Generalized Overlap Regions for Communication Optimization in Data-Parallel Programs

Arun Venkatachar J. Ramanujam* Ashwath Thirumalai
Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803-5901, USA
{arun,jxr,ash}@ee.lsu.edu

Abstract

Data-parallel languages such as High Performance Fortran, Vienna Fortran and Fortran D include directives for alignment and distribution that describe how data and computation are mapped onto the processors in a distributed-memory multiprocessor. A compiler for these language that generates code for each processor has to compute the sequence of local memory addresses accessed by each processor and the sequence of sends and receives for a given processor to access non-local data. While the address generation problem has received much attention, issues in communication have not been dealt with extensively. A novel approach for the management of communication sets and strategies for local storage of remote references is presented. Algorithms for deriving communication patterns are discussed first. Then, two schemes that extend the notion of a local array by providing storage for non-local elements (called *overlap regions*) interspersed throughout the storage for the local portion are presented. The two schemes, namely *course padding* and *column padding* enhance locality of reference significantly at the cost of a small overhead due to unpacking of messages. The performance of these schemes are compared to the traditional buffer-based approach and improvements of up to 30% in total time are demonstrated. Several message optimizations such as offset communication, message aggregation and coalescing are also discussed.

1 Introduction

Programming massively parallel distributed memory machines involves partitioning data and computation across processors. Data-parallel languages like Fortran D [5], Vienna Fortran [3] and High Performance Fortran [8, 12] include directives for alignment and distribution that describe how data and computation are mapped onto the processors in a distributed-memory multiprocessor. In these languages, arrays are aligned to an abstract Cartesian grid called a *template;* the template is then distributed across the processors. The distribution of computation follows the *owner-computes* rule, *i.e.,* the processor performs only those computations (or assignments) for which it owns the left hand side variable. Access to non-local right hand side variables is achieved by inserting sends and receives. In order to generate node code we need to compute the sequence of sends and receives for a given processor to access non-local data in addition to computing the sequence of local memory addresses accessed by each processor. In addition, the compiler has to manage storage for non-local references and exploit data locality in code execution whenever possible.

Using the fact that the address generation algorithm is based on integer lattices [9, 19] and the set of array elements referenced can be generated by integer linear combinations of basis vectors for the lattice, we present algorithms for deriving the communication sets of each processor. Then, two schemes that extend the notion of a local array by providing storage for non-local elements (called *overlap regions*) interspersed throughout the storage for the local portion are presented. The two schemes, namely *course padding* and *column padding* enhance locality of reference significantly at the cost of a small overhead due to unpacking of messages. We compare the performance of these schemes to the traditional buffer-based approach and demonstrate performance improvements of up to 30% in total time. In addition, we discuss several additional optimizations such as offset communication, message aggregation and coalescing.

*Supported in part by an NSF Young Investigator Award CCR–9457768, and NSF grant CCR–9210422, and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

Several papers have addressed the code generation problem for distributed memory machines [1, 2, 4, 7, 9, 10, 11, 14, 15, 16, 17, 18, 19, 20, 21]. Many of these papers have developed good schemes for address generation but issues in communication optimization have not received much attention. Chatterjee *et al.* [4] describe the set of accesses as a finite state machine (FSM) and use the FSM to generate communication. Their model of communication is based on packing elements using an address-value pair before sending. At the receive end there is no need to unpack messages and processors use message buffers for the combined receive-execute phase. The code uses expensive conditionals and may suffer from locality when a large numbers of processors send messages. Extensions to multiple right hand side arrays and accesses have not been dealt with. Stichnoth [15] and Gupta *et al.* [7] discuss similar schemes for communication.

Kennedy, Nedeljkovic and Sethi [10] view the set of accesses as an integer lattice and make use of the repeating set of accesses. Their communication model performs packing of elements before communication. On the receive side each processor allocates overlap regions at the end of local storage (for each array) for storing non-local accesses. They then construct a table of accesses to index into the local as well as the overlap region. Their code does not require unpacking of messages but requires the added overhead of recomputing table entries; of the two tables of length k that they use, the entries in one are updated in each iteration. In effect a processor that performs N local computations computes a total of N addresses for each right hand side reference, computed using the two tables of length k. Communication generation issues are also dealt with by Thirumalai [17] and by Thirumalai, Ramanujam and Venkatachar [20]. In contrast to these papers, we present storage schemes for non-local references that help maintain data locality while reducing runtime overhead with $\texttt{CYCLIC}(k)$ distributions. In particular, recomputing the address pattern for references is eliminated. Our approach to provide storage area for non-local references was inspired by Gerndt's work [6] on overlap regions in the context of shift communication with `BLOCK` distribution. Also, we present adaptations of these storage schemes for several communication optimizations.

This paper is organized as follows. In Section 2, we provide a short review the problem of address generation algorithm based on integer lattices. In Section 3, we present algorithms to generate the necessary communication sets; these algorithms exploit the repeating communication pattern. We show in Section 4 how to generate the sends for each processor and propose a method to directly read and execute the received elements from the buffers. In Section 5, we discuss generalized overlap regions that provide local storage for non-local references and present algorithms to unpack data. Two different schemes, namely *course padding* and *column padding* are proposed to unpack data into local storage space. In the next section, we discuss message optimizations such as coalescing, aggregation and offset communication. Experimental results are presented in Section 7 and we conclude in Section 8 with a discussion.

2 Review of lattice-based local address generation

We consider an array A identically aligned to the template T; this means that if, $A(i)$ is aligned with $T(ai+b)$, then the alignment stride $a = 1$ and the alignment offset $b = 0$. Further let this template be distributed in a block-cyclic fashion with a block size of k across p processors. This is also known as a $\texttt{CYCLIC}(k)$ distribution [8, 12]. A typical HPF assignment statement is of the form $A(L : H : s) = \cdots$ where s is the access stride. Given an array statement with HPF-style data mappings, it is our aim to generate SPMD code along with communication for the different processors.

