parallel computing represents the only plausible way to continue to increase the computational power available to scientists and engineers. Parallel computers, however, 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 [8, 35]. The resulting programs are easily maintained, debugged, and ported 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. Multiple-instruction, multiple-data (MIMD) shared-memory machines such as the Cray Research Y-MP C90 are programmed with explicit synchronization and parallel loops found in Parallel Computing Forum (PCF) Fortran [24]. Single-instruction, multiple-data (SIMD) machines such as the Thinking Machines CM-2 are programmed using parallel array constructs found in Fortran 90 [3].

MIMD distributed-memory machines such as the Intel Paragon provide the most difficult programming model. Users must write message-passing Fortran 77 programs that deal with separate address spaces, synchronizing processors, and communicating data using messages. The process is time-consuming, tedious, and error-prone. Significant increases in source code size are not only common but expected.

Because parallel programs are extremely machine-specific, scientists are discouraged from utilizing parallel 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 establish a machine-independent programming model. It must be easy to use, yet perform with acceptable efficiency on different parallel architectures, at least for data-parallel scientific codes.

The question is whether any existing Fortran dialect suffices. Parallel Computing Fortran is undesirable because it is easy to inadvertently write programs with data races that produce indeterminate results. Message-passing Fortran 77 is portable, but difficult to use. Fortran 90 is promising, but may not be sufficiently flexible. What all these languages lack is a way to specify the decomposition and placement of data in the program.

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. Most parallel programming languages, however, provide no support for data decomposition [27].

For these reasons, we have 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. As shown in Figure 1, 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. We should note that our goal in designing Fortran D is not to support the most general data decompositions possible. Instead, our intent is to provide data decompositions that are both powerful enough to express data parallelism in scientific programs, and simple enough to permit the compiler to produce efficient programs.

An essential part of the research program is demonstrating a compiler technology capable of achieving both expressive power and efficiency in the generated code. To that end, we have embarked on a program to implement Fortran D for several different architectures. The most mature implementation, which we have been pursuing for over two years, is for MIMD distributed-memory machines. In this article we describe the design of that compiler. Its goal is to automate the task of deriving node programs based on the data decomposition. For these machines, it is particularly important to reduce both communication and load imbalance. We present a code generation strategy based on the concept of data dependence [23] that unifies and extends previous techniques.

The remainder of this article presents the data decomposition specifications in Fortran D, basic compiler analysis and code generation algorithms, and compiler optimizations to reduce communication cost and exploit pipeline parallelism. We conclude with a comparison to related work and a description of the current status of the compiler.

**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-grained 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-grained parallelism defined by the physical parallel machine.

Fortran D provides data decomposition specifications for these two levels of parallelism using DECOMPOSITION, ALIGN, and DISTIBUTE 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 one another. 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. Perfect alignment results if no subscripts are used. In the following example,

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

D is declared to be a 2D decomposition of size $N \times N$. Array A is then aligned with respect to $D$ with the dimensions permuted and offsets specified within each dimension.

After arrays have been aligned with a decomposition, the DISTIBUTE 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. Pre-defined 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 $D$ by (CYCLIC,:) partitions the rows of $D$ in a round-robin fashion among processors. These sample data alignment and distributions are shown in Figure 2.

```fortran
DECOMPOSITION D(N,N)
DISTIBUTE D(:,BLOCK)
DISTIBUTE D(CYCLIC,)
```

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

Many previous researchers have supplied data decomposition extensions [7, 22, 28, 30, 32, 37]. Because
its goal is to support both SIMD and MIMD architectures, Fortran D is the first language to implement both alignment and distribution specifications. Fortran D also provides a FORALL loop, irregular data distribution, 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 [10].

**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 to 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 exploit parallelism and reduce communication cost. It translates a Fortran D program into a single-program, multiple-data (SPMD) program with explicit message-passing that executes 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 [7, 28, 37]. The compiler is subdivided into three major phases—program analysis, program optimization, and code generation. The structure of the compiler is shown in Figure 3.

**Program Analysis**

**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 the 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 [23]. 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 [1]. 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 [34].

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 prototype Fortran D compiler is being developed in the context of the ParaScope programming environment and incorporates the analysis capabilities described in the following paragraphs [6, 21].

