Compilation Techniques for Block-Cyclic Distributions

Seema Hiranandani
Ken Kennedy
John Mellor-Crummey
Ajay Sethi

CRPC-TR95521-S
March, 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

To Appear in Proceedings of Internationan Conference on Supercomputing '94, July 11-15, Manchester, England.

Compilation Techniques for Block-Cyclic Distributions [*]

Seema Hiranandani
Ken Kennedy
John Mellor-Crummey
Ajay Sethi
Department of Computer Science
Rice University
Houston, TX 77251-1892
[seema,ken,johnmc,sethi]@rice.edu

Abstract

Compilers for data-parallel languages such as Fortran D and High-Performance Fortran use data alignment and distribution specifications as the basis for translating programs for execution on MIMD distributed-memory machines. This paper describes techniques for generating efficient code for programs that use block-cyclic distributions. These techniques can be applied to programs with symbolic loop bounds, symbolic array dimensions, and loops with non-unit strides. We present algorithms for computing the data elements that need to be communicated among processors both for loops with unit and non-unit strides, a linear-time algorithm for computing the memory access sequence for loops with non-unit strides, and experimental results for a hand-compiled test case using block-cyclic distributions.

1 Introduction

Data parallel languages such as High-Performance Fortran (HPF) [11, 18] and Fortran D [15] have attracted considerable attention as promising languages for writing portable parallel programs. These languages support an abstract model of parallel programming in which users annotate a single-threaded program with data alignment and distribution directives. Compilers for MIMD distributed-memory machines use these directives to partition the program's computation as the basis for deriving a SPMD program to be executed on each node of the parallel machine.

HPF and Fortran D support three principal types of data distribution directives for partitioning arrays among the processors in a parallel machine: block, cyclic, and block-cyclic. A distribution directive is associated with a particular axis of an array, indicating how the array will be partitioned along that axis. A block distribution along an array axis indicates that the array will be partitioned along that axis into a set of equal length intervals, one for each processor assigned to the array axis. Block distributions are the distribution of choice for nearest neighbor stencil-based computations. A cyclic distribution along an array axis indicates that the data will be partitioned into unit intervals which are assigned to processors in a round-robin fashion. Block and cyclic distributions are just common cases of the more general block-cyclic distribution which specifies the length of the intervals into which an array axis will be partitioned. A more precise definition of these distributions is given in section 3. Cyclic and block-cyclic distributions are useful for writing efficient and load-balanced dense matrix algorithms on distributed-memory machines [5].

Previous research has focused on compilation strategies for handling block and cyclic distributions efficiently [19, 17]. Block-cyclic distributions have only been studied in detail recently. Chatterjee *et. al.* [4] present a general solution for generating local addresses and communication sets for data-parallel programs with block-cyclic distributions. Stichnoth *et. al.* [22] look at the problem of generating communication sets for block-cyclically distributed arrays. Gupta *et. al.* [7] compute the communication and local index sets using *virtual processor approach*. In this paper, we present faster and conceptually more intuitive algorithms for generating communication sets, a linear-time algorithm for computing the memory access sequence for loops with non-unit strides, and a list of formulae useful for compiling programs with block-cyclic distributions.

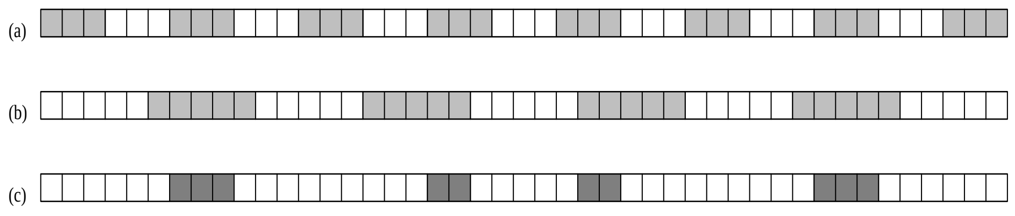
The structure of the paper is as follows. Section 2 briefly reviews the organization of the Rice Fortran 77D compiler which we use as a vehicle explaining our techniques. Section 3 introduces the terminology and notation used throughout the rest of the paper. Sections 4 and 5 present algorithms and analyses to compile Fortran D programs with block-cyclic distributions. Sections 6 presents a hand-compiled Gaussian elimination example and experimental results that show the impact of using block-cyclic distributions with different block sizes. The paper concludes with a brief summary of our contributions.

2 Fortran D Compiler

To compile Fortran D for MIMD distributed-memory machines, the Rice Fortran 77D compiler uses data alignment and distribution directives to partition the data and computation among the available processors, and then introduces communication operations to transfer values as necessary. By using aggressive compile-time analysis and optimization to statically partition the computation and schedule communication, the Fortran 77D compiler can generate programs that are far more efficient than other compilers that rely

[*] To appear in Proceedings of International Conference on Supercomputing '94, July 11-15, Manchester, England

```
REAL A(n), B(n)
DISTRIBUTE A(BLOCK_CYCLIC(3)), B(BLOCK_CYCLIC(5))
do i = 1, 45
    A(i) = 𝓕(B(i))
enddo
```



(a) Elements of A owned by processor 0 ≡ elements of B needed by processor 0
(b) Elements of B owned by processor 1
(c) Elements which processor 1 needs to send to processor 0 ≡ (a) ∩ (b)

Figure 1: Data ownership and communication for a block-cyclic data distribution.

on *run-time resolution* to explicitly calculate the ownership and communication for each reference at run time [3, 20, 23]. Below, we briefly review the sequence of steps performed by the Rice Fortran 77D compiler; details of the compilation process are described elsewhere [13, 14].

1) Analyze program The compiler performs scalar dataflow analysis, symbolic analysis, and dependence testing to determine the type and level of all data dependencies [16].

2) Partition data The compiler determines the decomposition of each array and uses alignment and distribution statements to calculate the array section owned by each processor.

3) Partition computation The compiler uses the "owner computes" rule to partition the computation among the processors. Each processor only computes values of data it owns [3, 20, 23]. The left-hand side (*lhs*) of each assignment statement in a loop nest is used to calculate the set of loop iterations that cause a processor to assign to local data. This iteration set represents the work that must be performed by the processor.

4) Analyze communication The compiler uses the computation partition to calculate the non-local data accessed by each processor for each right-hand side (*rhs*) reference to a distributed array. References that cause non-local accesses are marked since they require insertion of communication.

5) Optimize communication The compiler examines each marked non-local reference and uses results of data decomposition, symbolic and dependence analysis to determine the legality of optimizations to reduce communication costs. *Regular section descriptors* (RSDs) are built for the sections of data to be communicated. RSDs compactly represent rectangular array sections and their higher dimension analogs [10].

6) Manage storage The compiler identifies the extent and type of non-local data accesses represented by RSDs to calculate the storage required for non-local data. For RSDs representing array elements contiguous to the local array section, the compiler reserves storage using *overlaps* created by extending the local array bounds [6]. Otherwise, temporary buffers or hash tables are used for storing non-local data.

7) Generate code The compiler uses the results of previous stages to generate a SPMD message-passing program for nodes of a MIMD distributed-memory machine. To accomplish this, the compiler reduces array and loop bounds, introduces guards to instantiate the data and computation partitions, uses RSDs representing non-local data accesses to generate calls to data-buffering routines and to insert calls *send* and *recv* or collective communication routines as appropriate. The compiler inserts code to use run-time resolution to determine work and communication partitions when complex subscript expressions defy compile-time analysis.

