	50	152	21	12	51	150
64	42	14	107	297	42	23	104	298
128	85	28	223	588	83	46	222	591
256	169	54	463	1167	169	90	458	1171
512	333	108	961	2336	337	191	953	2326

Table 3.5: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 3$, $s = 7$ and for $s = 11$

k	S=7				S=11			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	1	3	17	2	2	3	17
4	3	3	6	28	3	3	6	26
8	6	5	12	44	6	6	12	44
16	11	8	25	79	11	12	25	80
32	21	16	51	151	21	23	51	154
64	42	32	108	298	42	50	108	300
128	83	64	221	588	85	99	223	589
256	169	128	463	1173	169	203	464	1171
512	334	263	960	2337	338	437	951	2345

Table 3.6: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 3$, $s = 23$ and for $s = 99$

k	S=23				S=99			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	2	3	17	2	6	3	17
4	3	5	6	26	3	12	6	26
8	6	9	12	44	6	23	12	44
16	11	19	25	81	11	47	25	82
32	21	40	51	152	22	98	52	151
64	43	84	109	299	43	208	108	302
128	85	179	225	591	86	447	226	595
256	170	411	463	1173	172	932	469	1181
512	339	868	959	2342	336	1919	969	2360

Table 3.7: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 5$, $s = 3$ and for $s = 5$

k	S=3				S=5			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	1	3	17	2	1	3	17
4	3	1	6	26	3	2	6	27
8	6	2	12	44	6	4	12	44
16	11	4	24	80	11	7	25	81
32	21	8	51	153	21	12	51	151
64	42	15	107	298	42	23	107	299
128	85	28	220	589	85	46	220	589
256	169	54	462	1169	169	90	462	1171
512	337	108	954	2336	337	191	948	2337

Table 3.8: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 5$, $s = 7$ and for $s = 11$

k	S=7				S=11			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	1	3	17	2	2	3	18
4	3	3	6	26	3	3	6	26
8	6	4	12	44	6	6	12	44
16	11	8	25	79	11	12	25	81
32	21	17	52	154	21	23	51	152
64	42	32	107	298	42	49	107	300
128	85	64	222	589	85	99	223	592
256	169	129	463	1174	169	203	460	1173
512	337	263	949	2334	338	437	961	2347

Table 3.9: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 5$, $s = 23$ and for $s = 99$

k	S=23				S=99			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	2	3	17	2	6	3	17
4	3	5	6	26	3	12	6	26
8	6	9	12	44	6	23	12	44
16	11	19	25	79	11	47	25	81
32	22	39	52	154	22	97	52	152
64	43	84	108	300	43	210	109	302
128	85	179	225	593	84	448	227	596
256	170	409	462	1176	172	932	470	1178
512	339	868	959	2347	343	1915	975	2359

Table 3.10: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 9$, $s = 3$ and for $s = 5$

k	S=3				S=5			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	1	3	17	2	1	3	18
4	3	1	6	26	3	2	6	26
8	6	2	12	45	6	4	12	45
16	11	4	24	81	11	6	24	80
32	21	8	51	150	21	12	51	153
64	42	15	107	299	42	23	106	299
128	85	28	222	587	85	46	220	591
256	169	54	461	1167	169	90	464	1167
512	337	108	948	2340	337	192	952	2339

Table 3.11: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 9$, $s = 7$ and for $s = 11$

k	S=7				S=11			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	1	3	17	2	2	3	17
4	3	2	6	26	3	3	6	26
8	6	5	12	44	6	6	12	44
16	11	9	25	79	11	12	25	81
32	21	17	52	153	21	23	51	151
64	42	32	105	299	42	50	108	299
128	85	64	224	592	85	98	223	594
256	169	128	462	1172	169	203	461	1173
512	337	263	954	2338	338	437	954	2346

Table 3.12: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 9$, $s = 23$ and for $s = 99$

k	S=23				S=99			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	2	3	17	2	6	3	18
4	3	5	6	26	3	12	6	26
8	6	9	12	45	6	24	12	44
16	11	19	25	80	11	47	25	82
32	21	40	52	154	22	99	52	152
64	43	87	108	300	43	209	108	303
128	85	183	224	592	86	449	222	598
256	170	408	463	1181	172	928	468	1185
512	339	868	959	2354	343	1914	972	2374

Table 3.13: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 13$, $s = 3$ and for $s = 5$

k	S=3				S=5			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	1	3	18	2	1	3	18
4	3	2	6	26	3	2	6	26
8	6	2	12	44	6	4	12	44
16	11	4	24	81	11	7	24	81
32	21	8	51	154	21	12	51	154
64	42	14	107	298	42	24	106	299
128	85	28	221	591	85	45	222	591
256	169	56	461	1169	167	90	464	1175
512	337	108	955	2338	337	191	961	2342

Table 3.14: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 13$, $s = 7$ and for $s = 11$

k	S=7				S=11			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	2	3	18	2	2	3	18
4	3	2	6	26	3	3	6	26
8	6	4	12	45	6	6	12	45
16	11	8	25	81	11	13	25	82
32	21	17	52	151	21	25	52	154
64	42	33	107	300	42	50	106	300
128	85	64	223	592	85	98	222	593
256	169	128	463	1177	169	203	465	1176
512	337	263	957	2342	334	437	963	2347

Table 3.15: Table generation times (in microseconds) for two-level algorithms on an ultrasparc for the case $p = 32$, $a = 13$, $s = 23$ and for $s = 99$

k	S=23				S=99			
	ITABLE	SEARCH	NOREP	RIACS	ITABLE	SEARCH	NOREP	RIACS
2	2	3	3	19	2	6	3	18
4	3	4	6	26	3	12	6	27
8	6	9	12	44	6	23	12	44
16	11	19	25	80	11	47	25	81
32	21	40	52	155	22	99	52	156
64	43	86	108	302	43	208	109	303
128	85	183	225	593	86	452	224	597
256	170	405	460	1180	172	932	472	1185
512	339	868	956	2362	336	1913	969	2362

CHAPTER 4

MULTIPLE INDUCTION VARIABLES

The address generation problem becomes quiet complex if the array subscripts contain multiple induction variables (MIV). The challenge now is to generate the access sequence and hence the SPMD code for the case when the array A has been assumed to be distributed *block-cyclic*. This chapter aims at solving this problem by presenting efficient runtime techniques for address generation assuming the array has been distributed with *block-cyclic* distribution. We also address the issue of finding the starting element on each processor for a given global start. This method of finding *start* element can be extended for addressing issues in address generation for the case of one-level mapping. All the methods discussed in this chapter are implemented and the results obtained prove the efficacy of our methods.

4.1 PROBLEM STATEMENT

Consider a loop nest which contains an array A whose array subscripts contain multiple induction variable as shown below.

```
        REAL A(N)
!HPF$   TEMPLATE T(N)
!HPF$   PROCESSORS PROCS(p)
!HPF$   ALIGN A(j) WITH T(j)
!HPF$   DISTRIBUTE T(CYCLIC(k)) ONTO PROCS
        do i1 = 0, n1
             do i2 = 0, n2
                  A(l1 + i1 s1 + l2 + i2 s2) = ···
             enddo
        enddo
```

where s_1 and s_2 are the outer and inner loop strides respectively. Consider the following code in which the array A has been distributed CYCLIC(4) on to 4 processors, s_1 is 37 and s_2 is 2.

```
        REAL A(150)
!HPF$   TEMPLATE T(150)
!HPF$   PROCESSORS PROCS(4)
!HPF$   ALIGN A(i) WITH T(i)
!HPF$   DISTRIBUTE T(CYCLIC(4)) ONTO PROCS
        do i1 = 0, 3
             do i2 = 0, 9
                  A(37i1 + 2i2) = ···
```

Processor 0				Processor 1				Processor 2				Processor 3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143

(a): Layout of elements for array A for the case of MIV.

Processor 0	0	2	16	49	51	80	82	113	115	129

(b): Global addresses of elements accessed on Processor 0.

Processor 0	0	2	4	13	15	20	22	29	31	33

(c): Local addresses of elements accessed on Processor 0.

