Preliminary Experiences with the Fortran D Compiler

Seema Hiranandani
Ken Kennedy
Chau-Wen Tseng

CRPC-TR93307
April 1993

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

Preliminary Experiences with the Fortran D Compiler

Seema Hiranandani Ken Kennedy Chau-Wen Tseng
seema@cs.rice.edu *ken@cs.rice.edu* *tseng@cs.rice.edu*

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251-1892
Tel: (713) 527-6077
Fax: (713) 285-5136

Abstract

Fortran D is version of Fortran enhanced with data decomposition specifications. Case studies illustrate strengths and weaknesses of the prototype Fortran D compiler when compiling linear algebra codes and whole programs. Statement groups, execution conditions, inter-loop communication optimizations, and array kills for replicated arrays are identified as new compilation issues. On the Intel iPSC/860, the output of the prototype Fortran D compiler approaches the performance of hand-optimized code for parallel computations, but needs improvement for linear algebra and pipelined codes. The Fortran D compiler outperforms the CM Fortran compiler (2.1 beta) by a factor of four or more on the TMC CM-5 when not using vector units. We find that that the success of the prototype compiler is linked to the ratio of communication to computation inherent in the computation; the compiler most closely approaches the performance of hand-optimized code when communication overhead is a small percentage of total execution time.

1 Introduction

Fortran D is an enhanced version of Fortran that allows the user to specify how data may be partitioned onto processors. It was inspired by the observation that modern high-performance architectures demand that careful attention be paid to data placement by both the programmer and compiler. Fortran D is designed to provide a simple yet efficient machine-independent data-parallel programming model that shifts much of the burden of machine-dependent optimizations to the compiler. It has contributed to the development of High Performance Fortran (HPF), an informal Fortran standard adopted by researchers and vendors for programming massively-parallel multiprocessors [16].

The success of HPF hinges on the development of compilers that can provide performance satisfactory to users. The goal of our research with the Fortran D compiler is identify important compilation issues and explore possible solutions. Previous work has described the design and implementation of a prototype Fortran D compiler for regular dense-matrix computations [14, 17, 18]. This paper describes our preliminary experiences with that compiler. Its major contributions include 1) advanced compilation techniques needed for complex loop nests, 2) empirical comparison of the prototype Fortran D compiler against hand-optimized code on the Intel iPSC/860 and the CM Fortran compiler on the TMC CM-5, and 3) identifying the connection between computation and compiler performance.

In the remainder of this paper, we briefly introduce the Fortran D language and illustrate how Fortran D programs are compiled. We use case studies to introduce a number of compilation problems and their solutions. To evaluate the efficiency of the Fortran D compiler, we compare the performance of its output for these programs and some kernels against hand-optimized versions on the Intel iPSC/860. We also compare

the Fortran D and CM Fortran compilers on the Thinking Machines CM-5 by translating representative kernels into CM Fortran. Results are discussed and used to point out directions for future research. We conclude with a comparison with related work.

2 Background

2.1 Fortran D Language

In Fortran D, the DECOMPOSITION statement declares an abstract problem or index domain. The ALIGN statement maps each array element onto the decomposition. The DISTRIBUTE statement groups elements of the decomposition and aligned arrays, mapping them to a parallel machine. Each dimension 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 Basic Fortran D Compilation

Given a data decomposition, the Fortran D compiler automatically translates sequential programs into efficient parallel programs. The two major steps in compiling for MIMD distributed-memory machines are partitioning the data and computation across processors, then introducing communication for nonlocal accesses where needed. The compiler applies a compilation strategy based on data dependence that incorporates and extends previous techniques. We briefly describe each major step of the compilation process below, details are presented elsewhere [14, 17, 18]:

1. **Analyze Program.** The Fortran D compiler performs scalar dataflow analysis, symbolic analysis, and dependence testing to determine the type and level of all data dependences.

2. **Partition data.** The compiler analyzes Fortran D data decomposition specifications to determine the decomposition of each array in a program. Alignment and distribution statements are used to calculate the array section owned by each processor.

3. **Partition computation.** The compiler partitions computation across processors using the "owner computes" rule—where each processor only computes values of data it owns [5, 26, 32]. The left-hand side (*lhs*) of each assignment statement is used to calculate its *local iteration set*, the set of loop iterations that cause a processor to assign to local data.

4. **Analyze communication.** Based on the computation partition, references that result in nonlocal accesses are marked.

5. **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 [3, 32].

6. **Manage storage.** Buffers or "overlaps" [32] created by extending the local array bounds are allocated to store nonlocal data.

7. **Generate code.** The compiler *instantiates* the communication, data and computation partition determined previously, generating the SPMD program with explicit message-passing that executes directly on the nodes of the distributed-memory machine.

We refer to collections of data and computation as *index sets* and *iteration sets*, respectively. In the Fortran D compiler, both are described using Fortran 90 triplet notation.

2.3 Prototype Compiler

The prototype Fortran D compiler is implemented as a source-to-source Fortran translator in the context of the ParaScope parallel programming environment [4, 8]. It utilizes existing tools for performing dependence analysis, program transformations, and interprocedural analysis [9, 13, 20]. The current implementation supports:

- inter-dimensional alignments
- 1D BLOCK and CYCLIC distributions
- loop interchange, fusion, distribution, strip-mining
- message vectorization, coalescing, aggregation
- vector message pipelining
- broadcasts, collective communications, point-to-point messages
- SUM, PRODUCT, MIN, MAX, MINLOC, MAXLOC reductions
- fine-grain and coarse-grain pipelining (preset granularity)
- relax "owner computes rule" for reductions, private variables
- nonlocal storage in overlaps, buffers
- loop bounds reduction, guard introduction
- global $\leftrightarrow$ local index conversion
- interprocedural reaching decompositions, overlap offsets
- common blocks
- I/O (performed by processor 0)
- generation of calls to the Intel NX/2 message-passing library

For simplicity, the prototype compiler requires that all array sizes, loop bounds, and number of processors in the target machine to be compile-time constants. All subscripts must also be of the form c or $i+c$, where c is a compile-time constant and i is a loop index variable. These restrictions are not due to limitations of our compilation techniques, but reflect the immaturity of the prototype compiler.

3 Compilation Case Studies

Examples in previous work mostly dealt with individual stencil computation kernels from iterative partial difference equation (PDE) solvers. In this section, we illustrate the Fortran D compilation process for linear algebra kernels, large subroutines, and whole programs. We point out strengths and weaknesses of the prototype compiler using case studies of four example programs and subroutines: DGEFA, SHALLOW, DISPER, and ERLEBACHER. For these more complex codes, we find that the Fortran D compiler needs to:

- Robust translation of global/local loop bounds and index variables. Re-indexing accesses into temporary buffers.
- Partition computation in complex non-uniform loop bodies across processors, using statement groups to guide loop bounds & index variable generation. Apply loop distribution and guard generation as needed.
- Compile loop nests containing execution conditions that may affect the iteration space.
- Exploit pipeline parallelism, perform inter-loop communication optimizations.
- Use array kill analysis to eliminate communication for multi-reductions performed by replicated array variables.

3.1 DGEFA

We begin with DGEFA, a key subroutine in LINPACK written by Jack Dongarra *et al.* at Argonne National Laboratory. It is also the principal computation kernel in the LINPACKD benchmark program. DGEFA performs LU decomposition through Gaussian elimination with partial pivoting. Its memory access patterns are quite different from stencil computations, and is representative of linear algebra computations. As many linear algebra algorithms involve factoring matrices, CYCLIC and BLOCK_CYCLIC data distributions are desirable for maintaining good load balance. These distributions and the prevalence of triangular loop nests pose additional challenges to the Fortran D compiler.

