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

CRPC-TR92203
March 1992

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

In *Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

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

Alok Choudhary
Geoffrey Fox
Sanjay Ranka

Northeast Parallel Architectures Center
111 College Place
Syracuse University
Syracuse, NY 13244-4100

Seema Hiranandani
Ken Kennedy
Charles Koelbel
Chau-Wen Tseng

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

Abstract

We present an integrated approach to compiling Fortran 77D and Fortran 90D 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* into FORALL loops without loss of information. Second, *loop fusion*, *partitioning*, and *sectioning* optimizations are essential for both Fortran D dialects.

1 Introduction

Parallel computing on distributed-memory machines is very cost-effective, but it is hindered by both the difficulty of parallel programming and lack of portability of the resulting programs. We propose to solve this problem by developing the compiler technology needed to automate translation of Fortran D to different parallel architectures. 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.

Fortran D provides data decomposition specifications that can be applied to Fortran 77 and Fortran 90 [8] to produce Fortran 77D and Fortran 90D, respectively. In this paper, we describe a unified strategy for compiling both Fortran 77D and Fortran 90D into efficient SPMD (Single Program Multiple Data) message-passing programs. In particular, we concentrate on the design of a prototype Fortran 90D compiler for the Intel iPSC/860 and Delta, two MIMD distributed-memory machines.

The principal issues involved in compiling Fortran 90D are *partitioning* the program across multiple nodes and *scalarizing* it for execution on each individual node. Previous work has described the partitioning process [20, 21]. In this paper we demonstrate how to integrate partitioning with scalarization, and show that an efficient portable run-time library can ease the task of compiling Fortran D.

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 common Fortran D back end. The design of the run-time 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

We briefly overview aspects of Fortran D relevant to this paper. These extensions can be added to either Fortran 77 or Fortran 90. The complete language is described elsewhere [16].

2.1 Data Alignment and Distribution

In Fortran D, the DECOMPOSITION statement declares an abstract problem or index domain. The ALIGN statement maps 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 groups decomposition elements, mapping them and any array elements aligned with them to the finite resources of the physical machine. Each dimension of the decomposition is distributed in a block, cyclic, or block-cyclic manner; the symbol ":" marks dimensions that are not distributed. Because the alignment and distribution statements are executable, dynamic data decomposition is possible.

2.2 Forall

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 not get any value written by a different iteration of the loop. Instead, it will get the *old* value at that memory location (*i.e.,* the value at that location before the execution of the FORALL loop) or it will get some new value written on the current iteration. Similarly, a merging semantics ensures that a deterministic value is obtained after the FORALL if several iterations assign to the same memory location.

Another way of viewing the FORALL loop is that it has copy-in/copy-out semantics. In other words, each iteration gets its own copy of the entire data space that exists before the execution of the loop, and writes its results to a new data space at the end of the loop. Since no values depend on other iterations, the FORALL loop may be executed in parallel without synchronization. However, communication may still be required before the loop to acquire nonlocal values, and after the loop to update or merge nonlocal values. Single-statement Fortran D FORALL loops are identical to those supported in CM FORTRAN [34].

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 1. 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 common back end for both the Fortran 77D and Fortran 90D avoids duplication of effort.
- Decoupling the Fortran 77D and Fortran 90D front ends allows them to become machine independent.
- Providing a common intermediate form helps us experiment with defining an efficient compiler/programmer interface for programming the nodes of a massively parallel machine.

3.2 Intermediate Form

To compile both dialects of Fortran D using a single back end, we must select an appropriate intermediate form. In addition to standard computation and control flow information, the intermediate form must capture three important 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, we believe that the primitive operations of the intermediate form should be relatively low-level operations that can be translated simply for single-processor execution.

We have chosen Fortran 77 with data decompositions, FORALL, and intrinsic functions to be the intermediate form for the Fortran D compiler. We show later that this form preserves all of the information available in a Fortran 90 program, but maintains the flexibility of Fortran 77. Parallelism and communication can be determined by the compiler for simple computations, and specified by the user using FORALL and intrinsic functions for complex computations.

3.3 Node Interface

Another topic of interest in the overall strategy is the node interface—the node 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 communication and run-time libraries based on Express, a collection of portable message-passing primitives [30]. Evaluating our experiences with this node interface is the first step towards defining an "optimal" level of support for programming individual nodes of a parallel machine.

