Efficient Address Generation for Block-Cyclic Distributions

Ken Kennedy
Nenad Nedeljković
Ajay Sethi

CRPC-TR94497-S
December, 1994

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

Submitted to the 9th ACM International Conference on Supercomputing (ICS '95).

Efficient Address Generation for Block-Cyclic Distributions*

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 make programming of distributed-memory machines easier, and resulting programs more portable and efficient. Advanced features of these languages require new methods in both compilers and run-time systems. We present efficient techniques for generating local memory addresses, in the exact order as specified by the original program, for computations involving references to arrays with *cyclic*(k) distribution, the most general regular data distribution provided in data-parallel languages. Our method exploits the repetitive pattern of memory accesses to handle arbitrary affine subscripts, while minimizing the space and time overhead. Extensive experimental results indicate the efficiency of our approach in practice.

1 Introduction

Distributed-memory machines are widely regarded as the most promising means for high performance computing. However, the message-passing programming model, typically associated with these machines, makes it difficult to take full advantage of parallel computing power. This has resulted in the development of data-parallel languages, such as Fortran D [9], Vienna Fortran [2], and most recently High Performance Fortran (HPF) [7, 13]. These languages provide familiar single address space and data mapping directives that allow the programmer to specify how array data should be distributed across processors. A compiler then uses these directives to partition the computation and generate SPMD (Single Program Multiple Data) code to be executed by each processor.

Compilation of programs that access arrays with *block* or *cyclic* distribution has been studied extensively [5, 12, 15]. A more general regular distribution is *block-cyclic* distribution (*cyclic*(k) in HPF), in which an array is first divided into blocks of size k, and then these blocks are assigned to processors in a cyclic fashion. An example of this distribution is shown in Figure 1 (we assume that array elements are numbered starting from 0). The generality of *cyclic*(k) distribution poses a challenging problem of address computation for array references. As pointed out by Knies et al., if full address generation were to be performed for each access to an array with *cyclic*(k) distribution, the resulting overhead would be unacceptable [11]. Therefore, there is a strong need for compiler and run-time techniques that would minimize the

*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.

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 1 Layout of array elements distributed with *cyclic*(4) distribution over 4 processors.

cost of generating memory addresses that are accessed while performing a computation over block-cyclically distributed arrays.

Several efforts to address some of the difficulties in compiling programs with *cyclic*(k) distribution have been described in the literature. Ancourt et al. use a linear algebra framework for compiling *independent* loops in HPF [1]. Although they can handle arbitrary affine array subscripts, the generated loop bounds and local array subscripts can be quite complex, and thus introduce a significant overhead. Furthermore, the assumption of independent parallelism allows them to enumerate loop iterations in any order, which is, in general, not always possible. Gupta et al. address the problem of array statements 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. 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 the run-time address resolution. Stichnoth et al. use intersections of array slices for communication generation [14]. Their approach is similar to, and has the same drawback as, the virtual-cyclic scheme mentioned above. The method described by Chatterjee et al. is based on exploiting the repetitive pattern of memory accesses while traversing a regular section of an array with *cyclic*(k) distribution [3]. They show how each processor can generate the correct sequence of its local memory accesses using lookups into a table that has at most k entries, and present a table construction algorithm that takes roughly $O(k \log k)$ time.

In our previous work [10], we described a linear-time algorithm for constructing the table needed to generate local memory accesses, and presented experimental results to assert the practical efficiency of our method. In this paper, we show how the ideas used in developing the algorithm can be applied to resolve the time versus space tradeoff present in the table lookup technique. Furthermore, we extend the address generation method based on the table lookup to handle array references with arbitrary affine subscripts. The efficiency of the proposed schemes is demonstrated through a series of experimental results.

The rest of this paper is organized as follows. In Section 2 we briefly review our linear-time algorithm for generating tables for simple regular sections and describe how the same ideas can be used to eliminate memory overhead with minimal penalty in the execution time. In Section 3 we show how array references

with single subscripts containing multiple loop induction variables can be handled by a simple extension to a table lookup based address generation. Our method for dealing with multiple coupled subscripts (two or more subscripts containing the same induction variable) and generalization to arbitrary affine subscripts are presented in Section 4. Finally, we conclude in Section 5 by summarizing our contributions and indicating directions for future research.

2 Simple Regular Sections

We first describe how the memory access sequence is generated for simple regular sections, which correspond to array references with subscripts containing a single loop induction variable (SIV subscripts). Given an array A distributed with $cyclic(k)$ distribution over p processors and the loop

do $i = l, u, s$
  $A(i) = 100.0$
enddo

the problem is to find the sequence of local memory locations that a given processor m must access while performing its share of computation. This is equivalent to finding the starting location for processor m and the gaps between every two memory locations corresponding to successive accesses of array elements that belong to processor m. Since the offset of an array section element within its block uniquely determines the offset of the next array element accessed by the same processor, and since the offsets range between 0 and $k-1$, the sequence of offsets must have a cycle whose length is at most k. The cycle of offsets induces the cycle in the sequence of local memory gaps, which together with the processor's starting location is sufficient to specify the complete memory access sequence.

Chatterjee et al. [3] have shown that both the starting location and the table of local memory gaps can be found by solving a set of k linear Diophantine equations

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

For each solvable equation (i.e., whenever $\text{GCD}(s, pk)$ divides i), they find the smallest nonnegative solution for j. The minimum of all these solutions gives the first array element $A(l + js)$ accessed by processor m, while the sorted set of the solutions represents the initial cycle in the memory access sequence, which is then used to construct the memory gap table.

The complexity of the method described by Chatterjee et al. is $O(k \log k + \min(\log s, \log p))$. We have developed an improved table-construction algorithm that avoids sorting of the initial sequence and achieves $O(k + \min(\log s, \log p))$ running time [10]. Array elements are treated as points in $\mathcal{Z}^2$ with the y-coordinate corresponding to the number of the row to which the array element belongs, and the x-coordinate corresponding to its offset within that row. Using the fact that the accessed array elements form an integer lattice, we have shown how to choose a lattice basis that allows for simple and fast enumeration of array accesses.

