Efficient Computation of Address Sequences in Data Parallel Programs Using Closed Forms for Basis Vectors

Ashwath Thirumalai*
Zycad Corporation
47100 Bayside Parkway
Fremont, CA 94538
ashwath@zycad.com

J. Ramanujam†
Dept. of Electrical & Computer Engineering
Louisiana State University
Baton Rouge, LA 70803
jxr@ee.lsu.edu

Corresponding author: J. Ramanujam

Proposed running head: Closed forms for basis vectors of array section lattice

Abstract

Arrays are mapped to processors through a two-step process—alignment followed by distribution—in data-parallel languages such as High Performance Fortran. This process of mapping creates disjoint pieces of the array that are locally owned by each processor. An HPF compiler that generates code for array statements must 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 an approach to the address sequence generation problem using the theory of integer lattices. The set of elements referenced can be generated by integer linear combinations of basis vectors. Unlike other work on this problem, we derive closed form expressions for the basis vectors as a function of the mapping of data. Using these basis vectors and exploiting the fact that there is a repeating pattern in the access sequence, we derive highly optimized code that generates the pattern at runtime. The code generated uses table-lookup of the pattern. Experimental results show that our approach is faster than other solutions to this problem.

Keywords: compiler optimizations, High Performance Fortran, array statements, data distribution, distributed memory machines, address sequence generation, integer lattices, closed forms for basis vectors, lattice enumeration, code generation.

*This work was done when the author was a graduate student at Lousiana State University.

†Supported in part by an NSF Young Investigator Award CCR–9457768 with a match from The Portland Group Inc., an NSF grant CCR–9210422, and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

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 the efforts such as High Performance Fortran (HPF) [9], Fortran D [6], and Vienna Fortran (which grew out of the earlier SUPERB effort) [3] 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, 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, an HPF compiler must generate code (for each processor) that accesses only locally owned pieces directly, and inserts communication for non-local accesses.

In order to generate node code we need to know the sequence of local memory addresses accessed by each processor and the sequence of sends and receives for a given processor to access non-local data. A regular access pattern in terms of the global data structure can become irregular within each locally allocated piece. For each example, an array section $A(\ell\colon h\colon s)$ 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 (node code) 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 uses closed form expressions for a suitable set of basis vectors for the lattice. Exploiting the pattern of repetition in accesses, we generate efficient code at compile time that determines the access pattern with negligible runtime overhead. Other researchers in this area [4, 11, 23] generate basis vectors at runtime; unlike these, our approach uses closed forms for basis vectors. Experiments on several platforms indicate the efficacy of our approach.

Table 1: Symbols used in this paper.

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
Γ	array section lattice
Γ_m	part of Γ incident on processor m

Communication sets for array statements can be derived by two applications of our algorithm for address generation [21, 22, 23, 24, 25]. Hence, we do not discuss communication set generation in this paper.

This paper is organized as follows. In Section 2, we present the problem setting and discuss related work. Section 3 discusses the derivation of basis vectors, and presents a code generation method using the basis vectors. In Section 4, we present efficient solutions for the compile time derivation of the best basis vectors. Experimental results and comparisons with existing solutions are provided in Section 5. Section 6 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 [9]. 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(a). Though the elements of A are stored in the processor memories in a linear fashion as shown in Figure 1(b), we adopt a two-dimensional view of the storage allocated (as used by Chatterjee *et al.* [4]) for the array as shown in Figure 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. An array assignment statement (in HPF) is of the form:

$$A(\ell{:}\, h{:}\, s) = \cdots \tag{1}$$

where s is the access stride, and ℓ and h are the lower and upper bounds, respectively (Table 1 summarizes the notation). Given an array statement with HPF-style data mappings[1], it is our aim to generate the address sequence for different processors.

Consider the case of an array aligned identically to a template that is distributed `CYCLIC(4)` onto 3 processors, which is accessed with a stride of 5 ($p = 3$, $k = 4$, and $s = 5$). Figure 2(a) shows the allocation of the array elements along with the corresponding global addresses. The array elements accessed are marked and the corresponding local addresses are shown in Figure 2(a). While the global access stride is constant (5 in this case), the local access sequence does not have a constant stride on any processor. For example, the local addresses of elements accessed in processor 1 are $1, 10, 12, 19, 21, \cdots$. The address generation problem is to enumerate this sequence; the communication generation problem is to derive the required communication, given the access sequence. Figure 2(b) shows the global addresses of array A using the two-dimensional view that we adopt. Each element that is accessed is written as a two-component vector. In the global view, the first component of a vector that denotes an array element is the course number to which that array element belongs; the second component gives the offset of the element from the start of the course. In the local view, the first component of a vector that denotes an array element is the course number to which that array element belongs; the second component gives the offset of the element from the start of the course on the processor to which the element belongs. Figure 2(c) shows the two-dimensional view using local addresses.

2.1 Related work

Several papers have addressed this code generation problem. Koelbel [12] derived techniques for

[1]We present our approach for an alignment stride, $a = 1$. For $a > 1$, we use an approach similar to the one developed by Chatterjee *et al.* [4]. This involves two applications of our efficient code generation algorithm; thus, we do not discuss non-unit alignment strides.

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
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 1(a): Mapping of an array identically aligned to a template T which is CYCLIC(k) distributed among p processors.

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

Figure 1(b): Storage pattern of the array shown in Figure 1(a) in the local memory of Processor 0.

Figure 1: Global and local addresses for data mappings

compile-time address and communication generation for BLOCK and CYCLIC distribution for non-unit stride accesses containing a single loop index variable. Chatterjee *et al.* [4] derived a purely runtime 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 addresses accessed, followed by sorting these addresses to derive the accesses in linear order. Gupta *et al.* [8] 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 [4]. 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 [18, 19] presented a framework (that bears similarities to the approach of Gupta *et al.* [8]) to enumerate local addresses and generation of communication sets. 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 [14] 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.* [8]. van Reeuwijk *et al.* presented a technique which requires the solution of linear Diophantine equations. Benkner [2] presented a solution for code generation for block-cyclic distributions in Vienna Fortran 90. Recently, Kennedy, Nedeljkovic, and Sethi [11] derived an $O(k)$ algorithm for address

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

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

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

(b): two-dimensional view of global addresses

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

(c): two-dimensional view of local addresses

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

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. Our earlier research [23] has shown how to improve upon the algorithm of Kennedy, Nedeljkovic, and Sethi by overcoming the need to solve the additional $k-1$ linear Diophantine equations; in addition, we presented an efficient loop-nest based approach to enumerate the array addresses in order to derive the address pattern [23]. A special case for deriving access patterns similar to that discussed in Section 4.2 was presented by Hiranandini *et al.* [10]; their technique does not generalize to as many cases as our technique since it is not based on lattice theory. Wijshoff [26] 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 either code generation.

