Code Generation for Complex Subscripts in Data-Parallel Programs

J. Ramanujam Swaroop Dutta Arun Venkatachar
Dept. of Electrical & Computer Engineering
Louisiana State University, Baton Rouge, LA 70803-5901, USA
{jxr, swaroop, arun}@ee.lsu.edu

Abstract

Data parallel languages like High Performance Fortran, demand efficient compile and run-time techniques for tasks such as address generation. Array references with arbitrary affine subscripts can make the task of compilers for such languages highly involved. This paper deals with the efficient address generation in programs with array references having two types of commonly encountered affine references, namely coupled subscripts and subscripts containing multiple induction variables (MIVs). Methods discussed in the paper utilize the repetitive pattern of the memory accesses. In the case of MIV, we address this issue by presenting runtime techniques which enumerate the set of addresses in lexicographic order. Our approach to the problem incorporates a general approach of computing in $O(k)$ time, the start element on a processor for a given global start element. Several methods are proposed and evaluated here for generating the access sequences for MIV based on problem parameters. With coupled subscripts, we present two construction techniques, namely searching and hashing which minimize the time needed to construct the tables. Extensive experiments were conducted and the results were then compared with [8] to indicate the efficiency of our approach.

1 Introduction

Languages such as High Performance Fortran (HPF) [6] and Vienna Fortran [2] are being used to program massively parallel distributed memory machines. The compiler directives in these languages allow a programmer to specify the details about the mapping of the data onto processors. Two types of distribution that have been studied extensively are the `BLOCK` and the `CYCLIC` distributions. A more general type of distribution is the block-cyclic or `CYCLIC`(k) distribution, which imposes an additional burden on the compiler by rendering address generation of the memory accesses involved expensive. Therefore the need arises for time- and space-efficient techniques to generate these addresses. Several earlier papers deal with address generation for `CYCLIC`(k) distribution for simple subscripts [1, 3, 5, 7, 8, 10, 12, 13, 14, 15, 16, 17]. The array references in general may involve complex subscript functions. This introduces an additional burden on the compiler to generate efficient run-time SPMD code. The two particularly interesting types of affine subscripts that occur commonly in such programs are the *multiple induction variables* (MIV's) and the *coupled subscripts* [8, 11]. An MIV is the affine combination of two or more nested loop indices in a single dimension. Coupled subscripts arise when the same loop index variable occurs in two distinct dimensions. Methods for address generation for one-dimensional arrays with `CYCLIC`(k) distribution do not apply directly to these more complex affine subscript cases discussed in this paper [8, 16]. This paper deals with address generation for the above mentioned affine subscripts.

Our technique for code generation for MIVs involve building tables for exploiting the pattern of accesses. As part of this, we have presented a new general method of computing start element on a processor. This method does not incur much runtime overhead, and serves as a big advantage in order to generate starting elements on each processor for a given global start. The complexity of this method is $O(k)$ when the block size k is greater than the stride s, and $O(1)$ otherwise. We use this technique of computing start to present an $O(pk^2)$ algorithm to compute accessed elements in case of MIV. The reduced complexity and benefit of constructing several pattern tables proves to be beneficial during execution for MIV. We present a spectrum of techniques with different combination of compile-time and runtime overheads. Experiments show the efficiency of our techniques.

In the case of coupled subscripts, we present two methods of address generation which for large block sizes are 100 times faster than the method reported in [8]; for small values of block sizes, these methods show comparable performance. In addition to address generation, the performance of the SPMD node code generated by the compiler depends significantly on the data structures used to encode the derived address tables [4, 8, 9]. In an earlier paper, we [4] discuss issues related to the same and presents extensive results. For the case of coupled subscripts, we show that node code using the *reset* table introduced in [4] outperforms code using the *NextOffset* table [7] by as much as a factor of two.

The paper is organized as follows. Section 2 deals with subscripts with multiple induction variables (MIVs). We discuss a novel way to compute the start element on a processor, and show trade-offs between execution time, and space requirements for data structures used in node code. In Section 3, we discuss address generation and node code for coupled subscripts. Experiments demonstrate the efficacy of our methods. Section 4 concludes with a summary and point to further avenues of research.

2 Multiple Induction Variables

The address generation problem becomes quite complex if the array subscripts contain multiple induction variables (MIV). The challenge now is to generate the access sequence and then the SPMD code for the case when the array A is assumed to be distributed *block-cyclic*. This section solves this problem through the use of efficient runtime techniques. We also address the issue of quickly computing the starting element on each processor for a given global start. This method of finding *start* element can be extended for addressing issues in address generation for the case of one-level mapping. All the methods discussed in this section are implemented and the results obtained prove the efficacy of our methods.

2.1 MIV address generation problem statement

Consider a loopnest that accesses an array A whose subscripts contain multiple induction variable as shown below:

```
!HPF$   PROCESSORS PROCS(p)
!HPF$   DISTRIBUTE A(CYCLIC(k)) ONTO PROCS
        do i1 = 0, n1
              do i2 = 0, n2
                    A(l1 + i1 s1 + l2 + i2 s2) = ···
              enddo
        enddo
```

Here, s_1 and s_2 are the outer and inner loop strides respectively. Consider the case where $k = 4$, $p = 4$, $s_1 = 37$, $s_2 = 2$, and $l_1 = l_2 = 0$. The distribution of the array A onto the processors is shown in Figure 1(a). The elements in shaded square boxes indicate the starting elements for each outer loop iterations whereas the elements in shaded circles indicate the starting elements for different processors for each outer loop iteration. For example, element 37 is a shaded box as it corresponds to the starting element for the second iteration of the outer loop. Elements 41, 45, 49 are in circles as they are the corresponding starting elements on remaining processors due to the second outer loop iteration. The elements accessed in the inner loop for each outer loop iteration which are not starting elements of either kind is shown as non-shaded square box over them. The problem now is to generate the access sequence for this type of array layout on each processor. Figure 1(b) shows the elements accessed due to the given loop on Processor 0; the SPMD code needs to generate the local addresses of these global elements on each processor. Figure 1(c) shows the local addresses which need to be generated by this SPMD code. This section deals with generating this set of elements in lexicographic order using runtime approaches.

2.2 Related work on MIVs

Kennedy, Nedeljkovic and Sethi [8] discuss the issue of address generation for the case of MIV. They view the problem as an integer lattice and use their approach discussed in their earlier work [7] to generate memory gap patterns. Since there is a pattern of repetition both at the outer and inner loop level, they construct memory gap tables at both levels of the loop. For each iteration of the outer loop, the inner loop behaves as a one-level mapping problem and a memory gap table just like the case discussed by Kennedy, Nedeljkovic and Sethi [7] is constructed. However for the outer loop, they present two methods of computing the memory gap table. They use the fact that there could be a maximum of *pk* different global offsets. In the first method, for each outer loop iteration they compute the global memory gap between the global starting locations for that outer loop and the first element in the inner loop accessed on a processor (in the current outer loop iteration). These gaps are stored in a table called (ΔS) which is constructed in $O(pk)$ time.

