CODE GENERATION AND OPTIMIZATION FOR HIGH PERFORMANCE FORTRAN

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
Ashwath Thirumalai
B.E. in Electronics, Bangalore University, India, 1992
August 1995

ACKNOWLEDGMENTS

First and foremost, I am grateful to my brother, Srikanth Thirumalai, for his untiring support throughout my academic career. His patience, guidance and encouragement helped make this thesis possible.

Next, I would like to thank my parents for the considerable help, advice and financial support they have provided throughout my life.

I would like to thank my advisor, Dr. J. Ramanujam, for his constant and unfailing enthusiasm, encouragement and financial support throughout this work. I will never fail to admire his ability to come up with solutions to problems, faced by us, on the spur of the moment.

I would also like to thank Dr. Subhash C. Kak and Dr. R. Vaidyanathan for being part of my committee.

This research was supported in part by the National Science Foundation grants CCR-9210422 and CCR-9457768 and by the Louisiana Board of Regents under contract LEQSF(1991-94)-RD-A-09.

TABLE OF CONTENTS

List of Tables

List of Figures

ABSTRACT

In data-parallel languages such as High Performance Fortran and Fortran D, arrays are mapped to processors through a two step process of alignment followed by distribution. This process of mapping creates disjoint pieces of the array that are locally owned by each processor. A compiler that generates code for each processor has to compute the sequence of local memory addresses accessed by each processor and the sequence of sends and receives for a given processor to access non-local data. This thesis presents a novel approach based on integer lattices for the address generation problem. The set of accesses can be expressed as integer linear combinations of basis vectors. A linear time algorithm that computes the basis vectors as a function of the mapping is presented. Using these basis vectors, a loop nest that enumerates the addresses as points in the lattice generated by the basis vectors is derived. Experimental results presented here demonstrate that this approach is superior to other solutions. In addition to this runtime technique, closed form expressions for basis vectors as functions of the data mapping are derived. This precludes the need for expensive runtime computation of basis vectors, requiring only the enumeration of addresses using a loop nest. Algorithms for deriving the communication patterns for each processor are also presented. These patterns are used in packing data to be sent. In addition, three different techniques are presented for accesses to non-local data. The first algorithm accesses the received data directly out of the buffers. The other two strategies are generalizations of overlap regions that allocate storage for the non-local data as well. The received elements are unpacked into this memory allocated. Experiments show that the communication generation times incurred by these techniques are negligible compared to message passing overheads on current machines.

CHAPTER 1

INTRODUCTION

Distributed memory multiprocessors are attractive for high performance computing in that they offer potentially high levels of flexibility, scalability and performance. However, programming these machines to realize their promised performance (which requires a full orchestration of the execution through careful partitioning of computation and data, and placement of message passing) remains a difficult task. The extreme difficulty of writing correct and efficient programs is a major obstacle to the widespread use of parallel high-performance computing. The main objective of efforts such as High Performance Fortran (HPF) [8], Fortran D [4], and Vienna Fortran [2] (which grew out of the earlier SUPERB effort) is to raise the level of programming on distributed memory machines while retaining the object code efficiency derived, for example, from message passing.

Languages such as Fortran D [4], Vienna Fortran [2] and High Performance Fortran (HPF) [8] include directives such as `ALIGN` and `DISTRIBUTE` that describe how data is distributed among processors in a distributed-memory multiprocessor. For example in HPF, arrays are mapped to processors in two steps as shown in Fig. 1.1. In the first step, arrays are *aligned* to an abstract discrete Cartesian grid called a *template;* the template is then distributed onto the processors.

The effect of this two-level mapping onto p processors is to create p disjoint pieces of the array, with each processor being able to address only data items in its locally allocated piece. Thus, an HPF compiler must generate code that executes on each processor (referred to as node code) that accesses only locally owned pieces directly, and inserts communication for non-local accesses. Here we show an example of HPF code and the resulting node code (assume 4 processors, hence `PROCS` = 4) that we need to generate:

```
        REAL A(200), B(200)
!HPF$   TEMPLATE T(200)
!HPF$   PROCESSORS PROCS(4)
!HPF$   ALIGN A WITH T
!HPF$   ALIGN B(i) WITH T(i+1)
!HPF$   DISTRIBUTE T(CYCLIC(25)) ONTO PROCS
        do i = 2, 200
             A(i) = B(i-1)
        enddo
```

Fig. 1.2 shows the data distribution for the above HPF code; the node code for this case is shown below.

```
     REAL A(200), B(200)
     my$p = myproc() {* returns 0 ··· 3 *}
```

```
do i = 2, 200
     if (my$p .EQ. owner(B(i-1))) then
          send B(i-1) to owner(A(i))
     endif
     if (my$p .EQ. owner(A(i))) then
          receive B(i-1) from owner(B(i-1))
          A(i) = B(i-1)
     endif
enddo
```

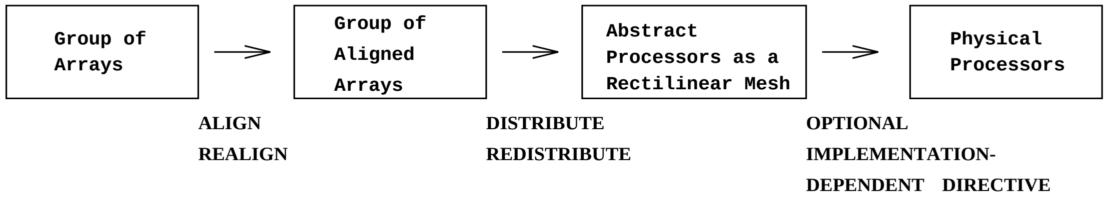

Figure 1.1: The HPF data mapping model

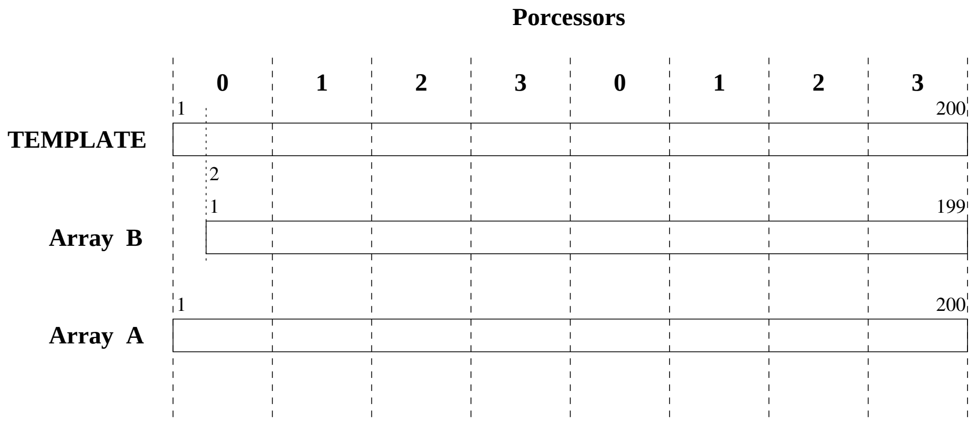

Figure 1.2: Data distribution for the sample HPF code

An HPF compiler should generate a highly optimized version of the above node code as shown below.

```
REAL A(50), B(50)
my$p = myproc() {* returns 0 ··· 3 *}
lb$1 = max(my$p*25, 2) - (my$p*25)
do i = lb$1, 50
      A(i) = B(i-1)
enddo
```

In the above code the arrays A and B are perfectly aligned and hence the compiler does not have to insert any communication commands.

In order to generate node code we need to know the sequence of local memory addresses accessed by each processor and the sequence of sends and receives for a given processor to access non-local data. A regular access pattern in terms of the global data structure can appear to be irregular within each locally allocated piece. For example, an array section $A(\ell : h : s)$ exhibits a regular access sequence of stride s; but with an HPF-style data mapping, the access sequence can become irregular. The array section $A(\ell : h : s)$ is equivalent to the set of elements $A(\ell + js)$ where j is an integer and $0 \leq j \leq \lfloor \frac{h-l}{s} \rfloor$. In this thesis we present efficient algorithms for generating local memory access patterns and interprocessor communication sets for the various processors given the alignment of arrays to a template and the distribution of the template onto the processors. Our solution is based on viewing the access sequence as an integer lattice, and involves the derivation of a suitable set of basis vectors for the lattice. Given the lattice basis, we enumerate the lattice by using loop nests; this allows us to generate efficient code that incurs negligible runtime overhead in determining the access pattern. Chatterjee *et al.* [3] presented an $O(k \log k)$ algorithm (where k is the block size – see Chapter 2 for definition) for this problem; the algorithm proposed in this thesis is an $O(k)$ algorithm. Recently, Kennedy *et al.* [10] have also presented an $O(k)$ algorithm. Our algorithm is two to three times faster than the algorithm of Kennedy *et al.* as demonstrated by experiments. Experiments on several platforms indicate the efficacy of our approach compared to that of Chatterjee *et al.* as well.

This thesis also proposes closed form expressions for generating the basis vectors of the access sequence lattice for certain problem instances. In an experiment [17] with p (the number of processors) in powers of 2 from 1 to 1024, k (block size) in powers of 2 from 1 to 16384, and s (the access stride) ranging from 1 to 99 in unit increments, we derived the optimal basis vectors for over 91% of the problem instances. For the remaining problem instances, we can use closed form expressions to generate basis vectors that are close to the best pair of basis vectors. We go on to show that we can generate correct code with these approximate basis vectors.

Using the fact that the set of elements referenced can be generated by integer linear combinations of basis vectors, we also present algorithms for deriving the communication sets of each processor. We first show that the communication pattern repeats after a certain number of accesses. Next, we present algorithms that determine the send and receive patterns based on our address generation strategy. In addition, we present an algorithm that packs the elements that need to be sent to other processors into send buffers. We then present an algorithm to access non-local elements needed for an assignment directly out of the receive buffers. Two novel techniques for unpacking data in the receive buffers based on a generalization of overlap areas [5] are presented next. In these methods we allocate local storage for non-local elements accessed by a processor and unpack the elements in the receive buffer into them.

This thesis is organized as follows. In Chapter 2, we present the problem setting and discuss related work. Chapter 3 outlines our approach using lattices and presents key mathematical results which are used later in the thesis. We then present our linear algorithm for determining basis vectors, and contrast it with the algorithm of Kennedy *et al.* [10]. We show how to determine address sequences by lattice enumeration, and explain the use of lattice basis vectors to generate a loop nest that determines the address sequence. Finally we demonstrate the efficacy of our approach using experimental results comparing our solution to those of Chatterjee *et al.* [3] and Kennedy *et al.* [10]. Chapter 4 presents efficient solutions for the compile time derivation of the best basis vectors. Chapter 5 presents algorithms for deriving communication sets for the different processors. Here we present models for packing the data to be sent and unpacking the data that needs to be received. Finally Chapter 6 concludes with a summary and points to further research directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

In HPF, arrays are mapped to processors in two steps. In the first step, arrays are aligned to an abstract discrete Cartesian grid called a template. HPF restricts alignment along each dimension to be a function of one variable only. Hence, $A(i)$ aligned with $T(aj + b)$ (a and b are integer constants) is a typical HPF alignment. This alignment statement directs the compiler to co-locate all array elements aligned to the same template cell. We consider an array A identically aligned to the template T; this means that if, $A(i)$ is aligned with $T(ai + b)$, then the alignment stride $a = 1$ and the alignment offset $b = 0$.

The template cells are then distributed among the different processors in next step of mapping arrays to processors. In HPF the different dimensions of the template can be distributed in `CYCLIC`, `BLOCK`, `CYCLIC`(k)(also known as block-cyclic [8] distribution) or not distributed at all. If a template is distributed in a block-cyclic (`CYCLIC`(k)) fashion with a block size of k across p processors *i.e.*, we allocate k elements of the array to each processor and when we run out of processors, we wrap around and allocate the next k elements to the first processor and so on. If $k = 1$, the distribution is called `CYCLIC`, and if $k = \frac{N}{p}$, where N is the size of the template, the distribution is called a `BLOCK` distribution.

We assume that arrays have a lower limit of zero, and processors and local addresses are numbered from zero onward. This mapping of the elements of A to the processor memories is shown in the Fig. 2.1(a). Though the elements of A are stored in the processor memories in a linear fashion as shown in Fig. 2.1(b), we adopt a two-dimensional view of the storage allocated for the array as shown in Fig. 2.1(a). We view the global addresses as being organized in terms of courses; each course consists of pk elements, a contiguous set of k elements being allocated per processor. In the two-dimensional view we adopt, the first dimension denotes the course number (starting from zero), and the second dimension denotes the offset from the beginning of the course.

An array assignment statement (in HPF) is of the form

$$A(\ell : h : s) = \cdots\cdots \tag{2.1}$$

where s is the access stride, and ℓ and h are the lower and upper bounds respectively. Given an array statement with HPF-style data mappings, it is our aim to generate node code along with communication for the different processors.

Consider the case of an array aligned identically to a template that is distributed `CYCLIC(4)` onto 3 processors, which is accessed with a stride of 5 ($p = 3, k = 4$ and $s = 5$). Fig. 2.2(a) shows the allocation of the array elements along with the corresponding global addresses. The array elements accessed are marked and the corresponding local addresses are shown in Fig. 2.2(a). While the global access stride is constant (5 in this case), the local access sequence does not have a constant stride on any processor. For example, the local addresses of elements accessed in processor 1 are $1, 10, 12, 19, 21, \ldots$. The

Processor 0			Processor 1			$\cdots$	Processor p-1		
0	$\cdots$	k-1	k	$\cdots$	2k-1	$\cdots$	(p-1)k	$\cdots$	pk-1
pk	$\cdots$	(p+1)k-1	(p+1)k	$\cdots$	(p+2)k-1	$\cdots$	(2p-1)k	$\cdots$	2pk-1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(a): Mapping of an array identically aligned to a template T which is cyclic(k) distributed among p processors.

Global index (Processor 0)	0	$\cdots$	k-1	pk	$\cdots$	(p+1)k-1	$\cdots$
Local index (Processor 0)	0	$\cdots$	k-1	k	$\cdots$	2k-1	$\cdots$

(b): Storage pattern of the array shown in Fig. 2.1(a) in the local memory of Processor 0.

Figure 2.1: Global and local addresses for data mappings

Table 2.1: Symbols used in this thesis.

A	a distributed array
T	the template to which array A is aligned
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
$A(\ell : h : s)$	a regular section of array A (array section)
ℓ	lower bound of regular section of A
h	upper bound of regular section of A
s	stride of regular section of A
m	processor number
Γ	array section lattice
Γ_m	part of Γ incident on processor m

Processor 0				Processor 1				Processor 2			
$\boxed{0^{0}}$	1	2	3	4	$\boxed{5^{1}}$	6	7	8	9	$\boxed{10^{2}}$	11
12	13	14	$\boxed{15^{7}}$	16	17	18	19	$\boxed{20^{4}}$	21	22	23
24	$\boxed{25^{9}}$	26	27	28	29	$\boxed{30^{10}}$	31	32	33	34	$\boxed{35^{11}}$
36	37	38	39	$\boxed{40^{12}}$	41	42	43	44	$\boxed{45^{13}}$	46	47
48	49	$\boxed{50^{18}}$	51	52	53	54	$\boxed{55^{19}}$	56	57	58	59
$\boxed{60^{20}}$	61	62	63	64	$\boxed{65^{21}}$	66	67	68	69	$\boxed{70^{22}}$	71
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a): Global and local addresses. Superscripts denote local addresses.

Processor 0				Processor 1				Processor 2			
$\boxed{0,0}$	0,1	0,2	0,3	0,4	$\boxed{0,5}$	0,6	0,7	0,8	0,9	$\boxed{0,10}$	0,11
1,0	1,1	1,2	$\boxed{1,3}$	1,4	1,5	1,6	1,7	$\boxed{1,8}$	1,9	1,10	1,11
2,0	$\boxed{2,1}$	2,2	2,3	2,4	2,5	$\boxed{2,6}$	2,7	2,8	2,9	2,10	$\boxed{2,11}$
3,0	3,1	3,2	3,3	$\boxed{3,4}$	3,5	3,6	3,7	3,8	$\boxed{3,9}$	3,10	3,11
4,0	4,1	$\boxed{4,2}$	4,3	4,4	4,5	4,6	$\boxed{4,7}$	4,8	4,9	4,10	4,11
$\boxed{5,0}$	5,1	5,2	5,3	5,4	$\boxed{5,5}$	5,6	5,7	5,8	5,9	$\boxed{5,10}$	5,11
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(b): 2-dimensional view of global addresses.

Processor 0				Processor 1				Processor 2			
$\boxed{0,0}$	0,1	0,2	0,3	0,0	$\boxed{0,1}$	0,2	0,3	0,0	0,1	$\boxed{0,2}$	0,3
1,0	1,1	1,2	$\boxed{1,3}$	1,0	1,1	1,2	1,3	$\boxed{1,0}$	1,1	1,2	1,3
2,0	$\boxed{2,1}$	2,2	2,3	2,0	2,1	$\boxed{2,2}$	2,3	2,0	2,1	2,2	$\boxed{2,3}$
3,0	3,1	3,2	3,3	$\boxed{3,0}$	3,1	3,2	3,3	3,0	$\boxed{3,1}$	3,2	3,3
4,0	4,1	$\boxed{4,2}$	4,3	4,0	4,1	4,2	$\boxed{4,3}$	4,0	4,1	4,2	4,3
$\boxed{5,0}$	5,1	5,2	5,3	5,0	$\boxed{5,1}$	5,2	5,3	5,0	5,1	$\boxed{5,2}$	5,3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(c): 2-dimensional view of local addresses.

Figure 2.2: Global and local addresses of the accessed elements of array A along with the 2-dimensional view for the case $p = 3$, $k = 4$ and $s = 5$.

address generation problem is to enumerate this sequence; the communication generation problem is to derive the required communication given the access sequence. Fig. 2.2(b) shows the global addresses of array A using the two-dimensional view that we adopt. Each element that is accessed is written as a two-component vector. In the global view, the first component of the vector is the course number to which the corresponding array element belongs; the second component gives the offset of the element from the start of the course. In the local view, the first component of the vector is the course number to which that array element belongs; the second component gives the offset of the element from the start of the course on the processor to which the element belongs. Fig. 2.2(c) shows the two-dimensional view using local addresses.

Several papers have addressed this code generation problem. Koelbel [12] derived techniques for compile-time address and communication generation for `BLOCK` and `CYCLIC` distribution for non-unit stride accesses. Chatterjee *et al.* [3] derived a technique that identifies a repeating access pattern characterized as a finite-state machine. To locate the start element of the *FSM* on a given processor m, they had to find the smallest element of $A(\ell : h : s)$ that belonged to m. An element $A(i)$ belongs to a processor m if its offset in the course to which it belongs lies in the range $[km, k(m+1)-1]$. The offset of $A(i)$ in the course to which it belongs is $(i \bmod pk)$. Every element that belongs to $A(\ell : h : s)$ can be written as $(l + sj)$, where j is a non-negative integer and is lesser than $\left\lfloor \frac{h-l}{s} \right\rfloor$. Hence

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

Chatterjee *et al.* [3] show the above inequality to be equivalent to solving a set of k linear Diophantine equations

$$\{sj - pkq = \delta \mid km - l \leq \delta \leq km - l + k - 1\}$$

where q is a non-negative integer. The above equations have a solution if $\text{GCD}(s, pk)$ divides δ. They find the smallest solution of each of the solvable Diophantine equations. By sorting these solutions they obtain the the access pattern and by obtaining the minimum of these solutions they get the start of the *FSM*. Their $O(k \log\ k)$ algorithm involves a solution of k linear Diophantine equations to determine the pattern of addresses accessed, followed by sorting these addresses to derive the accesses in linear order. In this thesis, we show that sorting the elements is not required.