We treat the global address space as a two dimensional space; every element of A has an address of the form (x, y) in this space, where x is the course (or row) to which this element belongs and y is the offset of the element in that course. Given a global address i of an array element, the processor m that owns the element is $m = (i \text{ div } k) \bmod p$ and the local address of that element on processor $m = k(i \text{ div } pk) + i \bmod k$.

We [19] presented an address generation algorithm based on integer lattices. We showed that the set of array elements referenced can be generated by integer linear combinations of the basis vectors of the lattice. Let B be the basis matrix of the two dimensional space, where,

$$B = \begin{bmatrix} \alpha_1 & \beta_1 \\ \alpha_2 & \beta_2 \end{bmatrix}$$

It was shown [19] that the determinant of B is the access stride s itself. The access pattern for the array A repeats after every $\frac{s}{\gcd(s,pk)}$ courses of elements. Thus, the pattern of accesses in the first $\frac{s}{\gcd(s,pk)}$ courses can be used to determine the rest of the local addresses accessed. Using a transformation $T = B^{-1}$ and Fourier-Motzkin elimination [13, 21], we derive the code shown below for address pattern generation. The code for $A(L : H : s)$ is easily obtained by making minor modifications to the loop bounds of the nested loop before applying the transformation T [19].

$len = 0$
do $u = \left\lceil \frac{(-mk-k+1)\beta_1}{s_1} \right\rceil , \left\lfloor \frac{(s_1-1)\beta_2 - mk\beta_1}{s_1} \right\rfloor$
  if $\beta_1 = 0$
    $lower = \left\lceil \left(\frac{mk - u\alpha_2}{\beta_2} \right) \right\rceil; upper = \left\lfloor \left(\frac{mk+k-1-u\alpha_2}{\beta_2} \right) \right\rfloor$
  else
    $lower = \left\lceil \max\left(\frac{mk-u\alpha_2}{\beta_2}, \frac{-u\alpha_1}{\beta_1} \right) \right\rceil; upper = \left\lfloor \min\left(\frac{mk+k-1-u\alpha_2}{\beta_2}, \frac{s_1-1-u\alpha_1}{\beta_1} \right) \right\rfloor$
  endif
  do $v = lower , upper$
    $\mathit{loc\_pattern}_m[len] = (\alpha_1 u + \beta_1 v)k + \alpha_2 u + \beta_2 v - mk$
    $\mathit{glob\_pattern}_m[len] = (\alpha_1 u + \beta_1 v)pk + \alpha_2 u + \beta_2 v$
    $len = len + 1$
  enddo
enddo

3 Communication Set Generation

Communication set generation involves computing, for each processor, the set of elements to be sent, the processors to which these sets are sent, the set of elements received and the processors from which these are received. We show that there is a pattern of communication which can be exploited in communication set generation. We then present algorithms for deriving these communication patterns. The send communication pattern stores the index of the destination processor and the set of elements to be sent before the communication pattern starts repeating. The receive communication pattern on the other hand stores the index of the processor sending the data and the address of the element to be received. These communication patterns are used to derive both the send and receive communication sets for each processor. In the next sections, we use these pattern to pack and unpack (if necessary) messages. We discuss communication generation for the following array statement with HPF-style data mappings.

$$A(L_1 : H_1 : s_1) = \mathcal{F}\left(B(L_2 : H_2 : s_2)\right) \tag{1}$$

where arrays A and B are distributed using a $\texttt{CYCLIC}(k)$ distribution over p processors. Extensions to cases with different block sizes for the arrays as well as a different number of processors over which the arrays are distributed are straightforward and are not discussed here. Optimizations for multiple right hand side array sections are discussed in Section 6.

3.1 Receive Pattern Generation

The access pattern of the LHS array in the global address space repeats itself after pk elements are accessed, i.e. after every s_1 rows, where s_1 is the access stride of the LHS array. In order to generate the receive set, we run the code generated as described in the previous section to obtain the global address of the first pk elements accessed on the LHS. For each of these elements we calculate the processor that owns the corresponding RHS element. The pattern of elements which are needed to perform computation and are local to the processor are also computed. The length of the communication pattern, len_l, corresponds to len in the code discussed in the last section. The receive pattern generation code is shown below.

do $j = 0, len_l - 1$
  $\mathit{index}_l = \mathit{glob\_LHS\_pattern}_m[j]$
  $\mathit{RIndPattern}_m[j] = \mathit{index}_r = \frac{s_2}{s_1}(\mathit{index}_l - L_1) + L_2$
  $\mathit{RProcPattern}_m[j] = (\mathit{index}_r \text{ div } k) \bmod p$
  if $m = \mathit{RProcPattern}_m[j]$
    $\mathit{RPattern}_m[j] = (k(\mathit{index}_r \text{ div } pk) + \mathit{index}_r \bmod k)$
  endif
enddo

3.2 Send Pattern Generation

Our approach to the send set generation is very similar to the receive set generation shown in Section 3.1. It should be obvious that there is a one to one correspondence between the LHS array element and the RHS array elements. So the send pattern for a processor is the same as the receive pattern calculated assuming that the RHS element is actually the LHS element. Here again we notice that the send pattern should repeat after every s_2^{th} row. len_r is the length of the communication pattern which corresponds to len in the code discussed in the last section. The pattern generation code is shown below.

do $j = 0, len_r - 1$
  $index_r = glob\_RHS\_pattern_m[j]$
  $SIndPattern_m[j] = index_l = \frac{s_1}{s_2}(index_r - L_2) + L_1$
  $SProcPattern_m[j] = (index_l \text{ div } k) \bmod p$
  if $m \neq SProcPattern_m[j]$
    $SPattern_m[j] = loc\_RHS\_pattern_m[j]$
  endif
enddo

Example: Here we illustrate communication set generation for the array assignment statement $A(0 : H_1 : 5) = B(0 : H_2 : 3)$ where A and B are aligned identically with the template T, and $p = 4$ and $k = 8$. Figure 1 shows the elements accessed on the LHS and RHS on all processors. The communication data structures, namely the pattern tables for processor 0 are shown below.