In contrast to these approaches for the address generation problem (except [11] and [23]), our solution is based on the theory of integer lattices. Array accesses are viewed as points of an integer lattice; an integer lattice is completely described by a set of *basis vectors* and a point that belongs to the lattice. Unlike the work of Kennedy, Nedeljkovic, and Sethi [11] and our earlier work [23]—which generate the basis vectors at runtime and then derive the access pattern using the basis vectors—we present closed form expressions for basis vectors. Thus, the only overhead incurred at runtime is in deriving the address pattern. Using the closed form expressions for the basis vectors, we derive 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 p (the number of processors) in powers of 2 from 2 to 1024, k (block size) in powers of 2 from 1 to 1024, and s (the access stride) ranging from 1 to 99 in unit increments, we derived the best pair of basis vectors using the closed form expressions for 82% of the problem instances. When solutions to known special cases were included, this number increases to 91%. The best pair of basis vectors result in the lowest address pattern generation time. In the remaining cases, our solution uses the best compile-time determined basis vectors. Our solution in all these cases outperforms existing solutions to this problem.

3 Basis Vectors and Access Pattern Derivation

3.1 Basis Vectors

We assume that A is identically aligned to the template T. As it is evident from Figure 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 number of the course 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}; such a notation is used throughout this paper. Similarly, A_{loc} refers to the locally allocated piece of array A on a processor.

Given an HPF-style data mapping, the elements that belong to an array section form a lattice [23]. We refer to the set of elements (addresses) over all processors accessed by $A(\ell\colon h\colon s)$ with distribution parameters (p, k) as the *array section lattice,* $\Gamma = \langle A(\ell\colon h\colon s), p, k\rangle$. The address sequence determination problem involves

- deriving a set of *basis vectors* for the lattice Γ; and
- using the basis vectors, enumerating the points that belong to the lattice Γ.

In particular, we derive *extremal basis vectors,* i.e., those basis vectors using non-negative linear combinations of which every point in the lattice can be generated. Using the extremal basis vectors, we derive a loop transformation that enumerates the address sequence. Next, we show that the vectors $(1, -(pk \bmod s))$ and $(0, s)$ form an extremal basis set for the array section lattice, using a two-dimensional view of global addresses. Informally, since the access stride is s, two consecutive points that belong to the same course differ by the vector $(0, s)$. If s divides pk, then the access pattern for the first course is the same as that for each of the remaining courses. Thus, in this case, it is sufficient to generate only the access pattern for the first course. If s does not divide pk, assuming that the element 0 is accessed, the last element accessed on course number zero is $(0, pk - pk \bmod s)$, and the first element accessed on course number one is $(1, s - pk \bmod s)$. The largest element on course one that is smaller than $(1, s - pk \bmod s)$ is $(1, -pk \bmod s)$.

Theorem 1 *The vectors $(0, s)$ and $(1, -(pk \bmod s))$ form a pair of basis vectors for the array section lattice.*

Proof: Without loss of generality, let us assume that the first array element accessed lies on the origin of a two-dimensional space. To show that $(0, s)$ and $(1, -(pk \bmod s))$ form a pair of basis vectors all we need to show is that, no other point belonging to the integer lattice lies in the triangle formed by the points $(0, 0)$, $(0, s)$ and $(1, -(pk \bmod s))$ [7] (Theorem 4, Page 20).

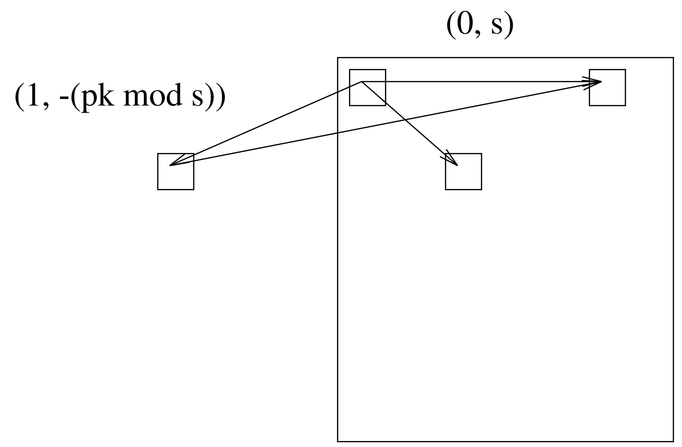

Figure 3: The vectors $(0, s)$ and $(1, -pk \bmod s)$ are extremal basis vectors.

Let us assume that (x, y) belongs to the integer lattice and also lies in the triangle formed by the above points. Hence the constraints on x and y are, $0 \leq x \leq 1$ and $-(pk \bmod s) \leq y \leq s$. That is the point (x, y) either lies on course 0 or course 1. Since the access stride is s no point belonging to the lattice falls between $(0, 0)$ and $(0, s)$. From Figure 3 it is obvious that the only points on course 1 that can satisfy the above constraints are $(1, -(pk \bmod s))$ and $(1, s - (pk \bmod s))$. $(1, s - (pk \bmod s))$ can lie in the triangle formed by the above three points only if the area of the triangle is zero. This is impossible since the vectors $(0, s)$ and $(1, -(pk \bmod s))$ are linearly independent. Hence no point belonging to the integer lattice lies in the triangle. Therefore, $(0, s)$ and $(1, -(pk \bmod s))$ form a pair of basis vectors. $\square$

Theorem 2 $(0, s)$ *and* $(1, -(pk \bmod s))$ *form a pair of extremal basis vectors.*

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

A *basis matrix,* B is a collection of basis vectors written as columns. Consider the example in Figure 2 where $p = 3$, $k = 4$, and $s = 5$; here $pk \bmod s = 12 \bmod 5 = 2$. The basis vectors are $(0, 5)$ and $(1, -2)$. Figure 4 shows that the lattice generated by these basis vectors is the set of array elements accessed. The elements accessed are the intersections of the two families of lines along the vectors $(0, 5)$ and $(1, -2)$.

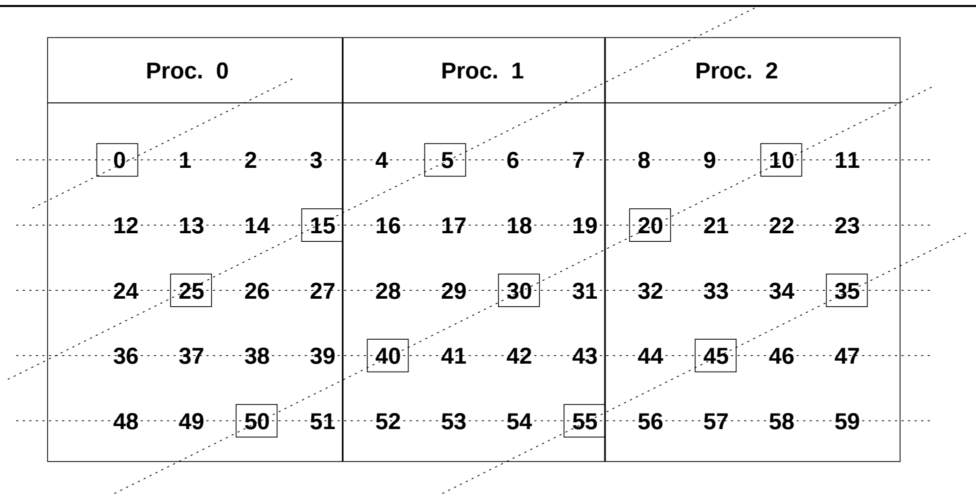

Figure 4: The vectors $(1, -2)$ and $(0, 5)$ are extremal basis vectors.

