Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines

Seema Hiranandani
Ken Kennedy
Chau-Wen Tseng

CRPC-TR91162
April, 1991

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

Revised August, 1991.

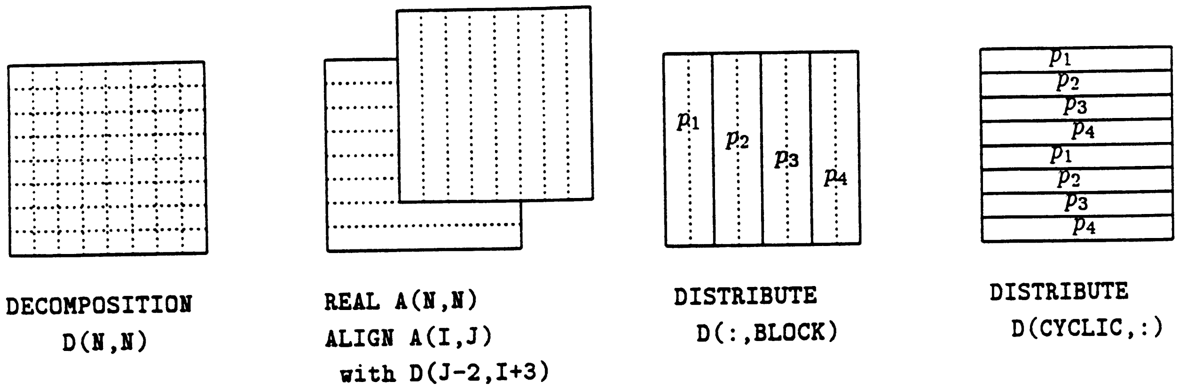


Figure 1: Fortran D Data Decomposition Specifications

dence [22] that unifies and extends previous techniques.

The rest of this paper presents the data decomposition specifications in Fortran D, basic compiler analysis and code generation algorithms, and compiler optimizations to reduce communication costs and load imbalance. We conclude with a description of the current status of the compiler and comparison with related work.

2 Fortran D Language

The data decomposition problem can be approached by considering the two levels of parallelism in data-parallel applications. First, there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. We call this the *problem mapping* induced by the structure of the underlying computation. It represents the minimal requirements for reducing data movement for the program given an unlimited number of processors, and is largely independent of any machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machine. We call this the *machine mapping* caused by translating the problem onto the finite resources of the machine. It is affected by the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D provides data decomposition specifications for these two levels of parallelism using the `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` statements. A decomposition is an abstract problem or index domain; it does not require any storage. Each element of a decomposition represents a unit of computation. The `DECOMPOSITION` statement declares the name, dimensionality, and size of a decomposition for later use.

The `ALIGN` statement is used to map arrays onto decompositions. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or across dimensions. The alignment of arrays to decompositions is specified by placeholders in the subscript expressions of both the array and decomposition. In the example below,

```
REAL A(N,N)
DECOMPOSITION D(N,N)
ALIGN A(I,J) with D(J-2,I+3)
```

D is declared to be a two dimensional decomposition of size $N \times N$. Array A is then aligned with respect to D with the dimensions permuted and offsets within each dimension.

After arrays have been aligned with a decomposition, the `DISTRIBUTE` statement maps the decomposition to the finite resources of the physical machine. Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Predefined attributes are `BLOCK`, `CYCLIC`, and `BLOCK_CYCLIC`. The symbol ":" marks dimensions that are not distributed. Choosing the distribution for a decomposition maps all arrays aligned with the decomposition to the machine. Scalars and unaligned arrays are replicated, *i.e.*, owned by all processors.

In the following example, distributing decomposition D by `(:,BLOCK)` results in a column partition of arrays aligned with D. Distributing A by `(CYCLIC,:)` partitions the rows of D in a round-robin fashion among processors. These sample data alignment and distributions are shown in Figure 1.

```
DECOMPOSITION D(N,N)
DISTRIBUTE D(:,BLOCK)
DISTRIBUTE D(CYCLIC,:)
```

Note that data distribution does not subsume alignment. For instance, the `DISTRIBUTE` statement alone cannot specify that one 2-D array be mapped with the transpose of another.

Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines*

Seema Hiranandani Ken Kennedy Chau-Wen Tseng

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

Massively parallel MIMD distributed-memory machines can provide enormous computation power. However, the difficulty of developing parallel programs for these machines has limited their accessibility. This paper presents compiler algorithms to automatically derive efficient message-passing programs based on data decompositions. Optimizations are presented to minimize load imbalance and communication costs for both loosely synchronous and pipelined loops. These techniques are employed in the compiler being developed at Rice University for Fortran D, a version of Fortran enhanced with data decomposition specifications.

1 Introduction

It is widely recognized that parallel computing represents the only plausible way to continue to increase the computational power available to computational scientists and engineers. However, parallel computers are not likely to be widely successful until they are easy to program. A major component in the success of vector supercomputers is the ability of scientists to write Fortran programs in a "vectorizable" style and expect vectorizing compilers to automatically produce efficient code [9, 32]. The resulting programs are easily maintained, debugged, and portable across different vector machines.

Compare this with the current situation for programming parallel machines. Scientists wishing to use such machines must rewrite their programs in an extension of Fortran that explicitly reflects the architecture of the underlying machine, such as a message-passing dialect for MIMD distributed-memory machines, array syntax for SIMD machines, or an explicitly parallel dialect with synchronization for MIMD shared-memory machines.

The goal of the Fortran D project is to establish the feasibility of a machine-independent parallel programming model. It must be easy to use yet perform with acceptable efficiency on different parallel architectures, at least for a significant portion of scientific codes. Advanced language, compiler, and environment support will be vital.

Of the different parallel architectures, MIMD distributed-memory machines provide the most difficult programming model. Users must write message-passing programs that deal with separate address spaces, synchronizing processors using messages, collective communication, and processor interconnection topologies. The process is time-consuming, tedious, and error-prone. Three to ten-fold blowups in source code size are not only common but expected. Even worse, the resulting parallel programs are extremely machine-specific. Scientists are thus discouraged from utilizing these machines because they risk losing their investment whenever the program changes or a new architecture arrives.

We propose to solve this problem by developing the compiler technology needed to automate translation of Fortran programs to MIMD distributed-memory machines. We find that selecting a data decomposition is one of the most important intellectual steps in developing data-parallel scientific codes. Though many techniques have been developed for automatic data decomposition, we feel that the compiler will not be able to choose an efficient data decomposition for all programs. To be successful, the compiler needs additional information not present in vanilla Fortran.

Current parallel programming languages provide little support for data decomposition [25]. We have therefore developed an enhanced version of Fortran that introduces data decomposition specifications. We call the extended language Fortran D, where "D" suggests data, decomposition, or distribution. We believe that if a Fortran D program is written in a data-parallel programming style with reasonable data decompositions, it can be implemented efficiently on a variety of parallel architectures.

In this paper, we describe the design of a prototype Fortran D compiler for the iPSC/860, a MIMD distributed-memory machine. The goal of the compiler is to automate the task of deriving node programs based on the data decomposition. For these machines, it is particularly important to minimize both communication costs and load imbalance. We present a code generation strategy based on the concept of *data depen-*

*This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center.

Dependence testing Dependence testing determines the existence of data dependences between array references by examining their subscript expressions. Dependences found are characterized by their dependence level, as well as by distance and direction vectors. This information is used to guide subsequent compiler analysis and optimization.