$RIndPattern_0$	0	3	21	39	42	60	78	81
$RProcPattern_0$	0	0	2	0	1	3	1	2
$RPattern_0$	0	3		15				

$SIndPattern_0$	0	3	6	33	36	39	66	69
$SProcPattern_0$	0	0	1	2	3	0	1	2
$SPattern_0$			6	9	12		18	21

For the above example, if we run the code discussed in Section 2 we obtain the local and global addresses of elements accessed in the communication pattern. Using this table if we run the code discussed in Section 3.1 and in Section 3.2 for processor 0, we can build the communication pattern tables as shown above.

The $RIndPattern_0$ table stores the global indices of the elements on RHS which are needed by the LHS before the communication pattern starts repeating. $RPattern_0$ stores the local indices of the elements on RHS which are needed by LHS and are local for processor 0. Similarly $SIndPattern_0$ table stores the global indices of the elements on RHS which are accessed before the communication pattern starts repeating. $SPattern_0$ stores the local indices of the elements on RHS which are needed by other processors. $RProcPattern_0$ stores the index of processors from which processor 0 receives data and $SProcPattern_0$ stores the index of processors to which processor 0 sends data. The send and receive pattern information are needed to generate the communication algorithms as discussed in the next section.

4 Buffer-based Communication Algorithm

In order to perform computations locally on a processor, the processor needs to receive all the non-local data using interprocessor communication. The communication between processors involves packing of data into buffers before being sent from one processor to another. A compiler which uses owner-computes rule needs to perform all the sends and receives before execution. At the receive end, the processor can either unpack the non-local elements into its local space or can read directly from the receive buffers during execution.

In this section we present an algorithm for the send and the receive-execute phases. In the send phase, data to be sent from one processor to another are packed into message buffers using the send pattern discussed in Section 3.2. On the receive end, the non-local elements are read directly from the communication buffers during execution. This method does not require unpacking of data before execution. We assume that each processor maintains separate send and receive buffers for all the p processors.

Processor 0								Processor 1								Processor 2								Processor 3							
$\boxed{0}$	1	2	3	4	$\boxed{5}$	6	7	8	9	$\boxed{10}$	11	12	13	14	$\boxed{15}$	16	17	18	19	$\boxed{20}$	21	22	23	24	$\boxed{25}$	26	27	28	29	$\boxed{30}$	31
32	33	34	$\boxed{35}$	36	37	38	39	$\boxed{40}$	41	42	43	44	$\boxed{45}$	46	47	48	49	$\boxed{50}$	51	52	53	54	55	56	57	58	59	$\boxed{60}$	61	62	63
64	$\boxed{65}$	66	67	68	69	$\boxed{70}$	71	72	73	74	$\boxed{75}$	76	77	78	79	$\boxed{80}$	81	82	83	84	$\boxed{85}$	86	87	88	89	$\boxed{90}$	91	92	93	94	$\boxed{95}$
96	97	98	99	$\boxed{100}$	101	102	103	104	$\boxed{105}$	106	107	108	109	$\boxed{110}$	111	112	113	114	$\boxed{115}$	116	117	118	119	$\boxed{120}$	121	122	123	124	$\boxed{125}$	126	127
128	129	$\boxed{130}$	131	132	133	134	$\boxed{135}$	136	137	138	139	$\boxed{140}$	141	142	143	144	$\boxed{145}$	146	147	148	149	$\boxed{150}$	151	152	153	154	$\boxed{155}$	156	157	158	159

(a): Global addresses of array A accessed for $p = 4$, $k = 8$, and $s = 5$.

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	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159

(b): Global addresses of array B accessed for $p = 4$, $k = 8$, and $s = 3$.

Figure 1: Global addresses of arrays A and B accessed in the array statement $A(0 : H_1 : 5) = B(0 : H_2 : 3)$ with $p = 4$ and $k = 8$.

4.1 Send Phase

To perform the send, we use the send pattern discussed in Section 3.2 to determine where the elements in the *SPattern*$_m$ table have to be sent. We then pack the elements to be sent into the corresponding *SendBuffer*. The algorithm to perform the send is shown below. For processor m, we run the following code:

Scount$_q$ $= 0$, where $0 \leq q \leq p - 1$
baseadd $= 0$
do $j = 0, \frac{max_B}{s_2 k}$
    do $i = 0$, *sizeof*(*SProcPattern*$_m$) $- 1$
        $q =$ *SProcpattern*$_m[i]$
        if $m \neq q$
            address $=$ *baseadd* $+$ *SPattern*$_m[i]$
            SendBuffer$_q$[*Scount*$_q$] $= B_{loc}$[*address*]
            Scount$_q$ $=$ *Scount*$_q$ $+ 1$
        endif
    enddo
    baseadd $=$ *baseadd* $+ s_2 k$
enddo

address $=$ *baseadd*; $i = 0$
while (*address* $< max_B$)
    $q =$ *SProcpattern*$_m[i]$
    if $m \neq q$
        address $=$ *baseadd* $+$ *SPattern*$_m[i]$
        SendBuffer$_q$[*Scount*$_q$] $= B_{loc}$[*address*]

$$\quad\quad Scount_q = Scount_q + 1$$
endif
$i = i + 1$
endwhile

where max_B is the local address of the last element of B allocated on processor m that is accessed by the RHS reference on any processor.

The outer loop steps through separate blocks of length equal to the length of the communication pattern. The inner loop steps through the size of the *SProcPattern*$_m$ table. For each iteration of the inner loop we look up an entry in the *SProcPattern*$_m$ table to determine where the elements need to be sent. A similar look up into the *SPattern*$_m$ table will determine the local address of the element to be sent. The length of the outer loop is incremented each time by $baseadd$ which keeps track of the number of elements to be added to the *SPattern*$_m$ entry in order to index the right block. Communication between processors take place when all the data to be sent has been packed into the *SendBuffers*.