Figure 1 shows the original program as well as the output produced by the prototype Fortran D compiler. For good load balance we choose a column-cyclic distribution, scattering array columns round-robin across processors. The Fortran D compiler then uses this data decomposition to derive the computation partition. Two important steps are generating proper loop bounds & indices and indexing accesses into temporary messages buffers; both techniques are described elsewhere [31]. In addition, we found statement groups and identifying MAX/MAXLOC reductions to be necessary.

3.1.1 Statement Groups

Whole programs and linear algebra codes tend to possess large diverse loop nests, with many imperfectly nested statements and triangular/trapezoidal loops. These complex loops increase the difficulty of partitioning the computation and calculating appropriate local and global loop indices and bounds. Recall that an iteration set represents the set of loop iterations that will be executed by each processor. The Fortran D compiler partitions computation by modifying loop bounds to the union of all iteration sets of statements in the a loop, then inserting explicit guards for statements that are executed only on a subset of those iterations.

To aid in this process, we found it useful in the Fortran D compiler to partition statements into *statement groups* during partitioning analysis. Statements are put into the same group for a given loop if their iteration sets for that loop and enclosing loops are the same. We mark a loop as *uniform* if all its statements belong to the same statement group. Uniform loop nests are desirable because they may be partitioned by reducing loop bounds; no explicit guards need to be inserted in the loop. Calculating statement groups can determine whether a loop nest is uniform and guide code generation for non-uniform loops.

An immediate application of statement groups is *loop distribution*, a program transformation that separates independent statements inside a single loop into multiple loops with identical headers. If the Fortran D compiler detects a non-uniform loop nest, it attempts to distribute the loop around each statement group, producing smaller uniform loop nests. If loop distribution is prevented due to recurrences carried by the loop, the Fortran D compiler must insert explicit guards for each statement group to ensure they are executed only by the appropriate processor(s) on each loop iteration. Statement groups also help because they identify groups of statements that can share the same guard expression.

```
{* Original Fortran D Program *}
SUBROUTINE DGEFA(n,a,ipvt)
  INTEGER n,ipvt(n),j,k,l
  DOUBLE PRECISION a(n,n),al,t
  DISTRIBUTE a(:,CYCLIC)
  do k = 1, n-1
    {* Find max element in a(k:n,k) *}
S1  l = k
S2  al = dabs(a(k, k))
    do i = k + 1, n
      if (dabs(a(i, k)) .GT. al) then
S3      al = dabs(a(i, k))
S4      l = i
      endif
    enddo
S5  ipvt(k) = l
    if (al .NE. 0) then
S6    if (l .NE. k) then
        t = a(l,k)
        a(l,k) = a(k,k)
        a(k,k) = t
      endif
      {* Compute multipliers in a(k+1:n,k) *}
      t = -1.0d0/a(k,k)
      do i = k+1, n
        a(i, k) = a(i, k) * t
S7    enddo
      {* Reduce remaining submatrix *}
S8    do j = k+1, n
        t = a(l,j)
        if (l .NE. k) then
          a(l,j) = a(k,j)
          a(k,j) = t
        endif
        do i = k+1, n
          a(i, j) = a(i, j) + t*a(i, k)
        enddo
S9    enddo
    endif
  enddo
S10 ipvt(n) = n
end
```

```
{* Compiler Output for 4 Processors *}
SUBROUTINE DGEFA(n,a,ipvt)
  INTEGER n,ipvt(n),j,k,l
  DOUBLE PRECISION a(n,n/4),al,t,dp$buf1(n)
  do k = 1, n-1
    k$ = ((k - 1) / 4) + 1
    {* Find max element in a(k:n,k$) *}
    if (my$p .EQ. MOD(k - 1, 4)) then
      l = k
      al = dabs(a(k, k$))
      do i = k + 1, n
        if (dabs(a(i, k$)) .GT. al) then
          al = dabs(a(i, k$))
          l = i
        endif
      enddo
      broadcast l, al
    else
      recv l, al
    endif
    ipvt(k) = l
    if (al .NE. 0) then
      if (my$p .EQ. MOD(k - 1, 4)) then
        if (l .NE. k$) then
          t = a(l,k$)
          a(l,k$) = a(k,k$)
          a(k,k$) = t
        endif
        {* Compute multipliers in a(k+1:n,k$) *}
        t = -1.0d0/a(k,k$)
        do i = k+1, n
          a(i, k$) = a(i, k$) * t
        enddo
      endif
      {* Reduce remaining submatrix *}
      if (my$p .EQ. MOD(k - 1, 4)) then
        buffer a(k+1:n, k$) into dp$buf1
        broadcast dp$buf1(1:n-k)
      else
        recv dp$buf1(1:n-k)
      endif
      lb$1 = (k / 4) + 1
      if (my$p .LT. MOD(k, 4)) lb$1 = lb$1+1
      do j = lb$1, n
        t = a(l,j)
        if (l .NE. k) then
          a(l,j) = a(k,j)
          a(k,j) = t
        endif
        do i = k+1, n
          a(i, j) = a(i, j) + t*dp$buf1(i-k)
        enddo
      enddo
    endif
  enddo
  ipvt(n) = n
end
```

Figure 1 DGEFA: Gaussian Elimination with Partial Pivoting

DGEFA is a prime example of how statement groups work. During compilation, the Fortran D compiler partitions the statements of the loop body into five statement groups. The first statement group ($S_1 - -S_4$) finds the pivot, and is executed by one processor per iteration of the k loop. The second group is the statement S_5 and is executed by all processors. The third group ($S_6 - -S_7$) calculates multipliers. Like the first group, it is executed by only one processor. The fourth statement group ($S_8 - -S_9$) calculates the remaining submatrix, and is executed by all processors. The fifth and final group (S_{10}) is also executed by all. Because loop k contains two variety of iteration sets, it is non-uniform. Its iterations are executed by all processors, and explict guards are introduced for the first and third statement groups.

3.1.2 MIN/MAX and MINLOC/MAX Reductions

Putting statements S_1 through S_4 in the same statement group requires detecting it as a reduction. The Fortran D compiler recognizes it as a MAX/MAXLOC reduction by detecting that the *lhs* of an assignment *al* at statement S_3 is being compared against its *rhs* in an enclosing IF statement. The level of the reduction is set to the k loop, since it is the deepest loop enclosing a use of *al*. The reduction is thus carried out by the i loop, which only examines a single column of a. Since array a has been distributed by columns, the reduction may be computed locally by the processor owning the column. The Fortran D compiler inserts a guard to ensure the reduction is performed by the processor owning column k, then broadcasts the result. This is also an example of how the compiler relaxes the owner computes rule for reductions and private variables.

For MIN/MAX and MINLOC/MAXLOC reductions, the Fortran D compiler must also search for initialization statements for the *lhs* of assignment statements in the k loop, assigning them the same iteration set as the body of the reduction. Statements S_1 and S_2 are identified as initialization statements for the MAX/MAXLOC reduction at S_3. By putting them in the same statement group as the reduction, the Fortran D compiler avoids inserting an additional broadcast to update the value of *al* at S_2.

3.2 SHALLOW

SHALLOW is a 200 line benchmark weather prediction program written by Paul Swarztrauber, National Center for Atmospheric Research (NCAR). It is a stencil computation that applies finite-difference methods to solve shallow-water equations. SHALLOW is representative of a large class of existing supercomputer applications. The computation is highly data-parallel and well-suited for MIMD distributed-memory machines.