In an exhaustive study of this problem Stichnoth [16] presented a framework to enumerate local addresses and generation of communication sets. Gupta *et al.* [7] derived the *virtual-block* and the *virtual-cyclic* schemes. The virtual block (cyclic) scheme views the global array as a union of several cyclically (block) distributed arrays. The virtual cyclic scheme does not preserve the access order in the case of `DO` loops (this is not a problem for array assignments). Ancourt *et al.* [1] use a linear algebra framework to generate code for fully parallel loops in HPF, which does not exploit repeating sequence of accesses. Recently, building on the finite-state machine approach of Chatterjee *et al.*, Kennedy *et al.* [10] derived a runtime solution that determines the basis vectors by solving

$2k-1$ linear Diophantine equations. The improvement over [3] comes from avoiding the sorting step at the expense of solving an additional set of $k-1$ linear Diophantine equations. In this thesis we show that solving $2k-1$ equations is not required. Our algorithm requires the solution of only k linear Diophantine equations. In addition, our lattice enumeration algorithm (given basis vectors) is different from and faster than [10].

Chatterjee *et al.* [3] observed that the communication pattern repeats after pk elements are accessed in the global address space and hence could be modeled as a finite state machine. While their algorithm requires only a single scan of all the right hand side elements, it involves expensive conversions form global address space to the local address space. Gupta *et al.* [7] and Stichnoth [16] proposed communication generation schemes that exploit message vectorization. Both their algorithms require p scans of all the right hand side elements. Kennedy *et al.* [11] improved the communication algorithms presented by Chatterjee *et al.* and presented an algorithm based on overflow regions to unpack received data into local storage areas. As the non-local elements of arrays are packed at the end of the arrays, packing into overflow regions is not cache-friendly. In this thesis we discuss an algorithm for accessing non-local elements directly out of receive buffers. This completely eliminates the unpacking overhead. We also present two algorithms based on a generalization of overlap areas to unpack non-local data into local storage areas. The unpacking algorithms presented in this thesis maintain data locality and hence perform better than other solutions to this problem.

CHAPTER 3

A LATTICE BASED APPROACH FOR ADDRESS GENERATION

In this chapter we present a novel technique based on integer lattices for the address generation problem presented in the previous chapter. We first show that the accessed array elements of an array section belong to an integer lattice. We then provide a linear time algorithm to obtain the basis vectors of the integer lattice. We then go on to use these basis vectors to generate a loop nest that determines the access sequence. The problem of basis determination forms the core of code generation for HPF array statements; thus, a fast solution for this problem improves the performance of several facets of an HPF compiler. In this chapter we also provide a few optimizations of our basis determination algorithm.

3.1 ASSUMPTIONS

We present our approach to address generation for an alignment stride, $a = 1$. For $a > 1$, we use an approach similar to the one developed in [3], which involves two applications of our algorithm. Therefore, we do not discuss non-unit alignment strides. Communication sets for array statements can be derived by two applications of our algorithm for address generation [17].

We assume that A is identically aligned to the template T. As it is evident from Fig. 2.1(a), we assign a block of k cells of the template to each of the p processors and then wrap around and assign the rest of the cells in a similar fashion. As mentioned earlier, we treat the global address space as a two dimensional space and every element of an array $A(i)$ has an address of the form $(x, y) = (i \operatorname{div} pk, i \bmod pk)$ in this space. Here x gives the course number to which this element belongs and y gives the offset of the element in that course. We refer to the two-dimensional view of an array A as A_{2D}; this notation is used throughout this thesis.

3.2 LATTICES

We use the following definitions of a lattice and its basis [6].

Definition 3.1 *A set of points $\vec{x_1}, \vec{x_2}, \cdots, \vec{x_k}$ in $\mathcal{R}^n$ is said to be* independent *if these points do not belong to a linear subspace of $\mathcal{R}^n$ of dimension less than k.*

For $k = n$, this is equivalent to the condition that the determinant of the matrix X whose columns are $\vec{x_i}, i = 1, \ldots, n$ is non-zero, *i.e.,* $det\left([\vec{x_1}\vec{x_2}\cdots\vec{x_n}]\right) \neq 0$. Now we state the following definition from the theory of lattices [6]; see [6] for a proof.

Definition 3.2 *Let $\vec{b_1}, \vec{b_2}, \cdots, \vec{b_n}$ be n independent points. Then the set Λ of points $\vec{q}$ such that*

$$\vec{q} = u_1\vec{b_1} + u_2\vec{b_2} + \cdots + u_n\vec{b_n}$$

(where $u_1, \ldots, u_n$ are integers) is called a lattice. The set of vectors $\vec{b_1}, \cdots, \vec{b_n}$ is called a basis of Λ. A matrix $B = \vec{b_1}\vec{b_2}\cdots\vec{b_n}$ is called a basis matrix.

Definition 3.3 *Let Λ be a discrete subspace of $\mathcal{R}^n$ which is not contained in an $(n-1)$-dimensional linear subspace of $\mathcal{R}^n$. Then Λ is a lattice.*

We refer to the set of global addresses (elements) over all processors accessed by $A(\ell\!:\!h\!:\!s)$ with distribution parameters (p, k) as $\Gamma = \langle A(\ell\!:\!h\!:\!s), p, k\rangle$. We refer to the set of local addresses accessed by processor m in executing its portion of $A(\ell\!:\!h\!:\!s)$ with distribution parameters (p, k) as Γ_m. By construction, Γ is a discrete subgroup of $\mathcal{R}^2$ and in general, is not contained in a 1-dimensional linear subspace of $\mathcal{R}^2$; if a single vector can be used to generate the elements accessed, our algorithm handles this as a special case. Thus, without loss of generality, $\Gamma = \langle A(\ell\!:\!h\!:\!s), p, k\rangle$ is a lattice. Similarly Γ_m is also a lattice.

In order to find the sequence of local addresses accessed on processor m, one needs to:

1. find a set of basis vectors of the lattice Γ_m; and

2. enumerate the points in Γ_m using integer linear combinations of the basis vectors.

Our solution to Step 1 (presented in Section 3.3) uses the fact that a basis for the lattice can be computed from a knowledge of some of the points in the lattice. Our solution to Step 2 (presented in Section 3.6) uses the fact that if every point in the lattice can be generated as non-negative integer linear combinations of a suitable pair of basis vectors, then these points can be enumerated by a two-level loop (each level with a step size of 1); in addition, this two-level nest can be derived by applying the linear transformation B^{-1} where B is the basis of the lattice.

Definition 3.4 *A basis B of the lattice Λ is called an* extremal basis *if the set of points $\vec{q}$ that belong to Λ can be written as*

$$\vec{q} = u_1\vec{b_1} + u_2\vec{b_2} + \cdots + u_n\vec{b_n}$$

where $u_1, \ldots, u_n$ are non-negative integers.

This thesis presents an algorithm for determining an extremal basis of the array section lattice $\langle A(\ell\!:\!h\!:\!s), p, k\rangle$ and shows how to use the extremal basis to generate the address sequence efficiently.

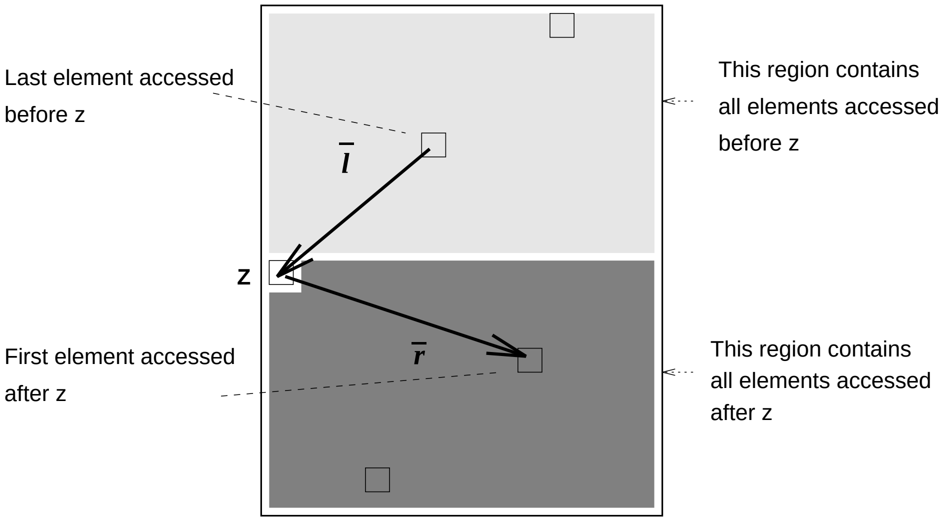

Figure 3.1: Explanation of basis determination

3.3 DETERMINATION OF BASIS VECTORS

In this section, we show how to derive a pair of extremal basis vectors for the lattice Γ_m. In order to do that, we state a key result that allows us to find a basis for the lattice given a set of points in the lattice.

Result 3.1 *Let $\vec{b_1}, \vec{b_2}, \cdots, \vec{b_n}$ be independent points of a lattice Λ in $\mathcal{R}^n$. Then Λ has a basis $\{\vec{a_1}, \vec{a_2}, \cdots, \vec{a_n}\}$ such that*

$$\vec{b_i} = \sum_{k=1}^{i} u_{ki}\vec{a_k} \qquad\qquad (i = 1, \ldots, n)$$

where $u_{ii} > 0$ and $0 \leq u_{ki} < u_{ii}$ $(k < i; i = 1, \ldots, n)$. In addition, the set of points $\left\{\vec{b_1}, \vec{b_2}, \cdots, \vec{b_n}\right\}$ is a basis of the lattice Λ if and only if $u_{ii} = 1$.

While this result allows us to decide if a given set of independent points form a basis of the lattice, it is not constructive. But for $n = 2$, we derive the following theorem which allows us to construct a basis for the array section lattice on processor m. We use the vector $\vec{o}$ to refer to the origin of the lattice.

Theorem 3.1 *Let $\vec{b_1}$ and $\vec{b_2}$ be independent points of a lattice Λ in $\mathcal{R}^2$ such that the closed triangle formed by the vertices $\vec{o}$, $\vec{b_1}$ and $\vec{b_2}$ contains no other points of Λ. Then $\left\{\vec{b_1}, \vec{b_2}\right\}$ is a basis of Λ.*

Proof: Let $\{\vec{a_1}, \vec{a_2}\}$ be any basis of Λ. From Result 3.1, we can write

$$\begin{aligned} \vec{b_1} &= u_{11}\vec{a_1} \\ \vec{b_2} &= u_{12}\vec{a_1} + u_{22}\vec{a_2} \end{aligned}$$

where $u_{11} > 0, u_{22} > 0$ and $0 \leq u_{12} < u_{22}$. Based on the hypothesis, the side of the triangle connecting vertices $\vec{o}$ and $\vec{b_1}$ does not contain other points of Λ. Therefore, $u_{11} = 1$.

Let us assume that $u_{22} > 1$. If $u_{12} = 0$, the triangle formed by $\vec{o}$, $\vec{b_1}$ and $\vec{b_2}$ contains the point $\vec{a_2} \in \Lambda$; similarly, if $u_{12} \geq 1$, the triangle formed by $\vec{o}$, $\vec{b_1}$ and $\vec{b_2}$ contains the point $\vec{a_1} + \vec{a_2} \in \Lambda$. This contradicts our hypothesis. Hence, $u_{22} = 1$. Since, $u_{11} = u_{22} = 1$, it follows from Result 1 that the vectors $\vec{b_1}$ and $\vec{b_2}$ form a basis of Λ. □

Thus, in order to determine a basis of the array section lattice on processor m, we need to find three points (one of which can be considered as the origin without loss of generality) not on a straight line such that the triangle formed by them contains no other points belonging to the lattice. Let $\vec{x_1}$, $\vec{x_2}$, and $\vec{x_3}$ be three consecutively accessed elements of the array section lattice on processor m. If $\vec{x_1}$, $\vec{x_2}$, and $\vec{x_3}$ are independent points (do not lie on a straight line), then the vectors $\vec{x_2} - \vec{x_1}$ and $\vec{x_3} - \vec{x_2}$ form a basis for Γ_m. Recall that we view the array layout as consisting of several courses on each processor with each course consisting of k elements; this allows us to refer to each of the k columns on a processor. For the array section $A(\ell : h : s)$, let c_f be the first column in which an element is accessed and let c_l be the last column in which an element is accessed. Let z_f be some element accessed in column c_f and let z_l be some element accessed in column c_l by a processor. Let x^{prev} denote the element accessed immediately before x in lexicographic order on a processor, and x^{next} denote the element accessed immediately after x in linear order on a processor. We do not discuss the case where z_f and z_f^{next} (or z_f^{prev}) are in the same column. This case is handled separately and is easily detected by our technique.

Theorem 3.2 *The set of points $\{z_f^{prev}, z_f, z_f^{next}\}$ generate a basis of Γ_m.*

Proof: From Theorem 3.1, the set of points $\{z_f^{prev}, z_f, z_f^{next}\}$ generate a basis of Γ_m if there are no lattice points in the triangle enclosing them and if they are independent. By construction, these are consecutive points in the lattice Γ_m and therefore, there no lattice points in the triangle (if any) enclosing them. Suppose these are not independent, *i.e.,* they lie on a straight line. This implies one of the following two cases:

Case (a): Column(z_f^{prev}) < Column(z_f) < Column(z_f^{next}).

Case (b): Column(z_f^{next}) < Column(z_f) < Column(z_f^{prev}).

Neither of these cases hold, since Column$(z_f) = c_f$ is the first column on processor m in which any element is accessed. Therefore, the three points are independent. Hence the result. □

Similarly, the set of points $\{z_l^{prev}, z_l, z_l^{next}\}$ also generate a basis of Γ_m. We use the set $\{z_f^{prev}, z_f, z_f^{next}\}$. We refer to the vector $\vec{z_f} - \vec{z_f^{prev}}$ as $\vec{l} = (l_1, l_2)$ and the vector $\vec{z_f^{next}} - \vec{z}$ as $\vec{r} = (r_1, r_2)$. Again by construction, $l_1 > 0$, $r_2 > 0$, $l_2 < 0$ and $r_1 \geq 0$. This is illustrated in Fig. 3.1 on p. 13.

Our algorithm shown in Fig. 3.2 uses these ideas to derive a basis for the lattice Γ_m. Lines 4-12 compute the smallest global address accessed in each of the k columns on processor m. This is done by solving a linear Diophantine equation for each column, in exactly the same way as suggested by Chatterjee *et al.*; see [3] for details on the derivation (a brief outline of this is provided in Chapter 2). We use the fact that the access pattern repeats after every $\frac{s}{\text{GCD}(s,pk)}$ courses. Note that z_f^{next} is the smallest address larger than z_f (if any) and z_f^{prev} is the largest address smaller than z_f (if any). Depending on whether z_f is the first, intermediate, or the last element accessed in the set of $\frac{s}{\text{GCD}(s,pk)}$ courses, our algorithm computes the values z_f^{prev}, z_f, and z_f^{next} (lines 16–28 in Fig. 3.2). Line 1 (extended Euclid) is $O(\log \min(s, pk))$. Lines 4–12 require $O(k)$ time. Lines 13–28 need $O(1)$ time. Since any solution to the address generation problem requires one GCD calculation, we conclude that our algorithm runs in $O(k)$.

3.4 Extremal Basis Vectors

In this section, we show that the basis vectors generated by our algorithm in Fig. 3.2 form an extremal set of basis vectors.

Input: Layout parameters (p, k), regular section $(\ell : h : s)$, processor number m.
Output: The *start* memory location, *end* memory location, length, and the basis vectors $\vec{r} = (r_1, r_2)$ and $\vec{l} = (l_1, l_2)$ for processor m.
Method:

$(d, x, y) \leftarrow$ EXTENDED-EUCLID(s, pk)
length $\leftarrow$ 0; start $\leftarrow h + 1$; $amax \leftarrow -1$; $bmin \leftarrow \frac{pks}{d}$; $\lambda \leftarrow d\lceil\frac{km-\ell}{d}\rceil$
$amin \leftarrow bmax \leftarrow$ z $\leftarrow \ell + \frac{s}{d}(\lambda x + pk\lceil\frac{-\lambda x}{pk}\rceil)$
for $\lambda = d\lceil\frac{km-\ell}{d}\rceil + d$, $km - \ell + k - 1$, d **do**
  loc $\leftarrow \ell + \frac{s}{d}(\lambda x + pk\lceil\frac{-\lambda x}{pk}\rceil)$
  if loc $<$ z **then** /* loc is accessed before z */
    $amax \leftarrow$ max($amax$,loc); $amin \leftarrow$ min($amin$,loc)
  else /* loc is accessed after z */
    $bmax \leftarrow$ max($bmax$,loc); $bmin \leftarrow$ min($bmin$,loc)
  endif
  length $\leftarrow$ length $+ 1$
enddo
if length = 0 **then** /* No work for processor */
  return $\perp$, $\perp$, 0, $\perp$, $\perp$, $\perp$, $\perp$
endif
if length = 1 **then** /* No work for processor */
  return z, z, 1, $\perp$, $\perp$, $\perp$, $\perp$
endif
if $amin =$ z **and** $amax = -1$ **then** /* above is empty */
  $l_1 \leftarrow \lfloor\frac{z}{pk}\rfloor + \frac{s}{d} - \lfloor\frac{bmax}{pk}\rfloor$; $l_2 \leftarrow z$ mod $k - bmax$ mod k
  $r_1 \leftarrow \lfloor\frac{bmin}{pk}\rfloor - \lfloor\frac{z}{pk}\rfloor$; $r_2 \leftarrow bmin$ mod $k - z$ mod k
else if $bmin = \frac{pks}{d}$ **and** $bmax =$ **z then** /* below is empty */
  $l_1 \leftarrow \lfloor\frac{z}{pk}\rfloor - \lfloor\frac{amax}{pk}\rfloor$; $l_2 \leftarrow z$ mod $k - amax$ mod k
  $r_1 \leftarrow \lfloor\frac{amin}{pk}\rfloor + \frac{s}{d} - \lfloor\frac{z}{pk}\rfloor$; $r_2 \leftarrow amin$ mod $k - z$ mod k
else /* above & below not empty */
  $l_1 \leftarrow \lfloor\frac{z}{pk}\rfloor - \lfloor\frac{amax}{pk}\rfloor$; $l_2 \leftarrow z$ mod $k - amax$ mod k
  $r_1 \leftarrow \lfloor\frac{bmin}{pk}\rfloor - \lfloor\frac{z}{pk}\rfloor$; $r_2 \leftarrow bmin$ mod $k - z$ mod k
endif
start $\leftarrow amin$
end $\leftarrow bmax$
return start, end, length, l_1, l_2, r_1, r_2

Figure 3.2: Algorithm for determining basis vectors

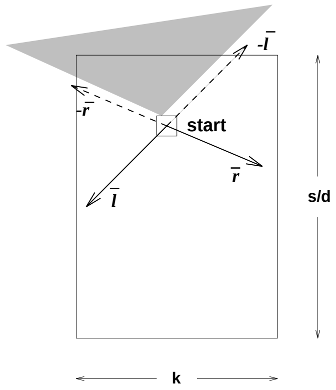

Figure 3.3: No lattice point is a non-positive linear combinations of basis vectors

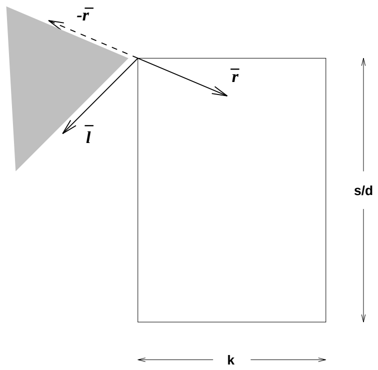

Figure 3.4: No lattice points in region spanned by vectors $\vec{l}$ and $-\vec{r}$

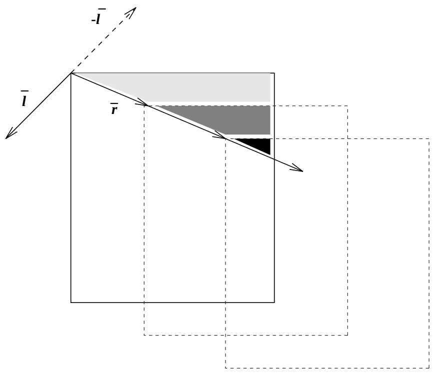

Figure 3.5: No lattice points in region spanned by linear combination with one negative coefficient

Theorem 3.3 *The lattice Γ_m (the projection of the array section lattice on processor m) contains only those points which are non-negative integer linear combinations of the basis vectors $\vec{l}$ and $\vec{r}$.*

Proof: Let $\vec{z}$ be the starting point of the lattice Γ_m. As $\vec{r}$ and $\vec{l}$ are the basis vectors of the lattice, any point $\vec{q}$ belonging to the lattice can be written as

$$\vec{q} = \vec{z} + v_1\vec{l} + v_2\vec{r}$$

where $\vec{r} = (r_1, r_2)$ and $\vec{l} = (l_1, l_2)$. Also, $l_1 > 0, l_2 < 0, r_1 \geq 0$ and $r_2 > 0$ by construction. Suppose $\vec{q} \in \Gamma_m$ and that either one or both of v_1 and v_2 are negative integers. There are two cases to consider:

Case (1): ($v_1 < 0$ and $v_2 < 0$) $v_1 l_1 + v_2 r_1 < 0$
As both v_1 and v_2 are negative, it is clear from Fig. 3.3 that $\vec{q}$ lies in the region above the start element $\vec{z}$. This contradicts our earlier assumption that $\vec{z}$ is the start element on processor m. Hence, $v_1 < 0$ and $v_2 < 0$ cannot be true.