**Scalar dataflow analysis.** Control flow, control dependence, and live range information are computed during the scalar dataflow analysis.
phase. In addition, scalar and array 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 program analysis and optimization. Consider the following example:

\[
\begin{align*}
\text{do } & 1 = 1, n - 1 \\
F(i) & = F(i)/B(n) \\
\end{align*}
\]

If constant propagation is able to produce a constant value for \(n\), or if \(n\) is identified as a loop-invariant expression, the Fortran D compiler can communicate \(B(n)\) with an efficient broadcast preceding the loop.

Symbolic analysis also recognizes reductions, operations such as SUM, MIN, or MAX 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 later use.

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 the distance and direction vectors. This information is used to guide subsequent compiler analysis and optimization.

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 decomposition. 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.

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 and 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 article we will write RSDs as \([l_1; u_1; s_1, \ldots]\), where \(l_1, u_1\), and \(s_1\) indicate the lower bound, upper bound, and the step of the \(i\)th 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. Figure 4 is an example illustrating the program and the node program produced when array \(A\) is block-distributed across four processors.

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 state-
Local iteration sets. Once the local index set for each array has been calculated, the Fortran D compiler can partition 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 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):

$$\begin{align*}
\text{Proc}(1) &= [1 : \text{time}, 2 : 25, 2 : 99] \\
\text{Proc}(2) &= [1 : \text{time}, 1 : 25, 2 : 99] \\
\text{Proc}(3) &= [1 : \text{time}, 1 : 24, 2 : 99]
\end{align*}$$

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 have 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, in 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.

Because boundary conditions for iteration and index sets can be handled in the same manner, we will restrict our discussion to an example case for iteration sets. When partitioning the Jacobi example, the pre-, mid-, and post-iteration sets are calculated by the Fortran D compiler as illustrated in Figure 6.

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. A pre- or post-iteration set may also be empty if that boundary condition does not exist.

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.

Private variables and reductions. Statements performing assignments to scalar and replicated array variables present a special challenge for the Fortran D compiler. Naive application of the owner computes rule would cause every processor to execute the assignment on all iterations. The assignment, however, can often be partitioned because its value is used only in the current loop iteration. These cases are readily recognized, since the variable being assigned will have been labeled private during dependence

To partition statement $S$, the Fortran D compiler calculates the union of the iteration sets of all statements that use $S$. These statements can be identified by tracking all true dependence edges with $S$ as its source. This union becomes the iteration set for $S$. The process is simplified if the iteration sets are calculated in reverse order for statements in each loop nest.

For instance, consider the loop in Figure 7. Because $QA$ is a replicated scalar, the owner computes rule would assign all loop iterations as its source. This union becomes the iteration set for the statement. 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, a statement or group of statements will be marked during dependence analysis as performing a reduction. To parallelize the reduction, an iteration set that partitions the computation across processors should be selected. To reduce data movement, the Fortran D compiler may partition the computation using the local iteration set of a variable on the righthand side (rhs). Communication will later be inserted to accumulate the partial results.

Communication Analysis. Once partitioning analysis determines how data and work are partitioned 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 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 only local accesses occur, the rhs reference 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:

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

Since the local index set for $B$ is $[1:100, 1:25]$, $B(i, j - 1)$ 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.

Program Optimization

The program optimization phase of the Fortran D compiler utilizes the results of program analysis to improve program performance. Its two primary goals are to exploit parallelism and reduce communication overhead. Most computations are fully data-parallel; parallelism is discovered and utilized during partitioning analysis. More advanced optimizations are required to exploit parallelism for pipeline computations. We defer their discussion to the subsection "Parallelism Optimizations."

In this section we concentrate on optimizations to reduce communication overhead. It is particularly useful to consider cross-processor dependences—dependences whose endpoints are executed by different processors.

**Message Vectorization.** A naive but workable algorithm known as run-time resolution inserts guarded send

![Figure 4. Converting global to local indices](image)

![Figure 5. Jacobi](image)

![Figure 6. Augmented iteration set for Jacobi](image)

![Figure 7. Livermore Kernel 23](image)
and/or recv operations directly preceding each nonlocal reference [7, 28, 37]. Unfortunately, this simple approach generates many small messages that prove extremely inefficient due to communication overhead [18, 28].

