Compiler Support for Machine-Independent Parallel Programming in Fortran D

Seema Hiranandani
Ken Kennedy
Chau-Wen Tseng

CRPC-TR 91132
January 1991

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

In *Compilers and Runtime Software for Scalable Multiprocessors*, (J. Saltz and P. Mehrotra, eds.), North-Holland, Amsterdam, The Netherlands, 1992.

Compiler Support for Machine-Independent Parallel Programming in Fortran D *

Seema Hiranandani Ken Kennedy Chau-Wen Tseng

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

Abstract

Because of the complexity and variety of parallel architectures, an efficient machine-independent parallel programming model is needed to make parallel computing truly usable for scientific programmers. We believe that Fortran D, a version of Fortran enhanced with data decomposition specifications, can provide such a programming model. This paper presents the design of a prototype Fortran D compiler for the iPSC/860, a MIMD distributed-memory machine. Issues addressed include data decomposition analysis, guard introduction, communications generation and optimization, program transformations, and storage assignment. A test suite of scientific programs will be used to evaluate the effectiveness of both the compiler technology and programming model for the Fortran D compiler.

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, it is not likely to be a success unless parallel computers are as easy to use as the conventional vector supercomputers of today. A major component of the success of vector supercomputers is the ability to write machine-independent vector programs in a subdialect of Fortran. Advances in compiler technology, especially automatic vectorization, have made it possible for the scientist to structure Fortran loops according the rules of "vectorizable style" [Wol89, CKK89], which are well understood, and expect the resulting program to be compiled to efficient code on any vector machine.

Compare this with the current situation for parallel machines. The scientist wishing to use such a machine must rewrite the program in some dialect of Fortran with extensions that explicitly reflect the architecture of the underlying machine, such as message-passing primitives for distributed-memory machines or multidimensional vector operations for synchronous data-parallel machines. In addition to being a lot of work, this conversion is daunting because the scientist risks losing his investment when the next high-end parallel machine requires a different set of extensions.

We are left with the question: Can this problem be overcome? In other words, is it possible to identify a subdialect of Fortran from which efficient parallel programs can be generated by a new and possibly more sophisticated compiler technology? Researchers working in the area, including ourselves, have concluded that this is not possible in general. Parallel programming is a difficult task in which many tradeoffs must be weighed. In converting from a Fortran program, the compiler simply is not able to always do a good job of picking the best alternative in every tradeoff, particularly since it must work solely with the text of the program. As a result, the programmer may need to add additional information to the program for it to be correctly and efficiently parallelized.

But in accepting this conclusion, we must be careful not to give up prematurely on the goal of supporting machine-independent parallel programming. In other words, if we extend Fortran to include information about the parallelism available in a program, we should not make those extensions dependent on any particular parallel machine architecture. From the compiler technologist's perspective, we need to find a suitable language for expressing parallelism and compiler technology that will translate this language to efficient programs on different parallel machine architectures.

Parallel Programming Models

Figure 1 depicts four different machine types and the dialect of Fortran commonly used for programming on each of them: Fortran 77 for the sequential machine, Fortran 90 for the SIMD parallel machine (*e.g.,* the TMC Connection Machine), message-passing Fortran

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

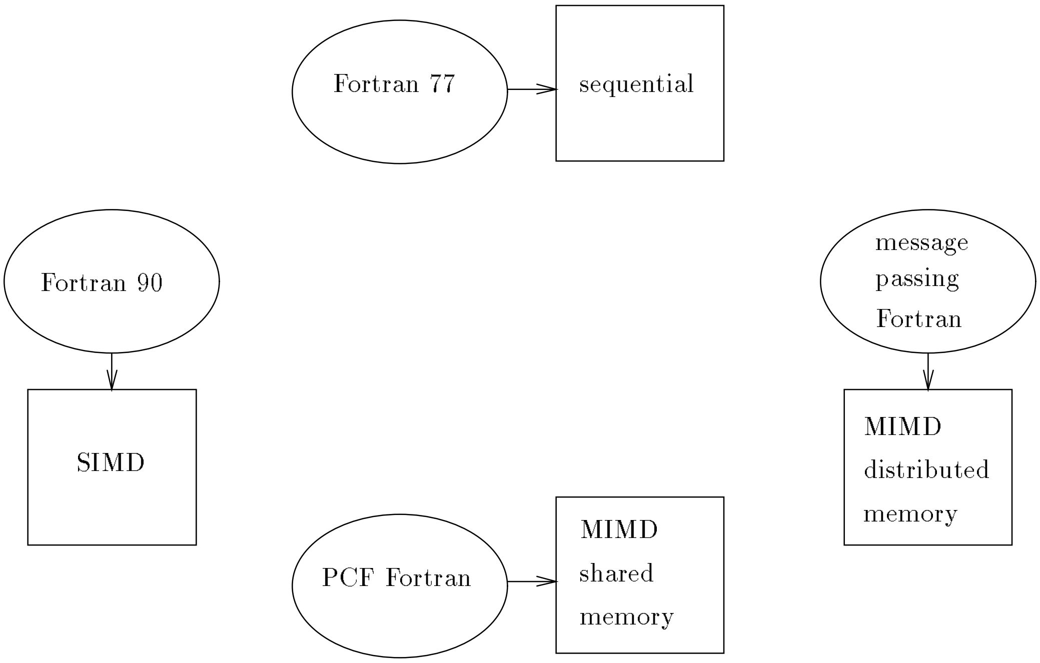

Figure 1: Fortran Dialects and Machine Architectures

for the MIMD distributed-memory machine (*e.g.,* the Intel iPSC/860) and Parallel Computer Forum (PCF) Fortran [Lea90] for the MIMD shared-memory machine (*e.g.,* the BBN TC2000 Butterfly). Each of these languages seems to be a plausible candidate for use as a machine-independent parallel programming model.

Research on automatic parallelization has already shown that Fortran is unsuitable for general parallel programming. However, message-passing Fortran looks like a promising candidate—it should be easy to implement a run-time system that simulates distributed memory on a shared-memory machine by passing messages through shared memory. Unfortunately, most scientific programmers reject this alternative because programming in message-passing Fortran is difficult and tedious. In essence, this would be reduction to the lowest common denominator: programming every machine would be equally hard.

Starting with PCF Fortran is more promising. In targeting this language to a distributed-memory machine, the key intellectual step is determining the data decomposition across the various processors. It seems plausible that we might be able to use the parallel loops in PCF Fortran to indicate which data structures should be partitioned across the processors—data arrays accessed on different iterations of a parallel loop should probably be distributed. So what is wrong with starting from PCF Fortran? The problem is that the language is nondeterministic. If the programmer inadvertently accesses the same location on different loop iterations, the result can vary for different execution schedules. Hence PCF Fortran programs will be difficult to develop and require complex debugging systems.

Fortran 90 is more promising, because it is a deterministic language. The basic strategy for compiling it to different machines is to block the multidimensional vector operations into submatrix operations, with different submatrices assigned to different processors. We believe that this approach has a good chance of success. In fact, we are participating in a project with Geoffrey Fox at Syracuse to pursue this approach. However, there are questions about the generality of this strategy. SIMD machines are not yet viewed as general-purpose parallel computers. Hence, the programs that can be effectively represented in Fortran 90 may be only a strict subset of all interesting parallel programs. We would still need some way to express those programs that are not well-suited to Fortran 90.

Machine-Independent Programming Strategy

For these reasons, we have chosen a different approach, one that introduces a new set of language extensions to Fortran. We believe that specifying the data decomposition is the most important intellectual step in developing large data-parallel scientific programs. Most parallel programming languages, however, are inadequate in this regard because they only provide constructs to express functional parallelism [PB90]. Hence, we designed a language that extends Fortran by introducing constructs that specify data decompositions. We call the extended language Fortran D, for obvious reasons. Fig-

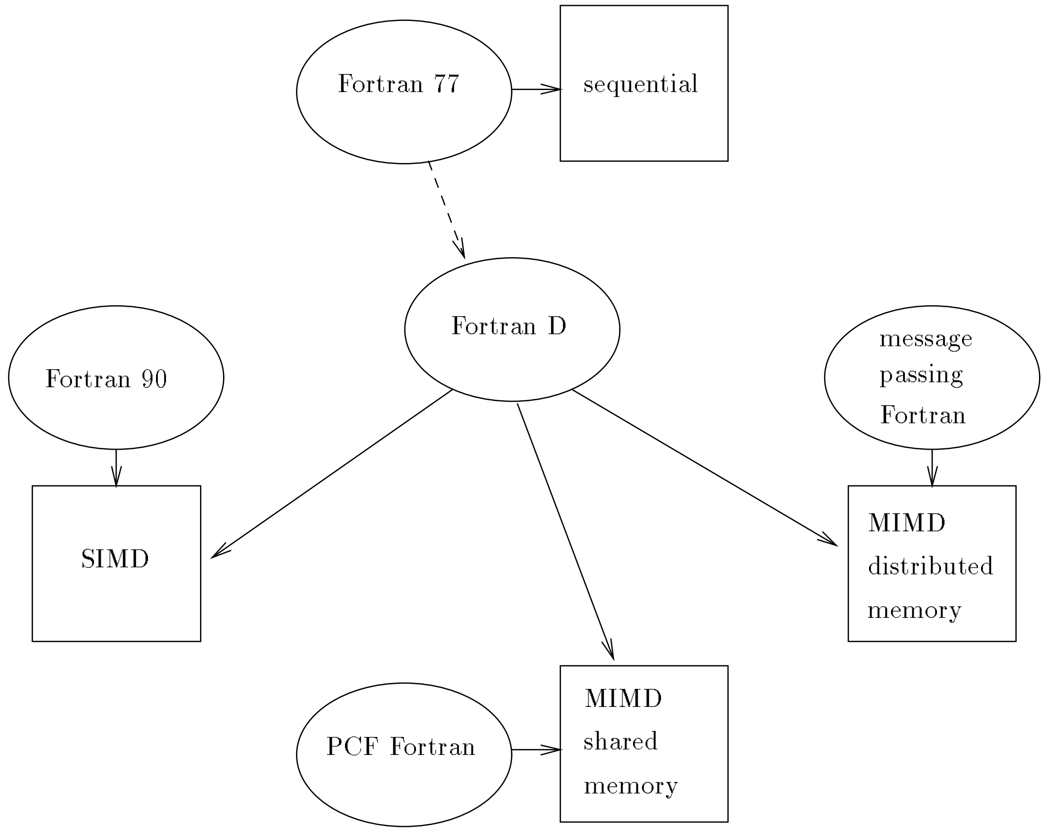

Figure 2: Machine-Independent Programming Strategy Using Fortran D

ure 2 shows our plan to use Fortran D as our machine-independent programming model. 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.