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

(a): Layout of elements for array A for the case of MIV.

Processor 0	0	2	16	49	51	80	82	113	115	129

(b): Global addresses of elements accessed on Processor 0.

Processor 0	0	2	4	13	15	20	22	29	31	33

(c): Local addresses of elements accessed on Processor 0.

Figure 1: Layout of elements accessed in the case of MIV with $p = 4$, $k = 4$, $s_1 = 37$, $s_2 = 2$, $l_1 = 0$ and $l_2 = 0$

However this starting location gaps gives the global address difference between the outer loop element to the processor starting element. So they need to translate this into the local index space for the processor, which can be expensive. In order to avoid this translation, they present a second method in which instead of storing global starting gaps, they store the local memory gaps between the starting locations corresponding to consecutive outer loop iterations in a table (ΔG) for processor m. In order to construct these tables, they need to find the starting locations on a processor for each outer loop iteration. They propose a constant time algorithm for the case when the inner loop stride is less than or equal to the block size ($s \leq k$) and an $O(pk)$ algorithm for the case when the inner loop stride is greater than the block size ($s > k$). Hence their overall algorithm has a complexity of $O(p^2k^2)$. In the case of (ΔS) method, they incur additional overhead, as they need to do the translation from global to local each time. With the (ΔG) table, they need to determine the global iteration number corresponding to each entry in the table, and they need to compute the local offset.

In order to avoid such expensive conversion and to reduce the complexity, we present an $O(pk)$ method to construct a table for the outer loop and an $O(k)$ solution for enumerating the starting element on each processor when access stride is greater than the block size and an constant time algorithm when the access stride is less than the block size for a given outer loop, thus deriving an $O(pk^2)$ algorithm. This coupled with the fact that we do not incur a global-to-local conversion makes this algorithm very efficient.

Next, we first present our method of computing the starting element on each processor for each outer loop iteration. Once we obtain the starting element on a processor, the enumeration of addresses accessed by the inner loop is nothing but address generation for one-level mapping with stride s_2.

2.3 A new method to find start element on processors

The problem of finding start element should be addressed by the compiler for any type of mappings or distributions. Hence finding an efficient way of computing the start element is critical for any method. This section discusses general methods for computing starting elements on different processors for any given global starting element using efficient runtime techniques for *block-cyclic* distributions. Consider Figure 2 which shows the accessed elements of the array

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

Figure 2: Stages in start computation ($s \le k$) when s = 3, p = 4, k = 4

section $A(9:63:3)$ where A is distributed CYCLIC(4) onto 4 processors. The global start is 9 and lies on processor 2. The elements within boxes are the elements accessed. Hence for Processor 0 we see that elements 18, 33, 48, and 51 are accessed. Element 18, 21, 9, and 12 are the starting elements for processors 0, 1, 2 and 3 respectively. The problem now is to find these starting elements on each processor given a global start. Since each processor has knowledge of only its own elements, we need to find the local address of these starting elements on a per processor basis.

Several researchers have discussed the problem of finding start and hence generating the access sequence for block-cyclic distributions. The method by Chatterjee *et al.* [3] incurs $O(k)$ time by solving k linear Diophantine equations. The solution by Kennedy, Nedeljkovic and Sethi [7] also incurs $O(k)$ overhead for finding the start element. Their method is very similar to the method of Chatterjee *et al.* [3]. However they also present new methods of finding start element [8]. These methods are divided mainly into two cases. If the stride is less than k and if *first* is the first element accessed in an iteration of the outer loop, the global index of the corresponding starting location for processor m is given by the expression $first + \left\lceil \frac{mk - first \bmod pk + pk}{s} \right\rceil s$. However for the case when the stride is greater than the block size, their algorithm has a complexity of $O(pk)$ and suffers from huge runtime overheads.

Our method presented in the next section aims at addressing the issue of start computation. We present a constant time algorithm for determining the start element when $s \le k$ and an $O(k)$ algorithm for $s > k$. The method used in constructing the $O(k)$ algorithm has very little runtime overheads and results in practical times at runtime.

2.3.1 Start computation for $s \le k$

When the stride is less than or equal to the block size, there is at least one element accessed on each course. Hence it becomes easy to find the starting element on each processor. Given a global start and the processor m_g which owns the global start we can determine the local address of the starting elements on all the remaining processors. Let $course_g$ be the course on which the global start exists.

For the processor m_g which owns the global start, the local address of that element will be the local address of the starting location. However for a processor m which is to the left of m_g there are no elements on $course_g$. Hence in order to find the local address of the starting element we use the left basis vector as generated by the one-level algorithm to go to an accessed element on the next course *i.e.,*$coures_g + 1$. From this point onwards we find a factor of s needed to jump into the processor $m's$ space such that we determine the first element accessed in that row. This factor can be determined by just one integer divide operation and hence we can determine the local address of the starting element. For example in Figure 2 we have 9 as the global start and processor m_g = 2. Processors $m = 0$ and $m = 1$ lie to the left of m_g *i.e.,* $m < m_g$. The basis vectors for this lattice as produced by the one-level algorithm [14] are $\vec{l} = (1, -1)$ and $\vec{r} = (0, 3)$. Hence the left vector $\vec{l}$ takes us to the next course ($course_g + 1$) to the element 24. In order to find the starting element for processor 0 we find the factor of s needed to enter the processor space. For the above example factor of 2 is sufficient to get to the element 18 which is the starting element for processor 0. Similarly a factor of 1 is sufficient to get to the element 21 for processor 1. If the processor m is to the right of m_g *i.e.,* when $m > m_g$, then the starting element is guaranteed to lie on $course_g$. We now need to find the factor of s needed to jump to this element. For the above example a factor of 1 is sufficient to get to the element 12 which is the starting element for processor 3. The complexity of the algorithm is $O(1)$.

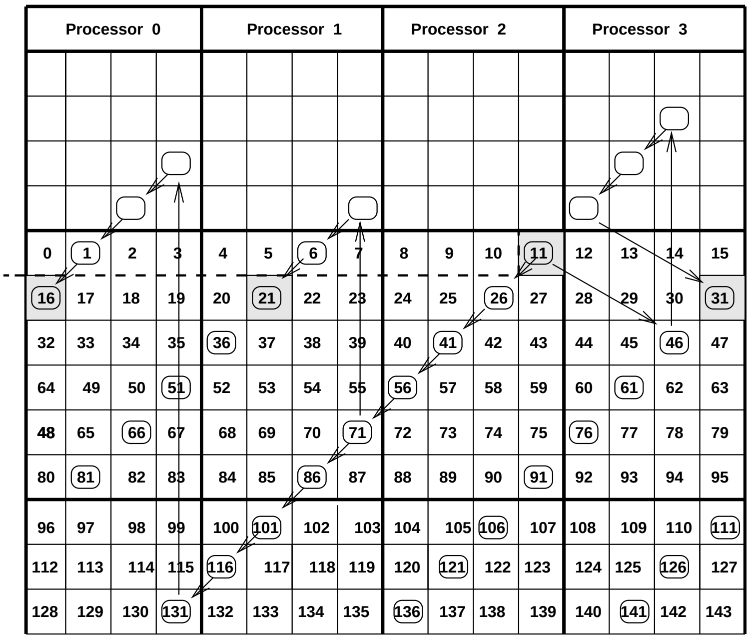

Figure 3: Stages in start computation ($s > k$) when s = 5, p = 4, k = 4, l = 11

2.3.2 Start computation for $s > k$

If the stride is greater than the block size, then there might be courses which might not have any accessed element on them. Hence it is not possible to find the starting element in constant time. Our algorithm presented here uses efficient runtime techniques for finding the start element in $O(k)$ time.