The most basic communication optimization performed by the Fortran D compiler is message vectorization. It uses the level of loop-carried data dependences to calculate whether communication may be legally performed at outer loops. This replaces many small messages with one large message, reducing both message start-up cost and latency.

**Algorithm.** We use the following algorithm from Balasundaram et al. and Gerndt to calculate the appropriate loop level to insert messages for nonlocal references [4, 12]. 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 commlevel is for a 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 $L$.

Otherwise, the deepest commlevel is for a loop-independent dependence with loop $L$ as the deepest loop enclosing 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 5. 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 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 inside the $k$ loop.

**Example 2: Successive over-relaxation (SOR).** In the code for SOR in Figure 8, communication analysis discovers that the rhs references $A(i + 1, j)$ and $A(i - 1, j)$ have nonlocal index sets. Since dependence analysis shows that the reference $A(i + 1, j)$ has a cross-processor true dependence carried on the $k$ loop, 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.

**Example 3: Red-black SOR.** In the code in Figure 9, communication analysis discovers that all rhs references except $V(i, j)$ possess nonlocal index sets. Dependence analysis, however, shows that the only cross-processor true dependences incident on the rhs references for statements $S_3$ and $S_4$ are carried 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. During the code generation phase they will generate messages at the beginning of the $k$ loop.

For statements $S_3$ and $S_4$, dependence analysis shows that the only cross-processor true dependences incident on their rhs references are loop-independent dependences from $S_1$ and $S_2$. Their commlevel is set to the $k$ loop because it is the deepest loop enclosing 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. Similar analysis causes us to insert tags (labeled independent) for all rhs references in $S_4$ at its enclosing $j$ loop.

**Communication Selection.** Message vectorization determines where communication should be inserted, but the Fortran D compiler also needs to select an efficient communication mechanism. We do this by comparing the subscript expression of each distributed dimension in the rhs with its aligned dimension in the lhs reference.

Consider the example shown in Figure 10. Message vectorization determines that communication can be extracted from both the $i$ and $j$ loops. 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.

In $S_1$, the aligned dimensions of both the lhs and rhs references contain constant offsets to the same loop index variable. For these stencil computations involving shifts, individual calls to send and recv primitives are very efficient. This is the case for the Jacobi, SOR, and Red-black SOR examples previously discussed.

**Collective Communication.** More complicated subscript expressions indicate the need for collective communication [25]. For example, the loop-invariant subscript for $B(c, j)$ in $S_2$ can be efficiently communicated using broadcast. Collective communication is also useful in performing transposes for differing alignments between lhs and rhs references, or accumulating partial
results for reductions. Collective communication is selected because these communication patterns are not well-described by individual messages, and can be performed significantly faster using special-purpose routines. For these cases the Fortran D compiler applies techniques pioneered by Li and Chen [25].

**Run-time Processing.** A third type of communication is needed to communicate the values needed by $B(f(i),j)$ in $S_3$. Because $f$ is an irregular function (e.g., an index array), the Fortran D compiler cannot precisely determine at compile time what communication is required. However, inspectors and executors may be created during code generation to combine messages at run-time [22, 26].

**Additional Optimizations.** The Fortran D compiler also performs other communication optimizations. Message coalescing combines messages for different references to the same array. Message aggregation combines messages from different arrays to the same processor, at the expense of copying them to a single contiguous buffer.

A number of optimizations seek to hide communication overhead by overlapping messages with computation. Message pipelining attempts to hide message transit time by separating send and recv primitives for element messages; vector message pipelining does the same for vectorized messages. Iteration reordering extracts local loop iterations to increase the amount of computation that can be overlapped. Nonblocking messages rely on architectural support to hide or eliminate message copy times. These optimizations are discussed in detail elsewhere [18, 22, 28].

Communication may also be optimized by considering interactions between all the loop nests in the program. Intra- and inter-procedural 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 array kill information may be used to eliminate or combine redundant messages. Relaxing the owner computes rule may also improve communication.

**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.

**Program Partitioning.** The Fortran D compiler partitions data by reducing array bounds so that each processor allocates storage only for locally owned data. In addition, the compiler must modify the program to ensure that each processor only executes a statement on loop iterations in its local iteration set.

