An Efficient Compile-Time Approach to Compute Address Sequences in Data Parallel Programs

Ashwath Thirumalai
J. Ramanujam*
Dept. of Electrical & Computer Engineering
Louisiana State University, Baton Rouge, LA 70803
Email: {ash,jxr}@gate.ee.lsu.edu

March 1995

Abstract

Arrays are mapped to processors through a two step process—alignment followed by distribution—in 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 a novel approach based on integer lattices. The set of elements referenced can be generated by integer linear combinations of basis vectors. Unlike other work on 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 pattern of repetition 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 better than that of other solutions to this problem.

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

*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.

1 Introduction

Distributed memory multiprocessors are attractive for high performance computing in that they offer potentially high levels of flexibility, scalability and performance. But, 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) [4], Fortran D [5], and Vienna Fortran (which grew out of the earlier SUPERB effort) [2] 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, a 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 statement $A(\ell\colon h\colon s)$ 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 [3, 9, 17] 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. Communication sets for array statements can be derived by two applications of our algorithm for address generation [15, 17, 18]. 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 [4]. 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 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

processor memories in a linear fashion as shown in Figure 1(b), we adopt a two-dimensional view of the storage allocated 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:

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

where s is the access stride, and ℓ and h are the lower and upper bounds, respectively. Given an array statement with HPF-style data mappings[1], it is our aim to generate SPMD code along with communication for the different processors.

Consider the case of an array aligned identically to a template that is distributed `CYCLIC(4)` onto 3 processors, which is accessed with a stride of 5 (p=3, k=4 and s=5). 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 2-dimensional view using

[1] We present our approach for an alignment stride, $a = 1$. For $a > 1$, we use an approach similar to the one developed in [3]. This involves two applications of our efficient code generation algorithm; thus, we do not discuss non-unit alignment strides.

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	$\boxed{1,32}$	1,4	1,5	1,6	1,7	$\boxed{1,8}$	1,9	1,10	1,11
2,0	$\boxed{2,1}$	2,2	2,3	2,4	2,5	$\boxed{2,6}$	2,7	2,8	2,9	2,10	$\boxed{2,11}$
3,0	3,1	3,2	3,3	$\boxed{3,4}$	3,5	3,6	3,7	3,8	$\boxed{3,9}$	3,10	3,11
4,0	4,1	$\boxed{4,2}$	4,3	4,4	4,5	4,6	$\boxed{4,7}$	4,8	4,9	4,10	4,11
$\boxed{5,0}$	5,1	5,2	5,3	5,4	$\boxed{5,5}$	5,6	5,7	5,8	5,9	$\boxed{5,10}$	5,11
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(b): 2-dimensional view of global addresses

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

(c): 2-dimensional view of local addresses

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

local addresses.

2.1 Related work

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 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 address accessed, followed by sorting these addresses to derive the accesses in linear order. In an exhaustive study of this problem, Stichnoth [14] presented a framework to enumerate local addresses and generation of communication sets. Gupta *et al.* [7] derived the *virtual-block* and the *virtual-cyclic* schemes. The virtual block (cyclic) scheme views the global array as a union of several cyclically (block) distributed arrays. The virtual cyclic scheme does not preserve the access order in the case of `DO` loops; this is not a problem for parallel array assignments. For large block sizes, this approach may suffer from cache misses [3]. Ancourt *et al.* [1] use a linear algebra framework to generate code for fully parallel loops in HPF, which does not exploit repeating sequence of accesses. Recently, building on the FSM approach of Chatterjee *et al.*, Kennedy *et al.* [9] derived a runtime algorithm that runs in $O(k)$ time. 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 [17], we have shown how to improve upon the algorithm of Kennedy *et al.* by overcoming the need to solve the additional $k-1$ linear Diophantine equations; in addition, we [17] presented an efficient loop-nest based approach to enumerate the array addresses in order to derive the address pattern.

In contrast to these approaches (except [9] and [17]), 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 in [9] and our earlier work [17]—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 which has a determinant which is the inverse of the access stride. In an experiment with p (the number of processors) in powers of 2 from 1 to 1024, k (block size) in powers of 2 from 1 to 16384, and s (the access stride) ranging from 1 to 99 in unit increments, we derived the best pair of basis vectors using the closed form expressions for 91% of the problem instances. 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 Access pattern derivation for array statements

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

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 has an address of the form (x, y) 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.