The basis vectors generated by the application of the one-level algorithm yields us vectors $\vec{l}$ and $\vec{r}$ which help us move from one accessed element to the other. Hence given a global start and the processor m_g which owns the start, we use the basis vectors to determine the starting elements on remaining processors. We also make use of the fact that the entire set of accesses repeat after $\frac{s}{\gcd(s,pk)}$ courses.

For a processor m which lies to the left of m_g we first determine the multiple of $\vec{l}$ needed to be able to get into the processor m's space. However it is not guaranteed at this point that we hit an element in the block which holds the starting element. Now we make use of the fact that if $course_g$ is the course on which the global start exists, we can be sure that the starting element will lie within the next $\frac{s}{\gcd(s,pk)}$ courses. So in order to ensure that we enter the block that holds the starting element, we find the factor of $\frac{s}{\gcd(s,pk)}$ needed to jump into this block. Once we enter this block we now can traverse the lattice using the basis vectors until we hit the start element, which is an $O(k)$ operation. Figure 3 shows a case where the stride is greater than the block size. The global start is 11 and $m_g = 2$. The basis vectors for this example are $\vec{l} = (1, -1)$ and $\vec{r} = (2, 3)$. In order to find the start on processor 0 we first find the factor of $\vec{l}$ needed to reach the first element 131 on m. This element lies in the second block of repeating accesses. We then find the factor (2 in this case) needed to jump to the block which contains $course_g$ (0 in this case). We then traverse the lattice from this point until we hit 16 which is the first element accessed after 11 on processor m.

For processor m that lies to the right of m_g, instead of using the left vector we use the right vector in order to get into the processor space. Then we find the factor needed to jump to the block which holds the starting element. Then traversing this block using the basis vectors will yield us the starting element. For the Processor 3 in Figure 3 we first use the right vector to get to element 46. We then jump to the block above the course which holds the global start and then traverse the lattice until we hit the element 31 which is the starting element on the processor.

Since the maximum amount of traversal required in the block is k, this is an $O(k)$ algorithm. This algorithm also does not have huge runtime overheads.

We use this algorithm to determine the starting elements on each processor for each outer loop iteration in the case of MIV. Here we assume that the array has been distributed block-cyclic. The detailed algorithm for block-cyclic for MIV is discussed in the next section.

2.4 Table-based methods for CYCLIC(k) distributed arrays with MIV subscripts

In case of block-cyclic distribution with block size k, referred to as CYCLIC(k), blocks of the template of size k are allocated to the processors in a cyclic fashion. If we need to enumerate elements in lexicographic order we need to enumerate the inner loop for any fixed outer loop iteration. In this section we present efficient runtime techniques to generate the set of accesses. We later contrast our technique with that of Kennedy, Nedeljkovic and Sethi [8].

Several methods which have addressed the issue of address generation for one-level mapping have used the repetitive pattern of accesses. The same pattern can be found in case of MIV for accesses in both inner and outer loops. Just by observing accesses of outer loops, we see that the pattern of accesses is found to repeat itself after $\frac{s_1}{\gcd(s_1,pk)}$ courses, while that of the inner loop alone repeats after $\frac{s_2}{\gcd(s_2,pk)}$ courses. As discussed in Section 2.2 in order to exploit these pattern of accesses we need a table of size at most pk for each processor m which records the local addresses of starting elements for each outer loop iteration. For the inner loop we maintain a pattern table of size at most k, as in the case of single index.

Our approach to the problem of finding memory access sequence for MIV makes use of the repetitive nature of accesses for both inner and outer loops. As mentioned before we record the local addresses of the starting elements for the first pk iterations of the outer loop into a *Start* table of size pk. This is constructed by running the algorithm as discussed in Section 2.3. This construction takes $O(pk^2)$ time as the start computation is of $O(k)$ for each outer loop iteration. The reduced complexity when compared to that of the method by Kennedy, Nedeljkovic and Sethi [8], as well as the faster way of computing start serves to be a big advantage. Further improvements can be incorporated by constructing a table to record the end elements for the same number of iterations. This table called *End*, can be constructed using the same approach as that of *Start* but with a slight modification.

The enumeration of the inner loop can be performed by using a pattern table just like the one discussed in Section 2.2. We use a nextoffset table which keeps a record of the indices needed to lookup the pattern table; the construction of which is similar to the one mentioned by Kennedy, Nedeljkovic and Sethi [8]. The initial index into this nextoffset table is the offset of the starting element. Hence in order to avoid expensive mod operations to compute this offset during runtime, we can pre-compute the same and record this into a table called *Offset* during the start computation stage. The construction of this table does not require any additional work as the information about the offset is already available during start computation.

The execution stage involves using all the above constructed tables, thus avoiding expensive runtime computations. The complete algorithm is as given in Figure 4.

The algorithm in Figure 4 first computes the pattern and nextoffset tables using the one-level algorithm for the inner loop. Lines 1–3 show the steps involved in the construction of these tables. These tables are needed to iterate through the accessed elements in the inner loop. Given that the outer loop pattern starts repeating after $\frac{s_1}{\gcd(s_1,pk)}$ courses, the next step in the algorithm is to compute and store the local starting and ending elements for the first repeating set of the elements in the outer loop iterations. This is done by iterating through each outer loop iteration and computing the starting and ending local addresses on processor m using the method to compute start as discussed in Section 2.3. Lines 5–9 show the steps involved in construction the *Start* and *End* table. The enumeration of the inner loop involves, a entry from the nextoffset table to determine the corresponding index into the pattern table. The index into this nextoffset is the offset of the starting element from the beginning of the course. As mentioned before, we can record and store this offset during the construction of the *Start* table into an *Offset* table. Line 6 shows the construction of *Offset* table along with *Start* computation. This build-up of the table will avoid calculation of the offsets during runtime. As discussed before, the construction of all the above mentioned tables incur $O(k)$ running time. All the above mentioned tables are computed only once before the code enters the execution stage.

