Parallel Processing Letters,

COMMUNICATION GENERATION FOR BLOCK-CYCLIC DISTRIBUTIONS

A. VENKATACHAR, J. RAMANUJAM, A. THIRUMALAI
Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Data-parallel languages such as High Performance Fortran, Vienna Fortran and Fortran D include directives such as alignment and distribution that describe how data and computation are mapped onto the processors in a distributed-memory multiprocessor. A compiler for HPF 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. In this paper, we present a novel approach for the generation of communication sets that exploits a pattern of send-receive index pairs. In addition, we present an algorithm for code generation. Experimental results demonstrate the viability of this technique.

Keywords: communication, code generation, data mapping, HPF, block-cyclic distribution

1. Introduction

Programming massively parallel distributed memory machines involves partitioning data and computation across processors. Languages such as Fortran D [5], Vienna Fortran [2] and High Performance Fortran [7] include mapping directives that describe how data is mapped onto the processors in a distributed-memory multiprocessor. In these languages, arrays are first 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 local memory addresses accessed by each processor and the sequence of sends and receives for a given processor to access non-local data.

Using the fact that the sequence of accesses form an integer lattice [8,17,19], the set of array elements referenced can be generated by integer linear combinations of basis vectors of that lattice. This also allows us to derive the communication sets of each processor. These algorithms exploit the repeating communication pattern. We also show how to generate the sends for each processor and propose a method to directly read and execute the received elements from the communication buffers.

This paper is organized as follows. Section 2 discusses related work. In Section 3 we briefly review the problem of address generation algorithm based on integer lattices. Section 4 presents algorithms to generate the necessary communication sets. We also present algorithms to generate send, receive and execution code. In Section 5 we discuss experimental results. Section 6 concludes with a discussion.

2. Related Work

Several papers have addressed the code generation problem for distributed memory machines [1,3,6,8,9,13,14,15,16,17,18,19,20,21,22]. Many of these papers have developed good schemes for

$s' = \frac{s}{gcd(s,pk)}$; $q = 0$

DO $u = \left\lceil \frac{(-mk-k+1)\beta_1}{s} \right\rceil, \left\lfloor \frac{(s'-1)\beta_2 - mk\beta_1}{s} \right\rfloor$

  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

    $lower = \left\lceil \max\left(\frac{mk-u\alpha_2}{\beta_2}, \frac{-u\alpha_1}{\beta_1} \right) \right\rceil$; $upper = \left\lfloor \min\left(\frac{mk+k-1-u\alpha_2}{\beta_2}, \frac{s'-1-u\alpha_1}{\beta_1} \right) \right\rfloor$

  endif

  DO $v = lower\ ,\ upper$

    $pattern_m[q] = k(\alpha_1 u + \beta_1 v - m) + \alpha_2 u + \beta_2 v$

    $q = q + 1$

  ENDDO

ENDDO

Fig. 1. Constructing the access pattern for $A(0 : H : s)$ for processor m

address generation but issues in communication optimization have not received much attention. Chatterjee *et al.* [3] 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. Stichnoth [14] and Gupta *et al.* [6] discuss similar schemes for communication.

Kennedy *et al.* [9] view the set of accesses as an integer lattice and make use of the repeating set of accesses. In their communication method, each processor allocates overflow regions at the end of local storage (for each array) for storing non-local references. The access table for each right hand side operand needs to be recomputed for every k local computations where k is the block size. In effect a processor that performs N local computations computes a total of N addresses for each right hand side reference, computed using two tables of length k each. This results in additional runtime overhead. Communication generation issues are also dealt with in [15,18,20,21].

3. Brief 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 [7]. 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.

In [17], we 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. Using the basis vectors, we derive a loop nest that enumerates the addresses. 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 in [17] that the determinant of B is the access stride s itself. Using a transformation $T = B^{-1}$ and Fourier-Motzkin elimination [10,11,22], we derive the code shown in Figure 1. See [17] for details. The access pattern for the array A repeats after every $s' = s/\gcd(s, pk)$ courses of elements. Thus, the pattern of accesses in the first s' courses can be used to determine the rest of the local addresses accessed. The code for $A(L : H : s)$ is easily obtained by using table lookups using this pattern [17,4]. In addition, code generation for two-level mapping is discussed in [12,20].

