Integer Lattice Based Methods for Local Address Generation for Block-Cyclic Distributions

J. Ramanujam

Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, Louisiana, USA

Summary. In data-parallel languages such as High Performance Fortran and Fortran D, arrays are mapped to processors through a two-step process involving alignment followed by distribution. 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. In this chapter, we present a novel approach to the address sequence generation problem based on integer lattices. When the alignment stride is one, the mapping is called a *one-level mapping*. In the case of one-level mapping, the set of elements referenced can be generated by integer linear combinations of basis vectors. Using the basis vectors we derive a loop nest that enumerates the addresses, which are points in the lattice generated by the basis vectors. The basis determination and lattice enumeration algorithms are linear time algorithms. For the *two-level mapping* (non-unit alignment stride) problem, we present a fast novel solution that incurs zero memory wastage and little overhead, and relies on two applications of the solution of the one-level mapping problem followed by a fix-up phase. Experimental results demonstrate that our solutions to the address generation problem are significantly faster than other solutions to this problem. In addition, we present a brief overview of our work on related problems such as communication generation, basis vector derivation, code generation for complex subscripts and array redistribution.

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 behind efforts such as High Performance Fortran (HPF) [13, 20], Fortran D [10], and Vienna Fortran (which grew out of the earlier SUPERB effort) [4] is to raise the level of programming on distributed memory machines while retaining the object code efficiency derived for example from message passing.

These languages include directives—such as align and distribute—that describe how data is distributed among the processors in a distributed-memory multiprocessor. For example, arrays in HPF are mapped to processors in two

steps: 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 for each processor (called node code) that accesses only the locally owned pieces directly, and inserts communication for non-local accesses.

In order to generate node code for each processor, 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. In this chapter we present efficient algorithms for generating local memory access patterns for the various processors given the alignment of arrays to a template and the distribution of the template onto the processors. For the case where the arrays are aligned identically to the template (also called *one-level mapping*), our solution [39] 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. [5] presented an $O(k \log k + \log \min(s, pk))$ algorithm (where k is the block size – see Section 2 for definition) for this problem; ours is an $O(k + \log \min(s, pk))$ algorithm. Recently, Kennedy et al. [16] have also presented an $O(k + \log \min(s, pk))$ algorithm. Note that all these algorithms require computing the $\gcd(s, pk)$ which is the reason for the $\log \min(s, pk)$ term in the complexity. Experiments demonstrate that our algorithm is 2 to 9 times faster than the algorithm of Kennedy *at al.* and 13 to 60 times faster than the algorithm in Chatterjee et al. Independently, Wang et al. conclude based on extensive experiments that "The LSU algorithm consistently outperforms the RIACS and Rice algorithm ..." [45]. Our solution to the address generation problem for alignment followed by distribution (i.e., two-level mapping) uses two applications of our solution to the one-level mapping problem followed by an efficient and novel fix-up phase. This second phase is up to 10 times faster than other current solutions that do not waste local memory.

This chapter is organized as follows. In Section 2, we present the problem setting and discuss related work. Section 3 thru 7 discuss one-level mapping in detail, while Sections 8 thru 10 address two-level mapping. Section 3 outlines our approach using lattices and presents key mathematical results which are used later in the chapter. In Section 4, we present our linear algorithm for determining basis vectors, and contrast it with the algorithm of Kennedy et al. In Section 5 we show how to determine address sequences by lattice enumer-

Processor 0			Processor 1			···	Processor p-1		
0	···	k-1	k	···	2k-1	···	(p-1)k	···	pk-1
pk	···	(p+1)k-1	(p+1)k	···	(p+2)k-1	···	(2p-1)k	···	2pk-1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a): Layout of an array distributed `CYCLIC`(k) onto p processors.

Global index	mk	···	mk+k-1	(p+m)k	···	(p+m+1)k-1	···
Local index	0	···	k-1	k	···	2k-1	···

(b): Local layout of array shown in Fig. 1.1(a) in Processor m.

Fig. 1.1. Global and local addresses for data mappings

ation using loop nests. We show how to use the lattice basis vectors derived in Section 4 to generate a loop nest that determines the address sequence. Section 6 discusses optimizations applied to the loop enumeration strategy presented in Section 5, specifically the `GO-LEFT` and `GO-RIGHT` schemes. Section 7 demonstrates the efficacy of our approach using experimental results comparing our solution to those of Chatterjee et al. and Kennedy et al. In Section 8 we introduce the two-level mapping problem in detail; we several new solutions to the two-level mapping problem in Section 9 and provide experimental results for this case in Section 10. In addition to these problems, our research group has addressed several additional problems in code generation and optimization such as communication generation, code generation for complex subscripts, runtime data structures, and runtime array redistribution; a brief outline of these is presented in Section 11. Section 12 concludes with a summary.

2. Background and Related Work

We consider an array A identically aligned to the template T; this means that if $A(i)$ is aligned with the template cell $T(ai + b)$, then the alignment stride a is 1 and the alignment offset b is 0. Further let template T be distributed in a block-cyclic fashion with a block size of k across p processors; this is also known as a `CYCLIC`(k) distribution [13]. 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 Figure 1.1(a). Though the elements of A are stored in the processor memories in a linear fashion as shown in Figure 1.1(b), we adopt a two-dimensional view of the storage allocated for the array as shown in Figure 1.1(a). We view the global addresses as being organized in terms of courses, each course consisting

Table 2.1. Symbols used in this chapter.

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
ℓ	lower bound of regular section of A
h	upper bound of regular section of A
s	stride of regular section of A
$A(\ell : h : s)$	a regular section of array A (array section)
A_{2D}	two-dimensional view of an array A
A_{loc}	local portion of array A allocated on a processor
m	processor number ($0 \leq m \leq p-1$)
Γ	array section lattice
Γ_m	part of Γ incident on processor m

of pk elements. In other words, 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. 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. We refer to the two-dimensional view of an array A as A_{2D}; and the element $A(i)$ has a 2-D address of the form $(x, y) = (i \operatorname{div} pk, i \bmod pk)$ in this space. Similarly the local address of an element $A(i)$ (denoted using A_{loc}) mapped to a processor m is $k * (i \operatorname{div} pk) + i \bmod k$.

An array section in HPF is of the form $A(\ell : h : s)$, where s is the access stride, and ℓ and h are the lower and upper bounds, respectively. Table 2.1 summarizes the notation. Given an array statement with HPF-style data mappings, it is our aim to generate the address sequence 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 7 ($p = 3, k = 4$ and $s = 7$). Figure 2.1 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 Figure 2.1. While the global access stride is constant (7 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 $3, 8, 14, 25, 31, \ldots$. The address generation problem is to efficiently enumerate this sequence.

Proc. 0				Proc. 1				Proc. 2			
0[0]	1	2	3	4	5	6	7[3]	8	9	10	11
12	13	14[6]	15	16	17	18	19	20	21[5]	22	23
24	25	26	27	28[8]	29	30	31	32	33	34	35[11]
36	37	38	39	40	41	42[14]	43	44	45	46	47
48	49[17]	50	51	52	53	54	55	56[16]	57	58	59
60	61	62	63[23]	64	65	66	67	68	69	70[22]	71
72	73	74	75	76	77[25]	78	79	80	81	82	83
84[28]	85	86	87	88	89	90	91[31]	92	93	94	95

Fig. 2.1. Layout of array A for $p = 3$ and $k = 4$ along with the global and local addresses of the accessed elements in $A(0 : 95 : 7)$ ($s = 7$); Superscripts denote local addresses.

2.1 Related work on one-level mapping

Several papers have addressed this code generation problem. Koelbel [19] derived techniques for compile-time address and communication generation for `BLOCK` and `CYCLIC` distribution for non-unit stride accesses containing a single loop index variable. MacDonald et al. [21] provided a simple solution for restricted case where the block sizes and the number of processors are powers of two. Chatterjee et al. [5] derived a purely runtime technique that identifies a repeating access pattern, which is characterized as a finite-state machine. Their $O(k \log k + \log \min(s, pk))$ 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. Gupta et al. [12] 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 parallel array assignments. For large block sizes, this approach may suffer from cache misses [5]. They present a strategy for choosing a virtualization scheme for each array involved in array statement, based on indexing overheads and not on cache effects. In an exhaustive study of this problem, Stichnoth [33, 34] presented a framework (that bears similarities to the approach of Gupta et al. [12]) to enumerate local addresses and generation of communication sets. Banerjee et al. [2] discuss code generation for regular and irregular applications. Coelho et al. [6] present a survey of approaches to compiling HPF.

Ancourt et al. [1] use a linear algebra framework to generate code for fully parallel loops in HPF; their technique does not work for `DO` loops. Midkiff [22] presented a technique that uses a linear algebra approach to enumerate local accesses on a processor; this technique is similar to the virtual block approach presented by Gupta et al. [12]. van Reeuwijk et al. [32] presented a technique which requires the solution of linear Diophantine equations. Benkner [3] presented a solution for code generation for block-cyclic distributions in Vienna Fortran 90.

Kennedy, Nedeljkovic, and Sethi [16] derived an $O(k + \log \min(s, pk))$ algorithm for address generation. The improvement over Chatterjee et al.'s algorithm comes from avoiding the sorting step at the expense of solving an additional set of $k - 1$ linear Diophantine equations. In Sections 3 thru 7, we present our improved solution to the one-level mapping problem. Unlike Kennedy et al. [16] who solve $k-1$ linear Diophantine equations, our approach requires the solution of just two linear Diophantine equations. In addition, we present an efficient loop-nest based approach to enumerate the array addresses in order to derive the address pattern. Wijshoff [46] describes access sequences for periodic skewing schemes (used in providing efficient data organization in parallel machines) using lattices and derived closed forms for the lattices. He does not discuss code generation. Wang et al. [45] discuss experiments with several address generation solutions and conclude that the strategy described by us in this chapter (and in [39]) is the best strategy overall.

Other work on one-level mapping from our group: In [36, 37, 38, 41], we presented closed form expressions for basis vectors for several cases. Using the closed form expressions for the basis vectors, we derived a non-unimodular linear transformation; the matrix associated with this transformation has a determinant equal to the inverse of the access stride. In an experiment with a large set of values for the parameters p (the number of processors), k (block size) and s (the access stride), we derived the best pair of basis vectors using the closed form expressions for 82% of the problem instances. In later sections, we show that basis vector generation dominates address generation. Recently, we [25] have derived a runtime solution for the basis vector generation problem whose complexity is $O(\log \min(s, pk))$, which is simply the complexity of computing the required gcd. In contrast, all the other algorithms known to date for basis generation have a complexity of $O(k + \log \min(s, pk))$ or worse.

2.2 Related work on two-level mapping

A few methods have been proposed to address the code generation problem for two-level mapping. The solution by Chatterjee et al. [5] involves two applications of the one-level algorithm, where the input strides are a for the first and $a * s$ for the second. They build two finite state machines which

will generate the access sequence for the *allocated* and the *accessed* elements. The next step involves using the finite state machines for the template space to rebuild a new finite state machine for the local address space. This fix-up step involves computing expensive integer divide and mod operations to determine the addresses of *accessed* elements in the local memory space. An added drawback of their technique when compared to our technique is the fact that their one-level pattern tables contain local memory gaps and not actual addresses. However their execution preserves lexicographic ordering and they do not incur any memory wastage.

