Compiling Fortran 77D and 90D
for MIMD Distributed-Memory Machines

Alok Choudhary* Geoffrey Fox* Seema Hiranandani† Ken Kennedy†
Charles Koelbel† Sanjay Ranka* Chau-Wen Tseng†

Abstract

Fortran D provides a set of data decomposition specifications for either Fortran 77 or Fortran 90. In this paper we present an integrated approach to compiling Fortran 77D and Fortran 90D programs into SPMD (Single Program Multiple Data) message-passing programs for efficient execution on MIMD distributed-memory machines. The integrated Fortran D compiler relies on two key observations. First, array constructs may be scalarized by the Fortran 90 front end into forall loops without loss of information. Second, loop fusion, partitioning, and sectioning optimizations in the Fortran D back end can be useful for both Fortran D dialects.

1 Introduction

Parallel computing on distributed-memory machines is very cost-effective, but is hindered by the lack of portability for the resulting programs. Our goal is to establish a machine-independent programming model for data-parallel programs that is easy to use, yet performs with acceptable efficiency on different parallel architectures. Toward this end we are developing the Fortran D language extensions, which are applicable to both Fortran 77 and Fortran 90. Hereafter, we use “Fortran D” to refer to concepts common to both dialects, and Fortran 77D/Fortran 90D for dialect-specific comments.

In this paper, we describe a unified strategy for compiling both Fortran 77D and Fortran 90D the Intel iPSC/860 and Touchstone, two MIMD distributed-memory machines. The principal issues involved in compiling Fortran 90D are partitioning the program across multiple nodes and scalarizing

*Northeast Parallel Architectures Center, 111 College Place, Syracuse University, Syracuse, NY 13244-4100
†Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251-1892
it for execution on each individual node. Previous work has described the partitioning process [20, 21, 22]. Here, we argue that efficient scalarization is also an interesting problem, and demonstrate how to integrate it with program partitioning. We also describe runtime library support for such a compiler.

The remainder of this paper presents a brief overview of the Fortran D language and compilation strategy, then describes the Fortran 90D and 77D front ends, and the Fortran D back end in detail. The design of the runtime library is discussed, and an example is used to illustrate the compilation process. We conclude with a discussion of related work.

2 Fortran D Language

In this section, we briefly overview aspects of Fortran D relevant to this paper. The complete language is described elsewhere [16].

Fortran D provides explicit control over data partitioning with alignment and distribution specifications. The DECOMPOSITION declaration specifies an abstract problem or index domain. The ALIGN statement specifies fine-grain parallelism, mapping each array element onto one or more elements of the decomposition. This provides the minimal requirement for reducing data movement for the program given an unlimited number of processors.

The Distribute statement specifies coarse-grain parallelism, grouping decomposition elements and mapping them and aligned array elements to physical processor memories. Each dimension of the decomposition is distributed in a block, cyclic, or block-cyclic manner, or is undistributed. Some sample data alignment and distributions are shown in Figure 1, the complete language is described in detail elsewhere [16].

Fortran D provides FORALL loops to permit the user to specify difficult parallel loops in a deterministic manner [4]. In a FORALL loop, each iteration uses only values defined before the loop or within the current iteration. When a statement in an iteration of the FORALL loop accesses a memory location, it will receive a value written on the current iteration, if there is one; otherwise, it gets the value at that memory location before the execution of the FORALL loop. A merging semantics ensures that a deterministic value is obtained after the FORALL if several iterations assign to the same memory location. This semantics has also been called copy-in/copy-out semantics.
3 Fortran D Compilation Strategy

3.1 Overall Strategy

Our strategy for parallelizing Fortran D programs for distributed-memory MIMD computers is illustrated in Figure 2. In brief, we transform both Fortran 77D and Fortran 90D to a common intermediate form, which is then compiled to code for the individual nodes of the machine. We have several pragmatic and philosophical reasons for this strategy. Sharing a back end for the Fortran 77D and Fortran 90D avoids duplication of effort and allows both compilers to immediately take advantage of new optimizations. Decoupling the Fortran 77D and Fortran 90D front ends allows them to be machine independent. Finally, one of our long-term research goals is to define an efficient compiler/programmer interface for programming the nodes of a massively parallel machine. Providing a common intermediate form helps us experiment with this interface.

