Advanced Compilation Techniques for Fortran D

Semma Hiranandani,
Ken Kennedy, John M. Crummy,
and Ajay Sethi

CRPC-TR93338
October 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Advanced Compilation Techniques for Fortran D

Seema Hiranandani
Ken Kennedy
John Mellor-Crummey
Ajay Sethi
Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

The Rice Fortran 77D compiler uses data decomposition specifications to automatically translate Fortran 77 programs for execution on MIMD distributed-memory machines. The current compiler prototype performs communication and parallelism optimizations in the presence of single-dimensional block distributions with constant values specified for array dimensions, loop bounds and the number of processors. The prototype also contains partial support for cyclic distributions under the same restrictions. This paper describes techniques that will enable the Fortran 77D compiler to generate efficient code for more complex programs that may contain symbolic loop bounds and array sizes, loops with non-unit strides and multidimensional block, cyclic and block-cyclic distributions. The generated code will allow the number of processors executing a Fortran 77D program to be a run-time parameter. Finally, we present empirical results for a hand-compiled test case show that block_cyclic data distributions can be effective for load balancing.

1 Introduction

The goal of the Fortran D language is to provide a simple yet efficient machine-independent parallel programming model. To enable programmers to write data-parallel programs that can be compiled and executed with good performance on a range of architectures, Fortran D shifts much of the burden of machine-dependent optimization to the compiler. To evaluate the Fortran D programming model, we implemented a prototype compiler based on the software infrastructure developed for the ParaScope programming environment [3]. While the current compiler prototype has enabled validation of the importance of communication and parallelism optimizations, it has many limitations that prevent it from successfully compiling many data-parallel programs. The restrictions in the current prototype are as follows. First, loop index variables must have bounds that are compile time constants and stride of either one or minus one. Second, arrays may be distributed in only one dimension and the distribution type must be block. Third, the number of processors executing a Fortran 77D program must be a compile-time constant. In this paper we describe algorithms, data structure enhancements, and a run-time library that will significantly improve and extend the analysis of the compiler and successfully relax the restrictions on its input programs.

Previous work has described algorithms for partitioning data and computation in the Rice Fortran 77D compiler, as well as its optimization and validation strategy [9]. Internal representations, program analysis, message vectorization, pipelining, and code generation algorithms for block distributions were presented elsewhere [8]. The principal contribution of this paper is to provide algorithms and techniques to compile complex Fortran D programs that may contain variable number of processors, symbolic loop bounds and array sizes, loops with non unit strides and multidimensional block, cyclic and block_cyclic distributions. Section 2 briefly reviews the organization of the prototype Rice Fortran 77D compiler. Section 3 reviews some terminology and notation used throughout the rest of the paper. Sections 4–8 present algorithms and analyses necessary to compile complex Fortran D programs. Section 9 presents some experimental results that assess the effectiveness of using a block-cyclic data distribution for load balancing in Gaussian elimination. The paper concludes with a short summary.

2 Fortran D Compiler

There are two major steps in compiling Fortran D for MIMD distributed-memory machines. The first step is partitioning the data and computation among the available set of processors. The second is introducing communication operations to transfer values as necessary. A simple compilation technique known as *run-time resolution* yields code that explicitly calculates the ownership and communication for each reference at run time [1, 14, 18], but resulting programs are likely to execute significantly slower than the original sequential code. By using aggressive compile-time analysis and optimization, the Fortran 77D compiler can generate much more efficient programs. Below, we briefly review the sequence of steps performed by the Rice Fortran 77D compiler; details of the compilation process are described elsewhere [8, 9]:

1) Analyze program The compiler performs scalar data-flow analysis, symbolic analysis, and dependence testing to determine the type and level of all data dependences [11].

2) Partition data The compiler analyzes Fortran D data decomposition specifications to determine the decomposition of each array in a program. Alignment and distribution statements are used to calculate the array section owned by each processor.

3) Partition computation The compiler partitions computation among the processors using the "owner computes" rule—each processor only computes values of data it owns [1, 14, 18]. 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 work partition computed above is used to calculate the non-local data accessed by each processor for each right-hand side (*rhs*) reference to a distributed array. References that result in non-local accesses are marked since they require communication to be inserted.

5) Optimize communication The compiler examines each marked non-local reference, using 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 [7]

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 instances of non-local data.

7) Generate code Finally, the Fortran D compiler uses the results of previous analysis and optimization to generate a *single-program, multiple-data* (SPMD) program that uses message passing to communicate values as necessary. This program can execute directly on the nodes of a MIMD distributed-memory machine. To generate the SPMD program, 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. For the following canonical loop nest,

```
do $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$
  $A(g(\vec{k})) = B(h(\vec{k}))$                                        (equation 1)
enddo
```

we define the sets (formal definitions of these sets are presented elsewhere [9]):

- $\mathit{image\_set}_B(t_p)$ is the set of indices of array B that cause a reference to a data element owned by processor t_p.
- $\mathit{iter\_set}_A(t_p)$ is the set of loop iterations that cause reference A to access data owned by processor t_p.
- $\mathit{index\_set}_B(t_p)$ is the set of indices of array B referenced by processor t_p on loop iterations contained in $\mathit{iter\_set}_A(t_p)$.
- $\mathit{in\_index\_set}_B(t_p)$ is the set of non local indices of array B referenced by processor t_p on loop iterations contained in $\mathit{iter\_set}_A(t_p)$.
- $\mathit{out\_index\_set}_B(t_p)$ is the set of indices of array B contained in $\mathit{image}_B(t_p)$ and referenced by other processors.
- $\mathit{send\_p\_set}_B(t_p)$ is the set of processors to whom t_p must send local elements of array B.
- $\mathit{receive\_p\_set}_B(t_p)$ is the set of processors from whom t_p must receive values of non-local elements of array B.
- $\mathit{rsd\_set}_B$ is the set of indices of array B that are referenced in the loop nest.
- $\mathit{out\_set}_A(t_p)$ is the set of indices of array A that cause local elements of array B on processor t_p to be referenced by other processors.
- P is the number of processors, numbered 0 . . . P-1.

4 Program Analysis

4.1 Partitioning Analysis

To perform partitioning analysis, the current Fortran 77D compiler prototype requires constant loop bounds, array dimensions, and number of processors. Here, we 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.

4.1.1 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.

```
Given:
  $a(l_1 : u_1, \ldots, l_n : u_n), b(l_1 : u_1, \ldots, l_n : u_n)$
  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 $d(d_1, \ldots, d_n)$
Loop nest:
  do $i_1 = lb_1, ub_1$
    $\ldots$
    do $i_n = lb_n, ub_n$
      $\ldots$
S1        $a(\ldots, g(i_k), \ldots) = b(\ldots, f(i_k), \ldots)$
    enddo
    $\ldots$
  enddo
```

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, it is convenient to consider decompositions with only one distributed dimension d_m. For arrays with more than one distributed dimension, the iteration set can be computed by interesecting the solutions for each single-distributed-dimension sub-problem.

Here we show how to compute the iteration set for a decomposition d in which dimension d_m is distributed using either a block or cyclic distribution; in section 6, we show how to compute the iteration set if d_m is distributed using a block_cyclic distribution. We assume 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, we defer computation of the iteration set to run time.

Block Distribution

Below, we describe the iteration sets constructed for block distributions based on the subscript type. We assume that d_m is distributed block wise and the block size $bs = \lceil (u_k - l_k + 1)/P \rceil$

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

```
if $(\lfloor (c_k - l_k)/bs \rfloor = t_p)$
    $iter\_set(t_p) = (lb_1 : ub_1, \ldots, lb_k : ub_k, \ldots, lb_n : ub_n)$
else
    $iter\_set(t_p) = \emptyset$
```

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

```
$lb_{k1} = max(t_p * bs + l_k, lb_k)$
$ub_{k1} = min((t_p + 1) * bs + l_k - 1, ub_k)$
$iter\_set(t_p) = (lb_1 : ub_1, \ldots, lb_{k1} : ub_{k1}, \ldots, lb_n : ub_n)$
```

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

```
$lb_{k1} = max(t_p * bs - c + l_k, lb_k)$
$ub_{k1} = min((t_p + 1) * bs - c + l_k - 1, ub_k)$
$iter\_set(t_p) = (lb_1 : ub_1, \ldots, lb_{k1} : ub_{k1}, \ldots, lb_n : ub_n)$
```

- **Linear Expression:** $g(i_k) \equiv c_1 * i_k + c_0$

$$lb_{k1} = max(lb_k, t_p * bs + l_k, \lceil \frac{t_p*bs+l_k-c_0}{c_1} \rceil)$$
$$ub_{k1} = min(ub_k, (t_p+1) * bs + l_k - 1, \lfloor \frac{(t_p+1)*bs+l_k-1-c_0}{c_1} \rfloor)$$
$$iter\_set(t_p) \;=\; (lb_1 : ub_1, \ldots, lb_{k1} : ub_{k1}, \ldots, lb_n : ub_n)$$

Cyclic Distribution

Below, we describe the iteration sets constructed for cyclic distributions based on the subscript type. We assume that dimension d_m in the loop nest above is distributed with a cyclic distribution.