A Fortran D program is a Fortran program augmented with a set of data decomposition specifications. If these specifications are ignored the program can be run without change on a sequential machine. Hence, the meaning of the program is exactly the meaning of the Fortran program contained within it—the specifications do not affect the meaning, they simply advise the compiler. Compilers for parallel machines can use the specifications not only to decompose data structures but also to infer parallelism, based on the principle that only the owner of a datum computes its value. In other words, the data decomposition also specifies the distribution of the work in the Fortran program.

In this paper we describe a project to build and evaluate a Fortran D compiler for a MIMD distributed-memory machine, namely the Intel iPSC/860. If successful, the result of this project will go a long way toward establishing that machine-independent parallel programming is possible, since it is easy to target a MIMD shared-memory machine once we have a working system for a MIMD distributed-memory machine. The only remaining step would be the construction of an effective compiler for a SIMD machine, like the Connection Machine. Such a project is being planned and will be the subject of a future paper.

The next section presents an overview of the data decomposition features of Fortran D. Section 3 discusses the compiler strategy for a single loop nest, and Section 4 looks at compilation issues for whole programs. Section 5 describes our approach to validating this work on a collection of real application programs. In Section 6 we describe the relationship of this project to other research in the area. We conclude in Section 7 with a discussion of future work.

2 Fortran D

The data decomposition problem can be approached by noting that there are 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

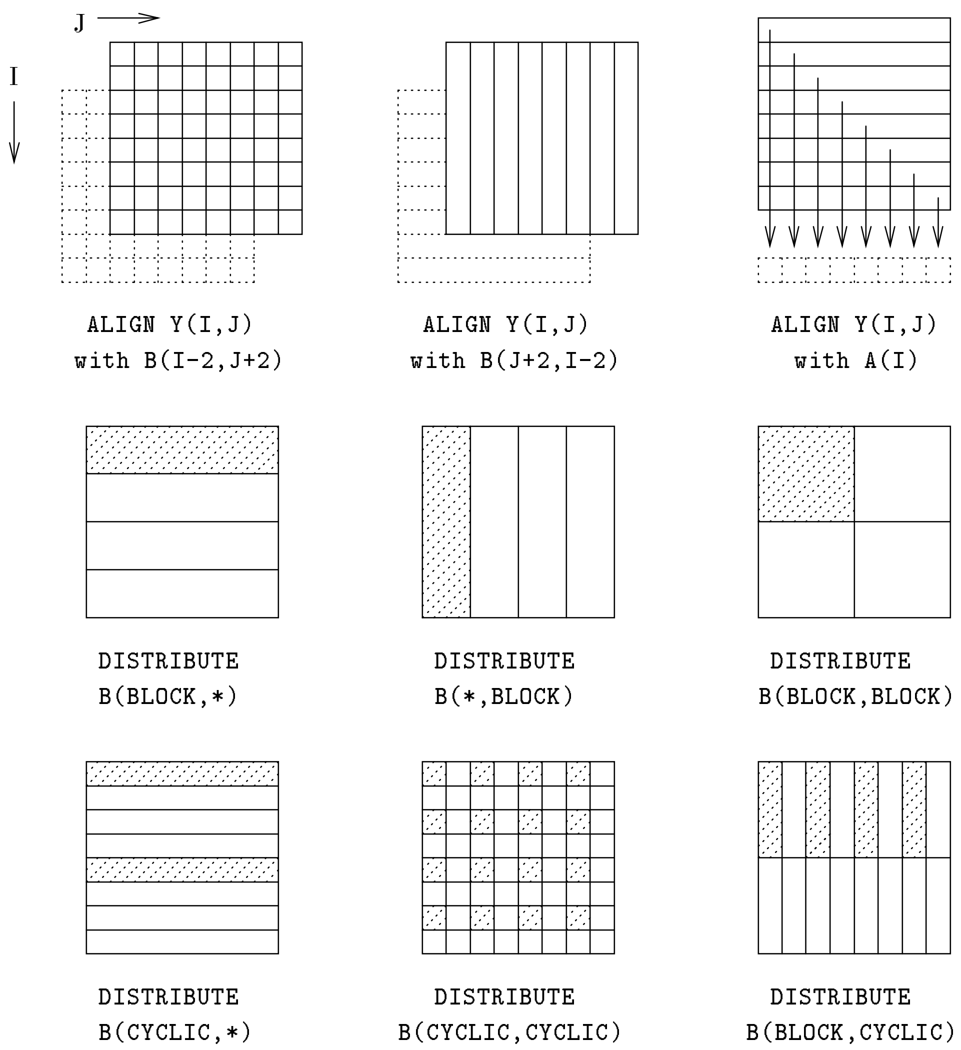

Figure 3: Fortran D Data Decomposition Specifications

of the underlying computation. It represents the minimal requirements for reducing data movement for the program, 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 dependent on the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine. Data distribution provides opportunities to reduce data movement, but must also maintain load balance. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D is a version of Fortran that provides data decomposition specifications for these two levels of parallelism. First, the `DECOMPOSITION` statement is used to declare a problem domain for each computation. The `ALIGN` statement is then used to describe a problem mapping. Finally, the `DISTRIBUTE` statement is used to map the problem and its associated arrays to the physical machine. We believe that our two phase strategy for specifying data decomposition is natural for the computational scientist, and is also conducive to modular, portable code.

2.1 Alignment and Distribution Specifications

Here we give a quick summary of some basic data decompositions supported in Fortran D. The `DECOMPOSITION` statement simply declares the name, dimensionality and size of the decomposition. The `ALIGN` statement specifies how arrays should be *aligned* with respect to one another, both within and across array dimensions. It represents the minimal requirements for reducing data movement for the program, 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

Arrays mapped to the same decomposition are automatically aligned with each other. There are two types of alignment. Intra-dimensional alignment specifies the alignment within each dimension. Inter-dimensional alignment takes place between dimensions. The alignment of arrays to decompositions is specified by placeholders in the subscript expressions of the array and decomposition. In the following example:

```
REAL X(N), Y(N,N)
DECOMPOSITION A(N), B(N,N)
ALIGN X(I) with A(I-1)
ALIGN Y(I,J) with B(J, I)
```

A and B are declared to be decompositions of size N and $N \times N$, respectively. Array X is aligned with respect to A with an offset of -1, and array Y is aligned with the transpose of B.

After arrays have been aligned with a decomposition, the `DISTRIBUTE` statement maps the decomposition to the finite resources of the physical machine. Data distribution provides opportunities to reduce data movement and load imbalance within the constraints specified by data alignment. Data distribution takes advantage of the coarse-grain parallelism, but its effectiveness is dependent on the topology, communication mechanisms, size of local memory, and number of processors in the underlying machine.

Data distribution is 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. Once a distribution is chosen for the decomposition, all the arrays aligned with the decomposition can be mapped to the machine. The following program fragment demonstrates Fortran D syntax, the data decompositions are shown in Figure 3.

```
REAL X(N), Y(N,N)
DECOMPOSITION A(N), B(N,N)
ALIGN X(I) with A(I-1)
ALIGN Y(I,J) with B(I-2,J+2)
ALIGN Y(I,J) with B(J+2,I-2)
ALIGN Y(I,J) with A(I)
DISTRIBUTE A(CYCLIC)
DISTRIBUTE B(BLOCK,*)
```

2.2 Regular and Irregular Distributions

In addition, data parallelism may either be regular or irregular. Regular data parallelism can be effectively exploited through the data decompositions shown. Irregular data parallelism, on the other hand, requires irregular data decompositions and run-time processing to manage the parallelism. In Fortran D, irregular distributions are defined by the user through an explicit data array, as shown in the example below.

```
INTEGER MAP(N)
DECOMPOSITION IRREG(N)
DISTRIBUTE IRREG(MAP)
```

In this example, the elements of MAP must contain valid processor numbers. IRREG(i) will be mapped to the processor indicated by MAP(i). MAP may be either distributed or replicated; distributed MAP arrays will consume less memory but may require extra communication to determine location. Fortran D also supports dynamic data decomposition, *i.e.,* changing the decomposition at any point in the program. The complete Fortran D language is described in detail elsewhere [FHK+90].

2.3 Distribution Functions

Distribution functions specify the mapping of an array or The `ALIGN` and `DISTRIBUTE` statements in Fortran D specify how distributed arrays are mapped to the physical machine. The Fortran D compiler uses the information contained in these statements to construct *distribution functions* that can be used to calculate the mapping of array elements to processors. Distribution functions are also created for decompositions and are used during the actual distribution of arrays onto processors.

The distribution function μ, defined below,

$$\mu_A(\vec{i}) \;=\; (\delta_A(\vec{i}), \alpha_A(\vec{i})) \;=\; (p, \vec{j})$$

is a mapping of the global index $\vec{i}$ of a decomposition or array A to a local index $\vec{j}$ for a unique processor p. Each distribution function has two component functions, δ and α. These functions are used to compute ownership and location information. For a given decomposition or array A, the owner function δ_A maps the global index $\vec{i}$ to its unique processor owner p, and the local index function α_A maps the global index $\vec{i}$ to a local index $\vec{j}$.