Figure 2 outlines the version of SHALLOW we used to test the Fortran D compiler; it was modified to eliminate I/O. Data can be partitioned quite simply by aligning all 2D arrays identically, then distributing the result column-wise. We chose to block distribute the second dimension, assigning a block of columns to each processor. The prototype Fortran D compiler was able to generate message-passing code fairly simply. The principal issues encountered during compilation were boundary conditions, loop distribution, and inter-loop communication optimizations.

3.2.1 Boundary Conditions

SHALLOW contains many code fragments solving boundary conditions for periodic continuations. As a result, the Fortran D compiler needed to insert explicit guards for many statement groups. These boundary conditions also required the creation of several individual point-to-point messages between boundary processors to transfer data required.

```
{* Original Fortran D Program *}
PROGRAM SHALLOW
  REAL u(N,N),v(N,N),p(N,N),unew(N,N),pnew(N,N),vnew(N,N),psi(N,N)
  REAL pold(N,N),uold(N,N),vold(N,N),cu(N,N),cv(N,N),z(N,N),h(N,N)
  DECOMPOSITION d(N,N)
  ALIGN u,v,p,unew,pnew,vnew,psi,pold,uold,vold,cu,cv,z,h WITH d
  DISTRIBUTE d(:,BLOCK)
  {* initial values of the stream function & velocities *}
  do j = 1,N-1
    do i = 1,N-1
      u(i+1,j) = -(psi(i+1,j+1)-psi(i+1,j))*dy
      v(i,j+1) = (psi(i+1,j+1)-psi(i,j+1))*dx
    enddo
  enddo
  do k = 1,Time
    {* periodic continuation *}
    ...
    {* compute capital u, capital v, z, and h *}
    do j = 1,N-1
      do i = 1,N-1
        cu(i+1,j) = .5*(p(i+1,j)+p(i,j))*u(i+1,j)
        cv(i,j+1) = .5*(p(i,j+1)+p(i,j))*v(i,j+1)
        z(i+1,j+1) = (fsdx*(v(i+1,j+1)-v(i,j+1))-fsdy*(u(i+1,j+1)
              -u(i+1,j))) / (p(i,j)+p(i+1,j) +p(i+1,j+1)+p(i,j+1))
        h(i,j) = p(i,j)+.25*(u(i+1,j)*u(i+1,j)+u(i,j)*u(i,j)+v(i,j+1)
              *v(i,j+1)+v(i,j)*v(i,j))
        enddo
    enddo
    {* periodic continuation *}
    ...
    {* compute new values u, v, and p *}
    do j = 1,N-1
      do i = 1,N-1
        unew(i+1,j) = uold(i+1,j)+tdts8*(z(i+1,j+1)+z(i+1,j))*(cv(i+1,j+1)
              +cv(i,j+1)+cv(i,j)+cv(i+1,j))-tdtsdx*(h(i+1,j)-h(i,j))
        vnew(i,j+1) = vold(i,j+1)-tdts8*(z(i+1,j+1) +z(i,j+1))*(cu(i+1,j+1)
              +cu(i,j+1)+cu(i,j)+cu(i+1,j))-tdtsdy*(h(i,j+1)-h(i,j))
        pnew(i,j) = pold(i,j)-tdtsdx*(cu(i+1,j)-cu(i,j))
              -tdtsdy*(cv(i,j+1)-cv(i,j))
      enddo
    enddo
  enddo
end
```

Figure 2 SHALLOW: Weather Prediction Benchmark

3.2.2 Loop Distribution

Because of the programming style used in writing SHALLOW, almost all loop nests were non-uniform, *i.e.*, contained statements with differing iteration sets. Fortunately, none of the loops carried recurrences, so the Fortran D compiler applies loop distribution to separate statements, creating uniform loop nests. Loop bounds reduction is then sufficient to partition the computation during code generation, excepting boundary conditions.

3.2.3 Inter-loop Message Coalescing and Aggregation

While loop distribution enables inexpensive partitioning of the program computation, it has the disadvantage of creating a large number of loop nests. In many cases these loop nests, along with loops representing boundary conditions, required communication with neighboring processors. The current Fortran D compiler prototype applies message coalescing and aggregation only within a single loop nest. Its output for SHALLOW thus missed many opportunities to coalesce or aggregate messages because the nonlocal references were located in loop nests not enclosed by a common loop. By applying message coalescing and aggregation manually across loop nests, we were able to eliminate about half of all calls to communication routines.

```
{* Original Fortran D Program *}
SUBROUTINE DISPER
  LOGICAL lsat(256)
  DOUBLE PRECISION ddx(256,8,8), ddy(256,8,8), ddz(256,8,8)
  DOUBLE PRECISION pmfr(256,8,8,4,5), gradx(256), grady(256), gradz(256)
  DECOMPOSITION d(256)
  ALIGN ddx(i,j,k),ddy(i,j,k),ddz(i,j,k) WITH d(i)
  ALIGN lsat(i,j,k,l),pmfr(i,j,k,l,m) WITH d(i)
  ALIGN gradz,grady,gradz WITH d
  DISTRIBUTE d(BLOCK)
  {* compute dispersion terms *}
  do j = 2,4
    do i3 = 1,8
      do i2 = 1,8
        do i1 = 1,256
          ...
S1        if ((i1 .NE. 1) .AND. (i1 .NE. 256)) then
S2          if (lsat(i1-1,i2,i3,j) .AND. lsat(i1+1,i2,i3,j)) then
S3            grady(i1)=(pmfr(i1+1,i2,i3,j,k)-pmfr(i1-1,i2,i3,j,k)) /
                (.5 * (ddy(i1+1,i2,i3) + ddy(i1-1,i2,i3)) + ddy(i1,i2,i3))
              ...
            endif
          endif
          ...
        enddo
      enddo
    enddo
  enddo
end
```

Figure 3 DISPER: Oil Reservoir Simulation

3.3 DISPER

DISPER is a 1000 line subroutine for computing dispersion terms. It is taken from UTCOMP, a 33,000 line oil reservoir simulator developed at the University of Texas at Austin. Like SHALLOW, DISPER is a stencil computation that is highly data-parallel and well-suited for the Fortran D compiler. Unfortunately, UTCOMP was originally written for a Cray vector machine. Arrays were linearized to ensure long vector lengths, then addressed through complex subscript expressions and indirection arrays. This style of programming, while efficient for vector machines, does not lend itself to massively-parallel machines.

To explore whether UTCOMP can be written in a machine-independent programming style using Fortran D or HPF, researchers at Rice rewrote DISPER to have regular accesses and simple subscripts on multidimensional arrays. Figure 3 shows a fragment of the rewritten form of DISPER. Its main arrays have differing sizes and dimensionality, but have the same size in the first dimension. Arrays were aligned along the first dimension and distributed block-wise. The resulting code was was for the most part compiled successfully by the prototype Fortran D compiler.

3.3.1 Execution Conditions

The major difficulty encountered by the Fortran D compiler was the existence of execution conditions caused by explicit guards in the input code. There are two types of execution conditions. Data-dependent execution conditions, such as the guard at S_2 in Figure 3, were not a problem. Message vectorization moves communication caused by such guarded statements out of the enclosing loops. Overcommunication may result if the statement is not executed, but the resulting code is still much more efficient than sending individual messages after evaluating each guard.