3.1.2 Data Decomposition Analysis

The Fortran D compiler requires a new type of program analysis to generate the proper program—it must determine the data decomposition for each reference to a distributed array.

Reaching decompositions Because data access patterns may change between program phases, Fortran D provides dynamic data decomposition by permitting executable `ALIGN` and `DISTRIBUTE` statements to be inserted at any point in a program. This complicates the job of the Fortran D compiler, since it must know the decomposition of each array.

We define *reaching decompositions* to be the set of decomposition specifications that may reach an array reference aligned with the decomposition; it may be calculated in a manner similar to *reaching definitions*. The Fortran D compiler will apply both intra- and interprocedural analysis to calculate reaching decompositions for each reference to a distributed array. If multiple decompositions reach a procedure, runtime or node splitting techniques such as *cloning* may be required to generate the proper code for the program.

3.1.3 Partitioning Analysis

After data decomposition analysis is performed, the program partitioning analysis phase of the Fortran D compiler divides the overall data and computation among processors. This is accomplished by first partitioning all arrays onto processors, then using the *owner computes* rule to derive the functional decomposition of the program. We begin with some useful definitions.

Iteration & index sets, RSDs An *iteration set* is simply a set of loop iterations—it describes a section of the work space. An *index set* is a set of locations in an array—it describes a section of the data space. In many cases, the Fortran D compiler can construct iteration or index sets using *regular section descriptors* (RSDs), a compact representation of rectangular sections (with some constant step) and their higher dimension analogs [14]. The union and intersection of RSDs can be calculated inexpensively, making them highly useful for the Fortran D compiler.

In this paper we will write RSDs as $[l_i{:}u_i{:}s_i, \ldots]$, where l_i, u_i, and s_i indicate the lower bound, upper bound, and step of the ith dimension of the RSD, respectively. A default unit step is assumed if not explicitly stated. In loop nests or multidimensional arrays, the leftmost dimension of the RSD corresponds to the outermost loop or the leftmost array dimension. The other dimensions are listed in order.

Global vs. local indices Because the Fortran D compiler creates SPMD node programs, all processors must possess the same array declarations. This forces all processors to adapt *local indices*. For instance, consider the following program and the node program produced when array A is block-distributed across four processors.

```
{* Original program *}   {* SPMD node program *}
REAL A(100)              REAL A(25)
do i = 1, 100            do i = 1, 25
  A(i) = 0.0               A(i) = 0.0
enddo                    enddo
```

The local indices for A on each processor are all [1:25], even though the equivalent global indices for A are [1:25], [26:50], [51:75], and [76:100] on processors 1 through 4, respectively. A similar conversion of loop indices may also occur, with the global loop indices [1:100] translated to the local loop indices [1:25].

Local index sets As the first step in partitioning analysis, the Fortran D compiler uses the Fortran D statements associated with each reaching decomposition to calculate the *local index set* of each array—the local array section owned by every processor. This creates the data partition used in the program.

We illustrate the analysis required to partition the Jacobi code in Figure 3. For this and all future examples we will be compiling for a four processor machine. In the example, both arrays A and B are aligned identically with decomposition D, so they have the same distribution as D. Because the first dimension of D is local and the second dimension is block-distributed, the local index set for both A and B on each processor (in local indices) is [1:100,1:25].

Local iteration sets Once the local index set for each array has been calculated, the Fortran D compiler uses it to derive the functional decomposition of the program. We define the *local iteration set* of a reference R on a processor to be the set of loop iterations that cause R to access data owned by the processor. It can be calculated by applying the inverse of the array subscript functions to the local index set of R, then intersecting the result with the iteration set of the enclosing loops.

The calculation of local index and iteration sets is vital to the partitioning analysis of the Fortran D compiler. When applying the *owner computes* rule, the set of loop iterations on which a processor must execute an assignment statement is exactly the local iteration set of the left-hand side (*lhs*). The Fortran D compiler can thus partition the computation by assigning iteration sets to each statement based on its *lhs*.

To demonstrate the algorithm, we will calculate the local iteration set for the assignment statement S_1 in the Jacobi example. Remember that the local index set of A is [1:100,1:25]. First we apply to it the inverse of

Because it has been designed to support programming on both SIMD and MIMD parallel architectures, Fortran D is the first language to provide both alignment and distribution specifications. It also supports irregular data distributions and dynamic data decomposition, *i.e.,* changing the alignment or distribution of a decomposition at any point in the program. The complete language is described in detail elsewhere [11].

3 Fortran D Compiler

The two major steps in writing a data-parallel program are selecting a data decomposition and using it to derive node programs with explicit data movement. We leave the task of selecting a data decomposition to the user or automatic tools. The Fortran D compiler automates the second step by generating node programs with explicit communication for a given data decomposition.

The main goal of the compiler is to minimize load imbalance and communication costs. It translates Fortran D programs into *single-program, multiple-data* (SPMD) form with explicit message-passing that execute directly on the nodes of the distributed-memory machine. The compiler partitions the program using the *owner computes* rule, by which each processor computes values of data it owns [8, 27, 34].

The prototype Fortran D compiler is being developed in the context of the ParaScope programming environment [7]. The compiler is subdivided into three major phases—program analysis, program optimization, and code generation. The structure of the compiler is shown in Figure 2.

1. Program analysis
   (a) Dependence analysis
   (b) Data decomposition analysis
   (c) Partitioning analysis
   (d) Communication analysis
2. Program optimization
   (a) Program transformations
   (b) Message vectorization
   (c) Collective communication
   (d) Runtime processing
3. Code generation
   (a) Program partitioning
   (b) Message generation
   (c) Storage management

Figure 2: Fortran D Compiler Structure

3.1 Program Analysis

3.1.1 Dependence Analysis

Dependence analysis is the compile-time analysis of control flow and memory accesses to determine a statement execution order that preserves the meaning of the original program. A *data dependence* between two references R_1 and R_2 indicates that they read or write a common memory location in a way that requires their execution order to be maintained [22]. We call R_1 the *source* and R_2 the *sink* of the dependence if R_1 must be executed before R_2. If R_1 is a write and R_2 is a read, we call the result a *true* (or *flow*) dependence.

Dependences may be either loop-independent or loop-carried. *Loop-independent* dependences occur on the same loop iteration; *loop-carried* dependences occur on different iterations of a particular loop. The *level* of a loop-carried dependence is the depth of the loop carrying the dependence [2]. Loop-independent dependences have infinite depth. The number of loop iterations separating the source and sink of the loop-carried dependence may be characterized by a dependence *distance* or *direction* [31].

Dependence analysis is vital to shared-memory vectorizing and parallelizing compilers. We show that it is also highly useful for guiding compiler optimizations for distributed-memory machines. The Fortran D compiler incorporates the following ParaScope dependence analysis capabilities [20].

Scalar dataflow analysis Control flow, control dependence, and live range information are computed during the scalar dataflow analysis phase. In addition, scalar variables are labeled *private* with respect to a loop if their values are used only within the current loop iteration; this is useful for eliminating unnecessary computation and communication.

Symbolic analysis Constant propagation, auxiliary induction variable elimination, expression folding, and loop invariant expression recognition are performed during the symbolic analysis phase of the Fortran D compiler. The goal of symbolic analysis is to provide a simplified program representation for the Fortran D compiler that improves additional program analysis and optimization. Consider the example below:

```
do ij = 1,len
 f(ij,n) = (f(ij,n)-tot(ij)) / B(n)
enddo
```

If constant propagation is able to produce a constant value for n, the ownership of $B(n)$ could be determined at compile time to generate simple efficient code. If the value of n cannot be discovered at compile time, analysis can still identify n as a loop invariant expression for the ij loop. The Fortran D compiler can then communicate $B(n)$ with an efficient *broadcast* preceding the loop, rather than individual guarded messages during each loop iteration.