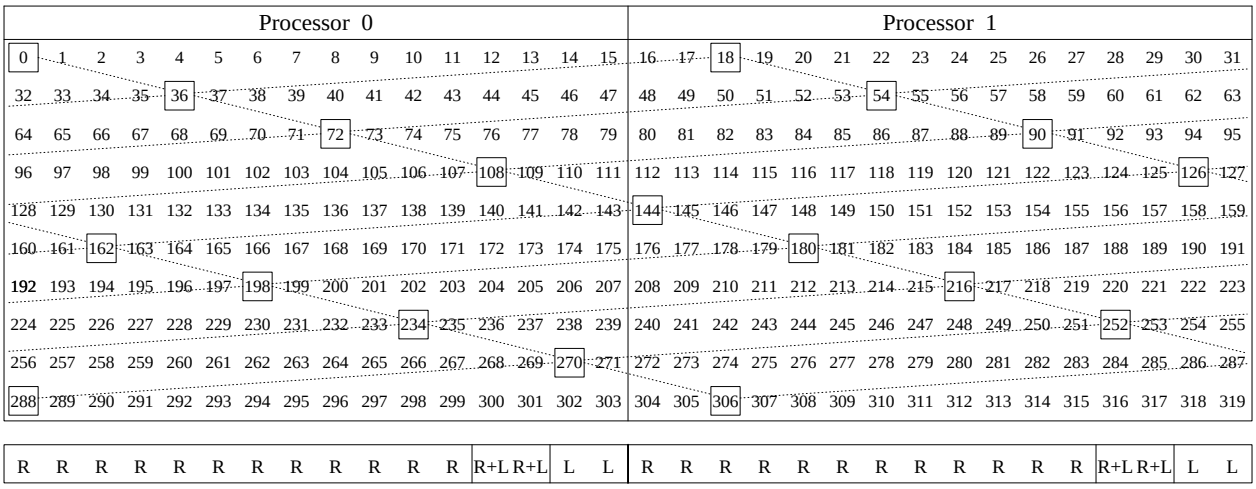

Figure 2 Layout of array elements distributed with *cyclic*(16) distribution over 2 processors. Rectangles indicate elements of the array section with lower bound $l = 0$ and stride $s = 18$.

We compute vector $R = (b_r, a_r)$ using the coordinates of the first array element accessed by processor 0 when lower bound $l = 0$ (not counting index 0 itself). Similarly, vector $L = (b_l, a_l)$ is computed by finding the last array element in the initial memory access cycle, and subtracting it from the coordinates of the array element that starts the next cycle. In the example in Figure 2 vector R is given by the array index 36, and thus $(b_r, a_r) = (4, 1)$. Vector L is given by the coordinates of the array index 288 relative to the array index 270, and therefore $(b_l, a_l) = (0, 9) - (14, 8) = (-14, 1)$.

It has been shown that vectors R and L form a lattice basis for regular section elements, and furthermore that the distance between two consecutive accesses must take one of the three possible values: R, L, or $R + L$ [10]. This is used to enumerate the indices of the accessed array elements in the increasing order, and to fill the table of local memory gaps based on the fact that vectors R and L correspond to local memory distances of $a_r k + b_r$ and $a_l k + b_l$, respectively.

While the maximal possible length of the cycle of local memory gaps is k, the actual cycle length for any given processor will depend on the $\text{GCD}(s, pk)$ and that processor's starting location. In the example in Figure 2 block size is 16, but the cycle length for either of the two processors is only 8. For simple regular sections each processor's starting location is fixed, and therefore only the entries corresponding to the offsets that are actually visited need to be stored in the table. However, when multiple loop induction variables are present within a single subscript (this case will be discussed in detail in Section 3), the starting location may vary with iterations of the outer loop. In Figure 3 we show how to construct the table of local memory gaps so that the entries corresponding to all offsets between 0 and $k - 1$ are filled.

As pointed out by Knies et al. address generation based on the local memory gap table makes a time versus space tradeoff. Since a table is needed for every array reference with different stride or distribution,

do $i = 0, k - b_r - 1$
  $\Delta\mathcal{M}[i] = a_r k + b_r$
  $Next[i] = i + b_r$
enddo

do $i = k - b_r, -b_l - 1$
  $\Delta\mathcal{M}[i] = (a_r k + b_r) + (a_l k + b_l)$
  $Next[i] = i + b_r + b_l$
enddo

do $i = -b_l, k - 1$
  $\Delta\mathcal{M}[i] = a_l k + b_l$
  $Next[i] = i + b_l$
enddo

Figure 3 Computing the memory gap and offset sequences.

for large block sizes this can introduce a substantial memory overhead. An important advantage of our algorithm is that the ideas used to generate the table can be extended to deal with the mentioned tradeoff. If the memory is a critical resource, our algorithm can be modified to simply return the vectors R and L. Each processor can then use these vectors to generate memory addresses on a demand-driven basis, by performing simple tests similar to those in Figure 3.

2.1 Experimental Results

We now compare the performance of different methods for local address generation. All the experiments in this and the following sections were done on an Intel iPSC/860 hypercube, using the *icc* compiler with -O4 optimization level and *dclock* timer. All times are reported in milliseconds and represent maximums over 32 processors.

Two versions of the SPMD node code are shown in Figure 4. In Figure 4(a) we use the memory gap and offset tables whose construction has been described above, while the code segment in Figure 4(b) does not require any tables since it generates the memory addresses to be accessed on a demand-driven basis.

We also compared these two methods with the full run-time generation of local memory addresses and with the virtual-block scheme proposed by Gupta et al. [6]. In the run-time resolution each processor executes all the loop iterations, and for each iteration it checks whether it owns the array element that is being assigned to, in which case it computes the local memory address for that array element and performs the assignment.