**Loop bounds reduction and guard introduction** are the two program transformations used to instantiate the computation partition. The Fortran D compiler first reduces the loop bounds so that each processor only executes iterations in the union of local iteration sets for all statements in the loop. It then inserts code to calculate boundary conditions, as shown the bounds generated for $j$ loop in Figure 11.

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 the compiler needs to add explicit guards based on membership tests for the local iteration set of the statement [7, 17, 28, 37].

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

- **Loop-independent messages.** For messages tagged at loop headers for loop-independent cross-processor dependences, the Fortran D compiler inserts calls to send and recv primitives preceding the loop header. For messages tagged at individual references, the

---

**Figure 8. Successive Over-Relaxation (SOR)**

**Figure 9. Red-black SOR**

**Figure 10. Communication selection example**

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

```fortran
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 = 3, N-1,2
      V(1,j) = (ω/4)*(V(1,j-1)+V(1-1,j)+V(1,j+1))+(1-ω)*V(1,j)
   enddo
   { *compute black points* }
   do j = 3, N-1,2
      V(1,j) = (ω/4)*(V(1,j-1)+V(1-1,j)+V(1,j+1))+(1-ω)*V(1,j)
   enddo
enddo
```

---

**Figure 10. Communication selection example**

---

**Figure 8. Successive Over-Relaxation (SOR)**

**Figure 9. Red-black SOR**

**Figure 10. Communication selection example**

---

**Figure 8. Successive Over-Relaxation (SOR)**

**Figure 9. Red-black SOR**

**Figure 10. Communication selection example**

---

**Figure 8. Successive Over-Relaxation (SOR)**

**Figure 9. Red-black SOR**

**Figure 10. Communication selection example**
Fortran D compiler inserts send and recv in the body of the loop preceding the reference. All messages are guarded so that the owners execute send and recipients execute recv. 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 preceding the loop nests enclosing $S_3$ and $S_4$ for Red-black SOR in Figure 9.

Loop-carried messages. The situation is more complex for messages representing loop-carried dependences. To calculate the data that must be communicated, we build the RSD for each rhs reference at the level of the loop L carrying the dependence. If iterations of L are executed by all processors, the Fortran D compiler inserts calls to send and recv primitives inside the loop header for L, at the beginning of the loop body. If the iterations of L are partitioned across processors, loop-carried messages represent data synchronization. The compiler inserts calls to recv preceding loop L, since they occur before the local iterations of L. Similarly, calls to send are inserted after L, since they are executed after the local iterations of L.

We illustrate message generation for two examples. For the Jacobi code in Figure 5, recall that the $k$ loop carries true dependences for the rhs references in $S_1$. These messages were tagged at the $k$ loop header as carried. We first compute RSDs for the data that need 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.

Proc(1) = [2:99, 3:26]  
Proc(2:3) = [2:99, 2:25]  
Proc(4) = [2:99, 2:25]

We subtract the local index set from these RSDs to determine the RSDs for the nonlocal index set. The nonlocal RSDs for Proc(1) and Proc(2:3) are both [2:99, 26] and 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 must be sent, we translate the local indices of the receiving processors to that of the sending processors, obtaining the section [2:99, 26−25] = [2:99, 1]. Since loop $k$ is executed by all processors, the messages are inserted at the beginning of the loop body. Messages for $B(i, j-1)$ are calculated in a similar manner. The communication generated is shown in Figure 11.

Now consider the SOR code depicted in Figure 8. Dependences for $A(i+1, j)$ are carried on the $k$ loop, causing vectorized messages to be inserted at the beginning of the $k$ loop body as in Jacobi. The communication generated for $A(i−1, j)$ is more complicated. Boundary conditions and dependences carried by the $i$ loop cause the following three RSDs to be generated at the level of the $i$ loop.

Proc(1) = [1:24, j]  
Proc(2 : ) = [0 : 24, j]  
Proc(4) = [0 : 23, j]

The local index set is subtracted from these RSDs to determine the
RSDs for the nonlocal index set, producing the empty set for Proc(1). The nonlocal RSDs for both Proc(2:3) and Proc(4) are [0, j] and are combined. This shows that processors 2 through 4 require data from their left neighbor. Iterations of the i loop are partitioned, alerting the Fortran D compiler that the send for A[i-1, j] occurs after the last local i loop iteration, and the recv occurs before the first local i loop iteration. It thus inserts the recv before the i loop and the send after the i loop, resulting in the code shown in Figure 12.