Basis vectors (and extremal basis vectors) are not necessarily unique. We use the following theorem, a proof of which can be found in Gruber and Lekkerkerker [7] (Theorem 7, Page 22).

Theorem 3 *Let Γ be a lattice with basis matrix B_1. Then a matrix B_2 is a basis of Γ if and only if there exists a unimodular matrix U such that $B_2 = B_1U$.*

An equivalent statement of this theorem is that if the integer matrix B_1 is a basis matrix for a lattice Γ, then any integer matrix B such that $det(B) = \pm det(B_1)$ is also a basis matrix of Γ. This result is used extensively in Section 4 to derive efficient basis vectors in several cases. Figure 4 shows the lattice generated by the basis matrix

$$B_1 = \begin{bmatrix} 1 & 0 \\ -2 & 5 \end{bmatrix}$$

Consider the matrix

$$B_2 = \begin{bmatrix} 1 & 1 \\ -2 & 3 \end{bmatrix}$$

Since

$$B_2 = \begin{bmatrix} 1 & 1 \\ -2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -2 & 5 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = B_1U$$

and U is unimodular, B_2 is also a basis matrix for the lattice shown in Figure 4. The lattice generated by the vectors $(1, -2)$ and $(1, 3)$ is shown in Figure 5.

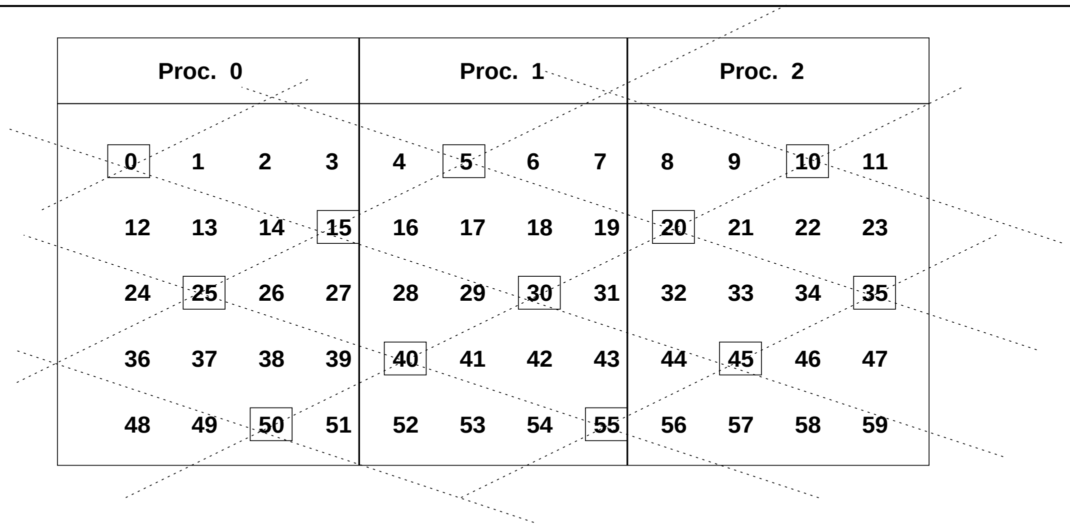

Figure 5: The vectors $(1, -2)$ and $(1, 3)$ are extremal basis vectors.

3.2 Access pattern generation for an array section with $l = 0$

The sequence of the array elements accessed (course by course) in a processor can be obtained by strip mining the array assignment statement in (1) with a strip length of pk and appropriately restricting the inner loop limits. A *basis matrix,* B is a collection of basis vectors written as columns. The basis matrix for the array section lattice is:

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

The determinant of B is the access stride s itself.

The access pattern for the array A repeats after every $\frac{s}{\gcd(s,pk)}$ courses of elements [4]. Thus, it is sufficient to store the access pattern of the elements in the first $\frac{s}{\gcd(s,pk)}$ courses and use this information to determine the remaining elements accessed. In this section, we assume $l = 0$ in (1); address pattern determination for array sections with arbitrary limits is discussed in Section 3.3. The code that iterates over the first $\frac{s}{\gcd(s,pk)}$ courses in the two-dimensional lattice shown in Figure 1(a) could be written as follows:

```
DO i = 0, s/gcd(s,pk) - 1
    DO j = 0, pk - 1
        A_2D[i, j] = ......
    ENDDO
ENDDO
```

As the access stride in (1) is s, only $\frac{1}{s}$ of the allocated space is spanned. In order to access only $\frac{1}{s}$ of the allocated space we need to apply a loop transformation to the above loop, the determinant of whose transformation matrix should be $\frac{1}{s}$ [15, 16, 27]. An obvious choice for a transformation matrix is the inverse of the basis matrix B. Also, in this case the first element allocated in a processor memory is mk, where m is the processor number. In order to obtain the address sequence for each of the processors, we need to apply the loop transformation to the following modified code.

```
DO i = 0, s/gcd(s,pk) - 1
    DO j = mk, k - 1 + mk
        A_2D[i, j] = ···
    ENDDO
ENDDO
```

The transformation matrix is

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

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

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

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

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

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

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

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

where $(u,v)^T = B^{-1}(i,j)^T$ and u and v are integers. Hence, $(i,j) = (u, vs - u(pk \bmod s))$. Using Fourier-Motzkin elimination [5, 27], we solve for u and v as follows:

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

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

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

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

```
DO u = 0, s/gcd(s,pk) - 1
     DO v = ⌈(u(pk mod s)+mk)/s⌉, ⌊(k-1+u(pk mod s)+mk)/s⌋
          A_2D[u, vs - u(pk mod s)] = ······
     ENDDO
ENDDO
```

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

```
α = pk mod s
β = mk
DO  u = 0, s/gcd(s,pk) - 1
     t = αu + β
     DO  v = ⌈t/s⌉, ⌊(k-1+t)/s⌋
```

```
        A_loc[uk + vs - t] = ······
    ENDDO
ENDDO
```

Here, β is the global address of the first element allocated to a processor. Our implementation of this code is highly optimized to take advantage of invariant code motion, common subexpression elimination, strength reduction, etc.

