Fast Address Sequence Generation for Data-Parallel Programs Using Integer Lattices

Ashwath Thirumalai and J. Ramanujam

Dept. of Electrical & Computer Engineering
Louisiana State University, Baton Rouge, LA 70803
({ash,jxr}@gate.ee.lsu.edu)

Abstract. 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 paper, we present a novel approach based on integer lattices. The set of elements referenced can be generated by integer linear combinations of basis vectors. Our linear algorithm determines the basis vectors as a function of the mapping. Using the basis vectors, we derive a loop nest that enumerates the addresses, which are points in the lattice generated by the basis vectors. Experimental results show that our approach is better than that of a recent linear time solution to this problem.

1 Introduction

Programming massively parallel distributed memory machines involves partitioning data and computation across processors. Languages such as Fortran D [4], Vienna Fortran [2] and High Performance Fortran (HPF) [7] include directives—such as align and distribute—that describe how data is distributed among the processors in a distributed-memory multiprocessor. For example, in HPF, arrays are mapped to processors in two steps: in the first step, arrays are *aligned* to an abstract discrete Cartesian grid called a *template;* 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, a HPF compiler must generate code for each processor (node code) that accesses only 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 paper 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. Our solution is based on viewing the access sequence as an integer lattice, and involves the derivation of a suitable set of basis vectors for the lattice. Given the lattice basis, we enumerate the lattice by using loop nests; this allows us to generate efficient code that incurs negligible runtime overhead in determining the access pattern. Chatterjee *et al.* [3] presented an $O(k \log k)$ algorithm (where k is the

[1] Supported in part by an NSF Young Investigator Award CCR–9457768, and NSF grant CCR–9210422, and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

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

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

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

(b): Local layout of array shown in Fig. 1(a) in Processor m.

Fig. 1. Global and local addresses for data mappings

block size – see Section 2 for definition) for this problem; ours is an $O(k)$ algorithm. Recently, Kennedy *et al.* [9] have also presented an $O(k)$ algorithm. 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.*

This paper is organized as follows. In Section 2, we present the problem setting and discuss related work. Section 3 outlines our approach using lattices and presents key mathematical results which are used later in the paper. 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 enumeration 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.* Section 8 concludes with a summary and points to further research directions.

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 $T(ai+b)$, then the alignment stride $a = 1$ and the alignment offset $b = 0$. Further let this template 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 [7]. If $k = 1$, the distribution is called CYCLIC, and if $k = \frac{N}{p}$, where N is the size of the template, the distribution is called a BLOCK distribution. We assume that arrays have a lower limit of zero, and processors and local addresses are numbered from zero onward. This mapping of the elements of A to the processor memories is shown in the Fig. 1(a). Though the elements of A are stored in the processor memories in a linear fashion as shown in Fig. 1(b), we adopt a two-dimensional view of the storage allocated for the array as shown in Fig. 1(a). We view the global addresses as being organized in terms of courses; each course consists of pk elements. 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.

Processor 0				Processor 1				Processor 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. 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.

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. 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). Fig. 2 shows the allocation of the array elements along with the corresponding global addresses. The array elements accessed are marked and the corresponding local addresses are shown in Fig. 2. 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 enumerate this sequence.

Several papers have addressed this code generation problem. Koelbel [10] derived techniques for compile-time address and communication generation for `BLOCK` and `CYCLIC` distribution for non-unit stride accesses. Chatterjee *et al.* [3] derived a technique that identifies a repeating access pattern, which is characterized as a finite-state machine. Their $O(k\ log\ k)$ algorithm involves a solution of k linear Diophantine equations to determine the pattern of address accessed, followed by sorting these addresses to derive the accesses in linear order. In this paper, we show that sorting the elements is not required to generate the addresses in lexicographic order. In an exhaustive study of this problem, Stichnoth [13] presented a framework to enumerate local addresses and generation of communication sets. Gupta *et al.* [6] derived the *virtual-block* and *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). Ancourt *et al.* [1] use a linear algebra framework to generate code for fully parallel loops in HPF and do not exploit the repetition of the access sequence. Recently, Kennedy *et al.* [9] derived an $O(k)$ algorithm that determines the basis vectors by locating $2k - 1$ addresses accessed using the solutions of $2k - 1$ linear Diophantine equations. The improvement over [3] comes from avoiding the sorting step at the expense of solving an additional set of $k - 1$ linear Diophantine equations. In this paper, we show that solving $2k - 1$ equations is not required. Our algorithm requires finding only $\min(s, k + 1)$ addresses that are accessed by the array section. In addition, our lattice enumeration algorithm (given the basis vectors) is different from and faster than [9].