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

Kaushik [15] uses processor virtualization to handle two-level mapping with block-cyclic distributions. His method involves the generation of addresses for both hole-compression and for the one without hole-compression. First, the amount of memory that must be allocated is determined using a regular section characterization for *block* and *cyclic* distributions. Then, this regular section characterization is extended to the virtual processor approach for handling *block-cyclic* distributions. His technique does not ensure lexicographic execution in the case of virtual cyclic approach. In addition, memory wastage that grows with the amount of allocated storage is incurred.

3. A Lattice Based Approach for Address Generation

In the next few sections, we present a novel technique based on integer lattices for the address generation problem presented in the previous section. 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. Also, we also provide a few optimizations of our basis determination algorithm.

3.1 Assumptions

In the next several sections, 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 [5], which involves two applications of our algorithm; this approach is discussed in Section 8 thru 10.

We assume that A is identically aligned to the template T. As it is evident from Figure 1.1(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 is the course number to which this element belongs and y is 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 chapter. Similarly, A_{loc} refers to the locally allocated piece of array A on a processor.

3.2 Lattices

We use the following definitions of a lattice and its basis [11].

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

For $k = n$, this is equivalent to the condition that the determinant of the matrix X whose columns are $\mathbf{x_i}$ $(1 \leq i \leq n)$ is non-zero, *i.e., det* $([\mathbf{x_1 x_2} \cdots \mathbf{x_n}]) \neq 0$. Now we state the following definition from the theory of lattices [11]; see [11] for a proof.

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

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

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

Definition 3.3. *Let* Λ *be a discrete subspace of* $\Re^n$ *which is not contained in an* $(n-1)$*-dimensional linear subspace of* $\Re^n$*. 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 $\Re^2$ and in general, is not contained in a 1-dimensional linear subspace of $\Re^2$; 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:

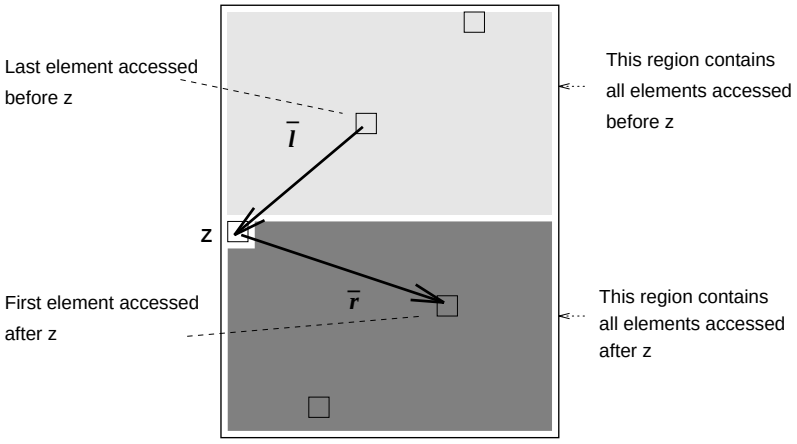

Fig. 3.1. Explanation of basis determination

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 4) 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 5) uses the fact that if (with a given origin) 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* $\mathbf{q}$ *that belong to Λ can be written as*

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

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

This chapter 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.

4. 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 4.1. Let $\mathbf{b_1}, \mathbf{b_2}, \cdots, \mathbf{b_n}$ be independent points of a lattice Λ in $\Re^n$. Then Λ has a basis $\{\mathbf{a_1}, \mathbf{a_2}, \cdots, \mathbf{a_n}\}$ such that

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

where $u_{ii} > 0$ and $0 \leq u_{ki} < u_{ii}$ $(k < i; i = 1, \ldots, n)$. In addition, the set of points $\{\mathbf{b_1}, \mathbf{b_2}, \cdots, \mathbf{b_n}\}$ 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 $\mathbf{o}$ to refer to the origin of the lattice.

Theorem 4.1. *Let* $\mathbf{b_1}$ *and* $\mathbf{b_2}$ *be independent points of a lattice* Λ *in* $\Re^2$ *such that the closed triangle formed by the vertices* $\mathbf{o}, \mathbf{b_1}$ *and* $\mathbf{b_2}$ *contains no other points of* Λ*. Then* $\{\mathbf{b_1}, \mathbf{b_2}\}$ *is a basis of* Λ*.*

Proof. Let $\{\mathbf{a_1}, \mathbf{a_2}\}$ be any basis of Λ. From Result 4.1, we can write

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

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 $\mathbf{o}$ and $\mathbf{b_1}$ 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 $\mathbf{o}, \mathbf{b_1}$ and $\mathbf{b_2}$ contains the point $\mathbf{a_2} \in \Lambda$; similarly, if $u_{12} \geq 1$, the triangle formed by $\mathbf{o}, \mathbf{b_1}$ and $\mathbf{b_2}$ contains the point $\mathbf{a_1} + \mathbf{a_2} \in \Lambda$. This contradicts our hypothesis. Hence, $u_{22} = 1$. Since, $u_{11} = u_{22} = 1$, it follows from Result 1 that the vectors $\mathbf{b_1}$ and $\mathbf{b_2}$ 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 $\mathbf{x_1}$, $\mathbf{x_2}$, and $\mathbf{x_3}$ be three consecutively accessed elements of the array section lattice on processor m. If $\mathbf{x_1}$, $\mathbf{x_2}$, and $\mathbf{x_3}$ are independent points (do not lie on a straight line), then the vectors $\mathbf{x_2} - \mathbf{x_1}$ and $\mathbf{x_3} - \mathbf{x_2}$ 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 $\mathbf{z_f}$ be some element accessed in column c_f and let $\mathbf{z_l}$ be some element accessed in column c_l by a processor. Let $\mathbf{x^{prev}}$ denote the element accessed immediately before $\mathbf{x}$ in lexicographic order on

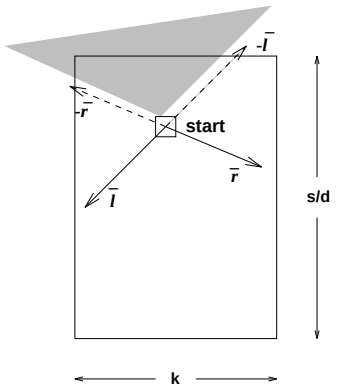

Fig. 4.1. No lattice point is a non-positive linear combination of basis vectors

a processor, and $\mathbf{x^{next}}$ denote the element accessed immediately after $\mathbf{x}$ in linear order on a processor. We do not discuss the case where $\mathbf{z_f}$ and $\mathbf{z_f^{next}}$ (or $\mathbf{z_f^{prev}}$) are in the same column. This case is handled separately and is easily detected by our technique.

Theorem 4.2. *The set of points* $\{\mathbf{z_f^{prev}}, \mathbf{z_f}, \mathbf{z_f^{next}}\}$ *generate a basis of* Γ_m.

Proof. From Theorem 4.1, the set of points $\{\mathbf{z_f^{prev}}, \mathbf{z_f}, \mathbf{z_f^{next}}\}$ 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): $\text{Column}(\mathbf{z_f^{prev}}) < \text{Column}(\mathbf{z_f}) < \text{Column}(\mathbf{z_f^{next}})$.
Case (b): $\text{Column}(\mathbf{z_f^{next}}) < \text{Column}(\mathbf{z_f}) < \text{Column}(\mathbf{z_f^{prev}})$.

Neither of these cases hold, since $\text{Column}(\mathbf{z_f}) = c_f$ 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 $\{\mathbf{z_l^{prev}}, \mathbf{z_l}, \mathbf{z_l^{next}}\}$ also generate a basis of Γ_m. We use the set $\{\mathbf{z_f^{prev}}, \mathbf{z_f}, \mathbf{z_f^{next}}\}$. We refer to the vector $\mathbf{z_f} - \mathbf{z_f^{prev}}$ as $\mathbf{l} = (l_1, l_2)$ and the vector $\mathbf{z_f^{next}} - \mathbf{z}$ as $\mathbf{r} = (r_1, r_2)$. Again by construction, $l_1 > 0$, $r_2 > 0$, $l_2 < 0$ and $r_1 \geq 0$. This is illustrated in Figure 3.1 on p. 9.

4.1 Basis Determination Algorithm

In order to obtain a basis for the lattice, we need to find three points belonging to the lattice not on a straight line such that the triangle formed by

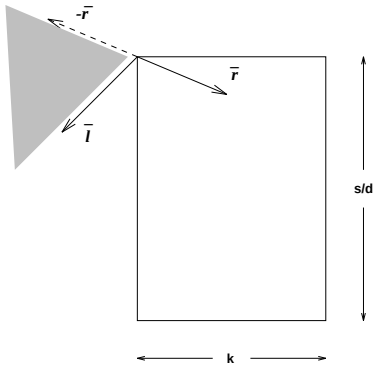

Fig. 4.2. No lattice points in region spanned by vectors **l** and **-r**

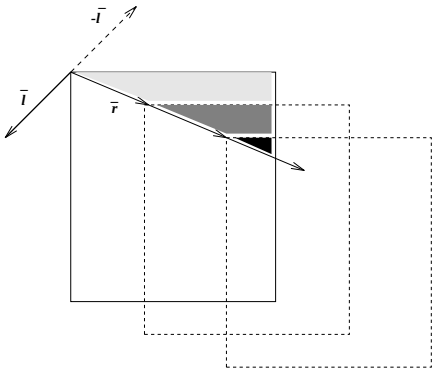

Fig. 4.3. No lattice points in region spanned by vectors **-l** and **r**

them contains no other lattice point. Chatterjee et al. [5] suggested a way to locate lattice points by solving linear Diophantine equation for each column with accessed elements. For details on their derivation refer [5]. The smallest solution of each of these solvable equations gives the smallest array element accessed in the corresponding column on a processor. 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 $\mathbf{z}_f$ be the first element accessed in column c_f. Since the access pattern on a given processor repeats after $\frac{pks}{\gcd(s,pk)}$ elements, the point

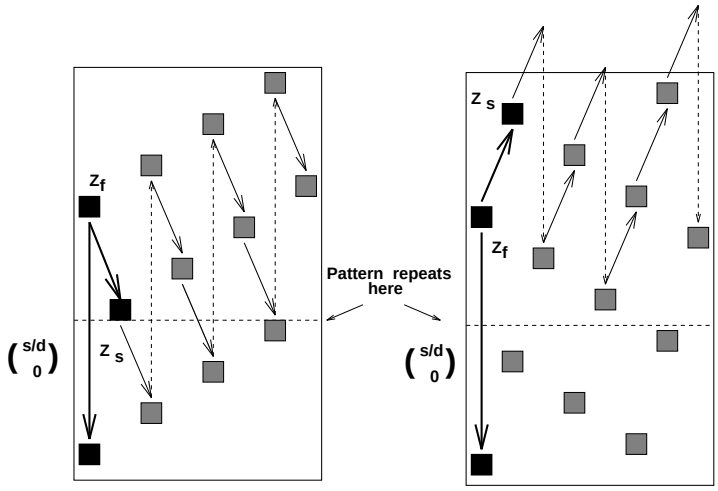

Fig. 4.4. Starting points in the columns generated by the vectors $\left(\frac{s}{d}, 0\right)$ and $\mathbf{z}_s - \mathbf{z}_f$.

accessed immediately after $\mathbf{z}_f$ in column c_f is $\mathbf{z}_f + \frac{pks}{\gcd(s,pk)}$. Now if c_s is the second column in which an element is accessed on the processor, and $\mathbf{z}_s$ is the first element accessed in it, then without loss of generality $\mathbf{z}_f$, $\mathbf{z}_f + \frac{pks}{\gcd(s,pk)}$ and $\mathbf{z}_s$ form a basis for the array section lattice. Hence the vectors $\left(\frac{s}{d}, 0\right)$ and $\mathbf{z}_s - \mathbf{z}_f$ form a pair of basis vectors of the array section lattice.