4.2 Receive-Execute Phase

The code for the receive-execute phase of execution of an array assignment statement in which accesses to non-local array elements are directly out of the buffers in which they are received by a processor is shown below. In this case we assume that *APattern*$_m$ stores the local access pattern for the array A and that the received elements are in *ReceiveBuffer*$_q$ for each q in $0 \leq q \leq p-1$. For processor m, we run the following code:

$Rcount_q = 0$, where $0 \leq q \leq p-1$
$baseadd_A = 0$; $baseadd_B = 0$
do $j = 0, \frac{max_A}{s_1 k}$
  do $i = 0, sizeof(RPattern_m) - 1$
    $q = RProcpattern_m[i]$
    $address = baseadd_A + Apattern_m[i]$
    if $m = q$
      $A_{loc}[address] = \mathcal{F}\left(B_{loc}[baseadd_B + RPattern_m[i]]\right)$
    else
      $A_{loc}[address] = \mathcal{F}\left(ReceiveBuffer_q[Rcount_q]\right)$
      $Rcount_q = Rcount_q + 1$
    endif
  enddo
  $baseadd_A = baseadd_A + s_1 k$
  $baseadd_B = baseadd_B + s_2 k$
enddo

$address = baseadd_A; i = 0$
while $(address < max_A)$
  $q = RProcpattern_m[i]$
  $address = baseadd_A + Apattern_m[i]$
  if m = q
    $A_{loc}[address] = \mathcal{F}\left(B_{loc}[baseadd_B + RPattern_m[i]]\right)$
  else
    $A_{loc}[address] = \mathcal{F}\left(ReceiveBuffer_q[Rcount_q]\right)$
    $Rcount_q = Rcount_q + 1$
  endif
  $i = i + 1$
endwhile

where max_A is the local address of the last element of A accessed on the processor.

In this case the receive pattern as discussed in Section 3.1 is used to determine whether the RHS element is local or non-local. If the element is local then the corresponding look-ups into the *APattern* and *RPattern* will determine the positions of both LHS and RHS element. If the element is non-local then it is fetched from the corresponding

ReceiveBuffer in order for the execution to take place. Hence this method constitutes the receive-execute phase. In this case, for each outer loop iteration, the index into the *Aloc* and *Bloc* is incremented by *baseadd_A* and *baseadd_B* which are used to step through the blocks on both LHS and RHS. Since the non-local elements are read directly from the buffer during execution, there is no unpacking overhead.

5 Generalized Overlap Regions for Block-Cyclic Distributions

Reading data directly out of buffers incurs no unpacking overhead but requires expensive conditionals to be executed in each iteration. In order to overcome this drawback, one can unpack the elements into overlap areas before execution. This unpacking of messages will not result in expensive conditionals and lead to improved locality of reference. In this section, we discuss two such strategies for unpacking the received elements into overlap regions.

Our algorithm unpacks the data from buffers into the local address space in order to exploit spatial locality. Since we provide storage for overlap regions, the notion of the local array has been extended; this results in a need to recompute the $RPattern_m$ and $SPattern_m$ tables. The overhead due to unpacking is significantly reduced because of the way in which the pattern table is used. Data locality is preserved because of the way we allocate storage for non-local accesses.

5.1 Course padding for overlap regions

As explained earlier, the communication pattern for an array assignment statement repeats after pk elements are accessed over the p processors. Hence the access pattern for the LHS array repeats after s_1 courses while that of the RHS array repeats after s_2 courses. On each processor we execute at most k local iterations before the communication pattern repeats, of which some RHS elements are local and others non-local. For each set of s_2k RHS elements, we allocate space equal to the number of non-local elements (referred to as *num_nonloc* in the following pages) in the receive index pattern. In the worst case every one of the k local iterations may require non-local RHS; thus, we might have to allocate space for at most k elements for each block of s_2 RHS courses. With course padding, after each set of s_2k local elements we allocate storage for holding non-local elements. Figure 2 shows an example of padding an array with courses. Note that padding courses in the RHS array results in a change in the addresses of the locally allocated portion of an array on a processor.

The algorithms for packing, unpacking and execute phases are shown below. We assume that the layout of array on the RHS takes into account the overlap regions. We also assume that the $RPattern_m$ and $SPattern_m$ tables have been recomputed for this type of storage. *num_nonloc* is the number of non-local elements (for each block of the communication pattern) to be padded in the new storage space before the communication pattern starts repeating. The value of *num_nonloc* is computed based on the information from the $RProcPattern_m$ table. Array elements received from processor q are in $ReceiveBuffer_q$ for each processor q where $0 \leq q \leq p-1$. *Apattern* stores the local access pattern of array A. Note that max_A is the local address of the last element of A accessed on processor m. Note that, in order to keep the presentation simple, we only present the algorithms for a full set of blocks; the code for the last incomplete block is similar to that discussed in Section 4.

In the packing phase (Step 1), each processor m scans its send patterns. For each non-local element i of $SPattern_m$, processor m packs all elements in the array section $(\text{SPattern}_m[i] : max_B : s_2k + \text{num\_nonloc})$ into contiguous locations in the *SendBuffer* corresponding to the processor that owns $SPattern_m[i]$.

Once all the elements to be sent have been packed communication takes place. We do not show the communication code here. At the receiving end each processor receives all the elements from other processors in the corresponding *ReceiveBuffers*. For each access on the LHS for the first repeating set we compute and record (Step 4) the corresponding entry from the new storage space of the RHS into a table of access called the $BPattern_m$ table. This table contains the sequence of accesses to both local and received data for the first repeating set; these entries are in lexicographic order. This table is constructed only once before the unpacking stage and does not require updates such as those required by the algorithm in [10].

The unpacking stage (Step 5) looks up the entries in the $BPattern_m$ table to determine where the incoming elements are to be unpacked. For each received element i of $BPattern_m$, processor m unpacks elements from the *ReceiveBuffers* to the array section $(\text{BPattern}_m[i] : max_B : s_2k + \text{num\_nonloc})$. The final stage (Step 6) involves execution wherein look-ups into the $APattern_m$ and $BPattern_m$ table will determine the location of elements on both LHS and RHS. Figure 3 illustrates course padding. The complete algorithm which performs packing, unpacking and execution on processor

m is shown below.