2.3.1 Regular Distributions

The formalism described for distribution functions are applicable for both regular and irregular distributions. An advantage of the simple regular distributions supported in Fortran D is that their corresponding distribution functions can be easily derived at compile-time. For instance, given the following regular distributions,

```
REAL X(N, 0:N-1), Y(N,N)
DECOMPOSITION A(N,N), B(N,N)
ALIGN X(I,J) with A(I, J+1)
ALIGN Y(I,J) with B(J, I)
DISTRIBUTE A(BLOCK, *);
DISTRIBUTE B(CYCLIC, *);
```

the compiler automatically derives the distribution functions in Figure 4. In the figure, the 2-D decompositions A and B are declared to have size (N, N). The

$$\mu_A^{(block,*)}(i,j) = (\lceil i/BlockSize\rceil, ((i-1) \bmod BlockSize + 1, j))$$
$$\mu_B^{(cyclic,*)}(i,j) = ((i-1) \bmod P + 1, (\lceil i/P\rceil, j))$$
$$\mu_X(i,j) = (\lceil i/BlockSize\rceil, ((i-1) \bmod BlockSize + 1, j+1))$$
$$\mu_Y(i,j) = ((j-1) \bmod P + 1, (\lceil j/P\rceil, i))$$

Figure 4: Distribution Functions

number of processors is P. For a block distribution, $BlockSize = \lceil N/P\rceil$.

2.3.2 Irregular Distributions

For an irregular distribution, we use an integer array to explicitly represent the component functions $\delta_A(\vec{i})$ and $\alpha_A(\vec{i})$. This is the most general approach possible since it can support any arbitrary irregular distribution. Unfortunately, the distribution must now be evaluated at run-time. In the following 1-D example,

```
INTEGER MAP(N), X(N)
DECOMPOSITION A(N)
ALIGN X(I) with A(I)
DISTRIBUTE A(MAP)
```

the irregular distribution for decomposition A is stored in the integer array `MAP`. The distribution functions for decomposition A and array X are then computed through run-time preprocessing techniques [SBW90, MV90].

3 Basic Compilation Strategy

In this section we provide a formal description of the general Fortran D compiler strategy. The basic approach is to convert Fortran D programs into *single-program, multiple-data* (SPMD) node programs with explicit message-passing. The two main concerns for the Fortran D compiler are 1) to ensure that data and computations are partitioned across processors, and 2) to generate communications where needed to access non-local data.

The Fortran D compiler is designed to exploit large-scale data parallelism. Our philosophy is to use the *owner computes* rule, where every processor only performs computation for data it owns [ZBG88, CK88, RP89]. However, the owner compute rule is relaxed depending on the structure of the computation. However, in this paper we concentrate on deriving a functional decomposition and communication generation by applying the owner computes rule.

We begin by examining the algorithm used to compile a simple loop nest using the owner computes rule. Correct application of the rule requires knowledge of the data decomposition for a program. As previously discussed, in Fortran D information concerning the ownership of a particular decomposition or array element is provided by the `ALIGN` and `DISTRIBUTE` statements.

3.1 Some Notation

We begin by describing some notation we will employ later in this paper.

> `DO` $\vec{k} = \vec{l}$ `to` $\vec{m}$ `by` $\vec{s}$
> $X(g(\vec{k})) = Y(h(\vec{k}))$
> `ENDDO`

In the example loop nest above, $\vec{k}$ is the set of loop iterations (also displayed as $[\vec{l} : \vec{m} : \vec{s}]$), X and Y are distributed arrays, and g and h are the array subscript functions for the left-hand side (lhs) and right-hand side (rhs) array references, respectively.

3.1.1 Image

We define the *image* of an array X on a processor p as follows:

$$image_X(p) = \{\vec{i} \mid \delta_X(\vec{i}) = p\}$$

The *image* for a processor p is constructed by finding all array indices that cause a reference to a local element of array X, as determined by the distribution functions for the array. As a result, *image* describes all the elements of array X assigned to a particular processor p. We also define t_p as *this processor*, a unique processor identification representing the local processor. Thus the expression $image_X(t_p)$ corresponds to the set of all elements of X owned locally.

3.1.2 Iteration Sets

We define the *iteration set* of a reference R for a processor p to be the set of loop iterations $\vec{j}$ that cause R to access data owned by p. Each element of the iteration set corresponds to a point in the iteration space, and is represented by a vector containing the iteration number for each loop in the loop nest.

The iteration set of a statement can be constructed in a very simple manner. Our example loop contains two references, $X(g(\vec{k}))$ and $Y(h(\vec{k}))$. The iteration set for processor p with respect to reference $X(g(\vec{k}))$ is simply $g^{-1}(image_X(p))$, the inverse subscript function g^{-1} applied to the image of the array X on processor p. Similarly, the iteration set with respect to reference $Y(h(\vec{k}))$ can be calculated as $h^{-1}(image_Y(p))$.

This property will be used in several algorithms later in the paper. In particular, notice that when using the owner computes rule, the iteration set of the lhs of an assignment statement for processor p is exactly the iterations in which that statement must be executed on p. For example, in the simple loop above, the function $g^{-1}(image_X(t_p))$ may be used to determine when t_p, the local processor, should execute the statement.

3.2 Guard Introduction

The guard introduction phase of the compiler ensures that computations in a program are divided correctly among the processors according to the owner computes rule. This may be accomplished by a combination of reducing loop bounds and guarding individual statements. Both approaches are based on calculating *iteration sets* for statements in a loop.

3.2.1 Loop Bounds Reduction

Since evaluating guards at run-time increases execution cost, the Fortran D compiler strategy is to reduce loop bounds where possible for each processor to avoid evaluating guard expressions. Figure 5 presents a straightforward algorithm for performing simple loop bounds reduction. The algorithm works as follows. First, the iteration sets of all the *lhs* are calculated for the local processor t_p. These sets are then unioned together. The result represents all the iterations on which a assignment will need to be executed by the processor. The loop bounds are then reduced to the resulting iteration set.

3.2.2 Mask Generation

In the case where all assignment statements have the same iteration set, loop bounds reduction will eliminate any need for masks since all statements within the reduced loop bounds always execute. However, loop bound reduction will not work in all cases. For instance, loop nests may contain multiple assignment statements to distributed data structures. The iteration set of each statement for a processor may differ, limiting the number of guards eliminated through bounds reduction. The compiler will need to introduce masks for the statements that are conditionally executed.

Figure 6 presents a simple algorithm to generate masks for statements in a loop nest. Each statement is examined in turn and its iteration set calculated. If it is equivalent to the iteration set of the previous statement, then the two statements may be guarded by the same mask. Otherwise, any previous masks must be terminated and a new mask created. We assume the existence of functions to generate the appropriate guard/mask for each statement based on its iteration set.

3.3 Communication Generation

Once guards have been introduced, the Fortran D compiler must generate communications to preserve the semantics of the original program. This can be accomplished by calculating SEND and RECEIVE iteration sets. For simple loop nests which do not contain *loop-carried* (inter-iteration) true dependences [AK87], These iteration sets may also be used to generate IN and OUT array index sets that combine messages to a single processor into one message. We describe the formation and use of these sets in more detail in the following sections.

3.3.1 Regular Computations

LOCAL, SEND, and RECEIVE Iteration Sets

We describe as *regular computations* those computations which can be accurately characterized at compile-time. In these cases the compiler can exactly calculate all communications and synchronization required without any run-time information. The first step is to calculate the following iteration sets for each reference R in the loop with respect to the local processor t_p:

- LOCAL – Set of iterations in which R results in an access to data local to t_p.
- SEND – Set of iterations in which R results in an access to data local to t_p, but the statement containing R is executed on a different processor.
- RECEIVE – Set of iterations in which the statement containing R is executed on t_p, but R results in an access to data not local to t_p.

The LOCAL, SEND, and RECEIVE iteration sets can be generated using the owner computes rule. Figure 7 shows the algorithm for regular computations. It starts by first calculating the iteration set for the *lhs* of each assignment statement with respect to the local processor t_p; this determines the LOCAL iteration set.

The iteration sets for each *rhs* of the statement are then constructed with respect to the t_p. Any element of the LOCAL iteration set that does not also belong to the iteration set for the *rhs* will need to access nonlocal data; it is put in the RECEIVE iteration set. Conversely, any elements in the iteration set for the *rhs* not also in the LOCAL iteration are needed by some other processor; it is put into the SEND iteration set. These iteration sets complete specify all communications that must be performed.

IN and OUT Index Sets

For loop nests which do not contain loop-carried true dependences, communications may be moved entirely outside of the loop nest and blocked together. In addition, messages to the same processor may also be combined to form a single message. These steps are desirable when communication costs are high, as is the case for most MIMD distributed-memory machines. The following array index sets are utilized for these optimizations:

- IN – Set of array indices that correspond to nonlocal data accesses. These data elements must be received from other processors in order to perform local computations.