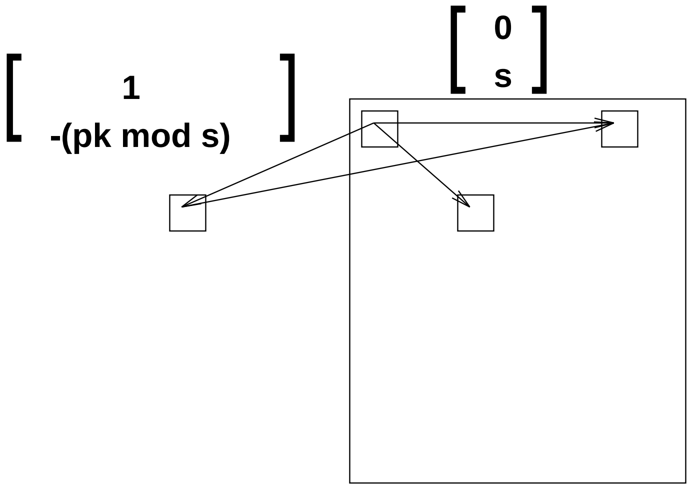

Figure 3: The vectors $(0, s)$ and $(1, -pk \bmod s)$ are extremal basis vectors.

Given a HPF-style data mapping, the elements that belong to an array section form a lattice [17]. The address sequence determination problem involves

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

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 $\begin{pmatrix} 1 \\ -(pk \bmod s) \end{pmatrix}$ and $\begin{pmatrix} 0 \\ s \end{pmatrix}$ form an extremal basis set for the array section lattice, using a 2-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 $(0, s)$ *and* $(1, -(pk \bmod s))$ *form a pair of basis vectors.*

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 $(1, -(pk \bmod s))$ [6] (Theorem 4, Page 20).

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. □

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. □

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. The *basis matrix,* B is a collection of basis vectors written as columns and is shown below:

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

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 [3]. 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 3.2. 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 $i = 0, \frac{s}{\gcd(s,pk)} - 1$
  do $j = 0, (pk - 1)$
    $A_{2D}(i, j) = \cdots\cdots$
  end do
end do

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 [13, 19] should be $\frac{1}{s}$. 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 $i = 0, \frac{s}{\gcd(s,pk)} - 1$
  do $j = mk, (k - 1) + mk$
    $A_{2D}[i, j] = \cdots$
  end do
end do

The transformation matrix is

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

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

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

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

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

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

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

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

where $\begin{bmatrix} u \\ v \end{bmatrix} = B^{-1} * \begin{bmatrix} i \\ j \end{bmatrix}$ and u and v are integers. So $\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} u \\ vs - u(pk \bmod s) \end{bmatrix}$.
Using Fourier-Motzkin elimination [19], 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 lead to:

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

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

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

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

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

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

Since the access pattern for the array A repeats after every $\frac{s}{\gcd(s,pk)}$ 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 4. 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 = $\lceil \frac{t}{3} \rceil, \left\lfloor \frac{3+t}{3} \right\rfloor$
    $pattern_1[i] = 4u + 3v - t$
    Increment i
  end do
end do

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

$\alpha = pk \bmod s$; $\beta = mk$; $k' = k - 1$
$i = 0$; $s' = \frac{s}{\gcd(s,pk)}$
`/* Compute and store the access pattern in array` $pattern_m$ `*/`
do $u = 0, s' - 1$
    $t = \alpha u + \beta$
    do $v = \lceil \frac{t}{s} \rceil , \lfloor \frac{k'+t}{s} \rfloor$
        $pattern_m[i] = uk + vs - t$
        Increment i
    end do
end do
length $= i + 1$;
`/* Use the access pattern stored in array` $pattern_m$ `and execute code */`
`/* N-1 is the upper bound of the array section */`
$N' = \frac{\lfloor \frac{N-1}{pk} \rfloor}{s'}$
$j = 0$
do $w = 0, N'$
    offset $= j * k$
    do $u = 0$, length $- 1$
        $A_{loc}[\text{offset} + pattern_m(u)]$
    end do
    $j = j + s'$
end do

Figure 4: Optimized code for A(0:N-1:s) computes local access pattern for processor m

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

3.2 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 nested loop. The outer loop iterates over every course of size k of array A assigned to that processor. Since the first non-empty course of the lattice is the $\left\lfloor \frac{l}{pk} \right\rfloor$th course, we need to modify the lower bound of the outer loop to $\left\lfloor \frac{l}{pk} \right\rfloor$; in addition to this we need avoid accessing elements to the right of l on course $\left\lfloor \frac{l}{pk} \right\rfloor$. This is done by generating two pattern tables for the processor that owns element l; the *preamble* holds the access pattern for the elements on that processor in course l and *pattern* stores the access pattern. For all other processors, a single access pattern is derived. When executing code, the processor that owns l first uses the *preamble* table, and then repeatedly uses the *pattern* table. The other processors use their *pattern* tables to execute code. The modified code for (1) is shown in Figure 5. In the rest of this paper for the sake of clarity we shall use loose upper and lower bounds, as boundary effects can be easily rectified as shown above.