$s_1' = \frac{s_1}{gcd(s_1, pk)}$; $q = 0$
DO $u = \left\lceil \frac{(-mk-k+1)\beta_1^l}{s_1} \right\rceil, \left\lfloor \frac{(s_1' f_l - 1)\beta_2^l - mk\beta_1^l}{s_1} \right\rfloor$
  if $\beta_1^l = 0$
    $lower = \left\lceil \left(\frac{mk - u\alpha_2^l}{\beta_2^l} \right) \right\rceil; upper = \left\lfloor \left(\frac{mk+k-1-u\alpha_2^l}{\beta_2^l} \right) \right\rfloor$
  else
    $lower = \left\lceil \max\left(\frac{mk-u\alpha_2^l}{\beta_2^l}, \frac{-u\alpha_1^l}{\beta_1^l} \right) \right\rceil; upper = \left\lfloor \min\left(\frac{mk+k-1-u\alpha_2^l}{\beta_2^l}, \frac{s_1' f_l - 1 - u\alpha_1^l}{\beta_1^l} \right) \right\rfloor$
  endif
  DO $v = lower, upper$
    $i = \frac{((\alpha_1^l u + \beta_1^l v) * pk + \alpha_2^l u + \beta_2^r v)}{s_1} s_2$
    $RProcpattern_m[q] = (i \text{ div } k) \text{ mod } p$
    if $m = RProcpattern_m[q]$
      $RPattern_m[q] = (k(i \text{ div } pk) + i \text{ mod } k)$
    endif
    $q = q + 1$
  ENDDO
ENDDO

(a): Code for computing the receive pattern for processor m

$s_2' = \frac{s_2}{gcd(s_2, pk)}$; $q = 0$
DO $u = \left\lceil \frac{(-mk-k+1)\beta_1^r}{s_2} \right\rceil, \left\lfloor \frac{(s_2' f_r - 1)\beta_2^r - mk\beta_1^r}{s_2} \right\rfloor$
  if $\beta_1^r = 0$
    $lower = \left\lceil \left(\frac{mk - u\alpha_2^r}{\beta_2^r} \right) \right\rceil; upper = \left\lfloor \left(\frac{mk+k-1-u\alpha_2^r}{\beta_2^r} \right) \right\rfloor$
  else
    $lower = \left\lceil \max\left(\frac{mk-u\alpha_2^r}{\beta_2^r}, \frac{-u\alpha_1^r}{\beta_1^r} \right) \right\rceil; upper = \left\lfloor \min\left(\frac{mk+k-1-u\alpha_2^r}{\beta_2^r}, \frac{s_2' f_r - 1 - u\alpha_1^r}{\beta_1^r} \right) \right\rfloor$
  endif
  DO $v = lower, upper$
    $i = \frac{((\alpha_1^r u + \beta_1^r v) * pk + \alpha_2^r u + \beta_2^r v)}{s_2} s_1$
    $SProcpattern_m[q] = (i \text{ div } k) \text{ mod } p$
    $SPattern_m[q] = (k(i \text{ div } pk) + i \text{ mod } k)$
    $q = q + 1$
  ENDDO
ENDDO

(b): Code for computing the send pattern for processor m

Fig. 2. Generation of communication patterns for $A(0 : H_1 : s_1) = B(0 : H_2 : s_2)$

4. Communication Set Generation

Communication involves sending those data items that a given processor owns and is required by other processors, and receiving data sent by other processors. We first show that there is a pattern present in the communication sets of each processor. We then present algorithms for deriving both the send pattern and receive pattern for each processor. These communication patterns are used to derive both the send and receive sets for each processor. We will discuss communication generation for the following array statement with HPF-style data mappings.

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

where arrays A and B are distributed using a $\texttt{CYCLIC}(k)$ distribution over p processors. Extension to cases with different block sizes for the arrays as well as a different number of processors over which the arrays are distributed is straightforward and are not discussed here.

4.1. Receive and Send Pattern Generation

Given the global address of a lhs element, we can easily determine the processor that owns the corresponding rhs element and the local address of the rhs element on that processor using the expressions,

$$m = (i \text{ div } k) \text{ mod } p \qquad\qquad Loc = k * (i \text{ div } pk) + i \text{ mod } k$$

where i is a global address, m is the processor to which i belongs, and Loc is the local address of i on processor m.

If s_1 and s_2 are the strides of the lhs and rhs respectively then the access pattern in the global address space repeats after $\frac{s_1}{gcd(s_1,pk)}$ and $\frac{s_2}{gcd(s_2,pk)}$ respectively. However the communication pattern (pair of processors that communicate) repeats itself after $lcm(\frac{s_1}{gcd(s_1,pk)}, \frac{s_2}{gcd(s_2,pk)})$ courses. In order to use the access patterns for generating the communication pattern we need to extend both the lhs and rhs access patterns by a factor f_l and f_r where $f_l = \frac{lcm(\frac{pk}{gcd(s_1,pk)}, \frac{pk}{gcd(s_2,pk)})}{\frac{pk}{gcd(s_1,pk)}}$ and $f_r = \frac{lcm(\frac{pk}{gcd(s_1,pk)}, \frac{pk}{gcd(s_2,pk)})}{\frac{pk}{gcd(s_2,pk)}}$ respectively. Hence the new set of courses on lhs and rhs which hold the communication pattern are $\frac{s_1}{gcd(s_1,pk)} f_l$ and $\frac{s_2}{gcd(s_2,pk)} f_r$ respectively.

