Generalized Overlap Regions for Communication Optimization in Data-Parallel Programs

A. Venkatachar, J. Ramanujam, and A. Thirumalai

Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803-5901, USA
`http://www.ee.lsu.edu/jxr`
{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*

[1] 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.

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.

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, 22]. 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 `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 `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 $\frac{s}{\text{GCD}\,(s,pk)}$ courses of elements. Thus, the pattern of accesses in the first $\frac{s}{\text{GCD}\,(s,pk)}$ courses can be used to determine the rest of the local addresses accessed. Using a transformation $T = B^{-1}$ and Fourier-Motzkin elimination [13, 22], 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 $v = lower\ , upper$
    $loc\_pattern_m[len] = (\alpha_1 u + \beta_1 v)k + \alpha_2 u + \beta_2 v - mk$
    $glob\_pattern_m[len] = (\alpha_1 u + \beta_1 v)pk + \alpha_2 u + \beta_2 v$
    $len = len + 1$
  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 elements 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 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 `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 5.

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
    index_l = glob_LHS_pattern_m[j]
    RIndPattern_m[j] = index_r = (s_2/s_1)(index_l − L_1) + L_2
    RProcPattern_m[j] = (index_r div k) mod p
    if m = RProcPattern_m[j]
        RPattern_m[j] = (k(index_r div pk) + index_r mod 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.

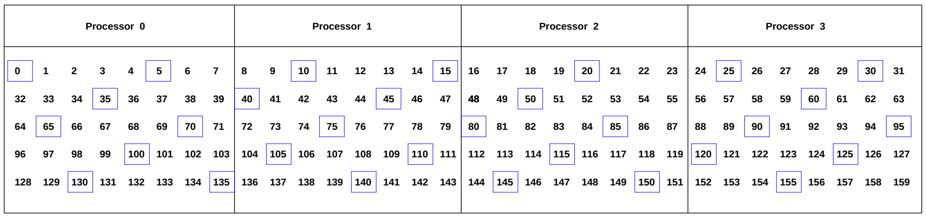
```
do j = 0, len_r − 1
    index_r = glob_RHS_pattern_m[j]
    SIndPattern_m[j] = index_l = (s_1/s_2)(index_r − L_2) + L_1
    SProcPattern_m[j] = (index_l div k) mod p
    if m ≠ SProcPattern_m[j]
        SPattern_m[j] = loc_RHS_pattern_m[j]
    endif
enddo
```

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 $k = 8$. The elements accessed on the LHS and RHS on all processors is shown in Figure 1. Tables shown below indicate entries in the communication data structures, namely the pattern tables for processor 0.

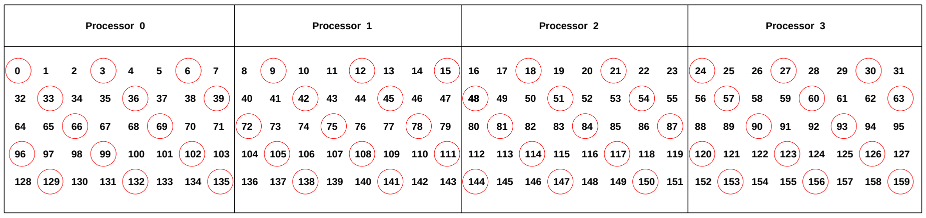
$RIndPattern_0$	0	3	21	39	42	60	78	81
$RProcPattern_0$	0	0	2	0	1	3	1	2
$RPattern_0$	0	3		15				

$SIndPattern_0$	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

Entries in the $RIndPattern_0$ and $RPattern_0$ tables indicate the global and local indices of the elements on RHS which are needed by the LHS before the communication pattern starts repeating. Some entries in the $RPattern_0$ are not computed as the algorithms to be discussed in later sections do not require these elements. Similarly $SIndPattern_0$ and $SPattern_0$ tables stores the global and local indices of the elements on RHS which are accessed before the communication pattern starts repeating. Some entries in the $SPattern_0$ table are also not computed due to the same reasons as mentioned above. $RProcPattern_0$ table 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. These pattern tables are needed to generate the communication algorithms to be discussed in the next section.