In the virtual-block scheme the array is treated as being *block* distributed across a large enough number of virtual processors, and these virtual processors are then cyclically mapped onto physical processors. If stride s is not greater than block size k, then all virtual processors own some of the array elements being accessed, i.e., all virtual processors are *active* [6]. In that case, each processor loops over all virtual processors mapped to it (each of these virtual processors corresponds to a block that the processor owns), computes the lower

```
Compute ΔM, Next, and start
i = start
offset = start mod k
while (i ≤ end) do
    A(i) = 100.0
    i = i + ΔM[offset]
    offset = Next[offset]
endwhile
```

(a) Table-based address computation.

```
Compute b_r, a_r, b_l, a_l, and start
i = start
offset = start mod k
while (i ≤ end) do
    A(i) = 100.0
    if (i < k − b_r) then
        i = i + (a_r k + b_r)
        offset = offset + b_r
    elseif (i ≥ −b_l) then
        i = i + (a_l k + b_l)
        offset = offset + b_l
    else
        i = i + (a_r k + b_r) + (a_l k + b_l)
        offset = offset + b_r + b_l
    endif
endwhile
```

(b) Demand-driven address generation.

Figure 4 Two versions of the SPMD node code for simple regular sections.

and upper bound of array elements accessed within each virtual processor, and performs the translation from the virtual processor's local index space to its own local index space. This translation is needed only for the lower and upper bound, since stride s in the index space of a virtual processor on processor m remains unchanged in the index space of m. However, if $s > k$, not all virtual processors are active. In that case, the solution by Gupta et al. is to scan all the active virtual processors (each containing exactly one array access) and, for each one of them, check whether it is located on processor m. This, essentially, is the same procedure as that performed in the run-time address resolution.

Table 1 contains execution times for all four methods. The measurements were performed with lower bound $l = 0$, while the upper bound was scaled in proportion to stride s, so that each processor accessed

		Table lookup	Demand driven	Virtual block	Run time
$k = 4$	$s = 3$	2.3	3.3	60.4	1071.8
	$s = 25$	2.6	3.1	1094.8	1064.7
	$s = 100$	3.1	3.2	1094.6	1065.6
$k = 16$	$s = 3$	2.3	2.7	16.3	1076.8
	$s = 25$	2.6	3.6	1097.4	1067.2
	$s = 100$	3.1	3.5	1095.1	1065.0
$k = 64$	$s = 3$	2.3	2.6	5.2	1072.3
	$s = 25$	2.6	3.2	32.3	1077.9
	$s = 100$	3.0	4.0	1097.7	1067.6
$k = 256$	$s = 3$	2.3	2.6	2.3	1063.2
	$s = 25$	2.6	2.9	9.3	1075.6
	$s = 100$	3.1	3.6	32.8	1078.2

Table 1 Execution times in milliseconds for different versions of loops with the SIV subscript.

10,000 array elements. For several reasons the reported times do not include the time spent in constructing the table or computing vectors R and L. First, if l, u, and s, as well as distribution parameters p and k, are known compile-time constants, then the tables (or vectors) can be computed by the compiler, without incurring any run-time cost. Second, even if this computation has to be done at run time, same tables (or vectors) would typically be reused for multiple array references in the program. And finally, our previous study [10] has shown that even for values of k as large as 512, table construction takes less than 1 millisecond, and thus would have no significant impact on results presented here.

Both the virtual-block scheme and the run-time address resolution perform significantly worse than the methods that take advantage of repetitive patterns in local memory addresses. While the run-time resolution consistently performs poorly, the performance of the virtual-block method depends on the values of s and k. As mentioned above, when $s > k$ the virtual-block scheme effectively reduces to run-time resolution, with small additional overhead incurred by virtual processor emulation. If $s \leq k$, the virtual-block method is significantly better than the run-time resolution, but it is only competitive with the other two methods when block size k is very large, and stride s is very small. In that case there are very few translations from a virtual processor's index space to a processor's index space, and many constant stride accesses are performed within each virtual processor.

Of particular interest is the comparison of the address generation method that uses memory gap and offset tables with the demand driven generation based on vectors R and L. As can be seen from Table 1, the performance penalty that one has to pay for direct generation of local addresses without using any tables is only minor. Therefore, if memory overhead is to be minimized, particularly when block sizes are large, this method can be used instead of the table lookup technique, without a significant performance degradation.

3 Multiple Induction Variables

We now show how the techniques described in the previous section can be extended to solve the problem of address generation for array subscripts containing multiple induction variables (MIV subscripts). Assuming again that the array A is distributed with $cyclic(k)$ distribution over p processors, our task is to generate the sequence of local memory addresses that a given processor m must access while executing its share of iterations of the following canonical loop nest

do $i = l_i,\ u_i,\ s_i$
  do $j = l_j,\ u_j,\ s_j$
    $A(i + j) = 100.0$
  enddo
enddo

The example in Figure 5 shows the array elements accessed in the the loop nest specified by $k = 4$, $p = 4$, $(l_i, u_i, s_i) = (0, 130, 37)$ and $(l_j, u_j, s_j) = (0, 18, 2)$. For a given iteration of the outer loop, a dark shaded square indicates the first array element accessed in that iteration, while lightly shaded squares show starting

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

Figure 5 Elements of an array distributed over 4 processors with $cyclic(4)$ distribution accessed in a doubly nested loop with parameters $(l_i, u_i, s_i) = (0, 130, 37)$ and $(l_j, u_j, s_j) = (0, 18, 2)$.

locations for different processors. For example, the first array element accessed in the second iteration of the outer loop is $A(37)$ owned by processor 1, and the starting locations for processors 2, 3, and 0 are $A(41)$, $A(45)$, and $A(49)$, respectively. All other array accesses are shown in non-shaded squares. The key observation is that for a given iteration of the outer loop, which determines each processor's starting location, the sequence of array accesses for a given processor depends on the stride of the innermost loop only (2, in our example). Therefore, in a manner similar to that described in Section 2, the access sequence can be generated using the table of memory gaps ($\Delta\mathcal{M}$). Moreover, the space versus time tradeoff can be handled in the same way as in the case of simple regular sections.