3 A Lattice Based Approach for Address Generation

In this section 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. In this section we also provide a few optimizations of our basis determination algorithm.

3.1 Assumptions

We present our approach to address generation for an alignment stride, $a = 1$. For $a > 1$, we use an approach similar to the one developed in [3], which involves two applications of our algorithm. Therefore, we do not discuss non-unit alignment strides. Communication sets for array statements can be derived by two applications of our algorithm for address generation [14].

We assume that A is identically aligned to the template T. As it is evident from Fig. 1(a), we assign a block of k cells of the template to each of the p processors and then wrap around and assign the rest of the cells in a similar fashion. As mentioned earlier, we treat the global address space as a two dimensional space and every element of an array $A(i)$ has an address of the form $(x, y) = (i \operatorname{div} pk, i \bmod pk)$ in this space. Here x gives the course number to which this element belongs and y gives the offset of the element in that course. We refer to the two-dimensional view of an array A as A_{2D}; this notation is used throughout this paper.

3.2 Lattices

We use the following definitions of a lattice and its basis [5].

Definition 1. A set of points $\mathbf{x_1}, \mathbf{x_2}, \cdots, \mathbf{x_k}$ in $\mathbb{R}^n$ is said to be *independent* if these points do not belong to a linear subspace of $\mathbb{R}^n$ of dimension less than k.

For $k = n$, this is equivalent to the condition that the determinant of the matrix X whose columns are $\mathbf{x_i}$, $i = 1, \ldots, n$ 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 [5]; see [5] for a proof.

Definition 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. A matrix $B = \mathbf{b_1 b_2} \cdots \mathbf{b_n}$ is called a basis matrix.

Definition 3. Let Λ be a discrete subspace of $\mathbb{R}^n$ which is not contained in an $(n-1)$-dimensional linear subspace of $\mathbb{R}^n$. Then Λ is a lattice.

We refer to the set of global addresses (elements) over all processors accessed by $A(\ell : h : s)$ with distribution parameters (p, k) as $\Gamma = \langle A(\ell : h : s), p, k \rangle$. We refer to the set of local addresses accessed by processor m in executing its portion of $A(\ell : h : s)$ with

