A Linear-Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs

Ken Kennedy
Nenad Nedeljković
Ajay Sethi

CRPC-TR94485-S
October, 1994

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Subimtted to the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95).

A Linear-Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs

Ken Kennedy
ken@rice.edu

Nenad Nedeljković
nenad@rice.edu

Ajay Sethi
sethi@rice.edu

Center for Research on Parallel Computation
Department of Computer Science, Rice University

Abstract

Data-parallel languages, such as High Performance Fortran, are designed to facilitate writing of portable programs for distributed-memory machines. Novel features of these languages call for development of new techniques in both compilers and run-time support systems. We present an improved algorithm for finding the local memory access sequence in computations involving regular sections of arrays with *cyclic*(k) distribution. Using the fact that regular section elements form an integer lattice we show how to find a lattice basis that allows for simple and fast enumeration of memory accesses. The complexity of our algorithm is shown to be lower than that of the previous solution for the same problem. In addition, the experimental results demonstrate the efficiency of our method in practice.

This work was supported in part by ARPA contract DABT63-92-C-0038 and NSF Cooperative Agreement Number CCR-9120008. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

High Performance Fortran (HPF) [7, 12] incorporates a set of Fortran extensions for portable data-parallel programming on distributed-memory machines. The most important of these extensions are *align* and *distribute* directives, which are used to describe how data should be distributed across processors in a parallel computer. Array elements are first aligned to *templates* (abstract spaces of indexed positions), and templates are then distributed onto a processor grid. Using this data mapping specification, the compiler must partition the arrays and generate SPMD code which will be executed on each processor.

Several variants of data-parallel Fortran that preceded HPF, such as Fortran D [9] and Vienna Fortran [2], also provided ways for the programmer to specify mapping of array data onto processors. Implementations of these languages included the support for *block* and *cyclic* distributions. Both of these are just special cases of *cyclic*(k) distribution in HPF, which first divides a template into contiguous blocks of size k and then assigns these blocks to processors in a round-robin fashion. Obviously, *cyclic* distribution is equivalent to *cyclic*(1), and *block* distribution of a template of size n onto p processors is equivalent to *cyclic*$(\lceil n/p \rceil)$.

An array A distributed with *cyclic*(k) distribution is effectively split into p subarrays, each being local to one processor. For a program involving the array section $A(l : u : s)$ with the lower bound l, upper bound u, and stride s, the compiler must determine the sequence of local memory addresses that a given processor must access when performing its share of computation over the array section. This problem and its solution were first described by Chatterjee et al. [4]. They give an algorithm for solving the memory address problem in $O(k \log k + \min(\log s, \log p))$ time, and show that any algorithm for this problem takes $\Omega(k)$ time. Hiranandani et al. present an algorithm that works in $O(k)$ time, but only if some special conditions are satisfied ($s \bmod pk < k$) [8]. In this paper we describe an improved algorithm that computes the memory address sequence for the general case in $O(k + \min(\log s, \log p))$ time. Experimental comparison with the method presented in [4] shows that the theoretically proven lower complexity of our algorithm is mirrored by its superior performance in practice. The proposed algorithm allows for simple and efficient implementation and, as such, is suitable for inclusion in compilers and run-time systems for HPF-like languages.

The remainder of this paper is organized as follows: In Section 2 we describe the problem and the solution proposed in [4] in some detail. In Section 3 we show how regular section accesses fit in the framework of integer lattice theory. This is further explored in Section 4, where we lay the theoretical foundation for our algorithm. The algorithm itself and analysis of its running time are presented in Section 5. We describe our implementation experience in Section 6, discuss related work in Section 7, and conclude in Section 8 by summarizing our contributions and indicating directions for future research.

2 Problem Statement

For array A distributed across p processors using *cyclic*(k) distribution, the layout of its elements in local processor memories can be visualized as a two-dimensional matrix, with each row divided into p blocks [4]. The location of array element $A(i)$ is determined by the processor holding $A(i)$, the block within this processor containing $A(i)$, and the offset of $A(i)$ within the block. For example, in Figure 1 array element

$A(108)$ has offset 4 in block 3 of processor 1 (we assume that array elements, offsets, blocks, and processors are numbered starting from zero).

The sequence of local memory locations that given processor m must access when performing its share of computation over the array section $A(l:u:s)$ can be described by the starting location and differences between memory locations of every two successive elements of the array section that belong to processor m. Given number of processors p, array distribution $cyclic(k)$, and regular section stride s, the offset of an array section element determines the offset of the next array section element on the same processor, and also determines the memory gap between those two elements. Since the offsets range from 0 to $k-1$, the sequence of offsets, as well as the sequence of memory gaps, must have a cycle whose length is at most k. Chatterjee et al. visualize the offset and memory gap sequence as the transition diagram of a finite state machine [4]. State transitions depend only on p, k, and s, whereas a processor's start state in the transition table also depends on the lower bound of the array section l and that processor's number m. Note that the upper bound u does not play any role in finding the transition table and the starting state, and is only used to find the last location for each processor.

The starting location for a given processor m is determined by the first element of the array section $A(l:u:s)$ that belongs to processor m. Since array element $A(i)$ belongs to processor m if and only if its offset relative to the beginning of its row ($i \bmod pk$) lies in the range $[km, k(m+1))$, finding the starting location is equivalent to finding the smallest nonnegative integer j such that

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

It is shown in [4] that this is equivalent to solving a set of k linear Diophantine equations

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

in variables j and q. Each individual equation has solutions if and only if GCD(s, pk) divides i. Furthermore, all solutions for each equation can be found by using the extended Euclid's algorithm for computing GCD.