Symbolic analysis also recognizes *reductions*, operations that are both commutative and associative. Once identified, reductions may be executed locally in parallel and the results combined efficiently using collective communication routines. Reduction operations are tagged during symbolic analysis for use in later phases of the communication analysis generation process.

are calculated in reverse order for statements in each loop nest.

For instance, consider the loop in Figure 4. Because QA is a scalar, the *owner computes* rule would assign all loop iterations as the iteration set for statement S_1. However, since the only use of QA occurs in the same loop iteration, it is classified as a private variable. The Fortran D compiler can thus assign S_1 the same iteration set as S_2, the only statement containing a true dependence with S_1 as its source.

In other cases dependence analysis will have marked the scalar assignment statement or a group of statements as a *reduction*. Any iteration set that ensures that the computation is partitioned exactly once across processors may be selected. To reduce data movement, the Fortran D compiler can assign an iteration set for reduction statements using the iteration set of a distributed array variable on the right-hand side (*rhs*).

3.1.4 Communication Analysis

Once partitioning analysis determines how data and work are distributed across processors, communication analysis determines which variable references cause nonlocal data accesses.

Computing nonlocal index sets In this phase, all *rhs* references to distributed arrays are examined. For each *rhs*, the Fortran D compiler constructs the index set accessed by each processor. The index set is computed by applying the inverse subscript functions of the *rhs* to the local iteration set assigned to the statement. The local index set is subtracted from the resulting RSD to check whether the reference accesses nonlocal array locations. If no nonlocal accesses occur, the *rhs* reference is local and may be discarded. Otherwise the RSD representing the *nonlocal index set* accessed by the *rhs* is retained.

If boundary conditions exist for the local iteration set of the statement, the Fortran D compiler must compute the index set for each group of processors assigned different iteration sets. In the worst case the index set for each processor must be calculated separately.

We show how index sets are computed for the Jacobi example. We first consider the four *rhs* references to B in statement S_1. The iteration set boundary conditions cause processors to be separated into three groups. The group of interior processors, *Proc*(2:3), have the local iteration set [1:*time*,1:25,2:99]. This derives the following index sets:

$$\begin{aligned}
B(i,j-1) &= [2:99,\ 0:24] \\
B(i-1,j) &= [1:98,\ 1:25] \\
B(i+1,j) &= [3:100,\ 1:25] \\
B(i,j+1) &= [2:99,\ 2:26]
\end{aligned}$$

Since the local index set for B is [1:100,1:25], $B(i-1,j)$ and $B(i+1,j)$ cause only local accesses and may be ignored. However, $B(i,j-1)$ and $B(i,j+1)$ access nonlocal locations [2:99,0] and [2:99,26], respectively. Both references are marked and their nonlocal index sets stored.

Computing the index sets using the local iteration sets for the other two groups, *Proc*(1) and *Proc*(4), does not yield additional nonlocal references. Examination of the index sets for the *rhs* reference to $A(i,j)$ in statement S_2 show that only local accesses occur.

3.2 Program Optimization

The program optimization phase of the Fortran D compiler performs optimizations to improve the performance of the resulting program. It is particularly useful to identify *cross-processor dependences*—dependences whose endpoints are executed by different processors. In this section we concentrate on basic optimizations, deferring discussion of advanced communication optimization and program transformation techniques to Section 3.4.

Communication selection A naive but workable algorithm for introducing communication is to insert guarded send and/or receive operations directly preceding each statement with a nonlocal reference. Unfortunately, this simple approach generates many small messages that may prove inefficient due to communication overhead. Communication optimizations combine these messages in order to achieve efficiency.

First the Fortran D compiler must determine what communication optimization to use for each *rhs* reference accessing nonlocal data. It does this by comparing the subscript expression of each distributed dimension in the *rhs* with the aligned dimension in the *lhs* reference. Consider the following example.

```
DECOMPOSITION D(n,n)
ALIGN A, B with D
DISTRIBUTE D(BLOCK,BLOCK)
do j = 2,n
  do i = 2,n
S1    A(i,j) = A(i,j-1)+B(i-1,j)
S2    A(i,j) = B(c,j)+B(j,i)
S3    A(i,j) = B(f(i),j)
  enddo
enddo
```

The arrays A and B are aligned identically and both dimensions are distributed, so we need to compare the first dimensions with each other, then the second. For the *rhs* references $A(i,j-1)$ and $B(i-1,j)$ in S_1 we apply *message vectorization* because simple expressions of the same loop index variable appear in the aligned dimensions of the *lhs* and *rhs* references.

For the *rhs* reference $B(c,j)$ in S_2 we apply *collective communication* because a constant term in a distributed dimension indicates the need for broadcast communication. For $B(j,i)$ in S_2 we apply collective communication because different loop index variables appear in the aligned dimensions, indicating either transpose or all-to-all communication. For the *rhs*

```
REAL A(100,100), B(100,100)
DECOMPOSITION D(100,100)
ALIGN A, B with D
DISTRIBUTE D(:,BLOCK)
do k = 1,time
  do j = 2,99
   do i = 2,99
S1     A(i,j) = (B(i,j-1)+B(i-1,j)+
                 B(i+1,j)+B(i,j+1))/4
   enddo
  enddo
  do j = 2,99
   do i = 2,99
S2     B(i,j) = A(i,j)
   enddo
  enddo
enddo
```

Figure 3: Jacobi

the subscript functions of the *lhs*, $A(i,j)$. This yields the unbounded local iteration set [:,1:25,1:100]. The first entry is ":" since all iterations of the k loop access local elements of A. The inverse subscript functions cause the j and i loops to be mapped to [1:25] and [1:100], respectively.

Next we intersect the unbounded iteration set with the actual bounds of the enclosing loops, since these are the only iterations that actually exist. The iteration set of the loop nest (in global indices) is [1:*time*,2:99,2:99]. Converting it into local indices for each processor and performing the intersection yields the following local iteration sets for each processor (in local indices):

$$\begin{array}{rcl} Proc(1) & = & [\,1:time,\ 2:25,\ 2:99\,] \\ Proc(2:3) & = & [\,1:time,\ 1:25,\ 2:99\,] \\ Proc(4) & = & [\,1:time,\ 1:24,\ 2:99\,] \end{array}$$

Similar analysis produces the same local iteration sets for statement S_2. Note how the local indices calculated for the local index set of each array has been used to derive the local indices for the local iteration set. The calculation of local index and iteration sets is described in greater detail elsewhere [17].

Handling boundary conditions Because alignment and distribution specifications in Fortran D are fairly simple, local index sets and their derived iteration sets may usually be calculated at compile time. In fact, for most regular computations local index and iteration sets are identical for every processor except for boundary conditions. When boundary conditions for each array dimension or loop are independent, as in the Jacobi example, the Fortran D compiler can store each boundary condition separately. This avoids the need to calculate and store a different result for each processor.