Case (2): ($v_1 < 0$ or $v_2 < 0$)
Without loss of generality we assume that the start element $\vec{z}$ on a processor lies in the first non-empty column.

Let $v_1 \geq 0$ and $v_2 < 0$. As shown in Fig. 3.4, $\vec{q}$ lies in the region to the left of $\vec{z}$ since $v_1 l_2 + v_2 r_2 < 0$. This contradicts our assumption that $\vec{q} \in \Gamma_m$. Hence $v_1 \geq 0$ and $v_2 < 0$ cannot be true.

If $v_1 < 0$ and $v_2 \geq 0$, then the next element accessed after the origin is either $\vec{z} + \vec{r}$ or $\vec{z} + \vec{r} - \vec{l}$. If the next accessed element of Γ_m is $\vec{z} + \vec{r} - \vec{l}$, then this point should be located on a course above $\vec{z}$ or on a course below $\vec{z}$. If this element is located on a course above $\vec{z}$, it would not be a point in the lattice Γ_m. If this element is located on a course below $\vec{z}$, then this element is lexicographically closer to $\vec{z}$ than the point $\vec{z} + \vec{r}$ which is impossible (due to the construction of $\vec{r}$). By the above arguments (as can be seen in Fig. 3.5), we have shown that the next element accessed after $\vec{z}$ can only be $\vec{z} + \vec{r}$. A repeated application of the above argument rules out the presence of a lattice point in the shaded regions in Fig. 3.5. If $\vec{z}$ is not in the first accessed column on processor m, similar reasoning can be used for the vector $\vec{l}$. Hence, the result. □

3.5 Improved Basis Determination Algorithm

In Section 3.3 we showed that the starting element on each processor could be obtained by solving at most k linear Diophantine equations. We notice that the smallest solution of each of the equations gives the smallest array element accessed in the corresponding column on a processor. We also proved that to determine a basis of the array section lattice, we need to find three points not on a straight line such that the triangle formed by them contains no other points. Using this we show that we can obtain a basis for an array section lattice which generates the smallest element in each column that belongs to Γ_m by solving only two linear Diophantine equations.

The first two consecutive points accessed on a given column and the first point accessed on the next solvable column form a triangle that contains no other points. Again, let c_f be the first column in which an element is accessed on a processor. Let $\vec{z_f}$ be the first element accessed in column c_f. Since the access pattern on a given processor repeats after $\frac{pks}{\text{GCD}\,(s,pk)}$ elements, the point accessed immediately after $\vec{z_f}$ in column c_f is $\vec{z_f} + \frac{pks}{\text{GCD}\,(s,pk)}$. Now if c_s is the second column in which an element is accessed on the processor, and $\vec{z_s}$ is the first element accessed in it, then without loss of generality$\vec{z_f}$, $\vec{z_f} + \frac{pks}{\text{GCD}\,(s,pk)}$ and $\vec{z_s}$ form a basis for the array section lattice. Hence the vectors, $\left(\frac{s}{d}, 0\right)$ and $\vec{z_s} - \vec{z_f}$ form a pair of basis vectors of the array section lattice

The elements $\vec{z_f}$ and $\vec{z_s}$ can be obtained by solving the first two solvable Diophantine equations in the algorithm explained in Fig. 3.2. Fig. 3.6 shows the basis vectors generated as explained above whereas Fig. 3.7 gives an outline of how the new basis vectors could be used to access the smallest array element accessed in each column for the case where $\vec{z_s}$ lies on a course above or below $\vec{z_f}$. Our basis determination algorithm works as follows. First we use the new basis to walk through the lattice to enumerate all the points on the lattice before the pattern starts to repeat. Then we use these points to locate three independent points that form a triangle that contains no other lattice point. Using these three points we obtain the new basis of the lattice which we use to walk through the lattice in lexicographic order. With this change to our previous algorithm, we now need to solve only two Diophantine equations to generate the basis of the array section lattice that enumerates the points accessed in lexicographic order, unlike the $2k - 1$ equations that Kennedy *et al.* [10] need. This new basis determination algorithm performs substantially better than that proposed by Kennedy *et al.* for higher values of k.

3.6 Address Sequence Generation by Lattice Enumeration

As mentioned earlier, we will treat the global address space as a two dimensional space, and each element of an array $A(i)$ has an address of the form $(x, y) = (i\text{div } pk, i \text{ mod } pk)$ in this space. We refer to the two-dimensional view of an array A as A_{2D}; this notation is used throughout this thesis. The sequence of the array elements accessed (course by course) in a processor can be obtained by strip mining the loop corresponding to $A(\ell : h : s)$ with a strip length of pk and appropriately restricting the inner loop limits. In the following analysis we assume that $\ell = 0$ and $h = N - 1$. At the end of this section we will show how the code generated for $A(0 : N - 1 : s)$ can be used to generate the code for $A(\ell : h : s)$. The code for the HPF array section $A(0 : N - 1 : s)$ that iterates over all the points in the two dimensional space shown in Fig. 2.1(a) could be written as follows:

do $i = 0, \lfloor \frac{N-1}{pk} \rfloor$
    do $j = 0, (pk - 1)$

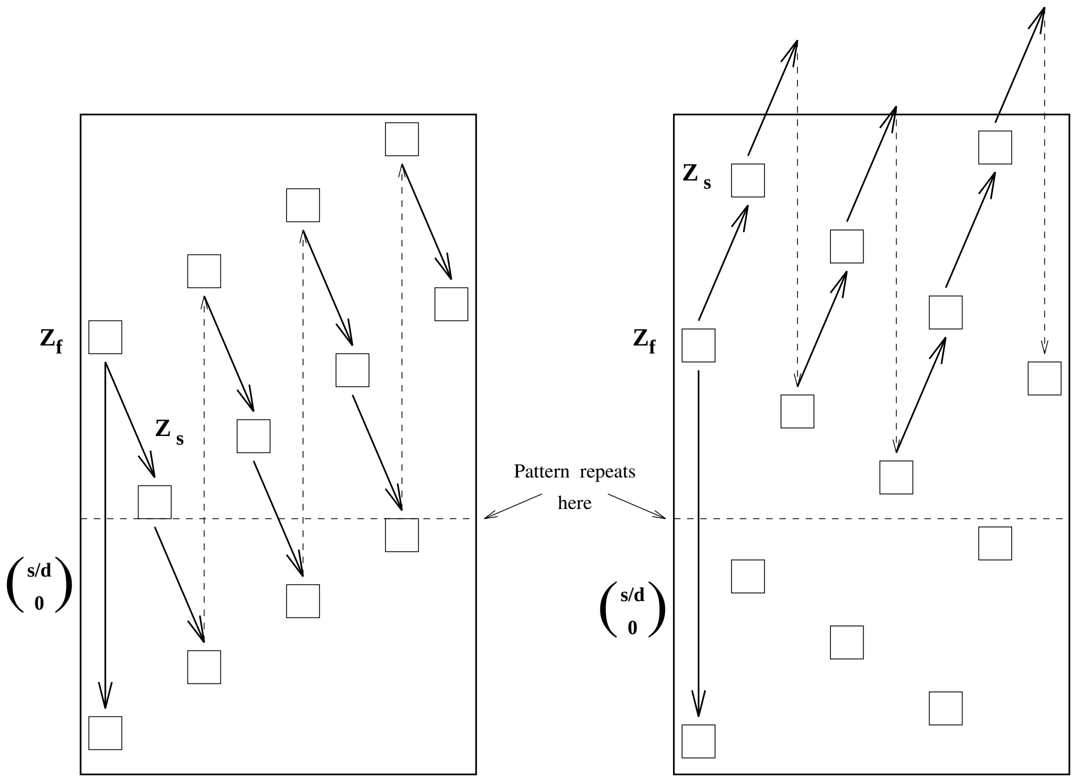

Figure 3.6: Start points in all the columns spanned using $\left(\frac{s}{d}, 0\right)$ and $\vec{z_s} - \vec{z_f}$ as the basis vectors.

Input: Layout parameters (p, k), regular section $(\ell : h : s)$, processor number m.
Output: $start$ address, end address, length, basis vectors $\vec{r} = (r_1, r_2)$ and $\vec{l} = (l_1, l_2)$ for m.
Method:

$(d, x, y) \leftarrow$ EXTENDED-EUCLID(s, pk); $length \leftarrow 2$; $start \leftarrow h + 1$
$amax \leftarrow -1$; $bmin \leftarrow \frac{pks}{d}$; $i \leftarrow d\lceil \frac{km-\ell}{d} \rceil$; $i\_end \leftarrow km - \ell + k - 1$
if $i > i\_end$ **then** /* No work for processor */
  return $\perp, \perp, 0, \perp, \perp, \perp, \perp$
endif
$amin \leftarrow bmax \leftarrow z_f \leftarrow \ell + \frac{s}{d}(ix + pk\lceil \frac{-ix}{pk} \rceil)$; $i+ = d$
if $i > i\_end$ **then** /* Only one element on processor */
  return $z_f, z_f, 1, \perp, \perp, \perp, \perp$
endif
$z_s \leftarrow \ell + \frac{s}{d}(ix + pk\lceil \frac{-ix}{pk} \rceil)$
if $z_s < z_f$ **then**
  $amin \leftarrow amax \leftarrow z_s$; $vec2 \leftarrow z_s - z_f$; $vec1 \leftarrow \frac{pks}{d} + vec2$
else
  $bmin \leftarrow bmax \leftarrow z_s$; $vec1 \leftarrow z_s - z_f$; $vec2 \leftarrow vec2 - \frac{pks}{d}$
endif
if $vec1 <= vec2$ **then**
  $loc \leftarrow loc + vec1$; $i \leftarrow i + d$
  while $i <= i\_end$ **do**
    if $loc > last$ **then**
      $loc \leftarrow loc - \frac{pks}{d}$
    endif
    if $loc < z$ **then** /* loc is accessed before z */
      $amax \leftarrow \max(amax, loc)$; $amin \leftarrow \min(amin, loc)$
    else /* loc is accessed after z */
      $bmax \leftarrow \max(bmax, loc)$; $bmin \leftarrow \min(bmin, loc)$
    endif
    $loc \leftarrow loc + vec1$; $i \leftarrow i + d$; $length \leftarrow length + 1$
  enddo
else
  $loc \leftarrow loc + vec2$; $i \leftarrow i + d$
  while $i <= i\_end$ **do**
    if $loc < first$ **then**
      $loc \leftarrow loc + \frac{pks}{d}$
    endif
    if $loc < z$ **then** /* loc is accessed before z */
      $amax \leftarrow \max(amax, loc)$; $amin \leftarrow \min(amin, loc)$
    else /* loc is accessed after z */
      $bmax \leftarrow \max(bmax, loc)$; $bmin \leftarrow \min(bmin, loc)$
    endif
    $loc \leftarrow loc + vec2$; $i \leftarrow i + d$; $length \leftarrow length + 1$
  enddo
endif
Lines 13 thru 27 of Algorithm in Fig. 3.2
return $start, end, length - 1, l_1, l_2, r_1, r_2$

Figure 3.7: Improved Algorithm for determining basis vectors

$A_{2D}(i,j) = \cdots\cdots$
end do
end do

We apply a loop transformation to the above loop nest to obtain the points of the lattice. Since the access pattern repeats after the first $\left(\frac{s}{d}\right)$ courses, we limit the outer loop in the above loop nest to iterate over the first $\left(\frac{s}{d}\right)$ courses only. In this case the global address of the first element allocated to the processor memory is mk, where m is the processor number. So in order to obtain the sequence of local addresses on processor m, we need to apply the loop transformation to the following modified code:

do $i = 0, \left(\frac{s}{d} - 1\right)$
do $j = mk, (k-1) + mk$
$A_{2D}[i,j] = \cdots$
end do
end do

The basis matrix for the lattice as derived in the last section is $B = \begin{bmatrix} l_1 & r_1 \\ l_2 & r_2 \end{bmatrix}$. Hence the transformation matrix T is of the form

$$T = B^{-1} = \frac{1}{\Delta}\begin{bmatrix} r_2 & -r_1 \\ -l_2 & l_1 \end{bmatrix},$$

where $\Delta = l_1 r_2 - l_2 r_1 = s$ (since $l_1 > 0, r_1 \geq 0, l_2 \leq 0$, and $r_2 > 0$). The loop bounds can be written as follows:

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} 0 \\ \frac{s}{d} - 1 \\ -mk \\ (k-1) + mk \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} BB^{-1}\begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} 0 \\ \frac{s}{d} - 1 \\ -mk \\ (k-1) + mk \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} l_1 & r_1 \\ l_2 & r_2 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} \leq \begin{bmatrix} 0 \\ \frac{s}{d} - 1 \\ -mk \\ (k-1) + mk \end{bmatrix}$$

$$\begin{bmatrix} -l_1 & -r_1 \\ l_1 & r_1 \\ -l_2 & -r_2 \\ l_2 & r_2 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} \leq \begin{bmatrix} 0 \\ \frac{s}{d} - 1 \\ -mk \\ (k-1) + mk \end{bmatrix}$$

where $\begin{bmatrix} u \\ v \end{bmatrix} = B^{-1} \begin{bmatrix} i \\ j \end{bmatrix}$ and u and v are integers. Therefore,

$$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} l_1 u + r_1 v \\ l_2 u + r_2 v \end{bmatrix}.$$

We now use Fourier-Motzkin elimination [22] on the following system of inequalities to solve for integral u and v:

$$\begin{aligned} -l_1 u - r_1 v &\leq 0 \\ l_1 u + r_1 v &\leq \frac{s}{d} - 1 \\ -l_2 u - r_2 v &\leq -mk \\ l_2 u + r_2 v &\leq (k-1) + mk \end{aligned}$$

If $r_1 > 0$ we have the following inequalities for u and v:

$$\frac{(-mk - k + 1)r_1}{s} \leq u \leq \frac{(\frac{s}{d} - 1)r_2 - mkr_1}{s}$$

$$\max\left(\frac{mk - ul_2}{r_2}, \frac{-ul_1}{r_1}\right) \leq v \leq \min\left(\frac{mk + k - 1 - ul_2}{r_2}, \frac{\frac{s}{d} - 1 - ul_1}{r_1}\right)$$

The node code for processor m if $r_1 > 0$ is:

do $u = \left\lceil \frac{(-mk-k+1)r_1}{s} \right\rceil, \left\lfloor \frac{(\frac{s}{d}-1)r_2 - mkr_1}{s} \right\rfloor$
  do $v = \left\lceil \max\left(\frac{mk-ul_2}{r_2}, \frac{-ul_1}{r_1}\right) \right\rceil, \left\lfloor \min\left(\frac{mk+k-1-ul_2}{r_2}, \frac{\frac{s}{d}-1-ul_1}{r_1}\right) \right\rfloor$
    $A_{2D}[l_1 u + r_1 v, l_2 u + r_2 v] = \cdots\cdots$
  end do
end do

If $r_1 = 0$ we have the following inequalities for u and v:

$$0 \leq u \leq \frac{(\frac{s}{d} - 1)r_2}{s}$$

$$\frac{mk - ul_2}{r_2} \leq v \leq \frac{mk + k - 1 - ul_2}{r_2}$$

The node code for processor m if $r_1 = 0$ is:

do $u = 0, \frac{(\frac{s}{d}-1)r_2}{s}$
  do $v = \left\lceil \frac{mk-ul_2}{r_2} \right\rceil, \left\lfloor \frac{mk+k-1-ul_2}{r_2} \right\rfloor$
    $A_{2D}[l_1 u, l_2 u + r_2 v] = \cdots\cdots$
  end do
end do

Example 3.1 Code generated for the case where $\ell = 0$, $p = 3$, $k = 4$ and $s = 11$ for processor 1.

Running through lines 1-12 of the algorithm shown in the Fig. 3.2 we notice that the set of smallest solutions for all the Diophantine equations is $\{88, 77, 66, 55\}$. Also $z = 88$, $amin = 55$, $amax = 77$, $bmin = 132$ and $bmax = 88$. Since the *below* section is empty we execute lines 20-23 of the algorithm in Fig. 3.2. So our algorithm returns, $\vec{l} = (1, -1)$ and $\vec{r} = (8, 3)$. The access pattern is shown in Fig. 3.8.
Hence the node code to obtain the access pattern for processor 1 is:

do $u = \left\lfloor \frac{-56}{11} \right\rfloor, \left\lfloor \frac{-2}{11} \right\rfloor$
  do $v = \left\lceil \max\left(\frac{4+u}{3}, \frac{-u}{8}\right) \right\rceil, \left\lfloor \min\left(\frac{7+u}{3}, \frac{10-u}{8}\right) \right\rfloor$
    $A_{2D}[u + 8v, -u + 3v] = \cdots\cdots$
  end do
end do

Now we show the iterations of the nested loop and the elements accessed; the elements indeed are accessed in lexicographic order.

u	$v_{lb} = \left\lceil \max\left(\frac{4+u}{3}, \frac{-u}{8}\right) \right\rceil$	$v_{ub} = \left\lfloor \min\left(\frac{7+u}{3}, \frac{10-u}{8}\right) \right\rfloor$	accessed elements
-5	1	0	
-4	1	1	(4, 7)
-3	1	1	(5, 6)
-2	1	1	(6, 5)
-1	1	1	(7, 4)

Converting the global two-dimensional address of the accessed elements to local addresses we get the local access pattern $\{19, 22, 25, 28\}$ on processor 1.