4 Fortran D Compiler

The Fortran D compiler thus consists of three parts. The Fortran 90D and 77D front ends process input programs into the common intermediate form. The Fortran D back end then compiles this to the SPMD message-passing node program. The Fortran D compiler is implemented in the context of the ParaScope programming environment [12].

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:

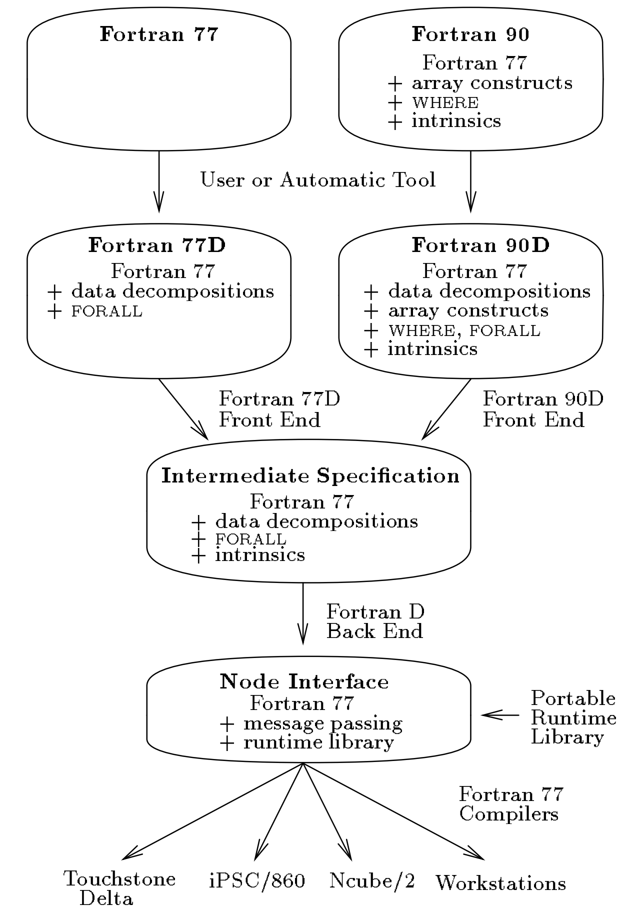

Figure 1: Fortran D Compilation Strategy

- **Scalarizing Fortran 90 Constructs.** Many Fortran 90 features are not present in our intermediate form. They must be translated into equivalent Fortran 77D statements.
- **Fusing Loops.** Simple scalarization results in many small loop nests. Fusing these loop nests can improve the locality of data accesses, simplify partitioning, and enable other program transformations.
- **Sectioning.** Fortran 90 array operations allow the programmer to access and modify entire arrays atomically, even if the underlying machine lacks this capability. The Fortran D compiler must divide array operations into *sections* that fit the hardware of the target machine [5, 6].

We defer both loop fusion and sectioning to the Fortran D back end. Loop fusion is deferred because even hand-written Fortran 77 programs can benefit significantly [24, 28]. Sectioning is needed in the back end because FORALL loops may also be present in Fortran 77D.

We assign to the Fortran 90D front end the remaining task, 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 [8].

	Sending & Receiving	Reduction	Multicasting	Irregular Operations	Special routines
Fortran 90D	CSHIFT EOSHIFT	DOTPRODUCT ALL, ANY, COUNT MAXVAL, MINVAL SUM, PRODUCT MAXLOC, MINLOC	SPREAD	PACK UNPACK RESHAPE TRANSPOSE	MATMUL

Table 1: Representative Intrinsic Functions of Fortran 90D

Array Constructs Fortran 90 *array constructs* allow entire arrays to be manipulated atomically. Array sections may also be specified using triplet notation. This enhances the clarity and conciseness of the program, and has the advantage of making parallelism explicit. It is the responsibility of the compiler to efficiently implement array constructs for scalar machines. Previous research has shown that this is a difficult problem [5, 6].

One problem is that when Fortran 90 array constructs are used in assignment statements, the entire right-hand side (*rhs*) must be evaluated before storing the results in the left-hand side (*lhs*). If an assignment statement utilizing array constructs is translated naively without adequate analysis, *rhs* array elements would need to be stored in temporary buffers to ensure that they are not overwritten before their values are used.

The Fortran 90 front end can defer this problem by relying on a key observation—the FORALL loop possesses copy-in/copy-out semantics identical to Fortran 90 assignment statements utilizing array constructs. Such statements may thus be translated into equivalent FORALL loops with no loss of information.