3.2 Intermediate Form

To compile both dialects of Fortran D using a single back end, we must have an appropriate intermediate form. The most important requirement is that the intermediate form must capture three aspects of the program:

- Data decomposition information, telling how data is aligned and distributed among processors.

- Parallelization information, telling when operations in the code are independent.

- Communication information, telling what data must be transferred between processors.

In addition, the primitive operations of the intermediate form should be operations that can be translated simply for single-processor execution.

We have chosen Fortran 77 with data decompositions, FORALL, and intrinsic functions as the intermediate form for the Fortran D compiler. We claim that this form preserves all of the information available in a Fortran 90 program, but maintains the flexibility of Fortran 77.

3.3 Node Interface

Another topic of interest in the overall strategy is the node interface — the program produced by the Fortran D compiler. It must be both portable and efficient. In addition, the level of the node interface should be neither so high that efficient translation to object code is impossible, nor so low that its workings are completely opaque to the user. We have selected Fortran 77 with calls to
a runtime library based on Express [30] as the node interface. We will eventually evaluate this experience to further refine our notion of an "optimal" support level.

4 Fortran D Compiler

The Fortran D compiler consists of three parts: two front ends to process input programs into the common intermediate form, and one back end to compile this to the node interface. The Fortran D compiler is implemented on top of the ParaScope programming environment to take advantage of its advanced features [12, 25].

4.1 Fortran 90D Front End

The function of the Fortran 90D front end is to scalarize the Fortran 90D program, translating it to an equivalent Fortran 77D program. This is necessary because the underlying machine executes computations sequentially, rather than on entire arrays at once as specified in Fortran 90. For the Fortran D compiler we find it useful to view scalarization as three separate tasks:

- **Scalarizing Fortran 90 Constructs.** Many Fortran 90 features are not present in our intermediate form and must be translated into equivalent Fortran 77D statements.

- **Fusing Loops.** Simple scalarization results in many small loop nests. Fusing these loop nests can improve performance, as we show below.

- **Sectioning.** The Fortran D compiler must divide Fortran 90's atomic array operations into sections that can execute on the target machine [5, 7].

We defer both loop fusion and sectioning to the Fortran D back end. We do this because Fortran 77D programs can also benefit from these optimizations. The Fortran 90D front end handles scalarizing Fortran 90 constructs that have no equivalent in the Fortran 77D intermediate form. There are three principal Fortran 90 language features that must be scalarized: array constructs, WHERE statements, and intrinsic functions [9].

Array Constructs  Fortran 90 array constructs allow entire arrays to be manipulated atomically. These operations enhance the clarity and conciseness of the program, and help to make parallelism explicit. Previous research has shown that efficiently implementing array constructs for scalar machines is a difficult problem [5, 7]. The Fortran 90 front end can defer these problems by relying on a key observation—the FORALL loop's semantics are identical to array assignment semantics. Array
constructs can therefore be translated into equivalent FORALL loops with no loss of information. Some optimizations to simplify the resulting index expressions are still useful, but not central to the design of our compiler.

**WHERE Statement**  Another Fortran 90 feature that has no Fortran 77 equivalent is the WHERE statement. It requires a boolean argument that is used to mask array operations, inhibiting assignments to some array elements. As was the case with the rhs above, the boolean argument must be completely evaluated before executing the body of the statement. Using the same insight above, the WHERE statement may be translated into equivalent IF nested within a FORALL statement.