FOR each loop nest $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$ DO
  $reduced\_iter\_set = \emptyset$
  FOR each statement$_i$ in loop with $lhs = X_i(g_i(\vec{k}))$
    $iter\_set = g_i^{-1}(image_{X_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
    $reduced\_iter\_set = reduced\_iter\_set \cup iter\_set$
  ENDFOR
  reduce bounds of loop nest to those in $reduced\_iter\_set$
ENDFOR

Figure 5: Reducing Loop Bounds Using Iteration Sets

FOR each loop nest $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$ DO
  $previous\_iter\_set = [\vec{l} : \vec{m} : \vec{s}]$
  FOR each statement$_i$ in order DO
    IF statement$_i$ = assignment AND lhs = global array $X_i(g_i(\vec{k}))$ THEN
      $iter\_set = g_i^{-1}(image_{X_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
    ELSE
      $iter\_set = [\vec{l} : \vec{m} : \vec{s}]$
    ENDIF
    IF $iter\_set = previous\_iter\_set$ THEN
      insert statement$_i$ after statement$_{i-1}$
    ELSE
      terminate previous mask if it exists
      create new mask for $iter\_set$ and insert statement$_i$ inside mask
      $previous\_iter\_set = iter\_set$
    ENDIF
  ENDFOR
ENDFOR

Figure 6: Generating Statement Masks Using Iteration Sets

FOR each statement$_i$ with $lhs = X_i(g_i(\vec{k}))$ in loop nest $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$ DO
  $local\_iter\_set_{X_i}^{t_p} = g_i^{-1}(image_{X_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
  FOR each rhs reference to a distributed array $Y_i(h_i(\vec{k}))$ DO
    $local\_iter\_set_{Y_i}^{t_p} = h_i^{-1}(image_{Y_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
    $receive\_iter\_set_{Y_i}^{t_p} = local\_iter\_set_{X_i}^{t_p} - local\_iter\_set_{Y_i}^{t_p}$
    $send\_iter\_set_{Y_i}^{t_p} = local\_iter\_set_{Y_i}^{t_p} - local\_iter\_set_{X_i}^{t_p}$
  ENDFOR
ENDFOR

Figure 7: Generating SEND/RECEIVE Iteration Sets (for Regular Computations)

```
FOR each statement_i with lhs = X_i(g_i(k⃗)) in loop nest k⃗ = l⃗ to m⃗ by s⃗ DO
  FOR each rhs reference to a distributed array Y_i(h_i(k⃗)) DO
    {* initialize In and Out index sets *}
    FOR proc = 1 to numprocs DO
      in_index_set_{Y_i}^{(t_p,proc)} = ∅
      out_index_set_{Y_i}^{(t_p,proc)} = ∅
    ENDFOR
    {* compute Out index sets *}
    FOR each j⃗ ∈ send_iter_set_{Y_i}^{t_p} DO
      send_p = δ_{X_i}(g_i(h_i^{-1}(μ_{Y_i}^{-1}(t_p, α_{Y_i}(h_i(j⃗))))))
      out_index_set_{Y_i}^{(t_p,send_p)} = out_index_set_{Y_i}^{(t_p,send_p)} ∪ {α_{Y_i}(h_i(j⃗))}
    ENDFOR
    {* compute In index sets *}
    FOR each j⃗ ∈ receive_iter_set_{Y_i}^{t_p} DO
      recv_p = δ_{Y_i}(h_i(j⃗))
      in_index_set_{Y_i}^{(t_p,recv_p)} = in_index_set_{Y_i}^{(t_p,recv_p)} ∪ {α_{Y_i}(h_i(j⃗))}
    ENDFOR
  ENDFOR
ENDFOR
```

Figure 8: Generating In/Out Index Sets (for Regular Computations)

- Out – Set of array indices that correspond to local data accessed by other processors. These data elements must be sent to other processors in order to permit them to perform their computations.

The calculation of In and Out index set for regular computations is depicted in Figure 8. The algorithm works as follows. Each element in the Send and Receive iteration sets is examined. Some combination of the subscript, mapping, alignment, and distribution functions and their inverses are applied to the element to determine the source or recipient of each message. The message to that processor is then stored in the appropriate In or Out index set, effectively blocking it with all other messages to the same processor.

More complicated algorithms are needed for loops with loop-carried dependences, since not all communication can be moved outside of the entire loop nest. To handle loop-carried dependences, In and Out index sets need to be constructed at each loop level. Dependence information may be used to calculate the appropriate loop level for each message, using the algorithms described by Balasundaram et al. and Gerndt [BFKK90, Ger90]. Messages in Send and Receive sets can then be inserted in the In or Out set at that loop level.

3.3.2 Irregular Computations

Irregular computations are computations that cannot be accurately characterized at compile-time.[1] It is not possible to determine the Send, Receive, In, and Out sets at compile-time for these computations. However, an *inspector* [SMC89, KMSB90] may be constructed to preprocess the loop body at run-time to determine what nonlocal data will be accessed. This in effect calculates the In index set for each processor. A global transpose operation between processors can then be used to calculate the Out index sets as well.

An inspector is the most general way to generate In and Out sets for loops without loop-carried dependences. Despite the expense of additional communications, experimental evidence from several systems [KMV90, WSBH91] proves that it can improve performance by blocking together communications to access nonlocal data outside of the loop nest. In addition it also allows multiple messages to the same processor to be blocked together. The Fortran D compiler plans to automatically generate inspectors where needed for irregular computations.

The structure of an inspector loop is shown in Figure 9. For compatibility with our treatment of regular

[1] Irregular computations are different from irregular distributions, which are irregular mappings of data to processors.

FOR each statement$_i$ with lhs = $X_i(g_i(\vec{k}))$ in loop nest $\vec{k}$ = $\vec{l}$ to $\vec{m}$ by $\vec{s}$ DO
    $local\_iter\_set^{t_p}_{X_i} = g_i^{-1}(image_{X_i}(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
    $receive\_iter\_set^{t_p}_{Y_i} = \emptyset$
    FOR each rhs reference to a distributed array $Y_i(h_i(\vec{k}))$ DO
      {∗ calculate In index sets for this processor ∗}
      FOR each $\vec{j} \in local\_iter\_set^{X_i}_{t_p}$ DO
        IF $(\delta_{Y_i}(h_i(\vec{j})) \neq t_p)$ THEN
          $receive\_iter\_set^{t_p}_{Y_i} = receive\_iter\_set^{t_p}_{Y_i} \cup \{\vec{j}\}$
          $recv_p = \delta_{B_i}(h_i(\vec{j}))$
          $in\_index\_set^{(t_p, recv_p)}_{Y_i} = in\_index\_set^{(t_p, recv_p)}_{Y_i} \cup \{\alpha_{Y_i}(h_i(\vec{j}))\}$
        ENDIF
      ENDFOR
      {∗ send In index sets to all other processors ∗}
      FOR $recv_p = 1$ to $numprocs$ DO
        IF $(recv_p \neq t_p)$ THEN
          $send(in\_index\_set^{(t_p, recv_p)}_{Y_i},\ recv_p)$
        ENDIF
      ENDFOR
      {∗ receive In index sets, convert into Out index sets ∗}
      FOR $send_p = 1$ to $numprocs$ DO
        IF $(send_p \neq t_p)$ THEN
          $receive(out\_index\_set^{(t_p, send_p)}_{Y_i},\ send_p)$
        ENDIF
      ENDFOR
    ENDFOR
ENDFOR

Figure 9: Inspector to Generate In/Out Index Sets (for Irregular Computations)

```
{* original loop to be transformed into send, receive, and compute loops *}
DO k⃗ = l⃗ to m⃗ by s⃗
   X(g(k⃗)) = Y(h(k⃗))
ENDDO
{* send loop *}
FOR send_p = 1 to numprocs DO
   IF (out_index_set_Y^(t_p,send_p) ≠ ∅) THEN
     buffer_values(out_value_set_Y^(t_p,send_p), out_index_set_Y^(t_p,send_p))
     send(out_value_set_Y^(t_p,send_p), send_p)
   ENDIF
ENDFOR
{* local compute loop *}
FOR each j⃗ ∈ {local_iter_set_X^t_p − receive_iter_set_Y^t_p} DO
   X(α_A(g(j⃗))) = Y(α_B(h(j⃗)))
ENDFOR
{* receive loop *}
FOR recv_p = 1 to numprocs DO
   IF (in_index_set_Y^(t_p,recv_p) ≠ ∅) THEN
     receive(in_value_set_Y^(t_p,recv_p), recv_p)
     store_values(in_value_set_Y^(t_p,recv_p), in_index_set_Y^(t_p,recv_p))
   ENDIF
ENDFOR
{* nonlocal compute loop *}
FOR each j⃗ ∈ receive_iter_set_Y^t_p DO
   X(α_X(g(j⃗)))) = get_value(Y(h(j⃗)))
ENDFOR
```

Figure 10: Send, Receive, and Compute Loops Resulting from IN/OUT Index Sets

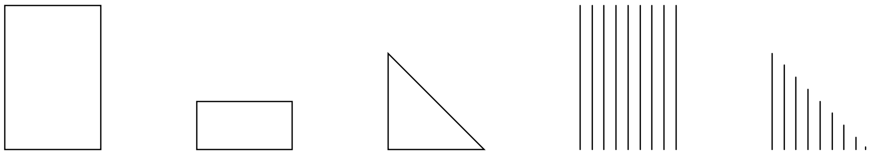

Figure 11: Regular Section Descriptors (RSDs)

computations, the Fortran D inspector also generates the LOCAL and RECEIVE iteration sets. In the first part of the inspector, the LOCAL iteration set is calculated for each statement based on the *lhs*. The *rhs* is examined for each element in the LOCAL iteration set. Any nonlocal references cause the iteration to be added to the RECEIVE iteration set. The owner and local index of the nonlocal reference are then calculated and added to the IN index set.

After the local IN sets have been calculated, a global transpose is performed in the remainder of the inspector. Each processor sends its IN index set for a given processor to that processor. Upon receipt, they become OUT index sets for the receiving processor.

3.3.3 Resulting Program

Once the SEND and RECEIVE sets have been calculated, the example loop nest is transformed into the loops pictured in Figure 10 [KMSB90]. In the *send* loop, every processor sends data they own to processors that need the data. The OUT index set for *rhs* of the statement in the example loop has already been calculated. However, the function *buffer_values()* must be used to actually collect the values at each index and the OUT set. The resulting values are then sent to the appropriate processor.