Figure 4.1: Figure to show the layout of elements accessed in case of multiple induction variables with $p = 4$, $k = 4$, $s_1 = 37$, $s_2 = 2$, $l_1 = 0$ and $l_2 = 0$

```
        enddo
    enddo
```

The distribution of the array A onto the processors is shown in Figure 4.1(a). The elements in shaded square boxes indicate the starting elements for each outer loop iterations whereas the elements in shaded circles indicate the starting elements for different processors for each outer loop iteration. For example, element 37 is a shaded box as it corresponds to the starting element for the second iteration of the outer loop. Elements 41, 45, 49 are in circles as they are the corresponding starting elements on remaining processors due to the second outer loop iteration. The elements accessed in the inner loop for each outer loop iteration which are not starting elements of either kind is shown as non-shaded square box over them. The problem now is to generate the access sequence for this type of array layout on each processor. Figure 4.1(b) shows the elements accessed due to the given loop on Processor 0; the SPMD code needs to generate the local addresses of these global elements on each processor. Figure 4.1(c) shows the local addresses which need to be generated by this SPMD code. This chapter deals with generating this set of elements in lexicographic order, using runtime approaches.

4.2 RELATED WORK

Kennedy, Nedeljkovic and Sethi [13] discuss the issue of address generation for the case of MIV. They view the problem as an integer lattice and use their approach discussed in their earlier work [12] to generate memory gap patterns. Since there is a pattern of repetition both at the outer and inner loop level, they construct memory gap tables at both levels of the loop. For each iteration of the outer loop, the inner loop behaves as a one-level mapping problem and a memory gap table just like the case discussed by Kennedy, Nedeljkovic and Sethi [12] is constructed. However for the outer loop, they present two methods of computing the memory gap table. They use the fact that there could be a maximum of *pk* different global offsets. In the first method, for each outer loop iteration they compute the global memory gap between the global starting locations for that outer loop and the first element in the inner loop accessed on a processor (in the current outer loop iteration). These gaps are stored in a table called (ΔS) which is constructed in $O(pk)$ time. However this starting location gaps gives the global address difference between the outer loop element to the processor starting element. So they need to translate this into the local index space for the processor, which can be expensive. In order to avoid this translation, they present a second method where instead of storing global starting gaps, they store the local memory gaps between the starting locations corresponding to consecutive outer loop iterations in a table (ΔG) for processor m. In order to construct these tables, they need to find the starting locations on a processor for each outer loop iteration. They propose a constant time algorithm for the case when the inner loop stride is less than or equal to the block size ($s \leq k$) and an $O(pk)$ algorithm for the case when the inner loop stride is greater than the block size ($s > k$). Hence their overall algorithm has a complexity of $O(p^2k^2)$. In the case of (ΔS) method, they incur additional overhead,

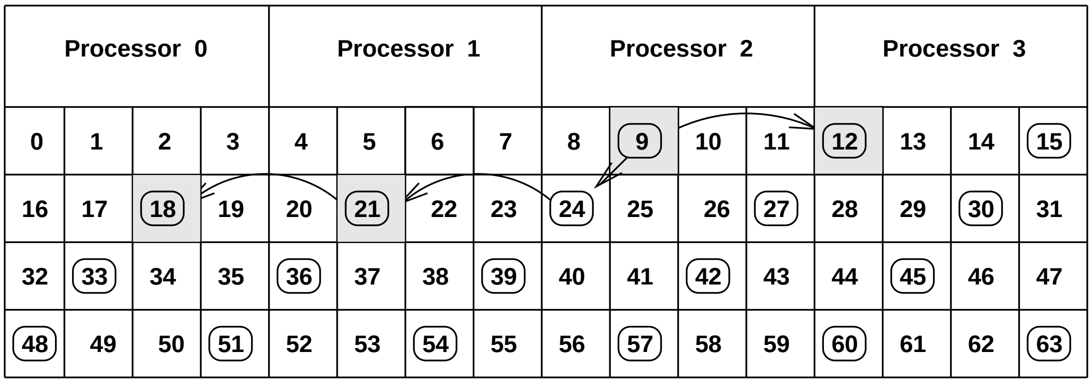


Figure 4.2: Stages in start computation ($s \leq k$) when s = 3, p = 4, k = 4

as they need to do the translation from global to local each time. With the (ΔG) table, they need to determine the global iteration number corresponding to each entry in the table, and they need to compute the local offset.

In order to avoid such expensive conversion and to reduce the complexity, we present an $O(pk)$ method to construct a table for the outer loop and an $O(k)$ solution for enumerating the starting element on each processor when access stride is greater than the block size and an constant time algorithm when the access stride is less than the block size for a given outer loop, thus deriving an $O(pk^2)$ algorithm. This coupled with the fact that we do not incur a global-to-local conversion makes this algorithm very efficient.

In the following sections we first present our method of computing the starting element on each processor for each outer loop iteration. Once we obtain the starting element on a processor, the enumeration of addresses accessed by the inner loop is nothing but address generation for one-level mapping with stride s_2.

4.3 METHOD TO FIND START ELEMENT ON PROCESSORS

The problem of finding start element should be addressed by the compiler for any type of mappings or distributions. Hence finding an efficient way of computing the start element is critical for any method. This section discusses general methods for computing starting elements on different processors for any given global starting element using efficient runtime techniques for *block-cyclic* distributions. Consider Figure 4.2 which shows the accessed elements of the array section $A(9 : 63 : 3)$ where A is distributed `CYCLIC`(4) onto 4 processors. The global start is 9 and lies on processor 2. The elements within boxes are the elements accessed. Hence for Processor 0 we see that elements 18, 33, 48, and 51 are accessed. Element 18, 21, 9, and 12 are the starting elements for processors 0, 1, 2 and 3 respectively. The problem now is to find these starting elements on each processor given a global start. Since each processor has knowledge of only its own elements, we need to find the local address of these starting elements on a per processor basis.

Several researchers have discussed the problem of finding start and hence generating the access sequence for block-cyclic distributions. The method by Chatterjee *et al.* [4] (reviewed in Chapter 1) incurs $O(k)$ time by solving k linear diophantine equations.

The solution by Kennedy, Nedeljkovic and Sethi [12] also incurs $O(k)$ overhead for finding the start element. Their method is very similar to the method of Chatterjee *et al.* [4]. However they also present new methods of finding start element [13]. These methods are divided mainly into two parts, the first part computes start element for the case $s \le k$ and is a constant time algorithm. The second case is when $s > k$ for which the worst case complexity is $O(pk)$.

If the stride is not greater than k and if *first* is the first element accessed in an iteration of the outer loop, the global index of the corresponding starting location for processor m is given by the following expression

$$first + \left\lceil \frac{mk - first \bmod pk + pk}{s} \right\rceil s.$$

However for the case when the stride is greater than the block size, their algorithm has a complexity of $O(pk)$ and suffers from huge runtime overheads.

Our method presented in the next section aims at addressing the issue of start computation. We present a constant time algorithm for determining the start element when $s \le k$ and a $O(k)$ algorithm for $s > k$. The method used in constructing the $O(k)$ algorithm has very little runtime overheads and results in practical times at runtime.

4.3.1 CASE WHEN $s \le k$

When the stride is less than or equal to the block size, there is at least one element accessed on each course. Hence it becomes easy to find the starting element on each processor. Given a global start and the processor m_g which owns the global start we can determine the local address of the starting elements on all the remaining processors. Let $course_g$ be the course on which the global start exists.