However, since FORALL loops specify individual element operations, indices are introduced. For simplicity, the index calculation is performed with respect to the *lhs*. In general, an array construct of the form:

A($l_1 : u_1 : s_1$) = B($l_2 : u_2 : s_2$)

where A and B are one dimensional arrays, is converted into:

```
FORALL i = $l_1$, $u_1$, $s_1$
  A(i) = B($l_2$ + ((i - $l_1$)/$s_1$) * $s_2$)
ENDFOR
```

Of course, the expression in the *rhs* is simplified as much as possible at compile time.

WHERE Statement Another Fortran 90 feature that has no Fortran 77 equivalent is the WHERE statement. It takes a boolean argument that is used to *mask* array operations, inhibiting assignments to array elements whose matching boolean flag has the value *false*. The boolean argument to the WHERE statement must be completely evaluated before the body of the statement may be executed.

Fortunately, the WHERE statement may be easily translated into equivalent IF and FORALL statements. Consider the following example where A is assumed to be an 1D N-element array. Because of FORALL copy-in/copy-out semantics, it is unnecessary at this point to explicitly store the value of the boolean argument to prevent it from being overwritten.

```
WHERE (A .EQ. 0)          FORALL i = 1,N
  A = 1.0                   IF (A(i) .EQ. 0) THEN
ELSEWHERE         ==>         A(i) = 1.0
  A = 0.0                   ELSE
ENDWHERE                      A(i) = 0.0
                            ENDIF
                          ENDFOR
```

Intrinsic Functions Intrinsic functions are fundamental to Fortran 90. They not only provide a concise means of expressing operations on arrays, but also identify parallel computation patterns that may be difficult to detect automatically. Fortran 90 provides intrinsic functions for operations such as shift, reduction, transpose, and matrix multiplication. Additional intrinsics are described in Table 1. To avoid excessive complexity and machine-dependence in the Fortran D compiler, we convert most Fortran 90 intrinsics into calls to customized run-time library functions.

The strategy used by the Fortran 90D front end is thus to preserve all intrinsic functions, passing them to the Fortran D compiler back end. 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, so the mask can be passed as an argument to the run-time library.

For example, consider the following reduction operation, where X is a scalar and A, B are arrays:

```
X = MAXVAL(A, A .EQ. B)
```

It should return the value of the element of A that is the maximum of all elements for which element of A is equal to the corresponding element of B. The Fortran 90D front end translates this to:

```
FORALL i = 1,N
  TMP(i) = A(i) .EQ. B(i)
ENDFOR
X = MAXVAL(A, TMP)
```

TMP can then be passed as an argument to the run-time routine MAXVAL. Temporary arrays may also be introduced when intrinsic functions return a value that is part of a Fortran 90 expression.

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. For example, in the previous example the temporary logical array TMP is aligned and distributed in the same manner as A and B. If A and B are distributed differently, then the temporary array is assigned the distribution of A, the first argument.

4.2 Fortran 77D Front End

The Fortran 77D front end does not need to perform much work since Fortran 77D is very close to the intermediate form. Its only task is to detect complex high-level parallel computations, replacing or annotating them by their equivalent Fortran 90 intrinsics. These intrinsic functions help the compiler recognize complex computations such as reductions and scans that are supported by the run-time library. With advanced program analysis, some operations such as DOTPRODUCT, SUM, TRANSPOSE, or MATMUL can be detected automatically with ease. Others computations such as COUNT or PACK may require user assistance.

4.3 Fortran D Back End

The Fortran D back end performs two main functions—it partitions the program onto the nodes of the parallel machine and completes the scalarization of Fortran D into Fortran 77. We find that the desired order for compilation phases is to apply loop fusion first, 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* [7, 36]. In addition, loop fusion does not increase the difficulty of later compiler phases. On the other hand, 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 particularly important for the Fortran D back end because scalarized Fortran 90 programs present many single-statement loop nests. Fusing such loops simplifies the partitioning process and enables additional optimizations.

Data dependence is a concept developed for vectorizing and parallelizing compilers to characterize memory access patterns at compile time [7, 26, 36]. A true dependence indicates definition followed by use, while an anti-dependence shows use before definition. Data dependences may be either loop-carried or loop-independent. Loop fusion is legal if it does not reverse the direction of any data dependence between two loop nests [5, 35, 36].

The current Fortran D back end fuses all adjacent loop nests where legal, if no loop-carried true dependences are introduced. This heuristic does not adversely affect the parallelism or communication overhead of the resulting program, and should perform well for the simple cases found in practice. More sophisticated algorithms are discussed elsewhere [17, 28, 35].