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



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

Fig. 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 Generalized Overlap Regions for Block-Cyclic Distributions

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

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 $\mathit{RPattern}_m$ and $\mathit{SPattern}_m$ 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.

4.1 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. For the example in Figure 1 the LHS pattern repeats after 5 courses, where as the RHS pattern repeats after 3 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. The reason for allocating extra storage after each block of s_2 courses on the RHS is to increase locality during execution. 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 for lexicographic execution are shown below. 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 simple and is not discussed here.

Let us consider Processor 0 from Figure 1. We have shown the computation of pattern tables for Processor 0 in Example 3.2. The array on the RHS needs to pack its elements into the $\mathit{ReceiveBuffers}_q$ where q is the index into the $\mathit{SProcPattern}_0$ table. This constitutes the packing phase (Step 3), where in, processor 0 scans its send patterns and for each non-local element i of $\mathit{SPattern}_0$, processor 0 packs all elements in the array section $\left(\mathrm{SPattern}_0[i] : max_B : s_2k + \mathrm{num\_nonloc}\right)$ into contiguous locations in the *SendBuffer* corresponding to the processor that owns $\mathit{SPattern}_0[i]$. This necessarily means that we make as many scans of the local space as the number of non-local elements before we pack all elements. Figure 3 shows the set of elements which are packed into *SendBuffers*. The array section involved in packing the first element to be sent to Processor 1 is (6 : 477 : 29), where 29 is the total number of elements including the space allocated for the 5 non-local elements before the pattern starts repeating.

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 2) 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]. For the above example the entries held by the *BPattern*$_m$ table are shown in Figure 3. Entries (0, 3, 15) correspond to local elements where as (24, 25, 26, 27, 28) correspond to non-local elements. These entries correspond to the first repeating set of communication pattern. Step 2 illustrates the code to compute the table. The code records consecutive locations into *BPattern*$_m$ table if the element is non-local, since course padding schemes allocates consecutive storage for non-locals.

The unpacking stage (Step 6) 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 $\left(\text{BPattern}_m[i] : max_B : s_2k + \text{num\_nonloc}\right)$. Figure /reffig:coupadex shows the array sections of memory into which the elements in the *ReceiveBuffers* have been unpacked. Here again we make as many scans of the local space as the number of non-local elements. Figure 3 shows the array sections into which consecutive elements from the *ReceiveBuffers* are unpacked.

Once all the elements are unpacked into the local space, the final stage (Step 7) involves lexicographic 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 local addresses of both LHS and RHS which are accessed during execution using course padding scheme. All the adresses are in lexicographic order. The complete algorithm which performs packing, unpacking and execution on processor m is shown below.

Step 1: The entire local space is assumed to be reorganized in order to account for the overlap regions. We allocate extra storage equal to the number of non-local elements to be received.

Step 2: Code to compute a table to store the locations of both local and non-local accesses on RHS. This table will be used during both unpacking and execution.

$$
\begin{array}{l}
\textit{index} = s_2 k \\
\textbf{do } j = 0, \textit{sizeof}(\textit{RProcPattern}_m) - 1 \\
\quad dest = \textit{RecvProcPattern}_m[j] \\
\quad \textbf{if } (m = dest) \\
\quad\quad \textit{BPattern}[j] = \textit{RPattern}_m[j] \\
\quad \textbf{else} \\
\quad\quad \textit{BPattern}[j] = \textit{index} \\
\quad\quad \textit{index} = \textit{index} + 1 \\
\textbf{enddo}
\end{array}
$$

Step 3: Packing: Involves packing of elements to be sent into the respective *SendBuffers*. Pattern tables discussed in Section 3.2 are used to pack these elements.