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

Run-time Processing. Run-time processing is applied to computations whose nonlocal data requirements are not known at compile time. An inspector [26] is constructed to pre-process 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 shows 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 [22, 26]. 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 using run-time resolution [7, 28, 37].

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 [12]. For programs with high locality of reference, overlaps are useful for generating clean code. They are, however, 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 look-up mechanism for arbitrary sets of nonlocal values [19].

The extent of all RSDs representing nonlocal accesses produced during message generation are examined to select the appropriate storage type for each array. 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 11 and 12 have been extended for overlap regions. If buffers 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.

Parallelism Optimizations

This section describes how the Fortran D compiler exploits parallelism found in pipelined computations. We begin by describing some program transformations.

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 [1, 21, 23, 34]. 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 to retain the meaning of the original program. However, their profitability criteria are now totally different.

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 [21, 23]. It can separate statements in loop nests with different local iteration sets, avoiding the need to evaluate 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 Fusion. Loop fusion combines multiple loops with identical headers into a single loop. It is legal if the direction of existing dependences are not reversed after fusion [23, 34]. Loop fusion can improve data locality, but its main use is to enable loop interchange and stripmining.

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 de-
Strip Mining. Strip mining increases the step size of an existing loop and adds an additional inner loop. The legality of strip-mining followed by loop interchange is determined in the same manner as for unroll-and-jam [21]. 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 following subsection.

**Pipeline Computations.** In loosely synchronous computations all processors execute SPMD programs in a loose lockstep, alternating between phases of local computation and synchronous global communication [11]. These problems 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 compiler identifies pipelined computations using cross-processor loops. We classify loops in numeric computations as either time-bound or space-bound. Time-bound loops correspond to time steps in the computation, with each iteration accessing much or all of the data space. They are usually outermost loops that need to be executed sequentially. In comparison, 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.

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., sweep over the distributed dimensions of the data space). Cross-processor loops may be calculated by considering all pairs of array references that cause loop-carried true dependences. If nonidentical subscript expressions occur in a distributed dimension of the array, index variables appearing in the subscript expressions belong to cross-processor loops. The algorithm for calculating cross-processor loops is shown in Figure 13. In most cases, cross-processor loops are loops carrying true dependences whose iterations have been partitioned across processors.

Figure 14 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 and 3 that do possess cross-processor loops are pipelined, since processors must wait in turn for computation to be completed.

**Exploiting Pipeline Parallelism.** Parallelism may be exploited in pipelined computations through message pipelining—sending a message when its value is first computed,
rather than waiting until its value is needed [28]. Rogers and Pingali applied this optimization to a Gauss-Seidel computation (a special case of SOR) that is distributed cyclically. More sophisticated approaches, however, are usually required.

Figure 15 illustrates the tradeoffs between communication and parallelism that must be considered when compiling pipelined computations. It presents the program text, data space traversal order, and a processor trace for three versions of the computation. In the processor trace, elapsed time proceeds from left to right. The computation for each processor is represented as a solid line, and messages are shown as dashed lines between processors.

In the original program execution order, message vectorization minimizes communication overhead but sequentializes the computation. Applying message pipelining...

---

**Figure 14.** Examples of cross-processor dependencies and loops

**Figure 15.** Examples of pipelined computations
ing alone is insufficient, since only the computation for the last row will be pipelined. The key observation is that for some pipelined computations, the program execution order must also be changed. We present two optimizations to exploit pipeline parallelism. Fine-grain pipelining interchanges cross-processor loops as deeply as possible to maximize parallelism. This is a major improvement over sequentialized computation, but incurs the most communication overhead since a message is sent for every nonlocal array element.

Coarse-grained pipelining uses strip mining and loop interchange to adjust the granularity of pipelining. A message is sent for each block of nonlocal data, decreasing communication overhead at the expense of some parallelism. Selecting an efficient block size depends on the data decomposition, processor topology, and ratio of communication to computation cost for the underlying machine. A detailed algorithm is presented elsewhere [18]. Empirical results show that exploiting pipeline parallelism is important for common scientific computations such as tridiagonal solvers.