Loop fusion also has the added advantage of being able to improve memory reuse in the resulting program. Modern high-performance processors are so fast that memory latency and bandwidth limitations become the performance bottlenecks for most scientific programs. Transformations such as loop fusion promote memory reuse and can significantly improve program efficiency for both scalar and vector machines [1, 5, 6, 26, 28, 33]. For instance, consider the following example.

```
FORALL i = 1,N               FORALL i = 1,N
  A(i) = i                     A(i) = i
ENDFOR               ==>       B(i) = A(i)*A(i)
FORALL i = 1,N               ENDFOR
  B(i) = A(i)*A(i)
ENDFOR
```

The occurrences of $A(i)$ in separate loops means that the memory location referenced by $A(i)$ in the first loop is likely to have been flushed from the cache by the reference in the second loop. If the two loops are fused, all accesses to $A(i)$ occur in the same loop iteration, allowing the value to be reused in a register or cache. For this example, we measured improvements of up to 30% for some problem sizes on an Intel i860, as shown in Figure 2. Additional transformations to enhance memory reuse and increase unit-stride memory accesses are also quite important; they are described elsewhere [24, 28].

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 compilation process below, details are discussed elsewhere [20, 21].

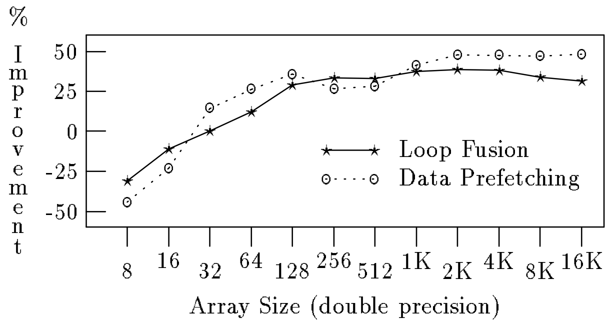

Figure 2: Effect of Scalarization Optimizations

- **Analyze Program.** Symbolic and data dependence analysis is performed.
- **Partition data.** Fortran D data decomposition specifications are analyzed to determine the decomposition of each array in a program.
- **Partition computation.** The compiler partitions computation across processors using the "owner computes" rule—where each processor only computes values of data it owns [13, 31, 38].
- **Analyze communication.** Based on the work partition, references that result in nonlocal accesses are marked.
- **Optimize communication.** Nonlocal references are examined to determine optimization opportunities. The key optimization, message vectorization, uses the level of loop-carried true dependences to combine element messages into vectors [9, 38].
- **Manage storage.** "Overlaps" [38] or buffers are allocated to store nonlocal data.
- **Generate code.** Information gathered previously is used to generate the SPMD program with explicit message-passing that executes directly on the nodes of the distributed-memory machine.

Two extensions are needed in the Fortran D back end to handle FORALL loops and intrinsics. During communication optimization, the Fortran D compiler treats all true dependences carried by FORALL loops as anti-dependences. This reflects the semantics of the FORALL loop and ensures that the message vectorization algorithm will place all communication outside the loop. In addition, during code generation intrinsic functions are translated into calls to the run-time library. Parameters are added where necessary to provide necessary data partitioning information.

4.3.3 Sectioning

The final phase of the Fortran D back end completes the scalarization process. After partitioning is performed, the compiler applies *sectioning* to convert FORALL loops into DO loops [5, 6] in the node program. The Fortran D back end detects cases where temporary storage may be needed using data dependence analysis. True dependences carried on the FORALL loop represent instances where values are defined in the loop and used on later iterations; they point out where the copy-in/copy-out semantics of the FORALL loop is being violated.

Proc	*Time* (milliseconds)							
	ALL	ANY	MAXVAL	PRODUCT	DOT PRODUCT	TRANSPOSE		
	$1K \times 1K$	$1K \times 1K$	$1K \times 1K$	$256K$	$256K$	256×256	512×512	$1K \times 1K$
1	580.6	606.2	658.8	90.1	164.8	58	299	-
2	291.0	303.7	330.4	50.0	83.0	118	575	-
4	146.2	152.6	166.1	25.1	42.2	87	395	-
8	73.84	77.1	84.1	13.1	22.0	61	224	1039
16	37.9	39.4	43.4	7.2	12.1	41	140	539
32	19.9	20.7	23.2	4.2	7.4	36	85	316