For the processor m_g which owns the global start, the local address of that element will be the local address of the starting location. However for a processor m which is to the left of m_g there are no elements on $course_g$. Hence in order to find the local address of the starting element we use the left basis vector as generated by the one-level algorithm to go to an accessed element on the next course *i.e.,*$coures_g + 1$. From this point onwards we find a factor of s needed to jump into the processor $m's$ space such that we determine the first element accessed in that row. This factor can be determined by just one integer divide operation and hence we can determine the local address of the starting element. For example in Figure 4.2 we have 9 as the global start and processor $m_g = 2$. Processors $m = 0$ and $m = 1$ lie to the left of m_g *i.e.,* $m < m_g$. The basis vectors for this lattice as produced by the one-level algorithm [27] are $\vec{l} = (1, -1)$ and $\vec{r} = (0, 3)$. Hence the

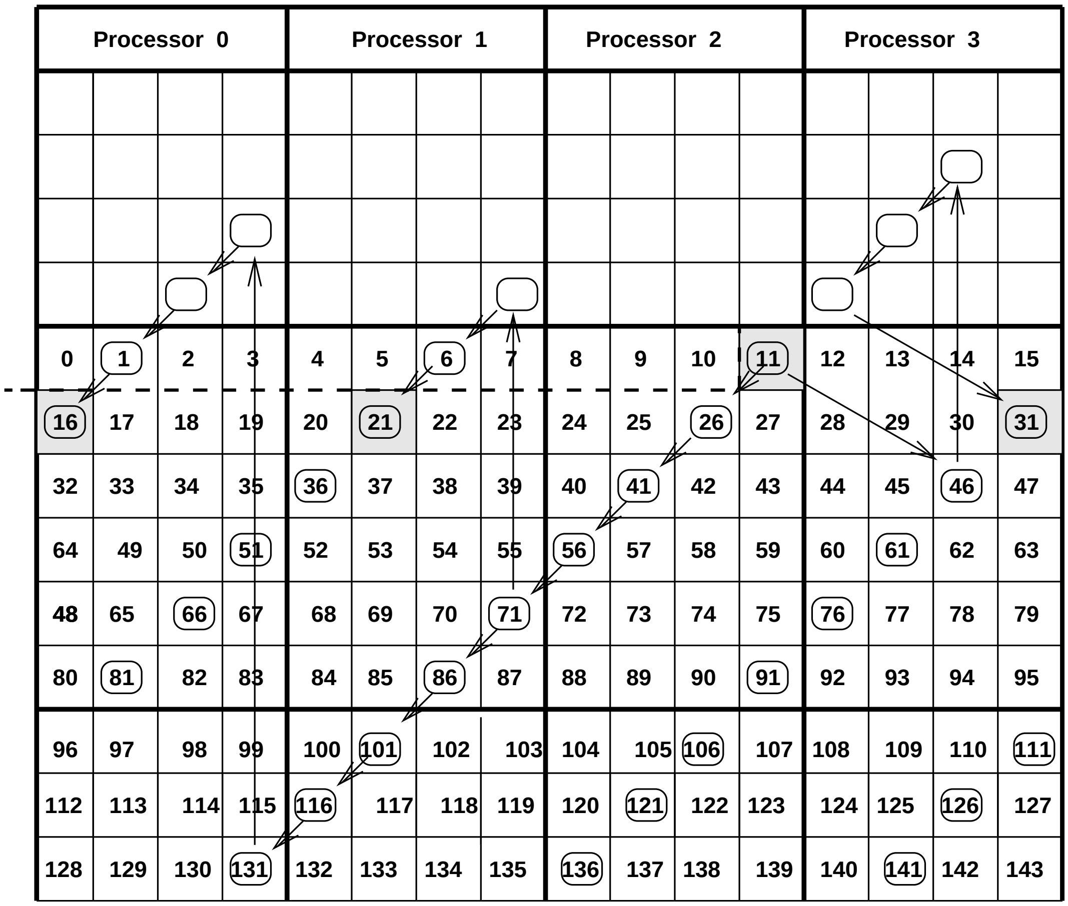


Figure 4.3: Stages in start computation ($s > k$) when s = 5, p = 4, k = 4, l = 11

left vector $\vec{l}$ takes us to the next course ($course_g + 1$) to the element 24. In order to find the starting element for processor 0 we find the factor of s needed to enter the processor space. For the above example factor of 2 is sufficient to get to the element 18 which is the starting element for processor 0. Similarly a factor of 1 is sufficient to get to the element 21 for processor 1.

If the processor m is to the right of m_g *i.e.,* when $m > m_g$, then the starting element is guaranteed to lie on $course_g$. We now need to find the factor of s needed to jump to this element. For the above example a factor of 1 is sufficient to get to the element 12 which is the starting element for processor 3.

The complexity of the algorithm is $O(1)$ and has a very little overhead during runtime.

4.3.2 CASE WHEN $s > k$

If the stride is greater than the block size, then there might be courses which might not have any accessed element on them. Hence it is not possible to find the starting element in constant time. Our algorithm presented here uses efficient runtime techniques for finding the start element in $O(k)$ time.

The basis vectors generated by the application of the one-level algorithm yields us vectors $\vec{l}$ and $\vec{r}$ which help us move from one accessed element to the other. Hence given a global start and the processor m_g which owns the start, we use the basis vectors to determine the starting elements on remaining processors. We also make use of the fact that the entire set of accesses repeat after $\frac{s}{\gcd(s,pk)}$ courses.

For a processor m which lies to the left of m_g we first determine the multiple of $\vec{l}$ needed to be able to get into the processor m's space. However it is not guaranteed at this point that we hit an element in the block which holds the starting element. Now we make use of the fact that if $course_g$ is the course on which the global start exists, we can be sure that the starting element will lie within the next $\frac{s}{\gcd(s,pk)}$ courses. So in order to ensure that we enter the block that holds the starting element, we find the factor of $\frac{s}{\gcd(s,pk)}$ needed to jump into this block. Once we enter this block we now can traverse the lattice using the basis vectors until we hit the start element, which is an $O(k)$ operation. Figure 4.3 shows a case where the stride is greater than the block size. The global start is 11 and $m_g = 2$. The basis vectors for this example are $\vec{l} = (1, -1)$ and $\vec{r} = (2, 3)$. In order to find the start on processor 0 we first find the factor of $\vec{l}$ needed to reach the first element 131 on m. This element lies in the second block of repeating accesses. We then find the factor (2 in this case) needed to jump to the block which contains $course_g$ (0 in this case). We then traverse the lattice from this point until we hit 16 which is the first element accessed after 11 on processor m.

For processor m that lies to the right of m_g, instead of using the left vector we use the right vector in order to get into the processor space. Then we find the factor needed to jump to the block which holds the starting element. Then traversing this block using the basis vectors will yield us the starting element. For the Processor 3 in Figure 4.3 we first use the right vector to get to element 46. We then jump to the block above the course which holds the global start and then traverse the lattice until we hit the element 31 which is the starting element on the processor.

Since the maximum amount of traversal required in the block is k, this is an $O(k)$ algorithm. This algorithm also does not have huge runtime overheads.

We use this algorithm to determine the starting elements on each processor for each outer loop iteration in the case of MIV. Here we assume that the array has been distributed block-cyclic. The detailed algorithm for block-cyclic for MIV is discussed in the next section.

4.4 TABLE BASED METHODS TO SOLVE MULTIPLE INDUCTION VARIABLES FOR CYCLIC(K) DISTRIBUTIONS

In case of block-cyclic distribution with block size k, referred to as `CYCLIC(`k`)`, blocks of the template of size k are allocated to the processors in a cyclic fashion. If we need to enumerate elements in lexicographic order we need to enumerate the inner loop for any fixed outer loop iteration. In this section we present efficient runtime techniques to generate the set of accesses. We later contrast our technique with that of Kennedy, Nedeljkovic and Sethi [13].

Several methods which have addressed the issue of address generation for one-level mapping have used the repetitive pattern of accesses. The same pattern can be found in case of MIV for accesses in both inner and outer loops. Just by observing accesses of outer loops, we see that the pattern of accesses is found to repeat itself after $\frac{s_1}{\gcd(s_1,pk)}$ courses, while that of the inner loop alone repeats after $\frac{s_2}{\gcd(s_2,pk)}$ courses. As discussed in Section 4.2 in order to exploit these pattern of accesses we need a table of size at most pk for each processor m which records the local addresses of starting elements for each outer loop iteration. For the inner loop we maintain a pattern table of size at most k, as in the case of single index.

Our approach to the problem of finding memory access sequence for MIV makes use of the repetitive nature of accesses for both inner and outer loops. As mentioned before we record the local addresses of the starting elements for the first pk iterations of the outer loop into a *Start* table of size pk. This is constructed by running the algorithm as discussed in Section 4.3. This construction takes $O(pk^2)$ time as the start computation is of $O(k)$ for each outer loop iteration. The reduced complexity when compared to that of the method by Kennedy, Nedeljkovic and Sethi [13], as well as the faster way of computing start serves to be a big advantage. Further improvements can be incorporated by constructing a table to record the end elements for the same number of iterations. This table called *End*, can be constructed using the same approach as that of *Start* but with a slight modification.