3.3 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 5.

3.4 Discussion

The idea of detecting a pattern that repeats after the first $\frac{s}{\gcd(s,pk)}$ courses was introduced by Chatterjee *et al.* [3]. They use a runtime approach to the problem; the complexity of the algorithm that they present for detecting the pattern is $O(klogk)$. We use a compile-time approach; our algorithm has a complexity of $O\left(\frac{k}{\gcd(s,pk)}\right)$ for the case where $s < k$ as we need to run through the nested loop at most k times to find the pattern. This is the only runtime overhead we incur. We handle the case where $s \geq k$ by finding the best basis vectors that enumerate the accessed indices in the right order. This is described in the next section.

4 Closed form expressions for basis vectors and pattern determination when $s \geq k$

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 $\begin{pmatrix} 0 \\ s \end{pmatrix}$ is a poor choice for one of the basis vectors. Since $s \geq k$

```
α = pk mod s; k' = k − 1
β = mk − l; u = course_l = ⌊l/(pk)⌋
if (l mod (pk)) > (mk + k − 1)
    Increment course_l
endif
if ⌊l/k⌋ mod p = m
    i = 0;
    do v = (l mod k), k − 1, s
        preamble_m[i] = uk + v
        Increment i
    end do
    preamble_length = i + 1;
    Increment course_l
endif
i = 0;
do u = course_l, course_l + s/gcd(s,pk) − 1
    t = αu + β
    do v = ⌈t/s⌉, ⌊(k'+t)/s⌋
        pattern_m[i] = uk + vs − t
        Increment i
    end do
end do
```

Figure 5: Code for pattern determination for an array section with lower bound l for processor m

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 $\begin{pmatrix} 1 \\ -(pk \bmod s) \end{pmatrix}$ we will scan the accessed elements of the array A along a pair of slanted parallel lines each of which have at least one accessed element. So unlike the previous case, with this new set of basis vectors we would end up skipping all the empty courses. Next, we derive closed form expressions for the best pair of basis vectors as a function of the problem parameters s, p, and k, and present the optimized code for each case.

4.1 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 $\begin{pmatrix} 1 \\ s-(pk \bmod s) \end{pmatrix}$. We next prove that the above vector and $\begin{pmatrix} 1 \\ -(pk \bmod s) \end{pmatrix}$ form an extremal set of basis vectors.

Theorem 3 *$(1, -(pk \bmod s))$ and $(1, s - (pk \bmod s))$ form a pair of extremal basis vectors when $0 < s - (pk \bmod s) < k$.*

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

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

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

$$(x, y) = a * (1, -(pk \bmod s)) + b * (1, s - (pk \bmod s))$$

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 transformation matrix is of the form:

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

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

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

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

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

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

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

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

Now using Fourier-Motzkin elimination we solve for u and v as follows,

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

Hence,

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

Finally, the code for access pattern determination for processor m is:

$i = 0;$

do u = $\left\lceil \frac{-mk-k+1}{s} \right\rceil, \left\lfloor \frac{\mathcal{C}(s-(pk \bmod s))-mk}{s} \right\rfloor$

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

    $pattern_m[i] = u(k-(pk \bmod s)) + v(k+s-(pk \bmod s)) - mk$

    Increment i

  end do

end do

As before, (for example, in Figures 4 and 5), 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 1 is shown below:

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

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

u	$v_{lb} = \left\lceil \max\left(\frac{2u+4}{3}, -u\right) \right\rceil$	$v_{ub} = \left\lfloor \min\left(\frac{2u+7}{3}, \mathcal{C} - u\right) \right\rfloor$	local index $= 2u + 7v - 4$
-1	1	1	1
0	2	2	10
1	2	3	12, 19

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

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

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

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.

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

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

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

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

Theorem 4 $(1, -(pk \bmod s))$ *and* $(q, s - q(pk \bmod s))$ *form a pair of extremal basis vectors when* $0 < s - q(pk \bmod s) < k$ *where* $q = \lfloor \frac{s}{pk \bmod s} \rfloor$.