3 Terminology

Here, we briefly review some terminology and notation that is used throughout the remainder of the paper. We use P to denote the number of processors, numbered 0 through $P-1$. For the following canonical loop nest,

```
do $\vec{i} = \vec{l}$ to $\vec{u}$ by $\vec{s}$
  $A(g(\vec{i})) = B(f(\vec{i}))$
enddo
```

we define the following sets; formal definitions of these sets are presented elsewhere [14].

- $image\_set_B(p)$ is the set of indices of array B that are owned by processor p.
- $iter\_set_A(p)$ is the set of loop iterations that cause reference A to access data owned by processor p.
- $index\_set_B(p)$ is the set of indices of array B referenced by processor p on loop iterations contained in $iter\_set_A(p)$.
- $send\_p\_set_B(p)$ is the set of processors to whom p must send local elements of array B.
- $recv\_p\_set_B(p)$ is the set of processors from whom p must receive values of non-local elements of array B.
- rsd_set_B is the set of indices of array B that are referenced in the loop nest.

```
    REAL A(n,n), B(n,n)
    DISTRIBUTE A(:, BLOCK_CYCLIC(8))
    DISTRIBUTE B(:, BLOCK_CYCLIC(8))
    do k = 1, n
        do i = k+1, n
$S_1$           A(i,k) = $\mathcal{F}$(B(i,k))
        enddo
        do j = k+1, n
            do i = k+1, n
$S_2$               A(i,j) = $\mathcal{G}$(A(i,j), A(i, k))
            enddo
        enddo
    enddo
```

Figure 2: Code fragment with block-cyclic distributions.

As a convenient notation for describing sequences of indices that arise with block-cyclic distributions, we use a quadruplet [l:u:b:c] inspired by the Fortran 90 triplet notation. The components are, respectively, the lower bound, upper bound, block width, and the cycle length which is equal to the product $b * P$.

The elements of a 1D distributed array owned by a particular processor can be represented using a quadruplet. Figure 1(a) shows a 1D array A partitioned among two processors. Processor 0 owns elements [1:3], [7:9], [13:15], and so on; this can be represented as [1:n:3:6] in quadruplet notation. The quadruplet notation also can represent block and cyclic distributions by setting $c = n$ and $b = \lceil \frac{n}{P} \rceil$ for a block distribution and $c = P$ and $b = 1$ for a cyclic distribution. For the sake of convenience we continue to represent contiguous ranges as [l:u]. Sections of multi-dimensional arrays can be represented using an instance of the [l:u:b:c] or [l:u] sequence notation for each array dimension.

Clearly, the quadruplet notation is insufficient to represent the sequence of data accessed on a processor for loops with non-unit stride since the memory access gap is non-constant (details of how this case is handled are presented in section 5). Despite the incomplete expressiveness of the quadruplet notation, we use it for the sake of convenience with the understanding that the compiler will handle cases outside the scope of this simple notation by using appropriate data-structures.

4 Program Analysis

In this section, we describe how to synthesize the analysis results needed to compile data parallel programs in presence of block-cyclic distributions. Before getting into the details of the analysis phases, we illustrate the compilation steps using a simple example.

The program fragment shown in Figure 2 is similar to Gaussian Elimination with pivoting but without a loop-carried dependence from S_2 to S_1. Statement S_1 corresponds to the computation of the pivot while statement S_2 corresponds to row elimination with column indexing. Performing partition and communication analysis (refer to [12] for details) yields the following sets for statement S_2:

$$\begin{aligned} image\_set_A(p) &= [1:n][p*8+1:n:8:P*8] \\ iter\_set_A(p) &= [1:n][lb:n:8:P*8][k+1:n] \\ index\_set_A(p) &= [k+1:n][1:n] \end{aligned}$$

The image set indicates that the columns of B described by the quadruplet $[p*8+1 : n : 8 : P*8]$ in rows 1 through n are owned by processor p. The iteration set follows from the loop nesting order, resulting in an index domain spanning [*range of k*][*range of j*][*range of i*]; the formula for computing *lb*, the lower bound, is described in Section 4.1. To simplify presentation, we have shown only the index set for the reference A(i,k) since all we are really interested in is $(index\_set_A \setminus image\_set_A)$, the set difference. This set difference indicates the columns not owned by the processor. In this case, the difference is non-empty and each processor needs to receive all the columns owned by other processors. In other words, each processor needs to *broadcast* its columns. Since there exists a dependence from the statement S_1 to S_2, the processor which computes the k*th* column needs to perform a *broadcast* after executing the loop containing statement S_1. Figure 3 shows a hand-compiled version of the source fragment shown in Figure 2 that was translated using this strategy.

```
do kk = 1, n, 8
  do k = kk, min(kk+7, n)
    k$ = ...
    if (my$p owns the kth column) then
      do i = k+1, n
        A(i,k$) = $\mathcal{F}$(B(i,k$))
      enddo
      broadcast A(k+1:n, k$)
    else
      receive A(k+1:n, k$)
    endif
    lb$1 = ...
    ub$1 = ...
    do j = lb$1, ub$1
      do i = k+1, n
        A(i,j) = $\mathcal{G}$(A(i,j), A(i,k$))
      enddo
    enddo
  enddo
enddo
```

Figure 3: Hand-compiled block-cyclic example.

The block-cyclic distribution enables an optimization. Since each processor owns a block of columns, it can compute the pivot for one whole (or part) block of columns it owns. Instead of sending a separate message for each column, a processor can send one message for each sub-block of a block of columns. Since current distributed memory machines have high communication costs, reducing the number of messages with this optimization has the potential to improve performance over that attainable using a cyclic distribution. Figure 4 shows hand-compiled version of the source fragment shown in Figure 2 that was translated using this blocking strategy to vectorize of broadcasts.

4.1 Loop Indices and Bounds Generation

Consider the program fragment shown in the top half of Figure 5. The bottom half of the figure shows its corresponding SPMD node program using the functions *LowerLoopBound*, *UpperLoopBound*, *LocalLoopIndex*, *GlobalLoopIndex* and *Owner*. These formulae to compute these quantities are given below. The functions are specific to the example in the figure since they assume unit loop strides and the free variables L, b, and P in the functions refer to the lower bound of the X array, the block size for the block-cyclic distribution, and the number of processors, respectively. Note that the lower bound of the array in the SPMD code is 1.

```
do kk = 1, n, 8
if (my$p owns columns kk to kk+7) then
  do k = kk, min(kk+7, n)
    k$ = ...
    do i = k+1, n
      A(i,k$) = $\mathcal{F}$(B(i,k$))
    enddo
    *buffer* A(k+1:n, k$)
  enddo
  *broadcast buffer*
else
  *receive buffer*
endif
do k = kk, min(kk+7, n)
  k$ = ...
  lb$1 = ...
  ub$1 = ...
  do j = lb$1, ub$1
    do i = k+1, n
      A(i,j) = $\mathcal{G}$(A(i,j), *buffer*(...))
    enddo
  enddo
enddo
enddo
```

Figure 4: Hand-compiled code with vectorized broadcasts.