The enumeration of the inner loop can be performed by using a pattern table just like the one discussed in Section 4.2. We use a nextoffset table which keeps a record of the indices needed to lookup the pattern table; the construction of which is similar to the one mentioned by Kennedy, Nedeljkovic and Sethi [13]. The initial index into this nextoffset table is the offset of the starting element. Hence in order to avoid expensive mod operations to compute this offset during runtime, we can pre-compute the same and record this into a table called *Offset* during the start computation stage. The construction of this table does not require any additional work as the information about the offset is already available during start computation.

The execution stage involves using all the above constructed tables, and thus avoiding expensive runtime computations. The complete algorithm is as given in Figure 4.4.

The algorithm in Figure 4.4 first computes the pattern and nextoffset tables using the one-level algorithm for the inner loop. Lines 1–3 show the steps involved in the construction of these tables. These tables are needed to iterate through the accessed

Input: Layout parameters (p, k), loop limits $(0 : n_1 : s_1)$ for outer loop and $(0 : n_2 : s_2)$ for the inner loop.
Output: Accessed elements
Method:

$d_1 \leftarrow \gcd(s_1, pk)$
$d_2 \leftarrow \gcd(s_2, pk)$
$(length, \Delta M, \Delta Nextoffset) \leftarrow one\_level(p, k, s_2, m, d_2)$
for $i = 0, pk - 1$ **do**
  $(Start[i], Offset[i]) \leftarrow start\_computation(m, i_1 s_1, \vec{l}, \vec{r}, m_g, d_1)$
  $End[i] \leftarrow end\_computation(m, i_1 s_1 + n_2 s_2, \vec{l}, \vec{r}, m_g, d_1)$
enddo
$(t, count) \leftarrow 0$
for $i = 0, n_1, s_1$ **do**
  $j_{start} \leftarrow Start[count] + t$
  $j_{end} \leftarrow End[count] + t$
  $local\_offset \leftarrow Offset[count]$
  $count \leftarrow count + 1$
  if $(count = pk)$ **then**
    $t \leftarrow t + \frac{s_1}{d_1}$
    $count \leftarrow 0$
  endif
  while $(j_{start} \leq j_{end})$ **do**
    $A[j] \leftarrow \cdots$
    $j_{start} \leftarrow j_{start} + \Delta M[local\_offset]$
    $local\_offset \leftarrow \Delta Nextoffset[local\_offset]$
  enddo
enddo

Figure 4.4: Algorithm to determine the accessed element in case of MIV for `CYCLIC(`k`)` distributions

△ M	2	2	2	2

(a)

△ Nextoffset	2	3	0	1

(b)

Start	0	13	20	29	40	·········

(c)

Offset	0	1	0	11	0	·········

(d)

End	7	15	23	35	43	·········

(e)

Figure 4.5: Data structures used to find the access sequence for the case when the bounds for outer and inner loop are $(0 : n_1 : 37)$ and $(0 : 18 : 2)$ respectively with $p = 4$, $k = 4$, in case of *block-cyclic* distribution for MIV array statements

elements in the inner loop. Given that the outer loop pattern starts repeating after $\frac{s_1}{\text{gcd}\,(s_1,pk)}$ courses, the next step in the algorithm is to compute and store the local starting and ending elements for the first repeating set of the elements in the outer loop iterations. This is done by iterating through each outer loop iteration and computing the starting and ending local addresses on processor m using the method to compute start as discussed in Section 4.3. Lines 5–9 show the steps involved in construction the *Start* and *End* table. The enumeration of the inner loop involves, a entry from the nextoffset table to determine the corresponding index into the pattern table. The index into this nextoffset is the offset of the starting element from the beginning of the course. As mentioned before, we can record and store this offset during the construction of the *Start* table into an *Offset* table. Line 6 shows the construction of *Offset* table along with *Start* computation. This build-up of the table will avoid calculation of the offsets during runtime. As discussed before, the construction of all the above mentioned tables incur $O(k)$ running time. All the above mentioned tables are computed only once before the code enters the execution stage.

During execution, for each outer loop iteration, it needs an entry from the *Start*, *End* and *Offset* table to determine the starting element, starting offset and end element. It then enumerates the elements in the inner loop within the bounds of the start and end element. Lines 11–25 illustrate the execution order followed to generate accessed elements in case of MIV. The while loop in the code, shows the enumeration order for elements within the inner loop. This enumeration involves reading of entries from the nextoffset and pattern tables just like the way it is described in the method by Kennedy, Nedeljkovic and Sethi [13].

For the example in Figure 4.1, we show the entries in *Start*, *End* and *Offset* tables for Processor 0 in Figures 4.5(c), (d) and (e) respectively. Let us consider the element 37 due to the second outer loop iteration. The corresponding local start, end and offset elements for Processor 0 are found to be 13, 15, 1 respectively using entries from the *Start*, *End* and *Offset* tables respectively. Once all the required elements are found, the execution stage will enumerate addresses 13, 15 for this outer loop iteration. This involves reading of entries from the nextoffset and deltam table as shown in Figures 4.5(a) and (b). The next step involves determining the start, end and offset element for the third iteration of the outer loop (74). Elements 20, 23, 0 constitute the start, end and offset elements respectively. The execution stage enumerates elements 20,22. This process of execution for each outer loop iteration continues, until we enumerate all the inner loop elements for the complete set of outer loop iterations. The algorithm does not have the overhead of computing the end and start offset for each outer loop iteration.

4.5 EXPERIMENTS AND RESULTS

Several variations of the algorithm mentioned in the previous section can be developed based on the problem parameters. If the length of the inner loop is not large enough, building a pattern table and nextoffset table for the inner loop might bring in extra overhead. In this case we can adopt a demand driven approach for computing accessed elements in the inner loop. Similar analogy holds true if the extent of outer loop is not long enough. Here we may resort to computing the start and end element at runtime using the method mentioned in Section 4.3 rather than constructing a pk table. Similarly if both the extent of the inner and outer loop are not large enough we may benefit from not having to construct any tables.

In order to test and verify all the different possible benefits of constructing a table, we implemented different versions of the algorithm mentioned in Figure 4.4. Below we show the different variations of the MIV algorithms based on the different tables used in the algorithm.

1. Start table, offset table, end table along with the inner loop pattern table are constructed. This method is similar to the one discussed in Figure 4.4. We term this method as SEOD.

2. Start table, end table, inner loop pattern table are constructed in this variation of our algorithm. The offsets of the starting element are generated at runtime. We term this method as SED.

3. Start table, offset table, inner loop pattern table are constructed. End elements are computed at runtime. We term this method as SOD.

4. We construct only the inner loop pattern tables. The remaining entries are generated at runtime. We term this method as DELTAM.

5. This variation has no tables. The entire set of elements are constructed at runtime. We term this method as NO_TABLES.

All the above mentioned algorithms were implemented and run on a Sun ultrasparc. The number of processors was fixed at 32. All the algorithms were run for the same number of iterations per processor so that we have a base for our comparison. If $numits$ is the number of iterations executed per processor the stretch of the inner loop was chosen to be $s_2 * \left(\sqrt{numits * p} - 1\right)$. Our value for s_1 was based on the value chosen for s_2 and was equal to $s_2 * \sqrt{numits * p} + 1$. The stretch of the outer loop was then chosen to be $s_1 * \left(\sqrt{numits * p} - 1\right)$. We then varied the value of the block-size and recorded the time taken for the execution of the node-code for all the different variations of the algorithms. We also contrast our methods of solving address generation for MIV, to that of the method by Kennedy, Nedeljkovic and Sethi [13] which we term as RICE in our results. Their method involves construction of a start table which holds the local memory gaps between successive outer loop iterations. They compute the offsets and the end element at runtime. They however use the same inner loop structure as our method, by preserving a pattern and nextoffset table.