The elements $\mathbf{z}_f$ and $\mathbf{z}_s$ can be obtained by solving the first two solvable Diophantine equations in the algorithm (Lines 4 and 6) shown in Figure 4.5. Figure 4.4 shows the basis vectors generated as explained above whereas Figure 4.5 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 $\mathbf{z}_s$ lies on a course above or below $\mathbf{z}_f$. 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 the three independent points $\mathbf{z_f^{prev}}, \mathbf{z_f}, \mathbf{z_f^{next}}$ 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. Thus, 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. This new basis determination algorithm performs substantially better than that proposed by Kennedy et al. for large values of k.

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

```
1  (d, x, y) ← EXTENDED-EUCLID(s, pk); length ← 2; start ← h + 1
2  last ← ℓ + pks/d − 1; first ← ℓ; bmin ← last + 1; amax ← first − 1
3  i ← d⌈(km − ℓ)/d⌉; i_end ← km − ℓ + k − 1
4  if i > i_end then return ⊥, ⊥, 0, ⊥, ⊥, ⊥, ⊥        /* No element */
5  amin ← bmax ← z_f ← ℓ + (s/d)(ix + pk⌈−ix/pk⌉); i ← i + d
6  if i > i_end then return z_f, z_f, 1, ⊥, ⊥, ⊥, ⊥    /* One element */
7  loc ← z_s ← ℓ + (s/d)(ix + pk⌈−ix/pk⌉); length ← 2
8  if z_s < z_f then
9     amin ← amax ← z_s; vec2 ← z_s − z_f; vec1 ← pks/d + vec2
10    else
11       bmin ← bmax ← z_s; vec1 ← z_s − z_f; vec2 ← vec1 − pks/d
12    endif
13    if vec1 ≤ vec2 then
14       loc ← loc + vec1; i ← i + d
15       while i ≤ i_end do
16          if loc > last then
17             loc ← loc − pks/d
18          endif
19          if loc < z_f then                  /* loc is accessed before z_f */
20             amax ← max(amax, loc); amin
                  ← min(amin, loc)
21          else                               /* loc is accessed after z_f */
22             bmax ← max(bmax, loc); bmin
                  ← min(bmin, loc)
23          endif
24          loc ← loc + vec1; i ← i + d; length ← length + 1
25       enddo
26       else
27          loc ← loc + vec2; i ← i + d
28          while i ≤ i_end do
29             if loc < first then
30                loc ← loc + pks/d
31             endif
32             if loc < z_f then               /* loc is accessed before z_f */
33                amax ← max(amax, loc); amin
                     ← min(amin, loc)
34             else                            /* loc is accessed after z_f */
35                bmax ← max(bmax, loc); bmin
                     ← min(bmin, loc)
36             endif
37             loc ← loc + vec2; i ← i + d; length
                  ← length + 1
38          enddo
39          endif
40          (start, end, l_1, l_2, r_1, r_2) ← COMPUTE-VECTORS (z_f,
               p, k, s, d, amin, amax , bmin, bmax)
41          return start, end, length − 1, l_1, l_2, r_1, r_2
```

Fig. 4.5. Algorithm for determining basis vectors

Input: z_f, p, k, s, d, $amin$, $amax$, $bmin$, $bmax$.
Output: The *start* memory location, *end* memory location and the basis vectors $\mathbf{r} = (r_1, r_2)$ and $\mathbf{l} = (l_1, l_2)$ for processor m.
Method:

```
1   if amin = z_f and amax = −1 then                    /* above is empty */
2       l_1 ← ⌊z_f/pk⌋ + s/d − ⌊bmax/pk⌋
3       l_2 ← z_f mod k − bmax mod k
4       r_1 ← ⌊bmin/pk⌋ − ⌊z_f/pk⌋
5       r_2 ← bmin mod k − z_f mod k
6       else if bmin = pks/d and bmax = z_f then    /* below is empty */
7           l_1 ← ⌊z_f/pk⌋ − ⌊amax/pk⌋
8           l_2 ← z_f mod k − amax mod k
9           r_1 ← ⌊amin/pk⌋ + s/d − ⌊z_f/pk⌋
10          r_2 ← amin mod k − z_f mod k
11          else                  /* neither above nor below is empty */
12              l_1 ← ⌊z_f/pk⌋ − ⌊amax/pk⌋
13              l_2 ← z_f mod k − amax mod k
14              r_1 ← ⌊bmin/pk⌋ − ⌊z_f/pk⌋
15              r_2 ← bmin mod k − z_f mod k
16          endif
17          start ← amin
18          end ← bmax
19          return start, end, l_1, l_2, r_1, r_2
```

Fig. 4.6. `COMPUTE-VECTORS` procedure for basis vectors determination algorithm

4.2 Extremal Basis Vectors

In this section, we show that the basis vectors generated by our algorithm in Figure 4.5 form an extremal set of basis vectors.

Theorem 4.3. *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* $\mathbf{l}$ *and* $\mathbf{r}$.

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

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

where $\mathbf{r} = (r_1, r_2)$ and $\mathbf{l} = (l_1, l_2)$. Also, $l_1 > 0, l_2 < 0, r_1 \geq 0$ and $r_2 > 0$ by construction. Suppose $\mathbf{q} \in \Gamma_m$ 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 Figure 4.1 that $\mathbf{q}$ lies in the region above the start element $\mathbf{z}$. This contradicts our earlier assumption that $\mathbf{z}$ 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 $\mathbf{z}$ on a processor lies in the first non-empty column.

Let $v_1 \geq 0$ and $v_2 < 0$. As shown in Figure 4.2, $\mathbf{q}$ lies in the region to the left of $\mathbf{z}$ since $v_1 l_2 + v_2 r_2 < 0$. This contradicts our assumption that $\mathbf{q} \in \Gamma_m$. 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 $\mathbf{z} + \mathbf{r}$ or $\mathbf{z} + \mathbf{r} - \mathbf{l}$. If the next accessed element of Γ_m is $\mathbf{z} + \mathbf{r} - \mathbf{l}$, then this point should be located on a course above $\mathbf{z}$ or on a course below $\mathbf{z}$. If this element is located on a course above $\mathbf{z}$, it would not be a point in the lattice Γ_m. If this element is located on a course below $\mathbf{z}$, then this element is lexicographically closer to $\mathbf{z}$ than the point $\mathbf{z}+\mathbf{r}$ which is impossible (due to the construction of $\mathbf{r}$). By the above arguments (as can be seen in Figure 4.3), we have shown that the next element accessed after $\mathbf{z}$ can only be $\mathbf{z}+\mathbf{r}$. A repeated application of the above argument rules out the presence of a lattice point in the shaded regions in Figure 4.3. If $\mathbf{z}$ is not in the first accessed column on processor m, similar reasoning can be used for the vector $\mathbf{l}$. Hence, the result. □

4.3 Improvements to the algorithm for $s < k$

If $s < k$, it is sufficient to find only s+1 lattice points instead of k lattice points (as in [16]) in order to derive the basis vectors. Our implementation uses this idea which is explained next. Figure 4.7 shows that the access pattern repeats after s columns. A close inspection of the algorithm for determining the basis of the lattice for the case where $s < k$ reveals that the basis vector $\mathbf{r}$ will always be $(0, s)$. We also notice that since we access at least one element on every row, the first component of the basis vector $\mathbf{l}$, *i.e.,* l_1 must always be 1. Hence, if $s < k$, all we need to solve for is the second component of $\mathbf{l}$ *i.e.,* l_2. This results in a reduced number of comparisons for the $s < k$ case; in addition, there is no need to compute *bmin*, since it is not needed for the computation of l_2.

Consider the case where where $l = 36$, $p = 4$, $k = 16$ and $s = 5$ which is shown in Figure 4.7; We illustrate this case for processor number 2. Running through lines 1–38 of the algorithm in Figure 4.5 for this case, we get $\mathbf{z_f} = 96$, $amin = 36$, $amax = 46$, $bmin = 101$ and $bmax = 301$ for processor 2. Since the pattern of the smallest element accessed in a column repeats after every s columns, it is sufficient to enumerate the elements accessed in the last s columns to obtain *amax* and *bmax*. *amin* can be obtained by subtracting a suitable multiple of s from the smallest element in the last s columns lying above $\mathbf{z_f}$. By making these changes to the algorithm in Figure 4.5 for the case where $s < k$, we can obtain the required input for the `COMPUTE-VECTORS` procedure.

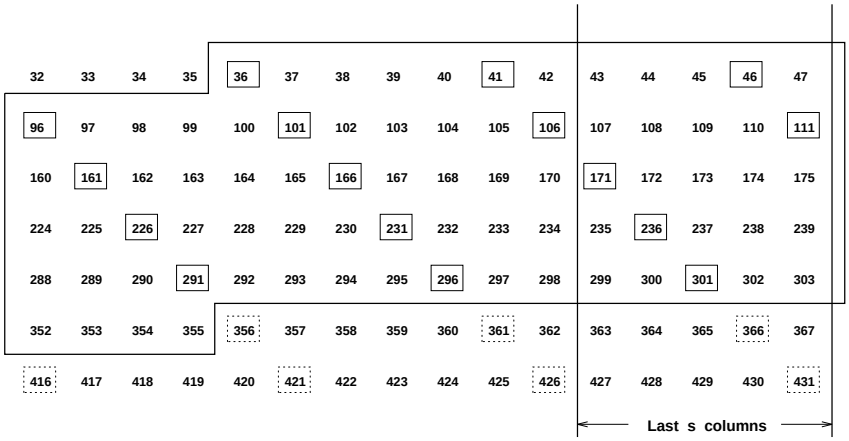

Fig. 4.7. Addresses of elements of array A accessed on processor 2 for the case $p = 4$, $k = 16$, $s = 5$ and offset $l = 36$

The improved algorithm to determine the basis vectors for each processor is as follows.

1. Solve the Diophantine equation corresponding to the first solvable column to obtain $\mathbf{z_f}$.
2. Solve the Diophantine equations corresponding to the last and last but one solvable columns to obtain $\mathbf{z_l}$ and $\mathbf{z_{l-1}}$ respectively. Use these two solutions in a similar way as shown in Figure 4.5 to generate a pair of basis vectors for the lattice **vec1** and **vec2** in terms of offsets.
3. Using the above basis vectors enumerate all the points in the last s columns starting from the last solvable column. By comparing these elements to $\mathbf{z_f}$, we get the smallest and the largest element accessed in these s columns that lie below $\mathbf{z_f}$, namely $bmin'$ and $bmax'$. Similarly, we find the smallest and the largest elements that lie above $\mathbf{z_f}$, namely $amin'$ and $amax'$.
4. If the region above $\mathbf{z_f}$ is not empty then, $amax = amax'$ and $amin = amin' - is$ (where is is a suitable multiple of s). If l lies on the processor then $amin = l$.
5. If both l and $l - s$ lie on the processor then $bmax = l + \frac{pks}{d}$. If the region below $\mathbf{z_f}$ is not empty then, $bmin = bmin' - is$ (where is is a suitable multiple of s) and $bmax = bmax'$.
6. Generate the lattice basis using the `COMPUTE-VECTORS` procedure.