Since the access pattern for the array A repeats after every $\frac{s}{\gcd(s,pk)}$ courses of elements, we can use the above code to derive the access pattern. The optimized code that computes the access pattern is shown in Figure 6. Here we show an example that illustrates our pattern generation technique.

Example 1 Pattern determination for the case where $p = 4$, $k = 4$, and $s = 3$.

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

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

$i = 0$

DO $u = 0, \frac{s}{\gcd(s,pk)} - 1$

  $t = u + 4$

  DO $v = \lceil \frac{t}{3} \rceil, \lfloor \frac{3+t}{3} \rfloor$

    $pattern_1[i] = 4u + 3v - t$

    $i = i + 1$

  ENDDO

ENDDO

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

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

1 $\alpha = pk \bmod s$
2 $\beta = mk$
3 $s' = \frac{s}{\gcd(s,pk)}$
4 $i = 0$

5 `/* Compute and store the access pattern in array` $pattern_m$ `*/`
6 `DO` $u = 0, s' - 1$
7     $t = \alpha u + \beta$
8   `DO` $v = \left\lceil \frac{t}{s} \right\rceil, \left\lfloor \frac{k-1+t}{s} \right\rfloor$
9         $pattern_m[i] = uk + vs - t$
10         $i = i + 1$
11   `ENDDO`
12 `ENDDO`
13 $\text{length} = i;$

14 `/* Use the access pattern stored in array` $pattern_m$ `and execute code */`
15 `/*` h `is the upper bound of the array section */`
16 `/* Execute the code for all the full number of patterns */`
17 $num_b = \frac{\left\lfloor \frac{h}{pk} \right\rfloor}{s'}$
18 `DO` $\text{base} = 0, (num_b - 1)s'k, s'k$
19   `DO` $u = 0, \text{length} - 1$
20         $A_{loc}[\text{base} + pattern_m(u)]$
21   `ENDDO`
22 `ENDDO`

23 `/* Execute the code for last partial pattern if any */`
24 $\text{owner}_h = \left\lfloor \frac{h}{k} \right\rfloor \bmod p$
25 **if** $(m == \text{owner}_h)$
26     $\text{last}_m = k \left\lfloor \frac{h}{pk} \right\rfloor + h \bmod k$
27 **elseif** $(m < \text{owner}_h)$
28     $\text{last}_m = k \left\lfloor \frac{h}{pk} \right\rfloor + k - 1$
29 **else**
30     $\text{last}_m = k \left\lfloor \frac{h}{pk} \right\rfloor - 1$
31 **endif**
32 $\text{base} = num_b\, s'\, k$
33 $\text{add} = \text{base}$
34 $u = 0$
35 `WHILE` $(\text{add} \le \text{last}_m)$
36     $\text{add} = \text{base} + pattern_m[u]$
37     $A_{loc}[\text{add}]$
38     $u = u + 1$
39 `ENDWHILE`

Figure 6: Optimized code for $A(0 : h : s)$ that shows the computation of the local access pattern (lines 1–13) and execution (lines 14–39) for processor m

```
$\alpha = pk \bmod s$
$\beta = mk - l$
$u = \text{course}_l = \left\lfloor \frac{l}{pk} \right\rfloor$
$i = 0$;
**if** $(l \bmod (pk)) > (mk + k - 1)$
    $\text{course}_l = \text{course}_l + 1$
**endif**
**if** $\left\lfloor \frac{l}{k} \right\rfloor \bmod p = m$
    DO $v = (l \bmod k), k - 1, s$
        $pattern_m[i] = uk + v$
        $i = i + 1$
    ENDDO
    DO $u = \text{course}_l + 1, \text{course}_l + \frac{s}{\gcd(s,pk)} - 1$
        $t = \alpha u + \beta$
        DO $v = \left\lceil \frac{t}{s} \right\rceil, \left\lfloor \frac{k-1+t}{s} \right\rfloor$
            $pattern_m[i] = uk + vs - t$
            $i = i + 1$
        ENDDO
    ENDDO
    $u = \text{course}_l + \frac{s}{\gcd(s,pk)}$
    $t = \alpha u + \beta$
    DO $v = s \left\lceil \frac{t}{s} \right\rceil - t, (l \bmod k) - 1, s$
        $pattern_m[i] = uk + v$
        $i = i + 1$
    ENDDO
**else**
    DO $u = \text{course}_l, \text{course}_l + \frac{s}{\gcd(s,pk)} - 1$
        $t = \alpha u + \beta$
        DO $v = \left\lceil \frac{t}{s} \right\rceil, \left\lfloor \frac{k-1+t}{s} \right\rfloor$
            $pattern_m[i] = uk + vs - t$
            $i = i + 1$
        ENDDO
    ENDDO
**endif**
```

Figure 7: Code for pattern determination for an array section with lower bound l for processor m

3.3 Pattern generation for the array section $(\ell : h : s)$

For the array section given in equation (1), the origin of the two-dimensional lattice is no longer the first element allocated to processor 0. We need to shift the origin in the loop shown before to the element with global address l. The global address of the first element allocated to a processor after taking the shift of origin into account is $\beta = mk - l$.

The code for access pattern determination for the array assignment statement shown in (1) consists of a loop nest. The outer loop iterates over every course of size k of array A assigned to that processor. Since the first non-empty course of the lattice is the $\left\lfloor \frac{l}{pk} \right\rfloor$th course, we need to modify the lower bound of the outer loop to $\left\lfloor \frac{l}{pk} \right\rfloor$; in addition to this we need to avoid accessing elements to the left of l on course $\left\lfloor \frac{l}{pk} \right\rfloor$. If a processor does not access any element on course $\left\lfloor \frac{l}{pk} \right\rfloor$, then we increment the outer loop bound by one. This is done in lines 5–7 of Figure 7. The processor that owns l executes lines 9–25 of Figure 7 and all other processors execute lines 27–33. Lines 27–33 are similar to those in Figure 6. The processor that owns l accesses l and all elements to the right of l on course number $\left\lfloor \frac{l}{pk} \right\rfloor$ in lines 9–12. Lines 13–19 access a set of full courses. Lines 20–25 access all elements in the last course up to but excluding the element $l + pk\frac{s}{\gcd(s,pk)}$.

3.4 Pattern generation for alignment offset b

For the array section given in equation (1), where the array A is aligned with template cell $T(a * i + b)$ where $a = 1$ (and b is any integer), the origin of the two-dimensional lattice is no longer the first array element allocated to processor 0. We need to shift the origin in the loop shown above to the element with global address $l + b$. So the global address of the first element allocated to a processor after taking the shift of origin into account is, $\beta = mk - l - b$. This modification needs to be done in the code shown in Figure 7.