3.7 Optimization of Loop Enumeration

A closer look at Example 3.1 reveals that even if we generated code that enumerates the points belonging to a family of parallel lines along the vector (l_1, l_2) by moving from one parallel line to the next along the vector (r_1, r_2), we would still access the elements in lexicographic order. Clearly, in this example, the above enumeration turns out to be more efficient than the earlier enumeration. We refer to this new method of enumeration as `GO-LEFT`, as we enumerate all points on a line along the vector (l_1, l_2) before we move to the next line along the other basis vector. For the same reasons, we refer to the earlier method of enumeration as `GO-RIGHT`. Next we show that the `GO-LEFT` method also enumerates elements in lexicographic order. If B (as shown in Section 3.6) is the basis matrix for the `GO-RIGHT` case then the basis for the `GO-LEFT` case is $B_L = \begin{bmatrix} r_1 & l_1 \\ r_2 & l_2 \end{bmatrix}$.
Hence the transformation matrix in the `GO-LEFT` case is ${B_L}^{-1}$.

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

Figure 3.8: Global addresses of the accessed elements of array A along with the 2-dimensional view for the case $p = 3$, $k = 4$ and $s = 11$.

3.7.1 GO-LEFT FIRST

Theorem 3.4 *Given a point $\vec{q}$ belonging to the lattice Γ_m and a pair of extremal basis vectors $\vec{l}$ and $\vec{r}$ obtained using the algorithm in Fig. 3.2, then on applying ${B_L}^{-1}$ as a transformation we maintain the access order.*

Proof: Since $\vec{l}$ and $\vec{r}$ are extremal basis vectors,

$$\vec{q} = \vec{z} + v_1\vec{l} + v_2\vec{r}$$

where $\vec{z}$ is the starting point of Γ_m and v_1 and v_2 are positive integers. So $\vec{q}^{next}$ could either be $\vec{q} + \vec{r}$ or $\vec{q} + \vec{l}$ or $\vec{q} + \vec{l} + \vec{r}$.

Let us assume that $\vec{q} + \vec{r} \in \Gamma_m$ and $\vec{q} + \vec{l} \in \Gamma_m$. This implies that $\vec{q} + \vec{r} + \vec{l} \in \Gamma_m$. With this assumption we can have the two following cases,

Case 1: $\vec{q} + \vec{r}$ is lexicographically closer to $\vec{q}$ than $\vec{q} + \vec{l}$.

Case 2: $\vec{q} + \vec{l}$ is lexicographically closer to $\vec{q}$ than $\vec{q} + \vec{r}$.

In Case 1 $\vec{q} + \vec{l}$ is lexicographically closer to $\vec{q}$ than $\vec{q} + \vec{r}$. So it should be clear that $\vec{q} + \vec{r}$ should be lexicographically closer to $\vec{q} + \vec{l}$ than to $\vec{q}$, which is impossible (due to the construction of $\vec{r}$ and $\vec{l}$). Hence our assumption that $\vec{q} + \vec{r} \in \Gamma_m$ and $\vec{q} + \vec{l} \in \Gamma_m$ is not true.

A similar argument can be used to show that out initial assumption is incorrect for Case 2 also.

From the above arguments we conclude that, given the starting point of Γ_m, we maintain the order of the points accessed by repeatedly adding $\vec{l}$ until we run out of Γ_m and then add a $\vec{r}$ and continue adding $\vec{l}$ until we run out of Γ_m again and so on. So the access order does not change on using B_L as the basis matrix, *i.e.,* applying ${B_L}^{-1}$ as the transformation. □

From the above theorem it is clear that `GO-LEFT` is always legal for the pair of basis vectors obtained using the algorithm shown in Fig. 3.2. In Chapter 4, we show that for some problem instances for which we have closed form expressions for the basis vectors it may not always be legal to `GO-LEFT`. The basis matrix for the `GO-LEFT` scheme is

$$B_L = \begin{bmatrix} r_1 & l_1 \\ r_2 & l_2 \end{bmatrix}.$$

The loop nest for $r_1 \neq 0$ is

do $u = \left\lceil \frac{mkl_1}{s} \right\rceil, \left\lfloor \frac{(\frac{s}{d}-1)(-l_2)+(mk+k-1)l_1}{s} \right\rfloor$
    do $v = \left\lceil \max\left(\frac{(mk+k-1)-ur_2}{l_2}, \frac{-ur_1}{l_1}\right) \right\rceil, \left\lfloor \min\left(\frac{mk-ur_2}{l_2}, \frac{\frac{s}{d}-1-ur_1}{l_1}\right) \right\rfloor$
        $A_{2D}[r_1u + l_1v, r_2u + l_2v] = \cdots\cdots$
    end do
end do

The node code for processor m if $r_1 = 0$ is

$$
\begin{array}{l}
\textbf{do } u = \left\lceil \frac{mkl_1}{s} \right\rceil, \left\lfloor \frac{(\frac{s}{d}-1)(-l_2)+(mk+k-1)l_1}{s} \right\rfloor \\
\quad \textbf{do } v = \left\lceil \max\left(\frac{(mk+k-1)-ur_2}{l_2}, 0\right) \right\rceil, \left\lfloor \min\left(\frac{mk-ur_2}{l_2}, \frac{\frac{s}{d}-1}{l_1}\right) \right\rfloor \\
\qquad A_{2D}[l_1 v, r_2 u + l_2 v] = \cdots\cdots \\
\quad \textbf{end do} \\
\textbf{end do}
\end{array}
$$

Now we need to decide when it is beneficial to `GO-LEFT`. The amount of work that needs to be done to evaluate the inner loop bounds is the same for each outer loop iteration in both the enumeration methods. So an enumeration is that results in fewer outer loop iterations is the scheme of choice. The number of elements accessed per line in the two cases is a function of the second components of the basis vectors. If $r_2 \leq -l_2$, we use `GO-RIGHT`; else, we use `GO-LEFT`.

Example 3.2 Code generated for the case where $\ell = 0$, $p = 3$, $k = 4$ and $s = 11$ for processor 1 when we choose to `GO-LEFT`.

The basis vectors obtained by running through the algorithms shown in Fig. 3.2 or Fig. 3.7 are $\vec{l} = (1, -1)$ and $\vec{r} = (8, 3)$. Hence the resulting node code for processor 1 is,

$$
\begin{array}{l}
\textbf{do } u = \left\lceil \frac{4}{11} \right\rceil, \left\lfloor \frac{17}{11} \right\rfloor \\
\quad \textbf{do } v = \lceil \max(3u-7, -8u) \rceil, \lfloor \min(3u-4, 10-8u) \rfloor \\
\qquad A_{2D}[8u+v, 3u-v] = \cdots\cdots \\
\quad \textbf{end do} \\
\textbf{end do}
\end{array}
$$

Now we shall run through the nested loop to show that the nested loop does indeed access the elements in the right order.

u	v_{lb}	v_{ub}	accessed elements
1	-4	-1	(4, 7), (5, 6), (6, 5), (7, 4)

Here we observe that unlike the previous example, we scan all the elements along a single line rather than 4 different lines. Clearly in this case going left is the better choice.

3.7.2 Address Generation for the Array Section $(\ell : h : s)$

We observe from the example in Sections 3.6 and 3.7.1 that the code generated for `GO-RIGHT` enumerates the points that belong to a family of parallel lines, *i.e.*, along the vector $\vec{r}$, by moving from one parallel line to the next within the family along the vector $\vec{l}$ and the code generated for `GO-LEFT` enumerates the points that belong to a family of parallel lines along the vector $\vec{l}$, by moving from one parallel line to the next within the family along the vector $\vec{r}$. So in the code derived in Section 3.6, the outer loop iterates over the set of parallel lines while the inner loop iterates over all the elements accessed in each line on a given processor.

From the previous example it can be seen that we may scan a few empty lines (*i.e.*, lines on which no element is accessed) in the beginning and the end. This can be avoided by evaluating a tighter lower bound for the outer loop using the $start$ and end elements evaluated in the algorithm shown in Fig. 3.2. The start line u_{start} and end line u_{end} can be evaluated as follows (using `GO-RIGHT` enumeration scheme):

$$\begin{aligned} l_1 u_{start} + r_1 v_{start} &= start \text{ div } pk - \ell \text{ div } pk \\ l_2 u_{start} + r_2 v_{start} &= start \text{ mod } pk - \ell \text{ mod } pk \end{aligned}$$

Hence,

$$u_{start} = \frac{r_2(start \text{ div } pk - \ell \text{ div } pk) - r_1(start \text{ mod } pk - \ell \text{ mod } pk)}{s};$$

similarly,

$$u_{end} = \frac{r_2(end \text{ div } pk - \ell \text{ div } pk) - r_1(end \text{ mod } pk - \ell \text{ mod } pk)}{s}.$$

The inner loop of the node code evaluates the start element for each iteration of the outer loop *i.e.*, each line traversed. In our implementation of the loop enumeration we use the start element of the previous line traversed to obtain the start element of the next line. This eliminates the expensive integer divisions involved in evaluating the start elements on the different lines. A simple algorithm for `GO-RIGHT` followed by us is given below.

1. Scan all the elements on the first line (u_{start}), starting at the $start$ element and then adding the right basis vector until you run out of the processor space.

2. From the previous start add a left basis vector and then add as many right basis vectors needed until you get back onto the processor space. The element thus obtained is the start for the new line. Starting at this element add the right basis vector until you run out of the processor space. Repeat this until the line immediately before the last line (u_{end}).

3. Obtain the start point on the last line as before. Scan all the elements along the line from the start by adding the right basis vector until you reach the end element.

The algorithm for `GO-LEFT` is given below.

1. Scan all the elements on the first line (u_{start}), starting at the $start$ element and then adding the left basis vector until you run out of the processor space.

2. From the previous start add a right basis vector and then add as many left basis vectors needed until you get back onto the processor space. The element thus

obtained is the start for the new line. Starting at this element add the left basis vector until you run out of the processor space. Repeat this until the line immediately before the last line (u_{end}).

3. Obtain the start point on the last line as before. Scan all the elements along the line from the start by adding the left basis vector until you reach the end element.

3.8 Experimental Results

We performed experiments on our pattern generation algorithm on an IBM RS-6000. We used the `cc` compiler with `-O3` level optimizations turned on; the UNIX system call `gettimeofday()` was used to measure time. When computing time for 32 processors, we timed the code that computes the access pattern for each processor, and report the maximum time over all processors. We experimented with block sizes in powers of 2 ranging from 4 to 1024 for 32 processors. The total times for the two different implementations ("Right" and "Zigzag") of the algorithm in [10] and our algorithm include basis and table generation times. The plots in Figs. 3.9–3.14 show the total times for the above three algorithms and the total time for pattern generation for the algorithm proposed by Chatterjee *et al.* [3]. For very small block sizes, all the methods have comparable performance. At block sizes from 16 onward, our solution outperforms the other three. For higher block sizes, our pattern generation algorithm performs 2 to 3 times faster than the two Rice [10] implementations.

From our choice of enumeration, we decide to use `GO-LEFT` for $s = pk - 1$ and use `GO-RIGHT` for $s = pk + 1$. Since the algorithms in [10] do not exploit this enumeration choice, our algorithm performs significantly better.

3.9 Summary

In this chapter we showed that array accesses in an array section with HPF-style data mappings belong to an integer lattice. We then presented efficient algorithms to determine the basis vectors of the lattice. We also presented a novel technique of enumerating the elements that are accessed using loop nests. Finally, we provided experimental results that show the pattern generation algorithm presented in this chapter to be the fastest known solution to the address generation problem.

Table 3.1: Times (in microseconds) for Loop enumeration, Right and Zigzag of Rice, and the Sort approach on an IBM RS-6000 $(p = 32, s = 7, s = 99)$

Block Size	$s = 7$				$s = 99$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
4	9	8	8	9	11	9	9	11
8	11	10	10	15	12	11	12	17
16	11	12	12	26	13	14	13	28
32	14	17	17	59	17	20	18	61
64	21	26	27	186	30	31	36	180
128	32	43	44	406	44	57	58	344
256	56	82	89	745	71	110	117	750
512	105	157	156	1401	119	181	211	1632
1024	200	314	328	2887	219	357	349	3284

Table 3.2: Times (in microseconds) for Loop enumeration, Right and Zigzag of Rice, and the Sort approach on an IBM RS-6000 $(p = 32, s = k - 1, s = k + 1)$

Block Size	$s = k - 1$				$s = k + 1$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
4	9	8	8	9	10	8	8	10
8	11	10	10	15	11	9	10	14
16	11	12	12	27	12	12	12	27
32	14	18	15	61	15	18	15	61
64	31	32	32	186	30	30	35	195
128	42	56	51	377	42	49	54	406
256	70	106	87	751	69	86	91	741
512	123	201	172	1418	123	166	165	1528
1024	235	446	315	4215	232	309	333	3222

Table 3.3: Times (in microseconds) for Loop enumeration, Right and Zigzag of Rice, and the Sort approach on an IBM RS-6000 ($p = 32, s = pk - 1, s = pk + 1$)

Block Size	$s = pk - 1$				$s = pk + 1$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
4	8	7	7	9	7	7	7	8
8	9	9	8	14	8	8	8	13
16	10	12	11	26	9	10	10	25
32	13	18	15	61	11	15	15	57
64	19	30	24	178	16	24	24	179
128	30	54	43	367	26	42	42	342
256	53	109	79	678	45	78	78	861
512	99	204	167	1680	83	157	151	1378
1024	191	399	316	3118	160	300	302	2957

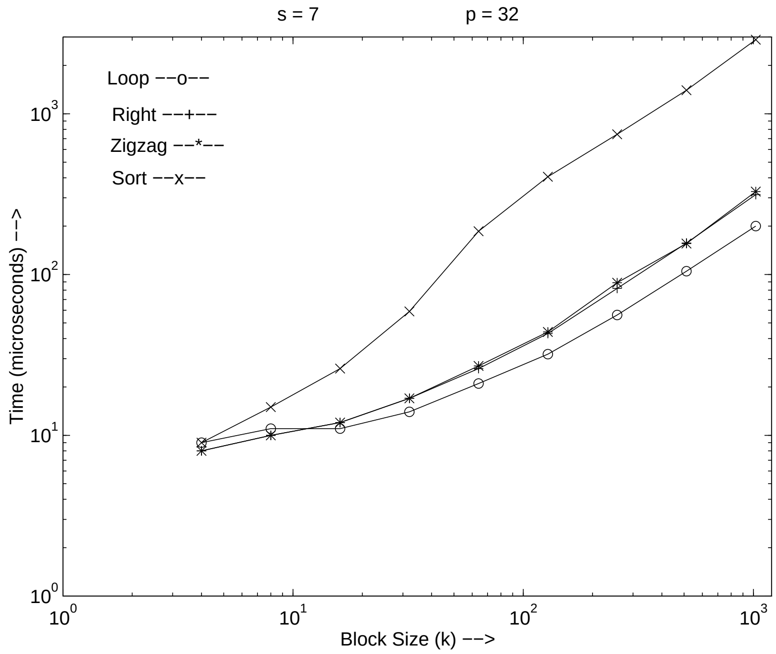

Figure 3.9: Total time for address generation for $p = 32$ processors and $s = 7$ for several block sizes

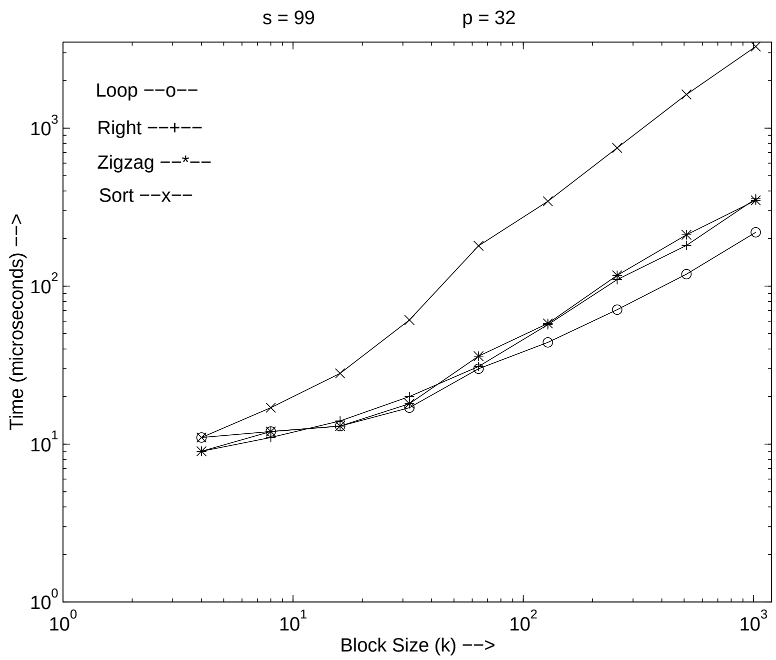

Figure 3.10: Total time for address generation for $p = 32$ processors and $s = 99$ for several block sizes

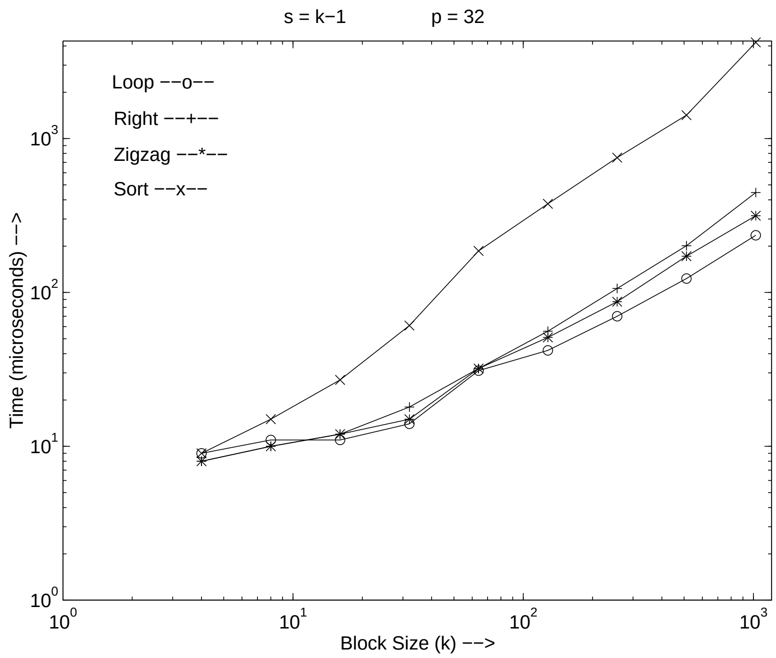

Figure 3.11: Total time for address generation for $p = 32$ processors and $s = k - 1$ for several block sizes

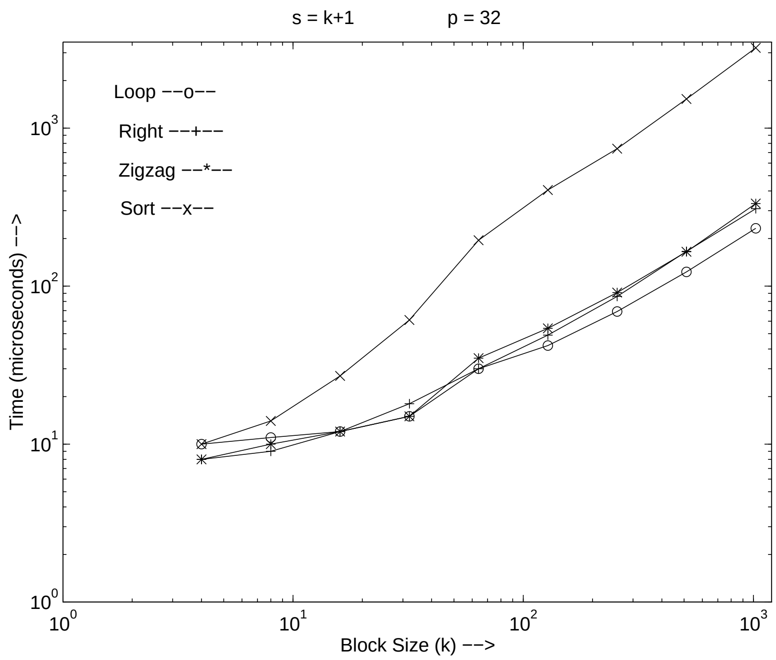

Figure 3.12: Total time for address generation for $p = 32$ processors and $s = k + 1$ for several block sizes