We may summarize independent boundary conditions for iteration or index sets as *pre, mid*, and *post* sets for each loop or array dimension. The mid set describes the interior uniform case. The pre and post iteration sets describe the boundary conditions encountered and their positions. These sets are represented in the Fortran D compiler by ***augmented*** iteration sets. Instead of a single section, each dimension of the augmented iteration set contains three component sections for the pre, mid, and post sets as well as their positions.

```
do l = 1,time
  do j = 2,6
   do k = 2,n
S1     QA = ZA(k,j+1)*ZR(k,j) + ...
S2     ZA(k,j) = ZA(k,j)+.175*(QA-ZA(k,j))
   enddo
  enddo
enddo
```

Figure 4: Livermore Kernel 23

Because boundary conditions for iteration and index sets can be handled in the same manner, we will just discuss an example case for iteration sets. When partitioning the Jacobi example, the following *pre, mid*, and *post* iteration sets are calculated by the Fortran D compiler:

$$\left[\ 1:time,\ \left\{\begin{array}{l} pre = [2:25]\ @p_1 \\ mid = [1:25] \\ post = [1:24]\ @p_4 \end{array}\right\},\ 2:99\ \right]$$

In the ***augmented*** RSD representing the pre, mid, and post iteration sets, "@" indicates the position for each pre or post set. If an interior processor is causing a boundary condition, processors between it and the edge will not be assigned loop iterations.

The iteration set for each processor is calculated by taking the Cartesian product of the pre, mid, and post iteration sets for each dimension of the augmented iteration set. Unfortunately not all boundary conditions may be succinctly represented by augmented iteration sets. In the worst case the Fortran D compiler is forced to derive and store an individual index or iteration set for each processor.

Partitioning scalar assignments Statements performing assignments to scalar variables also present a special case for the Fortran D compiler. Since scalars are replicated, naive application of the ***owner computes*** rule would cause every processors to execute the assignment on all iterations. However, this is usually not necessary.

In the majority of cases the value assigned to the scalar variable is used only in the current loop iteration. These situations are easily detected since the scalar variable has already been labeled ***private*** by the dependence analyzer. The Fortran D compiler assigns to these statements the iteration set consisting of the union of the iteration sets of all statements that use the value of the scalar. These can be determined by tracking all true dependence edges with the scalar as its source. The process is simplified if the iteration sets

However, dependence analysis shows that the only cross-processor true dependences incident on the *rhs* references for statements S_1 and S_2 are loop-carried dependences on the k loop from S_3 and S_4. The tags for these references (labeled as carried) are inserted at the header of the k loop. In the code generation phase they will generate messages that are executed on each iteration of the k loop.

For statements S_3 and S_4, dependence analysis will show that the only cross-processor true dependences incident on their *rhs* references are loop-independent dependences from S_1 and S_2. Tags are placed in the k loop because it is the deepest loop common to both the source and sink of these dependences. We insert tags (labeled independent) for all *rhs* references in S_3 at its enclosing j loop, since it is the next loop deeper than k enclosing S_3.

Similar analysis causes us to insert tags (labeled independent) for all *r*[illegible]eferences in S_4 at its enclosing j loop. As an addi[illegible]al optimization, we can move these tags to the j lo[illegible] enclosing S_3 to combine these messages. This is legal since we are moving tags to a statement that is at the same loop level and between the source and sink of the dependence. In the code generation phase these tags will cause vectorized messages to be generated before the j loop, to be executed on each k loop iteration.

3.2.2 Collective Communication

Communication patterns such as broadcasts and transposes are not well-described by individual RSDs, and may be performed faster using special purpose collective communication routines in any case. To take advantage of these cases, we apply techniques pioneered by Li and Chen to utilize collective communication primitives [23].

As previously described, opportunities for utilizing collective communication are recognized by the Fortran D compiler during communication selection by examing both the array reference and data decomposition information. Loop-invariant subscripts in distributed dimensions correspond to broadcasts. Differences in alignment between the *lhs* and *rhs* may lead to transpose or broadcast calls. Reductions may also require calls to collective communication routines. All such cases are tagged for use during code generation.

3.2.3 Runtime Processing

The presence of complex subscript expressions, index arrays, etc. mean that the Fortran D compiler cannot always determine at compile time what communication is required for a particular reference. We mark such references that do not cause loop-carried true dependences as candidates for applying the *inspector/executor* strategy to calculate nonlocal accesses at runtime [24]. For references that may cause loop-carried true dependences, the Fortran D compiler will resolve the communication at runtime using previously proposed runtime techniques [8, 27, 34].

3.2.4 Additional Optimizations

Other optimizations are being considered for the Fortran D compiler. Communication may be further optimized by considering interactions between all the loop nests in the program. Intra- and interprocedural dataflow analysis of array sections can show that an assignment to a variable is *live* at a point in the program if there are no intervening assignments to that variable. This information may be used to eliminate redundant messages. Calculating array kill information will also be helpful in eliminating both redundant computation and communication.

Replicating computations and processor-specific dead code elimination will be applied to eliminate communication. Data from different arrays being sent to the same processor may also be buffered together in one message to reduce communication overhead.

The *owner computes* rule provides the basic strategy of the Fortran D compiler. We may also relax this rule, allowing processors to compute values for data they do not own. For instance, suppose that multiple *rhs* of an assignment statement are owned by a processor that is not the owner of the *lhs*. Computing the result on the processor owning the *rhs* and then sending the result to the owner of the *lhs* could reduce the amount of data communicated. This optimization is a simple case of the *owner stores* rule proposed by Balasundaram [4].

In particular, for computations performing irregular data accesses it may be desirable for the Fortran D compiler to partition loops amongst processors so that each loop iteration is executed on a single processor, such as in KALI [21] and PARTI [24]. This technique may improve communication and provide greater control over load balance, especially for irregular computations. It also eliminates the need for individual statement guards and simplifies handling of control flow within the loop body.

3.3 Code Generation

Once program analysis and optimization is complete, the code generation phase of the Fortran D compiler utilizes information concerning local index and iteration sets, RSDs, and collective communication to create the actual SPMD node program.

3.3.1 Program Partitioning

Recall that during partitioning analysis, the Fortran D compiler applied the *owner computes* rule to calculate the local iteration set for each statement. In code generation, the compiler must modify the program to ensure that each processor only executes loop iterations and statements in its local iteration set.

Loop bounds reduction and ***guard introduction*** are the two program transformations used to partition the computation among processors. We first reduce the loop bounds so that each processor only executes iterations in the union of the local iteration sets for all the statements within the loop. The Fortran D compiler generates conditional assignments to loop bound reference $B(f(i), j)$ the unknown function f (possibly an index array) requires *runtime processing*. We describe these communication optimizations below.

3.2.1 Message Vectorization

The most basic communication optimization performed by the Fortran D compiler is *message vectorization*. Message vectorization uses the level of loop-carried cross-processor dependences to calculate whether messages may be legally combined into larger messages, enabling efficient program execution.

Algorithm We use the following algorithm from Balasundaram *et al.* and Gerndt to calculate the appropriate loop level for inserting messages for nonlocal references [5, 13]. We define the *commlevel* for loop-carried dependences to be the level of the dependence. For loop-independent dependences we define it to be the level of the deepest loop common to both the source and sink of the dependence.

To vectorize messages for a *rhs* reference R with a nonlocal index set, we examine all cross-processor true dependences with R as the sink. The deepest commlevel of all such dependences determines the loop level at which the message may be vectorized. If the deepest dependence is a loop-carried dependence carried by loop L, we insert a message tag for R marked *carried* at the header for loop L. This tag indicates that nonlocal data accessed by R must be communicated between iterations of loop L.