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 [33], CM FORTRAN [32], C* [29], DAPAPARALLEL C [13], DINO [30], MIMDIZER [15], PARAGON [9], and SPOT [31], but mostly builds on the following research projects.

SUPERB is a semiautomatic parallelization tool that supports arbitrary user-specified contiguous BLOCK distributions [12, 37]. 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.

Callahan and Kennedy propose distributed-memory compilation techniques based on data dependence [7]. 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, then applies numerous program transformations to optimize guards and messages.

ID NOUVEAU is a functional language enhanced with BLOCK and CYCLIC distributions [28]. Initially, send and receive statements are inserted to communicate each nonlocal array access. Message vectorization is applied to combine messages for previously written array elements. Loop jamming (fusion) and strip mining are used when writing array elements. Analysis is considerably simplified through the use of write-once arrays called I-structures. Global accumulate (reduction) operations are 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 [25]. Program analysis and optimization are different because they target a purely functional language. CRYSTAL pioneered the strategy of identifying collective communication opportunities used in the Fortran D compiler.

ASPAR is a Fortran compiler that performs simple dependence analysis using A-lists to detect parallel loops [20]. Loop iterations are then partitioned and used to automatically derive data decompositions. A micro-stencil technique suggested for the computation is used to generate a macro-stencil, identifying communication requirements. Collective communications are also generated.

KALI is the first compiler that supports both regular and irregular computations on MIMD distributed-memory machines [22]. 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.

ARF is a compiler that automatically generates inspector and executor loops for run-time preprocessing of programs with BLOCK, CYCLIC, and user-defined irregular distributions [36]. It was motivated by PARTI, 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 [26] and a hashed cache for nonlocal values [19].

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 relying on single-assignment semantics (e.g., CRYSTAL, ID NOUVEAU) or explicitly parallel programs (e.g., KALI, ARF). Precise analysis also allows the compiler to perform more optimizations. Most systems vectorize messages by extracting communication out of parallel regions, but systems such as SUPERB or Fortran D can also vectorize messages in sequential regions such as those found in SOR.

Second, the Fortran D compiler performs its analysis up front and uses the results to drive code generation, unlike transformation-based systems (e.g., Callahan and Kennedy, 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. This approach is simpler and provides greater flexibility. For instance, the Fortran D compiler may apply program transformations without the possibility of introducing deadlock due to message-reordering.

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 [4, 5, 16].

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

Acknowledgments
The authors wish to thank Vasantha Balasundaram, Geoffrey Fox, Marina Kalem, and Ulrich 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. This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center.

References
22. Kuck, D., Kuhn, R., Padua, D.,
The Time Has Come...

...to send for the latest copy of the free Consumer Information Catalog.

It lists more than 200 free or low-cost government publications on topics like money, food, jobs, children, cars, health, and federal benefits.

Don't waste another minute, send today for the latest free catalog and a free sample booklet. Send your name and address to:

Consumer Information Center
Department TH
Pueblo, Colorado 81009

A public service of this publication and the Consumer Information Center of the U.S. General Services Administration.


CR Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors) — multiple-instruction-stream, multiple-data-stream processors (MIMD), parallel processors; D.1.3 [Software]: Programming Techniques— concurrent programming; D.3.2 [Programming Languages]: Language Constructs—concurrent programming structures; D.3.4 [Programming Languages]: Processors—code generation, compilers, optimization, preprocessors

General Terms: Languages, Performance

Additional Key Words and Phrases: concurrent languages; distributed languages; distributed programming; Fortran D; parallel languages; parallel programming

About the Authors:

SEEMA HIRANANDANI is a researcher at the Center for Research on Parallel Computation at Rice University. Current research interests include compilation techniques for distributed-memory multiprocessors and runtime support for irregular computations.

KEN KENNEDY is Noah Harding Professor of computer science at the Computer Science Department at Rice University and director of the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center. Current research interests include extending techniques developed for automatic vectorization to programming tools for parallel computer systems and high-performance microprocessors.

CHAU-WEN TSENG is a doctoral candidate in the Computer Science Department at Rice University. Current research interests include parallel and distributed systems, optimizing and parallelizing compilers.

Authors' Present Address: Department of Computer Science, Rice University, Houston, TX 77251-1892; email: {seema, ken, tseng}@rice.edu

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/92/0800-066 $1.50