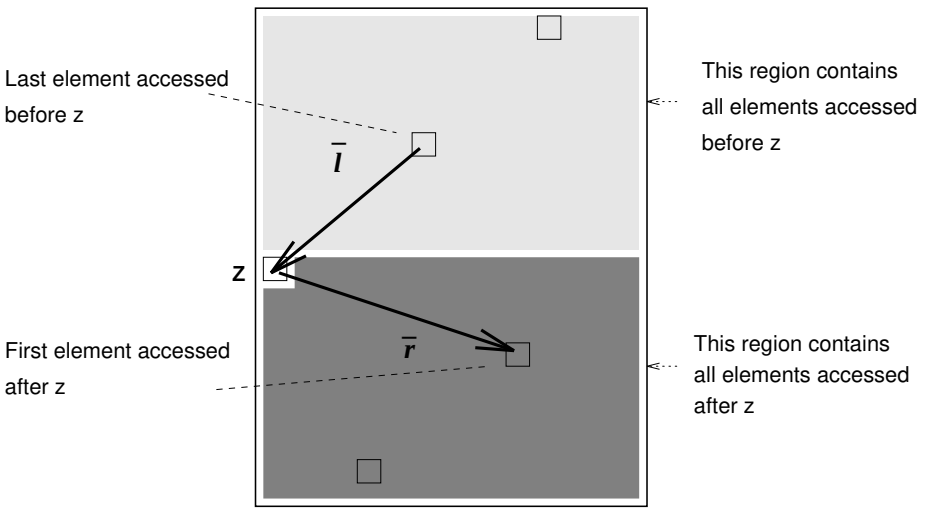

Fig. 3. Explanation of basis determination

distribution parameters (p, k) as Γ_m. By construction, Γ is a discrete subgroup of $\mathbb{R}^2$ and in general, is not contained in a 1-dimensional linear subspace of $\mathbb{R}^2$; if a single vector can be used to generate the elements accessed, our algorithm handles this as a special case. Thus, without loss of generality, $\Gamma = \langle A(\ell : h : s), p, k \rangle$ is a lattice. Similarly Γ_m is also a lattice.

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

1. find a set of basis vectors of the lattice Γ_m; and
2. enumerate the points in Γ_m using integer linear combinations of the basis vectors.

Our solution to Step 1 (presented in Section 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 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 paper 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 5. Let $\mathbf{b_1}, \mathbf{b_2}, \cdots, \mathbf{b_n}$ be independent points of a lattice Λ in $\mathbb{R}^n$. Then Λ has a basis $\{\mathbf{a_1}, \mathbf{a_2}, \cdots, \mathbf{a_n}\}$ such that

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

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 6. *Let* $\mathbf{b_1}$ *and* $\mathbf{b_2}$ *be independent points of a lattice* Λ *in* $\mathbb{R}^2$ *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 5, we can write

$$\mathbf{b_1} = u_{11}\mathbf{a_1}$$
$$\mathbf{b_2} = u_{12}\mathbf{a_1} + u_{22}\mathbf{a_2}$$

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 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.

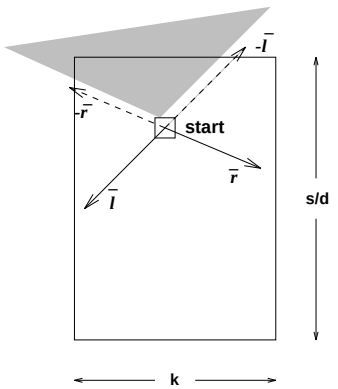

Fig. 4. No lattice point is a non-positive linear combinations of basis vectors

Theorem 7. *The set of points* $\{\mathbf{z_f^{prev}}, \mathbf{z_f}, \mathbf{z_f^{next}}\}$ *generate a basis of* Γ_m.

Proof. From Theorem 6, 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 Fig. 3 on p. 13.5.

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 them contains no other lattice point. Chatterjee *et al.* [3] suggested a way to locate lattice points by solving linear Diophantine equation for each column with accessed elements. For details on their derivation refer [3]. 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 $\mathbf{z}_f$