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,

$$(x, y) = a * (1, -(pk \bmod s)) + b * (q, s - q(pk \bmod s))$$

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 $0 < s - (pk \bmod s) < k$. □

The basis matrix for this case is

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

The transformation matrix is

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

As before the loop bounds can be written as follows:

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

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

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

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

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

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

Now using Fourier-Motzkin elimination we solve for u and v as follows,

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

Hence,

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

The pattern generation code for processor m is shown below.

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

4.4 Compile time generation of code for several special cases

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

4.5 Discussion

In an experiment [15] with p (the number of processors) in powers of 2 from 1 to 1024, k (block size) in powers of 2 from 1 to 16384, and s (the access stride) ranging from 1 to 99 in unit increments, we derived the optimal compile time solution for over 91% of the problem instances.

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

5 Experimental Results

We performed two sets of experiments on our algorithm. In the first set, we employed three machines namely, a SUN Sparcstation 2, a SUN Sparcstation 10, and an IBM RS-6000. We used the `cc` compiler with `-O3` level optimizations turned on; the function `gettimeofday()` was used to measure time. When computing time for 32 processors, we timed the code that computes the access pattern for each processor, and report the maximum time over all processors. Figures 6 and 7 illustrate experimental results for 32 processors with strides from 2 to 13 and block sizes ranging from 1 to 64, on these three machines. The overhead of computing the access pattern is very small, compared to the message start-up costs typically encountered in current distributed memory message passing machines (such as the IBM SP-2). In all figures, whenever one of the special cases listed in Section 4.4 is encountered, the pattern generation time reduces significantly. For example, whenever k is a multiple of s, the address pattern generation time drops significantly. In addition, for a power of 2 stride, the address pattern generation time is a linear in the block size with a slope near zero. This observation has independently been made by [12].

Next we compared our implementation that computes the access pattern to the implementation from Chatterjee *et al.* [3] referred to as RIACS in Figures 8 thru 13, and in Table 1. The plots in these figures are on a log-log scale. From the data in in these figures and in Table 1, it is clear that our approach outperforms the RIACS approach. With increasing values of block sizes, the difference in performance is very high. The overhead of sorting in the RIACS approach seems to dominate in these cases. This follows from the fact that the address generation time for the RIACS approach is $O(k \log k)$ and that the plot is a straight line. We have observed a similar performance difference between our solution in this paper and the solution of Kennedy *et al.* [9]. The reader is referred to [16] for detailed results.

In Figure 8, the address generation time drops as we increase k from 4 to 8, since the number of comparisons needed in the loop nest for $k = 4$ using basis vectors $\{(1,5),(1,-2)\}$ is higher than the loop nest for $k = 8$ which uses the basis vectors $\{(1,-4),(0,7)\}$. Figure 9 illustrates the impact of using the best basis vectors on the time needed to determine the address pattern. For $k = 4$ and $k = 8$, our solution gives only an approximation to the best basis vectors; hence the time needed is slightly higher than the approach of [3]. For larger values of k the pattern derivation time actually drops and then slowly increases, since our approach derives the best pair of basis vectors for those cases resulting in lower pattern derivation time. The address pattern generation times for our approach are much lower than that of [3] in Figures 10 and 11. The number of iterations of the outer loop in the pattern generation algorithm decreases from $k = 8$ to $k = 32$ and then increases; this explains the drop observed in these figures for our approach. The best pair of basis

Table 1: Comparison of address pattern generation times (in microseconds) of OUR approach and the RIACS [3] approach for 32 processors on an IBM RS6000

Block Size(k)	s = 7		s = 99		s = k-1		s = k+1		s = pk-1		s = pk+1	
	OUR	RIACS	OUR	RIACS	OUR	RIACS	OUR	RIACS	OUR	RIACS	OUR	RIACS
4	12	15	35	16	8	15	11	15	9	14	6	13
8	7	25	109	27	12	25	17	25	9	24	6	24
16	8	45	43	47	8	46	12	46	9	45	6	44
32	8	95	91	89	7	85	10	85	10	85	7	85
64	9	168	89	174	60	173	89	171	11	169	8	167
128	12	341	84	349	59	344	92	340	14	383	11	341
256	17	764	87	756	61	699	97	688	18	726	16	776
512	26	1564	98	1487	65	1477	107	1584	28	1407	25	1552
1024	45	3243	117	3162	72	3443	126	3413	47	3032	44	3173

vectors that we derive for $s = pk - 1$ and $s = pk + 1$ results in remarkably low pattern determination time as in evident in Figures 12 and 13.