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

```
if $(((c_k - l_k) \bmod P) = t_p)$
    $iter\_set(t_p) = (lb_1 : ub_1, \ldots, lb_k : ub_k, \ldots, lb_n : ub_n)$
else
    $iter\_set(t_p) = \emptyset$
```

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

```
if $(t_p \;<\; (lb_k \;-\; l_k) \bmod$ P) then
    $lb_{k1} = \lceil \frac{lb_k - l_k}{P} \rceil * P \;+\; 1 + \; t_p$
else
    $lb_{k1} = \lfloor \frac{lb_k - l_k}{P} \rfloor * P \;+\; 1 + \; t_p$
endif
if $(t_p \;<\; (ub_k \;-\; l_k) \bmod$ P) then
    $ub_{k1} = \lfloor \frac{ub_k - l_k}{P} \rfloor * P \;+\; 1 + \; t_p$
else
    $ub_{k1} = \lceil \frac{ub_k - l_k}{P} \rceil * P \;-\; P \;+\; 1 + \; t_p$
endif
$iter\_set(t_p) = (ub_1 : ub_1, \ldots, lb_{k1} : ub_{k1} : P, \ldots, lb_n : ub_n)$
```

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

```
if $(t_p \;<\; (lb_k \;+\; c \;-\; l_k) \bmod$ P) then
    $lb_{k1} = \lceil \frac{lb_k + c - l_k}{P} \rceil * P \;+\; 1 + \; t_p$
else
    $lb_{k1} = \lfloor \frac{lb_k + c - l_k}{P} \rfloor * P \;+\; 1 + \; t_p$
endif
if $(t_p \;<\; (ub_k \;-\; l_k) \bmod$ P) then
    $ub_{k1} = \lfloor \frac{ub_k + c - l_k}{P} \rfloor * P \;+\; 1 + \; t_p$
else
    $ub_{k1} = \lceil \frac{ub_k + c - l_k}{P} \rceil * P \;-\; P \;+\; 1 + \; t_p$
endif
$iter\_set(t_p) = (ub_1 : ub_1, \ldots, lb_{k1} : ub_{k1} : P, \ldots, lb_n : ub_n)$
```

- **Linear Expression:** $g(i_k) \equiv c_1 * i_k + c_0$

  To compute the iteration set, we first convert the subscript with linear expression 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 $lb_k * c_1 \;+\; c_2$, $ub_k * c_1 \;+\; c_2$ and c_1 respectively.

  To compute the iteration set, we need to determine the first iteration assigned to the processor. This is equivalent to finding the smallest non-negative integer j such that

$$(lb_k \;+\; c_1 * j \;-\; l_k) \;\bmod\; P \;=\; t_p$$

```
Given:  subscript, loop and decomposition information for lhs
 if
   constant loop bounds, simple subscript expression and
   standard distributions (block, block_cyclic, cyclic) then
   classify iteration set as Iter_simple
 else if
   symbolic loop bounds, subscript expressions are linear
   and the decomposition type is either block,
   block_cyclic or cyclic then classify iteration set as
   Iter_symbolic.
   Runtime library routines will be used to generate bounds
 else
   classify iteration set as Iter_complex.  The loop bounds will
   not be reduced.  Instead, run time guards will be generated
   to conform to the owner computes rule
```

Figure 1: Classification of Iteration Sets

The above equation reduces to

$$lb_k \,+\, c_1 * j \,-\, l_k \,-\, n * P \,=\, t_p$$

with an additional non-negative unknown n. This is equivalent to

$$c_1 * j \,-\, n * P \,=\, t_p \,-\, (lb_k \,-\, l_k)$$

We can use extended Euclid algorithm to find the general solution of this linear Diophantine equation. (Details of this solution can be found in the literature [12, 13].)

During the iteration set construction phase, we also classify each iteration set as one of the following: Iter_simple, Iter_symbolic or Iter_complex. The compiler uses the iteration type information to optimize the code generated. We describe the classification algorithm in the next section.

4.1.2 Classification of Iteration Sets

Figure 1 provides an algorithm for classifying iteration sets depending on the complexity of the loop and subscript information. The prototype compiler further classifies Iter_simple iteration sets to reflect on the type of boundary conditions that occur for block distributions. We will extend the boundary classification scheme to include cyclic and block_cyclic distributions.

4.2 Communication Analysis

4.2.1 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. The current implementation builds index sets for arrays that are distributed block or cyclic in a single dimension and contain simple subscript expressions. Furthermore, the loop bounds must be compile-time constants (triangular loops are allowed for cyclic). Below we extend the construction of index sets to include symbolic bounds with unit and non-unit strides for block and cyclic distributions. Index sets for block_cyclic distributions are computed in Section 6.

From the previous section, we observe that iter_set(t_p) is either $\emptyset$, $(lb_1 : ub_1, \ldots, lb_k : ub_k, \ldots, lb_n : ub_n)$ or $(lb_1 : ub_1, \ldots, lb_k : ub_k : s_k, \ldots, lb_n : ub_n)$. We now give the formulae to compute the index set for each case :

Case 1 : $iter\_set(t_p) \equiv \emptyset$
index_set$(t_p) = \emptyset$

Case 2 : $iter\_set(t_p) \equiv (lb_1 : ub_1, \ldots, lb_k : ub_k, \ldots, lb_n : ub_n)$
In this case, the index set is computed by substituting in the value of the $iter\_set(t_p)$. Note that the index set is independent of the distribution of the right-hand side array and that dimension l of array `b` is distributed.

- **Constant:** $f(i_l) \equiv c_l$

$$index\_set_b(t_p) \;=\; (lb_1 : ub_1, \ldots, c_l, \ldots, lb_n : ub_n)$$

- **Induction Variable Only:** $f(i_l) \equiv i_l$

$$index\_set_b(t_p) = (lb_1 : ub_1, \ldots, lb_l : ub_l, \ldots, lb_n : ub_n)$$

- **Simple Linear Expression:** $f(i_l) \equiv i_l + c$

$$index\_set_b(t_p) = (lb_1 : ub_1, \ldots, lb_l + c : ub_l + c, \ldots, lb_n : ub_n)$$

- **Linear Expression:** $f(i_l) \equiv c_1 * i_l + c_0$

$$index\_set(t_p) = (lb_1 : ub_1, \ldots, c_1 * lb_l + c_0 : c_1 * ub_l + c_0 : c_1, \ldots, lb_n : ub_n)$$

Case 3: $iter\_set(t_p) \equiv (lb_1 : ub_1, \ldots, lb_k : ub_k : s_k, \ldots, lb_n : ub_n)$ As in Case 2, the index sets are computed by substituting in the value of $iter\_set(t_p)$. The only interesting case is the Linear Expression case : f(i$_l$) $\equiv$ $c_1 * i_l + c_0$. For the linear expression, we have two subcases :

$l \neq k$ `(WLOG, we assume` $l < k$`) :`

$$index\_set(t_p) = (lb_1 : ub_1, \ldots, c_1{*}lb_l{+}c_0 : c_1{*}ub_l{+}c_0 : c_1, \ldots, lb_k : ub_k : s_k, \ldots, lb_n : ub_n)$$

$l = k$:

$$index\_set(t_p) = (lb_1 : ub_1, \ldots, c_1 * lb_l + c_0 : c_1 * ub_l + c_0 : c_1 : s_k * c_1, \ldots, lb_n : ub_n)$$

If $iter\_set(t_p)$ can not be determined at compile time, the index set computation will be performed at run time.

4.2.2 Communication Classification

Communication classification is crucial to our compilation strategy since it allows us to insert calls to fast collective communication primitives in the output program and optimize the communication at compile time whenever possible. We have redesigned the communication classification algorithm for the Fortran D compiler. The algorithm as implemented examines the subscript expressions of non-local references to determine the type of communication. Since the compiler does not examine the loop bounds, it occasionally classifies the resulting communication type incorrectly. By using the index, iteration and rsd sets in our algorithm we are able to capture the loop and subscript information and hence correctly classify the communication.

The classification algorithm is depicted in Figure 2. Note, at this point we assume that the iteration and index sets have already been constructed. The algorithm works as follows: Firstly, each non-local reference is classified as resulting in one of the following type of communication.

- Single Send Receive

- Shift
- Broadcast
- Gather
- All-to-All
- Inspector/Executor
- Run Time Resolution

The *in_index_set*, *out_index_set*, *send_p_set*, and *receive_p_set* are constructed based on the results of the communication classification algorithm. The *rsd_set*, *iter_set*, and *index_set* have a *type* associated for each dimension of the array they represent. If the type is `constant`, only a single index in that dimension is referenced. The function `shift_range` works as follows: $iter\_set_{Al}$ and $index\_set_{Bk}$ is of type `range` if we are able to represent the section accessed in triplet notation, (lo:up:step). If $iter\_set_{Al}$ and $index\_set_{Bk}$ are of type range, represented as $(lo_l : up_l : step_l)$ and $(lo_k : up_k : step_k)$ respectively and $(up_k - up_l) = (lo_k - lo_l)$ and $(step_k = step_l)$ then `shift_range` returns true. `all_procs` examines a dimension of the *rsd_set* and returns true if `proc_set`(*rsd_set*) includes all the processors. `proc_set` returns the set of processors that own indices $\in$ *rsd_set*. `one_proc` examines a dimension of the *rsd_set* and returns true if it consists of indices owned by a single processor.

5 Multidimensional Block & Cyclic Distributions

In previous sections we computed the index and iteration sets for block and cyclic distributions. Algorithms to construct IN and OUT index sets in the general case are described in [9].

Here, we present algorithms to construct specialized in_index, out_index, receive_p and send_p sets. These sets are constructed based on the communication type classified by the algorithm in Figure 2. By constructing specialized sets, the compiler will be able to generate optimized communication whenever possible and invoke fast collective communication primitives when needed. For Multidimensional distributions we need to specify the processor layout in more than one dimension. We will use the HPF PROCESSOR directive to specify the layout.