4.4 Complexity

Line 1 of the algorithm in Figure 4.5 is the extended Euclid step which requires $O(\log \min(s, pk))$ time. Lines 2 thru 40 require $O(\min(s, k))$ time.

Thus, the basis generation part of our algorithm is $O(\log \min(s, pk) + \min(s, k))$; the basis generation portion of the algorithm in Kennedy et al. [16] is $O(k + \log \min(s, pk))$. We note that the address enumeration part of both the algorithms is $O(k)$. Experiments have shown that the basis determination part dominates the total time in practice. See Section 7 for a discussion. Thus, our algorithm is superior to that of Kennedy et al. [16].

5. 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 $(x, y) = (i\text{div } pk, i \bmod pk)$ in this space. We refer to the two-dimensional view of an array A as A_{2D}. 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 Figure 1.1(a) could be written as follows:

```
DO i = 0, ⌊(N−1)/pk⌋
   DO j = 0, pk − 1
      A2D(i, j) = ·····
   ENDDO
ENDDO
```

We apply a non-unimodular loop transformation [23, 24] 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, (s/d − 1)
   DO j = mk, mk + k − 1
      A2D[i, j] = ···
   ENDDO
ENDDO
```

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:

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

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

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

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 [7] on the following system of inequalities to solve for integral u and v:

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

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

$$\left\lceil \frac{(-mk - k + 1) r_1}{s} \right\rceil \leq u \leq \left\lfloor \frac{(\frac{s}{d} - 1) r_2 - mk r_1}{s} \right\rfloor$$

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

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

```
DO u = ⌈(−mk−k+1)r1 / s⌉ , ⌊((s/d)−1)r2 − mkr1 / s⌋
    DO v = ⌈max((mk−ul2)/r2, −ul1/r1)⌉ , ⌊min((mk+k−1−ul2)/r2, (s/d)−1−ul1)/r1)⌋
        A2D[l1u + r1v, l2u + r2v] = ·····
    ENDDO
ENDDO
```

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

$$\begin{aligned} 0 \leq \quad & u \quad \leq \frac{r_2}{s}\left(\frac{s}{d} - 1\right) \\ \left\lceil \frac{mk - ul_2}{r_2} \right\rceil \leq \quad & v \quad \leq \left\lfloor \frac{mk + k - 1 - ul_2}{r_2} \right\rfloor \end{aligned}$$

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

```
DO u = 0, r2/s (s/d − 1)
    DO v = ⌈(mk − ul2)/r2⌉ , ⌊(mk + k − 1 − ul2)/r2⌋
        A2D[l1u, l2u + r2v] = ·····
    ENDDO
ENDDO
```

Example 5.1. Code generated for the case where $\ell = 0$, $p = 3$, $k = 4$ and $s = 11$ for processor 1.

The set of addresses generated by the algorithm in Figure 4.5 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 5 and 6 of the algorithm in Figure 4.6. So our algorithm returns $\mathbf{l} = (1, -1)$ and $\mathbf{r} = (8, 3)$ as the basis vectors. The access pattern is shown in Figure 5.1. The node code to obtain the access pattern for processor 1 is:

```
DO u = ⌈−56/11⌉ , ⌊−2/11⌋
    DO v = ⌈max((4+u)/3, −u/8)⌉ , ⌊min((7+u)/3, (10−u)/8)⌋
        A2D[u + 8v, −u + 3v] = ·····
    ENDDO
ENDDO
```

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

u	$v_{lb} = \lceil \max\left(\frac{4+u}{3}, \frac{-u}{8}\right) \rceil$	$v_{ub} = \lfloor \min\left(\frac{7+u}{3}, \frac{10-u}{8}\right) \rfloor$	accessed elements (2D)
-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 global addresses we get the global access pattern {55, 66, 77, 88} on processor 1 which gives the local access pattern {19, 22, 25, 28} on processor 1.

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

Fig. 5.1. Addresses of the accessed elements of array A along with the 2-dimensional view for the case $p = 3$, $k = 4$ and $s = 11$.

6. Optimization of Loop Enumeration: `GO-LEFT` and `GO-RIGHT`

A closer look at Figure 5.1 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 5) 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}. Next, we show that for the pair of basis vectors obtained using the algorithm shown in Figure 4.5, the `GO-LEFT` scheme is always legal.

Theorem 6.1. *Given a point* $\mathbf{q}$ *belonging to the lattice* Γ_m *and a pair of extremal basis vectors* $\mathbf{l}$ *and* $\mathbf{r}$ *obtained using the algorithm in Figure 4.5, then on applying* B_L^{-1} *as a transformation we maintain the access order.*

Proof. Since $\mathbf{l}$ and $\mathbf{r}$ are extremal basis vectors,

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

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

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

Case 1: $\mathbf{q}+\mathbf{r}$ is lexicographically closer to $\mathbf{q}$ than $\mathbf{q}+\mathbf{l}$.
Case 2: $\mathbf{q}+\mathbf{l}$ is lexicographically closer to $\mathbf{q}$ than $\mathbf{q}+\mathbf{r}$.

In Case 1, $\mathbf{q}+\mathbf{l}$ is lexicographically closer to $\mathbf{q}$ than $\mathbf{q}+\mathbf{r}$. So it should be clear that $\mathbf{q}+\mathbf{r}$ should be lexicographically closer to $\mathbf{q}+\mathbf{l}$ than to $\mathbf{q}$, which is impossible (due to the construction of $\mathbf{r}$ and $\mathbf{l}$). Hence our assumption that $\mathbf{q}+\mathbf{r} \in \Gamma_m$ and $\mathbf{q}+\mathbf{l} \in \Gamma_m$ 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 lexicographic order of the points accessed by repeatedly adding $\mathbf{l}$ until we run out of Γ_m and then add a $\mathbf{r}$ and continue adding $\mathbf{l}$ 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 Figure 4.5. The loop nest for $r_1 \neq 0$ is:

```
DO u = ⌈mkl1/s⌉, ⌊((s/d − 1)(−l2) + (mk + k − 1)l1)/s⌋
    DO v = ⌈max((mk + k − 1) − ur2)/l2, −ur1/l1)⌉, ⌊min((mk − ur2)/l2, (s/d − 1 − ur1)/l1)⌋
        A2D[r1u + l1v, r2u + l2v] = ·····
    ENDDO
ENDDO
```

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

```
DO u = ⌈mkl1/s⌉, ⌊((s/d − 1)(−l2) + (mk + k − 1)l1)/s⌋
    DO v = ⌈max((mk + k − 1) − ur2)/l2, 0)⌉, ⌊min((mk − ur2)/l2, s/d − 1)/l1)⌋
        A2D[l1v, r2u + l2v] = ·····
    ENDDO
ENDDO
```

Next, we need to decide when it is beneficial to use `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 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 block size k and second components of the basis vectors. If $r_2 \leq -l_2$, we use `GO-RIGHT`; else, we use `GO-LEFT`.

Example 6.1. Code generated for the case where $\ell = 0$, $p = 3$, $k = 4$ and $s = 11$ for processor 1 when we choose to `GO-LEFT`.

Input: *start*, *end*, $\mathbf{r} = (r_1, r_2)$, $\mathbf{l} = (l_1, l_2)$.
Output: The address sequence.
Method:
if $r_2 \leq -l_2$ **then** /* `GO-RIGHT` */

1. $u_{start} \leftarrow \dfrac{r_2(start \text{ div } pk - \ell \text{ div } pk) - r_1(start \bmod pk - \ell \bmod pk)}{s}$
2. $u_{end} \leftarrow \dfrac{r_2(end \text{ div } pk - \ell \text{ div } pk) - r_1(end \bmod pk - \ell \bmod pk)}{s}$
3. Scan all the elements on the first line (u_{start}), starting at the *start* element and then adding **r** until there are no more elements on this processor.
4. From the previous start add **l** and then add **r** as many times as necessary till you get back onto the processor space. The element thus obtained is the start for the new line. Starting at this element keep adding **r** until you run out of the processor space. Repeat this until the line immediately before the last line (u_{end}).
5. Obtain the start point on the last line as before. Scan all the elements along the line from the start by adding **r** until you reach the *end* element.

else /* `GO-LEFT` */

1. $u_{start} \leftarrow \dfrac{-l_2(start \text{ div } pk - \ell \text{ div } pk) + l_1(start \bmod pk - \ell \bmod pk)}{s}$
2. $u_{end} \leftarrow \dfrac{-l_2(end \text{ div } pk - \ell \text{ div } pk) + l_1(end \bmod pk - \ell \bmod pk)}{s}$
3. Scan all the elements on the first line (u_{start}), starting at the *start* element and then adding **l** until there are no more elements on this processor.
4. From the previous start add **r** and then add **l** as many times as necessary till you get back onto the processor space. The element thus obtained is the start for the new line. Starting at this element keep adding **l** until you run out of the processor space. Repeat this until the line immediately before the last line (u_{end}).
5. Obtain the start point on the last line as before. Scan all the elements along the line from the start by adding **l** until you reach the *end* element.

endif

Fig. 6.1. Algorithm for `GO-LEFT` and `GO-RIGHT`

The basis vectors obtained by running through the algorithms shown in Figure 4.5 are $\mathbf{l} = (1, -1)$ and $\mathbf{r} = (8, 3)$. Hence the resulting node code for processor 1 is:

```
DO u = ⌈4/11⌉, ⌊17/11⌋
    DO v = max(3u − 7, −8u), min(3u − 4, 10 − 8u)
        A_2D[8u + v, 3u − v] = ·····
    ENDDO
ENDDO
```

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.

6.1 Implementation

We observe from the example in Sections 5 and 6 that the code generated for `GO-RIGHT` enumerates the points that belong to a family of parallel lines, *i.e.*, along the vector $\mathbf{r}$, by moving from one parallel line to the next within the family along the vector $\mathbf{l}$ and the code generated for `GO-LEFT` enumerates the points that belong to a family of parallel lines along the vector $\mathbf{l}$, by moving from one parallel line to the next within the family along the vector $\mathbf{r}$. So in the code derived in Section 5, 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 Figure 4.5. The start line u_{start} and end line u_{end} can be evaluated as follows (using `GO-RIGHT` enumeration scheme):

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

Hence,

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

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

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. Figure 6.1 shows our algorithm for loop enumeration.

7. Experimental Results for One-level Mapping

We performed experiments on our pattern generation algorithm on a Sun Sparcstation 20. We used the `cc` compiler using the `-fast` optimization switch; the function `gettimeofday()` was used to measure time. When computing the 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 [16] and our algorithm include basis and table generation times. Tables 7.1(a)–7.1(c) show the total times for the above three algorithms and the total time for pattern generation for the algorithm proposed by Chatterjee et al. [5] (referred to as "Sort" in the tables). 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 9 times faster than the two Rice [16] algorithms. For larger block sizes, if $s < k$, our algorithm is 7 to 9 times faster than the Rice algorithms because of the need to find only $s + 1$ lattice points, instead of k lattice points, in order to find the basis vectors. In addition, for larger block sizes, experiments indicate that address enumeration time (given the basis vectors) for our algorithm is less than that of [16]. 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 [16] do not exploit this enumeration choice, our algorithm performs significantly better. In addition, our algorithm is 13 to 65 times faster than the approach of Chatterjee et al. [5] for large block sizes.

In addition to the total time, we examined the basis determination time and the access enumeration times separately. In general, the basis determination time accounts for 75% of the total address generation time and is about 3 times the actual enumeration time. The basis determination times are shown in Figure 7.1 and the enumeration times are shown in Figure 7.2. In these figures, we plot the times taken for our algorithm ("Loop"), and the best of the times for the two Rice implementations. Figure 7.1(a) shows that for $s = 7$, the basis generation time for Loop is practically constant while that for Rice increases with block size, k; for $k = 2048$, the basis generation time for Rice is about 50 times that of our algorithm. In Figure 7.1(b) ($s = 99$), it is clear that the basis generation time for our algorithm is nearly constant while that for Rice increases from $k = 128$ onward; this is because of the fact that our basis generation algorithm has a complexity $O(\min(s, k))$ while Rice

has a complexity $O(k)$. Figures 7.1(c)–(f) indicate that for large values of k, the Loop basis generation time is about half that of the Rice basis generation time. Figures 7.2(a)-(f) show that for small block sizes, the enumeration time for Loop and Rice are comparable, and that from $k = 64$ onwards, the enumeration time for Loop is lower than that of Rice. From these figures, it is clear that the Loop algorithm outperforms the Rice algorithm in both the basis determination and address enumeration phases of address generation.

8. Address Sequence Generation for Two-level Mapping

Non-unit alignment strides render address sequence generation even more difficult since the addresses can not directly be represented as a lattice; in this case, the addresses can be thought of as the composition of two integer lattices. This section presents solution to the problem of address generation for such a case when the data objects are mapped to processor memories using `CYCLIC`(k) distribution. We present several methods of generating the address sequence. Our approach involves construction of pattern tables which does not incur runtime overheads as compared to other existing solutions for this problem. We use two applications of the method described in the preceding sections to generate the pattern of accesses.

8.1 Problem statement

Consider the following HPF code

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

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

To recall, when the alignment stride $a > 1$, the mapping is called a *two-level mapping*. A non-unit-alignment-stride mapping results in many template cells that do not have any array elements aligned to. These empty template cells are referred to as *holes*. We need not allocate memory for *holes* in the local address space during mapping. The challenge then is to generate the sequence of accessed elements in this local address space ensuring that no storage is wasted on holes.

Figure 8.1(a) shows the distribution of the template cells onto a processor arrangement. For this example, the alignment stride a and access stride

Table 7.1. Total address generation times (in μs) for our technique (Loop), Right and Zigzag of Rice [16] and the Sort approach [5] on a Sun Sparcstation 10

(a) $p = 32; s = 3$ and $s = 5$

Block Size	$s = 3$				$s = 5$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
2	17	19	19	20	19	21	21	22
4	20	23	23	27	29	31	31	35
8	21	24	25	37	21	24	25	37
16	29	33	33	69	25	33	34	69
32	29	44	45	150	31	46	47	152
64	33	73	76	453	42	82	84	460
128	36	127	132	845	37	126	131	843
256	47	238	247	1638	48	236	246	1638
512	64	458	480	3213	67	457	472	3221
1024	104	902	936	6383	113	905	946	6384

(b) $p = 32; s = 7$ and $s = 9$

Block Size	$s = 7$				$s = 9$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
2	17	19	19	20	17	19	19	20
4	21	23	23	27	21	23	23	27
8	31	34	35	47	23	26	27	39
16	25	31	32	67	26	33	35	69
32	32	46	47	152	42	55	57	160
64	43	81	84	460	44	81	84	460
128	38	125	131	843	39	125	131	843
256	49	236	245	1638	50	236	245	1637
512	76	464	487	3222	69	454	470	3220
1024	105	891	938	6368	107	890	919	6392

(c) $p = 32; s = 11$ and $s = 99$

Block Size	$s = 11$				$s = 99$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
2	27	29	29	30	35	37	37	38
4	37	39	39	43	45	47	47	51
8	31	34	35	47	55	59	59	71
16	35	41	42	77	51	58	58	93
32	32	44	47	151	50	64	64	170
64	37	73	75	452	71	91	98	469
128	50	135	141	854	97	149	152	865
256	67	252	261	1654	151	256	276	1656
512	70	455	469	3221	114	453	470	3224
1024	108	890	918	6389	154	887	916	6392

Table 7.2. Total address generation times (in μs) for our technique (Loop), Right and Zigzag of Rice [16] and the Sort approach [5] on a Sun Sparcstation 10

(a) $p = 32; s = k - 1$ and $s = k + 1$

Block Size k	$s = k - 1$				$s = k + 1$			
	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
2	17	19	19	20	19	21	21	22
4	20	23	23	27	29	31	31	35
8	31	34	35	47	23	26	27	39
16	26	33	33	69	26	33	33	69
32	33	45	45	154	33	45	45	155
64	64	84	87	459	64	82	90	459
128	92	141	141	853	93	134	144	853
256	149	255	255	1642	150	240	255	1641
512	263	486	485	3219	263	459	486	3219
1024	491	946	944	6375	493	894	946	6375

(b) $p = 32; s = pk - 1$ and $s = pk + 1$

Block Size k	$s = pk - 1$				$s = pk + 1$			
	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
2	17	19	19	20	15	17	17	18
4	18	21	21	25	16	19	19	23
8	19	24	24	37	17	22	23	35
16	23	31	31	67	20	29	30	65
32	29	45	45	155	26	43	44	147
64	43	73	72	449	38	72	74	449
128	70	128	128	843	62	129	132	844
256	124	239	239	1631	109	243	250	1637
512	232	463	463	3209	204	474	486	3220
1024	449	909	909	6365	395	932	958	6388

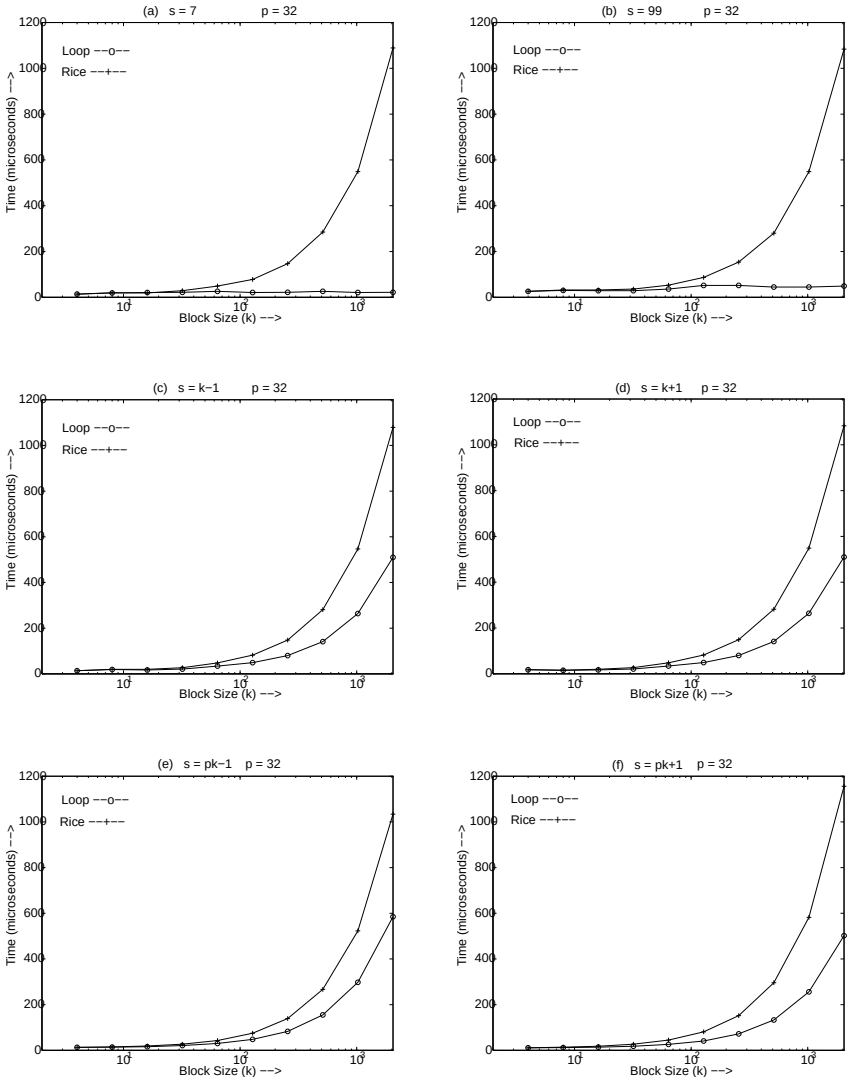

Fig. 7.1. Basis vector generation times for $p = 32$ processors for various block sizes and strides

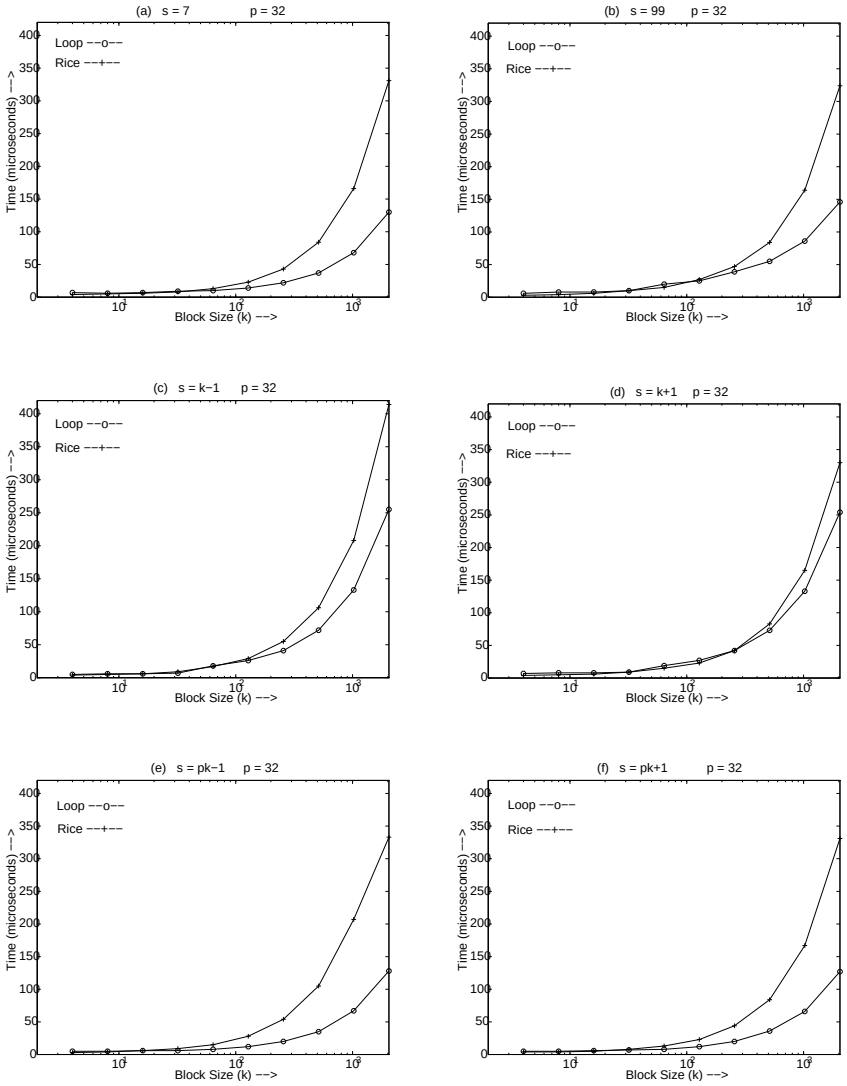

Fig. 7.2. Lattice enumeration times (given the basis vectors) for $p = 32$ processors for various block sizes and strides

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

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

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

(b): Local memory layout for template cells

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

(c): Local memory layout for array cells.

Fig. 8.1. Two-level mapping of array A when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

s are both equal to 3. The number of processors p is 4 and we assume a `CYCLIC`(4) distribution; this example is from Chatterjee et al. [5]. Now, we define a few terms used here. The set of global indices of array elements that are aligned to some template cell on a processor is called the set of *allocated* elements. The set of global indices of accessed array elements that are aligned to some template cell on that processor is called the set of *accessed* elements. These *accessed* elements are however a subset of *allocated* elements. For the given example, $\{0, 1, 6, 11, 16, \cdots\}$ is the set of *allocated* elements and $\{0, 6, 27, 33, 48, \cdots\}$ is the set of *accessed* elements for processor 0. Figure 8.1(b) shows the local address space of template cells on all the processors. The problem of deriving the *accessed* elements for this template space is similar to a one-level mapping problem where the stride s is replaced with $a * s$. However using this method we incur huge memory wastage and suffer from data locality resulting in higher execution times.

If one eliminates holes in this layout, we can have a significant savings in memory usage. This can be achieved by viewing the local address space as shown in Figure 8.1(c). This local address space does not have any memory wastage. However additional work at address generation time has to be done to switch from the template space to the local space. Due to the absence of these *holes* we can expect improved data locality and thus leading to faster execution times. The address generation problem now is to generate the set of elements *accessed* in this local address space, efficiently at runtime.

9. Algorithms for Two-level Mapping

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

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

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

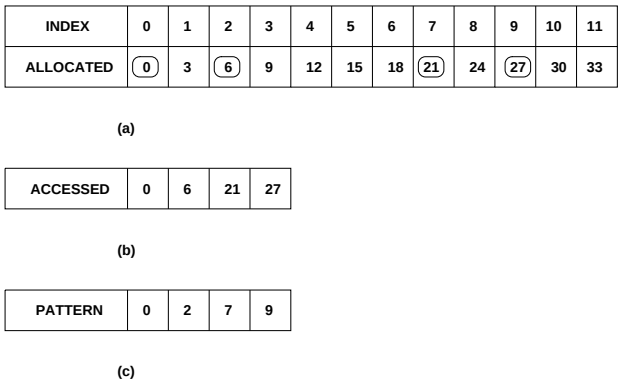

Fig. 9.1. Local addresses of *accessed* and *allocated* elements along with *two-level* access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

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

All these algorithms first view the address space as an integer lattice and use basis vectors to generate the access sequence of both *allocated* and *accessed* elements. The basis vectors are generated using our one-level address generation algorithm discussed earlier. Two applications of the one-level algorithm with input strides being a and $a * s$ in each case, generates the set of accesses for both *allocated* and *accessed* elements. Figure 9.1(a) shows the first set of repetition pattern of local addresses of the set of accesses for *allocated* elements. The numbers in boxes are the set of elements *accessed*, and the pattern of repetition of these elements is shown in Figure 9.1(b). In a compressed space we need to locate the position of these *accessed* elements, in the a set of *allocated* elements. So we record the positions of these entries in a separate table as shown in Figure 9.1(c). The main objective of these algorithms is to generate this switching table that helps in switching from the template space to the local compressed space. The construction of these switching tables is discussed in the following sections.

9.1 *Itable:* an algorithm that constructs a table of offsets

The main idea behind this algorithm is to construct a table of offsets, which is used to help switch from the non-compressed space to the compressed space. The algorithm exploits the repetition of accesses of both *allocated* and *accessed* elements. This algorithm first generates a two dimensional view of the set of accesses of both *allocated* and *accessed* elements for the non-compressed space. This is done by the application of the one-level algorithm with input strides being a and $a * s$ respectively. Recording the two-dimensional view of these sets does not incur any extra overhead such as expensive division and

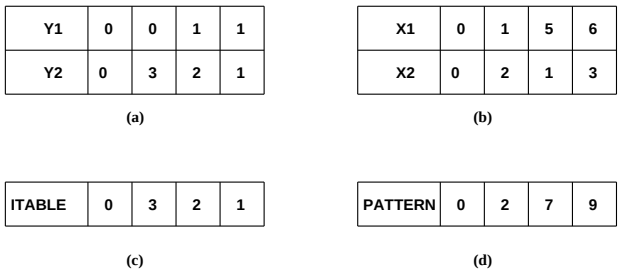

Fig. 9.2. Two dimensional coordinates of *allocated* and *accessed* elements along with *Itable* and *two-level* access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

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