Tables 4.1, and 4.2 show the times taken for the various algorithms for different block-sizes and inner loop strides s_2, when the iterations per processor was fixed at 2500 and

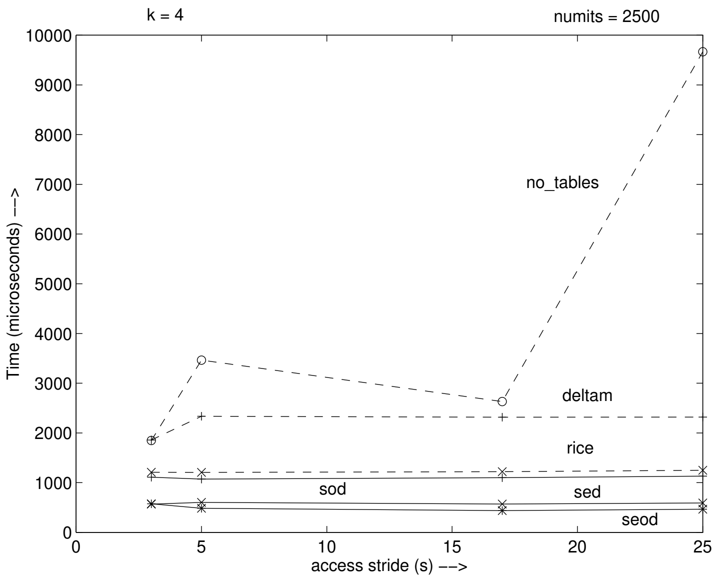


Figure 4.6: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 2500 and $k = 4$

5000 respectively. Figures 4.6–4.19 show the effect of the inner loop stride s_2 on the execution time as a function of the block size k.

These results demonstrate that the SEOD method perform very efficiently in almost all the cases as expected. This is due to the benefit we get by constructing all the useful pattern tables, thereby eliminating the associated runtime overheads. The method by RICE incurs high runtime overheads and hence is slower than the other methods. The times from SOD and SED methods lies in between the times of SEOD and RICE. The DELTAM and NO_TABLES methods always perform badly, due to the expensive conditionals and runtime overheads.

4.6 Chapter summary

In this chapter we have shown several techniques to generate accessed elements in an array, when the array subscripts contain multiple induction variables. Our techniques involve building tables for exploiting the pattern of accesses. We have presented a new general way of computing start element on a processor. This method does not incur much runtime overheads, and can serve to be big advantage in order to generate starting elements on each processor for a given global start. The complexity of this method is $O(k)$ when the block size is greater than the stride, and constant time otherwise. We use this technique of computing start to present an $O(pk^2)$ algorithm to compute accessed elements in case of MIV. The reduced complexity and benefit of constructing several pattern tables proves to be beneficial during execution for MIV.

Table 4.1: Execution times (in microseconds) for MIV algorithms on an ultrasparc for the case when $p = 32$ and number of iterations is 2500

k	s_2	SEOD	SED	SOD	RICE	DELTAM	NO − TABLES
4	3	576	568	1110	1209	1857	1849
	5	485	603	1071	1207	2336	3464
	17	437	571	1100	1220	2316	2631
	25	466	592	1132	1249	2320	9667
8	3	583	745	1149	1202	1931	2331
	5	444	575	1097	1204	1814	2159
	17	452	584	1081	1225	2400	2617
	25	479	660	1086	1238	2507	5968
16	3	445	752	1176	1193	1923	1838
	5	454	573	1045	1209	1813	1774
	17	487	579	1094	1214	2493	2475
	25	484	605	1127	1238	2736	4319
32	3	468	571	1046	1193	1922	2326
	5	445	572	1096	1197	1786	1901
	17	559	582	1089	1207	1808	1902
	25	479	668	1072	1232	1830	2440
64	3	451	608	1056	1183	1853	1855
	5	493	568	1037	1186	1787	2969
	17	579	694	1211	1201	1808	2494
	25	491	602	1079	1231	1830	1931
128	3	467	584	1049	1197	1825	2322
	5	467	628	1049	1232	1810	2162
	17	471	682	1227	1211	1805	1922
	25	553	620	1090	1232	1832	2577
256	3	480	593	1058	1160	1810	1845
	5	467	583	1065	1153	1797	1774
	17	470	622	1054	1198	1798	3200
	25	509	618	1081	1226	1832	1987

Table 4.2: Execution times (in microseconds) for MIV algorithms on an ultrasparc for the case when $p = 32$ and number of iterations is 5000

k	s_2	SEOD	SED	SOD	RICE	DELTAM	NO − TABLES
4	3	860	1089	1772	1840	2784	2904
	5	860	1046	1755	1853	3535	5881
	17	990	1159	1827	1901	3616	5326
	25	1070	1271	1980	2001	3719	19396
8	3	916	1082	1733	1829	2793	3885
	5	872	1128	1770	1839	2792	3565
	17	1104	1290	1986	2015	3830	5147
	25	1107	1234	2056	1979	4044	11695
16	3	876	1066	1755	1827	2772	2937
	5	955	1057	1757	1827	2813	2770
	17	1095	1280	2011	1983	3939	4027
	25	1129	1339	2104	2044	4385	8698
32	3	917	1084	1770	1807	2758	3873
	5	939	1061	1779	1806	2791	3019
	17	1120	1280	2043	1994	3099	3375
	25	1283	1356	1999	2095	3087	6739
64	3	930	1053	1739	1797	2771	2918
	5	917	1077	1743	1799	2772	5147
	17	1154	1332	2007	2024	3017	4930
	25	1290	1349	2038	2104	3059	3876
128	3	916	1065	1763	1793	2773	3892
	5	1129	1286	1895	1804	2772	3568
	17	1171	1368	1964	2008	2959	3633
	25	1225	1355	2021	2029	2931	6974
256	3	932	1073	1769	1759	2752	2932
	5	1093	1278	1897	1781	2772	2759
	17	1179	1336	2040	2037	3009	5951
	25	1266	1347	2061	2128	3114	3938

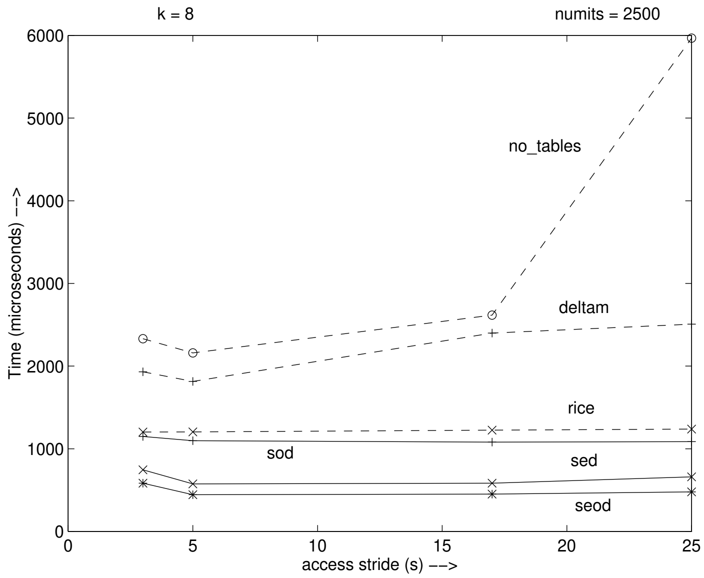


Figure 4.7: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 2500 and $k = 8$

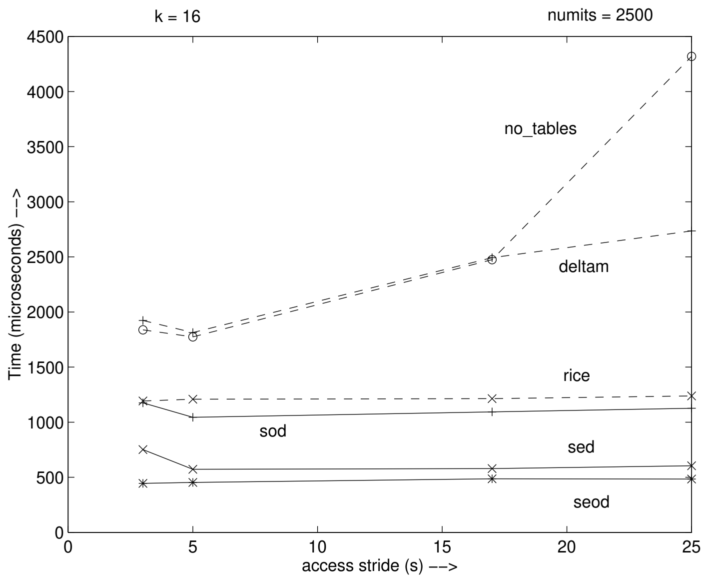


Figure 4.8: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 2500 and $k = 16$