$Owner(X(i)) \equiv \lfloor \frac{i-L}{b} \rfloor \bmod P$

$LocalLoopIndex(i) \equiv \lfloor \frac{i-L}{b*P} \rfloor b + ((i-L) \bmod b) + 1$

$GlobalLoopIndex(i\$,p) \equiv$

$\lfloor \frac{i\$-1}{b} \rfloor (P*b) + p*b + ((i\$-1) \bmod b) + L$

```
$LowerLoopBound(L_j,p) \equiv$

  $l_1 = L_j$ - L
  if ($p < \lfloor \frac{l_1}{b} \rfloor \bmod P$) then
      return $\lceil \frac{l_1}{b*P} \rceil * b + 1$
  else if ($p > \lfloor \frac{l_1}{b} \rfloor \bmod P$) then
      return $\lfloor \frac{l_1}{b*P} \rfloor * b + 1$
  else
      return $\lfloor \frac{l_1}{b*P} \rfloor * b + (l_1 \bmod b) + 1$
  endif

$UpperLoopBound(U_j,p) \equiv$

  $u_1 = U_j$ - L
  if ($p < \lfloor \frac{u_1}{b} \rfloor \bmod P$) then
      return $\lceil \frac{u_1}{b*P} \rceil * b$
  else if ($p > \lfloor \frac{u_1}{b} \rfloor \bmod P$) then
      return $\lfloor \frac{u_1}{b*P} \rfloor * b$
  else
      return $\lfloor \frac{u_1}{b*P} \rfloor * b + (u_1 \bmod b) + 1$
  endif
```

If the number of processors and the loop bounds are known at compile time, then the loop bounds formulae can be evaluated at compile time. If these quantities are symbolic variables, then the formulae would require run-time evaluation. In experiments that measured the overhead of performing similar calculations at run time in shared virtual memory systems, the overhead of these calculations was found to be insignificant [2].

We present the techniques to handle loops with non-unit strides and more complex array subscripts in Section 5.

```
       {* Original Program *}
       REAL X(L:U)
       DISTRIBUTE X(BLOCK_CYCLIC(b))
       do i = $L_i$,$U_i$
$S_1$    X(i) = $\mathcal{F}_1$(i)
         do j = $L_j$,$U_j$
$S_2$      X(j) = $\mathcal{F}_2$(j)
         enddo
       enddo


       {* SPMD Node Program *}
       REAL X(1:$\lceil (U-L+1)/P \rceil$)
       lb$ = *LowerLoopBound*($L_j$)
       ub$ = *UpperLoopBound*($U_j$)
       do i = $L_i$,$U_i$
         i$ = *LocalLoopIndex*(i)
         if ($p$ .eq.  Owner(X(i))) X(i$) = $\mathcal{F}_1$(i)
         do j = lb$,ub$
           j$ = *GlobalLoopIndex*(j, $p$)
           X(j) = $\mathcal{F}_2$(j$)
         enddo
       enddo
```

Figure 5: Loop indices and bounds generation.

4.2 Partitioning Analysis

We now describe the analysis required to compute iteration sets in the presence of symbolic loop bounds, array dimensions and number of processors. These iteration sets are used in two ways by the code generation phase: first, to reduce loop bounds so that each processor iterates only over the portion of the iteration space that causes it to reference data elements that it owns, and second to introduce guards to handle cases in which iteration sets are not identical for all processors.

4.2.1 Symbolic Iteration Sets

For each assignment statement in a loop nest, the compiler must compute an iteration set, parameterized by processor number, that represents the set of loop iterations that cause the processor to access the data it owns. Our discussion of iteration set construction will be based on the canonical loop nest shown below, which is the same as the loop given in Section 3.

```
Given:
  REAL $A(l_1 : u_1, \ldots, l_n : u_n), B(l_1 : u_1, \ldots, l_n : u_n)$
  DECOMPOSITION $D(l_1 : u_1, \ldots, l_n : u_n)$
  ALIGN $A(\ldots, m, \ldots)$, $B(\ldots, m, \ldots)$ WITH $D(\ldots, m, \ldots)$
  DISTRIBUTE $D(d_1, \ldots, BLOCK\_CYCLIC(b), \ldots, d_n)$
Loop nest:
  DO $i_1 = lb_1, ub_1$
    ...
    DO $i_n = lb_n, ub_n$
      ...
$S_1$     $A(\ldots, g(i_k), \ldots) = B(\ldots, f(i_l), \ldots)$
    ENDDO
    ...
  ENDDO
```

Note that in the loop given above the k*th* dimension of A, the l*th* dimension of B are aligned with the m*th* dimension of D (which is distributed block-cyclically). To determine a processor's iteration set for an assignment statement in a loop nest, the compiler examines the subscripted array reference on the left-hand side of the assignment, the array's alignment with its associated decomposition, and the distribution specification for the decomposition that reaches the assignment statement. Computing the iteration set involves reducing the index variable bounds for each distributed dimension of the decomposition corresponding to the lhs term. This problem of reducing the bounds is independent for each distributed dimension. For presentation purposes, we assume that only one dimension, d_m, is (block-cyclically) distributed. For arrays with more than one distributed dimension, the iteration set can be computed by intersecting the solutions for each singly-distributed dimension sub-problem.

In the above loop nest, it is assumed that the subscript function $g(i_k)$ has been simplified. In the cases where $g(i_k)$ is a constant, an induction variable, or a linear function of a single index variable, the iteration set can be computed at compile-time. For more complex subscript expressions, the compiler defers the computation of the iteration set to run time. In the following formulae, the functions *GlobalLowerLoopBound* and *GlobalUpperLoopBound* return the reduced lower and upper bounds for the loop in global indices. These functions are similar to the *LowerLoopBound* and *UpperLoopBound* functions given in Section 4.1.

- **Constant:** $g(i_k) \equiv c_k$

```
if $(Owner(A(\ldots, c_k, \ldots)) = p)$ then
    $iter\_set(p) = (lb_1 : ub_1, \ldots, lb_k : ub_k, \ldots, lb_n : ub_n)$
else
    $iter\_set(p) = \emptyset$
endif
```

- **Induction Variable Only:** $g(i_k) \equiv i_k$

$$lb_{k1} = GlobalLowerLoopBound(lb_k)$$
$$ub_{k1} = GlobalUpperLoopBound(ub_k)$$
$$iter\_set(p) = (lb_1 : ub_1, \ldots, lb_{k1} : ub_{k1} : b : b * P, \ldots, lb_n : ub_n)$$

- **Simple Linear Expression:** $g(i_k) \equiv i_k + c$

$$lb_{k1} = GlobalLowerLoopBound(lb_k - c)$$
$$ub_{k1} = GlobalUpperLoopBound(ub_k - c)$$
$$iter\_set(p) = (lb_1 : ub_1, \ldots, lb_{k1} : ub_{k1} : b : b * P, \ldots, lb_n : ub_n)$$

- **Linear Expression:** $g(i_k) \equiv c_1 * i_k + c_0$
To compute the iteration set, the subscript with linear expression is converted into an *induction variable only* subscript by changing the loop parameters. For example, if the original upper and lower loop bounds and the step are lb_k, ub_k and 1 respectively, then the transformed upper and lower loop bounds and step are $lb_k * c_1 + c_0$, $ub_k * c_1 + c_0$ and c_1 respectively.

  To compute the iteration set, the first iteration assigned to the processor needs to be determined. This is equivalent to finding the smallest non-negative integer j such that

$$Owner(A(lb_k * c_1 + c_0 + c_1 * j)) = p$$

  This equation gives a set of linear Diophantine equations. The solution to a similar set of equations is given in Section 5.