Next, in the *local compute* loop, loop iterations that assign and use only local data may be executed. These are elements that are in the LOCAL but not RECEIVE iteration sets. These iterations are executed immediately following the send loop to take advantage of communication latency.

In the *receive* loop, every processor receives nonlocal data sent from their owners in the send loop. The values received are mapped to their designated storage locations using the function *store_values()*. The indices corresponding to these values have already been calculated and stored in the IN index sets. Finally, in the *nonlocal compute* loop every processor performs computations for loop iterations that also require nonlocal data. The function *get_value()* is used to fetch nonlocal data from their designated storage locations.

3.4 Regular Sections

For the sake of efficiency, when generating communications the Fortran D compiler constructs approximations of the *image* for each distributed array using *regular section* or *data access descriptors* [CK87, BK89, HK90]. A regular section descriptor (RSD) is a compact representation of rectangular or right-triangular array sections and their higher dimension analogs. They may also possess some constant step. The union and intersection of RSDs can be calculated inexpensively, making them highly useful for the Fortran D compiler. RSDs have also proven to be quite precise in practice, due to the regular computation patterns exhibited by scientific programs [HK90]. Figure 11 shows some examples of regular section descriptors.

4 Compilation of Whole Programs

We have shown how the Fortran D compiler introduces guards and generates communications for a simple loop nest. When compiling whole programs containing multiple loop nests or procedures, the compiler faces much more complex problems, as well as many opportunities for optimization. Here we present a brief overview of the issues facing the Fortran D compiler.

4.1 Data Decomposition Analysis

Researchers have found that the computation structure may often change between different phases of a program. Relying on a single static data decomposition for these programs will result in excessive data movement [KLS90, KN90]. To overcome this problem, Fortran D permits data decomposition specifications to be inserted at any point in a program, providing dynamic data decompositions. However, to support a modular programming style, we restrict the scope of dynamically declared data decompositions to that of the current procedure. When a procedure returns, any dynamic data decompositions declared locally are restored to the values they possessed before the procedure was called.

Dynamic data decompositions complicate the job of the Fortran D compiler, since it needs to know how an array is decomposed in order to generate the proper guards and communications. We define the *reaching* data decompositions at a point P in the program to be the set of data decompositions that have a path to P not *killed* by another decomposition statement. In order to generate the correct code for each reference to a distributed array, the Fortran D compiler must perform global dataflow analysis to solve the reaching data decomposition problem. When there are multiple reaching decompositions, the compiler must either insert run-time routines to handle each decomposition, or apply node splitting techniques to allow compile-time resolution.

4.2 Program Transformations

One of the features of the Fortran D compiler is that it utilizes the results of dataflow and dependence analysis to apply program transformations that eliminate guards or improve communications. Transformations must be performed after decomposition analysis, since their profitability depend on the data decomposition present. The following sections show examples of a few transformations. We are investigating the usefulness of other transformations such as loop skewing and peeling. In the following examples, for the sake of clarity we ignore guards and assume that all arrays are identically aligned and have `BLOCK` distributions.

4.2.1 Loop Interchanging

Loop interchange can be used to move dependences to outer loops, enabling block communications.

```
{* dependence on loop j *}
do i = 1, n
   do j = 1, n
     perform_send(X(i,j-1))
     perform_recv(X(i,j-1))
     X(i,j) = X(i,j-1)
   enddo
enddo
{* after loop interchange *}
do j = 1, n
   perform_send(X(1:n,j-1))
   perform_recv(X(1:n,j-1))
   do i = 1, n
     X(i,j) = X(i,j-1)
   enddo
enddo
```

4.2.2 Strip Mining

Strip mine can be used to simplify guard introduction, reduce size of messages, and improve load balance.

```
{* original message too large *}
perform_send(X(1:n))
perform_recv(X(1:n))
do i = 1, n
   ...  = X(i)
enddo
{* strip mine reduces message size *}
do i = 1, n, strip
   perform_send(X(i:i+strip-1))
   perform_recv(X(i:i+strip-1))
   do i$ = i, i+strip-1
     ...  = X(i$)
   enddo
enddo
```

4.2.3 Loop Distribution

Loop distribution can be used to simplify guard introduction and enable other transformations such as loop interchange.

```
{* dependences on both loops i and j *}
do i = 1, n
   perform_send(X(i-1,1:n))
   perform_recv(X(i-1,1:n))
   do j = 1, n
     perform_send(Y(i,j-1))
     perform_recv(Y(i,j-1))
     X(i,j) = X(i-1,j)
     Y(i,j) = Y(i,j-1)
   enddo
enddo
{* after distribution & interchange *}
do i = 1, n
   perform_send(X(i-1,1:n))
   perform_recv(X(i-1,1:n))
   do j = 1, n
     X(i,j) = X(i-1,j)
   enddo
enddo
do j = 1, n
   perform_send(Y(1:n,j-1))
   perform_recv(Y(1:n,j-1))
   do i = 1, n
     Y(i,j) = Y(i,j-1)
   enddo
enddo
```

4.2.4 Align

Loop alignment [ACK87] can improve guard introduction.

```
{* statements require masks *}
do i = 1, n
   X(i) = i
   Y(i+1) = i
enddo
{* alignment eliminates masks in loop *}
X(1) = i
do i = 2, n
   X(i) = i
   Y(i) = i-1
enddo
Y(n+1) = n
```

4.3 Communications Optimization

A major goal of the Fortran D compiler is to aggressively optimize communications. We describe some techniques we will attempt in order to eliminate or combine messages.

4.3.1 Vectorize Messages

Generating communications for loops containing loop-carried true dependences is complex. A simple solution is to insert all communications at the deepest loop nesting level. directly preceding each reference to a nonlocal memory reference. Unfortunately, this approach generates large numbers of small messages that may prove inefficient because of high communications overhead and latency. Algorithms developed by Balasundaram *et al.* and Gerndt employ data dependence information to insert communications at the outermost loop allowable, without violating dependences [BFKK90, Ger90]. This enables messages to be vectorized, as in the following example:

```
{* dependence on loop j *}
do i = 1, n
   do j = 1, n
     perform_send(X(i,j-1))
     perform_recv(X(i,j-1))
     X(i,j) = X(i,j-1)
   enddo
enddo
{* dependence on loop i *}
do i = 1, n
   perform_send(X(i-1,1:n))
   perform_recv(X(i-1,1:n))
   do j = 1, n
     X(i,j) = X(i-1,j)
   enddo
enddo
```

Vectorizing messages are desirable because they combine many small messages into one large message, reducing message overhead. The Fortran D compiler will use the algorithm *comm* from [BFKK90] for determining the loop level for inserting communications.

Message vectorization is a special case of *prefetching* data; *i.e.,* fetching nonlocal data before it is used in a computation. More general data prefetching optimizations are possible. Like *instruction scheduling* or *software prefetching*, the goal of data prefetching is to reduce apparent latency by performing useful computation while waiting for expensive memory accesses. The KALI compiler, developed by Koelbel *et al.*, utilizes this strategy for individual parallel loops by computing loop iterations that access only local data while waiting for data from other processors [KMV90].

4.3.2 Utilize Collective Communications

Li and Chen showed that the compiler can take advantage of the highly regular communication patterns displayed by many computations [LC90a]. Rather than generating a large number of individual send and receive communication primitives, the compiler can instead take advantage of efficient collective communications libraries such as EXPRESS [EXP89], CRYSTAL_ROUTER [FJL+86], CRYSTAL communications [CCL89], and PARTI [SBW90]. The compiler will exploit these routines to reduce the cost of communications. The guiding principles are:

- Apply program analysis to identify communications patterns
- Utilize collective communications routines where profitable, even if *overcommunication*) result; *e.g.*, extra data/messages
- Recognize and replace reductions; *e.g.*, sum reductions

4.3.3 Global Dataflow Analysis

Communications may be optimized further by considering interactions between all the loop nests in the procedure. Global dataflow analysis 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. For instance, assume that messages in previous loop nests have already retrieved nonlocal elements for a given array. If those values are *live*, messages in succeeding loop nests may be eliminated or reduced. The precision of such analyses can be improved by using regular section descriptors to analyze liveness for array sections.

4.3.4 Combine or Eliminate Messages

The algorithms for generating communications described in Section 3 consider each statement individually. When compiling loop nests containing multiple statements, communications may be optimized by combining or eliminating messages based on other messages in the loop nest. The Fortran D compiler will examine the messages generated for each array at each loop level, starting at the most deeply nested loop. The compiler needs to determine whether any of the messages may be:

- Subsumed by other messages at that loop level
- Combined with other messages at that loop level
- Subsumed or reduced by messages at inner loops

In addition, if a processor is sending several messages for different arrays to the same processor, they may be combined into the same message.

4.3.5 Relax Owner Computes Rule

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 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. Consider the following loop:

```
do i = 1, n
   A(i) = B(i) + C(i)
enddo
```

Assume that `B(i)` and `C(i)` are mapped together to a processor different from the owner of `A(i)`. The amount of data communicated may be reduced by half if the computation is first performed by the processor owning `B` and `C`, then sent to the processor owning `A`. This optimization is a simple application of the "owner stores" rule proposed by Balasundaram [Bal91].

In particular, 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 and Arf [KMV90, WSBH91]. This technique may improve communication and provides greater control over load balance, especially for irregular computations. It also eliminates the need for individual statement masks and simplifies handling of control flow within the loop body. The Fortran D compiler will detect phases of the computation where the owner computes rule may be relaxed to improve communications or load balance.