4 Basis Vectors and Pattern Determination for $s > k$

It is evident from Figure 2 that in the case where $s > k$, at most one element is accessed per course in a given processor. Therefore, the vector $(0, s)$ is a poor choice for one of the basis vectors. Since $s > k$ in this case, given that a certain element is accessed the next element accessed would definitely not belong on the same course. Thus, the best choice for one of the basis vectors would be the vector that spans the next element accessed form the origin. So with this new basis vector and the vector $(1, -(pk \bmod s))$ we will scan the accessed elements of the array A along a pair of slanted parallel lines each of which have at least one accessed element. Unlike the previous case, with this new set of basis vectors we would end up skipping all the empty courses. Next, we derive closed form expressions for the best pair of basis vectors as a function of the problem parameters s, p, and k, and present the optimized code for each case.

4.1 Case where $0 < s - (pk \bmod s) < k$

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

Theorem 4 *The vectors* $(1, -(pk \bmod s))$ *and* $(1, s - (pk \bmod s))$ *form a pair of extremal basis vectors when* $0 < s - (pk \bmod s) < k$.

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

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

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

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

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

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

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

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

The code for access pattern determination for processor m derived using the transformation, B^{-1}, is:

$i = 0$;
$\mathcal{C} = \frac{s}{gcd(s,pk)} - 1$;
$\alpha = pk \bmod s$;
DO $u = \left\lceil \frac{-mk-k+1}{s} \right\rceil, \left\lfloor \frac{\mathcal{C}(s-\alpha)-mk}{s} \right\rfloor$
  DO $v = \left\lceil \max\left(\frac{mk+\alpha u}{s-\alpha}, -u\right) \right\rceil, \left\lfloor \min\left(\frac{mk+k-1+\alpha u}{s-\alpha}, \mathcal{C}-u\right) \right\rfloor$
    $pattern_m[i] = u(k-\alpha) + v(k+s-\alpha) - mk$
    $i = i + 1$
  ENDDO
ENDDO

As before, (for example, in Figures 6 and 7), the above code in our implementation is optimized.

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

Example 2 Code generation for the case where $p = 3$, $k = 4$ and $s = 5$.

Since in this case $(s-(pk \bmod s)) < k$, we use the suggested basis vectors. Memory allocation for elements of array A along with the global and local addresses of the accessed elements, for the case $p = 3$, $k = 4$, and $s = 5$ is shown in Figure 2. The pattern determination code for processor 2 is shown below:

$i = 0$;
DO $u = -2, 0$
  DO $v = \left\lceil \max\left(\frac{2u+8}{3}, -u\right) \right\rceil, \left\lfloor \min\left(\frac{2u+11}{3}, 4-u\right) \right\rfloor$
    $pattern_2[i] = 2u + 7v - 8$
    $i = i + 1$
  ENDDO
ENDDO

Here again we run through the first couple of iterations of the above loop and show that the code does indeed access the right elements in the right order. The outer loop has 3 iterations and a total of 4 points are accessed.

u	$v_{lb} = \left\lceil \max\left(\frac{2u+8}{3}, -u\right)\right\rceil$	$v_{ub} = \left\lfloor \min\left(\frac{2u+11}{3}, 4-u\right)\right\rfloor$	local index $= 2u + 7v - 8$
-2	2	2	2
-1	2	3	4, 11
0	3	3	13

4.2 Case where $0 < (pk \bmod s) < k$

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

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

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

$i = 0$;
$\mathcal{C} = \frac{s}{gcd(s,pk)} - 1$;
$\alpha = pk \bmod s$;
`DO` $u = \left\lceil \frac{mk}{s} \right\rceil, \left\lfloor \frac{\mathcal{C}\alpha+mk+k-1}{s} \right\rfloor$
    `DO` $v = \left\lceil \max\left(\frac{(s-\alpha)u-mk-k+1}{\alpha}, -u\right)\right\rceil, \left\lfloor \min\left(\frac{(s-\alpha)u-mk}{\alpha}, \mathcal{C} - u\right)\right\rfloor$
        $pattern_m[i] = u(k + s - \alpha) + v(k - \alpha) - mk$
        $i = i + 1$
    `ENDDO`
`ENDDO`

If both $(pk \bmod s)$ and $(s - (pk \bmod s))$ lie between 0 and k, it would help to choose the basis matrix corresponding to the smaller of the two. With this choice of a basis matrix, we end up with fewer number of outer loop iterations in the transformed loop nest. The resulting code will be more efficient as we end up making fewer upper and lower bound calculations for the inner loop in the transformed loop nest. To illustrate this, consider Example 2 again where, $p = 3$, $k = 4$, and $s = 5$. Since in this case $(pk \bmod s) < k$, we use the suggested basis vectors. Memory allocation for elements of array A along with the global and local addresses of the accessed elements for this case are shown in Figure 2. The pattern determination code for processor 2 is shown below:

$i = 0$;

```
DO u = 2,3
    DO v = ⌈max((3u−11)/2, −u)⌉, ⌊min((3u−8)/2, 4 − u)⌋
        pattern_2[i] = 7u + 2v − 8
        i = i + 1
    ENDDO
ENDDO
```

The table below shows the elements accessed. The outer loop now has 2 iterations and 4 elements are accessed. While the set of accessed elements is exactly the same as before, the time taken to generate the elements is lower.

u	$v_{lb} = \left\lceil \max\left(\frac{3u-11}{2}, -u\right)\right\rceil$	$v_{ub} = \left\lfloor \min\left(\frac{3u-8}{2}, 4-u\right)\right\rfloor$	local index $= 7u + 2v - 8$
2	-2	-1	2, 4
3	-1	0	11, 13

4.3 Case where $0 < s - q(pk \bmod s) < k$ for some integer q

If $pk \bmod s \geq k$ and $s - (pk \bmod s) \geq k$, an obvious extension to the previous cases is to check if there exists an integer q such that $0 < s - q(pk \bmod s) < k$, *i.e.,* $1 \leq s - q(pk \bmod s) \leq k - 1$. All the possible integer values for q are given by the following inequality:

$$\left\lceil \frac{s-k+1}{pk \bmod s} \right\rceil \leq q \leq \left\lfloor \frac{s-1}{pk \bmod s} \right\rfloor .$$

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

Theorem 5 *The vectors* $(1, -(pk \bmod s))$ *and* $(q, s - q(pk \bmod s))$ *form a pair of extremal basis vectors when* $1 \leq s - q(pk \bmod s) \leq k - 1$ *where* $q = \left\lceil \frac{s-k+1}{pk \bmod s} \right\rceil$ *and* $q \leq \lfloor \frac{s-1}{pk \bmod s} \rfloor$.

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

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

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

belongs to the integer lattice then,

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

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

The basis matrix for this case is

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

The pattern generation code for processor m is shown below.

