1

COMMUNICATION GENERATION AND OPTIMIZATION FOR HPF

A. Thirumalai, J. Ramanujam and A. Venkatachar

Department of Electrical and Computer Engineering,
Louisiana State University, Baton Rouge, LA 70803, USA.

1 INTRODUCTION

Programming massively parallel distributed memory machines involves partitioning data and computation across processors. In order to ease this difficult task, languages such as Fortran D [3] and High Performance Fortran [2] include directives that allow programmers to specify data mappings. In these languages, arrays are aligned to an abstract Cartesian grid called a *template* ; the template is then distributed across the various processors. 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. The distribution of computation in most compilers follows the *owner-computes* rule. That is, a 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 this paper, we present a novel approach to communication generation for general block-cyclic distributions based on the fact that the set of elements referenced form a lattice, i.e., they can be generated by integer linear combinations of basis vectors and that this set has a repeating pattern. Experiments on an IBM SP-2 show that the communication table generation times are small.

2 ADDRESS GENERATION

We consider an array A identically aligned to a template T, i.e., $A(i)$ is aligned with $T(ai+b)$, where the alignment stride $a = 1$ and the alignment offset $b = 0$. We assume that this template is distributed in a block-cyclic fashion onto p processors with a block size of k. This is also known as a $\texttt{CYCLIC}(k)$ distribution [2]. 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 code including the communication for each processor.

Under the mapping, we assign a set of k cells of the template to each of the p processors and then wrap around and assign the rest of the cells in a similar fashion. We refer to each set of pk elements as a course. The set of elements of A that are accessed form a *lattice* [10]; every lattice can be described by its *basis,* which is a collection of vectors, integer linear combinations of which can generate every point that belongs to the lattice. Since we want to enumerate the points of the lattice in lexicographic order, we use a set of *extremal basis* vectors non-negative integer linear combinations of which generate all the accessed points; this is useful in code generation, since nested loops can be used for lattice enumeration. The vectors $(1, -\ pk \bmod s)$ and $(0, s)$ form an extremal basis; details can be found in [9, 10].

Let $B = \begin{bmatrix} 1 & 0 \\ -(pk \bmod s) & s \end{bmatrix}$ be the basis matrix of the array section lattice; The determinant of B is the access stride s itself. Using a transformation $T = B^{-1}$ and Fourier-Motzkin elimination, we derive the code for processor $m (0 \leq m < p)$ as shown to the right. For details, the reader is referred to [8, 9, 10]. The access pattern for the LHS array A repeats after every $\frac{s}{\gcd(s,pk)}$ courses of elements. Hence, it is sufficient to store the access pattern of the elements in the first $\frac{s}{\gcd(s,pk)}$ courses and use this information to determine the remaining elements accessed. In the code shown to the right, the array U stores the access pattern. Details of code generation for an arbitrary array section $A(\ell : h : s)$ can be found in [8, 9, 10].

$\alpha = pk \bmod s$; $\beta = mk$; $w = 0$;
$s' = \frac{s}{\gcd(s,pk)}$; $N' = \lceil \frac{N}{s'} \rceil$;
DO $u = 0, s' - 1$
    $t = \alpha u + \beta$
    DO $v = \lceil \frac{t}{s} \rceil, \lfloor \frac{k'+t}{s} \rfloor$
        $U[w] = uk + vs - t$;
        Increment w;
    ENDDO
ENDDO
$length = w$; $offset = 0$;
DO $i = 0, N'$
    DO $j = 0, length - 1$
        $A[offset + U(j)] = \cdots$
    ENDDO
    $offset = offset + ks'$
ENDDO

3 COMMUNICATION GENERATION

Receive pattern generation

We consider a typical HPF-style assignment statement: $A(l_1 : h_1 : s_1) = B(l_2 : h_2 : s_2)$. Given a LHS element, we can easily determine the processor that owns the RHS element and the local address of the RHS element on that processor using the following expressions: $m = (i \text{ div } k)$ and $L = k * (i \text{ div } pk) + i \bmod k$, where i is a global address, m is the processor to which i belongs to and L is the local address of i on processor m. The access pattern in the global address space repeats itself after pk elements are accessed, i.e. after every s_1 courses, where s_1 is the access stride of the LHS array. To generate the receive pattern,

```
α = pk mod (s_1) ; k' = k − 1
β = m ∗ k − l_1 ; q = 0
DO  u = l_1/pk, s_1 − 1 + l_1/pk
  t = αu + β
  DO  v = ⌈t/s_1⌉, ⌊(k'+t)/s_1⌋
    i = (upk + mk + vs_1 − t − l_1)
    RProcpattern_m[q] = (i div k) mod p
    RPattern_m[q] = (k(i div pk) + i mod k)
    Increment q
  ENDDO
ENDDO
```

(a): Code for computing the receive pattern

it is sufficient to run the code generated in the previous sections 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 and local address of that element on it. This code is shown in Figure 1(a).