If the deepest dependence is a loop-independent dependence with loop L as the deepest loop common to both the source and sink. We insert a tag for R marked *independent* at the header of the next deeper loop enclosing R at level $L + 1$, or at R itself if no such loop exists. This tag indicates that nonlocal data accessed by R must be communicated at this point on each iteration of loop L. Additionally, the Fortran D compiler may move this tag to any statement in loop L between the source and the sink of the dependence in order to combine messages arising from different references.

Example 1: Jacobi We illustrate the message vectorization algorithm with three examples. First we examine the Jacobi code in Figure 3. In the communication analysis phase, we have already determined that for the given data decomposition only the *rhs* references $B(i, j-1)$ and $B(i, j+1)$ from S_1 access nonlocal locations. The only cross-processor true dependences incident on these references are loop-carried dependences from the definition to B in S_2. These dependences are carried on the k loop, so we insert their tags (labeled carried) at the header of the k loop. The code generation phase will later insert messages for those references at the beginning of the k loop.

Example 2: Successive over-relaxation (SOR) In the code for SOR in Figure 5, communication analysis discovers that the *rhs* references $A(i+1, j)$ and $A(i-1, j)$ have nonlocal index sets. Dependence analysis shows that the reference $A(i+1, j)$ has a cross-processor true dependence carried on the k loop, so we insert its tag (labeled carried) at the k loop header. The deepest loop-carried true dependence for reference $A(i-1, j)$ is carried on the i loop, so we insert its tag (also labeled carried) at the i loop header.

```
REAL A(100,100)
DECOMPOSITION D(100,100)
ALIGN A, B with D
DISTRIBUTE D(BLOCK,:)
do k = 1,time
  do j = 2,99
    do i = 2,99
      A(i,j) = (ω/4)*(A(i,j-1)+A(i-1,j)+
          A(i+1,j)+A(i,j+1))+(1-ω)*A(i,j)
    enddo
  enddo
enddo
```

Figure 5: Successive Over-Relaxation (SOR)

```
   REAL V(n,n)
   DECOMPOSITION D(n,n)
   ALIGN V with D
   DISTRIBUTE D(BLOCK,BLOCK)
   do k = 1,time
     {* compute red points *}
     do j = 1,n-1,2
       do i = 1,n-1,2
S1       V(i,j) = (ω/4)*(V(i,j-1)+V(i-1,j)+
            V(i,j+1)+V(i+1,j))+(1-ω)*V(i,j)
       enddo
     enddo
     do j = 2,n-1,2
       do i = 2,n-1,2
S2       V(i,j) = (ω/4)*(V(i,j-1)+V(i-1,j)+
            V(i,j+1)+V(i+1,j))+(1-ω)*V(i,j)
       enddo
     enddo
     {* compute black points *}
     do j = 1,n-1,2
       do i = 2,n-1,2
S3       V(i,j) = (ω/4)*(V(i,j-1)+V(i-1,j)+
            V(i,j+1)+V(i+1,j))+(1-ω)*V(i,j)
       enddo
     enddo
     do j = 2,n-1,2
       do i = 1,n-1,2
S4       V(i,j) = (ω/4)*(V(i,j-1)+V(i-1,j)+
            V(i,j+1)+V(i+1,j))+(1-ω)*V(i,j)
       enddo
     enddo
   enddo
```

Figure 6: Red-black SOR

Example 3: Red-black SOR In the code in Figure 6, communication analysis discovers that all *rhs* references except $V(i, j)$ possess nonlocal index sets.

and are therefore combined. This translates to processors 2 through 4 requiring nonlocal data from their left neighbor at every iteration of the j loop.

Additionally, we can recognize that the send occurs only after the last local i loop iteration, and the receive occurs only on the first local i loop iteration. We may thus move the receive before the i loop and the send after the i loop. A similar process is applied to create RSDs for reference $A(i+1,j)$ at the k loop, resulting in the code shown in Figure 8.

Loop-independent messages For messages tagged at loop headers for loop-independent cross-processor dependences, the Fortran D compiler inserts pairs of calls to send and receive routines preceding the loop header. For messages tagged at individual references, the Fortran D compiler inserts pairs of calls to send and receive routines in the body of the loop preceding the reference. All calls are guarded so that the owners execute the sends and recipients execute receives.

To calculate the data that must be sent, the Fortran D compiler builds the RSD for the reference at the loop level that the message is generated. This represents data sent on each loop iteration. This strategy is used to generate messages for the loop-independent dependences in the red-black code in Figure 6.

Collective Communication During communication optimization, opportunities for reductions and collective communication have been marked separately. Instead of individual sends and receives, the Fortran D compiler inserts calls to the appropriate collective communication routines. Additional communication is also appended following loops containing reductions to accumulate and broadcast the results of each reduction.

Runtime Processing Runtime processing is applied to computations whose nonlocal data requirements are not known at compile time. An *inspector* [24] is constructed to preprocess the loop body at runtime to determine what nonlocal data will be accessed. This in effect calculates the *receive* index set for each processor. A global transpose operation between processors is then used to calculate the *send* index sets. Finally, an *executor* is built to actually communicate the data and perform the computation.

An inspector is the most general way to generate *send* and *receive* sets for references without loop-carried true dependences. Despite the expense of additional communication, experimental evidence from several systems show that it can improve performance by grouping communication to access nonlocal data outside of the loop nest, especially if the information generated may be reused on later iterations [21, 24].

The inspector strategy is not applicable for unanalyzable references causing loop-carried true dependences. In this case the Fortran D compiler inserts guards to resolve the needed communication and program execution at runtime [8, 27, 34].

3.3.3 Storage Management

One of the major responsibilities of the Fortran D compiler is to select and manage storage for all nonlocal array references. There are three different storage schemes.

Overlaps Overlaps are expansions of local array sections to accommodate neighboring nonlocal elements [13]. For programs with high locality of reference, they are useful for generating clean code. However, overlaps are permanent and specific to each array, and thus may require more storage.

Buffers Buffers avoid the contiguous and permanent nature of overlaps. They are useful when storage for nonlocal data must be reused, or when the nonlocal area is bounded in size but not near the local array section.

Hash tables Hash tables may be used when the set of nonlocal elements accessed is sparse, as for many irregular computations. They also provide a quick lookup mechanism for arbitrary sets of nonlocal values [18].

The extent of all RSDs representing nonlocal array accesses produced during the message generation phase are examined to select the appropriate storage type for each array reference. If overlaps have been selected, array declarations are modified to take into account storage for nonlocal data. For instance, array declarations in the generated code in Figures 7 and 8 have been extended for overlap regions. If buffer arrays are used, additional buffer array declarations are inserted. Finally, all nonlocal array references in the program are modified to reflect the actual data location selected.

3.4 Advanced Optimizations

3.4.1 Program Transformations

Shared-memory parallelizing compilers apply program transformations to expose or enhance parallelism in scientific codes, using dependence information to determine their legality and profitability [2, 20, 22, 31]. Program transformations are also useful for distributed-memory compilers. The legality of each transformation is determined in exactly the same manner, since the same execution order must be preserved in order to retain the meaning of the original program. However, their profitability criteria are now totally different. We briefly describe some useful transformations in the Fortran D compiler.