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

4.4 Case where $0 < s - q(pk \bmod s) < k$ for no integer q

If we can not find a q such that $1 \le s - q(pk \bmod s) \le k - 1$, then we approximate the basis by using the basis vectors $(1, -pk \bmod s)$ and $(q, s - q(pk \bmod s))$ where $q = \left\lfloor \frac{s-1}{pk \bmod s} \right\rfloor$. In this case, we do not have the best pair of basis vectors. Nevertheless, the code we generate is correct and these vectors are extremal basis vectors.

4.5 Case where $s > pk$

When $s > pk$, we apply optimizations to the following basis matrix (similar to those presented in the previous subsections):

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

From Theorem 1 and from the fact that for $s > pk$, $pk \bmod s = pk$, the matrix

$$\begin{bmatrix} 1 & 0 \\ -pk & s \end{bmatrix}$$

is a basis matrix. Using Theorem 3, the basis matrix B can be written as:

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

and the matrix $\begin{bmatrix} s \operatorname{div} pk & 1 + (s \operatorname{div} pk) \\ 1 & 1 \end{bmatrix}$ is unimodular. A detailed proof that the columns of matrix B are extremal basis vectors for $s > pk$ can be found in our earlier paper [21, 22].

The cases for which we derive the best basis vectors when $s > pk$ [21, 22] are:

- $1 \leq s \bmod pk \leq k - 1$,
- $1 \leq pk - (s \bmod pk) \leq k - 1$,
- for some integer q, $1 \leq pk - q(s \bmod pk) \leq k - 1$.

4.6 Discussion

In an experiment [21, 22] with p (the number of processors) in powers of 2 from 2 to 1024, k (block size) in powers of 2 from 1 to 1024, and s (the access stride) ranging from 1 to 99 in unit increments, we derived the optimal compile time solution for over 91% of the problem instances. This includes known special cases and closed forms derived in this paper. Figure 8 shows the percentage of cases for which we derive the best basis vectors as a function of the number of processors and the range of variation of the stride. For the case where the stride was varied from 1 to 399, an average of 84% of the cases were covered by our technique.

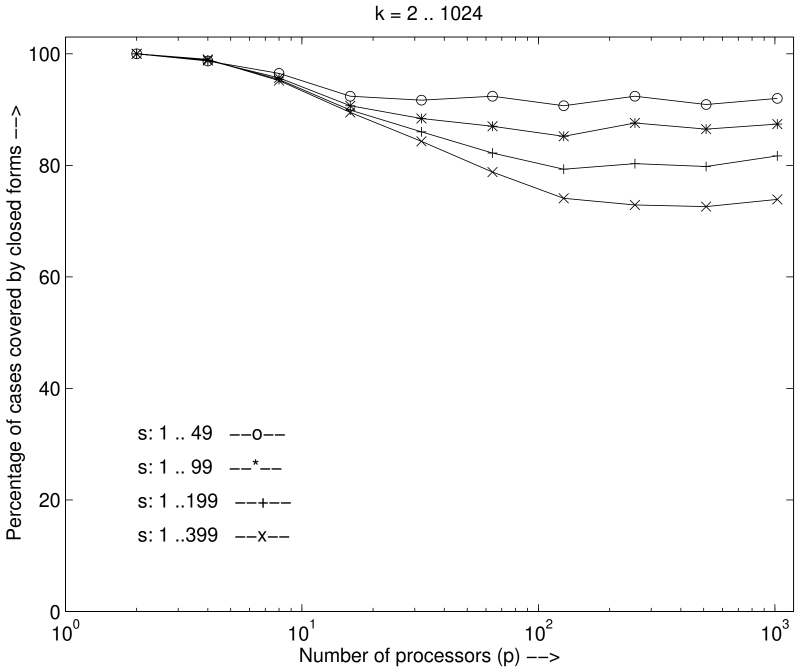

Figure 8: Percentage of cases covered by closed forms.

5 Experimental Results

In the experiments described below, we used the `cc` compiler with `-O3` level optimizations turned on; the function `gettimeofday()` was used to measure time. In all the experiments, we did not use the codes for special cases [4, 13] since we wanted to measure the performance in the general case. All experiments were done on an IBM SP-2. When computing time for p processors, the code that computes the access pattern for each processor was timed, and the maximum time over all processors is reported. The effect of varying the number of processors p was insignificant; therefore, we used 32 processors in our experiments. In addition, varying the lower bound of the array section had little impact on the results; so, we set l to zero.

We performed two sets of experiments on our algorithm. Figures 9 and 10 illustrate experimental results for 32 processors with strides in the range 2 to 13, and block sizes ranging from 1 to 64. These figures compare the performance of our algorithm (referred to as "CF") to the algorithm of Kennedy, Nedeljkovic, and Sethi [11] (referred to as "Rice" in the figures). Clearly, our approach outperforms the Rice approach for this set of experiments. The overhead of computing the access pattern is very small, compared to the message start-up costs typically encountered in current distributed memory message passing machines (such as the IBM SP-2). In all figures, whenever one of the special cases [4, 13] is encountered, the pattern generation time reduces significantly. For example, whenever k is a multiple of s, the address pattern generation time drops significantly. In addition, for a power of 2 stride, the address pattern generation time in CF is linear in the block size with a slope near zero. This observation has independently been made by MacDonald, Pase, and Meltzer [13].

Next we compared our implementation that computes the access pattern to three published runtime algorithms: loop enumeration (called "Loop" in the tables) [23], the algorithm of Kennedy, Nedeljkovic, and Sethi [11][2] (called "Rice" in the tables), and the implementation from Chatterjee *et al.* [4] (called "Sort" in the tables). The results are shown in Tables 2–4. We experimented with block sizes in powers of 2 ranging from 4 to 1024 for 32 processors; in each case we used strides of 7, 99, $k-1$, $k+1$, $pk-1$ and $pk+1$. The algorithm presented in this paper is referred to as "CF" in the tables. From the data in Tables 2–4, it is clear that our approach outperforms the three approaches. With increasing values of block sizes, the difference in performance is very high. We derive the best basis vectors for all cases except $s = 99$ with $k = 4$ to $k = 32$. In these cases, the CF approach takes more time to derive the access pattern (see Table 2). For large block sizes, the closed-form (CF) approach is 2 to 7 times faster than Loop, and 3 to 10 times faster than Rice; compared to Sort, CF is 30 to 84 times faster.

6 Conclusions

The success of data parallel languages such as High Performance Fortran critically depends on efficient compiler and runtime support. In this paper we presented an efficient algorithm for generating local memory access patterns for array statement execution in languages like High Performance Fortran, 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 uses closed form expressions for the basis vectors of the lattice. This allows us to determine the access pattern efficiently at runtime; this avoids the runtime overhead for basis vector generation incurred by the most efficient of the runtime approaches [11, 23]. Experiments on several platforms demonstrate the efficacy of our approach. Our approach outperforms the approaches of Chatterjee *et al.* [4], Kennedy, Nedeljkovic, and Sethi [11] and our earlier work [23]; the difference is more pronounced at large values of block size. We have used our technique in generating communication sets for data parallel programs [20, 24, 25]. Work is in progress for efficient solutions to two-level mapping that arises with non-unit alignment strides. Techniques for

[2] There are two implementations of their algorithm. The numbers reported here are the best of the two in each case.