Step 1: Packing:

$Scount_q = 0$, where $0 \le q \le p-1$
$baseadd = 0$
$blksize_B = s_2 k + num\_nonloc$
do $i = 0, sizeof(SProcPattern_m) - 1$
  $q = SProcpattern_m[i]$
  if $m \neq q$
    $loc_B = SPattern_m[i]$
    do $j = 0, \frac{max_B}{blksize_B}$
      $address = baseadd + loc_B$
      $SendBuffer_q[Scount_q] = B_{loc}[address]$
      $Scount_q = Scount_q + 1$
      $baseadd = baseadd + blksize_B$
    enddo
  endif
enddo

Step 2: Send data in the *SendBuffers* to the respective processors.

Step 3: Receive data from the processors into the appropriate *ReceiveBuffers*

Step 4: Compute a table to store the locations of both local and non-local accesses:

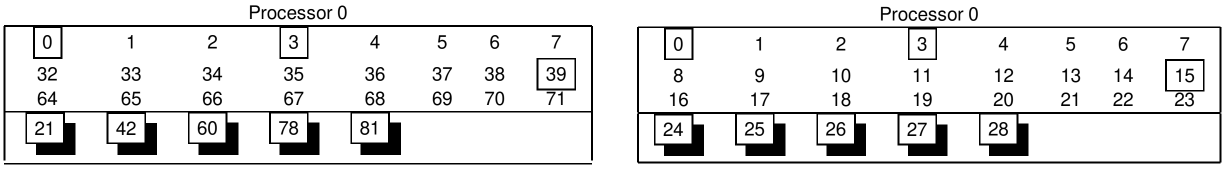
$index = s_2 k$
do $j = 0, sizeof(RProcPattern_m) - 1$
  $dest = RecvProcPattern_m[j]$
  if $(m = dest)$
    $BPattern[j] = RPattern_m[j]$
  else
    $BPattern[j] = index$
    $index = index + 1$
enddo

Step 5: Unpack:

$count_q = 0$ for $0 \le q \le p-1$
do $i = 0, sizeof(RProcPattern_m) - 1$
  $q = RProcPattern_m[i]$
  $loc_B = BPattern_m[i]$
  do $j = 0, \frac{max_A}{s_1 k}$
    $Bloc[loc_B] = ReceiveBuffer_q[count_q]$
    $count_q = count_q + 1$
    $loc_B = loc_B + s_2 k + num\_nonloc$
  endo
endo

Step 6: Execute:

$baseadd_A = baseadd_B = 0$
do $j = 0, \frac{max_A}{s_1 k}$



(a): Access pattern using global addresses. (b): Access pattern using local addresses.

Figure 2: RHS elements on processor 0 (using padded courses) for $A(0:\ :5) = B(0:\ :3)$, $p = 4$ and $k = 8$.

$$
\begin{aligned}
\mathit{SendBuffer}_1 &= \{6, 35, 64, 93, 122, 18, 47, 76, 105, 134\}\\
\mathit{SendBuffer}_2 &= \{9, 38, 67, 96, 125, 21, 50, 79, 108, 137\}\\
\mathit{SendBuffer}_3 &= \{12, 41, 70, 99, 128\}\\
\mathit{BPattern} &= \{0, 3, 24, 15, 25, 26, 27, 28\}\\
\text{UNPACKING:} &\\
B(24:144:29) &= \mathit{ReceiveBuffer}_2(0:4)\\
B(25:144:29) &= \mathit{ReceiveBuffer}_1(0:4)\\
B(26:144:29) &= \mathit{ReceiveBuffer}_3(0:4)\\
B(27:144:29) &= \mathit{ReceiveBuffer}_1(5:9)\\
B(28:144:29) &= \mathit{ReceiveBuffer}_2(5:9)
\end{aligned}
$$

Figure 3: Illustration of course padding on processor 0 for $A(0:795:5) = B(0:477:3)$, $p = 4$ and $k = 8$.

```
    do i = 0, sizeof(RProcPattern_m) − 1
        loc_A = APattern_m[j] + baseadd_A
        loc_B = BPattern_m[j] + baseadd_B
        Aloc[loc_A] = F (Bloc[loc_B])
    enddo
    baseadd_A = baseadd_A + s_1 k
    baseadd_B = baseadd_B + s_2 k + num_nonloc
enddo
```

5.2 Column padding for overlap regions

In case of padded columns we allocate local storage for non-local RHS elements that are accessed. Unlike the previous unpacking strategy, the extra local storage is to the right of the local address space. The total number of extra storage space that are padded to the RHS array is equal to the number of non-local elements being accessed. There can be storage for at most k extra elements before the pattern starts repeating. The location to which the received element is to be unpacked is dependent on the address of the element being received. Figure 4(a) and (b) shows an example of padding an array with columns.

We assume that the layout of array on the RHS takes into account the overlap regions. We also assume that the $\mathit{RPattern}_m$ and $\mathit{SPattern}_m$ tables have been recomputed for this type of storage. *num_nonloc* is the number of non-local elements (for each block of the communication pattern) to be padded in the new storage space before the communication pattern starts repeating. The value of *num_nonloc* is computed based on the information from the $\mathit{RProcPattern}_m$ table. Array elements received from processor q are in $\mathit{ReceiveBuffer}_q$ for each processor q where $0 \leq q \leq p-1$. *Apattern* stores the local access pattern of array A. Note that max_A is the local address of the last element of A accessed on processor m. Note that, in order to keep the presentation simple, we only present the algorithms for a full set of blocks; the code for the last incomplete block is similar to that discussed in Section 4.

The algorithm for packing (Step 1) is similar to the one discussed in Section 5.1. At the receiving end however, for each access on the LHS, based on the element received, we compute the *BPattern* table (Step 4) similar to the one constructed in Section 5.1 to record the sequence of accesses to both local and received data for the first repeating set; these entries are in lexicographic order.