During execution, for each outer loop iteration, it needs an entry from the *Start*, *End* and *Offset* table to determine the starting element, starting offset and end element. It then enumerates the elements in the inner loop within the bounds of the start and end element. Lines 11–25 illustrate the execution order followed to generate accessed elements in case of MIV. The while loop in the code, shows the enumeration order for elements within the inner loop. This enumeration involves reading of entries from the nextoffset and pattern tables just like the way it is described in the method by Kennedy, Nedeljkovic and Sethi [8].

For the example in Figure 1, we show the entries in *Start*, *End* and *Offset* tables for Processor 0 in Figures 5(c), (d) and (e) respectively. Let us consider the element 37 due to the second outer loop iteration. The corresponding local start, end and offset elements for Processor 0 are found to be 13, 15, 1 respectively using entries from the *Start*, *End* and *Offset* tables respectively. Once all the required elements are found, the execution stage will enumerate addresses 13, 15 for this outer loop iteration. This involves reading of entries from the nextoffset and deltam table as shown

Input: Layout parameters (p, k), loop limits $(0 : n_1 : s_1)$ for outer loop and $(0 : n_2 : s_2)$ for the inner loop.
Output: Accessed elements
Method:

```
d1 ← gcd(s1, pk)
d2 ← gcd(s2, pk)
(length, ΔM, ΔNextoffset) ← one_level(p, k, s2, m, d2)
for i = 0, pk − 1 do
   (Start[i], Offset[i]) ← start_computation(m, i1 s1, l⃗, r⃗, m_g, d1)
   End[i] ← end_computation(m, i1 s1 + n2 s2, l⃗, r⃗, m_g, d1)
enddo
t ← 0; count ← 0; i ← 0
while (i ≤ n1) do
   jstart ← Start[count] + t
   jend ← End[count] + t
   local_offset ← Offset[count]
   count ← count + 1
   if (count = pk) then
      t ← t + s1/d1
      count ← 0
   endif
   while (jstart ≤ jend) do
      A[j] ← ···
      jstart ← jstart + ΔM[local_offset]
      local_offset ← ΔNextoffset[local_offset]
   enddo
   i ← i + s1
enddo
```

Figure 4: Algorithm to determine the accessed element in case of MIV for CYCLIC(k) distributions

(a)

△ M	2	2	2	2

(b)

△ Nextoffset	2	3	0	1

(c)

Start	0	13	20	29	40	·········

(d)

Offset	0	1	0	11	0	·········

(e)

End	7	15	23	35	43	·········

Figure 5: Data structures used to find the access sequence for the case when the bounds for outer and inner loop are $(0 : n_1 : 37)$ and $(0 : 18 : 2)$ with $p = 4$, $k = 4$, in case of *block-cyclic* distribution for MIV subscripts

in Figures 5(a) and (b). The next step involves determining the start, end and offset element for the third iteration of the outer loop (74). Elements 20, 23, 0 constitute the start, end and offset elements respectively. The execution stage enumerates elements 20,22. This process of execution for each outer loop iteration continues, until we enumerate all the inner loop elements for the complete set of outer loop iterations. The algorithm does not have the overhead of computing the end and start offset for each outer loop iteration.

2.5 Experimental results for MIV

Several variations of the algorithm mentioned in the previous section can be developed based on the problem parameters. If the length of the inner loop is not large enough, building a pattern table and nextoffset table for the inner loop might bring in extra overhead. In this case we can adopt a demand driven approach for computing accessed elements in the inner loop. Similar analogy holds true if the extent of outer loop is not long enough. Here we may resort to computing the start and end element at runtime using the method mentioned in Section 2.3 rather than constructing a pk table. Similarly if both the extent of the inner and outer loop are not large enough we may benefit from not having to construct any tables.

In order to test and verify all the different possible benefits of constructing a table, we implemented different versions of the algorithm mentioned in Figure 4. Below we show the different variations of the MIV algorithms based on the different tables used in the algorithm.

1. Start table, offset table, end table along with the inner loop pattern table are constructed. This method is similar to the one discussed in Figure 4. We term this method as SEOD.

2. Start table, end table, inner loop pattern table are constructed in this variation of our algorithm. The offsets of the starting element are generated at runtime. We term this method as SED.

3. Start table, offset table, inner loop pattern table are constructed. End elements are computed at runtime. We term this method as SOD.

4. We construct only the inner loop pattern tables. The remaining entries are generated at runtime. We term this method as DELTAM.

5. This variation has no tables. The entire set of elements are constructed at runtime. We term this method as NO_TABLES.

All the above mentioned algorithms were implemented and run on a Sun UltraSparc. The number of processors was fixed at 32. All the algorithms were run for the same number of iterations per processor so that we have a base for our comparison. If $numits$ is the number of iterations executed per processor the stretch of the inner loop was chosen to be $s_2 * \left(\sqrt{numits * p} - 1\right)$. Our value for s_1 was based on the value chosen for s_2 and was equal to $s_2 * \sqrt{numits * p} + 1$. The stretch of the outer loop was then chosen to be $s_1 * \left(\sqrt{numits * p} - 1\right)$. We then varied the value of the block-size and recorded the time taken for the execution of the node-code for all the different variations of the algorithms. We also contrast our methods of solving address generation for MIV, to that of the method by Kennedy, Nedeljkovic and Sethi [8] which we term as RICE in our results. Their method involves construction of a start table which holds the local memory gaps between successive outer loop iterations. They compute the offsets and the end element at runtime. They however use the same inner loop structure as our method, by preserving a pattern and nextoffset table.

Tables 1, and 2 show the times taken for the various algorithms for different block-sizes and inner loop strides s_2, when the iterations per processor was fixed at 2500 and 5000 respectively. These results demonstrate that the SEOD method perform very efficiently in almost all the cases as expected. This is due to the benefit we get by constructing all the useful pattern tables, thereby eliminating the associated runtime overheads. The method by RICE incurs high runtime overheads and hence is slower than the other methods. The times from SOD and SED methods lies in between the times of SEOD and RICE. The DELTAM and NO_TABLES methods always perform badly, due to the expensive conditionals and runtime overheads.