Processor 0								Processor 1								Processor 2								Processor 3							
$\boxed{0}$	1	2	3	4	5	6	7	8	$\boxed{9}$	10	11	12	13	14	15	16	17	$\boxed{18}$	19	20	21	22	23	24	25	26	$\boxed{27}$	28	29	30	31
32	33	34	35	$\boxed{36}$	37	38	39	40	41	42	43	44	$\boxed{45}$	46	47	48	49	50	51	52	53	$\boxed{54}$	55	56	57	58	59	60	61	62	$\boxed{63}$
64	65	66	67	68	69	70	71	$\boxed{72}$	73	74	75	76	77	78	79	80	$\boxed{81}$	82	83	84	85	86	87	88	89	$\boxed{90}$	91	92	93	94	95
96	97	98	$\boxed{99}$	100	101	102	103	104	105	106	107	$\boxed{108}$	109	110	111	112	113	114	115	116	$\boxed{117}$	118	119	120	121	122	123	124	125	$\boxed{126}$	127
128	129	130	131	132	133	134	$\boxed{135}$	136	137	138	139	140	141	142	143	$\boxed{144}$	145	146	147	148	149	150	151	152	$\boxed{153}$	154	155	156	157	158	159
160	161	$\boxed{162}$	163	164	165	166	167	168	169	170	$\boxed{171}$	172	173	174	175	176	177	178	179	$\boxed{180}$	181	182	183	184	185	186	187	188	$\boxed{189}$	190	191
192	193	194	195	196	197	$\boxed{198}$	199	200	201	202	203	204	205	206	$\boxed{207}$	208	209	210	211	212	213	214	215	$\boxed{216}$	217	218	219	220	221	222	223
224	$\boxed{225}$	226	227	228	229	230	231	232	233	$\boxed{234}$	235	236	237	238	239	240	241	242	$\boxed{243}$	244	245	246	247	248	249	250	251	$\boxed{252}$	253	254	255
256	257	258	259	260	$\boxed{261}$	262	263	264	265	266	267	268	269	$\boxed{270}$	271	272	273	274	275	276	277	278	$\boxed{279}$	280	281	282	283	284	285	286	287
$\boxed{288}$	289	290	291	292	293	294	295	296	$\boxed{297}$	298	299	300	301	302	303	304	305	$\boxed{306}$	307	308	309	310	311	312	313	314	$\boxed{315}$	316	317	318	319

Figure 1 Layout of array elements distributed with $cyclic(8)$ distribution over 4 processors. Rectangles indicate elements of the array section with lower bound $l=0$, and stride $s=9$.

For each solvable equation, Chatterjee et al. find the solution having the smallest nonnegative j and use the minimum of these solutions to get the starting array section element $A(l + js)$ [4]. The last array section element can be found in a similar way using the upper bound u. Since our focus is on finding the memory gap sequence, and since the sequence itself is independent of u, we choose to deal with array sections for which only the lower bound l and stride s are specified. Furthermore, we assume that $s > 0$, as the case when s is negative can be treated analogously.

While we follow the approach from [4] for finding the starting location, our method differs in the way we find the offset and memory gap sequences. After finding the set of smallest positive solutions as described above, Chatterjee et al. sort this set to produce the sequence of array section indices that will be successively accessed by the processor [4]. Memory gap sequence can then be found by a simple linear scan through the sorted sequence of array indices. Since sorting the sequence requires $O(k \log k)$ time, it represents the dominating term in the overall complexity of the algorithm. In order to reduce the complexity of finding the memory gap sequence to $O(k)$, we show how array indices can be enumerated in increasing order without actually sorting the sequence.

HPF allows affine alignments between arrays and templates. In other words, array element $A(i)$ can be aligned to a template cell $ai+b$, for arbitrary a and b. Perfect alignment of an array to a template is given by $a = 1, b = 0$. Chatterjee et al. show that the memory access problem for any affine alignment can be solved by two applications of the algorithm for the perfect alignment [4]. Therefore, we present our algorithm only for the case of perfect alignment without loss of generality.

Similarly, since alignments and distributions of each dimension in a multidimensional array are independent of one another, memory access problem for d-dimensional array sections reduces to d applications of the algorithm for one-dimensional case. Consequently, we only need to describe our algorithm for one-dimensional array sections.

3 The Integer Lattice

Our approach is based on treating each array element as a point in $\mathcal{R}^2$ space with the origin corresponding to the array element with index 0, positive y-axis in the direction of increasing row numbers, and positive x-axis in the direction of increasing offsets with respect to the beginning of a row. For example, in Figure 1 the coordinates of the array element with index 108 are $(12, 3)$. Since the access pattern is independent of the starting location l, in deriving our method for enumerating regular section indices, we will assume $l = 0$. This implies that all array section indices (enclosed in rectangles in Figure 1) will be multiples of stride s.

The y-coordinate of an array element is equal to the number of the row to which that element belongs, and the x-coordinate is equal to its offset within that row. Since each row has pk elements, any regular section index and its corresponding coordinates in $\mathcal{R}^2$ satisfy the relationship:

$$pk\, y + x \;=\; i\, s,\; i \in \mathcal{Z}.$$

The following theorem gives the useful characterization of the set of all points in $\mathcal{R}^2$ corresponding to regular section indices.

Theorem 1 *Set* $\Lambda = \{(b,a) \in \mathcal{Z}^2 \mid pk\,a + b \; = \; i\,s, \; i \in \mathcal{Z}\}$ *is an integer lattice*[1].

Proof: Every discrete subset of $\mathcal{R}^n$ closed under subtraction is a lattice [10]. Let $(b_1, a_1) : pk\,a_1 + b_1 = i_1\,s$ and $(b_2, a_2) : pk\,a_2 + b_2 = i_2\,s$ $(i_1, i_2 \in \mathcal{Z})$ be two arbitrary points in Λ. Since

$$pk\,(a_1 - a_2) + (b_1 - b_2) = (i_1 - i_2)\,s,$$

point $(b_1 - b_2, a_1 - a_2)$ is also in Λ. By construction set Λ is discrete, and therefore it is an integer lattice. $\Box$

Our goal is to find a basis for Λ, i.e., the maximal set of linearly independent vectors in Λ from which we can generate all lattice points using integer linear combinations. Since $\Lambda \subset \mathcal{Z}^2$, any basis for Λ can have at most two vectors. If Λ can be generated using only a single vector, it is easy to see that pk must divide s. This special case can be trivially handled in the algorithm, and therefore we will assume that any basis for Λ contains exactly two vectors.

In order for vectors $(b_1, a_1) : pk\,a_1 + b_1 = i_1\,s$ and $(b_2, a_2) : pk\,a_2 + b_2 = i_2\,s$ $(i_1, i_2 \in \mathcal{Z})$ to form a basis, for every lattice point $(b, a) : pk\,a + b = i\,s$ in Λ there have to exist integers α and β, such that

$$(b, a) = \alpha\,(b_1, a_1) + \beta\,(b_2, a_2).$$

The solution of the system of equations

$$\begin{aligned} b_1\,\alpha + b_2\,\beta &= b \\ a_1\,\alpha + a_2\,\beta &= a \end{aligned}$$

is given by

$$(\alpha, \beta) = \left(\frac{ab_2 - a_2 b}{a_1 b_2 - a_2 b_1}, \frac{a_1 b - a b_1}{a_1 b_2 - a_2 b_1} \right) = \left(\frac{a i_2 - a_2 i}{a_1 i_2 - a_2 i_1}, \frac{a_1 i - a i_1}{a_1 i_2 - a_2 i_1} \right).$$

It can be shown that α and β will be integers for every $i \in \mathcal{Z}$, if and only if $a_1 i_2 - a_2 i_1 = \pm 1$. In other words, the necessary and sufficient condition for vectors (b_1, a_1) and (b_2, a_2) to form a basis is given by $|a_1 i_2 - a_2 i_1| = 1$.