4.3.6 Replicate Computation

The Fortran D compiler considers scalar variables to be replicated. All processors thus perform computations involving assignments to scalar variables. This causes redundant computation to be performed, but is profitable because it significantly reduces communication costs. A similar approach may be taken for computations on elements of distributed arrays. It may be more efficient to replicate computation on multiple processors, rather than incur the expense of communicating the value from the owner of that element. Consider the following loop:

```
do i = 3, n
   X(i) = f(i)
   Y(i) = (X(i-1) + X(i-2)) / 2
enddo
```

Assume that arrays `X` and `Y` are distributed arrays aligned identically onto the same decomposition, and that f is a function performing computation that does not require values from other processors. Straightforward compilation of this loop would cause messages to be generated, communicating the new values of `X(i-1)` and `X(i-2)` to the processor performing the assignment to `Y(i)`. However, if the Fortran D compiler replicates the computation of `X(i-1)` and `X(i-2)` on the receiving processor, it eliminates the need for any communications.

4.3.7 Eliminate Dead Computation

A side effect of replicating all scalar variables is that naive compilation frequently results in computations that generate processor-specific dead values, *i.e.,* values that are not used by the local processor. In these cases an obvious optimization is to not compute the scalar value. For instance, consider the following loop:

```
do i = 1, n
   sum = Y(i-1) + Y(i+1)
   X(i) = sum / 2
enddo
```

The Fortran D compiler must determine that the scalar variable `sum` is only used locally by the assignment to `X(i)` and guard it appropriately. Otherwise it will waste computation by calculating `sum` on all processors for all loop iterations. Worse yet, the compiler may generate communications to fetch nonlocal values of `Y`, adding significant communication costs!

4.3.8 Block Messages

The goal of many communication optimizations is reduce communications overhead by combining small messages together to form larger messages. However, the Fortran D compiler needs to be careful since physical machines have local memory and message buffering limits. Excessively large or long-lived messages may cause memory and buffer overflows that will abort or deadlock the program.

If information concerning the limits of the system can be fed to the compiler, it may block large messages by strip mining the loops containing communications. Messages may then be broken up and moved to the strip mined loop. The compiler would need to calculate a blocking factor that remains within the physical limits of the underlying parallel machine. Blocking is also useful as a means of compromising between communications and load balance.

4.4 Storage Management

Once guards and communications have been calculated, the Fortran D compiler must still select and manage storage for all nonlocal array references. The simplest approach is to allocate full-sized arrays on each processor. This requires the least change to the program, but may waste tremendous amounts of memory. More sophisticated storage management techniques manipulate both the location and lifetimes of nonlocal storage in order to reduce memory use and code complexity. Storage management may be separated into two phases, selection of the desired storage types, and coordinating their usage.

4.4.1 Determine Storage of Nonlocal Data

First, analysis must be performed to choose a storage type for nonlocal access of each array. There are several different storage types, described below:

- *Overlaps* are expansion of local array sections to accommodate neighboring nonlocal elements [Ger90]. Overlaps are useful for regular computations because they allow the generation of clear and readable code. However, for certain computations storage may be wasted because all array elements between the local section and the one accessed must also be part of the overlap. Storage is also wasted because overlaps are assigned to individual arrays, and cannot be reused for other arrays later in the program.

- *Persistent buffers* are designed to overcome the contiguous nature of overlaps. They are useful when the size of the nonlocal data is fixed, but is not necessarily in a neighboring area. For instance, a persistent buffer can be used to store the pivot for Gaussian elimination.

- *Temporary buffers* are used for nonlocal data with short live ranges. They may be reused after the loop, or even within the loop.

- *Hash tables* are mainly used to store nonlocal data for irregular distributions. They provide a nonlocal value cache that allows quick lookup for nonlocal values [MSMB90].

4.4.2 Maintain Overlap Areas, Temporaries, and Hash Tables

Once the type of storage is chosen, the compiler needs to perform analysis to determine the total amount of storage needed as overlaps, persistent buffers, or temporary buffers. It also needs to change all nonlocal array references assigned to buffers so that they access the appropriate buffer instead.

4.5 Interprocedural Issues

The presence of procedures adds significantly to the complexity of the compilation process, especially for data decomposition analysis. Just as within a procedure, we need to calculate which data decompositions reach every reference to a distributed array. If multiple decompositions reach a procedure, either run-time routines or interprocedural node splitting techniques such as *cloning* or *inlining* may be required to handle the code. Since Fortran D semantics limit the scope of all decompositions to the procedure in which they are declared, this can be solved as a forward interprocedural dataflow problem, using the techniques developed for the ParaScope environment [CKT86].

Unfortunately, the same semantics also require the compiler to insert calls to run-time data decomposition routines to restore the original data decomposition upon procedure return. Since dynamic data decomposition is an expensive operation, these calls should be eliminated where possible. We define a data decomposition specification to be *live* if it has a *reachable use*; *i.e.,* if there is a path from the specification to a reference to a distributed array that is not killed by a different specification. The live data decomposition problem may be solved as a backward interprocedural problem. Many compiler inserted run-time data decompositions can then be determined to be not *live*, and may be safely eliminated. It may also be possible to hoist dynamic data decompositions out of loops, dramatically improving the performance of loops containing data decomposition statements.

4.6 Compiler Architecture

In review, the primary goal of the Fortran D compiler is to produce a correct distributed-memory program based on the *owner computes* rule. To do this, the compiler introduces guards and generates communications to access nonlocal data. Information from dataflow, dependence, and interprocedural analysis is applied to enhance each phase of compilation. Another important goal of the compiler is to optimize communications and minimize load imbalance. The basic architecture of the compiler is shown below:

1. Data Decomposition Analysis
   (a) Determine reaching decompositions
   (b) Determine live decompositions
   (c) Optimizing decompositions
2. Program Transformations
   (a) Loop interchanging
   (b) Strip mining
   (c) Loop distribution
   (d) Align
3. Guard Introduction
   (a) Calculate SEND & RECEIVE iteration sets
   (b) Loop bounds reduction
   (c) Mask generation
4. Communication generation
   (a) Calculate SEND & RECEIVE index sets
   (b) Calculate IN & OUT index sets
   (c) Generate inspector/executors
5. Communication optimization
   (a) Vectorize messages
   (b) Utilize collective communications
   (c) Global dataflow analysis
   (d) Combine/eliminate messages
   (e) Relax owner computes rule
   (f) Replicate computation
   (g) Eliminate dead computation
   (h) Block messages
6. Storage Assignment
   (a) Determine storage of nonlocal data
   (b) Maintain overlap areas, temporaries, and hash tables

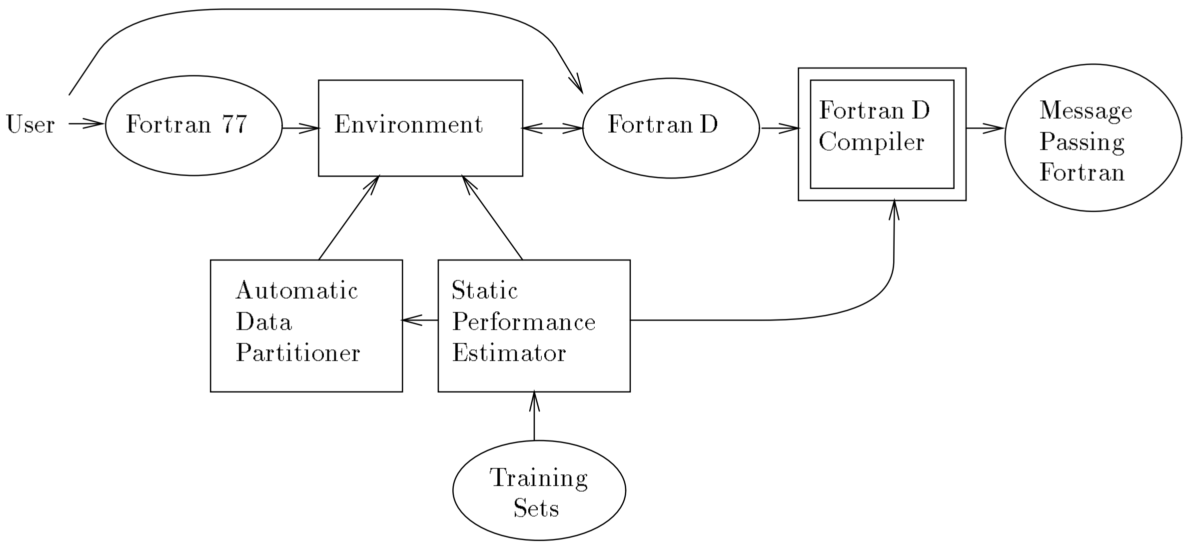

Figure 12: Fortran D Programming Environment

4.7 Fortran D Programming Environment

The compiler is a key element of the Fortran D programming system being developed at Rice University [HKK+91]. The structure of the programming system is shown in Figure 12. It is being developed in the context of the ParaScope parallel programming environment [CCH+88], and will take advantage of its analysis and transformation abilities [CKT86, KMT91]. Another part of the Fortran D system is a static performance estimator that will take as input a Fortran D or message-passing Fortran program and predicts its performance on the target machine [BFKK90, BFKK91]. The performance estimation is based on *training sets*, programs containing kernel computation and communications that can be executed on the target machine. The automatic data partitioner is the final component of the Fortran D programming system. It will utilize both the performance estimator and Fortran D compiler to automatically select and predict the performance of different data decompositions on the target machine.

5 Validation

We plan to establish whether our compilation scheme for Fortran D can achieve acceptable performance on the iPSC/860, a representative MIMD distributed-memory machine. We will use a benchmark suite currently being developed by Geoffrey Fox at Syracuse. This suite will consist of a collection of Fortran programs. Each program in the suite will have five versions:

(v1) the original Fortran 77 program,

(v2) the best hand-coded message-passing version of the Fortran program,

(v3) a "nearby" Fortran 77 program,

(v4) a Fortran D version of the nearby program, and

(v5) a Fortran 90 version of the program.