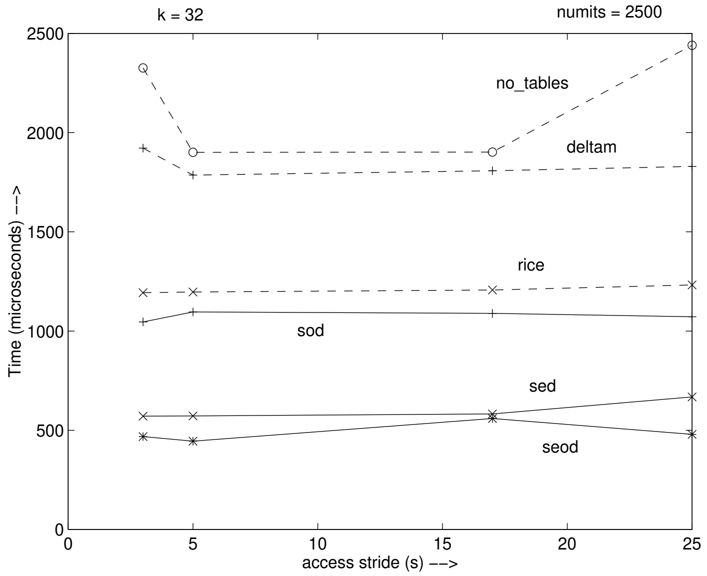


Figure 4.9: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 2500 and $k = 32$

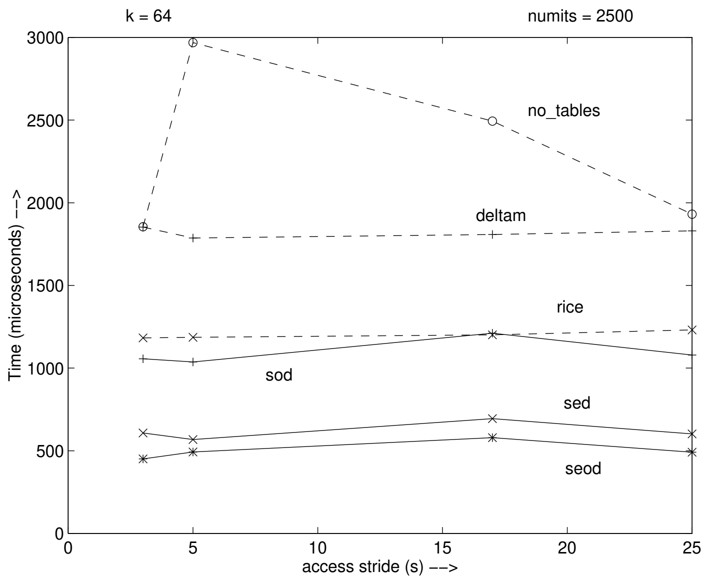


Figure 4.10: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 2500 and $k = 64$

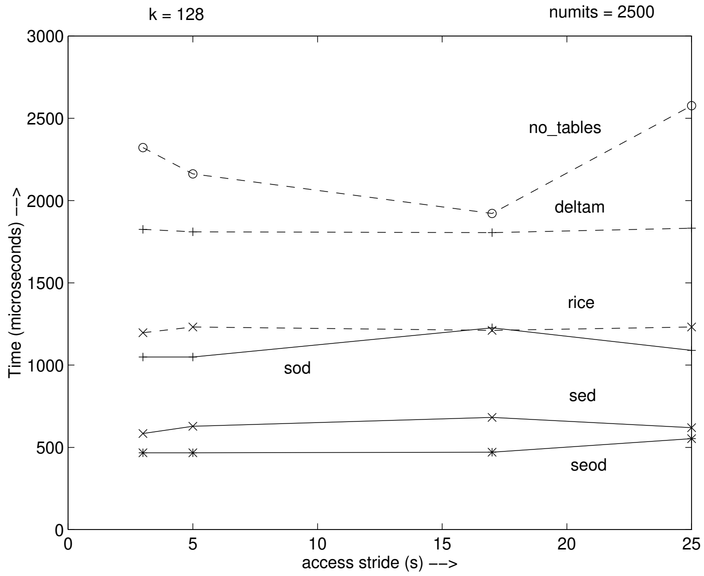


Figure 4.11: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 2500 and $k = 128$

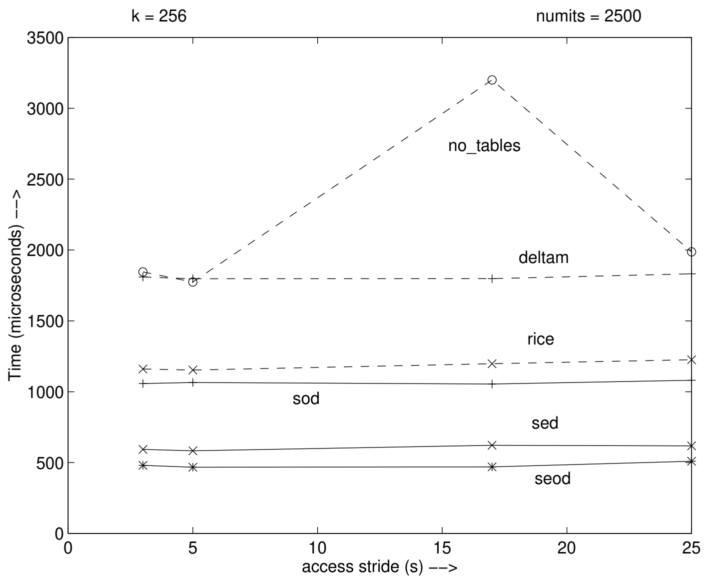


Figure 4.12: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 2500 and $k = 256$

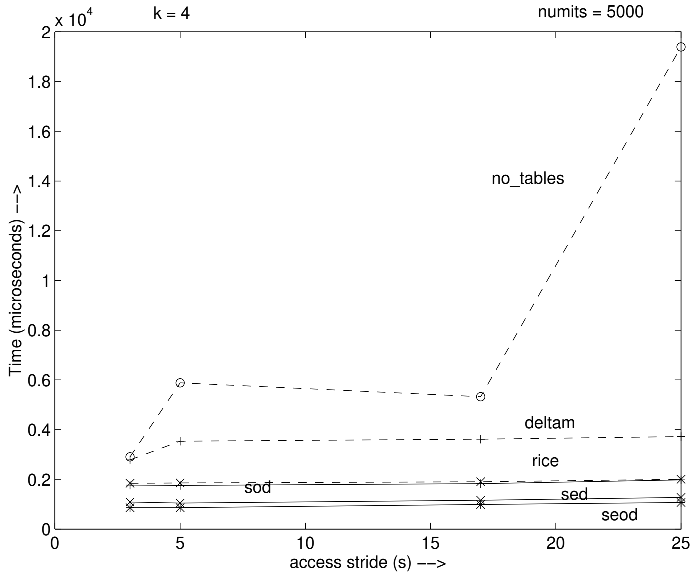


Figure 4.13: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 5000 and $k = 4$

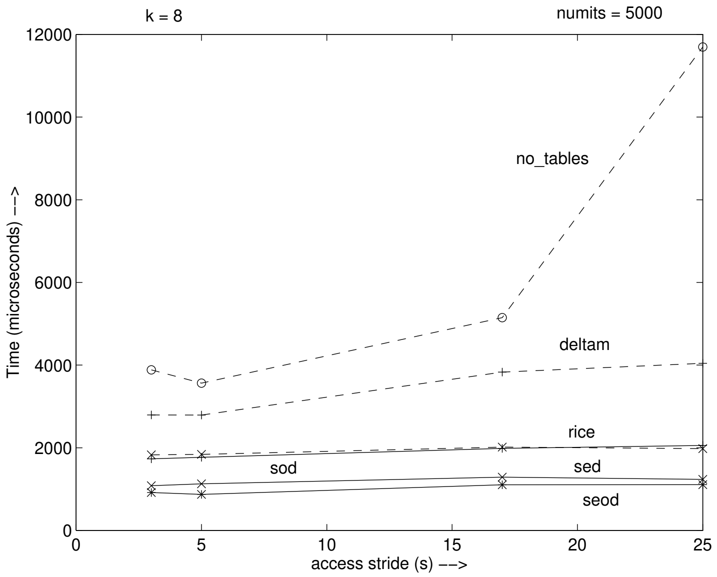


Figure 4.14: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 5000 and $k = 8$