5.1 IN Sets

The IN index set construction for multidimensional distributions is presented in Figure 3. The algorithm works as follows. We partition each dimension of the index_set based on the communication and distribution type. In the case of `shift`, `send receive` and `broadcast`, we partition the index_set such that each section belongs to a single processor. However, in the case of `all-to-all` or `gather` communication we do not break the index_set into sections since we will use fast collective communication primitives to handle the messages at run time whereas for `shift` and `send receive` the compiler may be able to determine the exact communication at compile time and thus generate more efficient code. The partitioned index_set is stored in in_index_set_dim. The in_index_set is the product set of in_index_set_dim and contains all the non-local sections. The receive_p_set contains m-tuples constructed by the projection $\prod : D \rightarrow out\_index\_set_{i1} \times .... \times out\_index\_set_{im}$ where i1, ... im are distributed dimensions of the array. The projection function creates for each n-tuple $\in$ in_index_set_dim, an m-tuple containing the receive processor co-ordinates.

5.2 OUT Sets

IN sets contain receive information. In order to generate deadlock free correct code, we need to construct OUT sets that contain send information. The OUT index set construction for multidimensional distributions is presented in Figure 4. image_set$_B(t_p)$ is the set of indices of array `B` that are owned by t_p and referenced in the loop nest. The function f maps the elements $\in$ image_set$_B(t_p)$ to the indices of the left hand side array `A` which cause the section of `B` to be referenced. These indices are stored in out_set. The out_set is partitioned based on the communication and distribution type. The elements are then mapped to the image_set using the inverse function (f$^{-1}(\vec{k})$) and stored in out_index_set_dim. The out_index_set

```
Given:  subscript, loop, decomposition, and dependence information
Loop nest in equation 1
A & B are mapped to the same decomposition
for $k = 1, dim_n$
if dim$_k$ of B is distributed
  calculate dim$_l$ of A and dim$_k$ of B mapped to dim$_i$ of the decomposition
  rsd_set$_{Bk}$ = section of B accessed in dim$_k$
  rsd_set$_{Al}$ = section of A accessed in dim$_l$
  iter_set$_{Al}(p_j)$ = iterations to be executed by $p_j$
  index_set$_{Bk}(p_j)$ = array section accessed by $p_j$
  if (type(rsd_set$_{Bk}$) = constant) & (type(rsd_set$_{Al}$) = constant)
    ctype$_k$ = SEND RECV
  else if shift_range(iter_set$_{Al}(p_j)$,index_set$_{Bk}(p_j)$)
    ctype$_k$ = SHIFT
  else if all_procs(rsd_set$_{Al}$) & one_proc(rsd_set$_{Bk}$)
    ctype$_k$ = BROADCAST
  else if all_procs(index_set$_{Bk}(p_j)$) & all_procs(rsd_set$_{Al}$)
    ctype$_k$ = ALL-TO-ALL
  else if one_proc(rsd_set$_{Al}$) & all_procs(rsd_set$_{Bk}$)
    ctype$_k$ = GATHER
  else if no loop carried true cross processor dependences
    ctype$_k$ = INSPECTOR/EXECUTOR
  else
    ctype$_k$ = RUNTIME RESOLUTION
endif
endfor
```

Figure 2: Communication Classification Algorithm

is the product set of out_index_set_dim. The send_p_set contains m-tuples constructed by the projection $\prod : D \rightarrow out\_index\_set_{i1} \times .... \times out\_index\_set_{im}$ where i1,...,im are distributed dimensions of the array.

5.3 Example

We use simple examples to illustrate the algorithms in Figures 3 and 4. In the first example the arrays A and B are distributed (block, block) and in the second example, they are distributed (block, cyclic). The loop nests and array sections accessed by each processor are shown in figure 5. We have written the loop nest in HPF to illustrate the use of PROCESSOR directive. For simplicity, we will use the algorithms to partition the **in_index_set**, **out_index_set**, **send_p** and **receive_p** sets for processor 2 whose coordinates are (1,0). P_2 is the size of the processor grid in the second dimension. my_c_2 is the processor co-ordinate in the second dimension.

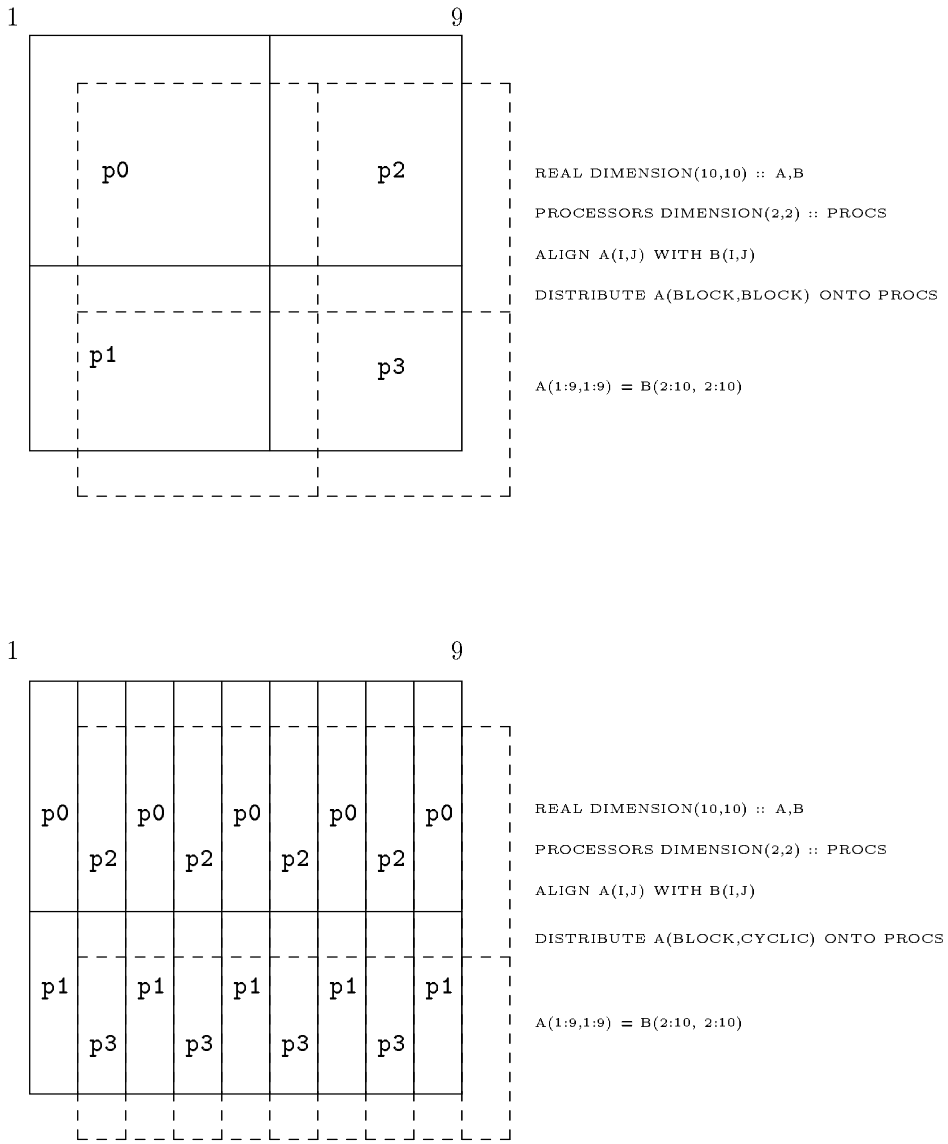
$\{\star$ **in_index_set partition for (BLOCK,BLOCK) distribution**$\star\}$
index_set$(2) = (((2:6),(6:10)))$
index_set_dim$_1(2) = ((2:5),(6:6))$
index_set_dim$_2(2) = ((6:10))$
in_index_set$(2) = (((6:6),(6:10)))$
send_p_set$(2) = ((1,1))$

```
Given :  index_set$_B$ for each processor
dtype = distribution type for each dimension
ctype = communication type for each dimension
bs = block size for each dimension distributed BLOCK
dim$_n$  =  # of dimensions of B
Calculate:  in_index_set$_B(t_p)$
{$\star$ Partition each dim of index set based on communication type $\star$}
 for j = 1, dim$_n$
   l:u:s $\in$ index_set$_{Bj}(t_p)$
   if (dtype$_j$ = ``:''  ) i.e.  not distributed
     in_index_set_dim$_j(t_p) = \cup$ (l:u:s)
   if (dtype$_j$ = BLOCK)
     if (ctype$_j$ = (Shift $\vee$ Send Recv $\vee$ Broadcast))
       While (l < u)
       in_index_set_dim$_j(t_p) = \cup$ (l:min(u, (l + bs$_j$- l)mod bs$_j$:1))
       l = l + min(u, l + bs$_j$ - l mod bs$_j$) + 1
     end if
     if (ctype$_j$ = (All-To-All $\vee$ Gather))
     in_index_set_dim$_j(t_p) = \cup$  (l:u:s)
   end if
   if (dtype$_j$ = CYCLIC)
     in_index_set_dim$_j(t_p) = \cup$  (l:u:s)
   endif
 endfor
{$\star$ Calculate in_index_set$_B(t_p)$ $\star$}
in_index_set$_B(t_p)$ = $\prod_{i=1}^{dim_n}$ in_index_set_dim$_i(t_p)$
for each $n$-tuple $< l_1 : u_1 : s_1, ...., l_n : u_n : s_n > \in \prod_{i=1}^{dim_n}$ in_index_set_dim$_i(t_p)$
{$\star$ Calculate receive processor co-ordinates $\star$}
 k = 0
 for i = 1, dim$_n$
   if dim$_i$ is distributed
     k = k + 1
     if (ctype$_i$ = (All-To-All $\vee$ Gather)) b$_k$ = $*$ i.e.  all processors
     if (dtype$_i$ = BLOCK) b$_k$ = l$_i$/bs$_i$
     else if (dtype$_i$ = CYCLIC) b$_k$ = l$_i$  mod  bs$_i$
 endfor
{$\star$ Add the m-tuple containing the processor co-ordinates$\star$}
 if $(t_p \neq < b_1, ...., b_m >)$
   receive_p_set$_B(t_p)$ =  $\cup$  $< b_1, ..., b_m >$
 else
   delete $n$-tuple $< l_1 : u_1 : s_1, ...., l_n : u_n : s_n >$
endfor
```