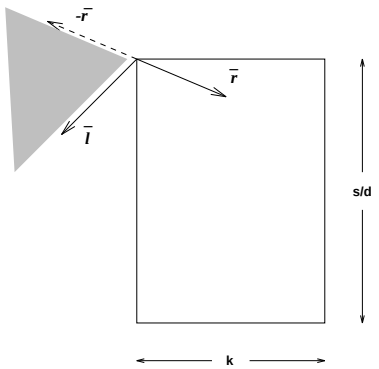

Fig. 5. No lattice points in region spanned by vectors **l** and **-r**

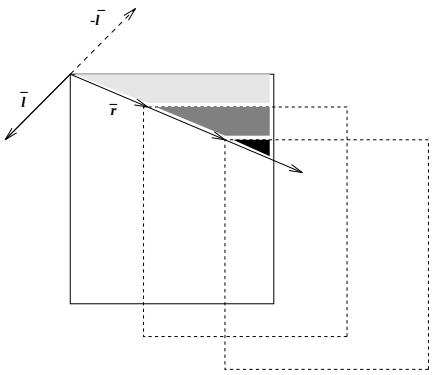

Fig. 6. No lattice points in region spanned by vectors **-l** and **r**

be the first element accessed in column c_f. Since the access pattern on a given processor repeats after $\frac{pks}{\text{GCD}\,(s,pk)}$ elements, the point accessed immediately after $\mathbf{z}_f$ in column c_f is $\mathbf{z}_f + \frac{pks}{\text{GCD}\,(s,pk)}$. Now if c_s is the second column in which an element is accessed on the processor, and $\mathbf{z}_s$ is the first element accessed in it, then without loss of generality $\mathbf{z}_f$, $\mathbf{z}_f + \frac{pks}{\text{GCD}\,(s,pk)}$ 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 Fig. 8. Fig. 7 shows the basis vectors generated as explained above whereas Fig. 8 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

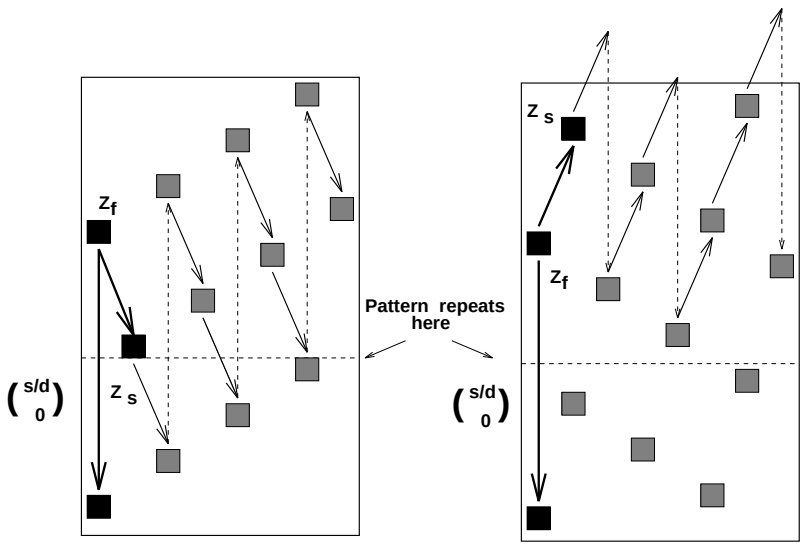

Fig. 7. Starting points in the columns generated by the vectors $\left(\frac{s}{d}, 0\right)$ and $\mathbf{z}_s - \mathbf{z}_f$.

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.

4.2 Extremal Basis Vectors

In this section, we show that the basis vectors generated by our algorithm in Fig. 8 form an extremal set of basis vectors.

Theorem 8. *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* **l** *and* **r**.

Proof. Let **z** be the starting point of the lattice Γ_m. As **r** and **l** are the basis vectors of the lattice, any point **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 Fig. 4 that **q** lies in the region above the start element **z**. This contradicts our earlier assumption that **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 **z** on a processor lies in the first non-empty column.