In order to generate the receive set, we need to run the code generated in Figure 2(a) to obtain the global address of the elements accessed before the communication pattern starts repeating. For each of these elements we calculate the processor that owns the corresponding rhs element and the local address of that element.

Our approach to the send set generation is very similar to the receive set generation discussed above. We make use of the fact that there is a one-to-one correspondence between the lhs and rhs array accesses. Thus, the receive pattern for a processor m in the array statement $A(L_1 : H_1 : s_1) = B(L_2 : H_2 : s_2)$ is the same as the send pattern for that processor for the array statement $B(L_2 : H_2 : s_2) = A(L_1 : H_1 : s_1)$. The code that generates the send pattern is shown in Figure 2(b). The send and receive pattern information are needed to generate the communication algorithms as discussed in the next section.

4.2. Communication Algorithm

Communication between processors involves packing of data before being sent from one processor to another. At the receive end, the processor can either unpack the elements before executing the code or can read directly from the receive buffers during execution. The method we use involves no unpacking of data and assumes that each processor maintains separate send and receive buffers for all the p processors.

- Run the Send loop as shown in Figure 2(b). The elements generated by this loop are stored in $SPattern_m$, the pattern of the processors to which these elements have to be sent is stored in $SProcpattern_m$, and the array elements to be sent to processor i are stored in $SendBuffer_i$ for each processor $i = 0, p-1$.
- For processor m, we run the following code:

  $Scount_q = 0$, where $0 \le q \le p-1$ and $Scount_q$ is an offset into $SendBuffer_q$ corresponding to the q^{th} processor.

  $offset = 0$
  DO $j = 0, \frac{max_2}{\frac{s_2}{gcd(s_2,pk)} f_r k}$
      DO $i = 0, sizeof(SPattern_m) - 1$
          $q = SProcpattern_m[i]$
          if $m \ne q$
              $SendBuffer_q[Scount_q] = B_{loc}[offset + SPattern_m[i]]$
              $Scount_q = Scount_q + 1$
          endif
      ENDDO
      $offset = offset + \frac{s_2}{gcd(s_2,pk)} f_r k$
  ENDDO

  where max_2 is the local address of the last element of B accessed on the processor.
- Send data in buffers to the respective processors.

Fig. 3. The send phase for $A(L_1 : H_1 : s_1) = B(L_2 : H_2 : s_2)$ on proc. m.

In the code for the send phase, each processor scans the entries in the $SProcpattern_m$ table to determine where the elements in the $SendPattern_m$ need to be sent and then pack the elements into the corresponding *SendBuffer*. Communication between processors take place when all the data to be sent has been packed into the *SendBuffers*. The code is shown in Figure 3.

The communication algorithm for an array assignment statement where accesses to non-local array elements are directly out of the buffers in which they are received by a processor is shown in Figure 4. For code execution this algorithm reads non-local elements out of the receive buffers and local elements from the information obtained from *RPattern* table. The overhead due to unpacking is completely avoided.

The algorithms shown in this paper can be simplified for these cases as well as the case where s_1 divides s_2 or vice-versa.

5. Experiments

Table 1 shows the times it takes for our method for generating the tables for address sequence accesses as compared to the methods from that of Gupta *et al.* [6] (referred to as OSU) and Chatterjee *et al.* [3] (referred to as RIACS). The OSU method uses a virtual processor approach [6]. Their method distributes the data onto virtual processors and then the maps the virtual processors onto physical processors. The RIACS method solves the problem of address generation by constructing a finite state machine for repeating set of accesses. The results from this table show a significant improvement in our method which uses the integer lattice approach for accessing elements.

In Tables 2 and 3 we compare our implemetation that performs packing, and execution to OSU [6] for two seperate cases. The tables show the time taken for packing, and execution for different sizes of array (N). The times obtained for these cases are significantly better than that obtained for OSU for all N. The overhead OSU incurs is due to the mapping of data from the virtual processors

- Receive data from the p processors
- Run the Receive loop as shown in Figure 2(a). We assume that elements received in this loop are stored in $RPattern_m$ and array elements received from processor i is stored in $ReceiveBuffer_i$ for each processor $i = 0, p-1$. $Apattern_m$ holds A's access pattern.
- For processor m, we run the following code:

$Rcount_q = 0$, where $0 \le q \le p-1$ and $Rcount_q$ is an offset into $ReceiveBuffer_q$ corresponding to the q^{th} processor.