Loop Distribution *Loop distribution* separates independent statements inside a single loop into multiple loops with identical headers. Loop distribution may be applied only if the statements are not involved in a recurrence and the direction of existing loop-carried dependences are not reversed in the resulting loops [20, 22].

```
REAL A(100,25), B(100,0:26)
lb1 = 1
ub1 = 25
if (Plocal = 1) lb1 = 2
if (Plocal = 4) ub1 = 24
do k = 1,time
  if (Plocal > 1) send(B(2:99,1), Pleft)
  if (Plocal < 4) send(B(2:99,25), Pright)
  if (Plocal < 4) recv(B(2:99,26), Pright)
  if (Plocal > 1) recv(B(2:99,0), Pleft)
  do j = lb1,ub1
   do i = 2,99
    A(i,j) = (B(i,j-1)+B(i-1,j)+
                B(i+1,j)+B(i,j+1))/4
   enddo
  enddo
  do j = lb1,ub1
   do i = 2,99
    B(i,j) = A(i,j)
   enddo
  enddo
enddo
```

Figure 7: Generated Jacobi

variables to handle any boundary conditions that may exist. For instance, consider the code generated for Jacobi in Figure 7.

With multiple statements in the loop, the local iteration set of a statement may be a subset of the reduced loop bounds. For these statements we need to also add explicit guards based on membership tests for the local iteration set of the statement [8, 27, 34].

3.3.2 Message Generation

The Fortran D compiler uses information calculated in the communication analysis and optimization phases to guide message generation. Non-blocking *sends* and blocking *receives* are inserted for the following types of messages:

Loop-carried messages For messages tagged at loop headers representing *rhs* references with loop-carried dependences, the Fortran D compiler inserts calls to send and receive routines at the beginning of the loop body. To calculate the data that must be communicated, we build the RSD for each *rhs* reference at the level of the loop carrying the dependence. The calls are guarded so that the owners execute the sends on the loop iterations where nonlocal references occur. As an additional optimization, if the send occurs only on the last loop iteration, it may be moved after the loop instead. Similarly, if the receive occurs only on the first loop iteration, it may be moved before the loop.

We illustrate message generation for two example codes, Jacobi and SOR, and describe how the communication for these code are computed. For the Jacobi code in Figure 3, recall that during message vectorization we determined that cross-processor loop-carried

```
REAL A(0:26,100)
lb2 = 1
ub2 = 25
if (Plocal = 1) lb2 = 2
if (Plocal = 4) ub2 = 24
do k = 1,time
  if (Plocal > 1) send(A(1,2:99), Pleft)
  if (Plocal < 4) recv(A(26,2:99), Pright)
  do j = 2,99
   if (Plocal > 1) recv(A(0,j), Pleft)
   do i = lb2, ub2
    A(i,j) = (ω/4)*(A(i,j-1)+A(i-1,j)+
         A(i+1,j)+A(i,j+1))+(1-ω)*A(i,j)
   enddo
   if (Plocal < 4) send(A(25,j), Pright)
  enddo
enddo
```

Figure 8: Generated SOR

true dependences existed for the *rhs* references in S_1 on the k loop. We thus tagged their messages at the k loop header as carried.

Now we need to compute using RSDs what data needs to be communicated. Boundary conditions cause three RSDs to be generated for each *rhs* reference. Below are the RSDs for the reference $B(i,j+1)$ at the k loop level.

$$\begin{aligned} Proc(1) &= [\,2:99,\,3:26\,] \\ Proc(2:3) &= [\,2:99,\,2:26\,] \\ Proc(4) &= [\,2:99,\,2:25\,] \end{aligned}$$

We subtract the local index set from these RSDs to determine the RSDs for the nonlocal index set. The nonlocal RSD for $Proc(1)$ and $Proc(2{:}3)$ are both [2:99, 26] and are therefore combined. The RSD for $Proc(4)$ consists of only local data and is discarded.

The sending processor is determined by computing the owners of the section [2:99,26] @ $Proc(1{:}3)$ resulting in $Proc(2{:}4)$ sending data to their left processors. To compute the data that needs to be sent, we transpose the nonlocal RSD to obtain the section [2:99,26−25] = [2:99,1]. Similar analysis is performed to the other array references on the right hand side. The communication generated is shown in Figure 7.

For the SOR code depicted in Figure 5, we perform additional optimizations. Due to boundary conditions, three RSDs are generated for each reference. Below are the RSDs generated for the reference $A(i-1,j)$ at the i loop level.

$$\begin{aligned} Proc(1) &= [\,1:24,\,j\,] \\ Proc(2:3) &= [\,0:24,\,j\,] \\ Proc(4) &= [\,0:23,\,j\,] \end{aligned}$$

The local index set is subtracted from these RSDs to determine the RSDs for the nonlocal index set. The nonlocal RSD for $Proc(1)$ produces the empty set. The nonlocal RSD for both $Proc(2{:}3)$ and $Proc(4)$ are $[0,j]$

Loop distribution is useful in separating statements in loop nests with different local iteration sets. This enables more opportunity for loop bounds reduction and avoids evaluating guards at runtime. Loop distribution may also separate the source and sink of loop-carried or loop-independent cross-processor dependences, allowing individual messages to be combined into a single vector message.

Loop Interchange *Loop interchange* swaps adjacent loop headers to alter the traversal order of the iteration space. It may be applied only if the source and sink of each dependence are not reversed in the resulting program. This may be determined by examining the distance or direction vector associated with each dependence [2, 31].

Strip Mining *Strip mining* increases the step size of an existing loop and adds an additional inner loop. Strip mining is always legal. The Fortran D compiler may apply strip mining in order to reduce storage requirements for computations. It may also be used with loop interchange to help exploit pipeline parallelism, as discussed in the next section.

3.4.2 Pipelined Computations

In *loosely synchronous* computations all processors execute SPMD programs in a loose lockstep, alternating between phases of local computation and synchronous global communication [12]. These problems are well structured; they achieve good load balance because all processors are utilized during the computation phase. For instance, Jacobi and red-black SOR are loosely synchronous. The Fortran D compilation strategy presented so far is well-suited to compiling such programs, since it identifies and inserts efficient vectorized or collective communication at appropriate points in the program.

However, a different class of computations contain loop-carried cross-processor data dependences that sequentialize computations over distributed array dimensions. Synchronization is required and processors are forced to remain idle at various points in the computation, possibly resulting in very poor load balance. We call these computations, such as SOR, *pipelined.* They present opportunities for optimization to exploit partial parallelism through pipelining, enabling processors to overlap computation with one another (hence the name). In this section we discuss how to identify and optimize pipelined computations.

Cross-Processor Loops The Fortran D compiler identifies pipelined computations using *cross-processor* loops. We classify loops in numeric computations as either space-bound or time-bound. *Space-bound* loops iterate over the data space, with each iteration accessing part of each array. These loops are usually parallel in data-parallel computations, but may be sequential if they cause a computation *wavefront* to sweep across the data space.

Time-bound loops, on the other hand, correspond to time steps in the computation, with each iteration accessing much or all of the data space. They usually enclose space-bound loops and need to be executed sequentially. The Fortran D compiler labels loops as *cross-processor* if they are sequential space-bound loops causing computation wavefronts that cross processor boundaries (*i.e.*, sweeps over the distributed dimensions of the data space). Cross-processor loops may be calculated using the algorithm in Figure 9.

Figure 10 illustrates cross-processor dependences and loops. We denote cross-processor loops as **do***. All loops in the example are space-bound loops that sweep the data space. In Loop 1, the i loop is cross-processor because the computation wavefront sweeps the i dimension across processors. There are no cross-processor loops in Loop 2 because the computation wavefront is internalized and does not cross processor boundaries. In Loop 3 both the i and j loops are cross-processor because the computation wavefront sweeps across processors in both dimensions.