4.3 Communication Analysis

4.3.1 Index Sets

Index sets are built for each distributed right-hand side array reference and contain the section of data accessed by a processor. They are used to determine the resulting communication.

From the previous section, observe that the iter_set(p) could either be $\emptyset$, $(lb_1 : ub_1, \ldots, lb_k : ub_k, \ldots, lb_n : ub_n)$ or $(lb_1 : ub_1, \ldots, lb_k : ub_k : b : b * P, \ldots, lb_n : ub_n)$. The index sets can be computed by substituting the value of iter_set(p) in the *rhs* subscript function f(i_l). Note that the index set is independent of the distribution of the referenced *rhs* array. If the iter_set(p) cannot be determined at the compile time, the index set computation needs to be done at the run time.

4.3.2 Communication Classification

Communication classification is a crucial step in the compilation process since it allows the compiler to insert calls to fast collective communication primitives in the output program and optimize the communication. Each non-local reference is classified as resulting in Single Send/Receive, Shift, Broadcast, Gather, All-to-All, Inspector/Executor or Run-time Resolution type of communication. The details of the algorithm can be found in [12].

4.4 Send and Receive Sets

In order to compute the send and receive sets (send_p_set and recv_p_set, respectively), the compiler needs to find out, in the most general case, the intersection of two block-cyclically distributed arrays. However, as the example in Figure 1 demonstrates, block-cyclic sets are not closed under intersection.

Stichnoth *et. al.* [22] treat the block-cyclic sets as a union of disjoint cyclic sets. Since the cyclic sets are closed under intersection, the intersection of the two block-cyclic sets can be determined by intersecting all possible pairs of the cyclic sets. An advantage of using this technique is that the cost of translation from global index space to local index needs to be incurred only once for the lower and upper bound and stride of each section. A more general approach, *virtual processor approach*, is taken by Gupta *et. al.*[8, 7]. Using this approach, a block-cyclic distribution can be viewed as a cyclic (or block) distribution on a set of virtual processors, which are block-wise (or cyclically) mapped to physical processors. With the virtual processor approach, explicit local-to-global and global-to-local translations are not needed for every index element communicated among the processors.

Mostly, the compiler needs to intersect two block-cyclic sets only once: to compute send_p_set and recv_p_set. An efficient approach which avoids the overheads of computing *mods, ceilings, etc.* while computing the intersection, is to treat each block-cyclic set as an array sorted in ascending order. The intersection can then be computed by a simple linear-time algorithm similar to the merge sort algorithm. For example, processor 1 needs to send [1:45:3:6] $\cap$ [6:45:5:10] elements to processor 0. In other words, the compiler needs to find the intersection

$$\{1, 2, 3, 7, 8, 9, 13, 14, 15, 19, 20, 21, \ldots, 43, 44, 45\} \cap$$
$$\{6, 7, 8, 9, 10, 16, 17, 18, 19, 20, \ldots, 36, 37, 38, 39, 40\}$$

Of course, the intersection can be computed without looking at all the array elements because the pattern repeats after LCM(Block-size(A),Block-size(B))*P (= LCM(3,5)*2

```
Input : Arrays A and B as the lhs and the rhs array
      respectively. The sender processor s, the destination
      processor d and the block sizes $bsize_1$ and $bsize_2$.
      We assume the presence of a function "next_elem"
      that uses the current location and the block-size
      to increment the index pointer to the next array
      element owned by the processor.
Output : The set of elements of array B which need
      to be sent from s to d.
Method :
s_index = first_elem($image\_set_B(s)$);
d_index = first_elem($index\_set_A(d)$);
while (s_index $\leq$ LCM($bsize_1$, $bsize_2$) * P and
      d_index $\leq$ LCM($bsize_1$, $bsize_2$) * P) do
    if (s_index < d_index) then
        next_elem(s_index, $bsize_1$);
    else if (s_index > d_index) then
        next_elem(d_index, $bsize_2$);
    else
        buffer(s_index);
        next_elem(s_index, $bsize_1$);
        next_elem(d_index, $bsize_2$);
    endif
endwhile
Using the elements belonging to the intersection (as
computed above) and the periodicity, buffer the rest
of the elements which need to be sent.
```

Figure 6: Algorithm for computing the intersection.

= 30, in this case) elements. Hence the time required to compute the intersection is $\mathcal{O}$(LCM(Block-size(A), Block-size(B))*P) which is better than the time complexity of the method suggested in [22]. Note that both the sender and receiver compute the intersection using the global indices. To pack and unpack the message, both the processors need to translate the global indices to local indices. However, by substituting in the value in the formula for *LocalLoopIndex* given in Section 4.1, we get
*LocalLoopIndex(i+LCM(Block-size(A),Block-size(B))*P) = LocalLoopIndex(i) + LCM(Block-size(A),Block-size(B)).*
Therefore, we can use the repetitive pattern to perform address translation for LCM(Block-size(A),Block-size(B))*P index elements only.

In case the number of processors or the loop bounds are unknown at compile time, the compiler needs to perform the intersection at run time. Figure 6 gives the algorithm for computing the array elements which need to be sent from one processor to another. The algorithm for computing the receive set is similar to that for the send set.

In practice, though, we do not expect to find arbitrary block sizes (like 3 and 5 in the Figure 1). Since the block sizes also affect the locality of the array accesses, and hence the memory hierarchy optimizations, we expect the arrays to have the same block sizes or block sizes which are powers of 2. In these cases our algorithm would provide the most benefits. In the case of perfectly aligned arrays, the intersection would still be a block-cyclic set with block size equal to the smaller of the two original block sizes. The send_p_set and recv_p_set sets can be computed trivially in this case.

5 Loops with Non-unit Stride

As mentioned earlier, in the presence of block-cyclic distributions, closed-form expressions for iter_set, send_p_set and recv_p_set cannot be written. In this section we will describe algorithms to construct these sets in the presence of loops with non-unit strides and/or non-unit array subscripts. We have identified two cases of data access patterns for which we provide fast solutions for computing the sets. If the data access pattern does not fit either form then we resort to the algorithms proposed by Chatterjee *et. al.*[4]. In the case of non-unit stride and/or non-unit array subscript, the data accessed in a processor's local memory results in a non-constant stride pattern. In the example below,

Processor 0				Processor 1				Processor 2				Processor 3			
$\boxed{1}$	2	3	4	5	$\boxed{6}$	7	8	9	10	$\boxed{11}$	12	13	14	15	$\boxed{16}$
17	18	19	20	$\boxed{21}$	22	23	24	25	$\boxed{26}$	27	28	29	30	$\boxed{31}$	32
33	34	35	$\boxed{36}$	37	38	39	40	$\boxed{41}$	42	43	44	45	$\boxed{46}$	47	48
49	50	$\boxed{51}$	52	53	54	55	$\boxed{56}$	57	58	59	60	$\boxed{61}$	62	63	64
65	$\boxed{66}$	67	68	69	70	$\boxed{71}$	72	73	74	75	$\boxed{76}$	77	78	79	80
$\boxed{81}$	82	83	84	85	$\boxed{86}$	87	88	89	90	$\boxed{91}$	92	93	94	95	$\boxed{96}$

Figure 7: Block-cyclic distribution with non-unit stride.

```
REAL A(5,96), B(5, 96)
PARAMETER(N$PROC = 4)
DECOMPOSITION D(5, 96)
ALIGN A with D
DISTRIBUTE D(:, BLOCK_CYCLIC(4))
DO 10 i = 1, 5
  DO 10 j = 1, 96, 5
    A(i,j) = ...
10 continue
```

the second dimension of array A is distributed block-cyclically among 4 processors, with a block size of 4. The layout of array A in processor memories is depicted in Figure 7. As illustrated by Chatterjee *et. al.*, the layout of the array in memory can be visualized as an array of courses and offsets. The offset of an array element is its offset within the course. As an example, $A(3)$ resides in course 1, offset 2 in Processor 0's memory. Figure 7 also illustrates the elements of array a that are referenced in the loop nest. For instance, Processor 0 accesses elements A(1), A(36), A(51), ... and so on. The access *stride* for a given array reference on a processor is distance in local memory between each access. For example, the *stride* for array reference $A(i,j)$ on processor 0 is (11, 3, 3, 3).