**Intrinsic Functions**  Intrinsic functions not only provide a concise means of expressing operations on arrays, but also identify computation patterns that may be difficult to detect automatically. Fortran 90 provides intrinsic functions for operations such as shift, reduction, transpose, and matrix multiplication. To avoid excessive complexity and machine-dependence in the Fortran D compiler, we convert most Fortran 90 intrinsics into calls to customized runtime library functions. However, some processing is necessary. Like the WHERE statement, some intrinsic functions accept a mask expression that restricts execution of the computation. The Fortran 90D front end may need to evaluate the expression and store it in a temporary boolean array before performing the computation. As with the other constructs, avoiding these temporaries is a key optimization.

**Temporary Arrays**  When the Fortran 90D front end needs to create temporary arrays, it must also generate appropriate Fortran D data decomposition statements. A temporary array is usually aligned and distributed in the same manner as its "master" array, i.e., the array whose computation created the need for the temporary.

### 4.2 Fortran 77D Front End

The Fortran 77D front end is simple because Fortran 77D is very close to the intermediate form. Its only task is to detect complex high-level parallel computations, replacing them by equivalent Fortran 90 intrinsics to take advantage of the optimized runtime library. With advanced program analysis, some operations such as DOTPRODUCT, SUM, or TRANSPOSE can be detected automatically[31].

### 4.3 Fortran D Back End

The Fortran D back end performs two main functions—partition the program onto the nodes of the parallel machine, and complete the scalarization of Fortran D into Fortran 77. This process is
organized as a series of data dependence-based transformations.

Data dependence is a concept developed for vectorizing and parallelizing compilers to characterize memory accesses at compile time [8, 27, 34]. True dependences indicate definition followed by use, anti-dependences show use before definition. Data dependences may be either loop-carried or loop-independent.

The transformations performed by the back end are loop fusion, followed by partitioning and sectioning. Loop fusion is performed first because it simplifies partitioning by reducing the need to consider inter-loop interactions. It also enables optimizations such as strip-mining and loop interchange [8, 34]. In addition, loop fusion does not increase the difficulty of later compiler phases. In comparison, sectioning is performed last because it can significantly disrupt the existing program structure, increasing the difficulty of partitioning analysis and optimization.

4.3.1 Loop Fusion

Loop fusion is legal if it does not reverse the direction of any data dependence between two loop nests [5, 6, 34]. Choosing how to fuse loops to maximize the number of parallel loops is NP-hard in general, even for singly nested loops without statement reordering [18]. The Fortran D back end fuses adjoining loop nests if and only if no new true loop-carried dependences are created for the fused loops. This is a heuristic which we have reason to think will perform well in practice.

Loop fusion has the added advantage, when used in concert with other transformations, of improving memory reuse in the resulting program, a significant benefit on modern processors [1, 5, 7, 27]. Our results in Section 6 give an example of this. We have described other transformations with the same goals elsewhere [24].

4.3.2 Program Partitioning

The major step in compiling Fortran D for MIMD distributed-memory machines is to partition the data and computation across processors, introducing communication where needed. We present a brief overview of the Fortran D partitioning algorithm below; details are discussed elsewhere [20, 21, 22]. Slight extensions are introduced here for compiling FORALL loops and intrinsics.

Analyze Program The Fortran D compiler performs aggressive scalar dataflow analysis and dependence testing.

Partition data The compiler analyzes Fortran D alignment and distribution statements calculate the array section owned by each processor.
Partition computation  The compiler partitions computation across processors using the “owner computes” rule — each processor only computes values of data it will store [13, 32, 36]. It relaxes this rule for reduction operations and assignments to private variables. This results in each processor being assigned a subset of the iterations of each loop.

Analyze communication  Once the work partition is calculated, it is used to mark references that may result in nonlocal accesses.

Optimize communication  The compiler examines each marked nonlocal reference, using results of the analysis phases, and attempts to reduce the communications costs. Message vectorization [32] and choosing the loop level at which to insert the communication[10, 17] are the key optimizations applied here.