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 efficient compile-time algorithms 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 [9, 17]. Experiments on several platforms indicate the efficacy of our approach. Our approach outperforms the approach of [3]; the difference is more pronounced at large values of block size. We have used our technique in generating communication sets for data parallel programs [18]. Work is in progress is minimizing memory wastage in the case of non-unit alignment strides. Techniques for incremental generation of access sequences and communication sets are being explored. We are planning to extend our approach to multi-dimensional array statements.

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. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortran – a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Language, Compilers and Runtime Environments for Distributed Memory Machines*, pages 39–62. 1992.

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

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

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

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

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

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

[9] K. Kennedy, N. Nedeljkovic, and A. Sethi. A Linear-Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs. Technical Report CRPC-TR-94485-S, Center for Research on Parallel Computation, Rice University, Nov. 1994.

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

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

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

[15] A. Thirumalai and J. Ramanujam. Code generation and optimization for array statements in HPF. Technical Report, Louisiana State University, Nov. 1994.

[16] A. Thirumalai and J. Ramanujam. Address Sequence Determination Using Closed Forms for Basis Vectors of the Array Section Lattice. Technical Report, Louisiana State University, Apr. 1995.

[17] A. Thirumalai and J. Ramanujam. Fast Address Sequence Generation for Data-Parallel Programs Using Integer Lattices. Submitted for publication, April 1995.

[18] A. Thirumalai and J. Ramanujam. HPF Array Statements: Communication Generation and Optimization. To appear in *Proc. 3rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers,* Troy, NY, May 1995.

[19] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Benjamin/Cummings Publishing Co., 1995.