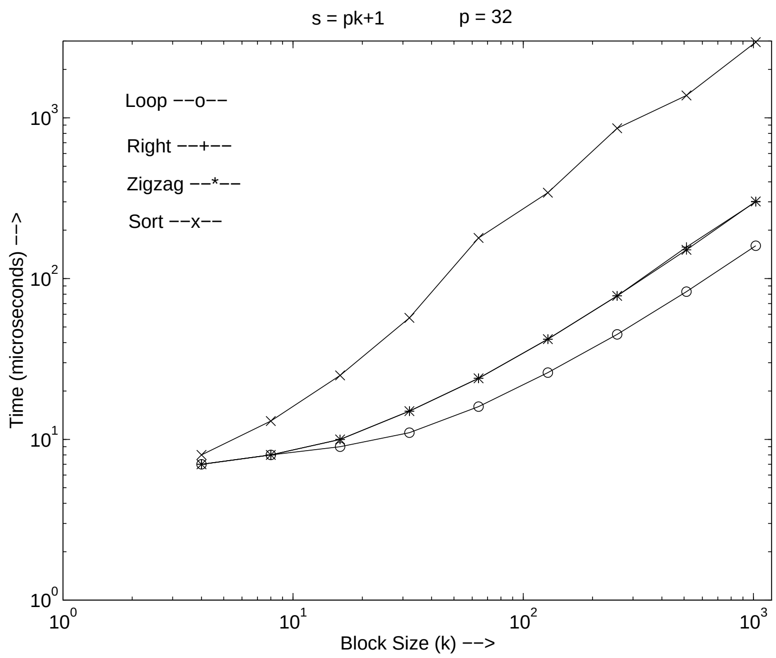

Figure 3.13: Total time for address generation for $p = 32$ processors and $s = pk + 1$ for several block sizes

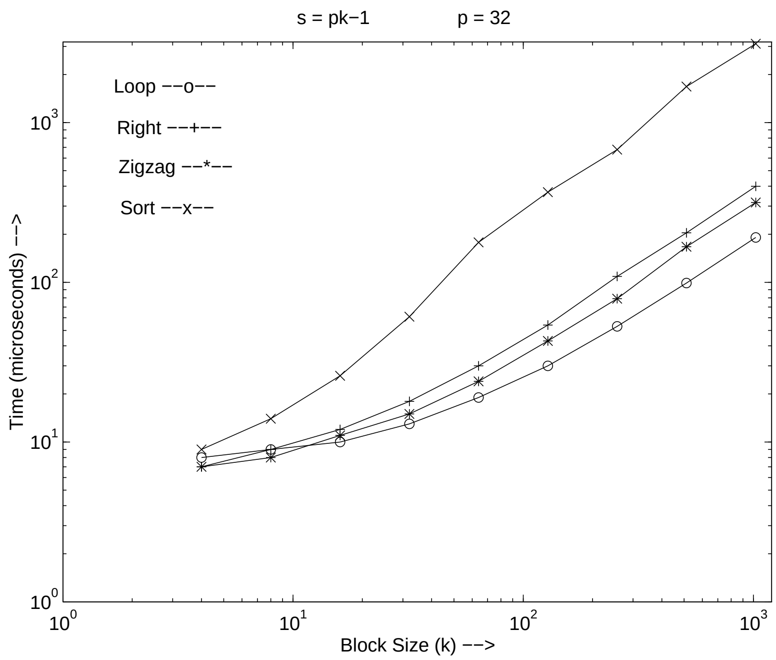

Figure 3.14: Total time for address generation for $p = 32$ processors and $s = pk - 1$ for several block sizes

CHAPTER 4

COMPILE-TIME APPROACH TO COMPUTE ADDRESS SEQUENCES

Closed form expressions result in avoiding having to solve the k Diophantine equations for basis determination as explained in the previous chapter. This virtually eliminates the runtime overhead incurred for basis generation by the algorithm presented in the previous chapter and [10]. Closed form solutions are also better than the improved algorithm presented in Section 3.5. As in the improved algorithm we solve two linear Diophantine equations to get a pair of basis vectors which we then use to enumerate the smallest elements accessed in each column. We use these elements to obtain the best pair of basis vectors for the array section lattice. Though the new algorithm improves upon the performance of the original algorithm, it requires a substantial run-time overhead as compared to evaluating closed form solutions. Experimental results provided at the end of this chapter reflect this.

4.1 CLOSED FORM EXPRESSIONS FOR BASIS VECTORS AND PATTERN DERIVATION FOR ARRAY STATEMENTS

Here we derive closed form expressions for *extremal basis vectors, i.e.*, those basis vectors using non-negative linear combinations of which every point in the lattice can be generated. Using the extremal basis vectors, we derive a loop transformation that enumerates the address sequence. Next, we show that the vectors $(1, -(pk \bmod s))$ and $(0, s)$ form an extremal basis set for the array section lattice, using a two dimensional view of global addresses.

4.1.1 ACCESS PATTERN GENERATION FOR AN ARRAY SECTION WITH $l = 0$

Informally, since the access stride is s, two consecutive points that belong to the same course differ by the vector $(0, s)$. If s divides pk, then the access pattern for the first course is the same as that for each of the remaining courses. Thus, in this case, it is sufficient to generate only the access pattern for the first course. If s does not divide pk, assuming that the element 0 is accessed, the last element accessed on course number zero is $(0, pk - pk \bmod s)$, and the first element accessed on course number one is $(1, s - pk \bmod s)$. The largest element on course one that is smaller than $(1, s - pk \bmod s)$ is $(1, -pk \bmod s)$.

Theorem 4.1 *The vectors* $(0, s)$ *and* $(1, -(pk \bmod s))$ *are basis vectors of the array section lattice* $\langle A(\ell : h : s), p, k\rangle$.

Proof: Without loss of generality, let us assume that the first array element accessed lies on the origin of the two dimensional address space. To show that $(0, s)$ and $(1, -(pk \bmod s))$ form a pair of basis vectors all we need to show is that no other point belonging to the integer lattice lies in the triangle formed by the points $(0, 0)$, $(0, s)$ and $(1, -(pk \bmod s))$ [6] (See Theorem 3.1 on p. 14).

Let us assume that (x, y) belongs to the integer lattice and also lies in the triangle formed by the above points. Hence the constraints on x and y are, $0 \leq x \leq 1$ and $-(pk \bmod s) \leq y \leq s$. That is the point (x, y) either lies on course 0 or course 1. Since the access stride is s no point belonging to the lattice falls between $(0, 0)$ and $(0, s)$. From Fig. 4.1 it is obvious that the only points on course 1 that can satisfy the above constraints are $(1, -(pk \bmod s))$ and $(1, s - (pk \bmod s))$. $(1, s - (pk \bmod s))$ can lie in the triangle formed by the above three points only if the area of the triangle is zero. This is impossible since the vectors $(0, s)$ and $(1, -(pk \bmod s))$ are linearly independent. Hence no point belonging to the integer lattice lies in the triangle. Therefore, $(0, s)$ and $(1, -(pk \bmod s))$ form a pair of basis vectors. □

Theorem 4.2 *The vectors* $(0, s)$ *and* $(1, -(pk \bmod s))$ *form a pair of extremal basis vectors of the array section lattice* $\langle A(\ell : h : s), p, k \rangle$.

Proof: We have already seen that $(0, s)$ and $(1, -(pk \bmod s))$ are basis vectors of the integer lattice. Let us assume, that $\vec{z}$ is the first point accessed on a given processor. Without loss of generality we can assume that $\vec{z}$ is at the origin of a two dimensional space. All the points in the lattice belong to a rectangular region which is a proper subset of the set of points which can be written as non-negative integer linear combinations of $(0, s)$ and $(1, -(pk \bmod s))$. Hence the vectors $(0, s)$ and $(1, -(pk \bmod s))$ form a pair of extremal basis vectors. □

A *basis matrix* is a matrix whose columns are basis vectors. The basis matrix of the array section lattice $\langle A(\ell : h : s), p, k \rangle$ is

$$B = \begin{bmatrix} 1 & 0 \\ -(pk \bmod s) & s \end{bmatrix}$$

The determinant of B is s, the access stride.

The access pattern for array A repeats after every $\frac{s}{\text{GCD}\,(s,pk)}$ courses of elements as noted in [3]. Therefore, it is sufficient to store the access pattern of the elements in the first $\frac{s}{\text{GCD}\,(s,pk)}$ courses and use this information to determine the remaining elements accessed. In this section, we assume $l = 0$ in $A(\ell : h : s)$; address pattern determination for array sections with arbitrary limits is discussed in Section 4.1.2. The code that iterates over the first $\frac{s}{\text{GCD}\,(s,pk)}$ courses in the two dimensional lattice shown in Fig. 2.1(a) could be written as follows:

do $i = 0, \frac{s}{\text{GCD}\,(s,pk)} - 1$
    do $j = 0, (pk - 1)$

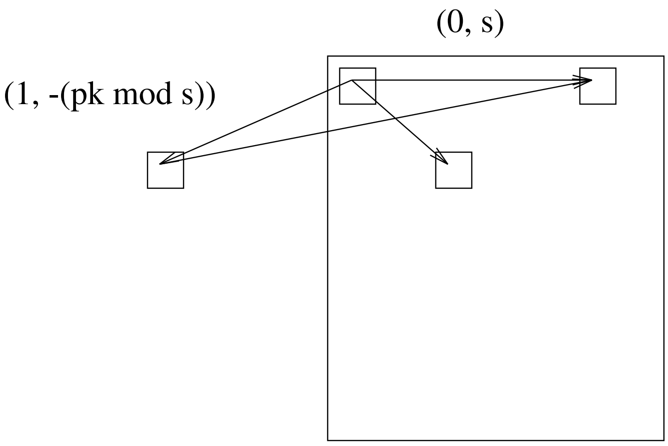

Figure 4.1: The vectors $(0, s)$ and $(1, -pk \bmod s)$ are extremal basis vectors.

$A_{2D}(i,j) = \cdots\cdots$
end do
end do

As the access stride is s, only $\frac{1}{s}$ of the allocated space is spanned. In order to access only $\frac{1}{s}$ of the allocated space we need to apply a loop transformation to the above loop, the determinant of whose transformation matrix [15, 22] should be $\frac{1}{s}$. The transformation matrix we use is the inverse of the basis matrix B. Also, in this case the first element allocated in a processor memory is mk, where m is the processor number. In order to obtain the address sequence for each of the processors, we need to apply the loop transformation to the following modified code.

do $i = 0, \frac{s}{\text{GCD}\,(s,pk)} - 1$
do $j = mk, (k-1) + mk$
$A_{2D}[i,j] = \cdots$
end do
end do

The transformation matrix is

$$B^{-1} = \begin{bmatrix} 1 & 0 \\ \frac{(pk \bmod s)}{s} & \frac{1}{s} \end{bmatrix}.$$

The effect of this transformation is to enumerate the addresses (starting from the origin) which are multiples of the vector $(0, s)$ till there are no more points; then start from the point that is at a distance $(1, -pk \bmod s)$ from the origin, and enumerate the addresses which are multiples of the vector $(0, s)$ till there are are no more points, and so on. In essence, we enumerate points that belong to a family of parallel lines (corresponding to the vector $(0, s)$), by moving from one parallel line to the next along the vector $(1, -pk \bmod s)$. In general, if

$$B = \begin{bmatrix} l_1 & r_1 \\ l_2 & r_2 \end{bmatrix},$$

we enumerate points that belong to a family of parallel lines, *i.e.,* along the vector (r_1, r_2), by moving from one parallel line to the next within the family along the vector (l_1, l_2). The loop bounds can be written as follows (using the fact that B is an invertible matrix):

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} 0 \\ \frac{s}{\text{GCD}\,(s,pk)} - 1 \\ -mk \\ (k-1) + mk \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} BB^{-1} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} 0 \\ \frac{s}{\text{GCD}\,(s,pk)} - 1 \\ -mk \\ (k-1) + mk \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -(pk \bmod s) & s \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \leq \begin{bmatrix} 0 \\ \frac{s}{\text{GCD}\,(s,pk)} - 1 \\ -mk \\ (k-1)+mk \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ pk \bmod s & -s \\ -(pk \bmod s) & s \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \leq \begin{bmatrix} 0 \\ \frac{s}{\text{GCD}\,(s,pk)} - 1 \\ -mk \\ (k-1)+mk \end{bmatrix},$$

where $\begin{bmatrix} u \\ v \end{bmatrix} = B^{-1} \begin{bmatrix} i \\ j \end{bmatrix}$ and u and v are integers. Hence,

$$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} u \\ vs - u(pk \bmod s) \end{bmatrix}.$$

Using Fourier-Motzkin elimination [22], we solve for u and v as follows:

$$\begin{aligned} -u &\leq 0 \\ u &\leq \frac{s}{\text{GCD}\,(s,pk)} - 1 \\ u(pk \bmod s) - sv &\leq -mk \\ -u(pk \bmod s) + sv &\leq (k-1)+mk \end{aligned}$$

Since u and v have to be integers, the above inequalities lead to:

$$\begin{aligned} 0 \leq \quad u \quad &\leq \frac{s}{\text{GCD}\,(s,pk)} - 1 \\ \left\lceil \frac{u(pk \bmod s)+mk}{s} \right\rceil \leq \quad v \quad &\leq \left\lfloor \frac{k-1+u(pk \bmod s)+mk}{s} \right\rfloor. \end{aligned}$$

Thus, the code for processor m for the first $\frac{s}{\text{GCD}\,(s,pk)}$ courses (using a two dimensional view of the global array A) is:

do $u = 0, \frac{s}{\text{GCD}\,(s,pk)} - 1$
  do $v = \left\lceil \frac{u(pk \bmod s)+mk}{s} \right\rceil, \left\lfloor \frac{k-1+u(pk \bmod s)+mk}{s} \right\rfloor$
    $A_{2D}[u, vs - u(pk \bmod s)] = \cdots\cdots$
  end do
end do

The code which is further simplified and uses local addresses of A_{loc} is shown below:

$\alpha = pk \bmod s$
$k' = k - 1$
$\beta = mk$

do $u = 0, \frac{s}{\text{GCD}\,(s,pk)} - 1$
    $t = \alpha u + \beta$
    do $v = \left\lceil \frac{t}{s} \right\rceil, \left\lfloor \frac{k'+t}{s} \right\rfloor$
        $A_{loc}[uk + vs - t] = \cdots\cdots$
    end do
end do

Here, β is the global address of the first element allocated to a processor.

Since the access pattern for the array A repeats after every $\frac{s}{\text{GCD}\,(s,pk)}$ courses of elements, we can use the above code to derive the access pattern. The optimized code that computes the access pattern is shown in Fig. 4.2. Here we show an example that illustrates our pattern generation technique.

Example 4.1 Pattern determination for the case where $p = 4$, $k = 4$ and $s = 3$.

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

Memory allocation for array A along with the global and local addresses of the accessed elements for the case p=4, k=4 and s=3 is shown above. The pattern generation code for processor 1 is shown next.

$i = 0$
do $u = 0, \frac{s}{\text{GCD}\,(s,pk)} - 1$
    $t = u + 4$
    do $v = \left\lceil \frac{t}{3} \right\rceil, \left\lfloor \frac{3+t}{3} \right\rfloor$
        $pattern_1[i] = 4u + 3v - t$
        Increment i
    end do
end do

Here we run through the iterations of the above loop and show that the code does indeed generate the access pattern for processor 1 in terms of local address offsets.

u	$v_{lb} = \left\lceil \frac{u+4}{3} \right\rceil$	$v_{ub} = \left\lfloor \frac{u+7}{3} \right\rfloor$	local index $= 3u + 3v - 4$
0	2	2	2
1	2	2	5
2	2	3	8, 11

$\alpha = pk \bmod s$
$\beta = mk$
$k' = k - 1$
$i = 0$
$s' = \frac{s}{\text{GCD}\,(s,pk)}$

`/* Compute and store the access pattern in` $pattern_m$ `*/`

do $u = 0, s' - 1$
    $t = \alpha u + \beta$
    do $v = \left\lceil \frac{t}{s} \right\rceil, \left\lfloor \frac{k'+t}{s} \right\rfloor$
        $pattern_m[i] = uk + vs - t$
        Increment i
    end do
end do
length $= i$

`/* Use` $pattern_m$ `and execute code */`
`/* N-1 is the upper bound of the array section */`

$N' = \frac{\left\lfloor \frac{N-1}{pk} \right\rfloor}{s'}$
$j = 0$
do $w = 0, N'$
    offset $= j * k$
    do $u = 0, \text{length} - 1$
        $A_{loc}[\text{offset} + pattern_m(u)]$
    end do
    $j = j + s'$
end do

Figure 4.2: Optimized code for $A(0 : N - 1 : s)$ computes local access pattern for processor m

4.1.2 Pattern Generation for the Array Section $(\ell : h : s)$

For the array section given in Equation 2.1, the origin of the two dimensional lattice is no longer the first element allocated to processor 0. We need to shift the origin in the loop shown before to the element with global address l. The global address of the first element allocated to a processor after taking the shift of origin into account is $\beta = mk - l$.

The code for access pattern determination for the array assignment statement shown in Equation 2.1 consists of a nested loop. The outer loop iterates over every course of size k of array A assigned to that processor. Since the first non-empty course of the lattice is the $\left\lfloor \frac{l}{pk} \right\rfloor^{th}$ course, we need to modify the lower bound of the outer loop to $\left\lfloor \frac{l}{pk} \right\rfloor$; in addition, we need to avoid accessing elements to the left of l on course $\left\lfloor \frac{l}{pk} \right\rfloor$. This is done by generating two pattern tables for the processor that owns element l; the first pattern table known as the *preamble*, holds the access pattern for the elements on that processor in course $\left\lceil \frac{l}{pk} \right\rceil$ and *pattern* stores the access pattern. For all other processors, a single access pattern is derived. When executing code, the processor that owns l first uses the *preamble* table, and then repeatedly uses the *pattern* table. The other processors use their *pattern* tables to execute code. The modified code for Equation 2.1 is shown in Fig. 4.3. In the rest of this paper for the sake of clarity we shall use loose upper and lower bounds, as boundary effects can be easily rectified as shown above.

4.1.3 Pattern Generation for Alignment Offset b

For the array section given in Equation 2.1, where the array A is aligned with template cell $T(ai + b)$ where $a = 1$ (and b is any integer), the origin of the two dimensional lattice is no longer the first array element allocated to processor 0. We need to shift the origin in the loop shown above to the element with global address $l + b$. So the global address of the first element allocated to a processor after taking the shift of origin into account is, $\beta = mk - l - b$. This modification needs to be done in the code shown in Fig. 4.3.

4.1.4 Go-Left First

It is not always legal to use `GO-LEFT` with $(0, s)$ and $(1, -(pk \bmod s))$ as the choice of basis vectors. It has been proved in Section 3.7.1 that it is legal to go left only if, for every point $\vec{q}$ in the lattice Γ_m, both $\vec{q} + (0, s)$ and $\vec{q} + (1, -(pk \bmod s))$ do not belong to Γ_m. This is possible if and only if $s + (pk \bmod s) \geq k$. Hence it is legal to go left only if $s + (pk \bmod s) \geq k$. The decision on the choice of enumeration method is similar to that explained in Section 3.7.1.