To properly handle FORALL loops, the Fortran D compiler treats all their carried true dependences as anti-dependences. This reflects the semantics of the FORALL loop and ensures that the message vectorization algorithm will insert all communication outside the loop.

Manage storage  The compiler collects the extent and type of nonlocal data accesses to calculate the storage required for nonlocal data. Overlaps [17] and temporary buffers are used depending on the classification of the nonlocal access.

Generate code  Finally, the Fortran D compiler uses the results of the above phases to generate the node interface program. Some of the tasks involved are reducing array and loop bounds instantiate the data and computation partition, generating the actual communications calls, and adjusting parameters needed for the intrinsic functions in the runtime library.

4.3.3 Sectioning

The final phase of the Fortran D back end completes the scalarization process by applying sectioning to convert FORALL loops into DO loops [5, 7]. True dependences carried on the FORALL loop represent instances where values are defined in the loop and used on later iterations thus violating FORALL semantics. The Fortran D uses this analysis in the back end to detect cases where temporary storage may be needed. This allows us to avoid the straightforward translation of

\[ A(2:N-1) = \frac{(A(1:N-2) + A(3:N))}{2} \]

into

DO i = 1, N-2
    TMP(i) = A(i-1)
ENDDO
DO i = 2, N-1  
    A(i) = (TMP(i) + A(i+1))/2  
ENDFOR

(where the temporary array is needed to preserve old values of A(i - 1)). Instead, the Fortran D compiler can reduce the amount of temporary storage required through data prefetching [7], producing

X = A(1)  
DO i = 2, n-1  
    Y = (X + A(i+1))/2  
    X = A(i)  
    A(i) = Y  
ENDFOR

5 Runtime Library

Fortran 90 intrinsic functions represent computations (such as TRANSPOSE and MATMUL) that may have complex communication patterns. It is possible to support these functions at compile time, but we have chosen to implement these functions in the runtime library instead to reduce the complexity and machine-dependence of the compiler.

The Fortran D compiler translates intrinsics into calls to runtime library routines using a standard interface. Additional information is passed describing bounds, overlaps, and partitioning for each array dimension. The runtime library is built on top of the Express [30] communication package to enable portability to other systems.

Table 1 presents a sample of performance numbers for a subset of the intrinsic functions on iPSC/860. A detailed performance study is presented in [2]. The times in the table include both the computation and communication times for each function. For most of the functions we were able to obtain almost linear speedups. In the case of TRANSPOSE function, going from one processor to two or four actually results in increase in the time due to the communication requirements. However, for larger size multiprocessors the times decrease as expected.

6 Example and Performance Results

In this section we demonstrate how an example Fortran 90D code is compiled into executable code for a 32 node iPSC/860. Figure 3 shows a Fortran 90D code fragment implementing one sweep of ADI integration on a 2D mesh, a typical numerical algorithm. Fortran D specifications partition the 2-D array into 32 blocks of rows, one for each processor. (We use a fixed number of processors
only to simplify the output programs.

We tested three versions of the ADI code on the iPSC/860, two of which are shown in Figures 4 and 5. For clarity, many details in the codes have been elided or simplified, and the second loop in the original program is deleted. First, the code in Figure 4 represents the simplest translation of Fortran 90D to our node interface. This is shown as the dotted line in Figure 6. Second, the code in Figure 5 represents the fully optimized version of the same program. Loop fusion has been applied, and used to enable pipelining of computation and communication. This appears as the solid line in Figure 6. Third, an intermediate code which applied only the loop fusion optimization to Figure 4 represented applying only Fortran 90 scalarization optimizations. This appears as the dashed line in Figure 6. All three programs were compiled under -O4 using Release 2.0 of if77, the iPSC/860 compiler. Timings were taken using dclock() on a 32 node Intel iPSC/860 with 8 Meg of memory per node.