Table 1: Execution times (in μs) for MIV algorithms on UltraSparc: $p = 32$, No. of iterations = 2500

k	s_2	SEOD	SED	SOD	RICE	DELTAM	NO − TABLES
4	3	576	568	1110	1209	1857	1849
	5	485	603	1071	1207	2336	3464
	17	437	571	1100	1220	2316	2631
	25	466	592	1132	1249	2320	9667
8	3	583	745	1149	1202	1931	2331
	5	444	575	1097	1204	1814	2159
	17	452	584	1081	1225	2400	2617
	25	479	660	1086	1238	2507	5968
16	3	445	752	1176	1193	1923	1838
	5	454	573	1045	1209	1813	1774
	17	487	579	1094	1214	2493	2475
	25	484	605	1127	1238	2736	4319
32	3	468	571	1046	1193	1922	2326
	5	445	572	1096	1197	1786	1901
	17	559	582	1089	1207	1808	1902
	25	479	668	1072	1232	1830	2440
64	3	451	608	1056	1183	1853	1855
	5	493	568	1037	1186	1787	2969
	17	579	694	1211	1201	1808	2494
	25	491	602	1079	1231	1830	1931
128	3	467	584	1049	1197	1825	2322
	5	467	628	1049	1232	1810	2162
	17	471	682	1227	1211	1805	1922
	25	553	620	1090	1232	1832	2577
256	3	480	593	1058	1160	1810	1845
	5	467	583	1065	1153	1797	1774
	17	470	622	1054	1198	1798	3200
	25	509	618	1081	1226	1832	1987

3 Coupled Subscripts

Two new table construction techniques have been proposed in this paper for references to arrays with CYCLIC(k_1, k_2) distribution that contain coupled subscripts. Fast table construction is essential for the efficient execution of programs. The problem is simple when the multidimensional arrays have independent subscripts. In this case multiple application of the algorithm for the one-dimensional case by [3] solves the problem. However when the same loop index appears in two different dimensions, the problem is more complex [1, 8, 11]. Table construction techniques form an important part of address generation for arbitrary affine subscripts.

3.1 Coupled subscripts: Problem statement and related work

Consider an example of a loop with coupled subscripts shown below.

```
!HPF$   PROCESSORS PROCS(p1, p2)
!HPF$   DISTRIBUTE A(CYCLIC(k1),CYCLIC(k2)) ONTO PROCS
        do j = 0, n1
             A(l1 + j s1, l2 + j s2) = ...
        enddo
```

Here, array A is distributed onto $p_1 \times p_2$ processors using CYCLIC(k_1) distribution along processor dimension one, and using CYCLIC(k_2) distribution along processor dimension two. The access strides for the loops are s_1 and s_2 respectively. As the above example demonstrates the subscript expressions $l_1 + js_1$ and $l_2 + js_2$ both contain the loop index j; we say that the two subscripts expressions are coupled by the loop index j. Let us compute the set of iterations for each dimension independently. Then, the memory addresses referenced on a processor correspond to

Table 2: Execution times (in μs) for MIV algorithms on UltraSparc: $p = 32$, No. of iterations = 5000

k	s_2	SEOD	SED	SOD	RICE	DELTAM	NO − TABLES
4	3	860	1089	1772	1840	2784	2904
	5	860	1046	1755	1853	3535	5881
	17	990	1159	1827	1901	3616	5326
	25	1070	1271	1980	2001	3719	19396
8	3	916	1082	1733	1829	2793	3885
	5	872	1128	1770	1839	2792	3565
	17	1104	1290	1986	2015	3830	5147
	25	1107	1234	2056	1979	4044	11695
16	3	876	1066	1755	1827	2772	2937
	5	955	1057	1757	1827	2813	2770
	17	1095	1280	2011	1983	3939	4027
	25	1129	1339	2104	2044	4385	8698
32	3	917	1084	1770	1807	2758	3873
	5	939	1061	1779	1806	2791	3019
	17	1120	1280	2043	1994	3099	3375
	25	1283	1356	1999	2095	3087	6739
64	3	930	1053	1739	1797	2771	2918
	5	917	1077	1743	1799	2772	5147
	17	1154	1332	2007	2024	3017	4930
	25	1290	1349	2038	2104	3059	3876
128	3	916	1065	1763	1793	2773	3892
	5	1129	1286	1895	1804	2772	3568
	17	1171	1368	1964	2008	2959	3633
	25	1225	1355	2021	2029	2931	6974
256	3	932	1073	1769	1759	2752	2932
	5	1093	1278	1897	1781	2772	2759
	17	1179	1336	2040	2037	3009	5951
	25	1266	1347	2061	2128	3114	3938

iterations common to both the dimensions. This cannot be deduced simply by two applications of the algorithm in [3]; we need to expand the tables up to the least common multiple of the periods (in terms of iterations) of the two dimensions. This forms the basis of our table generation techniques.

The Rice method [8] starts by computing the starting location on each processor. The general solutions for the iteration on each dimension is computed by solving the set of linear diophantine equations. For an array with CYCLIC(k_1, k_2) distribution accessed with strides s_1 and s_2, the method requires solving $k_1 \times k_2$ equations, solutions to which give common iterations. The common iterations are not in increasing order in general. Thus, one needs to use a sorting algorithm; this results in an overall complexity of $O(k_1 k_2 \log(k_1 k_2))$. While the theoretical complexity of the algorithm is low, implementations require expensive integer arithmetic. The table of iterations that is obtained is used to compute the set of global addresses referenced on the processor. A global-to-local address conversion is performed for the desired sequence of local addresses. The table obtained from the above process is used to generate the SPMD code. The SPMD code uses either a single two-dimensional table or two one-dimensional tables.

The method presented in this paper attempts to minimize the table generation overhead by avoiding solving for $k_1 \times k_2$ equations and the costly sorting method involved. The two method proposed by us in this paper are the *search* and the *hashing* methods. Both these are practically efficient methods to enumerate the common iterations. In addition, our methods do not incur global to local conversion in the address computation. These two methods are discussed in detail in the following paragraphs.

3.2 Search method for common iteration computation

The algorithm for computing common iterations using the *search* method is shown in Figure 6 The first phase of the algorithm is the computation of the ΔM_1, ΔM_2, $\Delta iter_1$, $\Delta iter_2$ tables. The ΔM_1 and ΔM_2 are the same tables constructed using algorithms in [7] and [14]. The iteration gap tables are two additional tables that are constructed.

Input: Compute $\Delta M_1, \Delta M_2, \Delta iter_1, \Delta iter_2, length_1, length_2$
Output: $reppat$ table (consisting of the set of all local addresses), and its length len
Method:

1 $d_1 \leftarrow \gcd(s_1, p_1k_1)$
2 $d_2 \leftarrow \gcd(s_2, p_2k_2)$
3 $rep \leftarrow \text{lcm}\,(\frac{p_1k_1}{d_1}, \frac{p_2k_2}{d_2})$
4 $rlen_1 \leftarrow length_1 \times \frac{rep*d_1}{p_1k_1};\ rlen_2 \leftarrow length_2 \times \frac{rep*d_2}{p_2k_2}$
5 Compute $start\_it_1$ and $start\_it_2$ on processor
6 $its\_rep_1 \leftarrow start\_it_1;\ its\_rep_2 \leftarrow start\_it_2;\ i \leftarrow 0;$
7 **while** $(i < rlen_1)$ **do**
8   **for** $k \leftarrow 0$ **to** $length_1 - 1$ **do**
9       $repits_1[i] \leftarrow \Delta iter_1[k] + its\_rep_1$
10   **enddo**
11    $its\_rep_1 \leftarrow its\_rep_1 + \frac{p_1k_1}{d_1}$
12    $i \leftarrow i + length_1$
13 **enddo**
14 Compute the replicated table $repits_2$ in the same manner as above
15 Choose the smaller of the table. Assume $repits_1$ is smaller. If $repits_2$ is smaller, the code is symmetric
16 $j \leftarrow 0;\ len \leftarrow 0$
17 **for** $i \leftarrow 0$ **to** $rlen_1 - 1$ **do**
18    $ptr \leftarrow j;\ temp \leftarrow repits_1[i]$
19   **for** $j \leftarrow ptr$ **to** $rlen_2 - 1$ **do**
20   **if** $(temp \leq repits_2[j])$ **then**
21   **if** $(temp = repits_2[j])$ **then**
22             $repcomits[len] \leftarrow temp$
23             $len \leftarrow len + 1$
24   Continue with next iteration i
25   **endif**
26   **endif**
27   **enddo**
28 **enddo**