In order to use the $\Delta\mathcal{M}$ table on processor m, we need to determine m's starting location for each iteration of the outer loop, i.e., given a dark shaded square, we need to find the corresponding lightly shaded square that belongs to processor m. As described in Section 2, this can be done by finding the minimum of the smallest nonnegative solutions of k linear Diophantine equations. However, incurring the $O(k)$ cost for every iteration of the outer loop is hardly acceptable, and therefore we must look for a more efficient solution.

If the inner loop stride s_j is not greater than the block size k, processor m's starting location can be computed in constant time. If processor m owns $A(first)$, the first array element accessed in a given outer loop iteration, then its starting location (in global index space) is *first* itself. If this is not the case, then m's starting location *start* can be computed in the following way:

$$
\begin{aligned}
&gap = mk - first \bmod pk \\
&\textbf{if } (first \bmod pk \geq k(m+1)) \textbf{ then} \\
&\quad gap = gap + pk \\
&\textbf{endif} \\
&start = first + \lceil gap/s_j \rceil s_j
\end{aligned}
$$

As an example, we show how the starting locations for processors 0 and 3 are computed for the second iteration of the outer loop in Figure 5. The first array element accessed is $A(37)$, and since $pk = 16$, we have $first \bmod pk = 5$. When $m = 0$, we get $k(m+1) = 4$, and therefore $gap = 0 - 5 + 16 = 11$ and $start = 37 + 12 = 49$. On the other hand, when $m = 3$, $k(m+1) = 16$ is greater than 5, and thus $gap = 12 - 5 = 7$ and $start = 37 + 8 = 45$.

If $s_j > k$, then the starting location for processor m can be computed using the following method, which although simple, turns out to be efficient in practice. Processor m needs to find the smallest t such that it owns $A(first + ts_j)$. In naive run-time resolution, this is done by incrementing t until an array element owned by processor m is reached (or $first + ts_j$ exceeds the loop upper bound). The ownership test requires expensive mod and divide operations. These can be avoided by working directly with the offsets. Let $offset = first \bmod pk$, be the offset of the first element with respect to the beginning of the row. Let $moveR = s_j \bmod pk$ and $moveL = pk - moveR$, the right and the left displacement due to incrementing the induction variable by s_j. For our example in Figure 5, $moveR = 2$ and $moveL = -14$. The starting location $start$ is found by incrementing and decrementing the $offset$ by $moveR$ and $moveL$ respectively, until it falls within the range of offsets corresponding to processor m. The following procedure computes the starting location for processor m:

$start = first$; $offset = first \bmod pk$
while ($offset < km$ **or** $offset \geq k(m+1)$) **do**
  if ($offset < km$) **then**
    $offset = offset + moveR$; $start = start + s_j$
  else
    $offset = offset + moveL$; $start = start + s_j$
  endif
endwhile

For the example shown in Figure 5, the starting location $start$ for processor $m = 0$ and $first = 74$ is computed as follows. The offset of the first element accessed by the third iteration of the outer loop is $(74 \bmod 16) = 10$, which is greater than $k(m+1) = 4$. Therefore, $moveL = -14$ is added to 10; the resulting $offset = -4$ is less than $km = 0$. Now, two additions of $moveR = 2$ to -4 bring the $offset$ within processor 0's range. Correspondingly, $s_j = 2$ is added three times to $first = 74$ to obtain $start = 80$, the required starting location.

Although we now have ways to compute a processor's starting location that are more efficient than solving k Diophantine equations, we still have not exploited the repetitive pattern of array accesses generated by the outer loop. In the same way that $\Delta\mathcal{M}$ table reflects this pattern in the inner loop, we would like to construct the table of gaps between starting locations corresponding to consecutive iterations of the outer loop. Since the starting location can possibly have pk different offsets with respect to the beginning of the row, we can find the cycle of these offsets by computing the starting locations for at most $pk + 1$ iterations of the outer loop. This cycle can then be used to compute the table ($\Delta\mathcal{G}$) of local memory gaps between

consecutive starting locations.

Although the maximal possible size of $\Delta\mathcal{G}$ table is pk, the actual table size will depend on the length of the cycle of starting location offsets, and is likely to be much smaller than pk. However, using the argument similar to that for filling all k entries of $\Delta\mathcal{M}$ table, it is easy to see that for loop nests with more than two loops a table with all pk entries might be needed.

For all first elements accessed by the outer loop that have the same offset with respect to the beginning of the row, the gap between the first element and its corresponding starting location for processor m will be constant. Therefore, we can construct a table such that each entry $\Delta\mathcal{S}[t]$ contains the number of array elements between the first element accessed by the outer loop iteration *first*, such that *first* $\bmod pk = t$, and the starting location for processor m. For example, the $\Delta\mathcal{S}[15]$ entry of the table for processor 2 is 10 because offset 15 corresponds to the first element, 111, accessed by the fourth iteration of the outer loop and the starting location for processor 2 is 121; therefore, the number of elements that need to be skipped is $121 - 111 = 10$. The starting location for processor 2 is $Local(111 + \Delta\mathcal{S}[111 \bmod 16]) = Local(121) = 29$, where $Local$ performs the translation from global to local index space.