Execution conditions that reshape the iteration space, on the other hand, pose a significant problem. For instance, the guard at S_1 in Figure 3 restricts the execution of statement S_3 on the first and last iteration of loop $i1$. It has in effect changed the iteration set for the assignment S_3, causing it to be executed on a

subset of the iterations. These guards are frequently used by programmers to isolate boundary conditions in a modular manner, avoiding the need to peel off loop iterations.

Unlike data-dependent execution conditions, these execution conditions always hold and can be detected at compile-time. If they are not considered, the compiler will generate communication for nonlocal accesses that never occur. Future versions of the Fortran D compiler will need to examine guard expressions. If its effects on the iteration set can be determined at compile-time, the iteration set of the guarded statements must be modified appropriately. Because this functionality is not present in the current Fortran D compiler, unnecessary guards and communication in the compiler output were corrected by hand.

3.4 ERLEBACHER

ERLEBACHER is a 800 line benchmark program written by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). It performs 3D tridiagonal solves using Alternating-Direction-Implicit (ADI) integration. Like Jacobi iteration and Successive-Over-Relaxation (SOR), ADI integration is a technique frequently used to solve PDEs. However, it performs vectorized tridiagonal solves in each dimension, resulting in computation wavefronts across all three dimensions of the data array.

Each sweep in ERLEBACHER consists of a set-up and computation phase, followed by forward and backward substitutions. Figures 4 and 5 illustrate the core computation performed by ERLEBACHER during a sweep of the Z dimension. We chose to distribute the Z dimension of all 3D arrays blockwise; all 1D and 2D arrays are replicated. Here we relate some issues that arose during compilation of Erlebacher to a machine with four processors, $P_0 \ldots P_3$.

3.4.1 Overlapping Communication with Computation

In ERLEBACHER, we discovered unexpected benefits for vector message pipelining, an optimization that separates matching *send* and *recv* statements to create opportunities for overlapping communication with computation [18]. Consider compilation of the setup phase in the Z dimension, shown in Figure 4. The Fortran D compiler first distributes the loops enclosing statements $S_1 \ldots S_4$ because they belong to two distinct statement groups. Message vectorization then extracts all communication outside of each loop nest. The Fortran D compiler then applies vector message pipelining.

We found vector message pipelining to be particularly effective here because it moves the *sends* C_3 and C_4 before the *recvs* in the first two loop nests. If C_3 and C_4 are left in their original positions before S_5, the computation will be idle until two message transfers complete, because the boundary processors P_0 and P_3 will need to first exchange messages before communicating to the interior processors. The prototype thus saved the cost of waiting for an entire message. More advanced analysis could determine that the statements $S_1 \ldots S_4$ are simply incarnations of statement S_5 created to handle periodic boundary conditions. We can perform the reverse of *index set splitting* and merge the loop bodies to simplify the resulting code.

3.4.2 Multi-Reductions

Another problem faced by the Fortran D compiler was handling reductions on replicated variables. A multidimensional reduction performs a reduction on multiple dimensions of an array. Finding the maximum value in a 3D array would be a 3D MAX reduction over an n^3 data set. We examine a special case of multidimensional reduction that we call a *multi-reduction*, where the program performs multiple reductions simultaneously. For instance, finding the maximum value of each column in a 3D array would be a 2D MAX multi-reduction composed of n^2 1D MAX reductions. Unlike normal multidimensional reductions, multi-

```
{* Original Fortran D Program *}
SUBROUTINE DZ3D6P
 REAL uud(n,n,n),uu(n,n,n)
 DECOMPOSITION dd(n,n,n)
 ALIGN uud, uu with dd
 DISTRIBUTE dd(:,:,BLOCK)
 do j = 1,n
  do i = 1,n
S1  uud(i,j,1) =
      F(uu(i,j,3),uu(i,j,n-1))
S2  uud(i,j,2) =
      F(uu(i,j,4),uu(i,j,n))
S3  uud(i,j,n-1) =
      F(uu(i,j,1),uu(i,j,n-3))
S4  uud(i,j,n) =
      F(uu(i,j,2),uu(i,j,n-2))
   enddo
  enddo
  do k = 3,n-2
   do j = 1,n
    do i = 1,n
S5   uud(i,j,k) =
       F(uu(i,j,k+2),uu(i,j,k-2))
    enddo
   enddo
  enddo
 end
```

```
{* Compiler Output for 4 Processors *}
SUBROUTINE DZ3D6P
REAL uud(n,n,n),uu(n,n,-1:(n/4)+2)
 n$ = n/4
 if (my$p .EQ. 0)
C1  send uu(1:n,1:n,1:2) to P3
 if (my$p .EQ. 3)
C2  send uu(1:n,1:n,n$-1:n$) to P0
 if (my$p .LT. 3)
C3  send uu(1:n,1:n,n$-1:n$) to my$p+1
 if (my$p .GT. 0)
C4  send uu(1:n,1:n,1:2) to my$p-1
 if (my$p .EQ. 0) then
  recv uu(1:n,1:n,n$+1:n$+2) from P3
  do j = 1,n
   do i = 1,n
S1   uud(i,j,1) = F(...)
S2   uud(i,j,2) = F(...)
   enddo
  enddo
 endif
 if (my$p .EQ. 3) then
  recv uu(1:n,1:n,-1:0) from P1
  do j = 1,n
   do i = 1,n
S3   uud(i,j,n$-1) = F(...)
S4   uud(i,j,n$) = F(...)
   enddo
  enddo
 endif
 if (my$p .GT. 0)
  recv uu(1:n,1:n,n$+1:n$+2) from my$p+1
 if (my$p .LT. 3)
  recv uu(1:n,1:n,-1:0) from my$p-1
 do k = lb$,ub$
  do j = 1,n
   do i = 1,n
S5   uud(i,j,k) = F(...)
   enddo
  enddo
 enddo
end
```

Figure 4 ERLEBACHER: Computation Phase in Z Dimension

reductions are directional in that they only transfer data across certain dimensions. This property allows the compiler to determine when communication is necessary. It also allows the problem to be partitioned in other dimensions so that no global reductions are required at the end.

The Fortran D compiler handles multi-reductions as follows. If the direction of the multi-reduction crosses a partitioned array dimension, then compilation proceeds as normal. The compiler produces code so that each processor computes part of every reduction in the multi-reduction, then inserts a global collective communication routine to accumulate the results. ERLEBACHER performs 2D SUM multi-reductions along each dimension of a 3D array for each of its three computation wavefronts. Consider statement S_1 in Figure 5, which performs a SUM multi-reduction in the Z dimension. Because this dimension is distributed, the compiler partitions the computation based on f, the distributed *rhs*, and inserts a call to *global-sum* to accumulate the results.