Figure 3: in_index_set and receive_p_set for multidimensional block and cyclic distributions

```
Given :  image_set$_B(t_p)$, data referenced and owned by processor t$_p$
dtype = distribution type for each dimension
ctype = communication type for each dimension
dim$_n$ = # of dimensions of B
bs = block size for each dimension distributed BLOCK
Calculate:  out_index_set$_B(t_p)$
{$\star$ construct out_set$_A(t_p)$ $\star$}
out_set$_A(t_p)$ = section of lhs accessed under image_set$_B(t_p)$
image_set$_B(t_p) \overset{f(\vec{k})}{\Longrightarrow}$ out_set$_A(t_p)$
{$\star$ partition each dim of out_set$_B$ based on communication type $\star$}
 for j = 1, dim$_n$
   l:u:s $\in$ image_set$_{Bj}(t_p)$
   if (dtype$_j$ = ``:''  )  i.e.  not distributed
   out_index_set_dim$_j(t_p) = \cup$ (l:u:s)
   else l:u:s $\in$ out_set$_A$(t$_p$)
   if (dtype$_j$ = BLOCK)
     if (ctype$_j$ = (Shift $\vee$ Send Recv $\vee$ Broadcast))
       While (l < u)
       out_index_set_dim$_j(t_p) = \cup$ (l:min(u,l+bs$_j$- 1 mod bs$_j$:1))
       l = l + min(u, l + bs$_j$ - 1 mod bs$_j$) + 1
     end if
     if (ctype$_j$ = (All-To-All $\vee$ Gather))
       out_index_set_dim$_j(t_p) = \cup$ $f^{-1}(j)$(l:u:s)
   end if
   if (dtype$_j$ = CYCLIC)
     if (ctype$_j$ = (Shift $\vee$ Send Recv $\vee$ Broadcast))
     out_index_set_dim$_j(t_p) = \cup$ $f^{-1}(j)$ (l:u:s)
   endif
 endfor
{$\star$ Calculate out_index_set$_B$ $\star$}
out_index_set$_B(t_p)$ = $\prod_{i=1}^{dim_n}$ out_index_set_dim$_i(t_p)$
for each $n$-tuple $< l_1 : u_1 : s_1, ..., l_n : u_n : s_n > \in \prod_{i=1}^{dim_n}$out_index_set_dim$_i$
{$\star$ Calculate send processor co-ordinates $\star$}
 k = 0
 for i = 1, dim$_n$
   if dim$_i$ is distributed
   k = k + 1
     if (dtype$_i$ = BLOCK) p$_i$ = $f$(l$_i$)/bs$_i$
     else if (dtype$_i$ = CYCLIC) p$_i$ = $f$(l$_i$)  mod  bs$_i$
       if (ctype$_i$ = (Shift $\vee$ Send Recv $\vee$ Gather)) $\vee$)) b$_k$ = p$_i$
     else if (ctype$_i$ = Broadcast $\vee$ All-to-All)
       b$_k$ = $*$, i.e.  all processors
 endfor
{$\star$ Add the m-tuple containing the processor co-ordinates $\star$}
 if $(t_p \neq < b_1, ...., b_m >)$
   send_p_set$_B(t_p)$ = $\cup$ $< b_1, ..., b_m >$
 else
   delete $n$-tuple $< l_1 : u_1 : s_1, ...., l_n : u_n : s_n >$
endfor
```

Figure 4: out_index_set and send_p_set for multidimensional block and cyclic distributions



```
REAL DIMENSION(10,10) :: A,B
PROCESSORS DIMENSION(2,2) :: PROCS
ALIGN A(I,J) WITH B(I,J)
DISTRIBUTE A(BLOCK,BLOCK) ONTO PROCS

A(1:9,1:9) = B(2:10, 2:10)
```

```
REAL DIMENSION(10,10) :: A,B
PROCESSORS DIMENSION(2,2) :: PROCS
ALIGN A(I,J) WITH B(I,J)
DISTRIBUTE A(BLOCK,CYCLIC) ONTO PROCS

A(1:9,1:9) = B(2:10, 2:10)
```

Figure 5: Multidimensional Distributions

{⋆out_index_set partition for (BLOCK, BLOCK) distribution⋆}
$\texttt{out\_index\_set}(2) \;=\; (((1:5),(6:7)))$
$\texttt{out\_index\_set\_dim}_1(2) \;=\; ((1:5))$
$\texttt{out\_index\_set\_dim}_2(2) \;=\; ((6:7))$
$\texttt{out\_index\_set}(2) \;=\; ((1:5),(6:7))$
$\texttt{receive\_p\_set}(2) \;=\; ((0,0))$

{⋆in_index_set partition for (BLOCK,CYCLIC) distribution⋆}
$\texttt{index\_set}(2) \;=\; (((2:6),(3:10:2)))$
$\texttt{index\_set\_dim}_1(2) \;=\; ((2:5),(6:6))$
$\texttt{index\_set\_dim}_2(2) \;=\; ((3:10:2))$
$\texttt{in\_index\_set}(2) \;=\; (((2:5),(3:10:2)),((6:6),(3:10:2)))$

$$\texttt{send\_p\_set}(2) \;=\; ((0,(my\_c_2+1) \mod P_2),(1,(my\_c_2+1) \mod P_2))$$

{⋆out_index_set partition for (BLOCK, CYCLIC) distribution⋆}

$\texttt{out\_index\_set}(2) \;=\; (((1:5),(2:10:2)))$
$\texttt{out\_index\_set\_dim}_1(2) \;=\; ((1:5))$
$\texttt{out\_index\_set\_dim}_2(2) \;=\; ((2:10:2))$
$\texttt{out\_index\_set}(2) \;=\; ((1:5),(2:10:2))$
$\texttt{receive\_p\_set}(2) \;=\; ((0,0))$

6 Block_Cyclic Distribution

In this section we present the analyses necessary for compiling codes with block_cyclic distributions. The SPMD code generated in presence of the block_cyclic distributions is significantly more complicated than that generated for either block or cyclic distribution.

6.1 Extensions to RSDs

First of all, we note that that the RSDs used for representing block and cyclic distributions can not represent the block_cyclic distributions (e.g. consider the array elements of array A owned by the Proc(0) (where P = 2) in Figure 8 : [1:3]+[7:9]+[13:15]+. . .). block_cyclic distributions do not result in convex regions. Therefore, it is not possible to represent such distributions using the closed form Fortran 90 triplet notation, used for block and cyclic distributions.

For the sake of convenience, we use the following notation. In order to accommodate block_cyclic distributions, we add another parameter, the block size, to the representation of the regular section descriptors.

Definition : A block_cyclic set [l:u:b:s] consists of the following :

- l = lower bound of the section,
- u = upper bound of the section,
- b = block size given in the block_cyclic distribute directive and
- s = the stride which is equal to b*P.

Using this definition, [1:3]+[7:9]+[13:15]+. . . is represented as [1:n:3:6]. The above-mentioned definition can be used to represent the block and cyclic distributions as before by setting s=n and b=$\lfloor \frac{n}{P} \rfloor$ for block distributions and s=P and b=1 for cyclic distributions.

We point out that this notation can not be used to represent the iteration sets. For example, in the loop nest below,

```
REAL X(n), Y(n)
DECOMPOSITION d(n)
ALIGN X, Y with d
DISTRIBUTE d(BLOCK_CYCLIC(b))
do i = $l_i$, $u_i$, $s_i$
    X(i) = $\mathcal{F}$(Y(i))
enddo
```

the first iteration executed by a processor depends on l_i which, in turn, determines the number of iterations executed by the processor the first time around. Therefore, we need to have two different block sizes and the above-mentioned notation can not represent this case. Also, if the step $s_i \neq 1$, then the memory access gap is non-constant (we handle this case in section 6.6) and once again the notation is insufficient.