We now present an $O(pk)$ method (linear in the table size) for constructing the $\Delta\mathcal{S}$ table. From Figure 5, it can be observed that each of the first elements $A(4)$, $A(6)$, $A(8)$, $A(10)$, $A(12)$ and $A(14)$ accessed by the outer loop have $A(16)$ as the starting location for processor 0. Therefore, the $\Delta\mathcal{S}$ table will have 12, 10, 8 , 6, 4 and 2 as the number of elements skipped for the first elements with offsets 4, 6, 8, 10, 12 and 14 respectively (i.e., $\Delta\mathcal{S}[4] = 12$, $\Delta\mathcal{S}[6] = 10$, and so on). This observation can be used to obtain a linear-time method for computing the $\Delta\mathcal{S}$ table for processor m: start with the offset km, find the offset of the next element (say, e) accessed for the processor m (using the method described in Section 2); all the non-local elements accessed (by following stride s_j) between the first element and the element e have e as the starting location on processor m; the number of elements skipped for each offset corresponding to different first elements can be computed easily. This process is repeated for all the offsets between km and $km + k - 1$; the uninitialized $\Delta\mathcal{S}$ table entries correspond to the first elements accessed by the outer loop that have no corresponding starting location on processor m.

Since the $\Delta\mathcal{S}$ table is generated based on the stride of the innermost loop only, the table can be used to find the starting location for a particular processor and a given first element (which depends on all enclosing loops whose induction variables are present in the reference) irrespective of the number of loops enclosing the innermost loop. However, it introduces a global to local translation for every iteration of the outer loop.

As with the $\Delta\mathcal{M}$ table, if the distribution and the loop parameters are known at compile time, the $\Delta\mathcal{S}$ or the $\Delta\mathcal{G}$ table can be computed at compile time. However, if the tables need to be computed at run time, since the table entries are not modified in the loop, they can be computed outside the loop nest. Note that the upper bound computation is simpler because the local upper bound need not be the actual last element accessed by the processor. The procedures for computing the local upper bound ($GetUpperBound$) and for global to local index conversion ($Local$) were presented in [8].

3.1 Experimental Results

In this section we compare the performance of various address generation approaches for array references with MIV subscripts. Figure 6 shows two versions of the SPMD node code corresponding to our canonical loop nest example. The code in Figure 6(a) uses the $\Delta\mathcal{S}$ table to compute starting locations for each iteration of the outer loop, while the code in Figure 6(b) does this using the $\Delta\mathcal{G}$ table. In the inner loop, both versions use the $\Delta\mathcal{M}$ table to generate local memory addresses. We also implemented the method that uses the $\Delta\mathcal{M}$ table for accesses within the inner loop, but generates starting locations without any tables using one of the two methods described above. This method of computing starting locations for outer loop iterations was also combined with the demand-driven generation of addresses in the inner loops in the same way as shown in Figure 4(b). In that way memory overhead incurred by tables is completely eliminated. Finally, we compared these methods with naive run-time resolution, since, to the best of our knowledge, this is the only other technique that guarantees that array accesses will not be reordered.

Table 2 contains execution times for different values of the block size k and the inner loop stride s_j. The stride of the outer loop s_i was varied together with s_j so that no array element was accessed twice. The upper bounds were scaled proportionally to the corresponding strides so that every processor executed 100 iterations of each loop, resulting in total number of 10,000 array accesses per processor.

As expected, the performance of run-time resolution is much worse than any of the other address generation methods. The best performance was achieved using the $\Delta\mathcal{G}$ table in the outer loop and the $\Delta\mathcal{M}$ table in the inner loop. The method that uses the $\Delta\mathcal{S}$ table in the outer loop is somewhat less efficient because it requires an additional translation from global to local index space for every iteration of the outer loop, as shown in Figure 6(a). In addition, while the $\Delta\mathcal{G}$ table contains only those entries corresponding to starting location offsets actually visited by the iteration of the outer loop, the size of the $\Delta\mathcal{S}$ table is always pk (although some entries might be uninitialized). Since in our tests each processor performed 100

		$\Delta\mathcal{G}$ & $\Delta\mathcal{M}$	$\Delta\mathcal{S}$ & $\Delta\mathcal{M}$	$\Delta\mathcal{M}$ Only	No tables	Run time
$k = 4$	$s = 3$	4.3	4.8	5.0	5.9	1087.8
	$s = 25$	4.5	4.9	5.5	5.9	1080.7
	$s = 100$	5.0	5.5	6.0	6.2	1080.6
$k = 16$	$s = 3$	4.2	4.8	4.9	5.3	1092.8
	$s = 50$	4.3	4.9	5.2	6.3	1083.2
	$s = 100$	4.8	5.4	6.0	6.4	1081.0
$k = 64$	$s = 3$	4.2	5.1	4.9	5.2	1088.3
	$s = 25$	4.4	5.2	5.1	5.5	1093.9
	$s = 100$	4.9	5.7	5.7	6.7	1083.5
$k = 256$	$s = 3$	4.3	6.5	4.9	5.1	1079.3
	$s = 25$	4.4	6.7	5.2	5.4	1091.5
	$s = 100$	5.0	7.2	5.7	6.1	1094.2

Table 2 Execution times in milliseconds for different versions of loops with the MIV subscript.

```
Compute ΔS, ΔM and Next
do i = l_i, u_i, s_i
   first_j = i + l_j
   start_j = Local(first_j + ΔS[first_j mod pk])
   end_j = GetUpperBound(i + u_j)
   j = start_j; offset = j mod k
   while (j ≤ end_j) do
      A(j) = 100.0
      j = j + ΔM[offset]
      offset = Next[offset]
   endwhile
enddo
```

(a) $\Delta\mathcal{S}$ and $\Delta\mathcal{M}$ tables.

```
Compute ΔG, ΔM, Next, and start_j
countG = 0
do i = l_i, u_i, s_i
   end_j = GetUpperBound(i + u_j)
   j = start_j; offset = j mod k
   while (j ≤ end_j) do
      A(j) = 100.0
      j = j + ΔM[offset]
      offset = Next[offset]
   endwhile
   start_j = start_j + ΔG[countG]
   countG = (countG + 1) mod Size(ΔG)
enddo
```

(b) $\Delta\mathcal{G}$ and $\Delta\mathcal{M}$ tables.