The key insight, as noted in [4], is that the offset of an element determines the offset of the next element on the same processor. Since the offsets range between 0 and (block-size-1), by pigeon hole principle, at least two of the first (block-size+1) local memory locations on any particular processor must have the same offset. Moreover, since the offset of the next element depends only on the offset of the current array element, we conclude that there exists a cycle of memory access gaps.

Suppose we wish to find the first element (if any) of the array section that resides on a processor. This is equivalent to finding the smallest non-negative integer j such that

$$\left\lfloor \frac{(L_i + s * j - L) \, mod \, (P * b)}{b} \right\rfloor = p$$

where L_i = lower loop bound, L = array lower bound, s = step size and b is the block size.
The above equation is equivalent to

$$b * p \leq (L_i + s * j - L) \, mod \, (P * b) \leq b * (p + 1) - 1$$

which is equivalent to finding an integer q such that

$$b * p - L_i + L \leq s * j - q * P * b \leq b * (p + 1) - L_i + L - 1$$

```
do l = 1, n
  ...
  ipnt = ipntp
  ipntp = ipntp+il
  ...
  do k = ipnt+2, ipntp, 2
    i = i + 1
    X(i) = X(k) - V(k)*X(k-1) - V(k+1)*X(k+1)
  enddo
  ...
enddo
```

Figure 8: Livermore kernel 2 (ICCG excerpt).

The above inequality can be written as a set of b linear Diophantine equations in the variables j and q,

$$\{s*j-q*P*b=\lambda | b*p-L_i+L \leq \lambda \leq b*(p+1)-L_i+L-1\}$$

The equations can be solved independently (solutions exist for an individual equation if and only if λ is divisible by GCD(s, b*P)). The general solution of a linear Diophantine equation can be found using the extended Euclid algorithm.

The extended Euclid algorithm gives not only the first such memory location, but a list of all the locations (array elements). We could then sort this list based on the array elements accessed by a processor and thereby compute the memory access gap sequence. Using this idea, Chatterjee *et. al.*give algorithms for computing the memory access gap sequence for loops with arbitrary array alignments and step size. The running time of the algorithm using this approach is $\mathcal{O}(\log min(s, P*b) + b\log b)$ which reduces to $\mathcal{O}(min(b\log b + \log s, b\log b + \log P))$.

In the following section, we present linear time algorithms for two cases. In the first case, the access stride, s, is less than the block size. In practice, this is the most commonly occurring case. As an example, consider the stripped down version of the loop (Figure 8) which appears in the Kernel 2 (Incomplete Cholesky-Conjugate Gradient) of the Livermore benchmark suite.

For the purpose of illustration, the access pattern corresponding to this case is depicted in Figure 9, where the access stride is 3. The Fortran D loop corresponding to such an access pattern is shown below :

```
REAL A(80)
PARAMETER(N$PROC = 5)
DECOMPOSITION D(80)
ALIGN A with D
DISTRIBUTE D(BLOCK_CYCLIC(4))
DO 10 i = 1, 80, 3
   A(i) = ...
10 continue
```

The second case, though not as common as the first one, has the access stride s with the constraint : $(s \mod (b*P)) < b$. This would occur, for example, when s is equal to the array column size and $b*P$ divides the column size. This occurs in linear algebra codes when one wants to access the consecutive row elements (instead of the column elements) of a linearized array. The access pattern, for case II, is depicted in Figure 10. The access stride is 23 and the total number of memory locations is 500.

Note that the second case subsumes the first one (because $(s \mod (b*P)) < b$ implies $s < b$). However, we are able to exploit the constraint that $s < b$ to achieve a simpler

Processor 0				Processor 1				Processor 2				Processor 3				Processor 4			
$\boxed{1}$	2	3	$\boxed{4}$	5	6	$\boxed{7}$	8	9	$\boxed{10}$	11	12	$\boxed{13}$	14	15	$\boxed{16}$	17	18	$\boxed{19}$	20
21	$\boxed{22}$	23	24	$\boxed{25}$	26	27	$\boxed{28}$	29	30	$\boxed{31}$	32	33	$\boxed{34}$	35	36	$\boxed{37}$	38	39	$\boxed{40}$
41	42	$\boxed{43}$	44	45	$\boxed{46}$	47	48	$\boxed{49}$	50	51	$\boxed{52}$	53	54	$\boxed{55}$	56	57	$\boxed{58}$	59	60
$\boxed{61}$	62	63	$\boxed{64}$	65	66	$\boxed{67}$	68	69	$\boxed{70}$	71	72	$\boxed{73}$	74	75	$\boxed{76}$	77	78	$\boxed{79}$	80

Figure 9: Example for case I (step = 3).

algorithm and we present it first. The algorithm for case II works almost identically to case I by treating $s \mod (b*P)$ as the step size (which is less than b). However, in this case, the algorithm also keeps track of the number of skipped rows to compute the memory gaps correctly.

The important property satisfied by both $s < b$ (Case I) and $s \mod b*P < b$ (Case II) is that if processor p executes the i*th* iteration then the (i+1)*th* iteration would be executed either by processor p itself or by processor $(p+1) \mod P$. This fact is used to achieve the linear-time algorithm by computing the offsets (and the memory access gaps) without actually solving the Diophantine equations.

5.1 Algorithms to Calculate the Memory Access Sequence

We now present algorithms to compute the local-memory access sequence for loops with non-unit stride. It is assumed that the data distribution is aligned perfectly with the decomposition. Otherwise, if the data distribution is aligned to the decomposition using an affine alignment, then we would need two applications of the following algorithms to get the memory access sequence.

Figure 11 gives the algorithm for Case I. Given an offset for an index element on processor p and a step size s, first, the number of iterations executed within the same course (= numLocalHops) is computed. Next, we step through the course using the stride s and store the memory access gap (which is simply s for these index elements) in $\Delta\mathcal{M}$ table. Using the location of the last index element accessed in the course (= lastLoc), we can compute the number of elements needed to be stepped through before reaching processor p again (which is e = (P-1)*b + (b-lastLoc)). e mod s then gives the number of array elements left after the last iteration on the $(p-1)$ mod P processor. Therefore, nextOffset = $s - e$ mod $s-1$ is the offset (within the next course) of the next index element accessed by p. Note that since $s < b$, the next element would lie in the next course and, therefore, local-memory access gap = b - lastLoc + nextOffset + 1. The algorithm iterates till it finds a cycle of memory access gaps.