Despite these shortcomings, we use the notation mentioned above for the sake of convenience with the understanding that the compiler will handle these cases by using appropriate data-structures to store boundary conditions. The current compiler successfully stores boundary conditions for block distributions [8]. Another way to handle this is to peel-off the first set of iterations and represent the remaining iterations using the notation described in this section.

```
     REAL A(n,n)
     DISTRIBUTE A(:, BLOCK_CYCLIC(8))
     do k = 1, n
         do i = k+1, n
$S_1$            A(i,k) = $\mathcal{F}$(A(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 6: Code Generation with Block-cyclic distribution

6.2 Partition & Communication Analysis

Consider the program depicted in Figure 6. Readers familiar with Gaussian Elimination with pivoting will realize that statement S_1 corresponds to the computation of the pivot while statement S_2 corresponds to row elimination with column indexing.

Performing the partition and communication analysis in a manner similar to the previous section yields the following sets for statement S_2 :

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

We will compute the lower bound lb in a short while. Note that the $index\_set_A(t_p)$ is computed for the reference A(i,k). Note that the $index\_set_A$ and $image\_set_A$ together indicate that each processor needs to receive all the columns not owned by the processor. 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.

The block_cyclic distribution allows another optimization. Since each processor owns a block of columns, it can compute the pivot for one whole (or part of) block of columns it owns and then *broadcast* the columns while maintaining the original computation order. That is, instead of sending separate message for each column, the processors can send one message for each sub-block of a block of columns. Since current distributed memory machines have high communication latency, this results in code which runs faster than the code with either the block or cyclic distributions. The issue of selecting the block size that gives the best performance for a particular target machine characteristics is a subject of current investigations.

The Figure 7 shows the code with message vectorization. The code also suggests the need for handling more complex message shapes than that handled by the current compiler: in the example the processors need to send and receive the messages with trapezoidal shape. We performed experiments on the Intel iPSC/860 which revealed that it is profitable to send the precise messages than the conservative 'rectangular' messages. Therefore, the Fortran D compiler needs to include routines in the run-time library to handle more complex messages.

6.3 Send and Receive Sets

To compute the send (out_index_set) and receive (in_index_set) sets, we need to find out, in the most general case, the intersection of two block-cyclically distributed arrays. But, as the example in Figure 8 demonstrates, block-cyclic sets are not closed under intersection.

Stichnoth [16] treats 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.

Generally, we need to intersect two block-cyclic sets only once : to compute the in_index_set and the

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

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

Figure 7: Hand compiled code : Block-cyclic distribution

out_index_set. A simple method to compute the intersection, in such a case, is to treat each block-cyclic set as an array sorted in ascending order. We can then compute the intersection by a simple linear-time algorithm similar to the merge sort algorithm. For example, to find out which elements processor 1 needs to send to processor 0, we need to determine [1:45:3:6] ∩ [6:45:5:10]. In other words, we need to find

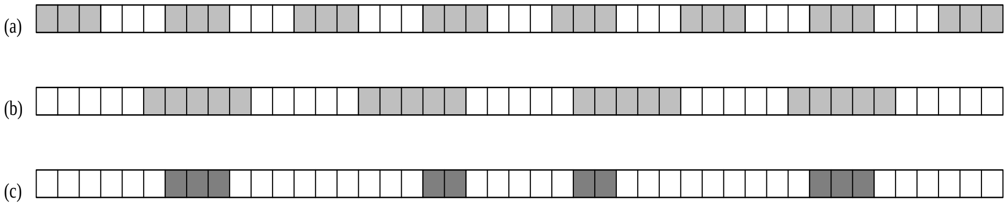
$$\{1,2,3,7,8,9,13,14,15,19,20,21,\ldots,43,44,45\} \cap \{6,7,8,9,10,16,17,18,19,20,\ldots,36,37,38,39,40\}$$

which is simply all the numbers occurring in both the sets. Of course, we need not look at all the array elements to determine the intersection because the pattern repeats after LCM(Block-size(A),Block-size(B))*P (= LCM(3,5)*2 = 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 compares favorably to the time complexity of the method suggested in [16].

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 9 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 8). 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 the case of perfectly aligned arrays, the intersection in such a case would still be a block-cyclic set with block size equal to the smaller of the two original block sizes. The IN and OUT sets can be computed trivially in this case.

```
REAL A(n), B(n)
DISTRIBUTE A(BLOCK_CYCLIC(3)), B(BLOCK_CYCLIC(5))
do i = 1, 45
    A(i) = $\mathcal{F}$(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 8: Intersection of Block-cyclic sets

6.4 Loop Indices and Bounds Generation

Consider the program depicted in the Figure 10. To generate the compiler output shown in the figure, we need to compute *LowerLoopBound*, *UpperLoopBound*, *LocalLoopIndex*, *GlobalLoopIndex* and *Owner*.

Let B be distributed block-cyclically with block size = b. Also, let $n = u - l + 1$. Here we give the formulae for a loop with unit stride. We handle the case of loops with non-unit strides in Section 6.6. We assume processors are numbered 0,...,(P-1). t_p *owns* B(i) if :

$$t_p = \left\lfloor \frac{(i-l) \bmod (b*P)}{b} \right\rfloor$$

$$LocalLoopIndex(i) = \left\lfloor \frac{i-l}{b*P} \right\rfloor b + ((i \bmod (b*P)) \bmod b + 1$$

$$GlobalLoopIndex(i, t_p) = \left\lfloor \frac{i-1}{b} \right\rfloor (P*b) + t_p * b + (i-1) \bmod b + l$$

LowerLoopBound(l_j) :

```
$l_1 = l_j$ - 1
lb$\$ = \lfloor \frac{l_1}{b*P} \rfloor * b + (l_1 \bmod b) + 1$
if $(t_p < \lfloor \frac{l_1 \bmod b*P}{b} \rfloor)$ then
    lb$\$ = \lceil \frac{l_1}{b*P} \rceil * b + 1$
else if $(t_p > \lfloor \frac{l_1 \bmod b*P}{b} \rfloor)$ then
    lb$\$ = \lfloor \frac{l_1}{b*P} \rfloor * b + 1$
endif
```

UpperLoopBound(u_j) :

```
$u_1 = u_j$ - 1
ub$\$ = \lfloor \frac{u_1}{b*P} \rfloor * b + (u_1 \bmod b) + 1$
if $(t_p < \lfloor \frac{u_1 \bmod b*P}{b} \rfloor)$ then
    ub$\$ = \lceil \frac{u_1}{b*P} \rceil * b$
else if $(t_p > \lfloor \frac{u_1\ mod\ b*P}{b} \rfloor)$ then
    ub$\$ = \lfloor \frac{u_1}{b*P} \rfloor * b$
endif
```

```
Input : Arrays A and B as the lhs and the rhs arrays 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" which, using the current location and the block-size,
    increments 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($index\_set_B(d)$);
while (s_index $\le$ LCM(bsize_1, bsize_2) $*$ P and
       d_index $\le$ LCM(bsize_1, bsize_2) $*$ P) do
    if (s_index $>$ proc_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 9: Algorithm for computing the intersection

6.5 Example

To demonstrate the compilation technique and other optimizations, we consider the DGEFA subroutine from Linpack. DGEFA is a key subroutine which performs Gaussian elimination with partial pivoting. Since the subroutine contains triangular loop, block-cyclic distribution is desirable for maintaining good load balance.

Figure 11 shows the original program as well as the hand-compiled Fortran D program. For good load balance, we choose a column block-cyclic distribution which distributes the blocks of width b in a round-robin fashion across the processors. Note that we use a relaxed owner-computes rule to replicate some computation and to avoid expensive and unnecessary communication [10].

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

```
{* Compiler Output *}
REAL B((U-L+1)/n$p)
lb$ = LowerLoopBound($L_j$)
ub$ = UpperLoopBound($U_j$)
do i = $L_i$,$U_i$
   i$ = LocalLoopIndex(i)
   if (Owner(B(i))) B(i$) = $\mathcal{F}_1$(i)
   do j = lb$,ub$
      j$ = GlobalLoopIndex(j, my$proc)
      B(j) = $\mathcal{F}_2$(j$)
   enddo
enddo
```

Figure 10: Loop Indices and Bounds Generation

```
{* 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 11: DGEFA: Gaussian Elimination with Partial Pivoting

We now consider various optimizations which can speed up the code in Figure 11. Interestingly, even without any optmizations, empirical results show that the DGEFA with block-cyclic distribution runs faster than the DGEFA with cyclic distribution. This is because in the case of cyclic distribution, for each column, (P − 1) processors have to wait for the processor which owns that particular column to find the maximum element and broadcast it to others. On the other hand, with block-cyclic distribution, though the number of broadcast messages remains the same, since the processors own a block of columns, once the processor starts executing, it can execute b iterations without waiting for a message. This reduces the time each processor spends waiting for other processors, thus improving the overall execution time.

An obvious optimization is to use message vectorization to reduce the number of messages broadcasted. We can coalesce the messages for a block of columns without changing the order of computation (Note that doing a similar thing for cyclic distribution changes the order). In other words, we strip-mine the outer k loop and then distribute the strip-mined loop. To vectorize the second set of broadcasts, we first need to divide the body of the "if (al .NE. 0) then" into two parts so that one contains the message passing statements (and the preceeding code) and the other contains the loop to reduce remaining submatrices. We enclose both parts within the same conditional and then perform another loop distribution. Of course, we need to determine whether it is legal to perform the loop distribution; in this case it is, as determined from array kill analysis and because the *recv*s define the values they receive and thereby kill the dependence.

The code after message vectorization is shown in Figure 12. Note that the loop can be further optimized by performing the classical optimizations like loop-invariant code motion and loop unswitching.

6.6 Loops with non-unit stride

In the case of block-cyclic distribution, we can not write closed-form expressions for the local iteration set, send & receive sets, etc. In the presence of non-unit stride loops or/and non-unit array subscripts, the block-cyclic distribution gives rise to non-constant (local) memory stride pattern.