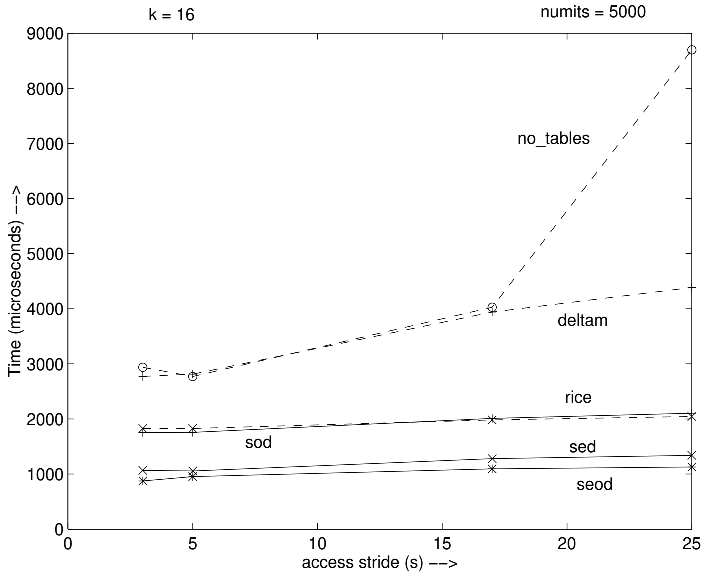


Figure 4.15: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 5000 and $k = 16$

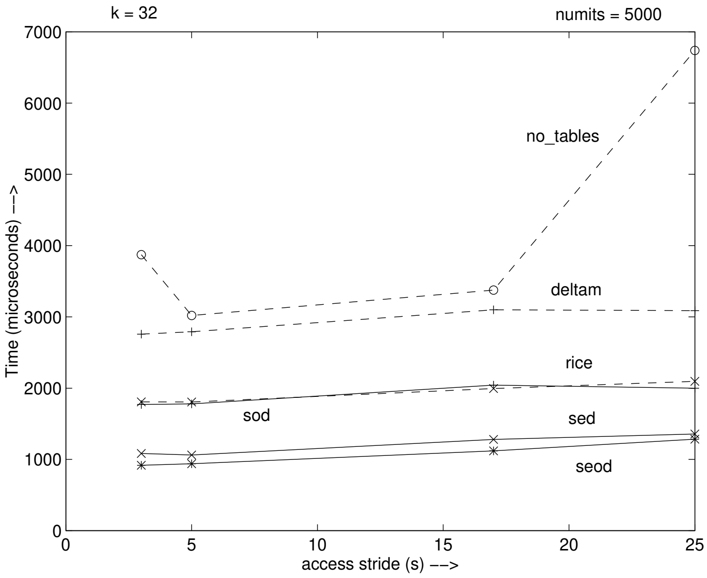


Figure 4.16: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 5000 and $k = 32$

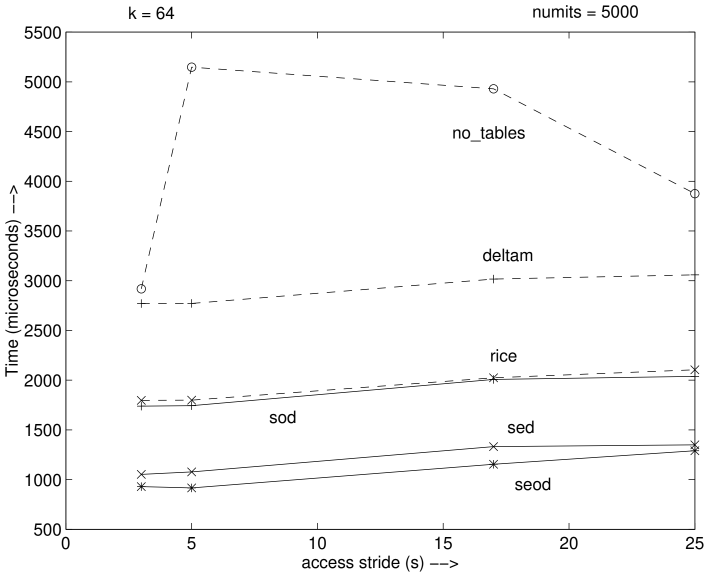


Figure 4.17: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 5000 and $k = 64$

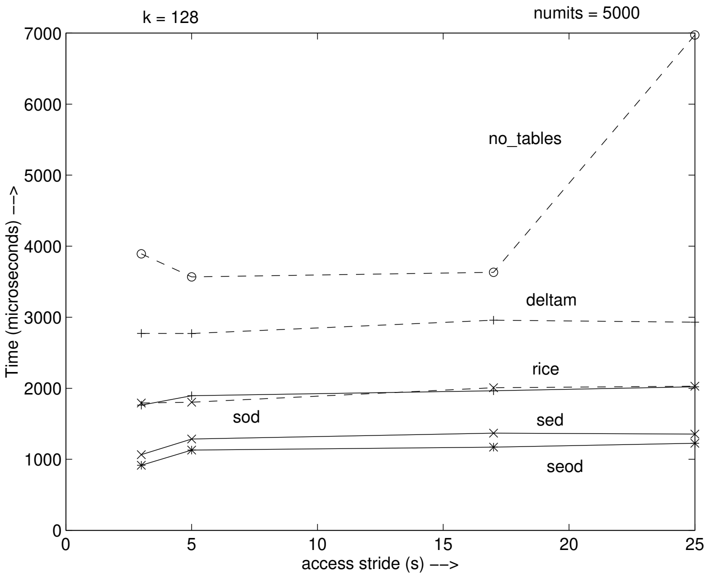


Figure 4.18: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 5000 and $k = 128$

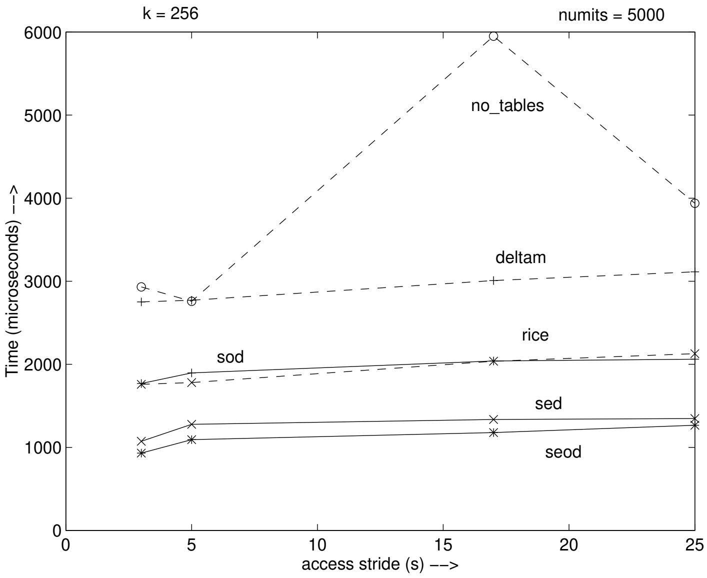


Figure 4.19: Total execution times for MIV algorithms for the case when $p = 32$ processors and number of iterations is 5000 and $k = 256$

CHAPTER 5

CONCLUSION

Compilers for data-parallel languages such as High Performance Fortran (HPF) should generate efficient node-code and provide effective run-time supports for distributed memory machines. In compiling array statements for distributed memory machines, the compiler needs to specify the sequence of local memory addresses for each array accessed by each processor. In addition it should generate the set of elements involved in communication between processors. In this thesis we have addressed issues concerning communication generation for distributed memory machines in case of block-cyclic distribution. We have also presented techniques to generate the local memory addresses for the case of non-unit alignment strides and for arrays which contain multiple induction variables.

In Chapter 2, we presented new ways of generating communication sets for node-code execution. We exploited the access pattern repetition and derived algorithms for determining the communication pattern. We then addressed the issue of treatment of non-local elements by presenting two schemes which extended the notion of a local array by providing storage for non-local elements (called *overlap regions*) interspersed throughout the storage for the local portion. The two schemes, namely *course padding* and *column padding* enhance locality of reference significantly at the cost of a small overhead due to unpacking of messages. The performance of these schemes was found to be up to 30% better than the traditional buffer-based approach. In addition, several message optimizations such as offset communication, message aggregation and coalescing were also discussed.