$\alpha = pk \bmod s$
$k' = k - 1$
$\beta = mk - l$
$u = course_l = \left\lfloor \frac{l}{pk} \right\rfloor$
if $(l \bmod (pk)) > (mk + k - 1)$
    Increment $course_l$
endif
if $\left\lfloor \frac{l}{k} \right\rfloor \bmod p = m$
    $i = 0$
    do $v = (l \bmod k), k - 1, s$
        $preamble_m[i] = uk + v$
        Increment i
    end do
    preamble_length $= i + 1$
    Increment $course_l$
endif
$i = 0$
do $u = course_l, course_l + \frac{s}{\text{GCD}\,(s,pk)} - 1$
    $t = \alpha u + \beta$
    do $v = \left\lceil \frac{t}{s} \right\rceil, \left\lfloor \frac{k'+t}{s} \right\rfloor$
        $pattern_m[i] = uk + vs - t$
        Increment i
    end do
end do

Figure 4.3: Code for pattern determination for an array section with lower bound l for processor m

4.2 CLOSED FORM EXPRESSIONS FOR BASIS VECTORS AND PATTERN DETERMINATION WHEN $s \geq k$

It is evident from Fig. 2.2 that in the case where $s > k$, at most one element is accessed per course in a given processor. Therefore, the vector (0, s) is a poor choice for one of the basis vectors. Since $s \geq k$ in this case, given that a certain element is accessed the next element accessed on a processor would definitely not belong to the same course. Thus, the best choice for one of the basis vectors would be the vector that spans the next element accessed form the origin. So with this new basis vector and the vector $(1, -(pk \bmod s))$ we will scan the accessed elements of the array A along a pair of slanted parallel lines each of which have at least one accessed element. So unlike the previous case, with this new set of basis vectors we would end up skipping all the empty courses. Next, we derive closed form expressions for the best pair of basis vectors as a function of the problem parameters s, p, and k, and present the optimized code for each case.

4.2.1 CASE WHERE $0 < s - (pk \bmod s) < k$

In this case, given that a certain location is accessed the next location accessed is at a vector distance of $(1, s - (pk \bmod s))$. We next prove that the above vector and $(1, -(pk \bmod s))$ form an extremal set of basis vectors.

Theorem 4.3 *$(1, -(pk \bmod s))$ and $(1, s - (pk \bmod s))$ form a pair of extremal basis vectors when $0 < s - (pk \bmod s) < k$.*

Proof: The vectors $(1, -(pk \bmod s))$ and $(1, s-(pk \bmod s))$ form a pair of basis vectors of the integer lattice as the determinant of the matrix whose columns are $(1, -(pk \bmod s))$ and $(1, s - (pk \bmod s))$ is s.

It has been proved earlier that $(1, -(pk \bmod s))$ is an extremal basis vector for integer lattice.

Since $s \geq k$, every course has at most one accessed element. Without loss of generality let us assume that the starting element on a processor is the origin of the two dimensional space. Let us assume that (x, y) belongs to the integer lattice then,

$$\begin{bmatrix} x \\ y \end{bmatrix} = a \begin{bmatrix} 1 \\ -(pk \bmod s) \end{bmatrix} + b \begin{bmatrix} 1 \\ s - (pk \bmod s) \end{bmatrix}$$

Since $(1, -(pk \bmod s))$ is an extremal basis vector for integer lattice, a can take only positive integer values. If b is negative then we will have more than one point on a line which is impossible. Hence, b can take only positive integer values.

Therefore, $(1, -(pk \bmod s))$ and $(1, s - (pk \bmod s))$ form a pair of extremal basis vectors when $0 < s - (pk \bmod s) < k$. □

With this choice of basis vectors, in the transformed iteration space, we will access the elements of A along parallel slanted lines having at least one accessed element. The basis matrix for this case is

$$B = \begin{bmatrix} 1 & 1 \\ -(pk \text{ mod } s) & s-(pk \text{ mod } s) \end{bmatrix}.$$

The transformation matrix is of the form:

$$B^{-1} = \frac{1}{s}\begin{bmatrix} s-(pk \text{ mod } s) & -1 \\ pk \text{ mod } s & 1 \end{bmatrix}$$

As before the loop bounds can be written as follows: $(\mathcal{C} = \frac{s}{gcd(s,pk)} - 1)$

$$\begin{aligned}
\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} &\le \begin{bmatrix} 0 \\ \mathcal{C} \\ -mk \\ (k-1)+mk \end{bmatrix} \\
\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} BB^{-1}\begin{bmatrix} i \\ j \end{bmatrix} &\le \begin{bmatrix} 0 \\ \mathcal{C} \\ -mk \\ (k-1)+mk \end{bmatrix} \\
\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ -(pk \text{ mod } s) & s-(pk \text{ mod } s) \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} &\le \begin{bmatrix} 0 \\ \mathcal{C} \\ -mk \\ (k-1)+mk \end{bmatrix} \\
\begin{bmatrix} -1 & -1 \\ 1 & 1 \\ pk \text{ mod } s & pk \text{ mod } s - s \\ -(pk \text{ mod } s) & s-(pk \text{ mod } s) \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} &\le \begin{bmatrix} 0 \\ \mathcal{C} \\ -mk \\ (k-1)+mk \end{bmatrix}
\end{aligned}$$

where $\begin{bmatrix} u \\ v \end{bmatrix} = B^{-1}\begin{bmatrix} i \\ j \end{bmatrix}$ and u and v are integers. Therefore,

$$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} u+v \\ v(s-(pk \text{ mod } s)) - u(pk \text{ mod } s) \end{bmatrix}.$$

Now using Fourier-Motzkin elimination [22] we solve for u and v as follows:

$$\begin{aligned}
-u-v &\le 0 \\
u+v &\le \mathcal{C} \\
u(pk \text{ mod } s) - (s-(pk \text{ mod } s))v &\le -mk \\
-u(pk \text{ mod } s) + (s-(pk \text{ mod } s))v &\le (k-1)+mk
\end{aligned}$$

Hence,

$$\frac{-mk-k+1}{s} \leq \quad u \quad \leq \frac{\mathcal{C}(s-(pk \bmod s))-mk}{s}$$
$$\max\left(\frac{mk+(pk \bmod s)u}{s-(pk \bmod s)}, -u\right) \leq \quad v \quad \leq \min\left(\frac{mk+k-1+(pk \bmod s)u}{s-(pk \bmod s)}, \mathcal{C}-u\right)$$

Finally, the node code for access pattern determination for processor m is: $(i \leftarrow 0)$

do u = $\left\lceil \frac{-mk-k+1}{s} \right\rceil, \left\lfloor \frac{\mathcal{C}(s-(pk \bmod s))-mk}{s} \right\rfloor$
  do v = $\left\lceil \max\left(\frac{mk+(pk \bmod s)u}{s-(pk \bmod s)}, -u\right)\right\rceil, \left\lfloor \min\left(\frac{mk+k-1+(pk \bmod s)u}{s-(pk \bmod s)}, \mathcal{C}-u\right)\right\rfloor$
    $pattern_m[i] = u(k-(pk \bmod s)) + v(k+s-(pk \bmod s)) - mk$
    Increment i
  end do
end do

As before (for example, in Figs. 4.2 and 4.3), the above code in our implementation is optimized and eliminates expensive integer operations where possible.

Here again we show an example that demonstrates that the code generated as shown above does indeed access the elements of A in the local memories of the processors in the required sequence.

Example 4.2 Code generation for the case where $p = 3$, $k = 4$ and $s = 5$.

Since in this case $(s - (pk \bmod s)) < k$, we use the suggested basis vectors. Memory allocation for elements of array A along with the global and local addresses of the accessed elements,for the case p=3, k=4 and s=5 is shown in Fig. 2.2. The pattern determination code for processor 1 is shown below:

$i = 0$;
do $u = -1, \left\lfloor \frac{8}{5} \right\rfloor$
  $t = 2u + 4$
  do $v = \left\lceil \max\left(\frac{t}{3}, -u\right)\right\rceil, \left\lfloor \min\left(\frac{t+3}{3}, 4-u\right)\right\rfloor$
    $pattern_1[i] = 4(u+v) + 3v - t$
    Increment i
  end do
end do

Here again we run through the first couple of iterations of the above loop and show that the code does indeed access the right elements in the right order.

u	v_{lb}	v_{ub}	local index $= 2u + 7v - 4$
-1	1	1	1
0	2	2	10
1	2	3	12, 19

4.2.2 CASE WHERE $0 < (pk \text{ mod } s) < k$

For the case where $s > k$, only one element is accessed per course. So even if we had interchanged the columns of the basis matrix shown in the previous section and used its inverse as the transformation matrix, we would access the same elements and in the lexicographic order. If $0 < s - (pk \text{ mod } s) < k$ is not true, we could check to see if $0 < (pk \text{ mod } s) < k$. So in this case, we choose

$$B = \begin{bmatrix} 1 & 1 \\ s - (pk \text{ mod } s) & -(pk \text{ mod } s) \end{bmatrix}$$

as the basis matrix. Here again in the transformed space we iterate over slanted parallel lines having at least one element. The node code for processor m with this choice of a basis matrix is:

$i = 0$;
do $u = \left\lceil \frac{mk}{s} \right\rceil, \left\lfloor \frac{\mathcal{C}(pk \text{mod} s)+mk+k-1}{s} \right\rfloor$
  do $v = \left\lceil \max\left(\frac{(s-(pk \text{mod} s))u-mk-k+1}{(pk \text{mod} s)}, -u\right)\right\rceil, \left\lfloor \min\left(\frac{(s-(pk \text{mod} s))u-mk}{(pk \text{mod} s)}, \mathcal{C} - u\right)\right\rfloor$
    $pattern_m[i] = k(u+v) + u(s - (pk \text{ mod } s)) - v(pk \text{ mod } s) - mk$
    Increment i
  end do
end do

If both $(pk \text{ mod } s)$ and $(s - (pk \text{ mod } s))$ lie between 0 and k, it would help to choose the basis matrix corresponding to the smaller of the two. With this choice of a basis matrix, we end up with fewer number of outer loop iterations in the transformed loop nest. The resulting code will be more efficient as we end up performing fewer upper and lower bound calculations for the inner loop in the transformed loop nest.

4.2.3 CASE WHERE $0 < s - q(pk \text{ mod } s) < k$

If $pk \text{ mod } s \geq k$ and $s - (pk \text{ mod } s) \geq k$, an obvious extension to the previous cases is to find an integer q such that $0 < s - q(pk \text{ mod } s) < k$. All the possible integer values for q are given by the following inequality:

$$\left\lceil \frac{s-k}{pk \text{ mod } s} \right\rceil \leq q \leq \left\lfloor \frac{s}{pk \text{ mod } s} \right\rfloor .$$

If $\left\lceil \frac{s-k}{pk \text{mod} s} \right\rceil \leq \left\lfloor \frac{s}{pk \text{mod} s} \right\rfloor$, then the next element accessed is at a vector distance of $(1, s - q(pk \text{ mod } s))$ where $q = \left\lfloor \frac{s}{pk \text{mod} s} \right\rfloor$. Next, we prove that $(1, -pk \text{ mod } s)$ and $(q, s - q * pk \text{ mod } s)$ are extremal basis vectors.

Theorem 4.4 $(1, -(pk \text{ mod } s))$ *and* $(q, s - q(pk \text{ mod } s))$ *form a pair of extremal basis vectors when* $0 < s - q(pk \text{ mod } s) < k$ *where* $q = \left\lfloor \frac{s}{pk \text{mod} s} \right\rfloor$.

Proof: $(1, -(pk \text{ mod } s))$ and $(q, s - q(pk \text{ mod } s))$ form a pair of basis vectors of the integer lattice as the determinant of the matrix whose columns are $(1, -(pk \text{ mod } s))$ and $(q, s - q(pk \text{ mod } s))$ is s.

It has been proved earlier that $(1, -(pk \text{ mod } s))$ is an extremal basis vector for integer lattice.

Since $s \geq k$, every course has at most one accessed element. Without loss of generality let us assume that the starting element on a processor is the origin of the two dimensional space. Let us assume that (x, y) belongs to the integer lattice then,

$$\begin{bmatrix} x \\ y \end{bmatrix} = a \begin{bmatrix} 1 \\ -(pk \text{ mod } s) \end{bmatrix} + b \begin{bmatrix} q \\ s - q * (pk \text{ mod } s) \end{bmatrix}$$

Since $(1, -(pk \text{ mod } s))$ is an extremal basis vector for integer lattice, a can take only positive integer values. From the construction of q it is clear that $(q, s - q(pk \text{ mod } s))$ is the first point accessed after the origin. If b were negative then (x, y) should lie on a course numbered less than zero, or should lie on a course whose number is between 0 and q; neither of these is possible. Thus, b can take only positive integer values. Therefore, $(1, -(pk \text{ mod } s))$ and $(q, s - q(pk \text{ mod } s))$ form a pair of extremal basis vectors when $0 < s - (pk \text{ mod } s) < k$. $\square$

The basis matrix for this case is

$$B = \begin{bmatrix} 1 & q \\ -(pk \text{ mod } s) & s - q(pk \text{ mod } s) \end{bmatrix}.$$

The transformation matrix is

$$B^{-1} = \frac{1}{s} \begin{bmatrix} s - q(pk \text{ mod } s) & -q \\ (pk \text{ mod } s) & 1 \end{bmatrix}.$$

As before the loop bounds can be written as follows:

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} 0 \\ \mathcal{C} \\ -mk \\ (k-1) + mk \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} BB^{-1} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} 0 \\ \mathcal{C} \\ -mk \\ (k-1) + mk \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & q \\ -(pk \text{ mod } s) & s - q(pk \text{ mod } s) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \leq \begin{bmatrix} 0 \\ \mathcal{C} \\ -mk \\ (k-1) + mk \end{bmatrix}$$

$$\begin{bmatrix} -1 & -q \\ 1 & q \\ pk \bmod s & -(s-q(pk \bmod s)) \\ -(pk \bmod s) & s-q(pk \bmod s) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \leq \begin{bmatrix} 0 \\ \mathcal{C} \\ -mk \\ (k-1)+mk \end{bmatrix}$$

where $\begin{bmatrix} u \\ v \end{bmatrix} = B^{-1} \begin{bmatrix} i \\ j \end{bmatrix}$ and u and v are integers. Thus,

$$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} u+qv \\ v(s-q(pk \bmod s)) - u(pk \bmod s) \end{bmatrix}.$$

Now using Fourier-Motzkin elimination [22] we solve for u and v as follows:

$$\begin{aligned} -u-qv &\leq 0 \\ u+qv &\leq \mathcal{C} \\ u(pk \bmod s) - (s-q(pk \bmod s))v &\leq -mk \\ -u(pk \bmod s) + (s-q(pk \bmod s))v &\leq (k-1)+mk \end{aligned}$$

Hence,

$$\begin{aligned} \frac{q(-mk-k+1)}{s} \leq \quad & u \quad \leq \frac{\mathcal{C}(s-q(pk \bmod s)) - qmk}{s} \\ \max\left(\frac{mk+(pk \bmod s)u}{s-q(pk \bmod s)}, -\frac{u}{q}\right) \leq \quad & v \quad \leq \min\left(\frac{mk+k-1+(pk \bmod s)u}{s-q(pk \bmod s)}, \frac{\mathcal{C}-u}{q}\right) \end{aligned}$$

The pattern generation code for processor m is shown next.

$i = 0$;
do $u = \left\lceil \frac{q(-mk-k+1)}{s} \right\rceil, \left\lfloor \frac{\mathcal{C}(s-q(pk \bmod s))-qmk}{s} \right\rfloor$
  do $v = \left\lceil \max\left(\frac{mk+(pk \bmod s)u}{s-q(pk \bmod s)}, -\frac{u}{q}\right)\right\rceil, \left\lfloor \min\left(\frac{mk+k-1+(pk \bmod s)u}{s-q(pk \bmod s)}, \frac{\mathcal{C}-u}{q}\right)\right\rfloor$
    $pattern_m[i] = k(u+qv) + v(s-q(pk \bmod s)) - u(pk \bmod s) - mk$
    Increment i
  end do
end do

4.2.4 COMPILE TIME GENERATION OF CODE FOR SEVERAL SPECIAL CASES

Our implementation recognizes several special cases where closed form expressions for the sequence of local addresses exist. These special cases include $p = 1$, $k = 1$ (`CYCLIC`), $k = \frac{N}{p}$ (`BLOCK`), $s = 1$, s divides k, k divides s, s divides p, s divides pk, and pk divides s.

4.2.5 Discussion

In an experiment [17] with p (the number of processors) in powers of 2 from 1 to 1024, k (block size) in powers of 2 from 1 to 16384, and s (the access stride) ranging from 1 to 99 in unit increments, we derived the optimal basis vectors for over 91% of the problem instances.

If we cannot find a q such that $0 < s - q(pk \bmod s) < k$, then we use $q = \left\lfloor \frac{s}{pk \bmod s} \right\rfloor$. In this case, we do not have the best pair of basis vectors. Nevertheless, the code we generate is correct. When $s \geq pk$, we can apply further optimizations to the algorithms presented in this section.

4.3 Experimental Results

We performed two sets of experiments on our algorithm on an IBM RS-6000. We used the `cc` compiler with `-O3` level optimizations turned on; the function `gettimeofday()` was used to measure time. When computing time for 32 processors, we timed the code that computes the access pattern for each processor, and report the maximum time over all processors. In the first set of experiments, we varied strides from 2 to 13 and block sizes ranging from 1 to 64, for 32 processors. The overhead of computing the access pattern is very small, compared to the message start-up costs typically encountered in current distributed memory message passing machines (such as the IBM SP-2). In all figures, whenever one of the special cases listed in Section 4.2.4 is encountered, the pattern generation time reduces significantly. For example, whenever k is a multiple of s, the address pattern generation time drops significantly. In addition, for a power of 2 stride (when $s \leq k$) the address pattern generation time is a linear in the block size with a slope near zero.

Next we compared our implementation that computes the access pattern to the implementation from Chatterjee *et al.* [3] referred to as RT in Figs. 4.6 thru 4.11, and in Table 4.1. The plots in these figures are on a log-log scale. From the data in these figures and in Table 4.1, it is clear that our approach outperforms the RT approach. With increasing values of block sizes, the difference in performance is very high. The overhead of sorting in the RT approach seems to dominate in these cases. We have observed a similar performance difference between our solution in this paper and the solution of Kennedy *et al.* [10]. The reader is referred to [21] for detailed results.