Table 2: Performance of Some Fortran 90 Intrinsic Functions

During simple translation of Fortran 90 array constructs or FORALL loops, arrays involved in loop-carried true dependences must be saved in temporary buffers to preserve their old values. For instance, consider the translation of the following concise Fortran 90 formulation of the Jacobi algorithm:

```
A(2:N-1) = 0.5 * (A(1:N-2) + A(3:N))
        ⇓
FORALL i = 2,N-1
  A(i) = 0.5 * (A(i-1) + A(i+1))
ENDFOR
        ⇓
DO i = 1,N-2
  TMP(i) = A(i-1)
ENDDO
DO i = 2,N-1
  A(i) = 0.5 * (TMP(i) + A(i+1))
ENDFOR
```

A loop-carried true dependence exists between the definition to $A(i)$ and the use of $A(i-1)$. A temporary array TMP is needed so that the old values of $A(i-1)$ are not overwritten before they are used. The values of $A(i+1)$ do not need to be buffered since they are used before being redefined.

The previous example is problematic because temporary storage is required for the values of $A(i-1)$. In some cases, the Fortran D compiler can eliminate buffering through program transformations such as *loop reversal*. In other cases, the compiler can reduce the amount of temporary storage required through *data prefetching*[6]. For instance, in the Jacobi example a more efficient translation would result in:

```
X = A(1)
DO i = 2,N-1
  Y = 0.5 * (X + A(i+1))
  X = A(i)
  A(i) = Y
ENDFOR
```

This reduces the temporary memory required significantly, from an entire array to two scalars. For this version of Jacobi, we measured improvements of up to 50% for certain problem sizes on an Intel i860, as shown in Figure 2.

5 Run-time 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 run-time library instead to reduce the complexity and machine-dependence of the compiler.

The Fortran D compiler translates intrinsics into calls to run-time library routines using a standard interface. Additional information is passed describing bounds, overlaps, and partitioning for each array dimension. The run-time library is built on top of the Express communication package to ensure portability across different architectures [30].

Table 2 presents some sample performance numbers for a subset of the intrinsic functions on an iPSC/860, details are presented elsewhere [2]. The times in the table include both the computation and communication times for each function. For large problem sizes, we were able to obtain almost linear speedups. In the case of TRANSPOSE function, going from one processor to two or four degrades execution time due to increased communication. However, speedup improves as the number of processors increases.

6 Fortran 90D Compilation Example

In this section we demonstrate how an example Fortran 90D program is compiled into message-passing Fortran 77, then measure its performance.

6.1 Compilation

Figure 3 shows a code fragment implementing one sweep of ADI integration on a 2D mesh, a typical (if short) numerical algorithm. Conceptually, the code is solving a tridiagonal system (represented by the arrays A and B) along each row of the matrix X. The tridiagonal systems are solved by a sequential method, but separate columns are independent and may be solved in parallel. The full version of ADI integration sweeps each dimension of the mesh, preventing completely parallel execution for any static data decomposition.

In the example, Fortran D data decomposition statements are used to partition the 2D array into blocks of columns. For clarity, we declare the number of processors (N$PROC) to be 32 at compile time. The Fortran 90D example is concise and convenient for the user, since it can be written for a single address space without requiring explicit communication. However, additional compilation techniques are required to generate efficient code. First, the Fortran 90D front end translates the program into intermediate form as shown in Figure 4, converting all array constructs into FORALL loops. Since no true dependences are carried on the FORALL loops, they may be directly replaced with DO loops.

The compilation process for the Fortran D back end merits closer examination. First, array bounds are reduced to the local sections plus overlaps. The local processor number is determined using *myproc*(), a library function; it is used to compute expressions for reducing loop bounds. Analysis determines that both I and J are *cross-processor* loops—loops carrying true dependences that sequentialize the computation across processors. To exploit pipeline parallelism, the Fortran D compiler interchanges such loops inward. We call this technique *fine-grain pipelining* [20, 21].

For this version of ADI integration, data dependences permit the Fortran D compiler to interchange the J loop inwards. However, if loop fusion is not performed, the imperfectly nested K loops inhibit loop interchange for loop I, forcing it to remain in place. During code generation, true dependences for nonlocal references carried on the I and J loop cause calls to *send* and *recv* to be

```
PARAMETER (N = 512, 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(1:N,I) = X(1:N,I) - X(1:N,I-1)*A(1:N,I)/B(1:N,I-1)
  B(1:N,I) = B(1:N,I) - A(1:N,I)*A(1:N,I)/B(1:N,I-1)
ENDDO
X(1:N,N) = X(1:N,N) / B(1:N,N)
DO J = N-1,1,-1
  X(1:N,J) = (X(1:N,J)-A(1:N,J+1)*X(1:N,J+1))/B(1:N,J)
ENDDO
```

Figure 3: ADI integration in Fortran 90D

inserted to provide communication and synchronization. Figure 5 shows the resulting program.[1] Unfortunately, the computation in the I loop has been sequentialized, since each processor has to wait for its predecessor to complete. Note that this is not due to communication placement; the values needed by the succeeding processor are simply computed last.