As an example, consider the array distribution shown in Figure 13. Here we assume that block size(≡ b) = 4, P = 4 and loop stride(≡ s) = 5. The layout of the array in the processor memories can be visualized as courses of blocks. The *offset* of an array element A(i) is its offset within the course. Since the elements accessed by processor 0 are 1, 36, 51, 66, 81, …, the *stride* for processor 0 is 11, 3, 3, 3, 11, and so on. In other words, after accessing element 1, the next element accessed by processor 0 is 36, which is 11 away from 1. Similarly, 51 is the third element after 36. We wish to compute these distances (which would be kept in the array $\Delta\mathcal{M}$ indexed by the offset). As mentioned in [2], 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 regular section that resides on a processor. This is equivalent to finding the smallest nonnegative integer j such that

$$((L + sj - L_A)\, mod\, (P * b))\, div\, b \;=\; t_p$$

where L = loop lower bound, L_A = array lower bound, and the rest are as defined before.
The above equation reduces to

$$b * t_p \;\leq\; (L + sj - L_A)\, mod\, P * b \;\leq\; b * (t_p + 1) - 1$$

which is equivalent to finding an integer q such that

$$b * t_p - L + L_A \;\leq\; sj \;-\; q(P * b) \;\leq\; b * (t_p + 1) - L + L_A - 1$$

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

$$\{sj \;-\; q(P * b) = \lambda \mid b * t_p - L + L_A \;\leq\; \lambda \;\leq\; b * (t_p + 1) - L + L_A - 1\}$$

```
{* Hand Compiled Output for 4 Processors *}
SUBROUTINE DGEFA(n,a,ipvt)
  INTEGER n,ipvt(n),j,k,l,l$buf(b),m,count$,offset$,owner$
  DOUBLE PRECISION a(n,n/4),al,al$buf(b),t,dp$buf1(b*n)
  do kk = 1, n-1, b
    owner$ = MOD((kk-1), (b*n$p))/b
    kk$ = ((kk - 1)/(b*n$p))b + MOD(MOD((kk-1), b*n$p), b) + 1
    k$ = kk$
    count$ = 1
    do k = kk, kk+b-1
      {* Find max element in a(k:n,k$) *}
      if (my$p .EQ. owner$) then
        l = k
        al = dabs(a(k, k$))
        do i = k + 1, n
          if (dabs(a(i, k$)) .GT. al) then
            al = dabs(a(i, k$))
            l = i
          endif
        enddo
        l$buf(count$) = l
        al$buf(count$) = al
        count$ = count$ + 1
      endif
      k$ = k$ + 1
    enddo
    if (my$p .EQ. owner$) then
      broadcast l$buf, al$buf
    else
      recv l$buf, al$buf
    endif
    k$ = kk$
    count$ = 0
    do k = kk, kk+b-1
      l = l$buf(count$+1)
      al = al$buf(count$+1)
      ipvt(k) = l
      if (al .NE. 0) then
        if (my$p .EQ. owner$) 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
          {* buffer a(k+1:n, k$) into dp$buf1 *}
          offset$ = count$*(n-kk)-(count$*(count$-1))/2+1
          do i = k+1, n
            dp$buf1(offset$) = a(i, k$)
            offset$ = offset$ + 1
          enddo
        endif
      endif
      k$ = k$ + 1
      count$ = count$ + 1
    enddo
    if (my$p .EQ. owner$) then
      broadcast dp$buf1(1:b*(n-kk)-b*(b-1)/2)
    else
      recv dp$buf1(1:b*(n-kk)-b*(b-1)/2)
    endif
    {* Reduce remaining submatrix *}
    k$ = kk$
    count$ = 0
    do k = kk, kk+b-1
      l = l$buf(count$+1)
      al = al$buf(count$+1)
      if (al .NE. 0.0d0) then
        lb$1 = LowerLoopBound(k+1)
        ub$1 = UpperLoopBound(n)
        do j = lb$1, ub$1
          t = a(l,j)
          if (l .NE. k) then
            a(l,j) = a(k,j)
            a(k,j) = t
          endif
          offset$ = count$*(NMAX-kk)-(count$*(count$-1))/2+1
          do i = k+1, n
            a(i, j) = a(i, j) + t * dp$buf1(offset$)
            offset$ = offset$ + 1
          enddo
        enddo
      endif
    enddo
  enddo
  ipvt(n) = n
end
```

Figure 12: DGEFA: Gaussian Elimination with Partial Pivoting

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 13: Block-cyclic distribution

```
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 14: Livermore Kernel 2 : ICCG Excerpt

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 extended Euclid algorithm.

The extended Euclid algorithm would give us not only the first such memory location but all the locations (array elements). We could then sort the array to list all the array elements accessed by the processor and hence find out the memory access gaps. Using this idea, Chatterjee, et al give algorithms for computing the memory access gap sequence for loops with arbitrary array alignments and step size [2]. The running time of the algorithm using this approach is $\mathcal{O}(\log min(s, P * b) + b \log b)$ which reduces to $\mathcal{O}(min(b \log b + \log s, b \log b + \log P))$.

However, in practice, most of the step sizes are small as compared to the block size used for the block-cyclic distribution. As an example, consider the stripped down version of the loop (Figure 14) which appears in the kernel 2 (Incomplete Cholesky-Conjugate Gradient) of Livermore suite. In the following section, we present a linear time algorithm for the two more commonly occurring cases. Figures 15 and 16 depict the two cases.

6.6.1 Algorithm to calculate the memory access sequence

The following algorithm computes 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 affine alignment, then we would need two applications of the following algorithm to get the memory access sequence.

Input : Offset of a valid iteration for processor 0 (offset$_0$), Block size (b), step (s), processor number (t_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 : The following is an algorithm to compute the steps for loops with block-cyclic distribution.

Case I : s < b

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	2	$\boxed{3}$	4	5	$\boxed{6}$	7	8	$\boxed{9}$	50	1	$\boxed{2}$	3	4	$\boxed{5}$	6	7	$\boxed{8}$	9	60
$\boxed{61}$	2	3	$\boxed{4}$	5	6	$\boxed{7}$	8	9	$\boxed{70}$	1	2	$\boxed{3}$	4	5	$\boxed{6}$	7	8	$\boxed{9}$	80

Figure 15: Example : Case I (step = 3)

```
count = 1;
$\Delta\mathcal{M}$[i] = NOT_DEFINED, i=0,...,b-1;
offset = s - ($t_p$*b - (offset$_0$ + $\lfloor\frac{t_p*b-(\text{offset}_0+1)}{s}\rfloor$*s));
while (count $\leq$ b) 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++)
        $\Delta\mathcal{M}$[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;
    count++;
endwhile
```

Case II : s *mod* b*P < b