```
1   d_1 ← gcd(a, pk)
2   d_2 ← gcd(a * s, pk)
3   (length_a, Y_1, Y_2) ← one_level(p, k, a, m, d_1)
4   (length_as, X_1, X_2) ← one_level(p, k, a * s, m, d_2)
5   for i = 0, length_a − 1 do
6       Itable[Y_2[i]] ← i
7   enddo
8   a_cou ← a / d_1
9   for i = 0, length_as − 1 do
10      Two_Level[i] ← ⌊X_1[i] / a_cou⌋ * length_a + Itable[X_2[i]]
11  enddo
12  return Two_level
```

Fig. 9.3. Algorithm that constructs the *Itable* for determining the two-level access pattern table

---

modular operations due to the way we generate the set of accesses using the one-level algorithm.

Figures 9.2(a) and (b) lists the two-dimensional coordinates of both *accessed* and *allocated* elements for the first set of pattern repetition for the example discussed previously. The second coordinates of both these sets indicate the offsets of the elements from the beginning of each course. A quick glance clearly indicates that the *accessed* elements are a subset of *allocated* elements. Using this information the algorithm first builds a table of offsets called the *Itable* for the first repetition pattern of allocated elements. The allocated access pattern repeats itself after every $\frac{a}{\gcd(a,pk)}$ courses. This table

records the order in which the offsets of *allocated* elements are accessed in lexicographic order.

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

A detailed listing of the algorithm is as shown in the Figure 9.3. Lines 1–4 generate the pattern tables for the case when stride is a and $a * s$. These tables record the two dimensional indices of elements accessed. Y_1 and Y_2 hold the two dimensional coordinates of the *allocated* elements while X_1 and X_2 hold the two dimensional coordinates for the *accessed* elements. Lines 5–7 construct the *Itable* that records the positions of offsets of *allocated* elements accessed in lexicographic order. The length of this table is always k. Lines 9–11 generate the two-level pattern table. For each element in the *accessed* element set, a corresponding entry in the *Itable* will help determine the location of this element in the *allocated* set.

Let us consider the example in Figure 8.1. We see that elements 0, 1, 6, 11 have offsets 0, 3, 2, 1 respectively from the beginning of the course. The two-dimensional coordinates for both *allocated* and *accessed* elements are as shown in Figures 9.2(a) and (b). Based on the entries in the Y_2 table, the *Itable* is constructed and is as shown in Figure 9.2(c). In this case the second coordinates of the *allocated* elements are same as that of the *Itable*, but in general the entries in the *Itable* depends on the value of $\gcd(a, pk)$. The *Itable* is always of size k, as there could be a maximum of k elements for each pattern of repetition. In order to construct the *Two-Level* pattern table, let us consider the third entry $(5, 1)$ from the table of *accessed elements* as shown in Figure 9.2(b). This means that the accessed element lies in course number 5 and hence falls in the second repetition block of *allocated* elements. The value 1 in the second coordinate corresponds to the offset of the *accessed* element from the beginning of the course. This serves as an index into the *Itable*. Hence the value at position 1 of the *Itable* will yield 3 as shown in Figure 9.2(c). This value gives the position of the *accessed* element in that particular repetition block. Since we know the number of elements present in a single block (which corresponds to 4 in this example), we can simply evaluate $4 * 1 + 3 = 7$, which gives us the position of the third element accessed in the

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

```
1  d_1 ← gcd(a, pk)
2  d_2 ← gcd(a * s, pk)
3  (length_a, Y_1, Y_2) ← one_level(p, k, a, m, d_1)
4  (length_as, X_1, X_2) ← one_level(p, k, a * s, m, d_2)
5  a_cou ← a / d_1
6  first ← X_1[0]
7  tmp_1 ← length_a * first / a_cou
8  tmp_2 ← first mod a_cou
9  for i = tmp_2 to a_cou − 1 do
10     lookup_acc_1[first] ← tmp_1
11     first ← first + 1
12 enddo
13 tmp_1 ← tmp_1 + length_a
14 last ← X_1[length_as − 1]
15 while (first ≤ last) do
16     i ← 0
17     while (i < a_cou and first ≤ last) do
18        lookup_acc_1[first] ← tmp_1
19        i ← i + 1
20        first ← first + 1
21     enddo
22     tmp_1 ← tmp_1 + length_a
23 enddo
24 for i = 0 to length_a − 1 do
25     itable[Y_2[i]] ← i
26 enddo
27 for i = 0 to length_as − 1 do
28     Two_Level[i] ← lookup_acc_1[X_1[i]] + itable[X_2[i]]
29 enddo
30 return Two_level
```

Fig. 9.4. *Itable**, a faster algorithm to compute the *itable* for determining the two-level access pattern by substituting integer divides with table lookups

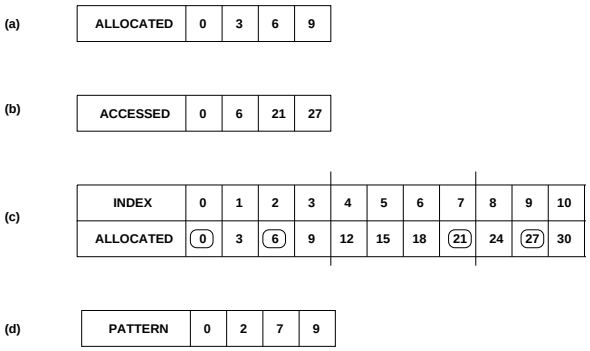

Fig. 9.5. Local addresses of *allocated* and *accessed* elements, the replicated *allocated* table, along with two-level access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

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