If loop fusion is enabled, the Fortran D back end will fuse the two inner K loops. This is legal because the dependence between the definition and use of B is carried on the I loop and is thus unaffected. Fusion is also conservative because it does not introduce any true dependences carried by the K loop. Fusing the K loops promotes reuse of A and B, but its main benefit is to enable the Fortran D back end to interchange the I and K loops, exposing pipeline parallelism. The resulting program is displayed in Figure 6. For simplicity, only the first loop is shown. The remaining loops are compiled in a similar manner as before.

To reduce communication overhead, we can also apply strip-mining in conjunction with loop interchange to adjust the granularity of pipelining. We call this technique *coarse-grain pipelining* [20, 21]. In the ADI example, we strip-mine the K loop by four (an empirically derived value), then interchange the resulting loop outside the I loop. Messages inserted outside the K loop allow each processor to reduce communication costs at the expense of some parallelism, resulting in Figure 7. Except for coarse-grain pipelining, all these versions of ADI integration were generated automatically by the Fortran D compiler.

6.2 Performance Results

To validate these methods, we executed these codes on an iPSC/860. The programs were compiled under -O4 using Release 3.0 of *if77*, the iPSC/860 compiler. Timings were taken for three double-precision problem sizes using *dclock*() on a 32 node Intel iPSC/860 with 8 Meg of memory per node. Results are shown using log scale in Figure 8. Timings are not provided where problem size exceeds available memory.

The original version of ADI (Figure 5) exploits pipeline parallelism in the J loop, but shows limited speedup, since the I loop is sequentialized. Fusing the K loops to improve memory reuse provides very little improvement in this case, yielding nearly identical results. Applying loop interchange after fusion to enable fine-grain pipelining (Figure 6) parallelizes the I loop as well, yielding significant speedup. Strip-mining to apply coarse-grain pipelining can improve efficiency an additional 10-50% (Figure 7). Pipelining comes closest to perfect speedup for large problems on a few processors.

We also compared the efficiency of pipelining versus dynamic data decomposition. By changing the distribu-

[1]Many details in the example programs have been elided or simplified; however, they are precisely equivalent to code generated and executed on the iPSC/860.

```
PARAMETER (N = 512)
REAL X(N,N), A(N,N), B(N,N)
DO I = 2,N
  FORALL K = 1,N
    X(K,I) = X(K,I) - X(K,I-1)*A(K,I)/B(K,I-1)
  ENDFOR
  FORALL K = 1,N
    B(K,I) = B(K,I) - A(K,I)*A(K,I)/B(K,I-1)
  ENDFOR
ENDDO
FORALL K = 1,N
  X(K,N) = X(K,N)/B(K,N)
ENDFOR
DO J = N-1,1,-1
  FORALL K = 1,N
    X(K,J) = (X(K,J)-A(K,J+1)*X(K,J+1))/B(K,J)
  ENDFOR
ENDDO
```

Figure 4: ADI in Intermediate Form

```
REAL X(512,0:17), A(512,17), B(512,0:16)
MY$P = myproc()      {* 0...31 *}
LB1 = MAX((MY$P*16)+1,2) - MY$P*16
UB1 = MIN((MY$P+1)*16,511) - MY$P*16
IF (MY$P .GT. 0) recv(X(1:N,0),B(1:N,0),MY$P-1)
DO I = LB1, 16
  DO K = 1,N
    X(K,I) = X(K,I) - X(K,I-1)*A(K,I)/B(K,I-1)
  ENDDO
  DO K = 1,N
    B(K,I) = B(K,I) - A(K,I)*A(K,I)/B(K,I-1)
  ENDDO
ENDDO
IF (MY$P .LT. 31) send(X(1:N,16),B(1:N,16),MY$P+1)
IF (MY$P .EQ. 31) THEN
  DO K = 1,N
    X(K,16) = X(K,16)/B(K,16)
  ENDDO
ENDIF
IF (MY$P .GT. 0) send(A(1:N,1),MY$P-1)
IF (MY$P .LT. 31) recv(A(1:N,17),MY$P+1)
DO K = 1,N
  IF (MY$P .LT. 31) recv(X(K,17),MY$P+1)
  DO J = UB1,1,-1
    X(K,J) = (X(K,J)-A(K,J+1)*X(K,J+1))/B(K,J)
  ENDDO
  IF (MY$P .GT. 0) send(X(K,1),MY$P-1)
ENDDO
```