As mentioned before, for case II, the algorithm works almost identically to that of case I by treating $s \mod (b*P)$ as the step size and keeping track of the number of skipped rows to compute the memory gaps correctly. We illustrate the algorithm for case II (given in Figure 12) using our example (Figure 10). In Figure 10, $s \mod (b*P)$, the horizontal displacement of the current index element from the index element accessed in the previous iteration, is 3 (for example, on Processor 0, element 24 has offset 3 w.r.t. the beginning of the course, while element 1 has offset 0. Therefore, the horizontal displacement from 1 to 24, G = 3). numLocalHops gives the number of consecutive iterations which access data on the same processor. In case consecutive iterations

Processor 0				Processor 1				Processor 2				Processor 3				Processor 4			
$\boxed{1}$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	$\boxed{24}$	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	$\boxed{47}$	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	$\boxed{70}$	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	$\boxed{93}$	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	$\boxed{116}$	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	$\boxed{139}$	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	$\boxed{162}$	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	$\boxed{185}$	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	$\boxed{208}$	209	210	211	212	213	214	215	216	217	218	219	220
221	222	223	224	225	226	227	228	229	230	$\boxed{231}$	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250	251	252	253	$\boxed{254}$	255	256	257	258	259	260
261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	$\boxed{277}$	278	279	280
281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	$\boxed{300}$
301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320
321	322	$\boxed{323}$	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340
341	342	343	344	345	$\boxed{346}$	347	348	349	350	351	352	353	354	355	356	357	358	359	360
361	362	363	364	365	366	367	368	$\boxed{369}$	370	371	372	373	374	375	376	377	378	379	380
381	382	383	384	385	386	387	388	389	390	391	$\boxed{392}$	393	394	395	396	397	398	399	400
401	402	403	404	405	406	407	408	409	410	411	412	413	414	$\boxed{415}$	416	417	418	419	420
421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	$\boxed{438}$	439	440
441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460
$\boxed{461}$	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480
481	482	483	$\boxed{484}$	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500

Figure 10: Example for case II (step = 23).

Input: Offset of a valid iteration for processor 0 (offset$_0$), Block size (b), step (s), processor number (p), number of processors (P).

Output: The $\Delta\mathcal{M}$ table. The algorithm can also be used to record the starting memory location and the length of the table.

Method:

```
$\Delta\mathcal{M}$[i] = NOT_DEFINED, i=0,...,b-1;
offset = s - ($p$*b - (offset$_0$ + $\lfloor \frac{p*b-(\text{offset}_0+1)}{s} \rfloor$*s));
while (true) do
    if ($\Delta\mathcal{M}$[offset] $\neq$ NOT_DEFINED) break;
    numLocalHops = $\lfloor \frac{b-(\text{offset}+1)}{s} \rfloor$;
    for (i=1; i $\leq$ numLocalHops; i++) do
        $\Delta\mathcal{M}$[offset] = s;
        offset = offset + s;
    endfor
    lastLoc = offset + 1;
    nextOffset = s - [(P-1)b + (b-lastLoc)] mod s - 1;
    $\Delta\mathcal{M}$[offset] = b - offset + nextOffset;
    offset = nextOffset;
endwhile
```

Figure 11: Algorithm for case I (s < b).

access data on the same processor (like the first and the second iterations which access elements 1 and 24, respectively), R gives the number of rows which are skipped between the two index elements. In our example, R = $\lfloor \frac{23}{4*5} \rfloor = 1$ and since block size, b, = 4 and G = 3, depending on the offset of the current index element, numLocalHops would be 0 or 1. Lastly, note that the index element accessed on Processor 0 after 24 is 162. T, the total number of rows skipped between any two such iterations, is computed using G, R and the number of elements which need to be stepped through (in other words, the total horizontal displacement required) after the last index element on the current processor. In Figure 10, for index element 24, T = $(\lfloor \frac{16}{3} \rfloor + 1) * 1 = 6$, which is the number of courses skipped on Processor 0 between index elements 24 and 162. Using these values, we can compute the local-memory access gaps.

Given the loop lower bound L$_i$ and the array lower bound L, for case I :

$$\text{offset}_0 = \begin{cases} (L_i - L) \bmod s, & if\ (L_i - L) \bmod s \neq 0 \\ s, & if\ (L_i - L) \bmod s = 0 \end{cases}$$

For case II, we need slightly more work :

Let $G = s \bmod (b * P)$, the processor which owns L_i be $p = \lfloor \frac{(L_i - L) \bmod (b*P)}{b} \rfloor$, the offset of L_i within its course be $o = (L_i - L) \bmod b$. Therefore, the number of elements left in the row, $e = b - (o+1) + (P - p - 1) * b$ and the number of elements left after the last iteration (in the sense that the next iteration would be executed by Processor 0) in the sequence (refer to Figure 10), $l = e \bmod G$. Now, offset$_0 = G - l - 1$.

An interesting fact which could be used to further speed up the algorithms is that the length of the memory access gap sequence cycle divides the block size. Others tricks like treating multiplications and divisions as shifts in case of step size or block size being powers of 2 can also be used to improve the running time of the algorithms.

Input: Offset of a valid iteration for processor 0 (offset_0), Block size (b), step (s), processor number (p), number of processors (P).

Output: The $\Delta\mathcal{M}$ table. The algorithm can also be used to record the starting memory location and the length of the table.

Method:

```
$\Delta\mathcal{M}$[i] = NOT_DEFINED, i=0,...,b-1;
G = Horizontal Displacement = s $mod$ (b * P);
R = Rows Skipped = $\lfloor \frac{s}{b*P} \rfloor$;
T = Total Rows Skipped
offset = G - ($p$*b - ($\text{offset}_0$ + $\lfloor \frac{p*b-(\text{offset}_0+1)}{G} \rfloor$*G));
while ($true$) do
    if ($\Delta\mathcal{M}$[offset] $\neq$ NOT_DEFINED) break;
    numLocalHops = $\lfloor \frac{b-(\text{offset}+1)}{G} \rfloor$;
    for (i=1; i $\leq$ numLocalHops; i++) do
        $\Delta\mathcal{M}$[offset] = b*R + G;
        offset = offset + G;
    endfor
    lastLoc = offset + 1;
    ElementsLeft = b * P - lastLoc;
    T = $\left(\lfloor \frac{\text{ElementsLeft}}{G} \rfloor + 1\right)$ * R;
    GlobalElemsLeft = ElementsLeft $mod$ G;
    nextOffset = G - GlobalElemsLeft - 1;
    $\Delta\mathcal{M}$[offset] = b * T + (b-(offset+1)) + (nextOffset+1);
    offset = nextOffset;
endwhile
```

Figure 12: Algorithm for case II (s *mod* b*P < b).

Complexity: Each element of the array $\Delta\mathcal{M}$ is filled at most once by the algorithm. As soon as an already filled array element is encountered, the algorithm stops. Therefore, the while loop (together with the inner for loop) iterates at most b times and hence it is an $\mathcal{O}(b)$ algorithm. Therefore, as compared to the method suggested in [4], not only is our approach conceptually more intuitive, the algorithms given above are an $\mathcal{O}(b)$.