```
1  d1 ← gcd(a, pk)
2  d2 ← gcd(as, pk)
3  (length_a, pattern_a) ← one_level(p, k, a, m, d1)
4  (length_as, pattern_as) ← one_level(p, k, a * s, m, d2)
5  Factor of replication f ← (a s k / d2) / (a k / d1)
6  Replicate pattern_a by factor f
7  i ← 0; j ← 0
8  while (j < length_as) do
9      if (pattern_as[j] = pattern_a[i]) then
10         Two_level[j] ← i
11         j ← j + 1
12     endif
13     i ← i + 1
14 enddo
15 return Two_level
```

Fig. 9.6. Algorithm that constructs a two-level access pattern table using linear search method

compressed local space. Figure 9.2(d) shows the positions of *accessed* elements among a set of *allocated* elements for the first set of pattern repetition. Next, we discuss some improvements to the algorithm that constructs the *Itable*.

9.2 Optimization of the *Itable* method

As can be seen from the algorithm in Figure 9.3, line 10 that computes the Two_Level pattern table includes expensive integer operations, an integer multiply and an integer divide. Here, we explore the possibility of reducing the number of these expensive operations in the *itable* algorithm. The key point to note is that in the expression $\left\lfloor \frac{X_1[i]}{a_{cou}} \right\rfloor * length_a$, both the quantities a_{cou} and $length_a$ are loop invariant constants. We improve the performance here by using table lookups. Instead of $length_{as}$ divisions we need only one division and one mod operation; these are needed to compute just the first entry $\left\lfloor \frac{X_1[0]}{a_{cou}} \right\rfloor * length_a$ and the rest can be calculated by exploiting the properties of numbers. This optimization of the *Itable* method is shown in Figure 9.4.

9.3 Search-based algorithms

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

The first step in performing these methods is to run the one-level algorithm to obtain the local addresses of the set of accesses for the first pattern of repetition for both *allocated* and *accessed* elements. These entries are in lexicographic order and are assumed to be stored in $pattern_a$ and $pattern_{as}$ tables respectively. Note that unlike other techniques we not use the memory gap table here since a significant fraction of the work involved in address generation for two-level mapping is in the recovery of the actual elements from the memory gap table [5]. These tables are as shown in Figures 9.5(a) and (b) for the example in Figure 8.1. The entries in these table correspond to the local addresses in a non-compressed template space. The table size of the former depends on $\frac{k}{\gcd(a,pk)}$, whereas that of the latter table depends on $\frac{k}{\gcd(as,pk)}$.

Here we see that not all elements in the *accessed* set are present in the *allocated* set for the first pattern of repetition. This is due to the fact that the *accessed* elements lie in different repetition blocks of the local address space. Hence we need to expand the set of *allocated* elements in order to represent all the elements in the *accessed* table, before the pattern starts repeating.

The total number of elements in the first repetition block of *allocated* table in the uncompressed space is $\frac{ak}{\gcd(a,pk)}$ whereas the total number of elements for the *accessed* table is $\frac{ask}{\gcd(as,pk)}$. Hence the factor needed to expand the *allocated* elements table is $\frac{\frac{ask}{\gcd(as,pk)}}{\frac{ak}{\gcd(a,pk)}} = s\frac{\gcd(a,pk)}{\gcd(as,pk)}$. Performing the required expansion is straight forward. It involves replicating the first set of *allocated* elements as many times as the factor of replication. This is accomplished by copying elements one at a time from the first pattern of *allocated* elements to the extended memory space with a suitable increment. Another possibility is to replicate on demand.

A search now has to be performed to locate the position of an *accessed* element in this new replicated table. Figure 9.5(c) shows the replicated table after expansion for the example discussed previously. The factor for replication in this case was found to be 3. Since the length of the table that holds the addresses of the *accessed* elements is never greater than the length of the table that holds the *allocated* elements, the algorithm needs to find the locations of common elements from two tables of different size. Several search algorithms can be implemented for finding the locations of *accessed* elements. We discuss an algorithm based on linear search. Several search algorithms (with and without the need for replication of the set of elements) that differ mainly in their complexities and the speed of execution can be found elsewhere [9, 31, 42].

Linear search The algorithm for linear search builds the two-level pattern table needed to switch from the template space to the local address space. Figure 9.6 lists the complete algorithm. Lines 1–4 discusses the build up of the *accessed* and *allocated* table. The next step is to find the factor for replication and is as shown in Line 5. This factor is used to replicate the *allocated* table. Once replication is performed, we now need to perform a simple search in order to locate the positions of each *accessed* element in the *allocated* table. Lines 8–14 shows the search algorithm. For each entry in the *accessed* table, it determines the location of this element in the replicated *allocated* set. As and when the location is determined, the position is recorded into a *Two-Level* pattern table. The entries in this table reflects the local address of the *accessed* element in the compressed space.

Figure 9.5 can be used to explain the functioning of this algorithm. Let us consider the element 21 from the *accessed* set as shown in Figure 9.5(b). The search involves finding the position of this element in the replicated set as shown in Figure 9.5(c). This element can be found at location 7. This entry is then stored in the *Two-level* pattern table. The complete pattern table for the example is as shown in Figure 9.5(d). Since the search is performed on a sorted table of length $f * len_a$ and no element of this table is accessed more than once, the complexity of the algorithm is $O(f * len_a)$. In addition to the linear search method discussed above, one could use binary search. Also, it

is possible to avoid replicating the elements by generating them on demand in the course of a search [9, 31, 42].

10. Experimental Results for Two-level Mapping

In order to compare all the above mentioned methods, we ran our experiments on a varying number of problem parameters. These experiments were done on a Sun UltraSparc 1 Workstation with Solaris 2. The compiler used was the Sun C compiler `cc` with the `-xO2` flag. Though the experiments were run for a large set of input values only a limited number of results and times are shown here. In each, case we report only the times needed to construct the two-level table, excluding the times taken construct the two one-level pattern tables as done in all the techniques discussed in this paper. We fixed the number of processors to $p = 32$ in all our experiments. For each value of alignment stride a, we varied both the block-size k and the access stride s. The optimized version of the algorithm that constructs the *Itable*, i.e., the version that replaces extensive divisions by table lookups (Figure 9.4) described in Section 9.2 is referred to as *Itable**. The best of the search algorithms that performs replication was chosen for the results and is referred to as *search* in the tables. The search algorithm that does not perform replication is termed as *norep* in the results. The method due to Chatterjee et al. [5] is termed as *riacs*. Tables 10.1–10.4 show the time it takes to build the two-level pattern tables.

The results indicate that the times taken by all the above mentioned methods depend on the value of k, s and a. If the access and alignment strides are small, the *Itable** and the *search* techniques are competitive; this is because the time taken for replication and the overhead in performing a search is very minimal. But as s and k increases we notice that the *search* starts performing worse. This is because as s and k start increasing the time for replication in the *search* dominates over search and renders this method inefficient. The construction of the *itable* forms the major part of time taken for two-level pattern build up. This construction is purely a function of a and k and not of s. Hence as s increases the times for *Itable** does not vary widely. The method *Itable** performs the best over a wide range of parameters.

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

Table 10.1. Table generation times (μs) for two-level mapping $p = 32$, $a = 2$

k	*s=3*				*s=5*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	1	3	18	2	1	3	18
8	2	2	6	26	2	2	6	26
16	2	2	12	43	2	4	12	43
32	4	4	24	81	4	6	25	79
64	6	7	51	153	6	12	51	150
128	10	14	107	297	10	23	107	299
256	21	28	222	589	21	46	219	586

k	*s=7*				*s=11*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	2	3	18	2	2	3	18
8	2	3	6	26	2	3	6	26
16	2	4	12	43	2	6	12	44
32	4	8	25	79	4	12	25	79
64	6	16	51	150	6	23	52	153
128	10	32	107	299	11	49	107	299
256	21	64	223	588	21	99	223	589

k	*s=23*				*s=99*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	2	3	18	2	6	3	18
8	2	5	6	26	2	12	6	26
16	2	9	12	44	2	23	12	45
32	4	19	25	81	4	47	25	80
64	6	40	51	154	6	97	52	154
128	10	85	108	299	10	210	108	300
256	21	183	224	590	22	446	226	594

11. Other Problems in Code Generation

In this section, we provide an overview of other work from our group on several problems in code generation and runtime support for data-parallel languages. These include our work on communication generation, code generation for complex subscripts, runtime data structures, support for operations on regular sections and array redistribution.

11.1 Communication generation

In addition to problems in address generation, we have explored techniques for communication generation and optimization [40, 42, 43, 44]. A compiler for languages such as HPF that generates node code (for each processor) has also to compute the sequence of sends and receives for a given processor to access non-local data. While the address generation problem has received

Table 10.2. Table generation times (μs) for two-level mapping $p = 32$, $a = 3$

k	*s=3*				*s=5*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	2	6	28	2	2	6	28
8	3	3	12	44	3	4	12	43
16	4	4	24	81	4	6	24	80
32	7	8	50	152	7	12	51	150
64	11	14	107	297	10	23	104	298
128	21	28	223	588	21	46	222	591
256	42	54	463	1167	43	90	458	1171

k	*s=7*				*s=11*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	3	6	28	2	3	6	26
8	3	5	12	44	3	6	12	44
16	4	8	25	79	4	12	25	80
32	7	16	51	151	7	23	51	154
64	11	32	108	298	11	50	108	300
128	21	64	221	588	22	99	223	589
256	42	128	463	1173	42	203	464	1171

k	*s=23*				*s=99*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	5	6	26	2	12	6	26
8	3	9	12	44	3	23	12	44
16	4	19	25	81	4	47	25	82
32	7	40	51	152	7	98	52	151
64	11	84	109	299	11	208	108	302
128	21	179	225	591	22	447	226	595
256	44	411	463	1173	44	932	469	1181

Table 10.3. Table generation times (μs) for two-level mapping $p = 32$, $a = 5$

k	*s=3*				*s=5*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	1	6	26	2	2	6	27
8	3	2	12	44	3	4	12	44
16	4	4	24	80	4	7	25	81
32	7	8	51	153	7	12	51	151
64	11	15	107	298	11	23	107	299
128	21	28	220	589	21	46	220	589
256	42	54	462	1169	42	90	462	1171

k	*s=7*				*s=11*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	3	6	26	2	3	6	26
8	3	4	12	44	3	6	12	44
16	4	8	25	79	4	12	25	81
32	7	17	52	154	7	23	51	152
64	11	32	107	298	11	49	107	300
128	21	64	222	589	21	99	223	592
256	42	129	463	1174	42	203	460	1173

k	*s=23*				*s=99*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	5	6	26	2	12	6	26
8	3	9	12	44	3	23	12	44
16	4	19	25	79	4	47	25	81
32	7	39	52	154	7	97	52	152
64	12	84	108	300	12	210	109	302
128	21	179	225	593	21	448	227	596
256	43	409	462	1176	44	932	470	1178

Table 10.4. Table generation times (μs) for two-level mapping $p = 32$, $a = 9$

k	*s=3*				*s=5*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	1	6	26	2	2	6	26
8	3	2	12	45	3	4	12	45
16	4	4	24	81	4	6	24	80
32	7	8	51	150	7	12	51	153
64	11	15	107	299	11	23	106	299
128	21	28	222	587	21	46	220	591
256	42	54	461	1167	42	90	464	1167

k	*s=7*				*s=11*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	2	6	26	2	3	6	26
8	3	5	12	44	3	6	12	44
16	4	9	25	79	4	12	25	81
32	7	17	52	153	7	23	51	151
64	11	32	105	299	11	50	108	299
128	21	64	224	592	21	98	223	594
256	42	128	462	1172	42	203	461	1173

k	*s=23*				*s=99*			
	*Itable**	*search*	*norep*	*riacs*	*Itable**	*search*	*norep*	*riacs*
4	2	5	6	26	2	12	6	26
8	3	9	12	45	3	24	12	44
16	4	19	25	80	4	47	25	82
32	7	40	52	154	7	99	52	152
64	11	87	108	300	11	209	108	303
128	21	183	224	592	22	449	222	598
256	42	408	463	1181	43	928	468	1185

much attention, issues in communication generation have received limited attention; see [15] and [18] for examples. A novel approach for the management of communication sets and strategies for local storage of remote references is presented in [43, 42]. In addition to algorithms for deriving communication patterns [40, 42, 44], two schemes that extend the notion of a local array by providing storage for non-local elements (called overlap regions) interspersed throughout the storage for the local portion are presented [43, 42]. The two schemes, namely course padding and column padding enhance locality of reference significantly at the cost of a small overhead due to unpacking of messages. The performance of these schemes are compared to the traditional buffer-based approach and improvements of up to 30% in total time are demonstrated. Several message optimizations such as offset communication, message aggregation and coalescing are also discussed.

11.2 Union and difference of regular sections