These examples make it clear how cross-processor loops may be used to classify computations. Computations such as Loop 2 that do not possess cross-processor loops are loosely synchronous, since all processors may execute in parallel. Computations such as Loops 1 & 3 that do possess cross-processor loops are pipelined, since processors must wait in turn for computation to be completed.

Improve Partial Parallelism Parallelism may be exploited in pipelined computations through *message pipelining*—sending a message when its value is computed, rather than when its value is needed [27]. Rogers and Pingali applied this optimization to a Gauss-Seidel (a special case of SOR) computation that is distributed cyclically.

For pipelined computations, transformations such as loop interchange and strip mining may also be needed to balance computation and communication. The algorithm for loop interchange is simple—in the case of a cyclic distribution, move cross-processor loops as far out as possible. For a block distribution, interchange cross-processor loops as deeply as possible. Strip mining the loop may also reduce communication overhead. Legality is determined in the same manner as for *unroll-and-jam* [20]. The strip size is machine dependent and will be determined empirically. These values will be fed into the compiler to enable calculation of the strip size.

Figure 11 depicts how loop interchange and strip mining may be used in conjunction to exploit pipeline parallelism. It also shows the tradeoffs between communication and computation that must be considered when compiling pipelined computations. Note that in this example message pipelining is insufficient—the computation order must also be changed.

INPUT:

Loop nest with index variables $\{i_1, \ldots, i_n\}$
List of all loop-carried true dependences
Data decomposition of all distributed arrays in loop nest

OUTPUT:

Loops = List of cross-processor loops

Loops $\leftarrow \emptyset$
for each loop-carried true dependence between references $A(f_1, \ldots, f_m)$ and $A(g_1, \ldots, g_m)$ **do**
  for each distributed dimension k of A **do**
    {* f_k and g_k are subscripts in dimension k, and $i_j \in \{i_1, \ldots, i_n\}$ *}
    if $f_k \neq g_k$ **or** f_k is not of form $\alpha i_j + \beta$ **then**
      for each index variable i_l present in either f_k or g_k **do**
        if loop i_l contains both references (*i.e.*, perfectly nested) **then**
          Loops $\leftarrow$ *Loops* $\cup \{i_l\}$
        endif
      endfor
    endif
  endfor
endfor

Figure 9: Algorithm to Calculate Cross-Processor Loops

	Loop 1	**Loop 2**	**Loop 3**

Data Decomposition	`DECOMPOSITION A(N,N)`
	`REAL X(N,N)`
	`ALIGN X(I,J) with A(I,J)`
	`DISTRIBUTE A(BLOCK,:)`
		`DECOMPOSITION A(N,N)`
		`REAL X(N,N)`
		`ALIGN X(I,J) with A(I,J)`
		`DISTRIBUTE A(:,BLOCK)`
			`DECOMPOSITION A(N,N)`
			`REAL X(N,N)`
			`ALIGN X(I,J) with A(I,J)`
			`DISTRIBUTE A(BLOCK,BLOCK)`

Loop Nest & Cross-Processor Loops	`do* i = 2, N`
	`do j = 1, N`
	`X(i,j) = X(i-1,j)`
	`enddo`
	`enddo`
		`do i = 2, N`
		`do j = 1, N`
		`X(i,j) = X(i-1,j)`
		`enddo`
		`enddo`
			`do* i = 2, N`
			`do* j = 2, N`
			`X(i,j) = X(i-1,j)+X(i,j-1)`
			`enddo`
			`enddo`

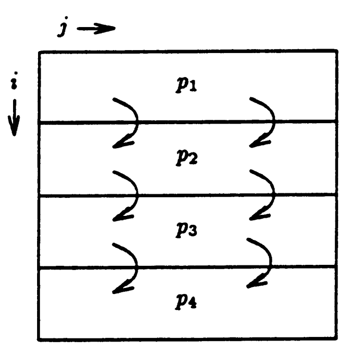
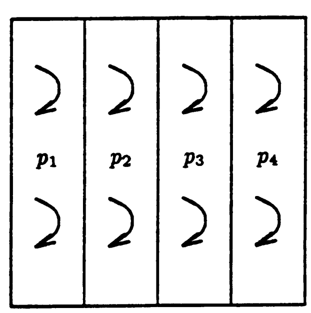
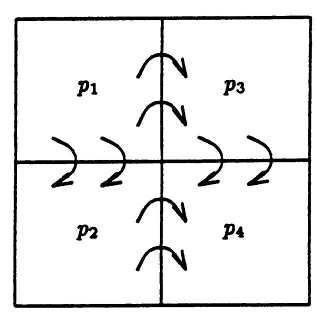
Data Space & Cross-Processor Dependences			

Figure 10: Examples of Cross-Processor Dependences and Loops

overhead. *Loop jamming (fusion)* and *strip mining* are applied when writing array elements to exploit parallelism through pipelining. Global *accumulate* (reduction) operations are recognized and supported. Unlike other systems, program partitioning produces distinct programs for each node processor.

Crystal is a high-level functional language compiled to distributed-memory machines using both automatic data decomposition and communication generation [23]. Program analysis and optimization is simplified because it targets a purely functional language. CRYSTAL pioneered the strategy of identifying collective communication opportunities used in the Fortran D compiler.

ASPAR is a compiler that automatically generates data decompositions and communication for Fortran programs with BLOCK distributions [19]. ASPAR performs simple dependence analysis using *A-lists* to detect parallel loops performing regular computations on distributed arrays. A *micro-stencil* is derived and used to generate a *macro-stencil* to identify communication requirements.

Kali is the first compiler that supports both regular and irregular computations on MIMD distributed-memory machines [21]. Since dependence analysis is not provided, programmers must declare all parallel loops. Instead of deriving a parallel program from the data decomposition, KALI requires that the programmer explicitly partition loop iterations onto processors using an *on clause*.

PARTI is a set of runtime library routines that support irregular computations on MIMD distributed-memory machines. PARTI is the first to propose and implement user-defined irregular distributions [24] and a hashed cache for nonlocal values [18]. PARTI has also motivated the development of ARF, a compiler that automatically generates *inspector* and *executor* loops for runtime preprocessing of programs with BLOCK, CYCLIC, and user-defined irregular distributions [33].

4.1 Comparison with Fortran D

The Fortran D compiler integrates more compiler optimizations than the first generation research systems described, and in addition possesses two main advantages. First, dependence analysis enables the compiler to exploit parallelism without functional specifications (*e.g.*, CRYSTAL, ID NOUVEAU) or explicitly parallel loops (*e.g.*, KALI, ARF). Precise analysis also allows the compiler to perform more optimizations. For instance, only SUPERB and ASPAR possess the dependence analysis capabilities needed to discover the parallelism in red-black SOR. However, their use of contiguous overlaps prevents them from exploiting the parallelism.

Second, the Fortran D compiler performs its analysis up front and uses the results to drive code generation, unlike transformation-based systems (*e.g.* PARASCOPE, ID NOUVEAU, SUPERB) that begin by inserting guards and element-wise messages, then apply program transformations and partial evaluation in order to produce more efficient code. The Fortran D approach is simpler and provides greater flexibility. For instance, the compiler may apply program transformations without the possibility of introducing deadlock due to message reordering.