5.2 Send and Recv Sets

Once we have computed the memory access sequence for an array access, the computation of the send and receive sets is comparatively easier. Consider the following example :

```
REAL A(n)
DISTRIBUTE A(BLOCK_CYCLIC(4))
do i = 1, N, 5
    A(i) = $\mathcal{F}$(A(i-1), A(i), A(i+1))
enddo
```

If P=4, then we would get the same memory access sequence as shown in Figure 7. Only those iterations which assign to the array elements A(i) s.t. its offset is 0 or 3 (= the block-size - 1) need to receive some data (corresponding to A(i-1) and A(i+1), respectively).

In general, in case of communication required because of *shifts*, we can find both the processors that need to send the data and the location of the array element within the owner processor [4] . Note that, in case of shifts, a processor communicates with at most two processors.

Suppose that element A(i) is located on processor p with offset o. We want to find the processor and local memory location of A(i-d). Let d = q(P*b)+r and $\Delta\mathcal{P} = \lceil (\text{r}-o)/\text{b} \rceil$, where b is the block size. Then the owner processor of A(i-d) is $(p - \Delta\mathcal{P}+\text{P})$ mod P. Since $0 \leq o \leq \text{b}$, ΔP can assume only two values.

The location of A(i-d) (say $\mathcal{M}'$) can be computed from the location of A(i) (say $\mathcal{M}$) as follows. We define $\Delta\mathcal{L}$ such that $\mathcal{M}' = \mathcal{M} + \Delta\mathcal{L}(o)$.

$$\begin{aligned} \Delta\mathcal{L}(o) &= ((o - r) \; mod \; b) - o - bq - \eta, \\ \eta &= \begin{cases} b & if \; (p * b - r + o) < 0, \\ 0 & otherwise. \end{cases} \end{aligned}$$

Since the memory access sequence algorithm can compute offsets also, we can determine the iterations that need communication as well as the elements that need to be sent without any extra work.

In the case of block-cyclic distributions with different block sizes, the send and receive sets can be computed by computing the local index sets, local iteration sets, etc. For more complicated patterns, for example in case of stride changes, there does not exist any simple lookup technique for generating the communication sets because the pattern of destination processors can have period longer than the block size b. In such cases, we resort to the inspector-executor model [21, 9] for irregular loops.

6 Example and Experimental Results

To explore the effects of block-cyclic data distributions, we experimented with the DGEFA subroutine from Linpack. DGEFA is a key subroutine which performs Gaussian elimination with partial pivoting. Since the subroutine contains a triangular loop, a cyclic or block-cyclic distribution is desirable for maintaining good load balance.

Figure 13 shows the original program as well as the hand-compiled Fortran D program that uses a column block-cyclic distribution of width b which distributes blocks of columns in a round-robin fashion across the processors.

Table 1 shows timings for DGEFA benchmark on an Intel iPSC/860 for various block sizes, numbers of processors and problem sizes. The results in the table show that non-unit block sizes provide the best performance in some cases, reducing execution time by 10% or more. Figure 14 shows an alternate view of these results, plotting measured speedups for different problem and block sizes. Improvements measured here for block-cyclic distributions are related to the following observations. If an array is distributed cyclically then for each column, $(\text{P} - 1)$ processors have to wait for the processor which owns that column to find the maximum element and broadcast it to others. On the other hand, with a block-cyclic distribution, processors own a block of adjacent columns and once a processor begins computing a sequence of pivots it can execute b iterations without waiting for a message. This results in some overlap of communication and computation which reduces the message latency seen by other processors.

From these experiments we conclude that block-cyclic distributions are potentially useful for DGEFA. Clearly, experience with a wider range of codes will be necessary to draw stronger conclusions about the overall effectiveness of block-cyclic distributions.

```
{* Original Fortran D Program *}
SUBROUTINE DGEFA(n,a,ipvt)
  INTEGER n,ipvt(n),j,k,l
  DOUBLE PRECISION a(n,n), al, t
  DISTRIBUTE a(:,BLOCKCYCLIC(b))
  do k = 1, n-1
    {* Find max element in a(k:n,k) *}
S1  l = k
S2  al = dabs(a(k, k))
    do i = k + 1, n
      if (dabs(a(i, k)) .GT. al) then
S3      al = dabs(a(i, k))
S4      l = i
      endif
    enddo
    ipvt(k) = l
    if (al .NE. 0) then
      if (l .NE. k) then
        t = a(l,k)
        a(l,k) = a(k,k)
        a(k,k) = t
      endif
      {* Compute multipliers in a(k+1:n,k) *}
      t = -1.0d0/a(k,k)
      do i = k+1, n
        a(i, k) = a(i, k) * t
      enddo
      {* Reduce remaining submatrix *}
      do j = k+1, n
        t = a(l,j)
        if (l .NE. k) then
          a(l,j) = a(k,j)
          a(k,j) = t
        endif
        do i = k+1, n
S5        a(i, j) = a(i, j) + t * a(i, k)
        enddo
      enddo
    endif
  enddo
  ipvt(n) = n
end
```

```
{* Hand Compiled Output for 4 Processors *}
SUBROUTINE DGEFA(n,a,ipvt)
  INTEGER n,ipvt(n),j,k,l
  DOUBLE PRECISION a(n,n/4), al, t, dp$buf1(n)
  do k = 1, n-1
    owner$proc = MOD((k-1), (b*n$p))/b
    k$ = ((k - 1)/(b*n$p))b + MOD(MOD((k-1), b*n$p), b) + 1
    {* Find max element in a(k:n,k$) *}
    if (my$p .EQ. owner$proc) then
      l = k
      al = dabs(a(k, k$))
      do i = k + 1, n
        if (dabs(a(i, k$)) .GT. al) then
          al = dabs(a(i, k$))
          l = i
        endif
      enddo
      broadcast l, al
    else
      recv l, al
    endif
    ipvt(k) = l
    if (al .NE. 0) then
      if (my$p .EQ. owner$proc) then
        if (l .NE. k) then
          t = a(l,k$)
          a(l,k$) = a(k,k$)
          a(k,k$) = t
        endif
        {* Compute multipliers in a(k+1:n,k$) *}
        t = -1.0d0/a(k,k$)
        do i = k+1, n
          a(i, k$) = a(i, k$) * t
        enddo
      endif
      {* Reduce remaining submatrix *}
      if (my$p .EQ. owner$proc) then
        buffer a(k+1:n, k$) into dp$buf1
        broadcast dp$buf1(1:n-k)
      else
        recv dp$buf1(1:n-k)
      endif
      lb$1 = LowerLoopBound(k+1)
      do j = lb$1, n/4
        t = a(l,j)
        if (l .NE. k) then
          a(l,j) = a(k,j)
          a(k,j) = t
        endif
        do i = k+1, n
          a(i, j) = a(i, j) + t * dp$buf1(i-k)
        enddo
      enddo
    endif
  enddo
  ipvt(n) = n
end
```

Figure 13: DGEFA: Gaussian elimination with partial pivoting.