```
count = 1;
$\Delta\mathcal{M}$[i] = NOT_DEFINED, i=0,...,b-1;
GlobalStep = s $mod$ b * P;
RowsSkipped = s $div$ b * P;
offset = GlobalStep - ($t_p$*b - (offset$_0$ + $\lfloor\frac{t_p*b-(\text{offset}_0+1)}{\text{GlobalStep}}\rfloor$*GlobalStep));
while (count $\leq$ b) do
    if ($\Delta\mathcal{M}$[offset] $\neq$ NOT_DEFINED) break;
    numLocalHops = $\lfloor\frac{b-(\text{offset}+1)}{\text{GlobalStep}}\rfloor$;
    for (i=1; i $\leq$ numLocalHops; i++)
        $\Delta\mathcal{M}$[offset] = b*RowsSkipped + GlobalStep;
        offset = offset + GlobalStep;
    endfor
    lastLoc = offset + 1;
    ElementsLeft = b * P - lastLoc;
    TotalRowsSkipped = $\left(\left\lfloor\frac{\text{ElementsLeft}}{\text{GlobalStep}}\right\rfloor + 1\right)$ * RowsSkipped;
    GlobalElemsLeft = ElementsLeft $mod$ GlobalStep;
    nextOffset = GlobalStep - GlobalElemsLeft - 1;
    $\Delta\mathcal{M}$[offset] = b * TotalRowsSkipped + (b-(offset+1)) + (nextOffset+1);
    offset = nextOffset;
    count++;
endwhile
```

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	2	3	4	5	6	$\boxed{7}$	8	9	50	1	2	3	4	5	6	7	8	9	60
61	2	3	4	5	6	7	8	9	$\boxed{70}$	1	2	3	4	5	6	7	8	9	80
81	2	3	4	5	6	7	8	9	90	1	2	$\boxed{3}$	4	5	6	7	8	9	100
101	2	3	4	5	6	7	8	9	10	1	2	3	4	5	$\boxed{6}$	7	8	9	20
21	2	3	4	5	6	7	8	9	30	1	2	3	4	5	6	7	8	$\boxed{9}$	40
41	2	3	4	5	6	7	8	9	50	1	2	3	4	5	6	7	8	9	60
61	$\boxed{2}$	3	4	5	6	7	8	9	70	1	2	3	4	5	6	7	8	9	80
81	2	3	4	$\boxed{5}$	6	7	8	9	90	1	2	3	4	5	6	7	8	9	200
201	2	3	4	5	6	7	$\boxed{8}$	9	10	1	2	3	4	5	6	7	8	9	20
21	2	3	4	5	6	7	8	9	30	$\boxed{1}$	2	3	4	5	6	7	8	9	40
41	2	3	4	5	6	7	8	9	50	1	2	3	$\boxed{4}$	5	6	7	8	9	60
61	2	3	4	5	6	7	8	9	70	1	2	3	4	5	6	$\boxed{7}$	8	9	80
81	2	3	4	5	6	7	8	9	90	1	2	3	4	5	6	7	8	9	$\boxed{00}$
301	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20
21	2	$\boxed{3}$	4	5	6	7	8	9	30	1	2	3	4	5	6	7	8	9	40
41	2	3	4	5	$\boxed{6}$	7	8	9	50	1	2	3	4	5	6	7	8	9	60
61	2	3	4	5	6	7	8	$\boxed{9}$	70	1	2	3	4	5	6	7	8	9	80
81	2	3	4	5	6	7	8	9	90	1	$\boxed{2}$	3	4	5	6	7	8	9	400
401	2	3	4	5	6	7	8	9	10	1	2	3	4	$\boxed{5}$	6	7	8	9	20
21	2	3	4	5	6	7	8	9	30	1	2	3	4	5	6	7	$\boxed{8}$	9	40
41	2	3	4	5	6	7	8	9	50	1	2	3	4	5	6	7	8	9	60
$\boxed{61}$	2	3	4	5	6	7	8	9	70	1	2	3	4	5	6	7	8	9	80
81	2	3	$\boxed{4}$	5	6	7	8	9	90	1	2	3	4	5	6	7	8	9	500

Figure 16: Example : Case II (step = 23)

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

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

For case II, we need slightly more work :
Let global step g = s mod b∗P, the processor which owns L be p = ((L-L_A) mod b∗P) div b, the offset of L within its course be o = ((L-L_A) mod b∗P) mod b. Therefore, the number of elements left in the row, e = b − (o+1) + (P-p)∗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 16), $l = e\, mod\, g$. Now, $\text{offset}_0 = g - l - 1$

Explanation : 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 compute the offsets (and the memory access gaps) without requiring to actually solve the Diophantine equations.

For case I, the algorithm works as follows :
Given an offset for the processor p, we first compute the set of iterations executed within the same course. Using the value of the last iteration in the course, 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, s − e mod s − 1 is the offset (within the next course) of the next iteration executed by p.

For case II, the algorithm works almost identically 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.

An interesting fact which could be used to further speed up the algorithm is that the length of the memory access gap sequence cycle divides the block size. Others tricks like treating mults and divs 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 algorithm.

Complexity Each element of the array $\Delta\mathcal{M}$ is filled at most once by the algorithm and as soon as an already filled array element is enountered, the algorithm stops. Therefore, the 'while' loop (together with the inner 'for' loop) iterates atmost b times and hence it is an $\mathcal{O}(b)$ algorithm. Therefore, not only is our approach conceptually more intuitive, the algorithms given above are an $\mathcal{O}(b)$.

6.7 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. For example, 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 before. 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 processor which need to send the data and the location of the array element within the owner processor. Note that, in case of shifts, a processor communicates with at most two processors.

Suppose that element A(i) is located on processor t_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 $\Delta\mathcal{P}$ = $\lceil$(r $-$ o)/b$\rceil$, where b is the block-size. Then the owner processor of A(i-d) is (t_p $-$ $\Delta\mathcal{P}$+P) mod P. Since 0 $\leq$ o $\leq$ 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 $\mathcal{M}' = \mathcal{M} + \Delta\mathcal{L}(o)$.

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

Now, since the memory access sequence algorithm can compute offsets also, we can determine the iterations which need communication as well as the elements which 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 [15] for irregular loops.

7 More General SPMD Code Replication Checking

As mentioned elsewhere in the literature [5], a sequential code segment can be converted to SPMD code if there are no storage-related dependences in the segment. While storage-related dependences are unsafe, flow dependences do not cause safety problems. Fortunately, storage-related dependences can always be removed by using additional storage [4]; one of the ways being *privatization*.

The Fortran D compiler recognizes that statements which cause either input dependences in a loop or output dependences impede parallelism. E.g. if there is a Fortran "read" statement in the loop or if there is a scalar variable which can not be privatized, the loop can not be converted into SPMD code. Currently, the compiler creates a guard around all "write" statements so that only processor 0 performs all the output.

The loops with function calls may be handled similarly. The compiler will make use of the interprocedural array section analysis (not yet fully implemented) and other procedure summary information to determine if the loops may be parallelized. If the functions have side-effects that may cause input or output dependence then the loop will not be converted to an SPMD loop.

8 Code Generation

Code generation is divided into two parts, partitioning and message generation. The partitioning phase applies loop bounds reduction and inserts guards, modifying the AST according to the information in the iteration sets. The message generation phase uses the information in the RSDs for to generate messages. We have described in previous sections, the extensions required for iteration and index sets. These sets will be used to generate the necessary communication.

8.1 Run Time Support

One of the key limitations of the code generation phase in the current Fortran D compiler is its lack of a run time library that may be used in the presence of symbolic index and iteration sets.

8.1.1 Loop Bounds Partition Library

Routines that return local upper and lower bounds based on the distribution of the array and the subscript or loop bounds expressions must be provided. The equations described in will be used to build the run time library. Below are some of the routines that must be provided.

```
block_upper()        block_upper_stride()
cyclic_upper()       cyclic_upper_stride()
block_cyclic_upper() block_cyclic_upper_stride()
block_lower()        block_lower_stride()
cyclic_lower()       cyclic_lower_stride()
block_cyclic_lower() block_cyclic_lower_stride()
```

8.1.2 Communication Library

The actual communication that may take place may be known only at run time due to symbolic index sets. The compiler must be able to generate code which determines at run time the communication that occurs. This is one of the major weaknesses of the current compiler. We begin by describing in the context of a simple loop, the transformations the compiler must perform.

```
   real a(100,100), b(100,100)
c  decomposition d(100,100)
c  align a,b with d
c  distribute d(cyclic, :)

   do j = 1, n
    do i = 1, n
     a(i,j) = b(i+k,j)
    enddo
   enddo
```

In the loop nest above, since the upper loop bounds are symbolic values and **k** occuring in the subscript expression is also a symbolic whose value is known at run time. The iteration and index sets are classified as symbolic and the communication classification algorithm sets the communication type as **shift**. The code generation phase uses this information to insert calls to routines that generate lower and upper bounds for cyclic distribution and messages for shift communication with one dimensional cyclic distribution. The psuedo code generated is shown below.

```
c  u = lower_cyclic(....)
```

```
c  l = upper_cyclic(....)
c  kloc = local offset

c  is there any communication required?
   if (mod(k, P) .ne. 0)
      1d_cyclic_shift_dim2(b, 1, n, cyclic,1,n, local, k, 0, temp)
       do j = 1, n
        do i = 1,u
         a(i,j) = temp(i)
        enddo
      enddo
    else
      do j = 1, n
        do i = 1,u
         a(i,j) = b(i+kloc,j)
        enddo
      enddo
   endif
```

The function 1d_cyclic_shift_dim2 contains code that determines the **`in_index_set, out_index_set, receive_p_set, and send_p_set`** for every processor. It uses the algorithm depicted in Figure 17. We need a run time library that contains functions that generate communication in the presence of symbolic values and distributions of the type block, cyclic, block_cyclic. Below are some of the functions to be included in the library.

```
1d_block_shift
1d_block_broadcast
1d_block_send_recv
1d_block_shift_dim2
1d_block_broadcast_dim2
1d_block_send_recv_dim2

1d_block_cyclic_shift
1d_block_cyclic_broadcast
1d_block_cyclic_send_recv
```

```
1d_cyclic_shift
1d_cyclic_broadcast
1d_cyclic_send_recv
1d_cyclic_shift_dim
1d_cyclic_broadcast_dim2
1d_cyclic_send_recv_dim2