Figure 6: Algorithm to compute the common iterations table using search

These tables are the iteration gaps between two successive address references on the same processor. There construction is simple and needs minor modification of the code when constructing the ΔM_1 and ΔM_2 tables. In fact similar to the $\vec{r} = (r_1, r_2)$ and $\vec{\ell} = (\ell_1, \ell_2)$ vectors that are formed for the memory gaps, three values may be formed for the iteration gaps; we will call these the *right_iteration_gap, left_iteration_gap* and the *right_left_iteration_gap.* The modifications that are needed in the code that computes the ΔM_1 table are as follows:

- right_iteration_gap $= \frac{r_1*pk+r_2}{s}$
- left_iteration_gap $= \frac{l_1*pk+l_2}{s}$
- right_left_iteration_gap = right_iteration_gap + left_iteration_gap

Phase 2 (Lines 3 and 4) of the algorithm computes the replication length of the table. This length defines the iteration number after which the iteration gap start repeating. Since in the single dimension case the iterations repeat after $\frac{p_1k_1}{d_1}$ iterations it explains the expression stated in the algorithm. Phase 3 of the algorithm uses this replication length to find the replication factor of each table. This table provides the factor by which each of the table has to be replicated. So the new lengths are $rlen_1$ and $rlen_2$. Lines 5–14 accomplish replication of the tables. Next, the algorithm chooses the smaller of the tables and increments it step wise. Each iteration of the smaller table is compared with the bigger table and if the two iterations match then the value of the iteration is stored in a new table called the $repcomits$ table. This table will then be used for generation of SPMD code and its subsequent execution. The search method employed in the above algorithm is a simple search in which the sorted sequence of the table is taken advantage of. The point of last search in the bigger of the two tables is saved and it is from this point that the search is continued when iterating the next time over the smaller table. A lot of other efficient searching methods can be applied to this algorithm but the

Input: Compute $\Delta M_1, \Delta M_2, \Delta iter_1, \Delta iter_2, length1, length2$
Output: $reppat$ table (consisting of the set of all 2-D local addresses), and its length, len
Method:
1 $d_1 \leftarrow \gcd(s_1, p_1k_1)$
2 $d_2 \leftarrow \gcd(s_2, p_2k_2)$
3 $rep \leftarrow \mathrm{lcm}\,(\frac{p_1k_1}{d_1}, \frac{p_2k_2}{d_2})$
4 $rlen_1 \leftarrow length_1 \times \frac{rep*d_1}{p_1k_1}$; $rlen_2 \leftarrow length_2 \times \frac{rep*d_2}{p_2k_2}$
5 Compute $start\_it_1$ and $start\_it_2$ on processor
6 $its\_rep_1 \leftarrow start\_it_1; its\_rep_2 \leftarrow start\_it_2$
7 Initialize hashtable[] to 0; $i \leftarrow 0$
8 **while** $(i < rlen_1)$ **do**
9    $j \leftarrow i$
10   **for** $k \leftarrow 0$ **to** $length_1 - 1$ **do**
11       $temp \leftarrow \Delta iter_1[k] + its\_rep_1$
12   hashtable$[temp] \leftarrow 1$
13       $j \leftarrow j + 1$
14   **enddo**
15    $its\_rep_1 \leftarrow its\_rep_1 + \frac{p_1k_1}{d_1}$
16    $i \leftarrow i + length_1$
17 **enddo**
18 Compute the replicated table $repits_2$ in the same manner as above
19 $i \leftarrow 0$; $len \leftarrow 0$
20 **while** $(i < rlen_2)$ **do**
21   **for** $k \leftarrow 0$ **to** $length_2 - 1$ **do**
22   temp $\leftarrow its\_rep_2 + \Delta iter_2[k]$
23   **if** (hashtable[temp] $= 1$) **then**
24   repcomits[len] $\leftarrow$ temp
25   len $\leftarrow$ len $+1$
26   **endif**
27       $its\_rep_2 \leftarrow its\_rep_2 + \frac{p_2k_2}{d_2}$
28       $i \leftarrow i + length_2$
29   **enddo**
30 **enddo**

Figure 7: Algorithm to compute the common iterations table using hashing

basic idea remains the same. The use of varied search methods and their combination thereof for table construction for another related problem of MIV can be viewed in the previous section on MIV in this paper and [16]. The same search methods can be used here also. The two tables which are computed are analogous in the sense one consists of the iterations while the other consists of the memory references. The advantages of using such a method for table construction is that there is no need to solve k_1k_2 diophantine equations and sorting is avoided. While theoretically the search method is inferior to the Rice method, it performs extremely well in practice.

3.3 Hashing method for common iteration enumeration

The hashing algorithm is shown in Figure 7; this is a simple and effective algorithm. Lines 1–6 of the algorithm are the same as the first six lines of the algorithm for searching. Line 7 in the algorithm creates and initializes a hashtable of size rep. This is the size of the table before the pattern starts repeating. Lines 8–17 show that the iteration values in the first dimension are read one by one from the iteration gap table. The iteration values are used to hash into the hashtable. So the temporary array consists of a series of zeros and ones as its entries. Next the iterations in the second dimension are considered. The iterations are again used to index into the temporary array. If at the indices in the temporary array the value is one then the index value is the common iteration in the two dimension and is stored in the $repcomits$ table. The algorithm returns the repcomits table which consists of all the common iteration in the two dimensions. The algorithm also returns the $reppat$ table which consists of the set of local addresses accessed on that processor. The length of these table are also returned. The method is called hashing because of the fact that the iteration values are the values which are used to index into the temporary array. Any hashing function could be used but this is the simplest function that can be used.

The advantage of this code is its simplicity. The code in comparison with the *search* code involves less conditionals and hence should be more time-effective. This fact is borne out from the experimental results that are presented at the end of the paper. The disadvantages are the same as the *searching* method, namely, that of space overhead and theoretical time complexity. This section describes the two new methods of table construction. This table is used during the SPMD node code execution.