The first version of the program shows almost no speedup. Here, sequentialization is caused by conflict between the natural parallelism (independent rows) and the data distribution (independent columns). Loop fusion provides a small improvement due to memory reuse. Pipelining, however, exposes parallelism to achieve the best performance. Note that this required both Fortran 90 and Fortran 77 optimizations to achieve. We conclude that some algorithms require sophisticated optimization both in the scalarization and partitioning phase; a compiler must incorporate both to compete with hand-crafted code.

7 Related Work

The Fortran D compiler is a second-generation distributed-memory compiler that integrates and extends many previous analysis and optimization techniques. Many distributed-memory compilers reduce communication overhead by aggregating messages outside of parallel loops [23, 26] or parallel procedures [19, 33], while others rely on functional or data-flow languages [28, 32].

In comparison, the Fortran D compiler uses dependence analysis to automatically exploit parallelism and extract communication even from partially sequential loops. The Fortran D compiler also applies analysis and optimization before code generation, unlike earlier transformation-based systems that apply program transformations and partial evaluation after inserting fine-grain communications [13, 32, 36].

Several other projects are also developing Fortran 90 compilers for MIMD distributed-memory
machines. This group includes ADAPT, based on a runtime intrinsic library [29], CM FORTRAN, which introduced alignment and layout specifications [3], and a proposal by Wu & Fox that also discussed program generation and optimization [35]. The last project has been most directly influential on our work. PARAGON is a version of C extended with array syntax, operations, reductions, permutations, and distribution specifications [14]. None of these systems describe scalarization or communication optimizations in detail, to which we give high priority.

We have already mentioned previous work on Fortran 90 implementation on scalar and vector machines [1, 5, 7, 27]. We use many of the same transformations, but in a new context.

8 Conclusions

This paper presented an integrated approach to compiling both Fortran 77D and 90D. Our approach is based on a common back-end implementing optimizations useful for both dialects. A key feature of this research is its emphasis on efficient scalarization of Fortran 90 constructs, which has an important effect on individual-node performance. We have also described how a portable runtime library can also reduce the complexity of the compiler.

The Fortran D compiler for MIMD distributed-memory machines is only a part of the Fortran D project; we also are working on Fortran 77D and Fortran 90D compilers for SIMD machines, translations between the two Fortran dialects, support for irregular computations, and environmental support for static performance estimation and automatic data decomposition [10, 11, 15].

9 Acknowledgements

We are grateful to the ParaScope and Fortran D research groups for their assistance, and to Parasoft for providing the Fortran 90 parser and Express. This research was supported by the Center for Research on Parallel Computation (CRPC), a National Science Foundation Science and Technology Center. Use of the Intel iPSC/860 was provided by the CRPC under NSF Cooperative Agreement Nos. CCR-8809615 and CDA-8619893 with support from the Keck Foundation. This work was sponsored by DARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or policy of the government, and no official endorsement should be inferred.
References


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

DISTRIBUTED D(:, BLOCK) DISTRIBUTED D(CYCLIC, :) 

Figure 1: Fortran D Data Decomposition Specifications

Fortran 77

Fortran 77D
+ data decompositions + FORALL

Intermediate Specification
+ data decompositions + FORALL + intrinsics

Fortran D
Back End

Node Interface
Fortran 77 + message passing + runtime library

Portable Runtime Library

Fortran 77 Compilers

Touchstone Delta iPSC/860 Ncube/2 Workstations

Fortran 90

Fortran 90D
+ array constructs + WHERE + intrinsics
+ data decompositions + FORALL + intrinsics