In Fig. 4.6, the address generation time drops as we increase k from 4 to 8, since the number of comparisons needed in the loop nest for $k = 4$ using basis vectors $\{(1,5),(1,-2)\}$ is higher than the loop nest for $k = 8$ which uses the basis vectors $\{(1,-4),(0,7)\}$. Fig. 4.7 illustrates the impact of using the best basis vectors on the time needed to determine the address pattern. For $k = 4$ and $k = 8$, our solution gives only an approximation to the best basis vectors; hence the time needed is slightly higher than the approach of [3]. For larger values of k the pattern derivation time actually drops and then slowly increases, since our approach derives the best pair of basis vectors for those cases resulting in lower pattern derivation time. The address pattern generation times for our

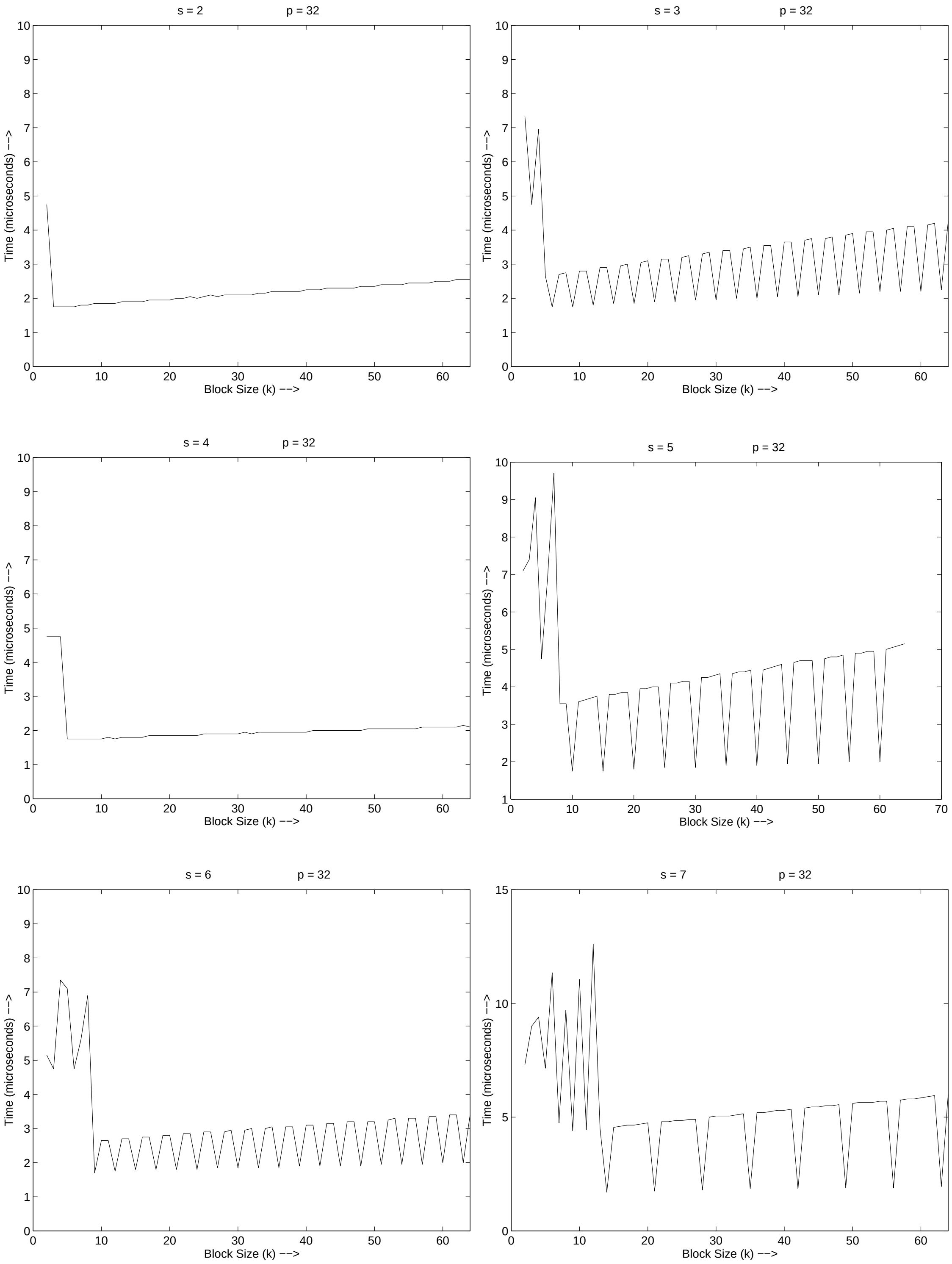

Figure 4.4: Effect of stride $(2 \cdots 7)$ and block size $(2 \cdots 64)$ for $p = 32$ processors

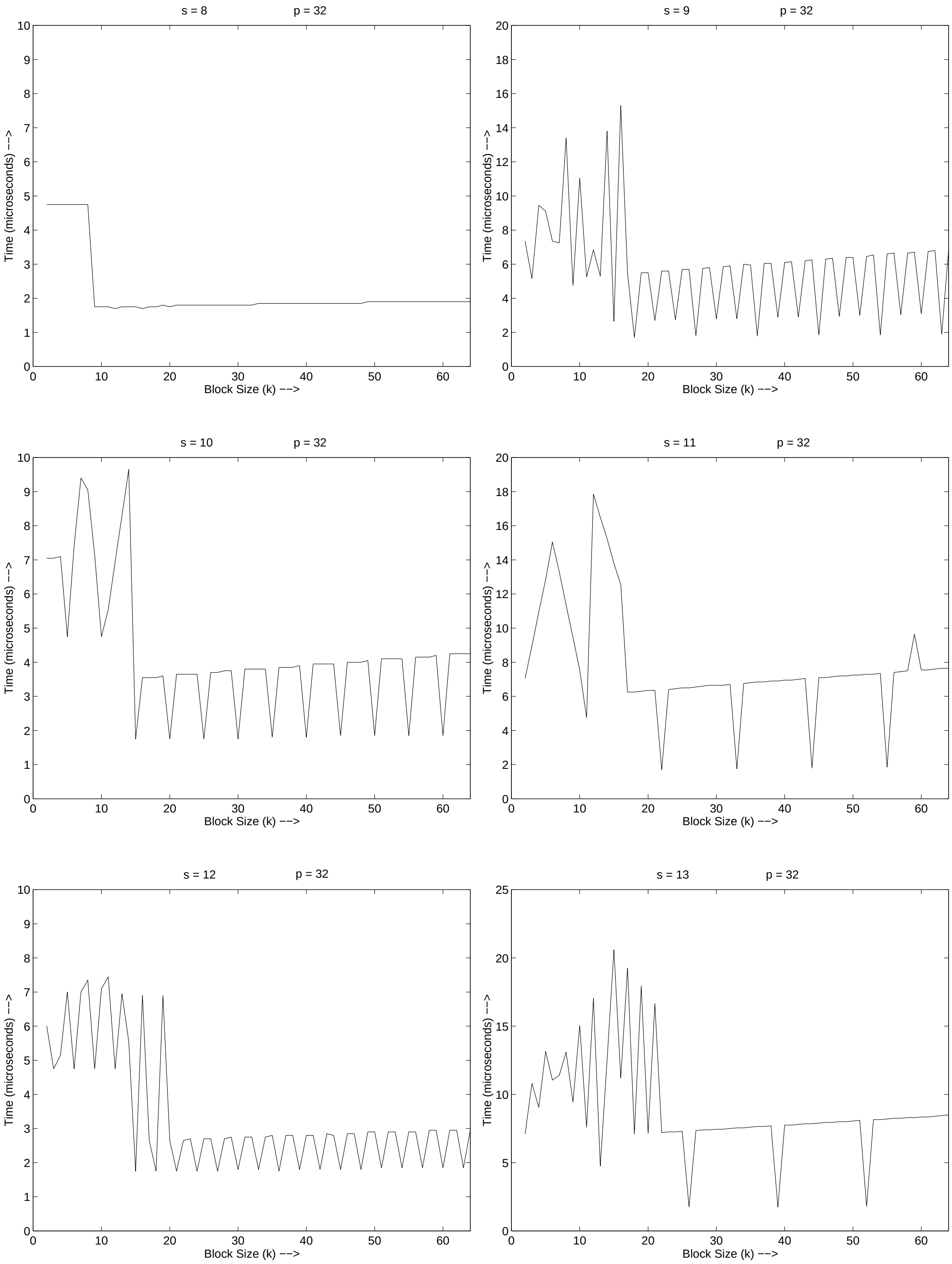

Figure 4.5: Effect of stride $(8 \cdots 13)$ and block size $(2 \cdots 64)$ for $p = 32$ processors

approach are much lower than that of [3] in Figs. 4.8 and 4.9. The number of iterations of the outer loop in the pattern generation algorithm decreases from $k = 8$ to $k = 32$ and then increases; this explains the drop observed in these figures for our approach. The best pair of basis vectors that we derive for $s = pk - 1$ and $s = pk + 1$ results in remarkably low pattern determination time as in evident in Figs. 4.10 and 4.11.

4.4 SUMMARY

In this chapter we derived closed form expressions for the basis vectors of the array section lattice. We identified all the problem instances for which closed form expressions are known. For the remaining cases, we also showed that closed form expressions could be used to obtain a pair of basis vectors closest to the best pair of basis vectors. Experimental results show this solution to the address generation problem to be faster than any solution that resorts to a runtime evaluation of basis vectors.

Table 4.1: Address pattern generation times (in microseconds) of (CF - closed form solutions) our approach and the (RT - run-time solutions) Riacs approach for 32 processors on an IBM RS6000

	s = 7		s = 99		s = k-1		s = k+1		s = pk-1		s = pk+1	
k	CF	RT	CF	RT	CF	RT	CF	RT	CF	RT	CF	RT
4	9	15	25	16	7	15	9	15	7	14	5	13
8	10	25	79	27	10	25	13	25	7	24	5	24
16	5	45	32	47	7	46	10	46	8	45	6	44
32	5	95	65	89	6	85	8	85	8	85	6	85
64	6	168	67	174	49	173	68	171	9	169	7	167
128	8	341	55	349	47	344	67	340	11	383	9	341
256	11	764	49	756	49	699	72	688	15	726	13	776
512	18	1564	57	1487	53	1477	79	1584	23	1407	21	1552
1024	31	3243	71	3162	61	3443	96	3413	38	3032	36	3173

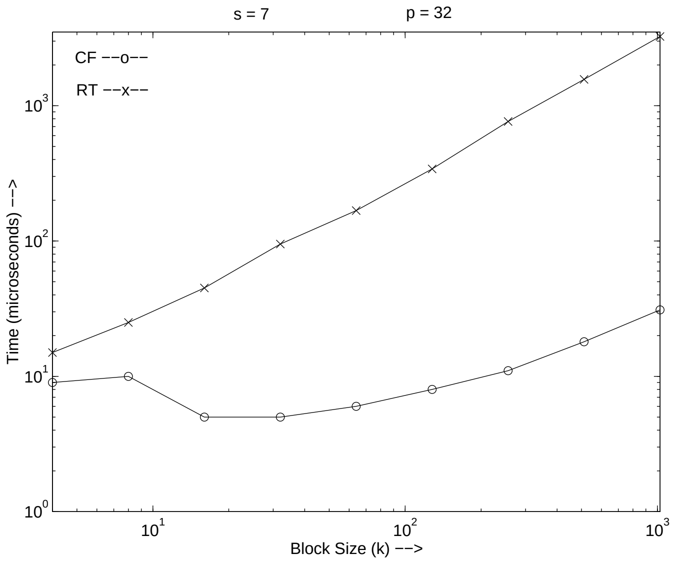

Figure 4.6: Comparison of CF to RT for $s = 7$ for $p = 32$ processors

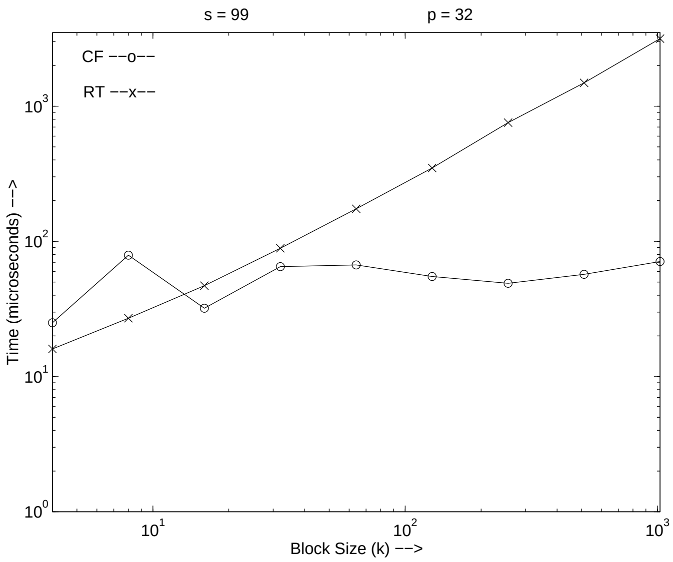

Figure 4.7: Comparison of CF to RT for $s = 99$ for $p = 32$ processors

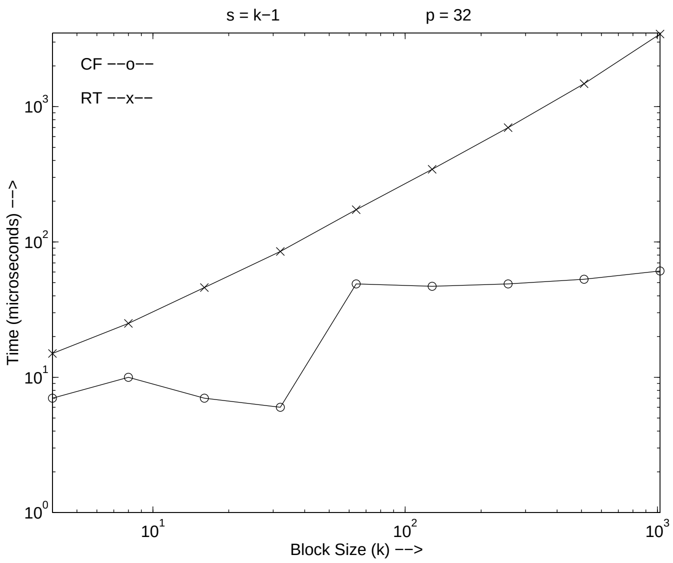

Figure 4.8: Comparison of CF to RT for $s = k - 1$ for $p = 32$ processors

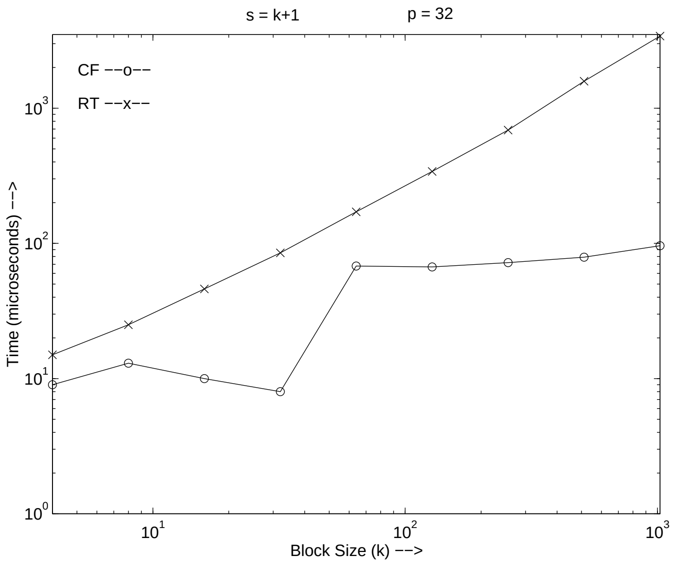

Figure 4.9: Comparison of CF to RT for $s = k + 1$ for $p = 32$ processors

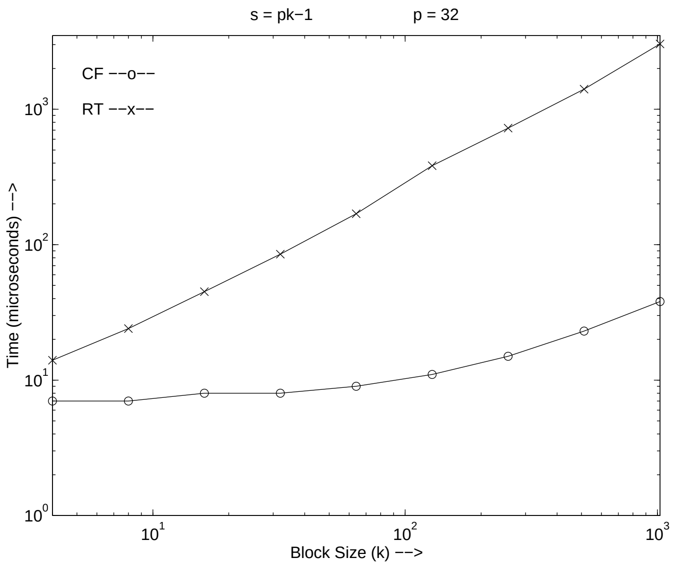

Figure 4.10: Comparison of CF to RT for $s = pk - 1$ for $p = 32$ processors

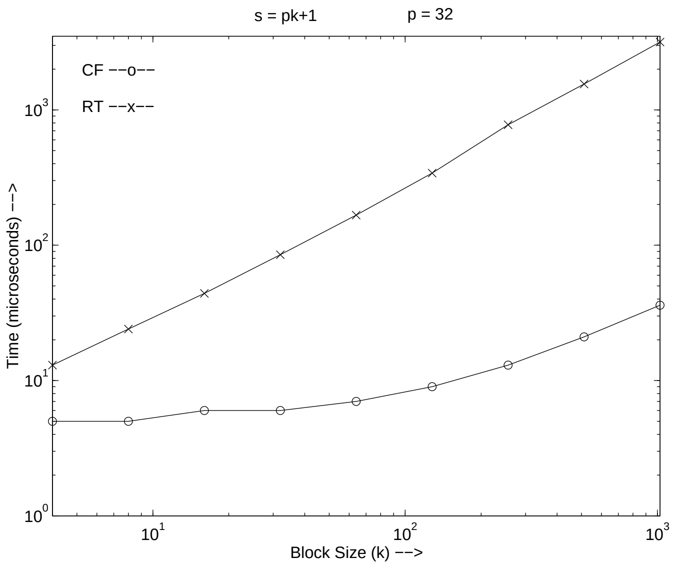

Figure 4.11: Comparison of CF to RT for $s = pk + 1$ for $p = 32$ processors

CHAPTER 5

COMPUTING COMMUNICATION SETS

Communication involves sending those data items that a given processor owns and is required by other processors, and receiving data sent by other processors. We first show that there is a pattern in the communication sets of each processor. We then present algorithms for deriving the communication patterns for each processor. The send pattern is then used to pack array elements to be sent to other processors. We then present an algorithm that uses the receive pattern to access the nonlocal elements right out of the temporary receive buffer. We also provide two algorithms for unpacking data received form other processors into local storage allocated. These unpacking algorithms are generalizations of the idea of over lap areas. A typical HPF-style assignment statement is:

$$A(l_1 : h_1 : s_1) = B(l_2 : h_2 : s_2) \tag{5.1}$$

5.1 RECEIVE PATTERN GENERATION

Given the global address of a LHS element, we can easily determine the processor that owns the corresponding RHS element and the local address of the RHS element on that processor using the following expressions,

$$m = (i \text{ div } k) \text{ mod } p \qquad\qquad L = k * (i \text{ div } pk) + i \text{ mod } k,$$

where i is a global address, m is the processor to which i belongs, and L is the local address of i on processor m.

It should be evident that the access pattern in the global address space repeats itself after pk elements are accessed, *i.e.*, after every s_1^{th} row where s_1 is the access stride of the LHS array. To generate the receive pattern, all we need to do is run the code generated in the previous sections 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 and local address of that element on it. The code is shown below.

$\alpha = pk \text{ mod } (s_1)$
$k' = k - 1$
$\beta = m * k - l_1$
$q = 0$
do u = $\frac{l_1}{pk}, s_1 - 1 + \frac{l_1}{pk}$
  $t = \alpha u + \beta$
  do v = $\left\lceil \frac{t}{s_1} \right\rceil, \left\lfloor \frac{k'+t}{s_1} \right\rfloor$
    $i = \frac{(upk+mk+vs_1-t-l_1)s_2}{s_1} + l_2$
    $RPattern_m[q] = k * (i \text{ div } pk) + i \text{ mod } k$

$RProcpattern_m[q] = (i \text{ div } k) \text{ mod } p$
Increment q
end do
end do

5.2 SEND PATTERN GENERATION

Our approach to send pattern generation is very similar to the receive pattern generation shown in the previous section. 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^{th} row. The code that generates the send pattern is shown below.