Figure 6 Two versions of the SPMD node code for the MIV subscript.

iterations of the outer loop, the size of the $\Delta\mathcal{G}$ could not exceed 100 even for the large block sizes. On the other hand, for $k = 256$, the $\Delta\mathcal{S}$ table had 8K elements and table lookups had poor locality, resulting in increased performance degradation.

As mentioned earlier, the size of the $\Delta\mathcal{G}$ table can be pk in the worst case. In the case, the $\Delta\mathcal{G}$ table memory overhead is unacceptable, methods that generate starting locations on a demand-driven basis can be applied. Both of the two approaches, the one using the $\Delta\mathcal{M}$ table (typically much smaller than $\Delta\mathcal{S}$ or $\Delta\mathcal{G}$ table) in the inner loop, and the other without any table space overhead, perform only slightly worse than the address generation based on using both the $\Delta\mathcal{G}$ and $\Delta\mathcal{M}$ tables, and therefore should be methods of choice if memory overhead needs to be reduced or completely eliminated.

4 Coupled Subscripts

All address generation methods presented so far deal only with one-dimensional arrays. Chatterjee et al. have shown that for multidimensional regular array sections (corresponding to array references with independent subscripts), the memory access problem reduces to multiple applications of the algorithm used for the one-dimensional case [3]. However, this is not necessarily true if subscripts are dependent, i.e., if two or more subscript positions contain the same loop induction variables.

The example in Figure 7(a) shows a three-deep loop nest and a two-dimensional array reference with both subscripts containing induction variable i, the outermost loop induction variable. We assume that array A is distributed with $cyclic(k_1)$ distribution over p_1 processors along its first dimension and with $cyclic(k_2)$ distribution over p_2 processors along its second dimension. Although two subscripts contain a common loop induction variable, as pointed out in Section 3, the $\Delta\mathcal{M}$ table for a subscript depends on the stride of the deepest loop only. Therefore, in this example, we can compute two independent tables: $\Delta\mathcal{M}_j$ for the subscript in the first dimension and $\Delta\mathcal{M}_k$ for the second dimension. The resulting SPMD code is shown in

Figure 7(b).

Based on the above observation we define two subscripts to be *coupled* if and only if the deepest loop induction variables occurring in two subscripts are identical. A typical example of a loop with coupled subscripts is shown below.

do $i = l, u$
  $A(s_1 i + c_1, s_2 i + c_2) = 100.0$
enddo

In order to use table lookups for address generation of array references with coupled subscripts, we first show how the method presented by Chatterjee et al. [3] can be extended to find the starting location. Let $l_1 = s_1 l + c_1$ and $l_2 = s_2 l + c_2$ be the values of the two subscripts in the first loop iteration. The starting location for a given processor (m_1, m_2) corresponds to the smallest nonnegative integer j which satisfies both of the following two inequalities

$$k_1 m_1 \leq (l_1 + s_1 j) \bmod p_1 k_1 < k_1(m_1 + 1), \text{ and}$$
$$k_2 m_2 \leq (l_2 + s_2 j) \bmod p_2 k_2 < k_2(m_2 + 1).$$

As shown by Chatterjee et al. [3] for the one-dimensional case, finding such a j is equivalent to finding the minimum of the smallest nonnegative solutions of the following set of simultaneous linear Diophantine equations

$$\{s_1 j - p_1 k_1 q_1 = i_1 \mid k_1 m_1 - l_1 \leq i_1 \leq k_1 m_1 - l_1 + k_1 - 1\}, \text{ and}$$
$$\{s_2 j - p_2 k_2 q_2 = i_2 \mid k_2 m_2 - l_2 \leq i_2 \leq k_2 m_2 - l_2 + k_2 - 1\}.$$

Let $d_1 = \text{GCD}(s_1, p_1 k_1) = \alpha_1 s_1 - \beta_1 p_1 k_1$ and $d_2 = \text{GCD}(s_2, p_2 k_2) = \alpha_2 s_2 - \beta_2 p_2 k_2$ where $\alpha_1, \beta_1, \alpha_2, \beta_2 \in \mathcal{Z}$. Two separate applications of the extended Euclid algorithm [4] determine d_1, α_1, β_1 and d_2, α_2, β_2. The

```
do i = l_i, u_i, s_i
  do j = l_j, u_j, s_j
    do k = l_k, u_k, s_k
      A(i + j, i + k) = 100.0
    enddo
  enddo
enddo
```

(a) Original loop.

```
Compute ΔM_j, Next_j, ΔM_k, and Next_k
do i = l_i, u_i, s_i
  Compute start_j and end_j
  j = start_j; offset_j = start_j mod k_1
  while (j ≤ end_j) do
    Compute start_k and end_k
    k = start_k; offset_k = start_k mod k_2
    while (k ≤ end_k) do
      A(j, k) = 100.0
      k = k + ΔM_k[offset_k]
      offset_k = Next_k[offset_k]
    endwhile
    j = j + ΔM_j[offset_j]
    offset_j = Next_j[offset_j]
  endwhile
enddo
```

(b) SPMD node code.

Figure 7 Example program with dependent MIV subscripts.

general solution for equations from the two sets is given by one-parameter families $j = (i_1\alpha_1 + p_1k_1\gamma_1)/d_1$, $q_1 = (-i_1\beta_1 + s_1\gamma_1)/d_1$ and $j = (i_2\alpha_2 + p_2k_2\gamma_2)/d_2$, $q_2 = (-i_2\beta_2 + s_2\gamma_2)/d_2$, respectively, where $\gamma_1, \gamma_2 \in \mathcal{Z}$. Since we are looking for the solution $j \geq 0$, we must have $\gamma_1 \geq \lceil -i_1\alpha_1/p_1k_1 \rceil$ and $\gamma_2 \geq \lceil -i_2\alpha_2/p_2k_2 \rceil$. The simultaneous solution of the two equations satisfies the equation