The "nearby" version of the program will utilize the same basic algorithm as the message-passing program, except that all explicit message-passing and blocking of loops in the program are removed. The Fortran D version of the program consists of the nearby version plus appropriate data decomposition specifications. The purpose of the program suite is to provide a fair test of the prototype compiler that does not depend on high-level algorithm changes, but does exercise its ability to optimize whole programs based on the structure of the computation and machine-dependent issues such as the number and speed of processors in the parallel machine.

Our validation strategy is depicted in Figure 13. We will compare the running time of the best hand-coded message-passing version of the program (v2) with the output of the Fortran D compiler for the Fortran D version of the nearby program (v4) We will view the project as successful if the Fortran D version is within a factor of two for approximately 75% of the programs in the validation suite. In effect we will be testing the limits of our machine-independent Fortran D programming model, as well as the efficiency and capabilities of our compiler technology. Future experiments will also compare the Fortran 90 compiler and programs for SIMD machines.

6 Relationship to Other Research

6.1 Programming Models and Languages

The proliferation of parallel architectures has focused much attention on machine-independent parallel programming. Some researchers have proposed

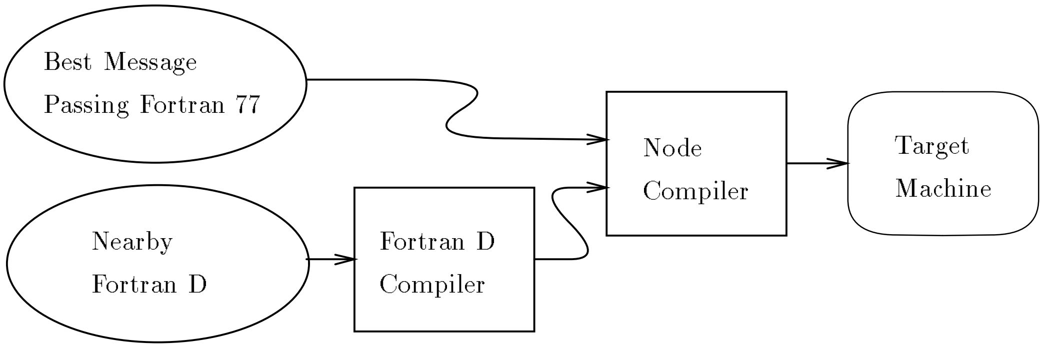

Figure 13: Fortran D Validation Strategy

elegant architecture-independent programming models such as *Bird-Meertens formalism* [Ski90] and the *Bulk-Synchronous* bridging model [Val90]. However, their suitability for scientific programming is unclear, and they also lack language or compiler support.

High-level parallel languages such as Linda [CG89], Strand [FT90, FO90], and Delirium [LS91] are valuable when used to *coordinate* coarse-grained functional parallelism. However, they tend to be inefficient for capturing fine-grain data parallelism of the type described by Hillis and Steele [HS86], because they lack both language and compiler support to assist in efficient data placement. Parallelism must also be explicitly specified when using these languages. As a result, language and compiler support is needed to automatically detect and exploit fine-grained data parallelism.

6.2 Compilation Techniques

The Fortran D compiler borrows heavily from previous research on compiling data-parallel application codes for distributed-memory machines. In the following sections, we look at related systems and their contributions to our Fortran D compilation strategy. First we look at some compilation techniques, then we examine existing compilation systems.

Gupta and Banerjee [GB90] propose a constraint-based approach to automatically calculate suitable data decompositions. They use simple alignments and distributions similar to those in Fortran D. Prins [Pri90] utilizes *shape refinement* in conjunction with linear transformations to specify data layouts and guide resulting data motion. These researchers do not discuss providing compiler support to generate communications.

Some researchers concentrate on computations within loops that only involve a single array. Ramanujam and Sadayappan [RS89] examine both the data and iteration space to derive a combined task and data partition of the loop nest. Hudak and Abraham [HA90] find a stencil-based approach useful for analyzing communications and deriving efficient rectangular or hexagonal data distributions. These researchers do not discuss generating communication for these complex distributions. They also make simplifying assumptions about the effects of the underlying processor topology, and do not consider collective communications.

Wolfe [Wol89, Wol90] describes transformations such as *loop rotation* for distributed-memory programs with simple `BLOCK` distributions. Callahan and Kennedy [CK88] propose methods for compiling programs with user-specified data distribution functions and using compiler inserted *load* and *store* commands to support nonlocal memory accesses. They also demonstrate how such programs can be optimized using transformations such as loop distribution, loop peeling, etc. The Fortran D compiler will apply many of the same transformations. BOOSTER [PvGS90] provides user-specified distribution functions defined as *program views*, but does not generate or optimize communications.

6.3 SIMD Compilation Systems

6.3.1 CM Fortran

CM Fortran [AKLS88, TMC89] is a version of Fortran 77 extended with vector notation, alignment, and data layout specifications. Programmers must explicitly specify data-parallelism in CM FORTRAN programs by marking certain array dimensions as parallel. The operating system of the underlying SIMD distributed-memory machines provides the illusion of infinite machine size through the use of virtual processors. This greatly simplifies the data distribution and communication generation responsibilities of the compiler, and has freed researchers to concentrate on techniques to automatically derive both static and dynamic data alignments [KLS88, TMC89, KLS90, KN90]. More recently, researchers have also begun to study *strip mining* and other techniques to avoid the inefficiencies of using virtual processors [Wei91].

6.3.2 C*

C* [RS87] is an extension of C similar to C++ that supports SIMD data-parallel programs. C* labels data as *mono* (local) or *poly* (distributed). There are no alignment or distribution specifications; the compiler automatically chooses the data decomposition. Parallel algorithms are specified as *actions* on a *domain*, an abstract data type implementation based on the C++ class. Communications are automatically generated by the compiler. As with CM Fortran, virtual processors are generated for each element of a domain and mapped to each physical processor. Researchers have also examined synchronization problems when translating SIMD programs into equivalent SPMD programs, as well as several communication optimizations [QH90].

6.3.3 DINO

DINO [RSW89, RSW90, RW90] is an extended version of C supporting general-purpose distributed computation. DINO supports `BLOCK`, `CYCLIC`, and special stencil-based data distributions with overlaps, but provides no alignment specifications. A DINO program contains a virtual parallel machine declared to be an *environment*. Parallelism is explicitly specified by *composite functions*. Nonlocal memory references must be annotated with the "#" operator. The DINO compiler then translates these references into communications. Passing distributed data as parameters to composite functions also generates nonlocal memory accesses. Special DINO language constructs are provided for reductions. DINO programs are deterministic unless special asynchronous distributed arrays are used. As with CM Fortran, DINO programs generate multiple processes per physical processor when large numbers of virtual processors are declared in the environment.

6.3.4 Paragon

PARAGON [CR89, Ree90] is a programming environment targeted at supporting SIMD programs on MIMD distributed-memory machines. It provides both language extensions and run-time support for task management and load balancing. Data distribution in PARAGON may either be performed by the user or the system. Parallel arrays are mapped onto *shapes* that consist of arbitrary rectangular distributions. Only the first two dimensions of each array may be distributed, and alignment is not supported. The location of each array element may be determined at run-time by checking the *distribution map* stored on each processor. Redistribution and replication of arrays and subarrays, as well as permutation and reduction mechanism are supported. Irregular distributions and run-time preprocessing support is being planned. PARAGON does not perform analysis or transformations to detect or enhance parallelism.

6.3.5 SPOT

SPOT [SS90, Soc90] is a point-based SIMD data-parallel programming language. Distributed arrays are defined as *regions*. Computations are specified from the point of view of a single element in the region, called a *point*. Locations relative to a given point are assigned symbolic names by *neighbor* declarations. An *iteration index* operator allows the programmer to specify whether nonlocal values from neighbors are from the current or previous iteration. This stencil-based approach allows the SPOT compiler to derive efficient *near-rectangular* data distributions. The compiler then generates computation and communication by expanding the single point algorithm to cover all points distributed onto a node. No alignment and or distribution specifications are provided. It is not clear how SPOT will support computation patterns that cannot be described by stencils, or those involving multiple arrays.

6.4 MIMD Compilation Systems

6.4.1 Crystal

CRYSTAL [CCL89, LC90b, LC90a] is a high-level functional language. The CRYSTAL compiler targets distributed-memory machines, performing both automatic data decomposition and communications generation. Programs are first separated into *phases*. Each phase has a different computation structure, represented by an *index domain*. Heuristics are employed to align data arrays with the index domain, both within and across dimensions, then the control structure of the program is derived. Communication patterns are synthesized from the computation, evaluated for a variety of block distributions, then matched with CRYSTAL collective communication routines. Later phases of the compiler generate message-passing C programs for the physical machine. Because it targets a functional language, CRYSTAL does not possess program analysis techniques for imperative languages such as Fortran. It is currently unclear whether the CRYSTAL language can express all scientific computations. Work in progress to adapt the CRYSTAL compiler for scientific Fortran codes will help answer this question.

6.4.2 Id Nouveau

ID NOUVEAU [RP89] is a functional language extended with single assignment arrays called *I-structures*. User-specified `BLOCK` distributions are provided. The basic *run-time resolution* algorithm is similar to the guard and message introduction phases of the Fortran D compiler, but without any attempt to eliminate redundant guards. Guard elimination is described as *compile-time resolution*; it is performed by calculating the set of *evaluators* and *participants* for each statement. Message presending and blocking optimizations are performed using vectorization transformations such as loop fusion and strip mining. Global accumulates are also supported.