If the multi-reduction does not cross any distributed dimensions, no information is transferred across distributed dimensions. A processor can thus evaluate some of the reductions comprising the multi-reduction using local data. This case occurs in the solution step in the X and Y dimensions in ERLEBACHER. Simple loop bounds reduction is sufficient to partition the reduction; no communication is needed. If all results are

```
{* Original Fortran D Program *}
SUBROUTINE TRIDVPK
 REAL a(n),b(n),c(n),d(n),e(n)
 REAL tot(n,n),f(n,n,n)
 DISTRIBUTE f(:,:,BLOCK)
 {* perform forward substitution *}
 ...
 {* perform backward substitution *}
 do k = 1,n
  do j = 1,n
   do i = 1,n
S1   tot(i,j) =
       tot(i,j)+d(k)*f(i,j,k)
   enddo
  enddo
 enddo
 do j = 1,n
  do i = 1,n
S2  f(i,j,n) =
      (f(i,j,n)-tot(i,j))*b(n)
  enddo
 enddo
 do j = 1,n
  do i = 1,n
S3  f(i,j,n-1) =
      f(i,j,n-1)-e(n-1)*f(i,j,n)
  enddo
 enddo
 do k = n-2,1,-1
  do j = 1,n
   do i = 1,n
S4   f(i,j,k) = f(i,j,k)-c(k)*
       f(i,j,k+1)-e(k)*f(i,j,n)
   enddo
  enddo
 enddo
end
```

```
{* Compiler Output for 4 Processors *}
SUBROUTINE TRIDVPK
 REAL a(n),b(n),c(n),d(n),e(n)
 REAL tot(n,n),f(n,n,0:(n/4)+1),r$buf1(n)
 {* perform forward substitution *}
 ...
 {* perform backward substitution *}
 n$ = n/4
 off$0 = my$p * n$
 do k = 1,n$
  k$ = k + off$0
  do j = 1,n
   do i = 1,n
    tot(i,j) = tot(i,j)+d(k$)*f(i,j,k)
   enddo
  enddo
 enddo
 global-sum tot(1:n,j:n)
 if (my$p .EQ. 3) then
  do j = 1,n
   do i = 1,n
    f(i,j,n$-1) = (f(i,j,n$)-tot(i,j))*b(n)
   enddo
  enddo
  do j = 1,n
   do i = 1,n
    f(i,j,n$-1) = f(i,j,n$-1)-e(n-1)*f(i,j,n$)
   enddo
  enddo
  buffer f(1:128, 1:128, n$) into rbuf$1(n*n)
  broadcast rbuf$1(1:n*n)
 else
  recv rbuf$1(1:n*n)
 endif
 do j = 1,n
  do i$ = 1,n,8
   i$up = i$+7
   if (my$p .LT. 3)
    recv f(i$:i$up, j, n$+1) from my$p+1
   do i = i$,i$+8
    do k = ub$,1,-1
     k$ = k + off$0
     f(i,j,k) = f(i,j,k)-c(k$)*f(i,j,k+1)
                - e(k$)*r$buf1(j*n+i-n)
    enddo
   enddo
   if (my$p .GT. 0)
    send f(i$:i$up, j, 1) to my$p-1
  enddo
 enddo
end
```

Figure 5 ERLEBACHER: Solution Phase in Z Dimension

needed, a global *concatenation* routine can be called to collect the results from each processor.

3.4.3 Array Kills

For instance, a multi-reduction is performed in the Y dimension solution step of ERLEBACHER, shown in Figure 6. Because the Y dimension of f is local, relaxing the owner computes rule allows each processor to compute its reductions locally. Unfortunately the multi-reduction is being computed for *tot*, a replicated array. The compiler thus inserts a global concatenation routine to collect values of *tot* from other processors. This concatenation is the only communication inserted in the computation sweeps in the X and Y dimensions, and turns out to be unnecessary. Array kill analysis would show that the values of *tot* only reach uses in the next loop nest S_2, where it is used only on iterations executed locally. Values for *tot* not computed locally

```
     {* Original Fortran D Program *}
     SUBROUTINE TRIDVPJ
      REAL a(n),b(n),c(n),d(n),e(n)
      REAL tot(n,n),f(n,n,n)
      DISTRIBUTE f(:,:,BLOCK)
      do k = 1,n
       do j = 1,n
        do i = 1,n
S1       tot(i,k) =
            tot(i,k) + d(j)*f(i,j,k)
        enddo
       enddo
      enddo
      do k = 1,n
       do i = 1,n
S2      f(i,n,k) =
            (f(i,n,k) - tot(i,k))*b(n)
       enddo
      enddo
     end
```

```
{* Compiler Output for 4 Processors *}
SUBROUTINE TRIDVPK
 REAL a(n),b(n),c(n),d(n),e(n)
 REAL tot(n,n),f(n,n,0:(n/4)+1)
 n$ = n/4
 off$0 = my$p * n$
 do k = 1,n$
  k$ = k + off$0
  do j = 1,n
   do i = 1,n
    tot(i,k$) = tot(i,k$) + d(j)*f(i,j,k)
   enddo
  enddo
 enddo
 global-concat tot(1:n,j:n)
 do k = 1,n$
  k$ = k + off$0
  do i = 1,n
   f(i,n,k) = (f(i,n,k) - tot(i,k$))*b(n)
  enddo
 enddo
end
```

Figure 6 ERLEBACHER: Solution Phase in Y Dimension

are not ever used. This information can be employed to eliminate the unnecessary global concatenation. Array kill analysis has not yet been implemented in the prototype compiler.

3.4.4 Exploiting Pipeline Parallelism

Finally, because the computational wavefront traverses across processors in the Z dimension, the Fortran D compiler must efficiently exploit pipeline parallelism [17]. In Figure 5, the compiler detects that the k loop enclosing statement S_4 is a cross-processor loop because it carries a true dependence whose endpoints are on different processors. To exploit pipeline parallelism, the compiler interchanges k innermost, then strip-mines the enclosing i loop to reduce the communication overhead. Note that the nonlocal reference to $f(i,j,n)$ has also been converted to a vectorized broadcast. The compiler replaced the reference with r$buf1$(j*n+i-n)$ to properly access data in the buffer array.

4 Empirical Evaluation of the Fortran D Compiler

To evaluate the status of the current Fortran D compiler prototype, the output of the Fortran D compiler is compared with hand-optimized programs on the Intel iPSC/860 and the output of the CM Fortran compiler on the TMC CM-5. Our goal is to validate our compilation approach and identify directions for future research. In many cases, problems sizes were too large to be executed sequentially on one processor. In these cases sequential execution times are estimates, computed by projecting execution times for smaller computations to the larger problem sizes. Empirical results are presented in both tabular and graphical form.

4.1 Comparison with Hand-Optimized Kernels

We begin by comparing the output of the Fortran D compiler against hand-optimized stencil kernels on the Intel iPSC/860 hypercube. Our iPSC timings were obtained on the 32 node Intel iPSC/860 at Rice University. It has 8 Meg of memory per node and is running under Release 3.3.1 of the Intel software. Each program was compiled under -O4 using Release 3.0 of *if77*, the iPSC/860 compiler. Timings were made using *dclock*(), a microsecond timer.