Table 2: Times (in microseconds) for closed form expressions (CF), Loop enumeration (LSU), Rice code, and the Sort approach on an IBM SP-2 ($p = 32, s = 7, s = 99$)

Block Size k	$s = 7$ CF	Loop	Rice	Sort	$s = 99$ CF	Loop	Rice	Sort
4	3	9	8	9	21	11	9	11
8	3	11	10	15	33	12	11	17
16	3	11	12	26	18	13	14	28
32	3	14	17	59	24	17	20	61
64	4	21	26	186	24	30	31	180
128	7	32	43	406	20	44	57	344
256	10	56	82	745	26	71	110	750
512	18	105	157	1401	34	119	181	1632
1024	34	200	314	2887	47	219	357	3284

Table 3: Times (in microseconds) for closed form expressions (CF), Loop enumeration (LSU), Rice code, and the Sort approach on an IBM SP-2 ($p = 32, s = k - 1, s = k + 1$)

Block Size k	$s = k - 1$ CF	Loop	Rice	Sort	$s = k + 1$ CF	Loop	Rice	Sort
4	2	9	8	9	2	10	8	10
8	3	11	10	15	3	11	9	14
16	4	11	12	27	10	12	12	27
32	5	14	15	61	9	15	18	61
64	16	31	32	186	23	30	30	195
128	20	42	51	377	28	42	49	406
256	31	70	87	751	39	69	86	741
512	51	123	172	1418	62	123	166	1528
1024	93	235	315	4215	107	232	309	3222

Table 4: Times (in microseconds) for closed form expressions (CF), Loop enumeration (LSU), Rice code, and the Sort approach on an IBM SP-2 ($p = 32, s = pk - 1, s = pk + 1$)

Block Size k	$s = pk - 1$ CF	Loop	Rice	Sort	$s = pk + 1$ CF	Loop	Rice	Sort
4	8	8	7	9	6	7	7	8
8	8	9	8	14	6	8	8	13
16	8	10	11	26	6	9	10	25
32	9	13	15	61	7	11	15	57
64	10	19	24	178	7	16	24	179
128	11	30	43	367	10	26	42	342
256	15	53	79	678	13	45	78	861
512	23	99	167	1680	21	83	157	1378
1024	38	191	316	3118	36	160	300	2957

incremental generation of access sequences and communication sets are being explored.

Acknowledgments We thank Ajay Sethi and Nenad Nedeljkovic for providing the Rice and Sort codes that we used in our experiments. We also thank Arun Venkatachar for his detailed comments on 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] S. Benkner. Handling block-cyclic distributed arrays in Vienna Fortran 90. In *Proc. International Conference on Parallel Architectures and Compilation Techniques,* Limassol, Cyprus, June 1995.

[3] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming,* 1(1):31–50, Fall 1992.

[4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, 1995.

[5] G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A),* 14:288–297, 1973.

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

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

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

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

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

[11] 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.

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

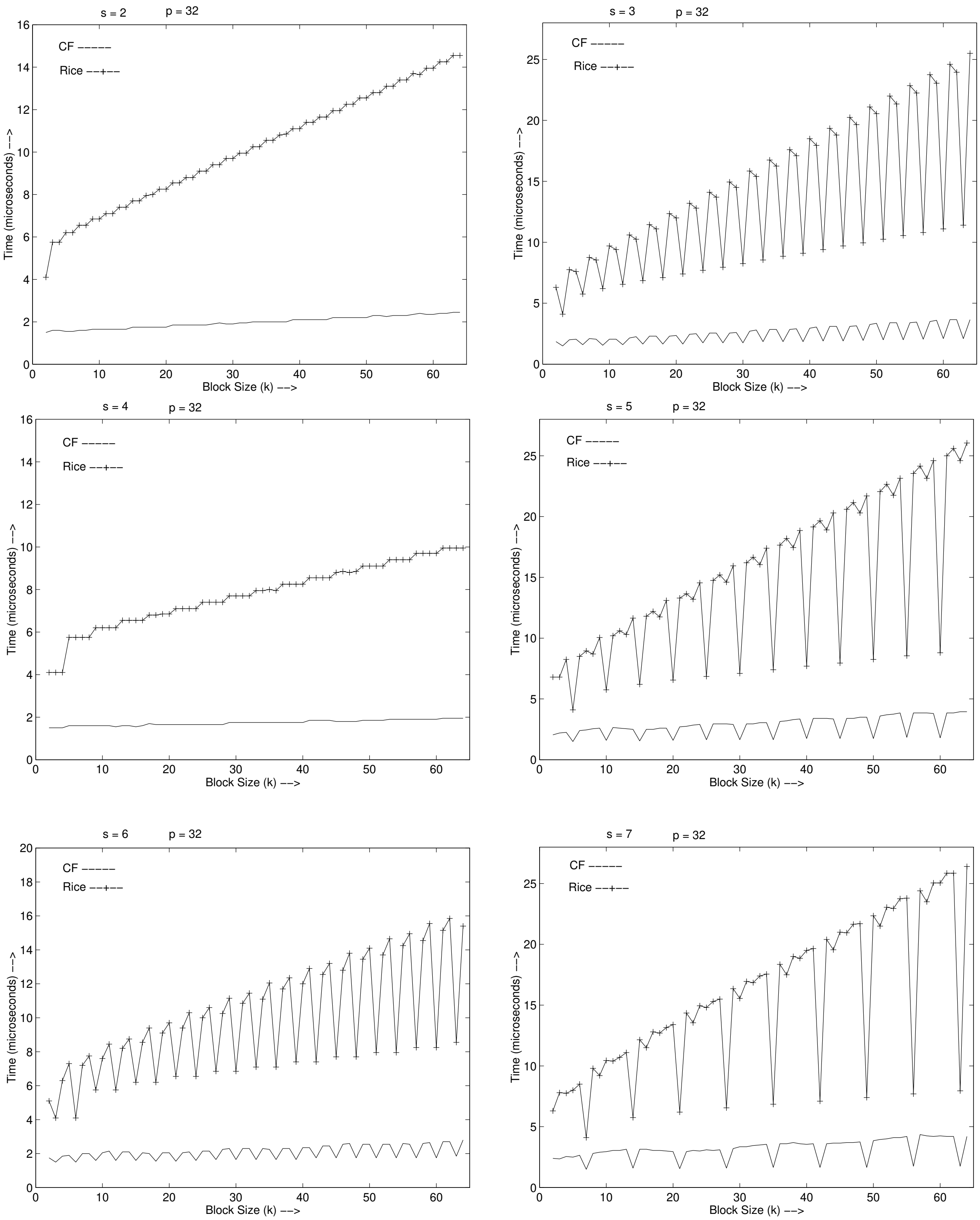

Figure 9: Effect of stride $(2 \cdots 7)$ and block size $(2 \cdots 64)$ for $p = 32$ processors