Program	Problem Size	Proc	Block Size (time in secs)							
			1	2	4	8	16	32	64	128
Gauss	256 × 256	1	*sequential time = 2.200*							
		2	0.974	0.990	0.985	0.982	0.992	1.003	1.084	1.198
		4	0.629	0.619	0.613	0.621	0.644	0.692	0.683	0.783
		8	0.537	0.505	0.485	0.481	0.490	0.598	0.825	1.348
		16	0.627	0.593	0.577	0.580	0.627	0.687	0.892	1.389
		32	0.756	0.717	0.702	0.725	0.728	0.808	0.997	1.495
	512 × 512	1	*sequential time = 17.53*							
		2	7.328	7.364	7.366	7.396	7.471	7.634	7.962	8.616
		4	4.043	4.022	4.035	4.094	4.208	4.440	4.857	5.459
		8	2.462	2.419	2.418	2.460	2.555	2.739	3.133	5.512
		16	2.106	2.010	1.966	1.957	2.058	2.444	3.328	5.627
		32	2.505	2.413	2.371	2.389	2.540	2.839	3.625	5.820
	$1K \times 1K$	1	*estimated sequential time = 140*							
		2	65.69	65.71	65.75	66.10	66.23	66.97	68.59	71.53
		4	34.07	34.09	34.20	34.62	34.99	36.09	38.20	41.96
		8	18.14	18.14	18.26	18.55	19.16	20.34	22.44	24.99
		16	10.46	10.47	10.54	10.78	11.26	12.14	14.31	25.32
		32		8.791	8.695	8.854	9.547	11.32	15.19	25.79

Table 1: Intel iPSC/860 execution times for Gaussian elimination with BLOCK-CYCLIC distribution.

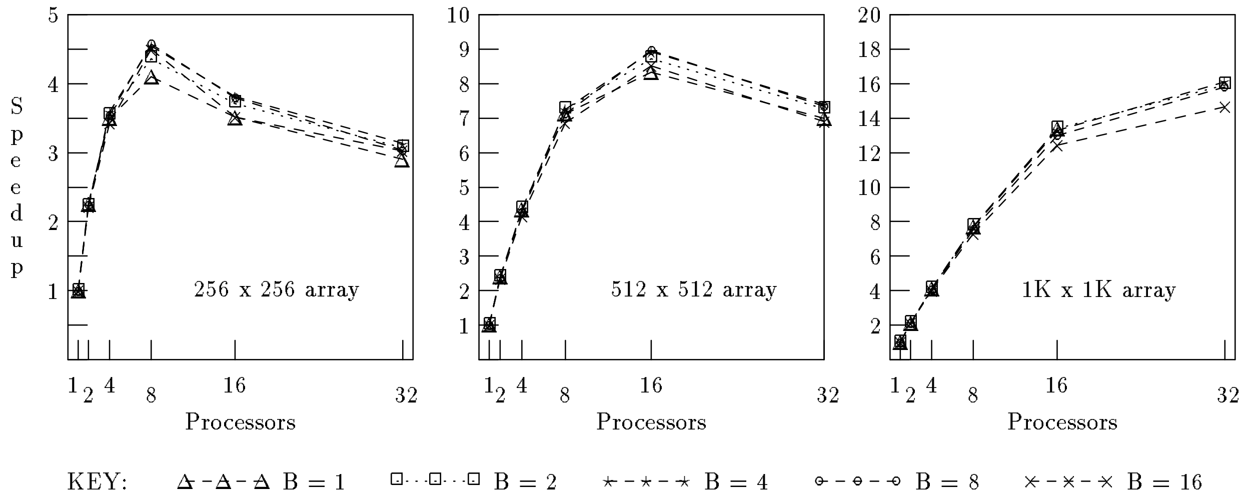


Figure 14: Gaussian elimination speedup results for different problem and block sizes.

7 Conclusions

A usable yet efficient machine-independent parallel programming model is needed to make large-scale parallel machines useful for scientific programmers. We believe that data-placement languages, such as Fortran D and HPF, can provide such a portable data-parallel programming model. The key to achieving good performance with data parallel languages is applying advanced compiler analysis and optimization to automatically exploit parallelism and manage communication. This paper describes techniques that will enable compilers for data parallel languages to handle block-cyclic distributions, extending the class of problems for which good performance can be achieved.

We described the analysis required to compile data parallel programs with block-cyclic distributions in the presence of symbolic loop bounds and array sizes, non-unit loop strides and variable number of processors. We also gave algorithms for computing the data elements that need to be communicated both for the loops with unit and non-unit strides, and presented a linear-time algorithm for computing non-constant local memory access pattern for loops with non-unit stride.

In our experimental results evaluating the effectiveness of block-cyclic distributions for DGEFA, using non-unit block sizes provided some improvements in performance in particular cases. However, larger improvements in performance could be realized if the number of messages could be reduced. In section 4, we described how message vectorization could be used with a block-cyclic distribution to reduce the number of messages. The DEFA code shown in Figure 13 contains a loop-carried dependence from S_5 to S_2/S_3 that complicates application of message vectorization. To employ message vectorization in this case, the compiler would need to restructure the computation. Since the pivot for a particular column does not depend on the columns occurring after it, the compiler could minimize the computation that needs to be done before broadcasting the pivots for a block of columns by deferring executing of statement S_5 for columns outside the current block until after the broadcast. To increase the computation and communication overlap, the processor also could initially compute and broadcast the pivot for only the first column in a block and then compute and broadcast the pivots for the rest of the columns in the block in a second message. By broadcasting the first pivot as soon as possible, other processors can perform elimination steps while the other processor computes the pivots for the rest of the columns. This optimization would maximize the communication and computation overlap. A similar optimization was discussed by Adve, *et. al.* [1] in the context of cyclic distributions.

To evaluate the potential benefits of block-cyclic distributions for data parallel codes, it is clear that further work is necessary. The techniques we have presented here provide a foundation for that work.

8 Acknowledgments

The authors wish to thank Reinhard von Hanxleden, Uli Kremer and Chau-Wen Tseng for their assistance, and Elana Granston and the referees for providing detailed comments and suggestions. This research was supported in part by the IBM corporation, the Center for Research on Parallel Computation (CRPC) through NSF Cooperative Agreement Number CCR-9120008, and ARPA under contract #DABT63-92-C-0038. Use of the Intel iPSC/860 was provided by the CRPC NSF Cooperative Agreement Nos. CCR-8809615 and CDA-8619893 with support from the Keck Foundation.

References

[1] V.S. Adve, C. Koelbel, and J.Mellor-Crummey. Performance analysis of data parallel programs. *Submitted to Supercomputing '94*, April 1994.

[2] F. Bodin, E.D. Granston, and T. Montaut. Experiences reducing false sharing in shared virtual memory systems (*under preparation*). Technical report, Center for Research on Parallel Computation, Rice University, 1994.

[3] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[5] J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 372–379, Williamsburg, VA, April 1992.

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

[7] S.K.S. Gupta, S.D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II, pages 301–305, 1993.

[8] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. Technical report OSU-CISRC-4/94-TR19, Department of Computer and Information Science, Ohio State University, Columbus, OH, March 1994.

[9] R. v. Hanxleden. Handling irregular problems with fortran d – a preliminary report. Technical Report CRPC-TR93339-S, Center for Research on Parallel Computation, Rice University, October 1993.

[10] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

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

[12] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Advanced compilation techniques for fortran d. Technical Report CRPC-TR93338, Center for Research on Parallel Computation, Rice University, October 1993.

[13] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[14] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[15] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[16] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

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

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

[19] C. Koelbel and P. Mehrotra. Compiling global name-space programs for distributed execution. ICASE Report 90-70, Institute for Computer Application in Science and Engineering, Hampton, VA, October 1990.

[20] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[21] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[22] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[23] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.