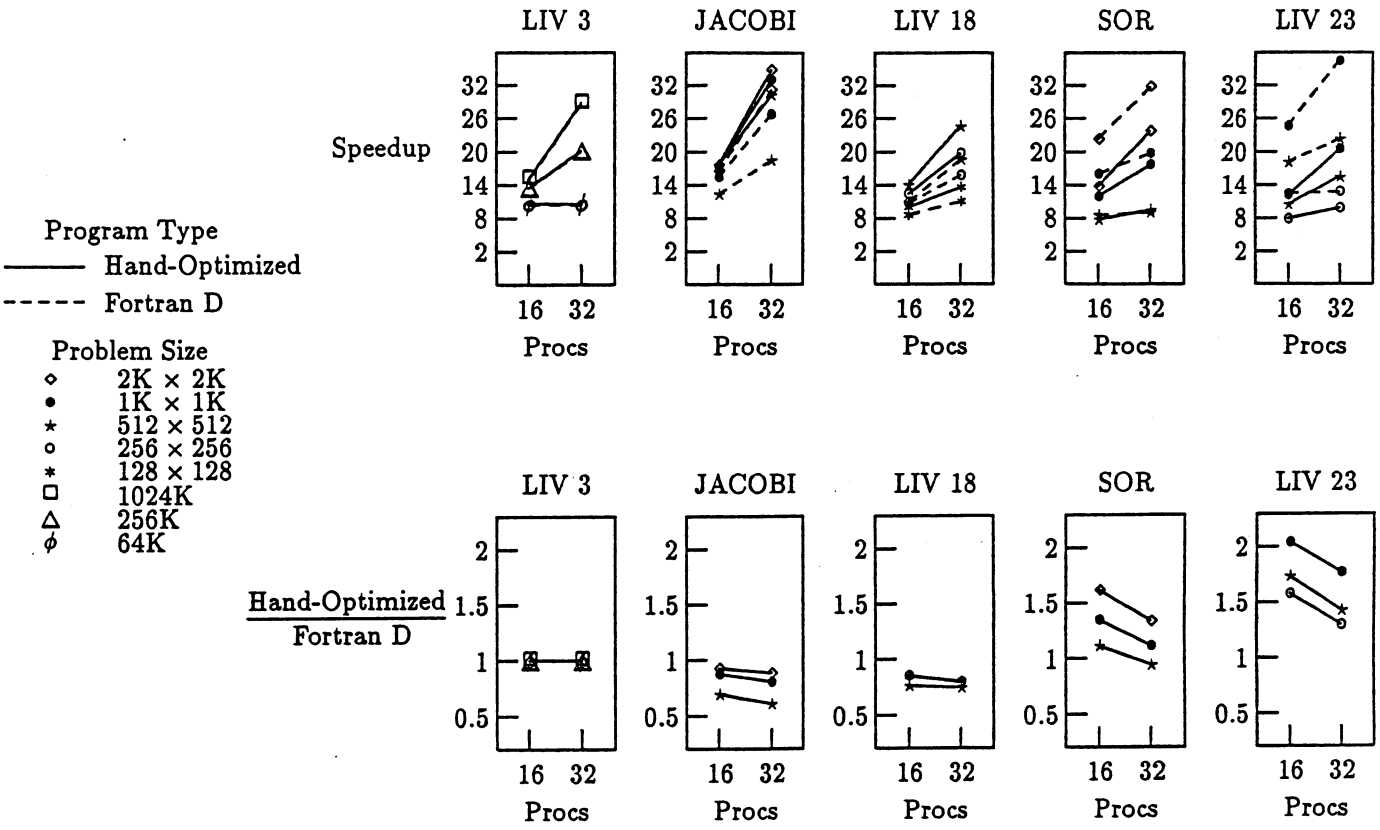

Figure 7 Speedups & Comparisons for Stencil Kernels (Intel iPSC/860)

The hand-optimized stencil kernels are taken from a previous study evaluating the effect of different communication & parallelism optimizations on overall performance [18]. We selected a sum reduction (Livermore 3), two parallel kernels (Livermore 18, Jacobi), and two pipelined kernels (Livermore 23, SOR). As before, all arrays are double precision and distributed block-wise in one dimension. Speedups for different problem and machine sizes are graphically displayed at the tope of Figure 7, with speedups plotted along the Y-axis and number of processors along the X-axis. Solid and dashed lines correspond to speedups for hand-optimized and Fortran D programs, respectively. Each line represents the speedup for a given problem size. The bottom of the figure compares the ratio of execution times between the hand-optimized and Fortran D versions of each kernel. Each line represents the ratio for a given problem size.

We found that the code generated for the inner product in Livermore 3 were identical to the hand-optimized versions, since the compiler recognized the sum reduction and used the appropriate collective communication routine. For parallel kernels, the output of the Fortran D compiler was within 50% of the best hand-optimized codes. The deficit was mainly caused by the Fortran D compiler not exploiting unbuffered messages in order to eliminate buffering and overlap communication overhead with local computation. The compiler-generated code actually outperformed the hand-optimized pipelined codes, even though the two message-passing Fortran 77 versions of the program were nearly identical. We thus assume the performance differences to be probably due to complications with the scalar i860 node compiler in the parameterized hand-optimized version.

4.2 Comparison with Hand-Optimized Subroutines and Programs

We now turn our attention to evaluating the performance of the Fortran D compiler for large subroutines and application codes. In the following sections, we display speedups and comparisons for SHALLOW and the three other codes studied, both in tables and graphically.

The top of Figure 8 displays speedups for each program graphically, with speedups plotted along the Y-axis and number of processors along the X-axis. Solid and dashed lines correspond to speedups for hand-optimized and Fortran D programs, respectively. Each line represents the speedup for a given problem size. The bottom of the figure compares the ratio of execution times between the hand-optimized and Fortran D versions of each program. Each line represents the ratio for a given problem size.

4.2.1 Results for SHALLOW

Table 1 contains timings for performing one time step of SHALLOW. It presents speedups as well as the ratio of execution times between hand-optimized and Fortran D versions of the program. We found the program to be ideal for distribute-memory machines. Computation is entirely data-parallel, with nearest-neighbor communication taking place between phases of each time step. The compiler output achieved excellent speedups (21–29), even for smaller problems. To evaluate potential improvements, we performed aggressive inter-loop message coalescing and aggregation by hand, halving the total number of messages. The hand-optimized versions of SHALLOW exhibited only slight improvements (1–10%) over the compiler-generated code, except when small problems were parallelized on many processors (12–26%). Communication costs apparently only contributed to a small percentage of total execution time, reducing the impact and profitability of advanced communication optimizations.

4.2.2 Results for DISPER

Like SHALLOW, DISPER is a completely data-parallel computation that requires only nearest-neighbor communications. Timings for DISPER in Table 2 show near-linear speedups for the output of the Fortran D compiler, once errors introduced by execution conditions were corrected. We also created a hand-optimized version of DISPER by applying aggressive inter-loop message message aggregation. The resulting message was large enough that it became profitable to also employ unbuffered *isend* and *irecv* messages. However, since communication overhead is small, the hand-optimized version only yielded minor improvements (1-3%) for the single problem size tested.

4.2.3 Results for DGEFA

Table 3 presents execution times and speedups for DGEFA, Gaussian elimination with partial pivoting. Results indicate that the Fortran D compiler output, shown in Figure 1, provided limited speedups (3–6) on small problems. For larger problems moderate speedups (11–16) were achieved. Due to the large number of global broadcasts required to communicate pivot values and multipliers, performance of DGEFA actually degrades when solving small problems on many processors.

To determine whether improved performance is attainable, we created a hand-optimized version of DGEFA based on optimizations described in the literature [11, 23]. First, we combined the two messages broadcast on each iteration of the outermost k loop. Instead of broadcasting the pivot value immediately, we wait until multipliers are also computed. The values can then be combined in one broadcast. Overcommunication may result when a zero pivot is found, since messages now include multipliers even if they are not used. However, combining broadcasts is still profitable as zero pivots occur rarely.

Second, we restructured the computation so that upon receiving the pivot for the current iteration, the processor P_{k+1} responsible for finding the pivot for the next iteration does so immediately. P_{k+1} performs row elimination on just the first column of the remaining subarray, scans that column to find a pivot and calculates multipliers. P_{k+1} then broadcasts the pivot and multipliers to the other processors before performing row elimination on the remaining subarray. Since row eliminations make up most of the computation in Gaussian elimination, each broadcast in effect takes place one iteration ahead of the matching receive, hiding communication costs by overlapping message latency with local computation.