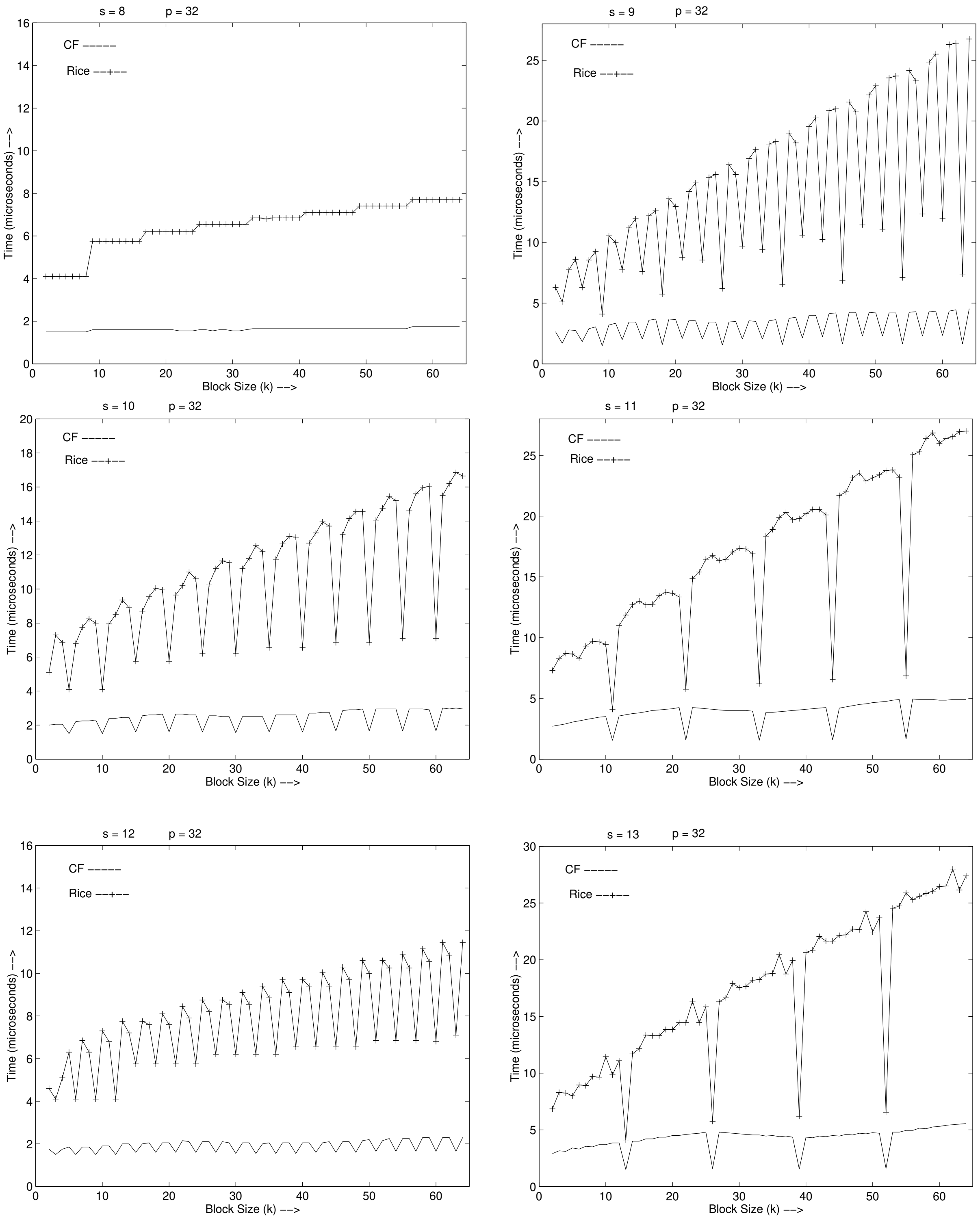

Figure 10: Effect of stride $(8 \cdots 13)$ and block size $(2 \cdots 64)$ for $p = 32$ processors

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

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

[15] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing 92,* pages 214–223, November 1992.

[16] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing,* 9(4):365–389, December 1995.

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

[18] J. M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[19] J. M. 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.

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

[21] A. Thirumalai and J. Ramanujam. Code generation and optimization for array statements in HPF. Technical Report, Louisiana State University, November 1994, revised August 1995.

[22] A. Thirumalai and J. Ramanujam. Address sequence determination using closed forms for basis vectors of the array section lattice. Technical Report, Louisiana State University, April 1995.

[23] 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.,* Eds., Lecture Notes in Computer Science, Vol. 1033, pages 191–208, Springer-Verlag, 1996.

[24] 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., Chapter 29, pages 311–316, Kluwer Academic Publishers, Norwell, MA, 1995.

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

[26] H. A. G. Wijshoff. *Data organization in parallel computers.* Kluwer Academic Publishers, Norwell, MA, 1989.

[27] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Publishing Co., Redwood City, CA, 1996.

Biographies

Biography of Ashwath Thirumalai

Ashwath Thirumalai received the Bachelor of Engineering degree in Electronics from the Bangalore University, India, in Spring 1992, and the M.S. degree in Electrical Engineering from the Louisiana State University in Summer 1995.

From 1993 to 1995 he was a Research Assistant in the Parallelizing Compilers group at the Louisiana State University. Since 1995 he has been with Zycad Corporation where he is a member of the Compilers group and is involved in compiler optimizations in the front end for Zycad's massively parallel platforms that deliver supercomputer-level performance for simulation of microprocessors, ASICs and system designs. His interests are in compiler construction, compiler optimizations and code generation for data parallel languages.

Biography of J. Ramanujam

J. Ramanujam received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Madras, in August 1983, and the M.S and Ph.D. degrees in Computer Science from The Ohio State University, Columbus, Ohio, in August 1987 and December 1990, respectively.

He is currently an Associate Professor of Electrical and Computer Engineering at Louisiana State University, Baton Rouge. His research interests are in compilers for high-performance computer systems, program transformations, parallel programming environments and operating systems, parallel input/output systems, parallel architectures and algorithms.

He received the National Science Foundation's Young Investigator Award in 1994. He has served on the Program Committee of the 8th International Conference on Supercomputing, 1995 and is a member of the High Performance Fortran Forum. He has taught tutorials on compilers for high-performance computers at several conferences such as the International Conference on Parallel Processing (1995, 1996), Supercomputing 94, Scalable High-Performance Computing Conference (SHPCC 94), and the International Symposium on Computer Architecture (1993 and 1994).