$$(i_1\alpha_1 + p_1k_1\gamma_1)/d_1 = j = (i_2\alpha_2 + p_2k_2\gamma_2)/d_2,$$

which is equivalent to

$$d_2p_1k_1\ \gamma_1 - d_1p_2k_2\ \gamma_2 = d_1i_2\alpha_2 - d_2i_1\alpha_1.$$

Let $i = d_1i_2\alpha_2 - d_2i_1\alpha_1$ and $d = \text{GCD}(d_2p_1k_1, d_1p_2k_2) = \alpha d_2p_1k_1 + \beta d_1p_2k_2$, $\alpha, \beta \in \mathcal{Z}$. The general solution for γ_1 and γ_2 is given by $\gamma_1 = (i\alpha + d_1p_2k_2\gamma)/d$ and $\gamma_2 = (-i\beta + d_2p_1k_1\gamma)/d$, where $\gamma \in \mathcal{Z}$. The minimal value of parameter γ that satisfies constraints on γ_1 and γ_2 is

$$\gamma = \max\left\{ \left\lceil \frac{\lceil \frac{-i_1\alpha_1}{p_1k_1} \rceil d - i\alpha}{d_1p_2k_2} \right\rceil, \left\lceil \frac{\lceil \frac{-i_2\alpha_2}{p_2k_2} \rceil d + i\beta}{d_2p_1k_1} \right\rceil \right\}.$$

We can back-substitute this value to compute γ_1 (or γ_2) and, consequently, the desired solution for j. The minimum of the smallest nonnegative solutions for j (across all pairs of equations that have solutions) determines the first array element $A(l_1+s_1j, l_2+s_2j)$ accessed by processor (m_1, m_2). The complete algorithm to determine the processor's starting location is shown in Figure 8.

Using the same idea as described by Chatterjee et al. [3] for the one-dimensional case, the table of memory gaps can be obtained by first sorting the initial sequence of array accesses and then computing the distances

---

Input: Distribution parameters (p_1, k_1), (p_2, k_2), loop parameters (l_1, s_1), (l_2, s_2), and processor number (m_1, m_2).
Output: The starting location $(start_1, start_2)$.
Method:

```
1   min = ∞
2   (d1, α1, β1) ← Extended-Euclid(s1, p1k1)
3   (d2, α2, β2) ← Extended-Euclid(s2, p2k2)
4   (d, α, β) ← Extended-Euclid(d2p1k1, d1p2k2)
5   do i1 = k1m1 − l1, k1m1 − l1 + k1 − 1
6     do i2 = k2m2 − l2, k2m2 − l2 + k2 − 1
7       i = d1i2α2 − d2i1α1
8       if (i1 mod d1 = 0 and i2 mod d2 = 0 and i mod d = 0) then
9         γ = max{ ⌈ ⌈(−i1α1)/(p1k1)⌉ d − iα / d1p2k2 ⌉, ⌈ ⌈(−i2α2)/(p2k2)⌉ d + iβ / d2p1k1 ⌉ }
10        j = (i1α1 + p1k1(iα + d1p2k2γ)/d)/d1
11        if (j < min) min = j
12      endif
13    enddo
14  enddo
15  (start1, start2) = (l1 + s1 min, l2 + s2 min)
```

Figure 8 Algorithm to compute the starting location for processor (m_1, m_2).

between every two consecutive locations. Both $\Delta\mathcal{M}$ and *Next* (analogous to those presented in Section 2) will be two-dimensional tables, with each entry containing two values, one for each array dimension.

Since an access sequence can be of length k_1k_2 in the worst case, table construction based on sorting this sequence has the complexity $O(k_1k_2\log(k_1k_2))$. In the one-dimensional case we were able to use the fact that regular section indices form an integer lattice to develop an algorithm that is linear in the size of table. Furthermore, we used the lattice basis vectors R and L (defined in Section 2) to efficiently generate local memory addresses at run time and save memory space needed to store the tables.

Vectors R and L correspond to memory gaps of the first and last element in each block of size k , i.e., elements with offsets 0 and $k-1$ within the block (see Figure 2). By analogy, in the two-dimensional case we can consider the access gaps from the four corners of a rectangular $k_1 \times k_2$ block, i.e., elements with offset pairs $(0,0)$, $(k_1-1,0)$, $(0,k_2-1)$ and (k_1-1,k_2-1) within the block. Access gaps for all other array elements can then be represented as positive integer linear combinations of these vectors. However, while in the case of simple regular sections the number of distinct memory gaps was bounded by a constant (the only possible gaps were R,L, and $R+L$), with two-dimensional coupled subscripts the number of possible memory gaps, and thus the number of required linear combinations of basis vectors can be arbitrary large. Therefore, finding a linear time algorithm for constructing the table of memory gaps in the presence of coupled subscripts remains an open problem.

A simple, though not as efficient, approach for computing local addresses without tables uses vectors R and L in each of the two coupled dimensions. Processor (m_1, m_2) uses the vectors in the first dimension to skip across array elements that are owned by some of the processors having m_1 as the first coordinate, and the vectors in the second dimension to skip across array elements owned by some of the processors having m_2 as the second coordinate. When an array element, with identical iteration counts in two dimensions and owned by processor (m_1, m_2) is found, the assignment is performed.

The techniques presented in this section can be extended to handle coupled subscripts with multiple induction variables in a manner similar to the way the techniques for single induction variables, presented in Section 2, were extended to handle multiple induction variables in Section 3.