1d_block_cyclic_shift_dim2
1d_block_cyclic_broadcast_dim2
1d_block_cyclic_send_recv_dim2
```

The list above is by no means complete. It simply lists the type of functions we will have in our run time library

8.1.3 Inspector/Executor and Run Time Resolution

When complex index sets are generated, the compiler will not be able to optimize the code and will have to resort to generating inspector/executors described in [17] in the case where true loop carried cross processor dependences do not exist and run time resolution otherwise. The classification algorithm will classify the communication and the run time system will perform the appropriate transformations to generate code.

9 Experimental Results

In this section we present the performance results of compiling Gaussian elimination with block-cyclic distribution.

9.1 Block-cyclic Distribution

Experiments reveal that we achieve the same speedup irrespective of whether we send the l$buf and al$buf as two separate messages or if we copy them both onto a buffer and then send the buffer as a

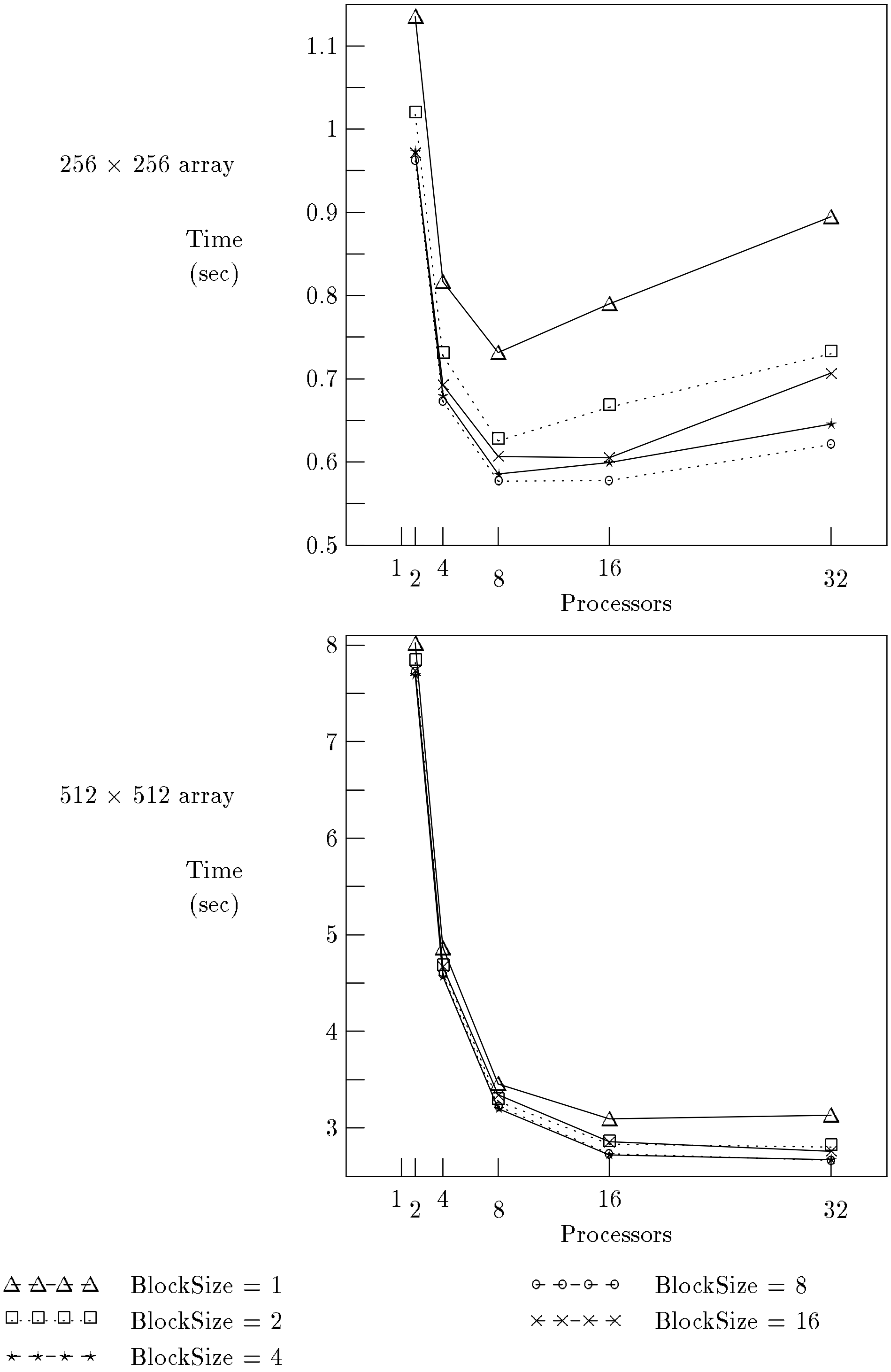
Program	Problem Size	Proc	Block Size (time in secs)							
			1	2	4	8	16	32	64	128
Gauss	256 × 256	1	*sequential time = 2.151*							
		2	1.135	1.017	0.973	0.962	0.971	0.992	1.081	1.248
		4	0.816	0.723	0.679	0.673	0.693	0.725	0.826	1.343
		8	0.731	0.625	0.586	0.577	0.606	0.640	0.924	1.441
		16	0.790	0.665	0.599	0.578	0.605	0.738	1.020	1.537
		32	0.895	0.730	0.646	0.621	0.707	0.836	1.117	1.633
	512 × 512	1	*sequential time = 17.53*							
		2	8.021	7.818	7.693	7.724	7.730	7.848	8.194	8.882
		4	4.869	4.659	4.566	4.591	4.669	4.826	5.197	6.049
		8	3.457	3.273	3.199	3.224	3.343	3.539	3.900	6.440
		16	3.095	2.834	2.724	2.732	2.855	2.993	4.280	6.819
		32	3.129	2.800	2.674	2.665	2.760	3.376	4.660	7.196
	$1K \times 1K$	1	*estimated sequential time = 140*							
		2	66.84	66.61	66.37	67.29	67.16	67.92	69.42	72.59
		4	36.33	36.10	35.97	36.51	36.81	37.62	39.15	43.52
		8	21.89	21.61	21.53	21.93	22.44	23.49	25.44	28.77
		16	15.67	15.24	15.12	15.41	16.00	17.00	18.46	
		32	13.33	12.80	12.64	12.86	13.42	14.06	19.97	

Table 1: Intel iPSC/860 Execution Times for Gaussian Elimination with BlockCyclic distr

single message.

Surprisingly, we also noted that it is faster to broadcast the precise trapezoidal region instead of the more conservative rectangular regions. One would guess that since the startup cost remains the same, sending a few extra array elements per message but simplifying the array index calculation will give better results; but our experiments point otherwise.

Table 1 contains the timings for the Gaussian Elimination with pivoting for the various block sizes, processors and the problem sizes. An important observation is that the block size which yields the best performance is independent of the number of processors and depends only on the original problem size. In other words, the best block size depends on the ratio of the communication and computation inherent in the algorithm used in the program and not on the number of processors. The following graphs show these results more clearly. Also, as expected, the block-cyclic distribution with appropriate block sizes outperforms both the cyclic and the block distribution.



10 Summary

A usable yet efficient machine-independent parallel programming model is needed to make large-scale parallel machines useful for scientific programmers. We believe that Fortran D, one of the first data-placement languages, can provide such a portable data-parallel programming model. The key to achieving good performance is applying advanced compiler analysis and optimization to automatically extract parallelism and manage communication. This paper describes techniques that will significantly extend the class of problems the Fortran D compiler can efficiently compile.

```
Given:
  $a(l_1 : u_1, ....., l_n : u_n)$
  align $a$, $b$ with $d$
  distribute $d(j_1, ..., j_n)$ s.t.  dimension $j_k$ is distributed
  CYCLIC and the remaining dimensions are not distributed
Loop nest:
  Do $i_1 \;=\; lb_1, ub_1$
    .......
    Do $i_n \;=\; lb_n, ub_n$
      ......
S$_1$         $b(..., i_k, ..) \;=\; a(f(i_1), ...i_k + c_k, ..., f(i_n))$
    Enddo
    ......
  Enddo
Send Receive Sets for Statement S$_1$:
  if $(t_p \; < \; (lb_k \; - \; l_k) \bmod$ P) then
    $lb_{k1} = \lceil \frac{lb_k \, - \, l_k}{P} \rceil * P \; + \; 1 + t_p$
  else
    $lb_{k1} = \lfloor \frac{lb_k \, - \, l_k}{P} \rfloor * P \; + \; 1 \; t_p$
  endif
  if $(t_p \; < \; (ub_k \; - \; l_k) \bmod$ P) then
    $ub_{k1} = \lfloor \frac{ub_k \, - \, l_k}{P} \rfloor * P \; + \; 1 + \; t_p$
  else
    $ub_{k1} = \lceil \frac{ub_k \, - \, l_k}{P} \rceil * P \; - \; P \; + \; 1 + \; t_p$
endif
  $iter\_set(t_p) \;=\; (ub_1 : ub_1, \ldots, lb_{k1} : ub_{k1} : P, \ldots, lb_n : ub_n)$
  $index\_set(t_p) \;=\; (lb_1 : ub_1, \ldots, lb_{k1} + c_k : ub_{k1} + c_k \; : P, \ldots, lb_n \; : ub_n)$

  if $(c_k \; - \; l_k \bmod P \neq 0)$
    $in\_index\_set(t_p) \;=\; (lb_1 : ub_1, \ldots, lb_{k1} + c_k \; : ub_{k1} + c_k \; : P, \ldots, lb_n \; : ub_n)$
  else
    $in\_index\_set(t_p) \;=\; \emptyset$
  endif

  $out\_index\_set(t_p) \;=\; in\_index\_set(t_p - c_k \bmod P + P) \bmod P$

  $receive\_p\_set(t_p) \;=\; (t_p + c_k \bmod P + P) \bmod P$
  $send\_p\_set(t_p) \;=\; (t_p - c_k \bmod P + P) \bmod P$
```

Figure 17: Shift Communication for Cyclic Distribution

In this paper we described the analysis required to compile Fortran D programs in the presence symbolic loop bounds and array sizes, non-unit loop strides and variable number of processors. We compute symbolic iteration and index sets and use them to generate efficient code. The iteration and index sets are used by the communication classification algorithm to select for each non local right hand side reference, the type of communication that occurs. We propose a code generation strategy that uses the communication type for each non local right hand side reference to generate efficient code. We provide the analysis necessary to generate communication in the presence of multidimensional block and cyclic distributions and optimize the messages when possible. Partitioning and communication analysis for block_cyclic distribution is dealt with in detail. We provide empirical results on our compilation strategy for block_cyclic distribution that provide an insight into the usefulness of our analyses.

11 Acknowledgements

The authors wish to thank Uli Kremer and Chau-Wen Tseng for their assistance on this paper. This research was supported by the IBM corporation, the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008 and by DARPA under contract #DABT63-92-C-0038. Use of the Intel iPSC/860 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement Nos. CCR-8809615 and CDA-8619893 with support from the Keck Foundation.

References

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

[2] 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.

[3] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[4] R. Cytron and J. Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, IL, August 1987.

[5] R. Cytron, J. Lipkis and E. Schonberg. A Compiler-Assisted approach to SPMD execution. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

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

[7] 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.

[8] 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.

[9] 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.

[10] S. Hiranandani, K. Kennedy, and C. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[11] K. Kennedy, K. S. M^{c}Kinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[12] C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Dept. of Computer Science, Purdue University, West Lafayette, IN, August 1990.

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

[14] 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.

[15] 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.

[16] 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.

[17] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

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