3.4 Coupled subscripts: Execution strategies

The table of common iterations obtained from the above table construction mechanisms can be used to produce efficient SPMD node code for execution on each processor. The SPMD node code provided in this section will use extensively the ideas presented in an earlier paper [4]. The two methods discussed here are the ones using the conditional statements, the next one uses the concept of a *reset* table. Both these as well as the technique from [8] are described next.

3.4.1 SPMD node code using the next-offset table (Rice)

The SPMD node code that is executed on each processor is presented below:

Input: $\Delta_2 M$

Compute $start_1, start_2, end_1, end_2$
Compute a two-dimensional nextoffset table
$a_1 \leftarrow start_1$
$a_2 \leftarrow start_2$
$o_1 \leftarrow start_1$ mod k_1
$o_2 \leftarrow start_2$ mod k_2
while $(a_1 \leq end_1) and (a_2 \leq end_2)$ **do**
  $array[a_1, a_2] \leftarrow \cdots$
  $(a_1, a_2) \leftarrow (a_1, a_2) + \Delta_2 M[o_1, o_2]$
  $(a_1, a_2) \leftarrow$ nextoffset$[o_1, o_2]$
enddo

3.4.2 SPMD node code using conditionals

The SPMD node code that is executed on each processor is presented below:

Input: ΔM

len $\leftarrow length(\Delta M)$
Compute $start_1, start_2, end_1, end_2$
$a_1 \leftarrow start_1$
$a_2 \leftarrow start_2$
$k \leftarrow 0$
while $(a_1 \leq end_1) and (a_2 \leq end_2)$ **do**
  $array[a_1, a_2] \leftarrow \cdots$
  $(a_1, a_2) \leftarrow (a_1, a_2) + \Delta M[k]$
  $k \leftarrow k + 1$
  if $(k = len)$ **then** $k \leftarrow 0$
  endif
enddo

3.4.3 SPMD node code using the reset table

The SPMD node code that is executed on each processor is presented below:

Input: ΔM

```
len ← length(ΔM)
for i ← 0 to len − 2 do
  reset[i] ← i + 1
enddo
reset[len − 1] ← 0
Compute start_1, start_2, end_1, end_2
a_1 ← start_1
a_2 ← start_2
k ← 0
while (a_1 ≤ end_1)and(a_2 ≤ end_2) do
  array[a_1, a_2] ← ···
  (a_1, a_2) ← (a_1, a_2) + ΔM[k]
  k ← reset[k]
enddo
```

3.5 Experimental results for coupled subscripts

All experiments were done on a Sun UltraSparc. For table generation, we used a fixed processor geometry of 8×4. Tables 3–5 show table generation times for the Rice method and the two methods presented in this paper, for several values of k_1 and k_2 for three different combinations of strides. It is clear that for small values of k_1 and k_2, the three methods are comparable. But, as the block sizes increase, the hash and search methods significantly outperform the Rice method by as much as a factor of 100.

Tables 6–8 show the execution times using the NextOffset method (called **Rice** in the tables) [8], the method (Section 3.4.2) using increment and test (called **Conditional** in the tables), and the reset method (Section 3.4.3). As observed by earlier experiments in [4, 8, 9], the Rice method is usually better than the conditional method; but, the *reset* table outperforms the other methods consistently, sometimes by as much as a factor of two in execution time. In addition, it is noticeable that Rice method of address generation uses up to 25% of the execution overhead for 10,000 iterations.

4 Summary

In this paper, we have shown several techniques to generate accessed elements in an array, when the array subscripts contain multiple induction variables and coupled subscripts. Our techniques involve building tables for exploiting the pattern of accesses. In the case of MIVs, we have presented a new general way of computing start element on a processor. This method does not incur much runtime overheads, and can serve to be big advantage in order to generate starting elements on each processor for a given global start. The complexity of this method is $O(k)$ when the block size is greater than the stride, and constant time otherwise. We use this technique of computing start to present an $O(pk^2)$ algorithm to compute accessed elements in case of MIV; the algorithm in [8] is $O(p^2k^2)$. The reduced complexity and benefit of constructing several pattern tables proves to be beneficial during execution for MIV. In the case of coupled subscripts, we presented two methods of address generation which outperform an existing technique [8] by as much as a factor of 100 for large block sizes, while demonstrating comparable performance for small block sizes. We also show that the performance of the node code using the *reset* table introduced by us in [4] is consistently better than other methods. Work is in progress for communication generation for complex subscripts, and in address and communication generation for general affine subscripts.

Acknowledgments

We gratefully acknowledge the support of the National Science Foundation through an NSF Young Investigator Award (CCR-9457768), an NSF grant (CCR-9210422) and an award from the Haliburton Foundation.

Table 3: Comparison of table construction times (in μsec). Processor geometry $= 8 \times 4$; $s_1 = 1$; $s_2 = 1$

k_1	k_2	**Rice**	**Search**	**Hashing**
4	4	29.9	26.8	19.7
4	8	35.9	24.1	20.2
8	4	45.9	22.2	21.1
8	8	71.9	32.8	23.6
16	4	54.2	27.8	24.9
16	8	93.5	32.5	32.8
8	16	93.0	25.3	24.8
16	16	178.6	30.7	26.9
32	4	95.2	40.5	32.4
32	8	148.9	37.2	34.1
8	32	147.6	32.0	30.4
32	32	542.9	45.9	46.0

k_1	k_2	**Rice**	**Search**	**Hashing**
64	4	189.1	58.2	47.1
64	8	278.7	56.7	48.4
8	64	252.4	46.0	42.2
64	64	1879.5	77.5	65.5
128	4	362.4	100.3	91.4
128	8	581.6	93.8	110.0
8	128	500.0	75.4	65.7
128	128	6952.6	146.7	116.5
256	4	754.3	179.9	186.2
256	8	1154.1	172.1	177.4
8	256	1023.9	126.3	115.8
256	256	27029.3	253.4	241.6

Table 4: Comparison of table construction times (in μsec). Processor geometry $= 8 \times 4$; $s_1 = 5$; $s_2 = 3$

k_1	k_2	**Rice**	**Search**	**Hashing**
4	4	23.4	24.6	22.6
4	8	30.7	30.9	30.8
8	4	34.6	32.1	31.8
8	8	58.5	28.7	31.4
16	4	63.8	34.5	31.3
16	8	86.8	36.8	33.5
8	16	72.7	34.6	29.1
16	16	140.2	41.0	33.8
32	4	105.2	43.7	35.1
32	8	158.1	43.0	37.8
8	32	134.7	38.0	36.0
32	32	463.0	57.5	45.1

k_1	k_2	**Rice**	**Search**	**Hashing**
64	4	210.7	65.0	54.2
64	8	310.1	67.3	54.8
8	64	269.7	52.4	48.0
64	64	1842.6	82.1	76.2
128	4	410.2	106.9	95.9
128	8	612.5	103.5	121.2
8	128	526.6	80.0	73.0
128	128	6858.9	136.3	118.5
256	4	872.6	202.7	201.8
256	8	1300.7	196.3	195.2
8	256	1068.5	138.8	130.8
256	256	27765.3	258.6	254.8