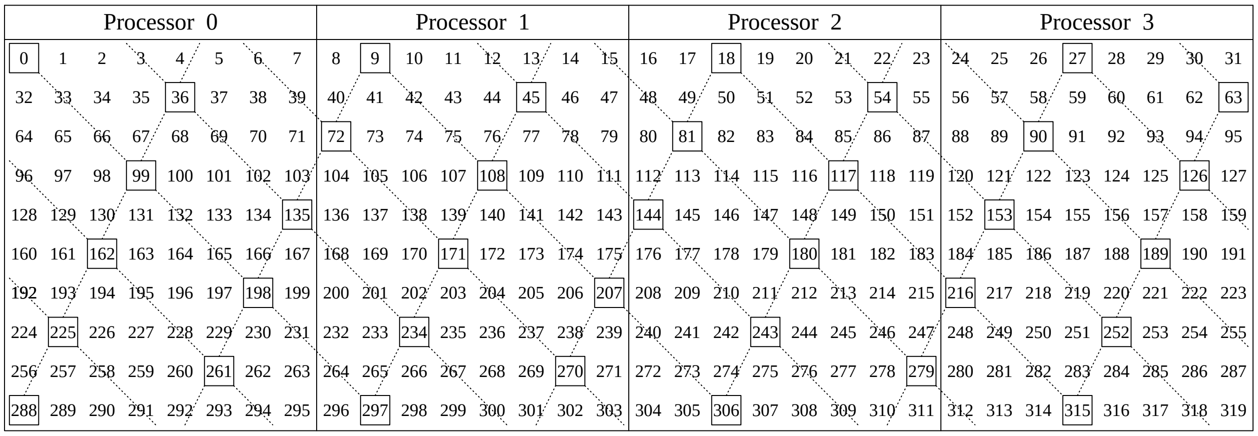
In the example in Figure 2 line segments between lattice points correspond to the vectors $(3, 3) : 3 \times 32 + 3 = 11 \times 9$ and $(-1, 2) : 2 \times 32 - 1 = 7 \times 9$. Since $3 \times 7 - 2 \times 11 = -1$, these vectors form a lattice basis.

Although we now have a simple test for deciding whether two given vectors form a basis, we still need a constructive method for finding a basis of the regular section lattice. If vector $(b, a) : pk\,a + b = i\,s$ belongs to a basis for Λ, then the segment joining $(0, 0)$ and (b, a) does not contain any other points from Λ. It can be shown that (b, a) satisfies this condition, if and only if $\textsc{gcd}(a, i) = 1$. We can now use this fact to find a basis of the lattice Λ.

Take (b_1, a_1) to be an arbitrary lattice point such that there are no other lattice points between $(0, 0)$ and (b_1, a_1); for example, this can be achieved by choosing $i_1 = 1$, $a_1 = s \text{ div } pk$, and $b_1 = s \bmod pk$. Since $\textsc{gcd}(a_1, i_1) = 1$, we can use the extended Euclid's algorithm to find a_2 and i_2, such that

$$a_1 i_2 - a_2 i_1 = 1.$$

[1]A lattice in $\mathcal{R}^n$ is the set of all integer linear combinations of a set of linearly independent vectors in $\mathcal{R}^n$.



Processor 0								Processor 1								Processor 2								Processor 3							
$\boxed{0}$	1	2	3	4	5	6	7	8	$\boxed{9}$	10	11	12	13	14	15	16	17	$\boxed{18}$	19	20	21	22	23	24	25	26	$\boxed{27}$	28	29	30	31
32	33	34	35	$\boxed{36}$	37	38	39	40	41	42	43	44	$\boxed{45}$	46	47	48	49	50	51	52	53	$\boxed{54}$	55	56	57	58	59	60	61	62	$\boxed{63}$
64	65	66	67	68	69	70	71	$\boxed{72}$	73	74	75	76	77	78	79	80	$\boxed{81}$	82	83	84	85	86	87	88	89	$\boxed{90}$	91	92	93	94	95
96	97	98	$\boxed{99}$	100	101	102	103	104	105	106	107	$\boxed{108}$	109	110	111	112	113	114	115	116	$\boxed{117}$	118	119	120	121	122	123	124	125	$\boxed{126}$	127
128	129	130	131	132	133	134	$\boxed{135}$	136	137	138	139	140	141	142	143	$\boxed{144}$	145	146	147	148	149	150	151	152	$\boxed{153}$	154	155	156	157	158	159
160	161	$\boxed{162}$	163	164	165	166	167	168	169	170	$\boxed{171}$	172	173	174	175	176	177	178	179	$\boxed{180}$	181	182	183	184	185	186	187	188	$\boxed{189}$	190	191
192	193	194	195	196	197	$\boxed{198}$	199	200	201	202	203	204	205	206	$\boxed{207}$	208	209	210	211	212	213	214	215	$\boxed{216}$	217	218	219	220	221	222	223
224	$\boxed{225}$	226	227	228	229	230	231	232	233	$\boxed{234}$	235	236	237	238	239	240	241	242	$\boxed{243}$	244	245	246	247	248	249	250	251	$\boxed{252}$	253	254	255
256	257	258	259	260	$\boxed{261}$	262	263	264	265	266	267	268	269	$\boxed{270}$	271	272	273	274	275	276	277	278	$\boxed{279}$	280	281	282	283	284	285	286	287
$\boxed{288}$	289	290	291	292	293	294	295	296	$\boxed{297}$	298	299	300	301	302	303	304	305	$\boxed{306}$	307	308	309	310	311	312	313	314	$\boxed{315}$	316	317	318	319

Figure 2 Line segments corresponding to basis vectors $(3,3)$ and $(-1,2)$.

By defining $b_2 = i_2\,s - pk\,a_2$, we complete the construction of (b_2, a_2), which together with (b_1, a_1), forms a basis for the lattice Λ.

Having a lattice basis allows us to construct all lattice points as integer linear combinations of basis vectors. However, our goal is to enumerate regular section indices within a given processor's range in increasing order, and for this not any lattice basis will do. We now describe the construction of the basis for Λ that makes this enumeration possible.