4.1 Experimental Results

We now compare the performance of different address generation methods in the presence of coupled subscripts. In addition to the run-time resolution, we compare the two versions of the SPMD code shown in Figure 9. Figure 9(a) shows the code that uses two-dimensional $\Delta\mathcal{M}$ and *Next* tables to generate local indices for each array access. Code based on separately incrementing array indices in each dimension until the identical iteration number is reached is shown in Figure 9(b). Due to the space consideration, we only show the code using the $\Delta\mathcal{M}$, *Next*, and $\Delta\mathcal{I}$ tables (the last one contains the iteration gaps between consecutive array accesses). However, in our experiment, this version was implemented using vectors R and L in both dimensions, since the intention was to completely eliminate memory overhead.

```
Compute ΔM and Next
Compute start_1 and start_2
Compute end_1 and end_2
i_1 = start_1; i_2 = start_2
offset_1 = i_1 mod k_1; offset_2 = i_2 mod k_2
while (i_1 ≤ end_1 and i_2 ≤ end_2) do
   A(i_1, i_2) = 100.0
   (i_1, i_2) = (i_1, i_2) + ΔM[offset_1, offset_2]
   (offset_1, offset_2) = Next[offset_1, offset_2]
endwhile
```

(a) Two-dimensional $\Delta\mathcal{M}$ table.

```
Compute ΔM_{1,2}, ΔI_{1,2}, and Next_{1,2}
Compute the start_1 and start_2
Compute the end_1 and end_2
i_1 = start_1; i_2 = start_2
offset_1 = i_1 mod k_1; offset_2 = i_2 mod k_2
iter_1 = iter_2 = (Global(start_1) − l)/s_1
while (i_1 ≤ end_1 and i_2 ≤ end_2) do
   A(i_1, i_2) = 100.0
   do
      do
         i_1 = i_1 + ΔM_1[offset_1];
         offset_1 = Next_1[offset_1]
         iter_1 = iter_1 + ΔI_1[offset_1];
      while (iter_1 < iter_2)
      while (iter_2 < iter_1) do
         i_2 = i_2 + ΔM_2[offset_2];
         offset_2 = Next_2[offset_2]
         iter_2 = iter_2 + ΔI_2[offset_2];
      endwhile
   while (iter_1 ≠ iter_2)
endwhile
```

(b) Two one-dimensional $\Delta\mathcal{M}$ tables.

Figure 9 Two versions of the SPMD node code with coupled subscripts.

Table 3 contains performance results for our canonical loop example with coupled subscripts on an 8×4 processor grid. In addition to the execution times, for each combination of a block size and a loop stride, we show the number of processors that are actually performing some array accesses. Since accesses due to array references with coupled subscripts are relatively sparse, it is quite likely that not all the processors will participate in the loop execution. Moreover, while we scaled the loop upper bound so that the maximum number of array accesses per processor is 10,000, not all the processors were accessing exactly 10,000 elements.

These facts had a particularly strong influence on the performance of the run-time resolution. It is very clear from Table 3 that the run-time resolution performs better as the number of active processors decreases. Furthermore, the performance is not uniform across different values of loop parameters that result in the same number of active processors. This is the consequence of the uneven distribution of array accesses among active processors. While the performance of the method with table lookups is, in essence, proportional to the maximum number of array accesses per processor, the performance of the run-time resolution is proportional to the sum of array accesses of all the processors. The performance of the code using vectors (based on the code presented in Figure 9(b)) lies in between the two extremes. It also degrades with the increase in the number of active processors, but it does so more gracefully, and is still significantly better than the run-time resolution.

Although in the worst case the $\Delta\mathcal{M}$ table can be of size $k_1 k_2$ and thus incur significant memory overhead, our experience indicates that this is not very likely to happen in practice. In all our test runs, where block

$k_1 \times k_2$	(s_1, s_2)	Active procs	$\Delta\mathcal{M}$ table	No tables	Run time
4×4	$(1, 1)$	8	6.6	16.3	337.6
	$(1, 3)$	24	6.6	28.7	639.8
	$(10, 5)$	8	6.4	17.5	333.6
4×8	$(1, 1)$	8	6.5	16.0	337.5
	$(1, 3)$	16	6.5	20.6	438.8
	$(10, 5)$	16	6.4	30.5	629.7
8×8	$(1, 1)$	8	6.7	15.9	337.0
	$(1, 3)$	24	6.7	32.5	838.3
	$(10, 5)$	8	6.6	17.7	334.8
8×16	$(1, 1)$	8	7.1	15.7	335.4
	$(1, 3)$	16	6.8	19.2	835.8
	$(10, 5)$	16	6.7	29.6	631.9
16×16	$(1, 1)$	8	7.3	15.6	335.4
	$(1, 3)$	24	7.2	29.8	835.8
	$(10, 5)$	8	7.2	16.7	334.3

Table 3 Execution times in milliseconds for different versions of loops with coupled subscripts.

and processor grid sizes were always powers of 2 (which is arguably the most common case), the table size never exceeded the maximum of k_1 and k_2. Since the array references with coupled subscripts are not as frequent as those whose subscripts contain single induction variables, and since the method that generates addresses without using any tables is from 2 to 5 times slower than the table lookup, using the tables in this case is probably a better choice.

5 Conclusions

Although data-parallel languages, such as High Performance Fortran, are becoming increasingly popular, many issues about their compilation are not fully resolved. In this paper we have presented efficient techniques for generating local addresses for array references with arbitrary affine subscripts. We have improved on the previously described table-based address generation scheme [3] in two ways. First, we have shown how ideas used to develop our linear-time table construction algorithm [10] can be used to generate local addresses without table lookups, with only insignificant performance degradation. Second, we have extended the table lookup method to references with multiple loop induction variables and coupled array subscripts. The generality of our technique and the efficiency with which different subscripts are handled, as demonstrated by our extensive experimental results, make it suitable for inclusion in compilers and run-time systems for HPF-like languages.

Our future work will deal with applying the address computation scheme presented here to generate communication sets and optimize communication placement. Moreover, the techniques for handling array references with coupled subscripts will be further investigated in order to improve their effectiveness.

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, 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.

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

[5] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.

[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] K. Kennedy, N. Nedeljković, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. Technical Report CRPC-TR94485-S, Center for Research on Parallel Computation, Rice University, October 1994.

[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. Compile-time generation of regular communications patterns. In *Proceedings of Supercomputing '91*, pages 101–110, Albuquerque, NM, November 1991.

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

[14] 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.

[15] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.