$\alpha = pk \text{ mod } (s_2)$
$k' = k - 1$
$\beta = m * k - l_2$
$q = 0$
do u = $\frac{l_2}{pk}, s_2 - 1 + \frac{l_2}{pk}$
  $t = \alpha u + \beta$
  do v = $\left\lceil \frac{t}{s_2} \right\rceil, \left\lfloor \frac{k'+t}{s_2} \right\rfloor$
    $i = upk + mk + vs_2 - t$
    $SPattern_m[q] = (k * (i \text{ div } pk) + i \text{ mod } k)$
    $i = \frac{(i-l_2)s_1}{s_2} + l_1$
    $SProcpattern_m[q] = (i \text{ div } k) \text{ mod } p$
    Increment q
  end do
end do

Example 5.1 Here we show the communication set generation for the array assignment statement, $A(0 : h_1 : 5) = B(0 : h_2 : 3)$ where A and B are aligned identically with the template T, and $p = 4$ and $k = 8$.

u	$v_{lb} = \left\lceil \frac{2u}{3} \right\rceil$	$v_{ub} = \left\lfloor \frac{2u+7}{3} \right\rfloor$	Send global index B	To Processor
0	0	2	0, 3, 6	0, 0, 1
1	1	3	33, 36, 39	2, 3, 0
2	2	3	66, 69	1, 2

The LHS access pattern repeats after five courses and the RHS access pattern repeats after one course. The send receive pattern repeats after five courses in this case. The send pattern for processor 0 is generated as follows:

do $u = 0, 2$
  $t = 2u$
  do $v = \lceil \frac{t}{3} \rceil, \lfloor \frac{7+t}{3} \rfloor$
    $i = 32u + 3v - t$
    $SPattern_0[q] = (8 * (i \text{ div } 32) + i \text{ mod } 8)$
    $i = \frac{i*5}{3}$
    $SProcpattern_0[q] = (i \text{ div } 8) \text{ mod } 4$
    Increment q
  end do
end do

The receive pattern for processor 0 is generated as follows:

do $u = 0, 4$
  $t = 2u$
  do $v = \lceil \frac{t}{5} \rceil, \lfloor \frac{7+t}{5} \rfloor$
    $i = \frac{(32u+5v-t)*3}{5}$
    $RPattern_0[q] = (8 * (i \text{ div } 32) + i \text{ mod } 8)$
    $RProcpattern_0[q] = (i \text{ div } 8) \text{ mod } 4$
    Increment q
  end do
end do

u	$v_{lb} = \lceil \frac{2u}{5} \rceil$	$v_{ub} = \lfloor \frac{2u+7}{5} \rfloor$	Receive global index B	From Processor
0	0	1	0, 3	0, 0
1	1	1	21	2
2	1	2	39, 42	0, 1
3	2	2	60	3
4	2	3	78, 81	1, 2

Finally we use the send and receive pattern information in the communication algorithms shown in the next section.

5.3 Communication Algorithm

In this section we present an algorithm to pack data to be sent from one processor to another using the send pattern generated as shown in Section 5.2. We then present three different schemes to use the data that is received by a processor. Section 5.3.2 discusses an approach that accesses non-local array elements directly out of the buffers in which they are received by a processor. Section 5.3.3 and Section 5.3.4 present algorithms that unpack the contents of the receive buffers into overlap areas. All the unpacking algorithms discussed here use the receive pattern generated as shown in Section 5.1.

Processor 0

$\boxed{0}$	1	2	3	4	$\boxed{5}$	6	7
32	33	34	$\boxed{35}$	36	37	38	39
64	$\boxed{65}$	66	67	68	69	$\boxed{70}$	71
96	97	98	99	$\boxed{100}$	101	102	103
128	129	$\boxed{130}$	131	132	133	134	$\boxed{135}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(a): Global addresses of the accessed elements of array A for $s_1 = 5$.

Processor 0

$\boxed{0}$	1	2	$\boxed{3}$	4	5	6	7
32	33	34	35	36	37	38	$\boxed{39}$
64	65	66	67	68	69	70	71
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Processor 1

8	9	10	11	12	13	14	15
40	41	$\boxed{42}$	43	44	45	46	47
72	73	74	75	76	77	$\boxed{78}$	79
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Processor 2

16	17	18	19	20	$\boxed{21}$	22	23
48	49	50	51	52	53	54	55
80	$\boxed{81}$	82	83	84	85	86	87
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Processor 3

24	25	26	27	28	29	30	31
56	57	58	59	$\boxed{60}$	61	62	63
88	89	90	91	92	93	94	95
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(b): Global addresses of array B elements accessed by processor 0 for $s_2 = 3$.

Figure 5.1: Access patterns for LHS and RHS elements for $A(\ : : 5) = B(\ : : 3)$, $p = 4$ and $k = 8$.

We assume that each processor maintains a separate send and receive buffer for all the p processors.

5.3.1 Packing the Send Set

Using the send pattern generated in Section 5.2, we can construct the set of elements that a processor needs to send to all the other processors by making a single pass over the array. This results is a substantially low message packing overhead. The packing algorithm presented here could be used for all the unpacking algorithms discussed later after making slight modifications to the send pattern generation. The send pack algorithm for the m^{th} processor is as shown below.

1. Run the Send loop as shown in Section 5.2. The elements generated by this loop are stored in *Spattern*, the pattern of the processors to which these elements have to be sent is stored in *SProcpattern*, and the array elements to be sent to processor i are stored in $\mathit{SendBuffer}_i$ for each processor $i = 0, p-1$.

2. For processor m, we run the following code:

   $\mathit{Scount}_q = 0$, where $0 \leq q \leq p-1$ and Scount_q is an offset into $\mathit{SendBuffer}_q$ corresponding to the q^{th} processor.
   do $j = 0, \frac{max_2}{s_2 k}$
       do $i = 0, \mathit{sizeof}(\mathit{Spattern}) - 1$
           $q = \mathit{SProcpattern}[i]$
           $\mathit{SendBuffer}_q[\mathit{Scount}_q] = B_{loc}[j * k + \mathit{Spattern}[i]]$
           $\mathit{Scount}_q + +$
       enddo
   enddo

   Note: max_2 is the local address of the last element of B accessed on the processor.

3. Send data in buffers to the respective processors.

5.3.2 Working out of the Buffer

We assume that a processor maintains a separate receive buffer for each of the p processors. Instead of allocating local storage for the non-local elements that are accessed in the array assignment and then having to unpack the received non-local elements, we present an algorithm in which we work directly out of the receive buffers; an access to a non-local element is treated as an access to the receive buffer corresponding to the processor that owns this element. The receive and execute algorithm for the m^{th} processor in this scheme is shown below. Note that max_1 is the local address of the last element of A accessed on processor m.

1. Receive: data from the p processors

2. Run the Receive loop as shown in Section 5.1. We assume that elements received are stored in *Rpattern* and array elements received from processor i is stored in $ReceiveBuffer_i$ $(0 \leq i \leq p-1)$. *Apattern* holds A's access pattern.

3. Execute:

$Rcount_q = 0$, where $0 \leq q \leq p-1$ and $Rcount_q$ is an offset into $ReceiveBuffer_q$ corresponding to the q^{th} processor.

```
do j = 0, max_1/(s_1 k)
    do i = 0, sizeof(Rpattern) - 1
        q = RProcpattern[i]
        if m = q
            A_loc[j k s_1 + Apattern[i]] = B_loc[Rpattern[i]]
        else
            A_loc[j k s_1 + Apattern[i]] = ReceiveBuffer_q[Rcount_q]
            Rcount_q ++
        endif
    enddo
enddo
```

5.3.3 Unpacking Data into Padded Courses

Unlike working out of the buffer, in this case we allocate local storage for the non-local RHS elements that are accessed. As explained earlier on in this chapter, the communication pattern for an array assignment statement repeats after pk elements are accessed. Hence the access pattern for the LHS array repeats after s_1 courses while that of the RHS array repeats after s_2 courses. On each processor we make at most k LHS accesses before the pattern repeats. In the worst case all assignments to LHS array elements may require non-local RHS elements. So we need to allocate space for k elements for each block of s_2 RHS courses. As a single course on a processor holds k elements, all we need to do is insert (pad) a single course to hold the non-local RHS elements after every s_2 courses. The number of courses that are padded to the RHS array using this approach is $\frac{\lceil \frac{h-l}{pk} \rceil}{s_2}$.

Fig. 5.2 shows an example of padding an array with courses. Padding courses in the RHS array results in a change in the local addresses of the array that are accessed. Hence the receive and execute algorithm for each processor is as shown below. Note that max_1 is the local address of the last element of A accessed on processor m.

1. Receive: data from the p processors

2. Run the Receive loop as shown in Section 5.1. We assume receive pattern generated by this loop is stored in *Rpattern*, while the pattern of the processors form which these elements are to be received is stored in *RProcpattern*. Array elements received

Processor 0

$\boxed{0}$	1	2	$\boxed{3}$	4	5	6	7
32	33	34	35	36	37	38	$\boxed{39}$
64	65	66	67	68	69	70	71
0	3	39	$\boxed{42}$	$\boxed{78}$	$\boxed{21}$	$\boxed{81}$	$\boxed{60}$
from Proc 0			from Proc 1		from Proc 2		from Proc 3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a): Access pattern using global addresses.

Processor 0

$\boxed{0}$	1	2	$\boxed{3}$	4	5	6	7
8	9	10	11	12	13	14	$\boxed{15}$
16	17	18	19	20	21	22	23
24	25	26	$\boxed{27}$	$\boxed{28}$	$\boxed{29}$	$\boxed{30}$	$\boxed{31}$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(b): Access pattern using local addresses.

Figure 5.2: Access patterns for RHS elements on processor 0 (using padded courses) for $A(\,:\,:5) = B(\,:\,:3)$, $p = 4$ and $k = 8$.

from processor i is stored in *ReceiveBuffer*$_i$ for each processor $i = 0, p-1$. *Apattern* holds A's access pattern.

3. Determine the number of elements to be received from each processor before the pattern repeats, from *RProcpattern*. Store the count corresponding to each processor in *count*$_i$ for each $i = 0, p-1$.

4. Unpack:

do $j = 0, \frac{max_1}{s_1 k}$
    do $i = 0, p-1$
        do $h = 0, count_i$
            $B_{loc}[(j+1)ks_2 + jk + count_{i-1} + h] = ReceiveBuffer_q[j * count_i + h]$
        enddo
    enddo
enddo

5. Execute:

Rcount$_q$ $=$ *count*$_q$, where $0 \leq q \leq p-1$ and *Rcount*$_q$ is an offset into the padded course corresponding to the q^{th} processor.
do $j = 0, \frac{max_1}{s_1 k}$
    do $i = 0, sizeof(Rpattern) - 1$
        $q = RProcpattern[i]$
        if m = q
            $A_{loc}[jks_1 + Apattern[i]] = B_{loc}[Rpattern[i]]$
        else
            $A_{loc}[jks_1 + Apattern[i]] = B_{loc}[(j+1)ks_2 + jk + Rcount_q]$
            $Rcount_q++$
        endif
    enddo
    Reset *Rcount*$_q$ to *count*$_q$ for all q.
enddo

The implementation of the above algorithm includes several optimizations (like strength reduction, loop unrolling *etc.*) which have been omitted from the discussion here to make the algorithm easy to understand. The total memory wastage of this storage algorithm (on processor m) is $\frac{\lceil \frac{h-l}{pk} \rceil}{s_2} \times$ number of local_elements of m.

5.3.4 UNPACKING DATA INTO PADDED COLUMNS

Here again we allocate local storage for non-local RHS elements that are accessed. Unlike the previous unpacking strategy, the extra local storage is in the form of extra columns added as shown in the Fig. 5.3. The number of columns that are padded to the RHS array is $\lceil \frac{k}{\min(s_1,s_2)} \rceil$, *i.e.*, storage for at least k extra elements is allocated. Thus the total amount of extra storage provided in this unpacking scheme is $s_2 \times \lceil \frac{k}{\min(s_1,s_2)} \rceil$.

5.4 DISCUSSION AND EXPERIMENTS

As we have assumed a single template model, closed form expressions for the communication set generation exist for the case where $k = 1$ (template is cyclically distributed) and $k = \left\lceil \frac{N}{p} \right\rceil$ (template is block distributed) [12]. The algorithms shown in Section 5.3 can be simplified for these cases as well as the case where, s_{lhs} divides s_{rhs} or vice-versa, as the send and receive set patterns repeat after $\frac{s}{\text{GCD}\,(s,pk)}$ rows itself.

Communication pattern generation times on an IBM SP-2 are shown in Table 5.1 on p. 67. Our code was compiled using the `mpcc` compiler with `-O3` optimizations turned on. These times include the time it takes for address generation for both the left and right hand side array sections.

5.5 SUMMARY

In this chapter we presented communication generations algorithms that require a single scan of all the right hand side elements on a processor to generate the send sets. We presented unpacking algorithms, based on a generalization of overlap areas, that maintain data locality and hence perform better than the algorithms presented by Kennedy *et al.* [11]. We also showed how to use the receive pattern to access non-local array elements right out of the the temporary buffers they are received in. This altogether eliminates the unpacking overhead.

Processor 0

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

(a): Access pattern using global addresses.

Processor 0

$\boxed{0}$	1	2	$\boxed{3}$	4	5	6	7	8	9	$\boxed{10}$
11	12	13	14	15	16	17	$\boxed{18}$	19	$\boxed{20}$	$\boxed{21}$
22	23	24	25	26	27	28	29	$\boxed{30}$	$\boxed{31}$	32
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(b): Access pattern using local addresses.

Figure 5.3: Access patterns for RHS elements on processor 0 (using padded columns) for $A(\,:\,:5) = B(\,:\,:3)$, $p = 4$ and $k = 8$.

Table 5.1: Communication pattern generation times

k	p	s_1	s_2	Comm. Gen. Time (microseconds)
4	3	5	3	41
4	3	3	5	40
8	4	3	5	49
8	4	5	3	49
8	32	7	99	84
8	32	99	7	84
16	32	7	99	128
16	32	99	7	127
32	32	7	99	240
32	32	99	7	228

CHAPTER 6

CONCLUSION

The extreme difficulty of writing correct and efficient programs is the major obstacle to the widespread use of parallel high-performance computing. The success of data parallel languages such as High Performance Fortran critically depends on efficient compiler and runtime support. In this thesis we presented efficient compiler algorithms for generating local memory access patterns and interprocessor communication sets for the various processors (node code) given the alignment of arrays to a template and a `CYCLIC(k)` distribution of the template onto the processors. Our solution is based on viewing the access sequence as an integer lattice, and involves the derivation of a suitable set of basis vectors for the lattice.

The basis vector determination algorithm presented in Chapter 3 runs in $O(k)$ time, where k is the block size. Kennedy *et al.*'s $O(k)$ algorithm for basis vector determination requires a solution of $2k - 1$ solutions of linear Diophantine equations, as opposed k solutions that our algorithm requires. In addition, k solutions is a lower bound. We also presented an improved algorithm that required only two solutions of Diophantine equations. Our pattern generation algorithm (using basis vectors) uses a loop nest to enumerate points in the array section lattice on a processor. This results in a reduction in the number of comparisons. Experiments on several platforms indicate the efficacy of our approach. In addition, our solution to the address generation problem outperforms the approach of [10] as well as [3].

In Chapter 4 we presented efficient compile-time algorithms for generating local memory access patterns for array statements. We derived closed form expressions for the basis vectors of the array section lattice. This allows us to determine the access pattern efficiently at run-time; this avoids the runtime overhead for basis vector generation incurred by the most efficient of the runtime approaches [10, 20]. Experiments on several platforms indicate that evaluating the closed form expressions at run-time performs significantly better than any other existing approach. The difference in performance between our approach and the other solutions to the address generation problem is more pronounced at large values of block size.

In Chapter 5 we presented techniques to generate communication sets for data parallel programs [18] by exploiting the fact that the array access patterns repeat. Experiments show that the packing and unpacking algorithms compare favorably to the communication generation algorithm using overflow regions presented by Kennedy *et al.* [11]. We expected such results as cache misses hurt Kennedy *et al.*'s more than it hurts our unpacking strategies (that localizes RHS accesses). We also notice a substantial performance improvement over the virtual processor approach proposed by Gupta *et al.* [7].

Further work needs to be done to extend the address generation and communication algorithms presented in this thesis towards minimizing memory wastage in the case of non-unit alignment strides. Techniques for incremental generation of access sequences and communication sets could also be explored. Further work could also be done on deriving efficient communication generation algorithms that support message coalescing and handle several special cases like shift communication. A logical extension to our approach is to perform similar analysis for multi-dimensional array statements for generalized alignments.

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] 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.

[3] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data parallel programs. In *Proc. of ACM Symposium on Principles and Practices of Parallel Programming*, pages 149–158, May 1993.

[4] G. Fox, S. Hiranandani, K. Kennedy, C Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D Language Specification. Technical Report TR-91-170, Dept. of Computer Science, Rice University, Dec. 1991.

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

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

[7] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical Report OSU-CISRC-94-TR19, Department of Computer and Information Science, The Ohio State University, April 1994.

[8] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[9] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Advanced compilation techniques for Fortran D. Technical Report CRPC-TR-93-338, Center for Research on Parallel Computation, Rice University, Oct. 1993.

[10] 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.

[11] 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 (Editors), Kluwer Academic Publishers, 1995.

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

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

[14] 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.

[15] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing 92,* pages 214–223, Nov. 1992. An expanded version of this paper titled "Beyond Unimodular Transformations," will appear in *The Journal of Supercomputing,* 1995.

[16] 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.

[17] 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.

[18] 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, 1995.

[19] 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.

[20] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs Using Integer Lattices. To appear in *Languages and Compilers for Parallel Computing,* U. Banerjee *et al.* (Editors), Lecture Notes in Computer Science, Springer-Verlag, 1995.

[21] 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.

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

VITA

Ashwath Thirumalai was born in Bombay, India on April 24, 1971. After completing his high school, he joined the Bachelor of Engineering program in Bangalore University, India in June 1988. He received his bachelor's degree in Electronics in Spring 1992.

Ashwath joined the M.S. program in the Louisiana State University in Spring 1993. During the first year of his stay at LSU he developed an active interest in compiler construction, compiler optimizations and code generation for data parallel languages. He has written book chapters and conference papers in this research area. He is a candidate for M.S. in Electrical Engineering, LSU at the Summer 1995 commencement.

Thirumalai, Ashwath, B.E. in Electronics, Bangalore University, 1992
Master of Science in Electrical Engineering, Summer Commencement, 1995
Major: Electrical Engineering
Code Generation and Optimization for High Performance Fortran
Thesis directed by Assistant Professor Jagannathan Ramanujam
Pages in thesis, 80. Words in abstract, 296.

ABSTRACT

In data-parallel languages such as High Performance Fortran and Fortran D, arrays are mapped to processors through a two step process of alignment followed by distribution. This process of mapping creates disjoint pieces of the array that are locally owned by each processor. A compiler that generates code for each processor has to compute the sequence of local memory addresses accessed by each processor and the sequence of sends and receives for a given processor to access non-local data. This thesis presents a novel approach based on integer lattices for the address generation problem. The set of accesses can be expressed as integer linear combinations of basis vectors. A linear time algorithm that computes the basis vectors as a function of the mapping is presented. Using these basis vectors, a loop nest that enumerates the addresses as points in the lattice generated by the basis vectors is derived. Experimental results presented here demonstrate that this approach is superior to other solutions. In addition to this runtime technique, closed form expressions for basis vectors as functions of the data mapping are derived. This precludes the need for expensive runtime computation of basis vectors, requiring only the enumeration of addresses using a loop nest. Algorithms for deriving the communication patterns for each processor are also presented. These patterns are used in packing data to be sent. In addition, three different techniques are presented for accesses to non-local data. The first algorithm accesses the received data directly out of the buffers. The other two strategies are generalizations of overlap regions that allocate storage for the non-local data as well. The received elements are unpacked into this memory allocated. Experiments show that the communication generation times incurred by these techniques are negligible compared to message passing overheads on current machines.