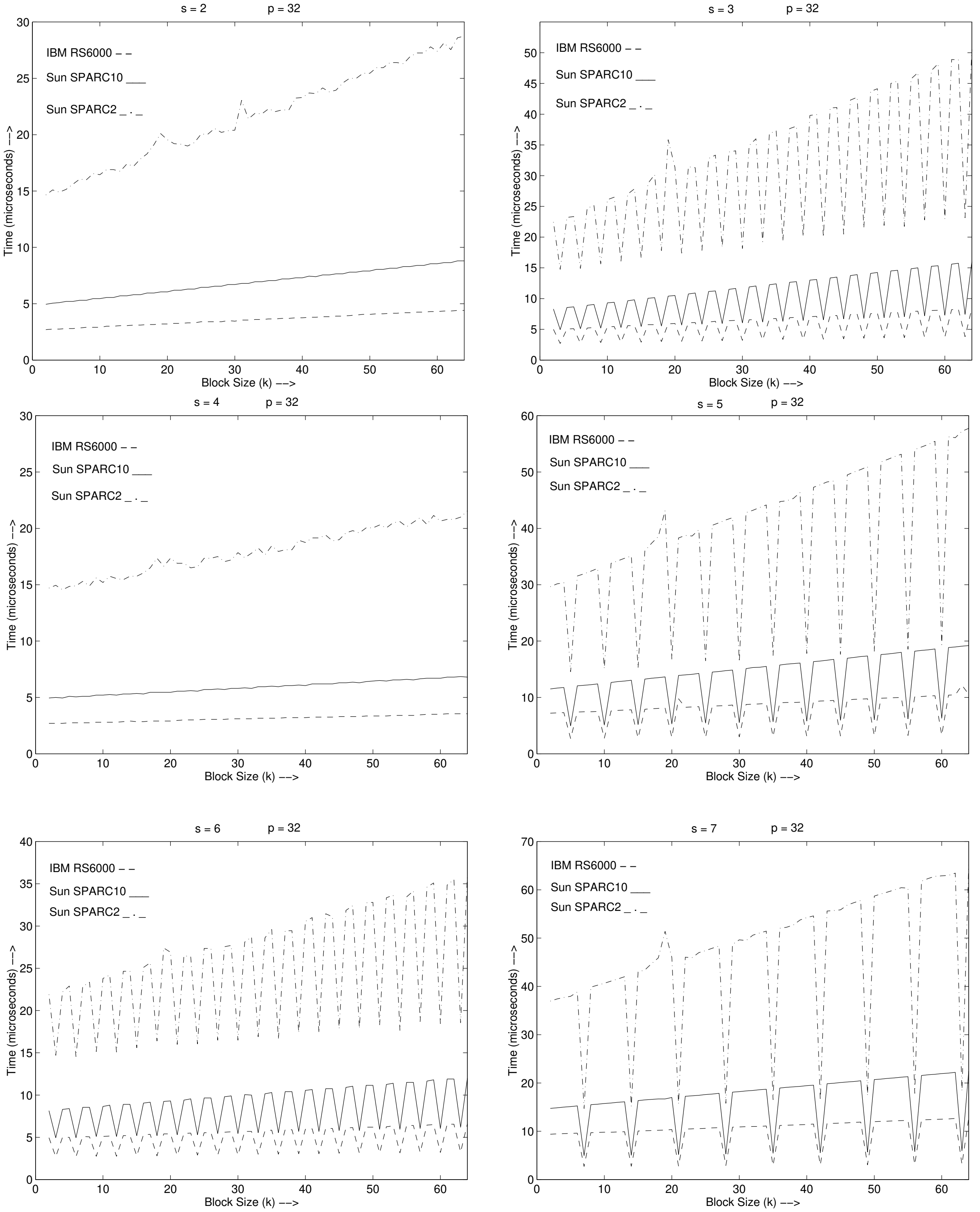

Figure 6: Effect of stride $(2 \cdots 7)$ and block size $(2 \cdots 64)$ for $p = 32$ processors