$$
\begin{array}{l}
\textit{Scount}_q = 0, \text{ where } 0 \leq q \leq p-1 \\
\textit{baseadd} = 0 \\
\textit{blksize}_B = s_2 k + \textit{num\_nonloc} \\
\textbf{do } i = 0, \textit{sizeof}(\textit{SProcPattern}_m) - 1 \\
\quad q = \textit{SProcpattern}_m[i] \\
\quad \textbf{if } m \neq q \\
\quad\quad loc_B = \textit{SPattern}_m[i] \\
\quad\quad \textbf{do } j = 0, \frac{max_B}{\textit{blksize}_B} \\
\quad\quad\quad \textit{address} = \textit{baseadd} + loc_B \\
\quad\quad\quad \textit{SendBuffer}_q[\textit{Scount}_q] = B_{loc}[\textit{address}] \\
\quad\quad\quad \textit{Scount}_q = \textit{Scount}_q + 1
\end{array}
$$

$baseadd = baseadd + blksize_B$
enddo
endif
enddo

Step 4: Send data in the *SendBuffers* to the respective processors.
Step 5: Receive data from the processors into the appropriate *ReceiveBuffers*
Step 6: Unpack: Involves unpacking non-local elements into the overlap regions allocated in the local space.

$count_q = 0$ for $0 \leq q \leq p-1$
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

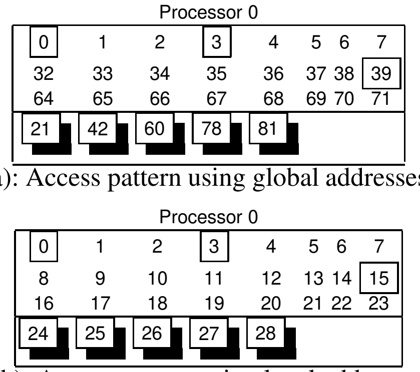
Step 7: Execute: Involves lexicographic order of execution.

$baseadd_A = baseadd_B = 0$
do $j = 0, \frac{max_A}{s_1 k}$
do $i = 0, sizeof(RProcPattern_m) - 1$
$loc_A = APattern_m[j] + baseadd_A$
$loc_B = BPattern_m[j] + baseadd_B$
$Aloc[loc_A] = \mathcal{F}\left(Bloc[loc_B]\right)$
enddo
$baseadd_A = baseadd_A + s_1 k$
$baseadd_B = baseadd_B + s_2 k + num\_nonloc$
enddo

4.2 Column padding for overlap regions

In case of column padding scheme, we also 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 $RPattern_m$ 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



(a): Access pattern using global addresses.

(b): Access pattern using local addresses.

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

SendBuffer_1	6 35 64 93 122 18 47 76 105 134
SendBuffer_2	9 38 67 96 127 21 50 79 108 137
SendBuffer_3	12 41 70 99 128

BPattern	0 3 24 15 25 26 27 28

B(24:144:29) = $\textit{ReceiveBuffer}_2(0:4)$
B(25:144:29) = $\textit{ReceiveBuffer}_1(0:4)$
B(26:144:29) = $\textit{ReceiveBuffer}_3(0:4)$
B(27:144:29) = $\textit{ReceiveBuffer}_1(5:9)$
B(28:144:29) = $\textit{ReceiveBuffer}_2(5:9)$

LHS	0 5 11 17 22 28 34 39 40 45 51 57 62 68 74 79 $\cdots\cdots$
RHS	0 3 24 15 25 26 27 28 29 32 53 44 54 55 56 57 $\cdots\cdots$

Fig. 3. Illustration of packing, unpacking and execution on processor 0 for $A(0 : 795 : 5) = B(0 : 477 : 3)$, $p = 4$ and $k = 8$.

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 simple and is not discussed here.