Figure 5: ADI without Loop Fusion

```
REAL X(512,0:17), A(512,17), B(512,0:16)
MY$P = myproc()      {* 0...31 *}
LB1 = MAX((MY$P*16)+1,2) - MY$P*16
DO K = 1,N
  IF (MY$P .GT. 0) recv(X(K,0),B(K,0),MY$P-1)
  DO I = LB1,16
    X(K,I) = X(K,I) - X(K,I-1)*A(K,I)/B(K,I-1)
    B(K,I) = B(K,I) - A(K,I)*A(K,I)/B(K,I-1)
  ENDDO
  IF (MY$P .LT. 31) send(X(K,16),B(K,16),MY$P+1)
ENDDO
```

Figure 6: ADI with Fine-grain Pipelining

```
REAL X(512,0:17), A(512,17), B(512,0:16)
MY$P = myproc()      {* 0...31 *}
LB1 = MAX((MY$P*16)+1,2) - MY$P*16
DO KK = 1,N,4
  IF (MY$P .GT. 0) THEN
    recv(X(KK:KK+3,0),B(KK:KK+3,0),MY$P-1)
  ENDIF
  DO I = LB1,16
    DO K = KK,KK+3
      X(K,I) = X(K,I) - X(K,I-1)*A(K,I)/B(K,I-1)
      B(K,I) = B(K,I) - A(K,I)*A(K,I)/B(K,I-1)
    ENDDO
  ENDDO
  IF (MY$P .LT. 31) THEN
    send(X(KK:KK+3,16),B(KK:KK+3,16),MY$P+1)
  ENDIF
ENDDO
```

Figure 7: ADI with Coarse-grain Pipelining

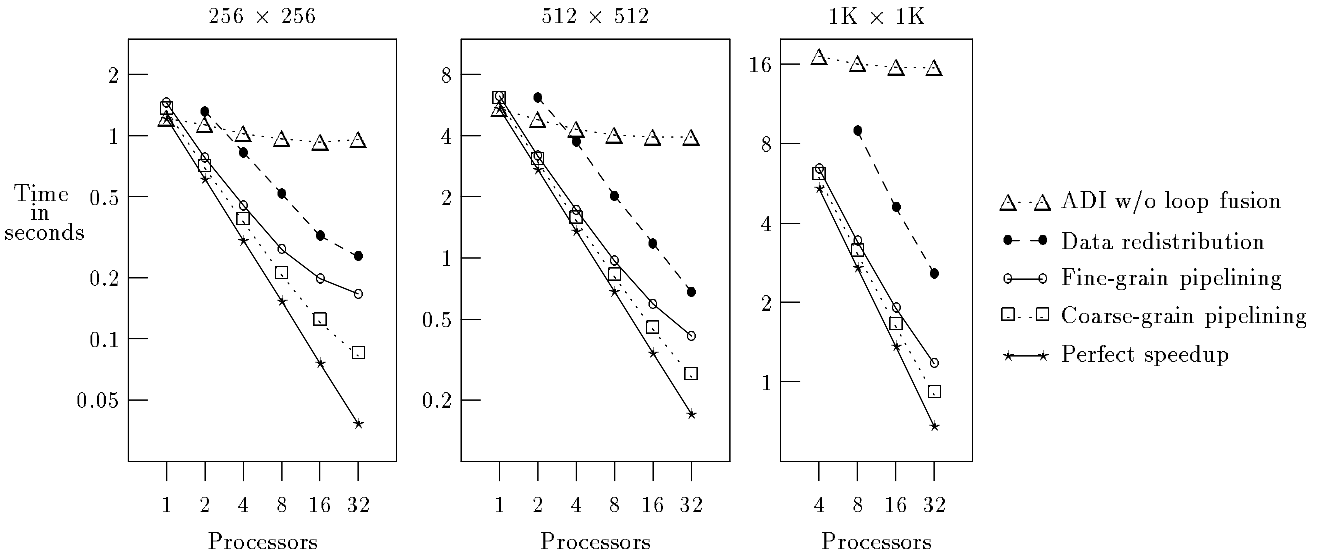

Figure 8: Performance of ADI Integration (double precision)

tion of data at run-time from columns to rows, all dependences in each sweep of ADI may be internalized, enabling completely parallel execution. Data must be redistributed twice, once to achieve the desired distribution, then a second time to return it to its original configuration. The cost of redistributing is approximated by the performance of the TRANSPOSE routine shown in Table 2.