5 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 Fortran D, a version of Fortran enhanced with data decompositions, provides such a portable data-parallel programming model. Its success will depend on the effectiveness of the compiler, as well as environmental support for automatic data decomposition and static performance estimation [5, 6, 16].

The current prototype of the Fortran D compiler performs message vectorization for block-distributed arrays. Though significant work remains to implement other optimizations presented in this paper, preliminary results lead us to believe that the Fortran D compiler will generate efficient code for a large class of data-parallel programs with only minimal user effort.

6 Acknowledgements

The authors wish to thank Vasanth Bala, Geoffrey Fox, Marina Kalem, and Uli Kremer for inspiring many of the ideas in this work. We are also grateful to the ParaScope and Fortran D research groups for their assistance in implementing the Fortran D compiler.

References

[1] E. Albert, K. Knobe, J. Lukas, and G. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, New Haven, CT, July 1988.

[2] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] F. André, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[4] V. Balasundaram. Translating control parallelism to data parallelism. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, TX, March 1991.

[5] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

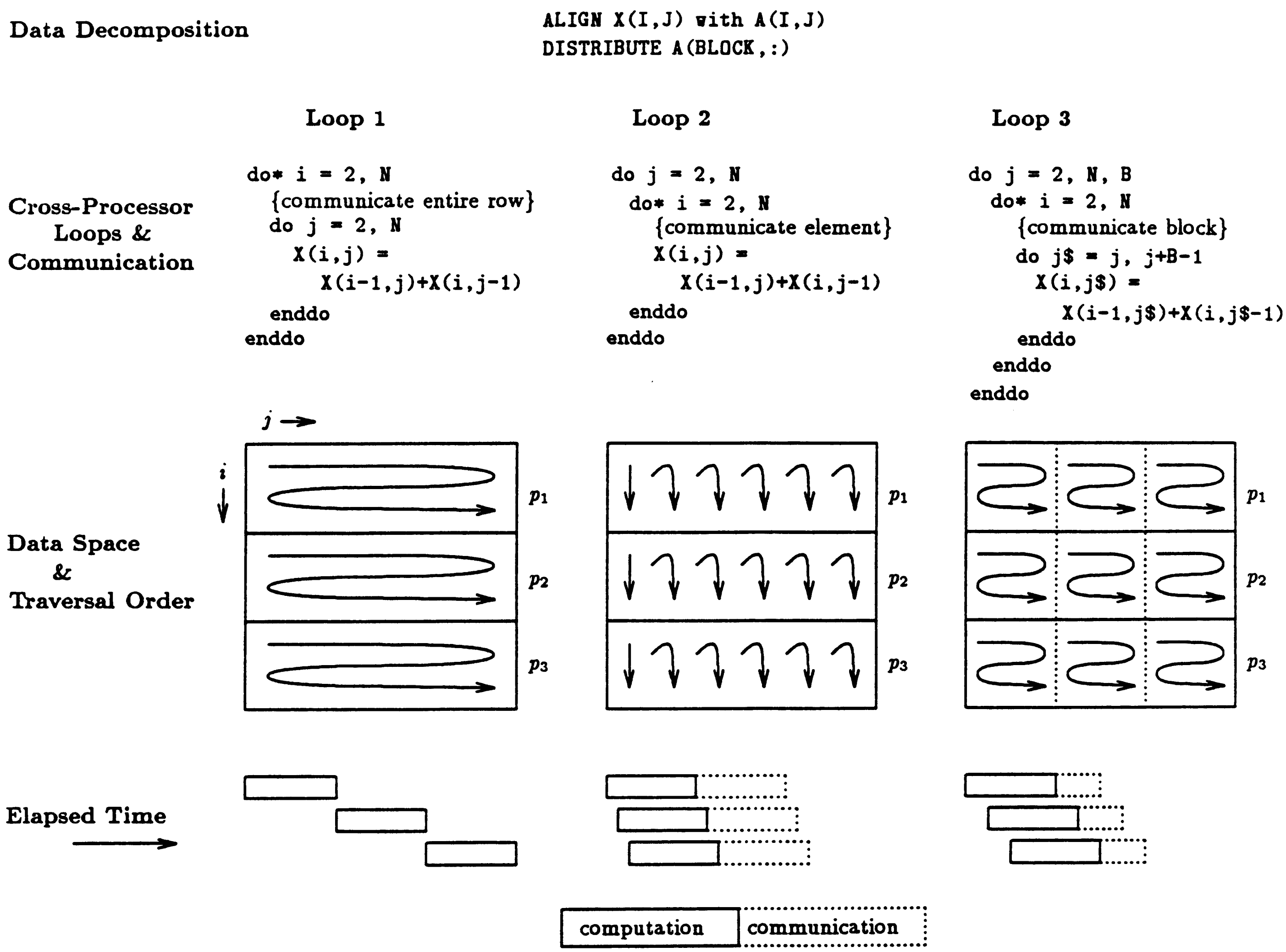


Figure 11: Examples of Pipelined Computations

4 Related Work

We view the Fortran D compiler as a second-generation distributed-memory compiler that integrates and extends analysis and optimization techniques from many other research projects. It is related to other distributed-memory compilation systems such as AL [30], CM FORTRAN [1], DINO [28], MIMDIZER [15], PANDORE [3], PARAGON [10], and SPOT [29], but mostly builds on the following research projects.

SUPERB is a semi-automatic parallelization tool that supports arbitrary user-specified contiguous BLOCK distributions [13, 34]. It originated *overlaps* as a means to both specify and store nonlocal data accesses. *Exsr* statements are inserted in the program to communicate overlap regions. Data dependence information is then used to apply *loop distribution* and vectorize these statements, resulting in vectorized messages. SUPERB also performs interprocedural analysis and code generation.

ParaScope is a parallel programming environment that supports a prototype distributed-memory compiler [8]. User-defined distribution functions are used to specify the data decomposition for Fortran programs. The compiler inserts *load* and *store* statements to handle data movement and then applies numerous program transformations to optimize guards and messages.

Id Nouveau is a functional language enhanced with BLOCK and CYCLIC distributions [26, 27]. Dependence analysis is avoided through the use of write-once arrays called *I-structures*. Initially, *send* and *receive* statements are inserted to communicate each nonlocal array access. *Message vectorization* is then applied to previously written array elements to reduce message

[6] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[7] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

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

[9] D. Callahan, K. Kennedy, and U. Kremer. A dynamic study of vectorization in PFC. Technical Report TR89-97, Dept. of Computer Science, Rice University, July 1989.

[10] A. Cheung and A. Reeves. The Paragon multicomputer environment: A first implementation. Technical Report EE-CEG-89-9, Cornell University Computer Engineering Group, Ithaca, NY, July 1989.

[11] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[12] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

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

[14] 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.

[15] R. Hill. MIMDizer: A new tool for parallelization. *Supercomputing Review*, 3(4):26–28, April 1990.

[16] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[17] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, to appear 1991.

[18] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12(4), August 1991.

[19] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[20] 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.

[21] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[22] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[23] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[24] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[25] C. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer*, 23(12):13–23, December 1990.

[26] K. Pingali and A. Rogers. Compiling for locality. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, June 1990.

[27] 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.

[28] M. Rosing, R. Schnabel, and R. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[29] D. Socha. Compiling single-point iterative programs for distributed memory computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[30] P.-S. Tseng. A parallelizing compiler for distributed memory parallel computers. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[31] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

[32] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[33] 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.

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