The algorithm for packing (Step 3) is exactly similar to the one discussed in Section 4.1. It is only the unpacking and execution which is different from that of course padding scheme. Figure 5 shows the elements padded into the *SendBuffers*.

At the receiving end however, we compute a table of access (Step 2) similar to the *BPattern* table as discussed in the previous section for the RHS to record the sequence of accesses to both local and received data for the first repeating set; these entries are in lexicographic order. The accesses computed, depends on the element to be received; whose information can be found from the pattern tables discussed earlier. Figure 5 shows the entries held by the *BPattern* table.

The unpacking stage (Step 6) 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 $\big(\text{BPattern}_m[i] : max_B : s_2k + \text{num\_nonloc}\big)$. The unpacking strategy is similar to the one discussed in the previous section. However the array sections involved in unpacking are different when compared to course padding. Figure 5 shows the array sections into which the received elements are unpacked. The entries in the *ReceiveBuffers* are exactly the same as in course padding.

The final stage (Step 7) 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 local addresses of both LHS and RHS which are accessed during execution using columnpadding scheme. The complete algorithm which performs packing, unpacking and execution on processor m is shown below.

Step 1: The entire local space is assumed to be reorganized in order to account for the overlap regions. We allocate extra storage equal to the number of non-local elements to be received.

Step 2: Compute *BPattern* which is the table that stores local addresses of the RHS elements needed including elements received from other processors.

Step 3: Packing: Involves packing of elements to be sent into the respective *SendBuffers*. Pattern tables discussed in Section 3.2 are used to pack these elements.

$Scount_q = 0$, where $0 \leq q \leq p-1$
$baseadd = 0$
$blksize_B = s_2k + num\_nonloc$
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 3: Send data in the *SendBuffers* to the respective processors.

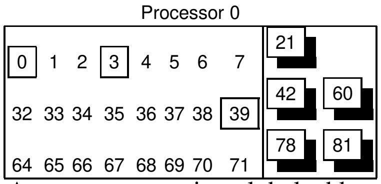
Step 4: Receive data from the processors into the appropriate *ReceiveBuffers*

Step 5: Unpack: Involves unpacking non-local elements into the overlap regions allocated in the local space.

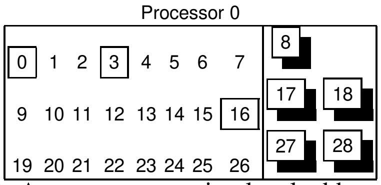
$count_q = 0$ for $0 \leq q \leq p-1$
do $i = 0, sizeof(RProcPattern_m) - 1$
  $q = RProcPattern_m[i]$
  $loc_B = BPattern_m[i]$
  do $j = 0, \frac{max_A}{s_1k}$
    $Bloc[loc_B] = ReceiveBuffer_q[count_q]$
    $count_q = count_q + 1$
    $loc_B = loc_B + s_2k + num\_nonloc$
  endo
endo

Step 6: Execute: Involves lexicographic order of execution.

$baseadd_A = baseadd_B = 0$
do $j = 0, \frac{max_A}{s_1 k}$
  do $i = 0, sizeof(RProcPattern_m) - 1$
    $loc_A = APattern_m[j] + baseadd_A$
    $loc_B = BPattern_m[j] + baseadd_B$
    $Aloc[loc_A] = \mathcal{F}\left(Bloc[loc_B]\right)$
  enddo
  $baseadd_A = baseadd_A + s_1 k$
  $baseadd_B = baseadd_B + s_2 k + num\_nonloc$
enddo



(a): Access pattern using global addresses.



(b): Access pattern using local addresses.

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

5 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.