6.4.3 SUPERB

SUPERB [ZBG88, Ger90] is a semi-automatic parallelization tool that supports arbitrary user-specified contiguous rectangular distributions. It performs dependence analysis to guide interactive program transformations in a manner similar to the ParaScope Editor [KMT91]. SUPERB originated the *overlap* concept as a means to both specify and store nonlocal data accesses. Once program analysis and transformation is complete, communication is automatically generated and blocked utilizing data dependence information. Some interprocedural analysis is supported. The Fortran D compiler uses overlaps for storing certain classes of nonlocal data. Major differences between SUPERB and the Fortran D compiler include support for data alignment, automatic compilation, collective communications, dynamic data decomposition, and storage choices for nonlocal values.

6.4.4 ASPAR, Express

ASPAR [IFKF90] is a compiler that performs automatic data decomposition and communications generation for loops containing a single distributed array. It utilizes collective communication primitives from the EXPRESS run-time system for distributed-memory machines [EXP89]. ASPAR performs simple dependence analysis using *A-lists* to detect parallelizable loops. The structure of the loop computation may be recognized as a *reduction* operation, in which case the loop is parallelized by replacing the reduction with the appropriate EXPRESS *combine* operation. If the loop performs regular computations on a distributed array, a *micro-stencil* is derived and used to generate a *macro-stencil* to identify communication requirements. Communications utilizing EXPRESS primitives are then automatically generated. ASPAR automatically selects `BLOCK` distributions; no alignment or distribution specifications are provided.

6.4.5 MIMDizer

MIMDIZER [SWW91] is an interactive parallelization system for MIMD shared and distributed-memory machines. Based on FORGE, it performs dataflow and dependence analyses and also supports loop-level transformations. Associated tools also graphically display call graph, control flow, dependence, and profiling information. When programming for distributed-memory machines, users interactively select `BLOCK` or `CYCLIC` distributions for selected array dimensions. *Code spreading* is applied interactively to loops to introduce parallelism. Alignment is not provided. MIMDIZER automatically generates communications corresponding to nonlocal memory accesses at the end of the parallelization session.

6.4.6 Pandore

PANDORE [APT90] is a compiler for distributed-memory machines that takes as input C programs extended with `BLOCK`, `CYCLIC`, and overlapping data distributions. Distributed arrays are mapped by the compiler onto a user-declared *virtual distributed machine* that may be configured as a vector, ring, grid, or torus. The compiler then outputs code in the *vdm_l* intermediate language. Calls to the PANDORE communication library to access nonlocal data is also automatically generated by the compiler. Guard introduction and communications optimization techniques are under development.

6.4.7 AL

AL [Tse90] is a language designed for the Warp distributed-memory systolic processor. The programmer utilizes DARRAY declarations to mark parallel arrays. The AL compiler then applies *data relations* to automatically align and distribute each DARRAY, detect parallelism, and generate communication. Only one dimension of each DARRAY may be distributed, and computations must be *linearly related.*

6.4.8 Oxygen

OXYGEN [RA90] is a compiler for the K2 distributed-memory machine. Unlike systems discussed previously, OXYGEN follows a functional rather than data decomposition strategy. Task-level parallelism is specified by labeling each parallel block of code with a *p_block* directive. Loop-level parallelism is specified by labeling parallel loops with either *split* or *scatter* directives. Decompositions are mapped onto the K2 architecture as *ring*, *rowwise*, or *colwise*. Distributed data arrays may be declared as *local*, *multicopy*, or *singlecopy*, corresponding to private, replicated, and distributed, respectively. Explicit communications directives for reductions and broadcast are also provided. The OXYGEN compiler then converts Fortran code with user directives into C++ node programs with communications. Messages are inserted at points in the program called *checkpoints* to enforce coarse-grain synchronization. Work is in progress to automatically generate OXYGEN directives for functional and data decomposition.

6.4.9 PARTI, ARF

PARTI [SBW90] is a set of run-time library routines that support irregular computations on MIMD distributed-memory machines. PARTI is first to propose and implement user-defined irregular distributions [MSS+88] and a hashed cache for nonlocal values [MSMB90]. PARTI has also motivated the development of ARF, a compiler designed to interface Fortran application programs with PARTI run-time routines. ARF supports `BLOCK`, `CYCLIC`, and user-defined irregular distributions, and generates *inspector* and *executor* loops for run-time preprocessing [KMSB90, WSBH91]. The goal of ARF is to demonstrate that inspector/executors can be automatically generated by the compiler. It does not currently generate messages at compile-time for regular computations.

6.4.10 Kali

KALI [KMV90, MV90] is the first compiler system that supports both regular and irregular computations on MIMD distributed-memory machines. Programs written for KALI must specify a virtual processor array and assign distributed arrays to `BLOCK`, `CYCLIC`, or user-specified distributions. Instead of deriving a functional decomposition from the data decomposition, KALI requires that the programmer explicitly partition loop iterations onto the processor array. This is accomplished by specifying an *on clause* for each parallel loop. Communication is then generated automatically based on the on clause and data distributions. An *inspector/executor* strategy is used for run-time preprocessing of communication for irregularly distributed arrays [KMSB90]. Major differences between KALI and the Fortran D compiler include mandatory on clauses for parallel loops, support for alignment, collective communications, and dynamic decomposition.

7 Conclusions and Future Work

An efficient yet usable 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 programming model. This paper presents the design of a compiler that translates Fortran D to distributed-memory parallel machines, as well as a strategy for evaluating its effectiveness.

The major features of the Fortran D compiler include a rich set of data decomposition specifications, sophisticated intraprocedural and interprocedural analyses, dynamic data decomposition, program transformation, communication optimization, and support for both regular and irregular problems. We expect to be able to generate efficient code for a large class of programs with only minimal effort from the scientific programmer.

The current version of the compiler generates code for a subset of the decompositions allowed in Fortran D, namely unaligned block decompositions. We are extending the implementation to handle other data decompositions. Significant work remains to develop and implement decision algorithms for the optimizations presented in this paper, as well as a whole program compilation algorithm.

8 Acknowledgements

The authors wish to thank Vasanth Balasundaram, Marina Kalem, and Uli Kremer for inspiring many of the ideas in this work. Our research group was formed largely due to their efforts. We also wish to acknowledge the assistance of Chuck Koelbel, especially in the area of irregular distributions. Finally, we are grateful to the ParaScope research group for providing the underlying software infrastructure for the Fortran D compiler.

References

[ACK87] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[AK87] 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.

[AKLS88] 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.

[APT90] 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.

[Bal91] 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.

[BFKK90] 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.

[BFKK91] 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.

[BK89] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[CCH+88] 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.

[CCL89] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, New Haven, CT, December 1989.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CK87] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CKK89] 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.

[CKT86] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[CR89] 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.

[EXP89] Parasoft Corporation. *Express User's Manual*, 1989.

[FHK+90] 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.

[FJL+86] 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, 1986.

[FO90] I. Foster and R. Overbeek. Bilingual parallel programming. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[FT90] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[GB90] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, October 1990.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency—Practice & Experience*, 2(3):171–193, September 1990.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[HK90] P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. Technical Report TR91-154, Dept. of Computer Science, Rice University, March 1991.

[HS86] W. Hillis and G. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[IFKF90] 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.

[KLS88] K. Knobe, J. Lukas, and G. Steele, Jr. Massively parallel data optimization. In *Frontiers88: The 2nd Symposium on the Frontiers of Massively Parallel Computation*, Fairfax, VA, October 1988.

[KLS90] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[KMSB90] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[KMT91] 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.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[KN90] K. Knobe and V. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[LC90a] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[LC90b] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[Lea90] B. Leasure, editor. *PCF Fortran: Language Definition, version 3.1*. The Parallel Computing Forum, Champaign, IL, August 1990.

[LS91] S. Lucco and O. Sharp. Parallel programming with coordination structures. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[MSMB90] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[MSS+88] 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.

[MV90] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. ICASE Report 90-69, Institute for Computer Application in Science and Engineering, Hampton, VA, October 1990.

[PB90] C. Pancake and D. Bergmark. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer*, 23(12):13–23, December 1990.

[Pri90] J. Prins. A framework for efficient execution of array-based languages on SIMD computers. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[PvGS90] E. Paalvast, A. van Gemund, and H. Sips. A method for parallel program generation with an application to the Booster language. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[QH90] M. Quinn and P. Hatcher. Data parallel programming on multicomputers. *IEEE Software*, September 1990.

[RA90] R. Ruhl and M. Annaratone. Parallelization of FORTRAN code on distributed-memory parallel processors. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[Ree90] A. Reeves. The Paragon programming paradigm and distributed memory compilers. Technical Report EE-CEG-90-7, Cornell University Computer Engineering Group, Ithaca, NY, June 1990.

[RP89] 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.

[RS87] J. Rose and G. Steele, Jr. C*: An extended C language for data parallel programming. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[RS89] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[RSW89] 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.

[RSW90] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, Dept. of Computer Science, University of Colorado, April 1990.

[RW90] M. Rosing and R. Weaver. Mapping data to processors in distributed memory computations. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[SBW90] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. ICASE Report 90-59, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[Ski90] D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12), December 1990.

[SMC89] J. Saltz, R. Mirchandaney, and K. Crowley. Runtime parallelization and scheduling of loops. In *Proceedings of the 1st Symposium on Parallel Algorithms and Architectures*, Santa Fe, NM, 1989.

[Soc90] D. Socha. Compiling single-point iterative programs for distributed memory computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[SS90] L. Snyder and D. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[SWW91] R. Sawdayi, G. Wagenbreth, and J. Williamson. MIMDizer: Functional and data decomposition; creating parallel programs from scratch, transforming existing Fortran programs to parallel. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, to appear 1991.

[TMC89] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 5.2-0.6 edition, September 1989.

[Tse90] 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.

[Val90] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[Wei91] M. Weiss. Strip mining on SIMD architectures. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[Wol89] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[Wol90] M. J. Wolfe. Loop rotation. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[WSBH91] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.