In order to address the issue involved in generating addresses for the case of non-unit alignment strides, we presented in Chapter 3 several techniques which construct pattern tables in order to reduce run-time overheads. Simple methods to enumerate address sequences using tables and search techniques were presented. It was shown that the search techniques performed well for small block and access strides where as the table of offset method performed significantly well for large block and access strides. Several experiments were performed to test the efficacy of our approaches. We showed a factor of 6 to 10 times improvements when compared to other existing methods. Savings in memory and the choice of method involved in constructing this table proves this technique to be the fastest existing solution to the problem.

Arrays which contain multiple induction variables were discussed in Chapter 4. We presented several efficient runtime techniques to enumerate the access sequences. We presented an $O(pk^2)$ algorithm for computing the accessed elements in case of MIV. This method involved a new general way of computing starting element on a processor for a given global start in $O(k)$ time. Several variations of the algorithm were discussed and the effect of pattern table generation was shown by running several experiments. Results indicated that our algorithms perform significantly better when compared to other existing solutions for this problem.

There are many other issues to be addressed in communication generation for block-cyclic distributions. Future work needs to be done to explore the benefits of message coalescing, message aggregation and in the case of offset communication. Communication for the two-level mappings can pose an interesting problem. One can also work towards performing communication for the case of arrays containing MIV. As far as MIV is concerned, one can work towards generating accessed sequences in non-lexicographic order.

BIBLIOGRAPHY

[1] A. Ancourt, F. Coelho, F. Irigoin and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proc. of the 4th Workshop on Compilers for Parallel Computers,* Delft, The Netherlands, December 1993.

[2] S. Benkner. Handling block-cyclic distributed arrays in Vienna Fortran 90. In *Proc. International Conference on Parallel Architectures and Compilation Techniques,* Limassol, Cyprus, June 1995.

[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, editors, *Language, Compilers and Runtime Environments for Distributed Memory Machines*, pages 39–62. 1992.

[4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. 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, Dept. of Computer Science, Rice University, Dec. 1991.

[6] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[7] P. M. Gruber and C. G. Lekkerkerker. *Geometry of numbers.* North-Holland Mathematical Library Volume 37, North-Holland, Amsterdam, 1987 (Second edition).

[8] S. Gupta, S. Kaushik, C. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. To appear in *Journal of Parallel and Distributed Computing*, 1996.

[9] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming,* 2(1-2):1–170, 1993.

[10] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. In *Proc. ACM International Conference on Supercomputing*. July 1994

[11] S. Kaushik. Compile-time and run-time strategies for array statement execution on distributed-memory machines. Ph.D. Thesis, Department of Computer and Information Science, The Ohio State University, 1995.

[12] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proc. of Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[13] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions In *Proc. ACM International Conference on Supercomputing,* Madrid, Spain, July 1995.

[14] K. Kennedy, N. Nedeljkovic, and A. Sethi. Communication generation for cyclic(k) distributions. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, B. Szymanski and B. Sinharoy (Eds.), Kluwer Academic Publishers, 1995.

[15] C. Koelbel. Compile-time generation of communication for scientific programs. In *Supercomputing '91*, pages 101–110, Nov. 1991.

[16] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran Handbook.* The MIT Press, 1994.

[17] T. MacDonald, D. Pase, and A. Meltzer. Addressing in Cray Research's MPP Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers* (Vienna, Austria, July 1992), Austrian Center for Parallel Computation, pp. 161–172.

[18] S. Midkiff. Local iteration set computation for block-cyclic distributions. In *Proc. International Conference on Parallel Processing*. Vol. II, pages 77–84, August 1995.

[19] J. Ramanujam. Non-unimodular transformations of nested loop. In *Proc. Supercomputing 92,* pages 214–223, Nov. 1992.

[20] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing,* 9(4):365-389, Dec. 1995.

[21] C. van Reeuwijk, H.J. Sips, W. Denissen, and E. M. Paalvast. Implementing HPF distributed arrays on a message-passing parallel computer system. CP Technical Report series, TR9506, Delft University of Technology, 1995.

[22] 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, Feb. 1993.

[23] E. Su, A. Lain, S. Ramaswamy, D.J. Palermo, E.W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proc. 1995 ACM International Conference on Supercomputing,* Barcelona, Spain, July 1995.

[24] A. Thirumalai. Code generation and optimization for High Performance Fortran. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, August 1995.

[25] A. Thirumalai and J. Ramanujam. Code generation and optimization for array statements in HPF. Technical Report TR-94-11-02, Dept. of Electrical and Computer Engineering, Louisiana State University, Nov. 1994. revised August 1995

[26] A. Thirumalai and J. Ramanujam. An efficient compile-time approach to compute address sequences in data parallel programs. In *Proc. 5th International Workshop on Compilers for Parallel Computers,* Malaga, Spain, pp. 581-605, June 1995.

[27] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs Using Integer Lattices. In *Languages and Compilers for Parallel Computing,* C.-H. Huang *et al.* (Editors), Lecture Notes in Computer Science, Vol. 1033, pages 191–208, Springer-Verlag, 1996.

[28] A. Thirumalai and J. Ramanujam. Address sequence determination using closed forms for basis vectors of the array section lattice. Technical Report, Dept. of Electrical and Computer Engineering, Louisiana State University, Apr. 1995.

[29] A. Thirumalai, J. Ramanujam, and A. Venkatachar. Communication generation and optimization for HPF. In *Languages, Compilers, and Run-Time Systems for Scalable Computers,* B. Szymanski and B. Sinharoy (editors), Kluwer Academic Publishers, Chap. 29, pages 311–316, Norwell, MA, 1995

[30] A. Venkatachar, J. Ramanujam and A. Thirumalai. Generalized overlap regions for communication optimization in data parallel programs. To appear in *Languages and Compilers for Parallel Computing,* U. Banerjee *et al.* (Editors), (Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing, August 1996). Lecture Notes in Computer Science, Springer-Verlag, 1997.

[31] Michael Wolfe. *High performance compilers for parallel computing.* Benjamin/Cummings Publishing Co., 1996.

VITA

Arun Venkatachar was born in Bangalore, India, on May 22, 1971. After completing his high school, he joined the Bachelor of Engineering program in Bangalore University, India, in June 1989. He received his bachelor's degree in Electronics in Spring 1993. He then went on to work as a Project Engineer at the Raman Research Institute, India, from August 1993 to August 1994. He was involved in the design and development of hardware cards for the Giant Meter Radio Telescope project.

Arun joined the master's program in the Louisiana State University in Fall 1994. He was hired as a Research Assistant at the Dept of Electrical Engineering where he worked in the area of Parallelizing compilers until Summer 1996. He has written book chapters and conference papers in this research area. While at school he has also involved himself in other activities. He served as the president of the Indian Student Association from August 1995 to August 1996. His next goal is to obtain practical experience in the design and development of compilers in the industry.

He is a candidate for a master of science degree in Electrical Engineering, Louisiana State University, at the Fall 1996 commencement.

Venkatachar, Arun T., B.E. in Electronics, Bangalore University, 1993
Master of Science in Electrical Engineering, Fall Commencement, 1996
Major: Electrical Engineering
Efficient Address and Communication Generation for Data-Parallel Programs
Thesis directed by Associate Professor Jagannathan Ramanujam
Pages in thesis, 95. Words in abstract, 272.

ABSTRACT

Programming distributed memory machines has remained a difficult task in spite of their great potential. As a result, languages such as High Performance Fortran (HPF) have been designed to ease programming while preserving high levels of performance. A compiler for such a language is responsible for distributing data and computation onto the processors based on directives from the user and for generating node code along with sends and receives for each processor. This thesis addresses issues in code generation for programs with HPF-style data mappings.

Algorithms for deriving the communication sets for each processor are presented; these algorithms exploit a repeating communication pattern. In addition, local storage schemes using the idea of generalized overlap regions are presented for storing the received data on each processor; this results in increased data locality. In addition, communication optimizations such as offset communication, message coalescing and aggregation are discussed. Experiments show an average of 30% improvement in execution time due to overlap regions.

Address generation for two-level mapping resulting from non-unit alignment strides is handled in this thesis using simple ideas such as the use of tables and search techniques. These techniques are shown to be six to ten times faster in practice than existing methods. The search technique works extremely well for small strides and block sizes, while the table-based method works well for larger block sizes.

Code generation for programs with array references involving multiple induction variables (MIV) where the arrays are block-cyclically distributed is also addressed in this thesis. A collection of runtime techniques with varying space and time overheads are presented, and their impact on execution time is clearly demonstrated through experiments.