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

Fig. 8. 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⌋; l_2 ← z_f mod k - bmax mod k
3     r_1 ← ⌊bmin/pk⌋ - ⌊z_f/pk⌋; r_2 ← bmin mod k - z_f mod k
4  else if bmin = pks/d and bmax = z_f then            /* below is empty */
5     l_1 ← ⌊z_f/pk⌋ - ⌊amax/pk⌋; l_2 ← z_f mod k - amax mod k
6     r_1 ← ⌊amin/pk⌋ + s/d - ⌊z_f/pk⌋; r_2 ← amin mod k - z_f mod k
7  else                                          /* above & below not empty */
8     l_1 ← ⌊z_f/pk⌋ - ⌊amax/pk⌋; l_2 ← z_f mod k - amax mod k
9     r_1 ← ⌊bmin/pk⌋ - ⌊z_f/pk⌋; r_2 ← bmin mod k - z_f mod k
10 endif
11 start ← amin
12 end ← bmax
13 return start, end, l_1, l_2, r_1, r_2
```

Fig. 9. `COMPUTE_VECTORS` subroutine for basis vectors determination algorithm

Let $v_1 \geq 0$ and $v_2 < 0$. As shown in Fig. 5, $\mathbf{q}$ 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 Fig. 6), 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 Fig. 6. 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

If $s < k$, it is sufficient to find only $s + 1$ lattice points instead of k lattice points (as in [9]) in order to derive the basis vectors. Our implementation uses this idea.

Line 1 of the algorithm in Fig. 8 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.* [9] is $O(\log min(s, pk) + k)$. 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. Thus, our algorithm is superior to that of Kennedy *et al.* [9].

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 Fig. 1(a) could be written as follows:

do $i = 0, \lfloor \frac{N-1}{pk} \rfloor$
  do $j = 0, pk - 1$
    $A_{2D}(i, j) = \cdots\cdots$
  end do
end do

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

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

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

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

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

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

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

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

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 [16] 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:

$$\left\lceil \frac{(-mk-k+1)r_1}{s} \right\rceil \leq u \leq \left\lfloor \frac{(\frac{s}{d}-1)r_2 - mkr_1}{s} \right\rfloor$$

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

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

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

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

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

$$\left\lceil \frac{mk - ul_2}{r_2} \right\rceil \leq v \leq \left\lfloor \frac{mk + k - 1 - ul_2}{r_2} \right\rfloor$$

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

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

Example 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 Fig. 8 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 Fig. 9. So our algorithm

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

returns $\mathbf{l} = (1, -1)$ and $\mathbf{r} = (8, 3)$ as the basis vectors. The access pattern is shown in Fig. 10. The node code to obtain the access pattern for processor 1 is:

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

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

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

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

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

A closer look at Fig. 10 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 Fig. 8, the `GO-LEFT` scheme is always legal.

Theorem 9. *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 Fig. 8, 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 Fig. 8. The loop nest for $r_1 \neq 0$ is:

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

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

do $u = \left\lceil \frac{mkl_1}{s} \right\rceil, \left\lfloor \frac{(\frac{s}{d}-1)(-l_2)+(mk+k-1)l_1}{s} \right\rfloor$
  do $v = \left\lceil \max\left(\frac{(mk+k-1)-ur_2}{l_2}, 0 \right) \right\rceil, \left\lfloor \min\left(\frac{mk-ur_2}{l_2}, \frac{\frac{s}{d}-1}{l_1} \right) \right\rfloor$
    $A_{2D}[l_1v, r_2u + l_2v] = \cdots\cdots$
  end do
end do

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

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

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

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 **r**, by moving from one parallel line to the next within the family along the vector **l** and the code generated for GO-LEFT enumerates the points that belong to a family of parallel lines along the vector **l**, by moving from one parallel line to the next within the family along the vector **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 Fig. 8. The start line u_{start} and end line u_{end} can be evaluated as follows (using GO-RIGHT enumeration scheme):

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

Hence,

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

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

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

1. Scan all the elements on the first line (u_{start}), starting at the $start$ element and then adding **r** until there are no more elements on this processor.
2. 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}).
3. 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.

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

1. Scan all the elements on the first line (u_{start}), starting at the $start$ element and then adding **l** until there are no more elements on this processor.