4 Constructing the Basis

We choose the first vector to correspond to the smallest regular section access on processor 0 (not counting index 0 itself). More precisely, we define $(b_r, a_r) : pk\,a_r + b_r = i_r\,s$ to be the lattice point with the smallest positive i_r, such that $0 \le b_r < k$. Complementary vector $(b_l, a_l) : pk\,a_l + b_l = i_l\,s$ is defined by the regular section index with the largest negative i_l, such that $0 \le b_l < k$. It is clear that $a_r \ge 0$ and $a_l < 0$. Furthermore, the case when $b_r = 0$ or $b_l = 0$ is easily detected in the algorithm, and therefore we can assume that $b_r > 0$ and $b_l > 0$. The two points are constructed so that there can be no lattice point with the first coordinate in the range $[0, k)$ whose corresponding regular section index is smaller than $i_r\,s$ or greater than $i_l\,s$. In other words, the triangle with vertices $(0,0)$, (b_r, a_r), and (b_l, a_l) contains no other lattice points.

Theorem 2 *Vectors $R = (b_r, a_r)$ and $L = (b_l, a_l)$ form a basis for the lattice Λ.*

Proof: Suppose $(b, a) : pk\,a + b = i\,s$ is a point in Λ that cannot be represented as an integer linear combination of R and L. This means that in the equality $(b, a) = \alpha\,R + \beta\,L$ at least one of α, β is not an integer. Let us now look at the point $(b_1, a_1) = \{\alpha\}\,R + \{\beta\}\,L$, where $\{x\} = x - \lfloor x \rfloor$ is the fractional part of x. Since

$$(b_1, a_1) = (b, a) - (\lfloor \alpha \rfloor R + \lfloor \beta \rfloor L)$$

and $(b, a) \in \Lambda$, we conclude that (b_1, a_1) is also in the lattice Λ.

By definition of the fractional part, we have $0 \leq \{x\} < 1$, for any $x \in \mathcal{R}$. From this, using the fact that at least one of α, β is not an integer, we get $0 < \{\alpha\} + \{\beta\} < 2$. We now consider the following two cases:

1. If $0 < \{\alpha\} + \{\beta\} \leq 1$, then (b_1, a_1) lies within the triangle with vertices $(0,0)$, (b_r, a_r), and (b_l, a_l), which contradicts the construction of R and L.

2. If $1 < \{\alpha\} + \{\beta\} < 2$, then we look at the point $(b_2, a_2) = R + L - (b_1, a_1)$. Since (b_1, a_1) is in Λ, so is (b_2, a_2). From the definition of (b_2, a_2) we have

$$(b_2, a_2) = (1 - \{\alpha\})\, R + (1 - \{\beta\})\, L.$$

   Using the fact that $1 < \{\alpha\} + \{\beta\} < 2$ we get $0 < (1 - \{\alpha\}) + (1 - \{\beta\}) < 1$, which means that (b_2, a_2) lies within the triangle with vertices $(0,0)$, (b_r, a_r), and (b_l, a_l), which is again in contradiction with the way R and L were constructed.

From the above we conclude that every $(b, a) \in \Lambda$ must be an integer linear combination of R and L. □

Basis vectors R and L for our example are shown in Figure 3. We now describe how to use these vectors to enumerate the regular section accesses for a given processor m in increasing order.

Let $(b_1, a_1) : pk\, a_1 + b_1 = i_1\, s$ be an arbitrary regular section index within the range of processor m, i.e., $mk \leq b_1 < (m+1)k$. If $b_1 + b_r < (m+1)k$, then by the construction of R, index $i_2\, s$, corresponding to the point

$$(b_2, a_2) = (b_1, a_1) + R, \tag{1}$$

is the smallest regular section index in the processor m's range that is larger than $i_1\, s$.

Suppose now that $b_1 + b_r \geq (m+1)k$, and let us look at the point $(b_2, a_2) : pk\, a_2 + b_2 = i_2\, s$ given by

$$(b_2, a_2) = (b_1, a_1) - L. \tag{2}$$

Since $i_l\, s < 0$, we have $i_2\, s = i_1\, s - i_l\, s > i_1\, s$.

If $b_2 \geq mk$, i.e., (b_2, a_2) belongs to processor m, by the construction of L, $i_2\, s$ is the smallest regular section index in this processor's range that is larger than $i_1\, s$.

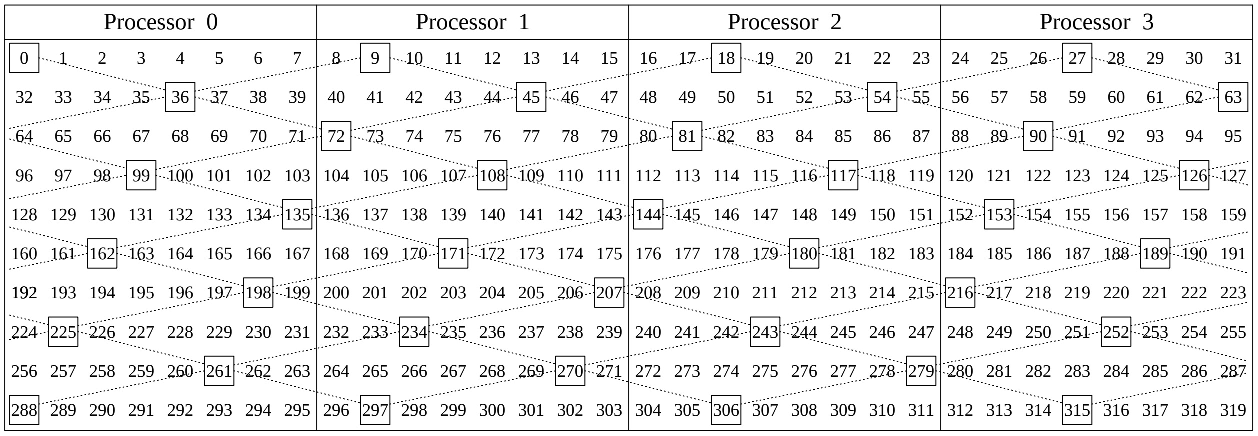
If (b_2, a_2) is outside the range of processor m, i.e., $b_2 < mk$, then the point

$$(b_3, a_3) = (b_2, a_2) + R = (b_1, a_1) - L + R \tag{3}$$

is within the given range ($b_3 = b_1 + b_r - b_l \geq (m+1)k - k = mk$). Regular section index $i_3\, s$ corresponding to (b_3, a_3) is obviously larger than $i_2\, s$, and consequently larger than $i_1\, s$. Since (b_3, a_3) is the first lattice point after (b_2, a_2) that belongs to processor m, we conclude that $i_3\, s$ is the smallest regular section index in the processor m's range that is larger than $i_1\, s$.