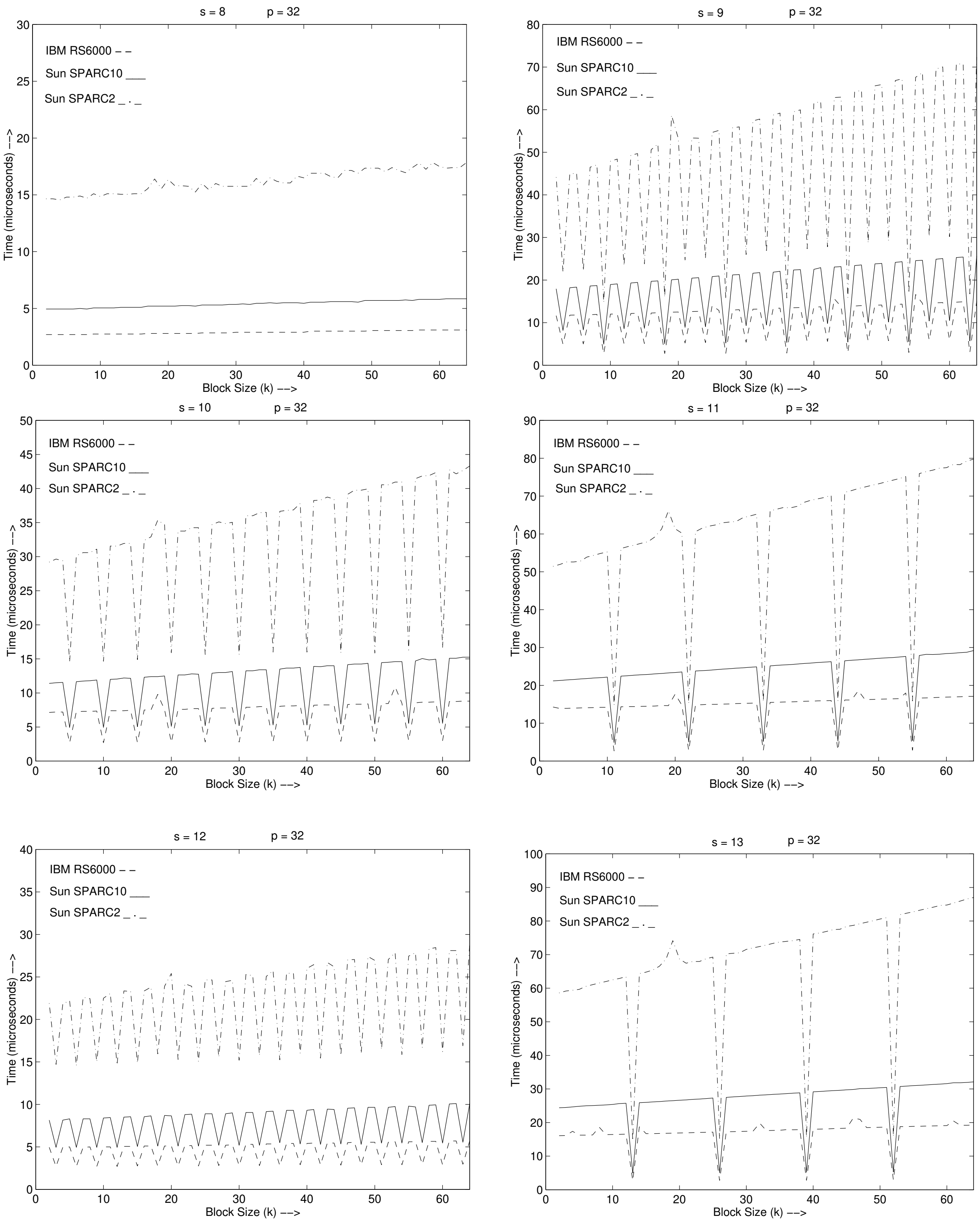

Figure 7: Effect of stride $(8 \cdots 13)$ and block size $(2 \cdots 64)$ for $p = 32$ processors

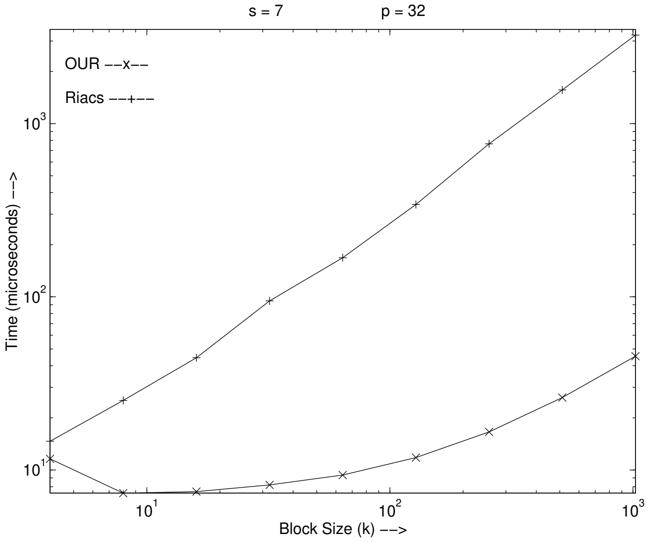

Figure 8: Comparison of our approach to RIACS solution for stride $= 7$ for $p = 32$ processors on an IBM RS6000 for various block sizes

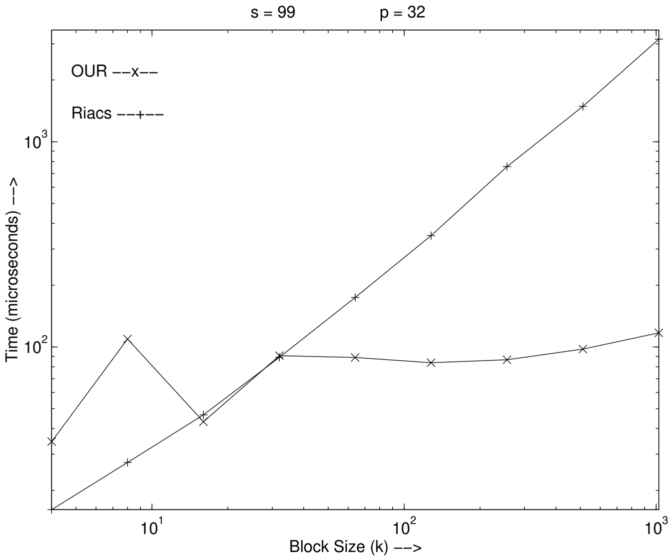

Figure 9: Comparison of our approach to RIACS solution for stride $= 99$ for $p = 32$ processors on an IBM RS6000 for various block sizes

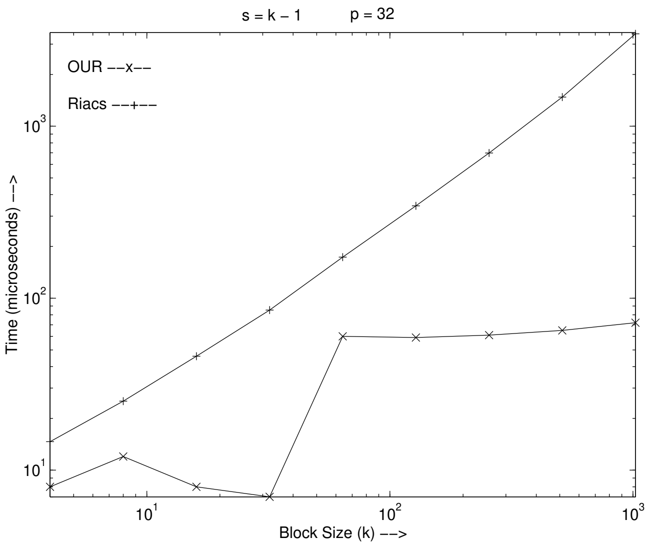

Figure 10: Comparison of our approach to RIACS solution for stride $= k - 1$ for $p = 32$ processors on an IBM RS6000 for various block sizes, k