5.1 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

$SendBuffer_1$	6 35 64 93 122 21 50 79 108 137
$SendBuffer_2$	10 39 68 97 126 24 53 82 111 140
$SendBuffer_3$	13 42 71 100 129

BPattern	0 3 8 16 17 18 27 28

B(8:144:29) = $ReceiveBuffer_2(0:4)$
B(17:144:29) = $ReceiveBuffer_1(0:4)$
B(18:144:29) = $ReceiveBuffer_3(0:4)$
B(27:144:29) = $ReceiveBuffer_1(5:9)$
B(28:144:29) = $ReceiveBuffer_2(5:9)$

LHS	0 5 11 17 22 28 34 39 40 45 51 57 62 68 74 79 $\cdots\cdots$
RHS	0 3 8 16 17 18 27 28 29 32 37 45 46 47 56 57 $\cdots\cdots$

Fig. 5. Illustration of packing, unpacking and execution on processor 0 for $A(0:795:5) = B(0:477:3)$, $p = 4$ and $k = 8$.

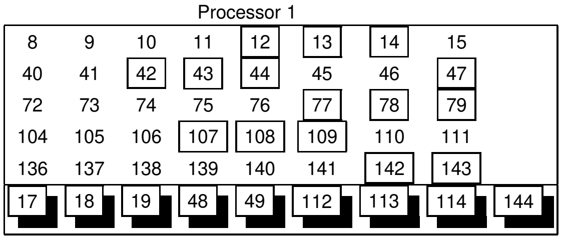
implemented using the idea of overlap areas and can be implemented by either course padding or column padding. Extra storage is allocated for only non-local elements and this leads to a change in the local access order (similar to that with course and column padding). As offset communication involves 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

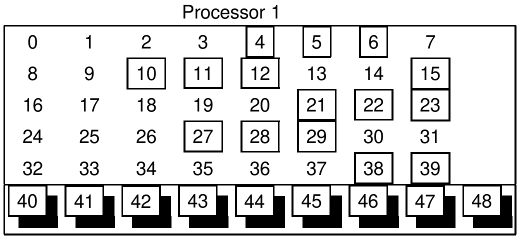
$$A(L_1 : H_1 : s) = \mathcal{F}\left(B(L_2 + \theta_1 : H_2 + \theta_1 : s), \cdots, B(L_2 + \theta_r : H_2 + \theta_r : s)\right)$$

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



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

Fig. 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 $k = 8$.

5.2 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.

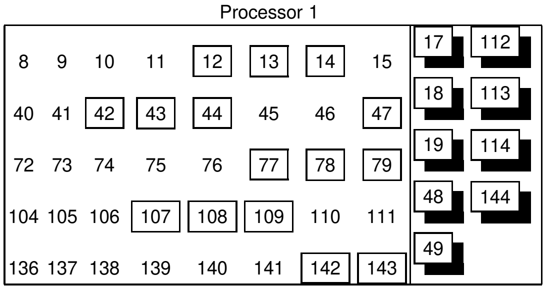
If there are multiple references to the same element of an array, then we can perform message coalescing and just send a single copy of the element. Here again, this optimization results in reduced communication volume. While performing message coalescing we have to keep track of the RHS array involved 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.

6 Experiments

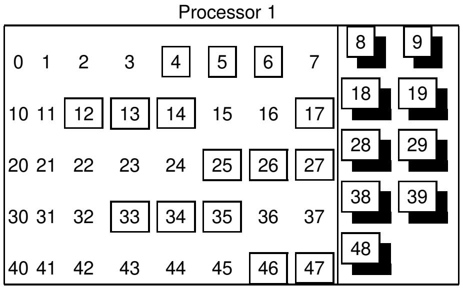
In order to verify the usefulness of overlap regions, we implemented all the above discussed algorithms and tested them for varying values of s_1, s_2 and k. Our execution model used the DAXPY operation

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

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



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



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

Fig. 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 $k = 8$.

---

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 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.

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.

k	s_1	s_2	Packing			Unpacking			Execution			Total		
			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

7 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. A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing,* 38(2):188–203, November 1996.
22. M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Publishing Co., Redwood City, CA, 1996.