$offset_1 = 0 \quad offset_2 = 0$

DO $j = 0, \frac{max_1}{\frac{s_1}{gcd(s_1,pk)} f_l k}$
  DO $i = 0, sizeof(RPattern_m) - 1$
    $q = RProcpattern_m[i]$
    if m = q
      $A_{loc}[offset_1 + Apattern_m[i]] = B_{loc}[offset_2 + RPattern_m[i]]$
    else
      $A_{loc}[offset_1 + Apattern_m[i]] = ReceiveBuffer_q[Rcount_q]$
      $Rcount_q = Rcount_q + 1$
    endif
  ENDDO
  $offset_1 = offset_1 + \frac{s_1}{gcd(s_1,pk)} f_l k$
  $offset_2 = offset_2 + \frac{s_2}{gcd(s_2,pk)} f_r k$
ENDDO

where max_1 is the local address of the last element of A accessed on the processor.

Fig. 4. The receive-execute phase for $A(L_1 : H_1 : s_1) = B(L_2 : H_2 : s_2)$ on processor m.

to the physical processors. All the codes were compiled on an IBM SP-2 using the `cc` compiler with `-O3` optimizations turned on.

6. Summary

In this paper we present techniques to generate communication sets using the integer lattice approach. We make use of the communication pattern present in communication to generate these sets for block-cyclic distributions. We also provide algorithms to pack the elements as well as to receive and execute. The method presented in this paper can be extended to handle shift communication and optimizations such as message aggregation and coalescing. Experiments show that our method is useful for communication generation. Further work on using overlap regions appear in [20,21]. Work is in progress for communication generation for complex subscripts.

Acknowledgements

The research reported in this paper is 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.

References

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

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

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

[4] S. Dutta and J. Ramanujam. Data structures for efficient execution of data-parallel programs with block-cyclic distributions. In *Proc. 6th International Workshop on Compilers for Parallel Computers,* Aachen, Germany, December 1996.

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

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

[8] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[9] 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.

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

[11] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing,* 9(4):365–389, 1995.

[12] J. Ramanujam, A. Venkatachar, and S. Dutta. Efficient address sequence generation for two-level mappings in HPF. Submitted for publication, 1996.

[13] C. van Reeuwijk, H.J. Sips, W. Denissen, and E. M. Paalvast. Implementing HPF distributed arrays on a message-passing parallel computer system. Computational Physics TR 9506, Delft University of Technology, 1995.

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

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

[16] 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.

[17] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs using integer lattices. In *Languages and Compilers for Parallel Computing,* Lecture Notes in Computer Science, Vol. 1033, pp. 191–208, Springer-Verlag, 1996.

[18] 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., Chapter 29, Kluwer Academic Publishers, Norwell, MA, 1995.

[19] 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.

[20] A. Venkatachar. Code generation and optimization for block-cyclic distributions. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, October 1996.

[21] A. Venkatachar, J. Ramanujam, and A. Thirumalai. Generalized overlap regions for communication optimization in data parallel programs. In *Languages and Compilers for Parallel Computing,* Lecture Notes in Computer Science, Vol. 1239, pp. 404–419, Springer-Verlag, 1997.

[22] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, 1996.

Table 1. Communication table generation times (in microseconds).

k	p	s_1	s_2	LSU	OSU	RIACS
4	3	5	3	16	254	15
4	3	3	5	16	256	15
8	4	3	5	21	450	28
8	4	5	3	21	450	28
8	32	7	99	23	465	31
16	32	7	99	25	555	53
32	32	7	99	29	528	118
8	32	99	7	23	465	31
16	32	99	7	25	464	53
32	32	99	7	29	467	118

Table 2. Communication packing, and execution time results(in microseconds) of our approach compared with that of OSU on an IBM SP-2 using the cc compiler and -O3 optimization for the case $k = 8, p = 4, s_1 = 5, s_2 = 3$.

	Packing Time		Execution Time	
N	BUF	OSU	BUF	OSU
50	5	44	14	1239
100	10	87	33	2682
200	20	172	702	6490
500	656	430	1767	25226
1000	1312	1057	3526	82865
2000	2618	2988	7051	106505
3000	3932	4653	10632	110178

Table 3. Communication packing, and execution time results(in microseconds) of our approach compared with that of OSU on an IBM SP-2 using the cc compiler and -O3 optimization for the case $k = 4, p = 3, s_1 = 5, s_2 = 3$.

	Packing Time		Execution Time	
N	BUF	OSU	BUF	OSU
50	4	49	6	1375
100	8	97	11	2961
200	15	192	22	7345
500	216	480	851	29634
1000	742	1209	1704	100219
2000	1484	3292	3392	105521
3000	2223	4661	5099	108391