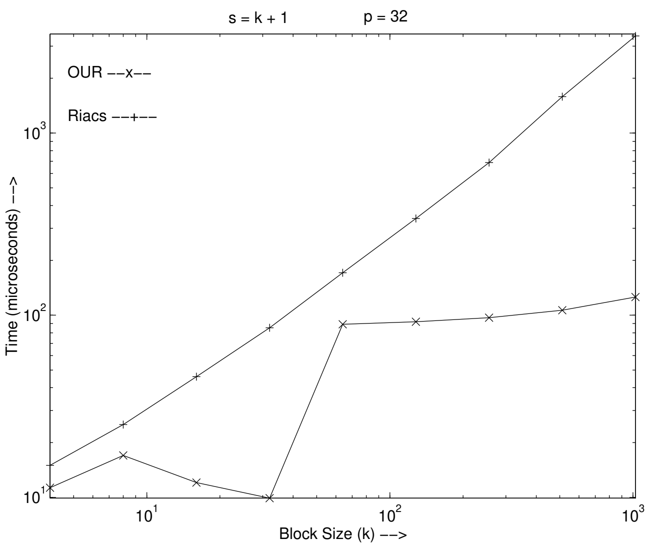

Figure 11: Comparison of our approach to RIACS solution for stride $= k + 1$ for $p = 32$ processors on an IBM RS6000 for various block sizes, k

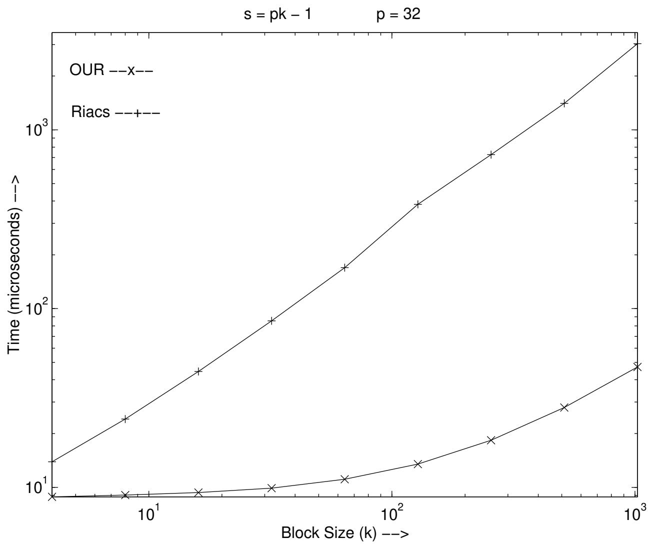

Figure 12: Comparison of our approach to RIACS solution for stride $= pk - 1$ for $p = 32$ processors on an RS6000 for various block sizes, k

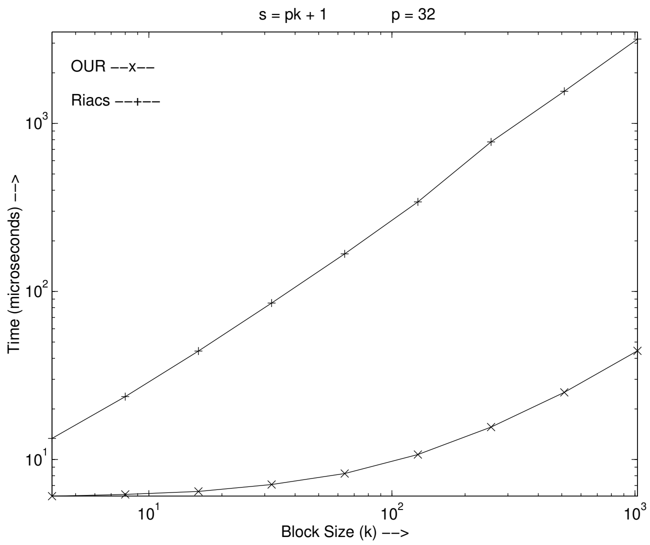

Figure 13: Comparison of our approach to RIACS solution for stride $= pk + 1$ for $p = 32$ processors on an IBM RS6000 for various block sizes, k