Operations on regular sections are very common in code generation. The intersection operation on regular sections is easy (since regular sections are closed under intersection). Union and difference of regular sections are needed for efficient generation of communications sets; unfortunately, regular sections are not closed under union and difference operations. We [9, 27] present an efficient runtime technique for supporting support for union and other operations on regular sections. These deal with both the generation of the pattern under these operations as well as with the efficient code that enumerates the resulting sets using the patterns.

11.3 Code generation for complex subscripts

The techniques presented in this chapter assumed simple subscript functions. Array references with arbitrary affine subscripts can make the task of compilers for such languages highly involved. Work from our group [9, 26, 29, 30, 42] deals with the efficient address generation in programs with array references having two types of commonly encountered affine references, namely coupled subscripts and subscripts containing multiple induction variables (MIVs). These methods utilize the repetitive pattern of the memory accesses. In the case of MIV, we address this issue by presenting runtime techniques which enumerate the set of addresses in lexicographic order. Our approach to the problem incorporates a general approach of computing in $O(k)$ time, the start element on a processor for a given global start element. Several methods are proposed and evaluated here for generating the access sequences for MIV based on problem parameters. With coupled subscripts, we present two construction techniques, namely searching and hashing which minimize the time needed to construct the tables. Extensive experiments were conducted and the results were then compared with other approaches to demonstrate the efficiency of our approach.

11.4 Data structures for runtime efficiency

In addition to algorithms for address sequence generation, we addressed the problem of how best to use the address sequences in [8, 9]. Efficient techniques for generating node code on distributed-memory machines is important. For array sections, node code generation must exploit the repetitive access pattern exhibited by the accesses to distributed arrays. Several techniques for the efficient enumeration of the access pattern already exist. But only one paper [17] so far addresses the effect of the data structures used in representing the access sequence on the execution time. In [8, 9], we present several new data structures along with node code that is suitable for both DO loops and FORALL constructs. The methods, namely *strip-mining* and *table compression* facilitate the generation of time-efficient code for execution on each processor. While strip-mining views the problem as a double nested loop, table compression proves to be a worthwhile data structure for faster execution. The underlying theory behind the data structures introduced is explained and their effects on all possible set of problem parameters is observed. Extensive experimental results show the efficacy of our approach. The results compare very favorably with the results of the earlier methods proposed by Kennedy et al. [16] and Chatterjee et al. [5].

11.5 Array redistribution

Array redistribution is used in languages such as High Performance Fortran to dynamically change the distribution of arrays across processors. Performing array redistribution incurs two overheads: (1) an *indexing overhead* for determining the set of processors to communicate with and the array elements to be communicated, and (2) a *communication overhead* for performing the necessary irregular all-to-many personalized communication. We have presented efficient runtime methods for performing array redistribution [14, 35]. In order to reduce the indexing overhead, precise closed forms for enumerating the processors to communicate with and the array elements to be communicated are developed for two special cases of array redistribution involving block-cyclically distributed arrays. The general array redistribution problem for block-cyclically distributed arrays can be expressed in terms of these special cases. Using the developed closed forms, a distributed algorithm for scheduling the irregular communication for redistribution is developed. The generated schedule eliminates node contention and incurs the least communication overhead. The scheduling algorithm has an asymptotically lower scheduling overhead than techniques presented in the literature. Following this, we have developed efficient table-based runtime techniques (based on integer lattices) that incur negligible cost [9, 28].

12. Summary and Conclusions

The success of data parallel languages such as High Performance Fortran and Fortran D critically depends on efficient compiler and runtime support. In this chapter we presented efficient compiler algorithms for generating local memory access patterns 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 to the one-level mapping problem 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 is $O(\log \min(s, pk) + \min(s, k))$ and requires finding $\min(s + 1, k)$ points in the lattice. Kennedy et al.'s algorithm for basis determination is $O(\log \min(s, pk) + k)$ and requires finding $2k - 1$ points in the lattice. Our loop nest based technique used for address enumeration chooses the best strategy as a function of the basis vectors, unlike [16]. Experimental results comparing the times for our basis determination technique and that of Kennedy et al. shows that our solution is 2 to 9 times faster for large block sizes. For the two-level mapping problem, we presented three new algorithms. Experimental comparisons with other techniques show that our solutions to the two-level mapping problem are significantly faster. In addition to these algorithms, we provided an overview of other work from our group on several problems such as

– efficient basis vector generation using an $O(\log \min(s, pk))$ algorithm [25];
– communication generation [40, 42, 43, 44];
– code generation for complex subscripts [9, 26, 29, 30, 42];
– effect of data structures for table lookup at runtime [8, 9];
– runtime array redistribution [9, 14, 28, 35]; (and)
– efficient support for union and other operations on regular sections [9, 27].

Work is in progress on the problem of code generation and optimization for general affine access functions in whole programs.

Acknowledgments

This work was supported in part by an NSF Young Investigator Award CCR–9457768 with matching funds from the Portland Group Inc. and the Halliburton Foundation, by an NSF grant CCR–9210422, and by the Louisiana Board of Regents through contracts LEQSF(RF/1995-96) ENH-TR-60 and LEQSF (1991-94)-RD-A-09. I thank Ashwath Thirumalai, Arun Venkatachar, and Swaroop Dutta for their valuable collaboration. I thank Nenad Nedeljkovic, James Stichnoth and Ajay Sethi for their comments on an earlier draft of this chapter. Nenad Nedeljkovic and Ajay Sethi provided the code for the two Rice algorithms and the Sort implementation, and S. Chatterjee provided the code for the RIACS implementation used in experiments on two-level mapping.

References

1. A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. *Scientific Programming,* 6(1):3–28, Spring 1997.
2. P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer,* 28(10):37–47, October 1995.
3. S. Benkner. Handling block-cyclic distributed arrays in Vienna Fortran 90. In *Proc. International Conference on Parallel Architectures and Compilation Techniques,* Limassol, Cyprus, June 1995.
4. B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming,* 1(1):31–50, Fall 1992.
5. S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, 1995.
6. F. Coelho, C. Germain, J. Pazat. State of the art in compiling HPF. *The Data Parallel Programming Model,* G. Perrin and A. Darte (Eds.), Lecture Notes in Computer Science, Volume 1132, pages 104–133, 1996.
7. G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A),* 14:288–297, 1973.
8. S. Dutta and J. Ramanujam. Data structures for efficient execution of programs with block-cyclic distributions. Technical Report TR-96-11-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, Jan. 1997. Preliminary version presented at the *6th Workshop on Compilers for Parallel Computers,* Aachen, Germany, December 1996.
9. S. Dutta. Compilation and run-time techniques for data-parallel programs. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, *in preparation.*
10. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.
11. P. Gruber and C. Lekkerkerker. *Geometry of numbers.* North-Holland Mathematical Library Volume 37, North-Holland, Amsterdam, 1987.
12. S. Gupta, S. Kaushik, C. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing,* 32(2):155–172, February 1996.
13. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming,* 2(1-2):1–170, 1993.
14. S. Kaushik, C. Huang, J. Ramanujam, and P. Sadayappan. Multiphase array redistribution: Modeling and Evaluation. Technical Report OSU-CISRC-9/94-TR52, Department of Computer and Information Science, The Ohio State University, September 1994. A short version appears in *Proc. 9th International Parallel Processing Symposium,* Santa Barbara, CA, pages 441–445, April 1995.
15. S. Kaushik. Compile-time and run-time strategies for array statement execution on distributed-memory machines. Ph.D. Thesis, Department of Computer and Information Science, The Ohio State University, 1995.
16. 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, pages 102–111, July 1995.
17. K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proc. ACM International Conference on Supercomputing,* Madrid, Spain, pages 180–184, July 1995.
18. K. Kennedy, N. Nedeljkovic, and A. Sethi. Communication generation for `CYCLIC`(k) distributions. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, B. Szymanski and B. Sinharoy (Eds.), Kluwer Academic Publishers, 1996.
19. C. Koelbel. Compile-time generation of communication for scientific programs. In *Proc. Supercomputing '91*, Albuquerque, NM, pages 101–110, November 1991.
20. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran handbook*. The MIT Press, 1994.
21. T. MacDonald, D. Pase, and A. Meltzer. Addressing in Cray Research's MPP Fortran. In *Proceedings of the 3rd Workshop on Compilers for Parallel Computers,* Vienna, Austria, pages 161–172, July 1992.
22. S. Midkiff. Local iteration set computation for block-cyclic distributions. In *Proc. International Conference on Parallel Processing*. Vol. II, pages 77–84, August 1995.
23. J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing 92,* Minneapolis, MN, pages 214–223, November 1992.
24. J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing,* 9(4):365-389, December 1995.
25. J. Ramanujam. Efficient computation of basis vectors of the address sequence lattice. Submitted for publication, 1997.
26. J. Ramanujam and S. Dutta. Code generation for coupled subscripts with block-cyclic distributions. Technical Report TR-96-07-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, July 1996.
27. J. Ramanujam and S. Dutta. Runtime solutions to operations on regular sections. Technical Report TR-96-12-03, Dept. of Elec. & Comp. Engineering, Louisiana State University, December 1996.
28. J. Ramanujam and S. Dutta. Efficient runtime array redistribution. Technical Report TR-97-01-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, January 1997.
29. J. Ramanujam, S. Dutta, and A. Venkatachar. Code generation for complex subscripts in data-parallel programs. To appear in *Proc. 10th Workshop on Languages and Compilers for Parallel Computing,* Z. Li et al., (Eds.), Minneapolis, MN, Springer-Verlag, 1997.
30. J. Ramanujam and A. Venkatachar. Code generation for complex subscripts with multiple induction variables in the presence of block-cyclic distributions. Technical Report TR-96-03-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, March 1996.
31. J. Ramanujam, A. Venkatachar, and S. Dutta. Efficient address sequence generation for two-level mappings in High Performance Fortran. Submitted for publication, 1997.
32. C. van Reeuwijk, H. Sips, W. Denissen, and E. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems,* 7(9):897–914, September 1996.
33. J. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

34. J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing,* 21(1):150–159, April 1994.
35. R. Thakur, A. Choudhary and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems,* 7(6):587–594, June 1996.
36. A. Thirumalai. Code generation and optimization for High Performance Fortran. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, August 1995.
37. 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, November 1994; revised August 1995
38. 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, pages 581–605, June 1995.
39. A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs using integer lattices. In *Languages and Compilers for Parallel Computing,* C.-H. Huang et al. (Editors), Lecture Notes in Computer Science, Vol. 1033, pages 191–208, Springer-Verlag, 1996.
40. 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 (Eds.), Kluwer Academic Publishers, 1996.
41. A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data-parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing,* 38(2):188–203, November 1996.
42. A. Venkatachar. Efficient address and communication generation for data-parallel programs. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, December 1996.
43. A. Venkatachar, J. Ramanujam and A. Thirumalai. Generalized overlap regions for communication optimization in data parallel programs. In *Languages and Compilers for Parallel Computing,* D. Sehr et al. (Editors), Lecture Notes in Computer Science, Vol. 1239, pages 404–419, Springer-Verlag, 1997.
44. A. Venkatachar, J. Ramanujam, and A. Thirumalai. Communication generation for block-cyclic distributions. *Parallel Processing Letters,* (to appear) 1997.
45. L. Wang, J. Stichnoth, and S. Chatterjee. Runtime performance of parallel array assignment: An empirical study. In *Proc. Supercomputing 96,* Pittsburgh, PA, November 1996.
46. H. Wijshoff. *Data organization in parallel computers.* Kluwer Academic Publishers, 1989.