Table 5: Comparison of table construction times (in μsec). Processor geometry $= 8 \times 4$; $s_1 = 2$; $s_2 = 7$

k_1	k_2	**Rice**	**Search**	**Hashing**
4	4	13.5	24.4	20.9
4	8	24.3	26.1	24.1
8	4	27.4	23.6	22.2
8	8	35.0	25.8	32.6
16	4	37.1	28.8	24.3
16	8	49.5	29.3	27.9
8	16	47.0	27.8	25.8
16	16	71.0	32.6	30.9
32	4	54.0	30.1	29.3
32	8	82.0	33.5	31.5
8	32	80.0	35.9	33.1
32	32	237.9	39.4	37.2

k_1	k_2	**Rice**	**Search**	**Hashing**
64	4	95.7	43.9	36.0
64	8	146.7	52.3	40.1
8	64	154.2	53.1	46.8
64	64	871.9	58.0	54.6
128	4	183.4	61.2	50.6
128	8	286.3	61.1	55.0
8	128	311.4	79.0	66.8
128	128	3355.8	90.0	83.2
256	4	365.8	111.5	100.7
256	8	568.2	102.6	99.3
8	256	617.6	138.8	122.7
256	256	13736.6	166.1	154.2

Table 6: Execution overhead (in μsec) for 10,000 iterations per processor; Processor geometry $= 8 \times 4$; $s_1 = 1$; $s_2 = 1$

k_1	k_2	**Rice**	**Conditional**	**Reset**
4	4	587.0	817.7	402.2
4	8	606.6	812.1	427.2
8	4	609.7	805.7	423.4
8	8	657.8	875.8	470.4
16	4	662.2	857.2	469.4
16	8	736.5	951.1	548.0
8	16	735.8	948.6	552.2
16	16	928.0	1155.0	735.5
32	4	1499.4	978.5	566.1
32	8	912.0	1107.5	703.7
8	32	880.2	1098.5	696.1
32	32	1891.8	2111.7	1744.3

k_1	k_2	**Rice**	**Conditional**	**Reset**
64	4	1690.0	1141.7	752.6
64	8	2006.3	1477.9	1036.0
8	64	1226.1	1445.0	1030.3
64	64	5718.2	6029.4	5654.6
128	4	2071.8	1519.7	1354.5
128	8	2664.8	2162.3	1960.2
8	128	1862.7	2080.2	1643.9
128	128	22368.5	21013.8	20718.3
256	4	3229.9	2262.9	1909.9
256	8	4091.2	3490.5	3156.8
8	256	3231.5	3383.8	2964.4
256	256	83481.4	81516.2	81189.8

Table 7: Execution overhead (in μsec) for 10,000 iterations per processor; Processor geometry $= 8 \times 4$; $s_1 = 5$; $s_2 = 3$

k_1	k_2	**Rice**	**Conditional**	**Reset**
4	4	591.9	758.9	400.2
4	8	609.1	770.2	419.6
8	4	610.1	794.5	423.1
8	8	649.4	835.8	455.6
16	4	652.0	874.0	465.2
16	8	736.1	965.6	540.4
8	16	717.0	908.0	540.1
16	16	873.9	1122.0	700.2
32	4	1499.2	983.4	576.6
32	8	890.2	1136.7	704.1
8	32	890.3	1082.5	697.3
32	32	1845.0	2032.0	1668.5

k_1	k_2	**Rice**	**Conditional**	**Reset**
64	4	1699.3	1151.8	768.2
64	8	2010.5	1511.7	1099.7
8	64	1229.8	1448.0	1039.4
64	64	5716.1	5847.5	5541.9
128	4	2098.9	1585.2	1374.0
128	8	2733.8	2213.6	1989.0
8	128	1872.0	2074.8	1687.1
128	128	20908.7	20991.8	20651.5
256	4	5800.2	2305.8	1937.5
256	8	6660.3	3579.4	3242.0
8	256	3254.3	3413.2	3072.2
256	256	85578.1	82371.3	82022.2

Table 8: Execution overhead (in μsec) for 10,000 iterations per processor; Processor geometry $= 8 \times 4$; $s_1 = 2$; $s_2 = 7$

k_1	k_2	**Rice**	**Conditional**	**Reset**
4	4	572.4	409.4	387.5
4	8	592.7	759.8	411.6
8	4	1042.5	759.6	405.5
8	8	654.2	787.2	439.5
16	4	1093.6	812.0	442.4
16	8	684.5	866.5	490.6
8	16	678.9	865.5	488.1
16	16	806.5	970.0	622.6
32	4	930.6	885.9	509.6
32	8	817.3	1030.2	608.6
8	32	809.9	1028.6	602.2
32	32	1486.5	1662.7	1275.2

k_1	k_2	**Rice**	**Conditional**	**Reset**
64	4	1573.0	1023.1	639.5
64	8	1364.2	1274.3	848.0
8	64	1049.6	1240.7	858.0
64	64	4232.7	4410.1	4080.5
128	4	1845.6	1333.4	881.0
128	8	2294.9	1788.7	1366.5
8	128	1532.1	1762.2	1336.5
128	128	15205.0	15255.4	15037.3
256	4	2469.3	1815.5	1532.5
256	8	3684.1	2764.5	2432.5
8	256	2529.6	2779.7	2329.1
256	256	59389.9	59275.1	58948.7

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

[4] S. Dutta and J. Ramanujam. Data structures for efficient execution of programs with block-cyclic distributions. Technical Report TR-96-11-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, Nov. 1996. Preliminary version presented at the *6th Workshop on Compilers for Parallel Computers,* Germany, Dec. 1996.

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

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

[7] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proc. of Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[8] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions In *Proc. ACM International Conference on Supercomputing,* Madrid, Spain, July 1995.

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

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

[11] S. Midkiff. Comuting the local iteration set of a block-cyclically distributed reference with affine subscripts. In *Proc. 6th Workshop on Compilers for Parallel Computers,* Aachen, Germany, December 1996.

[12] 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.

[13] E. Su, A. Lain, S. Ramaswamy, D.J. Palermo, E.W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proc. 1995 ACM International Conference on Supercomputing,* Barcelona, Spain, July 1995.

[14] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs Using Integer Lattices. In *Languages and Compilers for Parallel Computing,* C.-H. Huang *et al.* (Editors), Lecture Notes in Computer Science, Vol. 1033, pages 191–208, Springer-Verlag, 1996.

[15] A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data-parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing,* 38(2):188–203, November 1996.

[16] A. Venkatachar. Efficient address and communication generation for data-parallel programs. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, December 1996.

[17] L. Wang, J. Stichnoth, and S. Chatterjee. Runtime performance of parallel array assignment: An empiricial study. In *Proc. Supercomputing 96,* November 1996.