2. 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}).
3. 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.

7 Experimental Results

We performed experiments on our pattern generation algorithm on a Sun Sparcstation 10. We used the `acc` compiler using the `-fast` optimization switch; the function `gettimeofday()` was used to measure time. When computing time for 32 processors, we timed the code that computes the access pattern for each processor, and report the maximum time over all processors. 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 [9] and our algorithm include basis and table generation times. Tables 1(a)–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.* [3] (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 [9] algorithms. For larger block sizes, if $s < k$, our algorithm is 6 to 9 times faster than the Rice algorithms because of the need to find $s + 1$ 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 [9]. 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 [9] do not exploit this enumeration choice, our algorithm performs significantly better. In addition, our algorithm is 13 to 60 times faster than the approach of Chatterjee *et al.* [3] for large block sizes.

8 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 paper we presented efficient compiler algorithms for generating local memory access patterns and interprocessor communication sets for the various processors (node code) given the alignment of arrays to a template and a `CYCLIC(k)` distribution of the template onto the

processors. Our solution is based on viewing the access sequence as an integer lattice, and involves the derivation of a suitable set of basis vectors for the lattice. The basis vector determination algorithm 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 [9]. 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. Experiments on several platforms indicate the efficacy of our approach. Work is in progress on the problem of incremental generation of access sequences and communication sets.

Acknowlegdments We thank Nenad Nedeljkovic, Ajay Sethi and Arun Venkatachar for their comments on an earlier draft of this paper.

References

1. A. Ancourt, F. Coelho, F. Irigoin and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proc. of the 4th Workshop on Compilers for Parallel Computers,* Delft, The Netherlands, December 1993.
2. B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming,* 1(1):31–50, Fall 1992.
3. S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data parallel programs. In *Proc. of ACM Symposium on Principles and Practices of Parallel Programming*, pages 149–158, May 1993.
4. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report CRPC-TR90079, Rice University, Dec. 1990.
5. P. M. Gruber and C. G. Lekkerkerker. *Geometry of numbers.* North-Holland Mathematical Library Volume 37, North-Holland, Amsterdam, 1987 (Second Edition).
6. S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical Report OSU-CISRC-94-TR19, The Ohio State University, April 1994.
7. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming,* 2(1-2):1–170, 1993.
8. S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. In *Proc. ACM Intl. Conf. Supercomputing,* July 1994.
9. K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proc. of Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.
10. C. Koelbel. Compile-time generation of communication for scientific programs. In *Supercomputing '91*, pages 101–110, Nov. 1991.
11. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran Handbook*. The MIT Press, 1994.
12. J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing 92,* pages 214–223, Nov. 1992.
13. J. M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, Carnegie-Mellon University, Feb. 1993.
14. A. Thirumalai and J. Ramanujam. Code generation and optimization for array statements in HPF. Technical Report ECE-TR-94-11-02, Louisiana State University, Nov. 1994.
15. A. Thirumalai and J. Ramanujam. An efficient compile-time approach to compute address sequences in data parallel programs. In *Proc. 5th International Workshop on Compilers for Parallel Computers,* Malaga, Spain, pp. 581-605, June 1995.
16. Michael Wolfe. *High performance compilers for parallel computing.* Benjamin/Cummings Publishing Co., 1996.

Table 1. Times (in μs) for our technique (Loop), Right and Zigzag of Rice [9] and the Sort approach [3] on a Sun Sparcstation 10

(a) $p = 32; s = 7$ and $s = 99$

Block Size	$s = 7$				$s = 99$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
4	26	23	23	27	51	47	47	51
8	34	34	35	47	61	59	59	71
16	33	31	32	67	56	58	58	93
32	36	46	47	152	56	64	64	170
64	47	81	84	460	77	91	98	469
128	41	126	131	844	100	149	152	865
256	53	236	245	1639	118	256	276	1657
512	80	465	487	3224	122	454	471	3226
1024	108	892	940	6373	159	889	918	6397

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

Block Size	$s = k - 1$				$s = k + 1$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
4	24	23	23	27	35	31	31	35
8	34	34	35	47	29	26	27	39
16	29	33	33	69	32	33	33	69
32	36	45	45	154	38	45	45	155
64	69	84	87	460	70	82	90	459
128	97	142	141	853	98	135	144	853
256	154	256	255	1642	155	240	255	1642
512	268	487	486	3220	269	459	487	3220
1024	498	946	945	6377	499	894	947	6377

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

Block Size	$s = pk - 1$				$s = pk + 1$			
k	Loop	Right	Zigzag	Sort	Loop	Right	Zigzag	Sort
4	23	21	21	25	21	19	19	23
8	24	24	24	37	22	22	23	35
16	28	31	31	67	25	29	30	65
32	34	45	45	155	31	43	44	147
64	48	73	72	450	43	72	74	449
128	75	128	128	844	67	129	132	845
256	129	240	239	1633	114	243	250	1638
512	237	465	464	3211	210	475	487	3222
1024	455	912	911	6369	403	935	960	6394