Results for the hand-optimized version of DGEFA are presented in Table 3. The new algorithm showed little or no improvement for small problems or when few processors were employed. However, it increased performance by over 30% for large problems on many processors, yielding decent speedups (14–25). The Fortran D compiler can thus benefit from more aggressive optimization of linear algebra routines. Experience also indicates that programmers can achieve higher performance for linear algebra codes with block versions of these algorithms. The Fortran D compiler will need to provide BLOCK_CYCLIC data distributions to support these block algorithms.

4.2.4 Results for ERLEBACHER

Unlike SHALLOW and DISPER, ERLEBACHER is not fully data-parallel. It is a more complex program that requires global communication, and also contains computation wavefronts that sequentialize parts of the computation. For ERLEBACHER, the Fortran D compiler first performs interprocedural reaching decomposition and overlap analysis, then invokes local code generation for each procedure. The compiler inserts global communication for array SUM reductions, and also applies coarse-grain pipelining. Timings for ERLEBACHER in Table 4 show that the compiler-generated code is rather inefficient, with speedup peaking at 3–5 even for large programs.

To determine how much improvement is attainable, we applied additional optimizations to create three hand-optimized versions. Optimizations are cumulative from left to right, so each hand-optimized program contains optimizations applied in the previous version. In the "Array Kill" version we used interprocedural array kill analysis to eliminate global concatenation for local multi-reductions on replicated arrays in the X and Y sweeps. In the "Pipelining" version we also experimented with the granularity of coarse-grain pipelining performed during forward and backward substitution in the Z sweep. We found that a strip size around 16 yielded significantly better performance than the default strip size of 8 selected by the Fortran D compiler.

Finally, in the "Memory" version we also performed loop interchange to improve the data locality of each node program during forward and backward substitution in the Z sweep. The current algorithm for pipelining in the Fortran D compiler simply interchanges the cross-processor loop innermost, without taking data locality into account. It thus placed the k loop innermost in TRIDVPK. We interchanged the strip-mined i loop innermost by hand, improving data locality by restoring unit-stride memory accesses.

Timings show that all three optimizations contribute to improved performance. Using array kill information and adjusting pipelining granularity reduced communication costs, especially when many processors were used. Improving data locality of the node program helped most when few processors were used and large data sizes caused many cache misses. Together these optimizations yielded speedups of 5–9, improving performance by up to 50% over the Fortran D compiler-generated code.

Problem Size	Proc	Fortran D time	Fortran D speedup	Hand-Optimized time	Hand-Optimized speedup	Hand-Optimized / Fortran D
256 × 256	1	*sequential time = 0.728*				
	2	0.354	2.06	0.348	2.09	0.98
	4	0.195	3.73	0.188	3.87	0.96
	8	0.097	7.50	0.091	8.00	0.94
	16	0.056	13.0	0.049	14.86	0.88
	32	0.035	20.8	0.026	28.00	0.74
512 × 512	1	*estimated sequential time = 2.9*				
	2	1.529	1.90	1.521	1.91	0.99
	4	0.707	4.10	0.698	4.15	0.99
	8	0.377	7.69	0.368	7.88	0.98
	16	0.201	14.43	0.191	15.18	0.95
	32	0.107	27.10	0.095	30.53	0.89
$1K \times 1K$	1	*estimated sequential time = 11.6*				
	8	1.620	7.16	1.610	7.20	0.99
	16	0.755	15.36	0.739	15.70	0.98
	32	0.397	29.22	0.380	30.53	0.95

Table 1 Intel iPSC/860 Execution Times for SHALLOW (in seconds)

Problem Size	Proc	Fortran D time	Fortran D speedup	Hand-Optimized time	Hand-Optimized speedup	Hand-Optimized / Fortran D
256 × 8 × 8 × 4	1	*estimated sequential time = 39.0*				
	4	9.971	3.91	10.222	3.81	1.03
	8	5.040	7.74	4.979	7.83	0.99
	16	2.440	15.98	2.414	16.16	0.99
	32	1.284	30.37	1.240	31.45	0.97

Table 2 Intel iPSC/860 Execution Times for DISPER (in seconds)

Problem Size	Proc	Fortran D time	Fortran D speedup	Hand-Optimized time	Hand-Optimized speedup	Hand-Optimized / Fortran D
256 x 256	1	*sequential time = 2.151*				
	2	1.051	2.05	1.108	1.94	1.05
	4	0.744	2.89	0.683	3.15	0.92
	8	0.670	3.21	0.551	3.90	0.82
	16	0.695	3.09	0.644	3.34	0.93
	32	0.782	2.75	0.758	2.84	0.97
512 x 512	1	*sequential time = 17.53*				
	2	7.988	2.19	7.879	2.22	0.99
	4	4.786	3.66	4.322	4.06	0.90
	8	3.373	5.20	2.601	6.74	0.77
	16	2.908	6.03	2.259	7.76	0.78
	32	2.916	6.01	2.619	6.69	0.90
$1K \times 1K$	1	*estimated sequential time = 140*				
	2	66.74	2.10	68.91	2.03	1.03
	4	36.29	3.86	35.61	3.93	0.98
	8	21.83	6.41	18.93	7.40	0.87
	16	15.32	9.14	10.97	12.76	0.72
	32	12.96	10.80	9.654	14.50	0.74
$2K \times 2K$	1	*estimated sequential time = 1120*				
	8	160.45	6.98	145.83	7.68	0.91
	16	97.22	11.52	76.28	14.68	0.78
	32	68.86	16.26	44.62	25.10	0.65

Table 3 Intel iPSC/860 Execution Times for DGEFA (in seconds)

Problem Size	Proc	Fortran D time	Fortran D speedup	Hand-Optimized: Array Kill time	Array Kill speedup	Pipelining time	Pipelining speedup	Memory time	Memory speedup	Hand-Optimized / Fortran D
64 x 64 x 64	1	*sequential time = 1.577*								
	2	1.104	1.43	1.071	1.47	1.051	1.50	0.805	1.96	0.73
	4	0.765	2.06	0.726	2.17	0.630	2.50	0.586	2.69	0.77
	8	0.657	2.40	0.599	2.63	0.452	3.49	0.448	3.52	0.68
	16	0.539	2.93	0.427	3.69	0.312	5.05	0.311	5.07	0.53
	32	0.613	2.57	0.461	3.43	0.314	5.02	0.315	5.00	0.51
96 x 96 x 96	1	*estimated sequential time = 5.3*								
	4	1.677	3.17	1.590	3.33	1.311	4.06	1.151	4.60	0.70
	8	1.475	3.61	1.312	4.04	0.961	5.54	0.917	5.78	0.62
	16	1.492	3.57	1.189	4.46	0.824	6.46	0.813	6.52	0.54
	32	1.355	3.93	1.059	5.00	0.741	7.15	0.720	7.36	0.54
128 x 128 x 128	1	*estimated sequential time = 12.6*								
	8	3.341	3.77	3.101	4.06	2.508	5.03	1.905	6.61	0.59
	16	2.997	4.21	2.528	4.98	1.876	6.72	1.584	7.95	0.56
	32	2.683	4.70	2.146	5.87	1.497	8.42	1.347	9.35	0.50

Table 4 Intel iPSC/860 Execution Times for ERLEBACHER (in seconds)