Our results show that on the iPSC/860, dynamic data decomposition for this formulation of ADI integration achieves speedup. However, the resulting program is significantly slower than pipelining, even for small problems distributed over large numbers of processors, the expected best case for dynamic data decomposition. Our experiences show that some common algorithms, such as ADI integration, require significant amounts of optimization 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 [22, 25] or parallel procedures [18, 32], while others rely on functional language [27] or single assignment semantics [31]. In comparison, the Fortran D compiler uses dependence analysis to automatically exploit parallelism and extract communication even from sequential loops such as those found in ADI integration.

Several other projects are also developing Fortran 90 compilers for MIMD distributed-memory machines. ADAPT proposes to scalarize and partition Fortran 90 programs using a run-time library for Fortran 90 intrinsics [29]. The CM FORTRAN compiler compiles Fortran 90 with alignment and layout specifications directly to the physical machine, and can optimize floating point register usage [3]. The FORTRAN-90-Y compiler uses formal specification techniques to generate efficient code for the CM-2 and CM-5 [15]. PARAGON is a version of C extended with array syntax, operations, reductions, permutations, and distribution specifications [14]. Our compiler resembles the VIENNA FORTRAN 90 compiler derived from SUPERB [11, 38] and has also been influenced by a proposal by Wu & Fox that discussed program generation and optimization [37].

A number of researchers have studied techniques to reduce storage and promote memory reuse [1, 5, 6, 24, 26, 28, 33, 35]. These optimizations have proved useful for both scalar and parallel machines. The goal of the Fortran 90D compiler is to integrate these scalarization techniques with advanced communication and parallelism optimizations.

8 Conclusions

This paper presents an integrated approach to compiling both Fortran 77D and 90D based on a few key observations. First, using FORALL preserves information in Fortran 90 array constructs. Dividing the scalarization process into translation, loop fusion, and sectioning allows it to be easily integrated with the partitioning performed by the Fortran D compiler. A portable run-time library can also reduce the complexity and machine-dependence of the compiler. All optimizations except coarse-grain pipelining and data prefetching have been implemented in the current Fortran D compiler prototype.

Compiling 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 [9, 10, 19, 23].

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 this information does not necessarily reflect the position or policy of the government, and no official endorsement should be inferred.

References

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1979.

[2] I. Ahmad, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, and R. Thakur. Implementation and scalability of Fortran 90D intrinsic functions on distributed memory machines. Technical Report SCCS-256, NPAC, Syracuse University, March 1992.

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

[4] E. Albert, J. Lukas, and G. Steele, Jr. Data parallel computers and the FORALL statement. *Journal of Parallel and Distributed Computing*, 13(4):185–192, August 1991.

[5] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, April 1983.

[6] J. R. Allen and K. Kennedy. Vector register allocation. Technical Report TR86-45, Dept. of Computer Science, Rice University, December 1986.

[7] 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.

[8] ANSI X3J3/S8.115. Fortran 90, June 1990.

[9] 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.

[10] 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.

[11] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[12] 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.

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

[14] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[15] M. Chen and J. Cowie. Prototyping Fortran-90 compilers for massively parallel machines. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[16] 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.

[17] A. Goldberg and R. Paige. Stream processing. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 228–234, August 1984.

[18] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[19] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[20] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[21] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[22] 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.

[23] K. Kennedy and U. Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report TR91-155, Dept. of Computer Science, Rice University, April 1991.

[24] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[25] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

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

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

[28] K. S. McKinley. *Automatic and Interactive Parallelization.* PhD thesis, Rice University, April 1992.

[29] J. Merlin. ADAPTing Fortran-90 array programs for distributed memory architectures. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.

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

[31] 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.

[32] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, September 1991.

[33] V. Sarkar and G. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

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

[35] J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, January 1984.

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

[37] M. Wu and G. Fox. Compiling Fortran90 programs for distributed memory MIMD parallel computers. CRPC Report CRPC-TR91126, Center for Research on Parallel Computation, Syracuse University, January 1991.

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