The unpacking stage (Step 5) looks up the entries in the $BPattern_m$ table to determine where the incoming elements are to be unpacked. For each received element i of $BPattern_m$, processor m unpacks elements from the *ReceiveBuffers* to the array section $\left(\text{BPattern}_m[i] : max_B : s_2k + \text{num\_nonloc}\right)$. The final stage (Step 6) involves execution wherein look-ups into the $APattern_m$ and $BPattern_m$ table will determine the location of elements on both LHS and RHS. Figure 5 illustrates column padding. The complete algorithm which performs packing, unpacking and execution on processor m is shown below.

Step 1: Packing:

$Scount_q = 0$, where $0 \le q \le p-1$
$baseadd = 0$
$blksize_B = s_2k + num\_nonloc$
do $i = 0, sizeof(SProcPattern_m) - 1$
  $q = SProcpattern_m[i]$
  if $m \neq q$
    $loc_B = SPattern_m[i]$
    do $j = 0, \frac{max_B}{blksize_B}$
      $address = baseadd + loc_B$
      $SendBuffer_q[Scount_q] = B_{loc}[address]$
      $Scount_q = Scount_q + 1$
      $baseadd = baseadd + blksize_B$
    enddo
  endif
enddo

Step 2: Send data in the *SendBuffers* to the respective processors.

Step 3: Receive data from the processors into the appropriate *ReceiveBuffers*

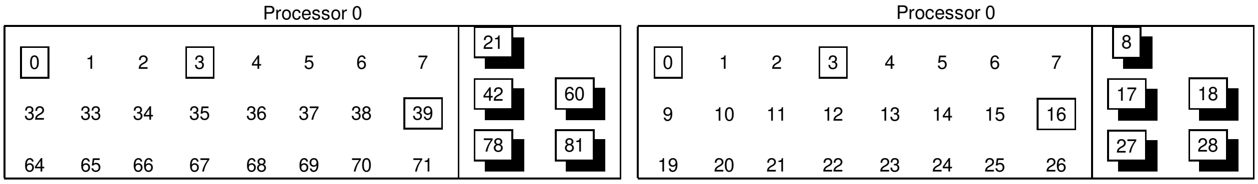
Step 4: Compute *BPattern* which is the table that stores local addresses of the RHS elements needed including elements received from other processors.

Step 5: Unpack:

$count_q = 0$ for $0 \le q \le p-1$
do $i = 0, sizeof(RProcPattern_m) - 1$
  $q = RProcPattern_m[i]$
  $loc_B = BPattern_m[i]$
  do $j = 0, \frac{max_A}{s_1k}$
    $Bloc[loc_B] = ReceiveBuffer_q[count_q]$
    $count_q = count_q + 1$
    $loc_B = loc_B + s_2k + num\_nonloc$
  endo
endo

Step 6: Execute:

$baseadd_A = baseadd_B = 0$



(a): Access pattern using global addresses. (b): Access pattern using local addresses.

Figure 4: RHS elements on processor 0 (using padded columns) for $A(0::5) = B(0::3)$, $p = 4$ and $k = 8$.

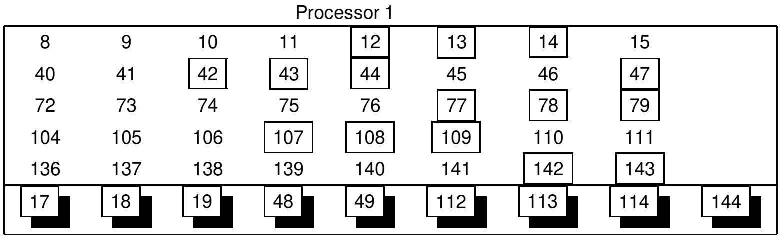
$$
\begin{aligned}
\mathit{SendBuffer}_1 &= \{6, 35, 64, 93, 122, 21, 50, 79, 108, 137\} \\
\mathit{SendBuffer}_2 &= \{10, 39, 68, 97, 126, 24, 53, 82, 111, 140\} \\
\mathit{SendBuffer}_3 &= \{13, 42, 71, 100, 129\} \\
\mathit{BPattern} &= \{0, 3, 8, 16, 17, 18, 27, 28\} \\
\text{UNPACKING:} & \\
B(8:144:29) &= \mathit{ReceiveBuffer}_2(0:4) \\
B(17:144:29) &= \mathit{ReceiveBuffer}_1(0:4) \\
B(18:144:29) &= \mathit{ReceiveBuffer}_3(0:4) \\
B(27:144:29) &= \mathit{ReceiveBuffer}_1(5:9) \\
B(28:144:29) &= \mathit{ReceiveBuffer}_2(5:9)
\end{aligned}
$$

Figure 5: Illustration of column padding on processor 0 for $A(0:795:5) = B(0:477:3)$, $p = 4$ and $k = 8$.

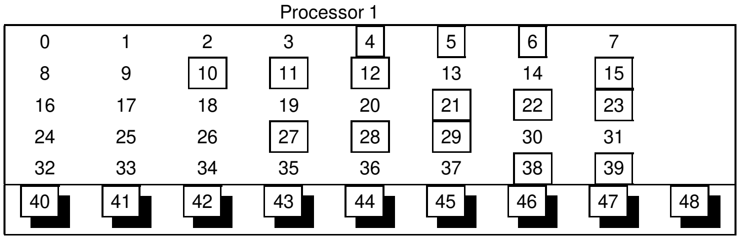
do $j = 0, \frac{max_A}{s_1 k}$
  do $i = 0, \mathit{sizeof}(\mathit{RProcPattern}_m) - 1$
    $\mathit{loc}_A = \mathit{APattern}_m[j] + \mathit{baseadd}_A$
    $\mathit{loc}_B = \mathit{BPattern}_m[j] + \mathit{baseadd}_B$
    $\mathit{Aloc}[\mathit{loc}_A] = \mathcal{F}\left(\mathit{Bloc}[\mathit{loc}_B]\right)$
  enddo
  $\mathit{baseadd}_A = \mathit{baseadd}_A + s_1 k$
  $\mathit{baseadd}_B = \mathit{baseadd}_B + s_2 k + \mathit{num\_nonloc}$
enddo