Thus, we have effectively proven the following:

Theorem 3 *The distance between two points corresponding to consecutive regular section accesses on the same processor can have one of three possible values: R, $-L$, or $R - L$.*



Processor 0								Processor 1								Processor 2								Processor 3							
$\boxed{0}$	1	2	3	4	5	6	7	8	$\boxed{9}$	10	11	12	13	14	15	16	17	$\boxed{18}$	19	20	21	22	23	24	25	26	$\boxed{27}$	28	29	30	31
32	33	34	35	$\boxed{36}$	37	38	39	40	41	42	43	44	$\boxed{45}$	46	47	48	49	50	51	52	53	$\boxed{54}$	55	56	57	58	59	60	61	62	$\boxed{63}$
64	65	66	67	68	69	70	71	$\boxed{72}$	73	74	75	76	77	78	79	80	$\boxed{81}$	82	83	84	85	86	87	88	89	$\boxed{90}$	91	92	93	94	95
96	97	98	$\boxed{99}$	100	101	102	103	104	105	106	107	$\boxed{108}$	109	110	111	112	113	114	115	116	$\boxed{117}$	118	119	120	121	122	123	124	125	$\boxed{126}$	127
128	129	130	131	132	133	134	$\boxed{135}$	136	137	138	139	140	141	142	143	$\boxed{144}$	145	146	147	148	149	150	151	152	$\boxed{153}$	154	155	156	157	158	159
160	161	$\boxed{162}$	163	164	165	166	167	168	169	170	$\boxed{171}$	172	173	174	175	176	177	178	179	$\boxed{180}$	181	182	183	184	185	186	187	188	$\boxed{189}$	190	191
192	193	194	195	196	197	$\boxed{198}$	199	200	201	202	203	204	205	206	$\boxed{207}$	208	209	210	211	212	213	214	215	$\boxed{216}$	217	218	219	220	221	222	223
224	$\boxed{225}$	226	227	228	229	230	231	232	233	$\boxed{234}$	235	236	237	238	239	240	241	242	$\boxed{243}$	244	245	246	247	248	249	250	251	$\boxed{252}$	253	254	255
256	257	258	259	260	$\boxed{261}$	262	263	264	265	266	267	268	269	$\boxed{270}$	271	272	273	274	275	276	277	278	$\boxed{279}$	280	281	282	283	284	285	286	287
$\boxed{288}$	289	290	291	292	293	294	295	296	$\boxed{297}$	298	299	300	301	302	303	304	305	$\boxed{306}$	307	308	309	310	311	312	313	314	$\boxed{315}$	316	317	318	319

Figure 3 Basis vectors $R = (4, 1)$ and $L = (5, -1)$.

Vector R can be found in much the same way as the starting location for processor m. We simply find the minimum of the smallest positive regular section indices over all offsets in the range $(0, k)$ that have at least one such index. Vector L is computed by finding the maximum of these indices, and taking the coordinates of this point relative to the point that starts the next sequence, i.e., the first positive index whose offset equals 0. In the example in Figure 3 the smallest positive index on processor 0 is 36 and therefore $R = (4, 1)$. The largest index in the first sequence is 261, and since the point that starts the next sequence is 288, we have $L = (5, 8) - (0, 9) = (5, -1)$.

5 The Linear-Time Algorithm

The linear-time algorithm to compute the local memory access sequence, which is based on Theorem 3 from the previous section, is given in Figure 4. As mentioned in Section 2, we use the approach described in [4] to compute the starting location for a given processor. First, we use the extended form of Euclid's algorithm [5] to compute $d = \text{GCD}(s, pk)$. Using this value, we can find the smallest regular section index for each offset in the range of processor m (lines 3–9). Unlike the algorithm in [4], which stores all these locations and later sorts them, we are only interested in the first location for the processor.

If the length of the sequence is greater than 1, then we compute the basis vectors R and L by considering the processor 0's memory access space for $l = 0$ (lines 16–22). Since we are interested in finding the first point after $(0, 0)$, we start with the loop lower bound 1 ($i = 0$ gives the first solution: $(0, 0)$). The smallest location gives us vector $R = (b_r, a_r)$, while $L = (b_l, a_l)$ is computed using the coordinates of the largest location relative to the point that starts the next sequence (lines 23–24).

Once we have found vectors R and L, we begin at the the starting location (line 26) and apply equation 1 from Section 4 until the range of offsets owned by processor m is exceeded (lines 29–33). The distance $R = (b_r, a_r)$ between two consecutive regular section indices results in the local memory access gap of $a_r\, k + b_r$.

Input: Distribution parameters (p, k), regular section parameters (l, s) and processor number (m).

Output: The $\Delta\mathcal{M}$ table and the length of the table $(length)$.

Method:

```
$start = \infty$; $length = 0$; $min = \infty$; $max = 0$
$(d, x, y) \leftarrow$ EXTENDED-EUCLID$(s, pk)$

$\triangleright$ Find the starting location for processor $m$.
for $i = km - l, km - l + k - 1$ do
    if $(i \bmod d = 0)$ then                          $\triangleright$ Equation has solutions.
        $loc = l + \frac{s}{d}(ix + pk\lceil -\frac{ix}{pk} \rceil)$
        $start = \min(start, loc)$; $length = length + 1$
    endif
endfor

if $(length = 0)$ then
    return $\emptyset$, $length$
else if $(length = 1)$ then
    $\Delta\mathcal{M}[0] = ks/d$
    return $\Delta\mathcal{M}, length$
endif

$\triangleright$ Find the parameters of the two families of lines from processor 0. Assume $l = 0$.
for $i = 1, k - 1$ do
    if $(i \bmod d = 0)$ then                          $\triangleright$ Equation has solutions.
        $loc = \frac{s}{d}(ix + pk\lceil -\frac{ix}{pk} \rceil)$
        $min = \min(min, loc)$; $max = \max(max, loc)$
    endif
endfor

$(b_r, a_r) = (min \bmod pk, min/pk)$
$(b_l, a_l) = (max \bmod pk, max/pk - s/d)$

$\triangleright$ Compute the memory access sequence.
$offset = start \bmod pk$
$i = 0$
while $(i < length)$ do
    while $(i < length$ and $offset + b_r < k(m + 1))$ do
        $\Delta\mathcal{M}[i] = a_r\, k + b_r$                    $\triangleright$ Apply equation 1
        $offset = offset + b_r$
        $i = i + 1$
    endwhile
    if $(i = length)$ break                            $\triangleright$ If the whole sequence has been found, stop

    $\Delta\mathcal{M}[i] = -(a_l\, k + b_l)$                   $\triangleright$ Apply equation 2
    $offset = offset - b_l$
    if $(x < km)$ then
        $\Delta\mathcal{M}[i] = \Delta\mathcal{M}[i] + a_r\, k + b_r$   $\triangleright$ Apply equation 3
        $offset = offset + b_r$
    endif
    $i = i + 1$
endwhile

return $\Delta\mathcal{M}, length$
```