User

User

Figure 2: Fortran D Compilation Strategy
<table>
<thead>
<tr>
<th>Proc</th>
<th>ALL (1K x 1K)</th>
<th>ANY (1K x 1K)</th>
<th>MAXVAL (1K x 1K)</th>
<th>PRODUCT (256K)</th>
<th>DOT PRODUCT (256K)</th>
<th>TRANSPOSE (512 x 512)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>580.6</td>
<td>606.2</td>
<td>658.8</td>
<td>90.1</td>
<td>164.8</td>
<td>299.0</td>
</tr>
<tr>
<td>2</td>
<td>291.0</td>
<td>303.7</td>
<td>330.4</td>
<td>50.0</td>
<td>83.0</td>
<td>575.0</td>
</tr>
<tr>
<td>4</td>
<td>146.2</td>
<td>152.6</td>
<td>166.1</td>
<td>25.1</td>
<td>42.2</td>
<td>395.0</td>
</tr>
<tr>
<td>8</td>
<td>73.84</td>
<td>77.1</td>
<td>84.1</td>
<td>13.1</td>
<td>22.0</td>
<td>213.0</td>
</tr>
<tr>
<td>16</td>
<td>37.9</td>
<td>39.4</td>
<td>43.4</td>
<td>7.2</td>
<td>12.1</td>
<td>121.0</td>
</tr>
<tr>
<td>32</td>
<td>19.9</td>
<td>20.7</td>
<td>23.2</td>
<td>4.2</td>
<td>7.4</td>
<td>69.0</td>
</tr>
</tbody>
</table>

Table 1: Performance of Some Fortran 90 Intrinsic Functions

PARAMETER (N = 512)
PARAMETER (N$PROC = 32)
REAL X(N,N), A(N,N), B(N,N)
DECOMPOSITION DEC(N,N)
ALIGN X, A, B WITH DEC
DISTRIBUTE DEC(BLOCK,:
DO I = 2, N
  X(I,1:N) = X(I,1:N) - X(I-1,1:N)*A(I,1:N)/B(I-1,1:N)
ENDDO
X(N,1:N) = X(N,1:N) / B(N,1:N)
J = N
DO I = 1, N-1
  J = J - 1
  X(J,1:N) = (X(J,1:N)-A(J+1,1:N)*X(J+1,1:N))/B(J,1:N)
ENDDO

Figure 3: ADI integration in Fortran 90D

REAL X(0:17,512), A(16,512), B(0:16,512)
MY$PROC = myproc() { 1...32 }
LB1 = MAX((MY$PROC-1)*16+1,2) - (MY$PROC-1)*16
IF (MY$PROC .GE. 2) THEN
  recv(X(0,1:512),B(0,1:512),MY$PROC-1)
ENDIF
DO I = LB1, 16
  DO K = 1, N
    X(I,K) = X(I,K) - X(I-1,K)*A(I,K)/B(I-1,K)
  ENDDO
  DO K = 1, N
    B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)
  ENDDO
ENDDO
IF (MY$PROC .LE. 31) THEN
  send(X(16,1:512),B(16,1:512),MY$PROC+1)
ENDIF

Figure 4: ADI Output (Unoptimized)
REAL X(0:17,512), A(16,512), B(0:16,512)
MY$PROC = myproc() {* 1...32 *}
LB1 = MAX((MY$PROC-C+1)*16+1,2) - (MY$PROC-1)*16
DO KK = 1, 512, 4
  IF (MY$PROC .GE. 2) THEN
    recv(X(0,KK:KK+3),B(0,KK:KK+4),MY$PROC-1)
  ENDIF
  DO I = LB1, 16
    DO K = KK, KK+3
      X(I,K) = X(I,K) - X(I-1,K)*A(I,K)/B(I-1,K)
      B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)
    ENDDO
  END
  IF (MY$PROC .LE. 31) THEN
    send(X(16,KK:KK+3),B(16,KK:KK+3),MY$PROC+1)
  ENDIF
ENDDO

Figure 5: ADI Output (w/ Loop Fusion & Pipelining)

![Graph](graph.png)

Figure 6: Performance of three versions of ADI integration