6 Message Optimizations

In this section, we discuss the use of message optimizations in code generation. We discuss three optimizations namely offset communication, message coalescing and aggregation. In case of multiple right hand side references to the same array, we can perform message coalescing optimization by sending only one copy of an element. We can also reduce the overhead of startup cost in communication by combining several messages—possibly due to different arrays referenced on the RHS—to be communicated between a sender and a receiver pair by performing message aggregation. We also discuss the idea of offset communication optimization which is when both LHS and RHS have the same distribution and access strides, by using the concept of overlap areas.



(a): Offset communication with padded courses: global address pattern.



(b): Offset communication with padded courses: local address pattern.

Figure 6: Access patterns for RHS elements on processor 1 (using offset communication using padded courses) for $A(0 : N-1 : 5) = \mathcal{F}\left(B(2 : N+1 : 5), B(3 : N+2 : 5), B(4 : N+3 : 5)\right)$, $p = 4$ and $k = 8$.

6.1 Offset Communication with Optimized Overlap Regions

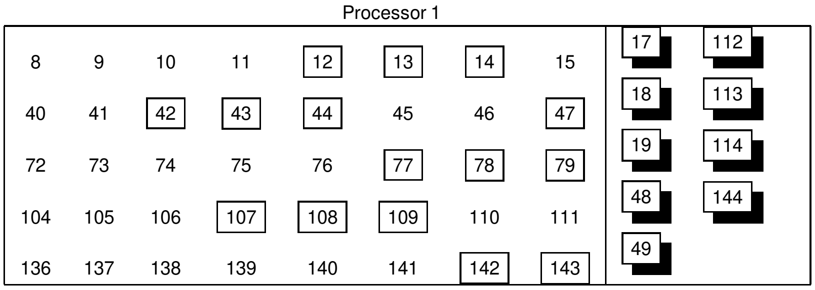
In case of offset communication the distribution and access strides of the left and right hand side arrays are identical but have different offsets. Offset communication is implemented using the idea of overlap areas and can be implemented by either course padding or column padding. Extra storage is allocated for only non-local elements and this leads to a change in the local access order (similar to that with course and column padding). As offset communication involves communication between only the neighboring virtual processors, the resulting code is very efficient.

Offset communication is highly effective if the offsets on the right hand side arrays are all less than k . If the offsets are greater than k, we have communication among virtual processors that are not neighbors rendering offset communication less effective. The amount of storage used depends mainly on the values of the offsets in the RHS arrays and the stride of the LHS array. If the offsets are all less than k and s, there are no repeating elements in the non-local accesses and hence one need not perform message coalescing on the elements to be communicated. On the other hand if any of the offsets is less than k but greater than s, there might be some repeating elements in the non-local accesses due to different terms on the RHS, and message coalescing may be useful here. Hence in this case the storage space is allocated based on elements received after performing message coalescing.
Consider the array statement

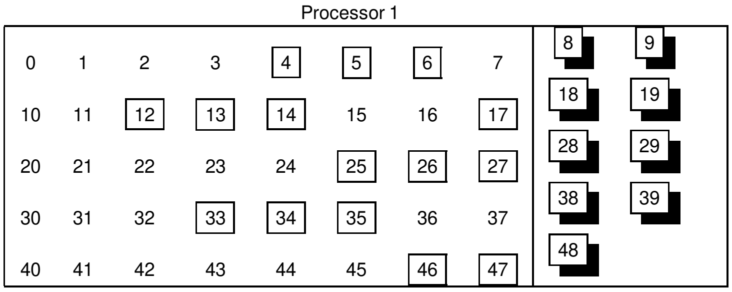
$$A(L_1 : H_1 : s) = \mathcal{F}\left(B(L_2+\theta_1 : H_2+\theta_1 : s), B(L_2+\theta_2 : H_2+\theta_2 : s), \cdots, B(L_2+\theta_r : H_2+\theta_r : s)\right)$$

where $|\theta_i| \leq k$ and $|\theta_i| \leq s$ (for $i = 1, \cdots, r$). With course padding, the number of extra memory storage padded for every s courses of local array B is $\sum_{i=1}^{r} |\theta_i|$. In the case of column padding, local storage allocated depends on the sign of the offset. If $\theta_i < 0$ the padding column is to the left of the storage for the course; if $\theta_i > 0$ the padding column is to the right. For each s courses of local storage for the array B, we need $\sum_{i=1}^{r} |\theta_i|$ units of extra storage for non-local references. Figures 6 and Figures 7 illustrate course and column padding respectively for the following array statement which requires offset communication.

$$A(0 : N : 5) = \mathcal{F}\left(B(2 : N+1 : 5), B(3 : N+2 : 5), B(4 : N+3 : 5)\right)$$



(a): Access pattern shown with global addresses for offset communication.



(b): Access pattern shown with local addresses for offset communication.

Figure 7: Access patterns for RHS elements on processor 1 (using offset communication using padded columns) for $A(0 : N-1 : 5) = \mathcal{F}\left(B(2 : N+1 : 5), B(3 : N+2 : 5), B(4 : N+3 : 5)\right)$, $p = 4$ and $k = 8$.

6.2 Message aggregation and coalescing

If there are references to multiple RHS arrays in a single statement, it is always better to combine all the messages going to the same processor into a single message and then send it. This optimization is called message aggregation and helps in reducing communication overhead. Message aggregation is then performed by aggregating the $SendBuffer[]_q$ (q is the receiving processor) for all the r RHS arrays on the right hand side. This results in a single message buffer being generated to be sent to a processor. In this case, the message header includes additional information that is necessary.