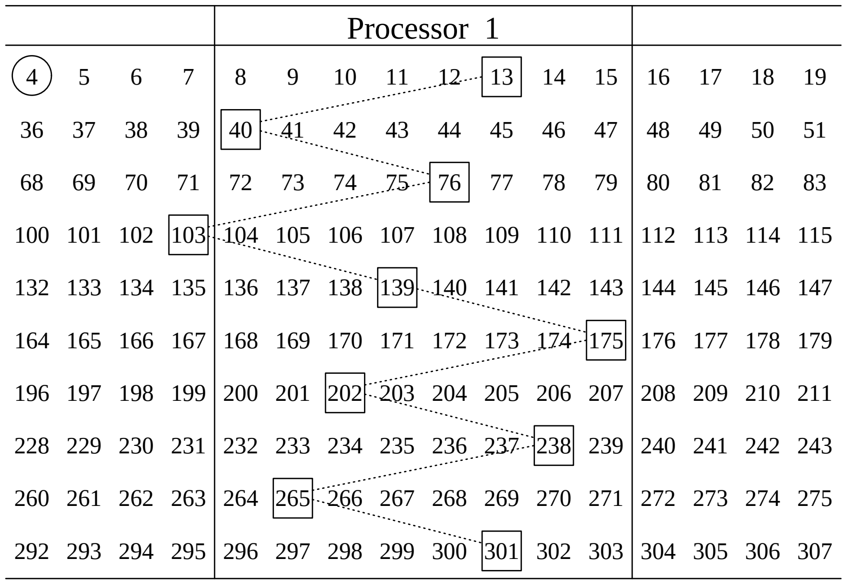
Figure 4 Algorithm to compute the memory access sequence.

If the range is exceeded we move to the next point using equation 2 (lines 35–36), and compute the corresponding memory gap. However, it is possible that this point is outside the processor m's range (line 37), in which case we apply equation 3 and adjust the memory gap accordingly (lines 38–39).

We illustrate the algorithm by showing how it computes the access sequence for processor 1 for the example in Figure 5. The input parameters are $p = 4$, $k = 8$, $l = 4$, $s = 9$ and $m = 1$. The lower bound of the regular section is enclosed in the circle, while the rectangles mark exactly those points that are visited in the algorithm.

Values returned by EXTENDED-EUCLID are $d = 1$, $x = -7$, and $y = 2$. Lines 4–9 compute $start = 13$ and set $length = 8$. Lines 17–22 find $min = 36$ and $max = 261$ (for processor 0 and with $l = 0$). Therefore, $(b_r, a_r) = (4, 1)$ and $(b_l, a_l) = (5, -1)$ (lines 23–24). Line 26 finds the offset of the first location on processor 1: *offset* $= 13$.

The outer **while** loop (lines 27–43) is executed five times. In the first iteration of this loop we do not execute the inner **while** loop (lines 29–33), because the point $(13, 0) + (4, 1) = (17, 1)$ (corresponding to index 49) exceeds the offset range of processor 1. Instead we visit the point $(13, 0) - (5, -1) = (8, 1)$, which corresponds to index 40 (line 35–36), and compute $\Delta\mathcal{M}[0] = -(-1 \times 8 + 5) = 3$. Since this point belongs to processor 1 no adjustment to $\Delta\mathcal{M}[0]$ is necessary. In the next iteration of the outer loop, we visit index 76 in the inner loop, setting $\Delta\mathcal{M}[1] = 12$ (lines 30–31) After terminating the inner loop, the next index visited is 103, which does not belong to processor 1, and therefore we move to the point 139, setting $\Delta\mathcal{M}[2] = 15$ (lines 38–39). The process is continued until we reach the first point of the next sequence, index 301, and at the end, $\Delta\mathcal{M} = [3, 12, 15, 12, 3, 12, 3, 12]$.



				Processor 1											
$\textcircled{4}$	5	6	7	8	9	10	11	12	$\boxed{13}$	14	15	16	17	18	19
36	37	38	39	$\boxed{40}$	41	42	43	44	45	46	47	48	49	50	51
68	69	70	71	72	73	74	75	$\boxed{76}$	77	78	79	80	81	82	83
100	101	102	$\boxed{103}$	104	105	106	107	108	109	110	111	112	113	114	115
132	133	134	135	136	137	138	$\boxed{139}$	140	141	142	143	144	145	146	147
164	165	166	167	168	169	170	171	172	173	174	$\boxed{175}$	176	177	178	179
196	197	198	199	200	201	$\boxed{202}$	203	204	205	206	207	208	209	210	211
228	229	230	231	232	233	234	235	236	237	$\boxed{238}$	239	240	241	242	243
260	261	262	263	264	$\boxed{265}$	266	267	268	269	270	271	272	273	274	275
292	293	294	295	296	297	298	299	300	$\boxed{301}$	302	303	304	305	306	307

Figure 5 Points visited by the algorithm when $p = 4$, $k = 8$, $l = 4$, $s = 9$, and $m = 1$.

5.1 Complexity

The running time of the extended Euclid's algorithm is $O(\log \min(s, pk))$ [5]. The loops in lines 4–9 and 17–22 of Figure 4 are both $O(k)$. Finally, the doubly nested loop in lines 28–42 does only $O(k)$ work; in fact, we show that, in the worst case, at most $2k + 1$ points are examined.

The algorithm visits all regular section indices belonging to the initial sequence, plus the first point in the next sequence. Since the length of the sequence is $\leq k$, this means that at most $k + 1$ points belonging to processor m are visited. In addition the algorithm could also visit some extra points, i.e., some regular section indices that are outside the processor m's range. This happens if after applying equation 2, the resulting point does not belong to processor m, and therefore we have to apply equation 3 (lines 37–40). In the worst case, the inner **while** loop could always be empty, and all points in the initial sequence could require the application of equation 3. This results in visiting k extra points, which together with $k+1$ indices belonging to processor m brings the total number of points examined to $2k + 1$.