Send pattern generation

Our algorithm for send pattern generation is very similar to the algorithm for receive pattern generation discussed above. We exploit the fact 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 courses. The code that generates the send pattern is shown in Figure 1(b).

Communication Algorithm

The communication algorithm for an array assignment statement is shown in Figures 2 and 3. We assume that each processor maintains a separate send and receive buffer for all the m processors. As we have assumed a single template model, closed form expressions for the communication set generation exist for the case where $k = 1$ (template is cyclically distributed) and $k = \left\lceil \frac{N}{p} \right\rceil$ (template is block distributed) [6]. The algorithm shown in Figures 2 and 3 simplifies for these cases as well as the case where, s_{lhs} divides s_{rhs} or vice-versa.

Experimental Results Communication generation times for an IBM SP-2 are shown in table 1.

Acknowledgments We gratefully acknowledge the support from NSF through an Young Investigator Award CCR–9457768, an NSF grant CCR–9210422, and

$\alpha = pk \bmod (s_2)$; $k' = k - 1$
$\beta = m * k - l_2$; $q = 0$
DO $u = \frac{l_2}{pk}, s_2 - 1 + \frac{l_2}{pk}$
  $t = \alpha u + \beta$
  DO $v = \left\lceil \frac{t}{s_2} \right\rceil, \left\lfloor \frac{k'+t}{s_2} \right\rfloor$
    $i = \frac{(upk+mk+vs_2-t-l_2)s_1}{s_2} + l_1$
    $SProcpattern_m[q] = (i \text{ div } k) \bmod p$
    $\text{SPattern}_m[q] = (k(i \text{ div } pk) + i \bmod k)$
    Increment q
  ENDDO
ENDDO

(b): Code for computing the send pattern

Figure 1 Computing the receive and send patterns for $A(l_1 : h_1 : s_1) = B(l_2 : h_2 : s_2)$

- Generate the send pattern as shown in Section 3. 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: `/* Send pack */`

  $Scount_q = 0$, where $0 \le q \le p-1$ and $Scount_q$ is an offset into $SendBuffer_q$ corresponding to the q^{th} processor.
  DO $j = 0, \frac{max_2}{s_2 k}$
    DO $i = 0, sizeof(Spattern_m) - 1$
      $q = SProcpattern_m[i]$
      $SendBuffer_q[Scount_q] = B_{loc}[j * k + Spattern_m[i]]$
      Increment $Scount_q$
    ENDDO
  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.

Figure 2 Send phase generation algorithm

- Receive data from the p processors
- Run the receive pattern generation code shown in Section 3. 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 $0 \leq q \leq p-1$ and $Rcount_q$ is an offset into $ReceiveBuffer_q$ corresponding to the q^{th} processor.

DO $j = 0, \frac{max_1}{s_1 k}$
  DO $i = 0, sizeof(Rpattern_m) - 1$
    $q = RProcpattern_m[i]$
    if m = q
      $A_{loc}[j * k + Apattern_m[i]] = B_{loc}[Rpattern_m[i]]$
    else
      $A_{loc}[j * k + Apattern_m[i]] = ReceiveBuffer_q[Rcount_q]$
      Increment $Rcount_q$
    endif
  ENDDO
ENDDO

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

Figure 3 Receive phase generation algorithm

k	p	s_{lhs}	s_{rhs}	Comm. Gen. Time (microseconds)
4	3	5	3	41
4	3	3	5	40
8	4	3	5	49
8	4	5	3	49
8	32	7	99	84
16	32	7	99	128
32	32	7	99	240
8	32	99	7	84
16	32	99	7	127
32	32	99	7	228

Table 1 Communication generation times for an IBM SP-2 system.

from the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

REFERENCES

[1] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data parallel programs. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pp. 149–158, May 1993.

[2] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[3] G. Fox, S. Hiranandani, K. Kennedy, C Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D Language Specification. TR-91-170, Rice University, Dec. 1991.

[4] S. Gupta, S. Kaushik, C. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. OSU-CISRC-94-TR19, Apr. 1994.

[5] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Advanced compilation techniques for Fortran D. Report CRPC-TR-93-338, Rice University, Oct. 1993.

[6] C. Koelbel. Compile-time generation of communication for scientific programs. In *Supercomputing '91*, pp. 101–110, Nov. 1991.

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

[8] A. Thirumalai and J. Ramanujam. Code generation and optimization for array statements in High Performance Fortran. Technical Report TR-94-11-02, Dept. of Electrical and Computer Engineering, Louisiana State University, Nov. 1994.

[9] 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, June 1995.

[10] A. Thirumalai and J. Ramanujam. Fast Address Sequence Generation for Data-Parallel Programs Using Integer Lattices. In *Proc. 8th Annual Workshop on Languages and Compilers for Parallel Computing,* Columbus, Ohio, August 1995.