If there are multiple references to the same element of an array, then we can perform message coalescing and just send a single copy of the element. Here again, this optimization results in reduced communication volume. While performing message coalescing we have to keep track of the RHS array involved in sending or receiving the common element. Special message handling has to be provided at the receiving end in order to perform correct unpacking and execution.

7 Experiments

In order to verify the usefulness of overlap regions, we implemented all the above discussed algorithms and tested them for varying values of s_1, s_2 and k. Our execution model used the DAXPY operation

$$A(L_1 : H_1 : s_1) = A(L_1 : H_1 : s_1) + \texttt{constant} * B(L_2 : H_2 : s_2)$$

In all our experiments we fixed the number of iterations per processor to be 4096. This allows us to measure the effect of the strides and block sizes for a fixed amount of computation. We used 32 processors in the experiments. All the experiments were conducted on a single node of an IBM SP-2 using the `cc` compiler and `-O3` optimization flag turned on. All the reported times are in microseconds. We do not discuss the communication time between processors as it varies widely across configurations and all the three methods incur the same inter-processor communication cost.

Table 1: Communication packing, unpacking and execution time results (in microseconds) of our algorithms on an IBM SP-2 using the `cc` compiler and `-O3` optimization for the case $p = 32$ and number of iterations = 4096.

			Packing			Unpacking			Execution			Total		
k	s_1	s_2	buffer	column	course	buffer	column	course	buffer	column	course	buffer	column	course
4	3	5	603	126	126	-	126	126	2563	2061	2059	3166	2313	2311
	5	11	617	150	150	-	150	148	2579	2130	2157	3196	2430	2455
	9	7	612	136	136	-	135	135	2590	2145	2157	3202	2416	2428
	11	3	603	126	126	-	126	126	2596	2078	2076	3199	2330	2328
16	3	5	569	128	128	-	128	128	2502	1978	1955	3071	2234	2211
	5	11	588	155	155	-	156	138	2520	2041	1981	3108	2352	2274
	9	7	601	142	142	-	144	135	2538	2043	2023	3139	2329	2300
	11	3	569	129	129	-	128	128	2523	1980	1973	3092	2237	2230
64	3	5	562	139	139	-	138	137	2488	1935	1927	3050	2212	2203
	5	11	580	172	172	-	151	139	2496	1952	1943	3076	2275	2254
	9	7	570	154	154	-	152	142	2407	2017	1955	3077	2323	2251
	11	3	561	140	140	-	139	138	2521	1960	1953	3082	2239	2231
256	3	5	558	175	175	-	172	171	2478	1925	1923	3036	2272	2269
	5	11	579	212	212	-	179	171	2489	1923	1928	3068	2314	2311
	9	7	567	192	192	-	180	173	2404	1951	1943	3071	2323	2308
	11	3	559	181	181	-	174	173	2520	1957	1936	3079	2312	2290

Table 1 lists the time it takes for packing, unpacking and execution (excluding unpacking overhead) for all the three methods mentioned in the paper. The model where we read directly from the buffers during execution is referred to as *buffer* in Table 1. The methods using course padding and column padding schemes are referred to as *course* and *column* respectively. Since the *buffer* scheme does not perform unpacking, the entries are dashes. The total execution time in the table corresponds to the sum of packing, unpacking and pure execution for both *course* and *column* where as it corresponds to the sum of packing and pure execution for *buffer*.

The results show that the *course* and *column* methods perform well when compared to reading data out of buffers directly. Based on the results we see a 30% improvement in the total time when compared to the *buffer* method. The low times for packing and unpacking is due to the fact that we reduced the number of look-ups in the final SPMD code and also due to the choice of building a *BPattern* table which reduces expensive runtime conditionals and recalculation of tables. Therefore, using overlap regions appears to be beneficial in executing data parallel programs on distributed-memory message passing machines. In addition, unpacking of data into overlap regions improves data locality.

8 Conclusions

In this paper, we have dealt with communication issues in code generation for data-parallel programs with block-cyclic distribution executing on distributed-memory message passing architectures. We presented a new approach for the management of communication sets and strategies for local storage of remote references. We derived algorithms for determining communication patterns. We then presented two schemes that extend the notion of a local array by providing storage for non-local elements (called *overlap regions*) interspersed throughout the storage for the local portion. The two schemes, namely *course padding* and *column padding* enhance locality of reference significantly at the cost of a small overhead due to unpacking of messages. The performance of these schemes was found to be up to 30% better than the traditional buffer-based approach. In addition, several message optimizations such as offset communication, message aggregation and coalescing were discussed. Extension of these schemes to two-level mapping is rather straightforward. Work is in progress on further message optimization issues and communication generation for more complex subscript expressions.

References

[1] A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. To appear in *Scientific Programming,* 1996.

[2] S. Benkner. Handling block-cyclic distributed arrays in Vienna Fortran 90. In *Proc. International Conference on Parallel Architectures and Compilation Techniques,* Limassol, Cyprus, June 1995.

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

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

[5] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Rice University, December 1990.

[6] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience,* 2(3):171–193, September 1990.

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

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

[9] 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.

[10] K. Kennedy, N. Nedeljkovic, and A. Sethi. Communication generation for cyclic(k) distributions. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, B. Szymanski and B. Sinharoy (Eds.), Kluwer Academic Publishers, 1995.

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

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

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

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

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

[16] 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.

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

[18] A. Thirumalai and J. Ramanujam. An efficient compile-time approach to compute address sequences in data parallel programs. In *Proc. 5th International Workshop on Compilers for Parallel Computers,* Malaga, Spain, pages 581-605, June 1995.

[19] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs using integer lattices. In *Languages and Compilers for Parallel Computing,* P. Sadayappan *et al.* (Eds.), Lecture Notes in Computer Science, Springer-Verlag, 1996.

[20] A. Thirumalai, J. Ramanujam, and A. Venkatachar. Communication generation and optimization for HPF. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, B. Szymanski and B. Sinharoy (Eds.), Kluwer Academic Publishers, 1995.

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