Therefore, we conclude that the running time of the algorithm is $O(\log \min(s, pk)) + O(k)$, which reduces to $O(k + \min(\log s, \log p))$. Since it was shown in [4] that the problem is $\Omega(k)$, our algorithm is $O(\min(\log s, \log p))$ away from being theoretically optimal. This term in the complexity equation comes from the use of the extended Euclid's algorithm to find the starting memory location for a given processor, and we believe that, in order to solve this problem for the general case, at least one GCD needs to be computed.

6 Experimental Results

Despite its theoretical advantage, the algorithm presented in Section 5 cannot be a method of choice for solving the memory access problem in compilers and run-time systems for HPF-like languages, unless it allows for an efficient practical implementation. We now describe our implementation experience, which shows that our algorithm is more efficient in practice than the method described in [4]. We also discuss the impact that the shape of the node code has on the overall performance.

6.1 Table Construction

The description of the algorithm in Figure 4 provides enough low-level details to directly convert the algorithm into working code. In order to perform a correct comparison with the algorithm from [4], we modified the code provided to us by Siddhartha Chatterjee [3] so that the segments common to both methods (lines 3–9 in Figure 4) were coded identically. Moreover, since the method by Chatterjee et al. requires sorting of the initial sequence of memory accesses, we tried to use the most efficient sorting routines available to us, so as not to obtain an unfair advantage over the algorithm in [4].

If input parameters p, k, l, and s for our algorithm are compile-time constants, then the compiler could compute the table of memory gaps ($\Delta\mathcal{M}$) for each processor. In that case the code that computes the parameters of the two line families (lines 16–24 in Figure 4) would have to be executed only once, and values of (b_r, a_r) and (b_l, a_l) could be reused. Furthermore, as noted in [4], if $\text{GCD}(s, pk) = 1$, then the local $\Delta\mathcal{M}$

sequences are cyclic shifts of one another, and after computing the table once, only the starting locations for all the processors need to be found. If values of some of the input parameters are not known at compile time, then the memory access sequence has to be computed at run time. In other words, every processor would run the algorithm from Figure 4 by supplying its processor number m. Our experiments were designed to compare the performance of the two methods in this case, when every processor has to execute the complete version of either algorithm.

Since the lower bound of the regular section has almost no influence on the running time of the algorithm, all our experiments were performed with $l = 0$. Similarly, the effects of varying the number of processors are only minor, and therefore we always used $p = 32$. The two varying parameters are block size k and stride s. We used powers of 2 for the block size, since these are the most likely values to be encountered in practice (cases when $k = 1$ or $k = 2$ are not reported because the amount of work done by either algorithm is negligible). We experimented with several values of stride s, and took into consideration two perhaps unusual cases: $s = pk - 1$, and $s = pk + 1$. The reason for this was that these cases result in reversely ($s = pk - 1$) and properly ($s = pk + 1$) sorted access sequences and as such are interesting to test the behavior of the sorting routine.

Table 1 contains the execution times that our algorithm and the method by Chatterjee et al. take to find the memory gap sequence ($\Delta\mathcal{M}$ table in Figure 4) for different values of k and s. In Figure 6 we plot the execution times for the case when $s = 7$. All measurements were performed on an Intel iPSC/860 hypercube, using the *icc* compiler with –O4 optimization level and *dclock* timer. Reported times represent maximums over all 32 processors and are given in microseconds.

While the difference in performance of the two algorithms is not significant for small values of k ($k = 4$, $k = 8$), as k increases the algorithm in Figure 4 clearly outperforms the algorithm described by Chatterjee et al. in [4]. It should be noted here that the implementation from [3] uses the linear-time radix sort for sorting the initial sequence when $k \geq 64$, which causes the relative performance gain achieved by our algorithm to be constant. However, if a sorting method that sorts the sequence in place were used, for larger values of k relative performance improvement would also increase.

	$s = 7$		$s = 99$		$s = k + 1$		$s = pk - 1$		$s = pk + 1$	
	Lines	Sorting	Lines	Sorting	Lines	Sorting	Lines	Sorting	Lines	Sorting
$k = 4$	48	56	60	68	52	65	44	53	40	48
$k = 8$	58	82	70	94	53	78	49	75	44	70
$k = 16$	60	138	76	145	65	134	60	140	54	133
$k = 32$	83	286	95	295	81	287	81	288	72	276
$k = 64$	122	775	140	749	132	747	124	735	109	727
$k = 128$	183	1384	232	1451	201	1453	203	1385	181	1367
$k = 256$	332	2708	394	2814	340	2730	368	2713	325	2776
$k = 512$	614	5550	679	5281	618	5328	698	5312	617	5262

Table 1 Execution times in microseconds for our algorithm (Lines), and the algorithm in [4] (Sorting).

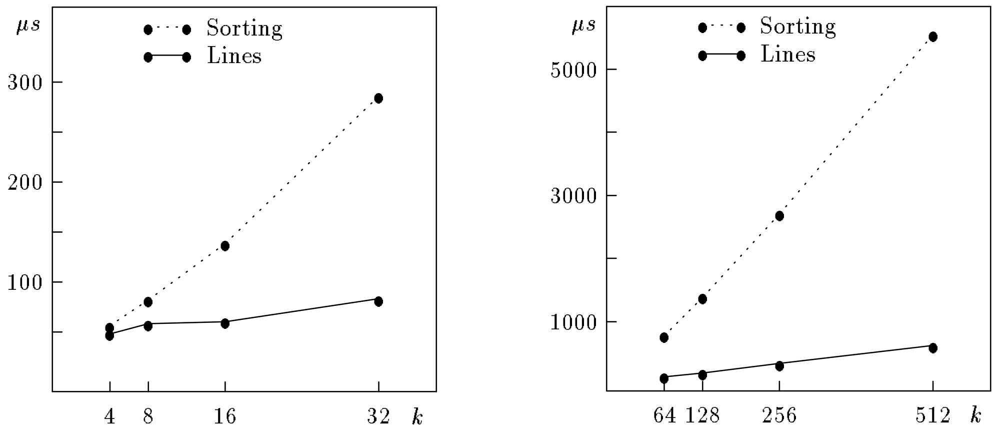


Figure 6 Performance of the two table construction algorithms for the case $s = 7$.

6.2 Code Generation

After the table of local memory gaps is constructed, each processor uses its table to access the array section elements that it owns. In order to achieve good performance great care has to be taken when generating the node code.

In Figure 7 we show four different ways to generate the node code based on the memory sequence table. The C code fragments correspond to the simple array assignment statement $A(l : u : s) = 100.0$. The code in 7(a) is identical to that proposed in [4]. In an attempt to remove the usually expensive mod operation, we replace it with a simple test in 7(b). A slight modification of the same idea is shown in 7(c).

While these three code versions require only the table of local memory gaps (`deltaM`), the code in 7(d) also requires the offset sequence table (`nextoffset`) and the offset of the starting memory location for a given processor. Although our algorithm as presented does not directly provide these values, it is easy to modify it to get this additional information. All that is needed is to insert stores into an array of offsets at every point where variable *offset* (lines 26, 31, 36, and 39 in Figure 4) is updated.

Experiments with different versions of the node code were performed in the same environment as described in the previous section. The code fragments from Figure 7 were executed on 16 processors. Lower bound l was always 0, while the upper bound was scaled in proportion to stride s, in order to keep the number of memory accesses constant. The execution times reported in Table 2 are for the case when each processor performed assignments to 10,000 array elements.

The most notable is the very poor performance of the code that uses mod operations compared to the other three versions. The version of the node code in Figure 7(c) is somewhat faster than that in 7(b), with the difference increasing with larger block sizes. The main reason for this was better instruction scheduling by the *icc* compiler in the version 7(c). The best performance was achieved using the code from Figure 7(d). Although this version requires two table lookups per array access, its simple structure makes it more efficient than the others, especially for smaller values of k.

```
base = startmem; i = 0;
while (base <= lastmem) {
   *base = 100.0;
   base += deltaM[i];
   i = (i+1) % tablesize;
}
```
(a)

```
base = startmem; i = 0;
while (base <= lastmem) {
   *base = 100.0;
   base += deltaM[i++];
   if (i == tablesize) i = 0;
}
```
(b)

```
base = startmem; i = 0;
while (TRUE) {
   for (i = 0; i < tablesize; i++) {
      *base = 100.0;
      base += deltaM[i];
      if (base > lastmem) goto done;
   }
}
done:
```
(c)

```
base = startmem;
i = startoffset;
while (base <= lastmem) {
   *base = 100.0;
   base += deltaM[i];
   i = nextoffset[i];
}
```
(d)

Figure 7 Possible versions of the node code.

It was noted by Knies et al. that the code based on table lookup makes a time versus space tradeoff [11]. This is particularly true for the code in Figure 7(d), which while being the fastest, requires two tables to be stored. However an important feature of our method is that the algorithm can be modified to return only vectors $R = (b_r, a_r)$ and $L = (b_l, a_l)$, without storing any tables. Using these values, every processor can generate the memory addresses as needed, using simple tests similar to those in lines 29 and 37 in Figure 4. In this way the memory overhead is eliminated, but the performance will still be much better than if we used the full run-time generation of addresses as described in [11].

Code shape		7(a)	7 (b)	7(c)	7(d)
$k = 4$	$s = 3$	17976	3217	3095	2292
	$s = 15$	18060	3450	3326	2532
	$s = 99$	18541	3916	3823	3065
$k = 32$	$s = 3$	17980	3276	2606	2299
	$s = 15$	18070	3504	2845	2547
	$s = 99$	18533	3983	3335	3083
$k = 256$	$s = 3$	18122	3316	2573	2357
	$s = 15$	18081	3533	2797	2598
	$s = 99$	18567	4000	3294	3149

Table 2 Execution times in microseconds for different node code versions.

7 Related Work

Besides the work by Chatterjee et al. [4] that has been extensively cited throughout this paper, several other researchers have also dealt with issues of compiling programs with *cyclic*(k) distribution.

Gupta et al. address the problem of array statements $(A(l_a:u_a:s_a) = B(l_b:u_b:s_b))$ involving block-cyclic distributions [6]. In their virtual-cyclic scheme, array elements are accessed in an order different from the order in a sequential program. While this is not a problem in perfectly parallel array assignments, the scheme cannot be used for arbitrary loops accessing block-cyclically distributed arrays. In the virtual-block scheme array accesses are not reordered, but if the array section stride is larger than the block size, their method effectively reduces to run-time address resolution.

In an approach similar to the virtual-cyclic scheme Stichnoth et al. use intersections of array slices for communication generation [13]. As mentioned above, a disadvantage of this method is that array accesses are reordered.

Ancourt et al. use a linear algebra framework for compiling *independent* loops in HPF [1]. Because of the independent parallelism they assume that loop iterations can be enumerated in any order. Generated loop bounds and local array subscripts can be quite complex, and thus introduce a significant overhead.

8 Conclusions

Widespread use of data-parallel languages, such as High Performance Fortran, will not come about until fast compilers and efficient run-time support systems are developed. In this paper, we have presented a new algorithm for generating the local memory access sequence for computations over regular sections of arrays that are distributed using HPF *cyclic*(k) distribution. We have shown that the theoretical complexity of our method is superior to the previously known solution for the same problem. Furthermore, our implementation experience indicates that the proposed algorithm is also more efficient in practice, which makes it a preferred choice for compilers and run-time systems for HPF-like languages.

Although we know how to efficiently compile codes that access regular sections of arrays with *cyclic*(k) distributions, many questions in HPF compilation remain unresolved. Some of the problems that need to be investigated are compiling programs that access diagonal or trapezoidal array sections and optimizing communication resulting from non-local accesses of array elements, both in the presence of *cyclic*(k) distribution. Only after these problems are fully and efficiently solved, can the use of such novel features of HPF become commonplace.

Acknowledgments

We would like to thank Siddhartha Chatterjee for providing us the code that implements the algorithm described in [4] and for insightful comments on an earlier draft of this paper. We also thank Debbie Campbell and Lani Granston for their help in proofreading various drafts of the paper.

References

[1] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

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

[3] S. Chatterjee. Private communication, October 1994.

[4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[6] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical Report OSE-CISRC-4/94-TR19, Department of Computer and Information Science, The Ohio State University, April 1994.

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

[8] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. In *Proceedings of the 1994 ACM International Conference on Supercomputing*, Manchester, England, July 1994.

[9] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[10] R. Kannan. Algorithmic geometry of numbers. In J. Traub, editor, *Annual Review of Computer Science.* Annual Reviews Inc., Palo Alto, CA, 1987.

[11] A. Knies, M. O'Keefe, and T. MacDonald. High Performance Fortran: A practical analysis. *Scientific Programming*, 3(3):187–199, Fall 1994.

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

[13] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.