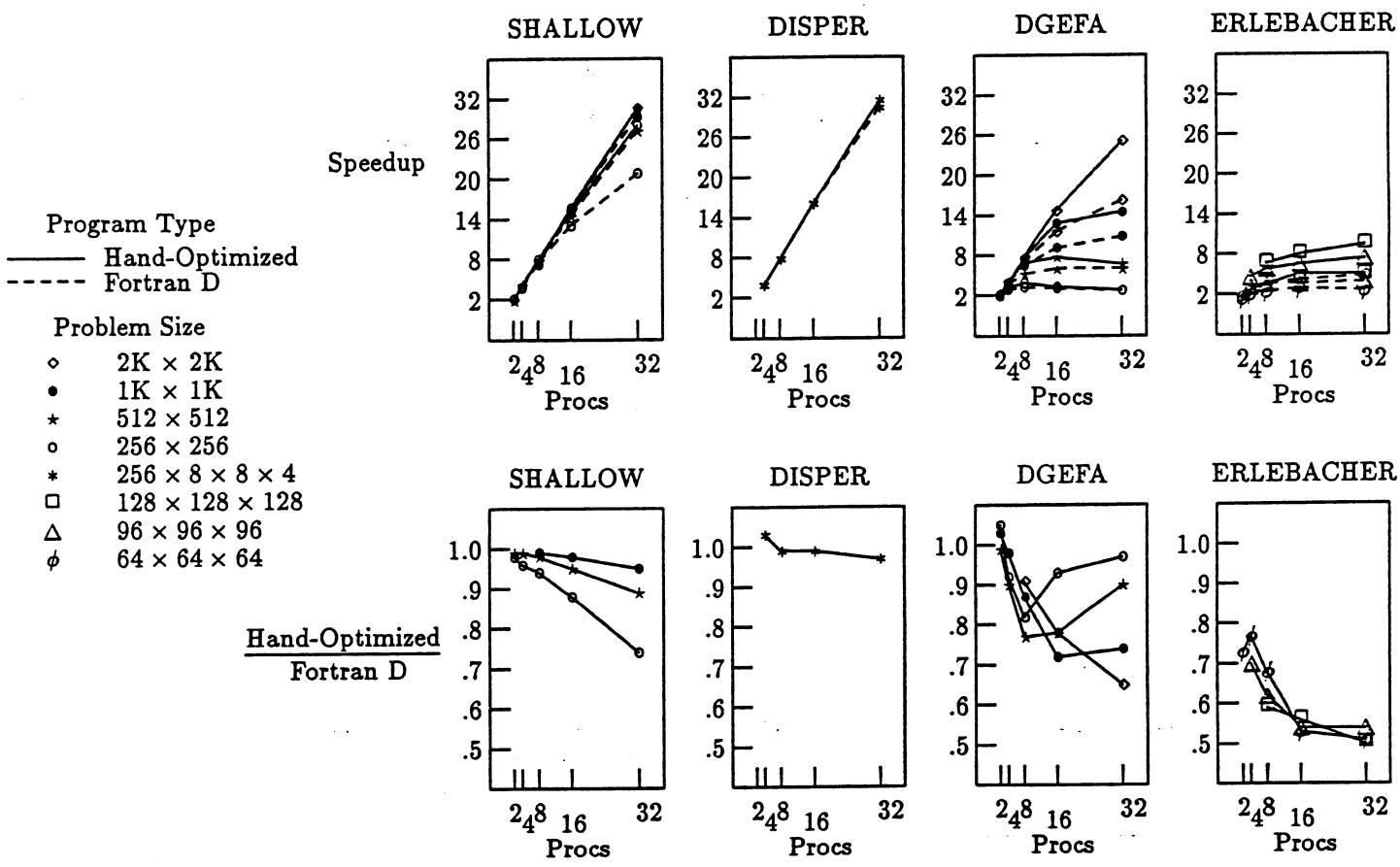

Figure 8 Speedups & Comparisons for Programs (Intel iPSC/860)

4.3 Comparison with CM Fortran Compiler

We also evaluated the performance of the Fortran D compiler against a commercial compiler. We selected the CM Fortran compiler, the most mature and widely used compiler for MIMD distributed-memory machines, and compared it against the Fortran D compiler on the Thinking Machines CM-5.

Our CM-5 timings were obtained on the 32 node CM-5 at Syracuse University. It has Sun Sparc processors running SunOS 4.1.2 and vector units running CMOST 7.2 S2. CM Fortran programs were compiled using *cmf* version 2.1 *beta*, with the -O and -vu flags. They were timed using *CM_timer_read_elapsed*(). CM Fortran programs were compared against message-passing Fortran 77 programs using CMMD version 3.0 *beta*, the CM message-passing library. Fortran 77 node programs were compiled using the Sun Fortran compiler *f77*, version 1.4, with the -O flag. They were linked with *cmmd* version 3.0 *beta*. Fortran 77 node programs were timed using *CMMD_node_timer_elapsed*().

4.3.1 Results for Kernels and Programs

The output of the Fortran D compiler was easily ported to the CM-5 by replacing calls to Intel NX/2 message-passing routines with equivalent calls to TMC CMMD message-passing routines. We converted program kernels into CM Fortran by hand for the CM Fortran compiler, inserting the appropriate LAYOUT directives to achieve the same data decomposition. The inner product in Livermore 3 was replaced by DOTPRODUCT, a CM Fortran intrinsic. Jacobi, Livermore 18, and SHALLOW can be transformed directly into CM Fortran. Loop skew and interchange must be applied to SOR and Livermore 23 to expose parallelism in the form of FORALL loops. A mask array *indx* is used to implement Gaussian elimination. The resulting CM Fortran code, except for SHALLOW, are shown in Figure 9.

The CM Fortran compiler can generate two versions of output. The first uses CM-5 vector units, the second only uses the Sparc node processor. Unfortunately, the current TMC Fortran 77 compiler does not generate code to utilize CM-5 vector units. Fortran D message-passing programs are thus forced to rely on the Sparc processor.[1] To permit a balanced comparison, we provide timings for CM Fortran programs using either Sparc or vector units. Table 5 shows the elapsed times we measured on the CM-5 for CM Fortran and Fortran D programs, as well as the ratio of execution times between CM Fortran and Fortran D code. Sequential execution times on a single Sparc 2 workstation are provided for comparison.

We also graphically present the execution times measured on the CM-5. Figure 10 displays measured execution speed. Execution times in seconds are plotted logarithmically along the Y-axis. The problem size is plotted logarithmically along the X-axis. Solid, dotted, and dashed lines represent the CM Fortran using Sparc, CM Fortran using vector units, and Fortran D using Sparc, respectively. All parallel execution times are for 32 processors. Figure 11 displays the ratio of execution times of both versions of CM Fortran code (sparc/vector) to Fortran D (sparc), plotting ratios along the Y-axis.

[1] For the final version of the paper we expect to be able to provide results for Fortran D programs using CM-5 vector units.

```
{* Livermore 3 (Inner Product) *}
  s = DOTPRODUCT(a,b)

{* Jacobi *}
  forall (j=2:N-1,i=2,N-1)
    a(i,j) = 0.25*(b(i-1,j)+b(i+1,j)+b(i,j-1)+b(i,j+1))
  b = a

{* Livermore 18 (Explicit Hydrodynamics) *}
  forall (k=2:N-1, j=2:N-1)
    za(j, k) = (zp(j-1, k+1) + zq(j-1, k+1) - zp(j-1, k) - zq(j-1, k))
        * (zr(j, k) + zr(j-1, k)) / (zm(j-1, k) + zm(j-1, k+1))
  forall (k=2:N-1, j=2:N-1)
    zb(j, k) = (zp(j-1, k) + zq(j-1, k) - zp(j, k) - zq(j, k)
        * (zr(j, k) + zr(j, k-1)) / zm(j, k) + zm(j-1, k))
  forall (k=2:N-1, j=2:N-1)
    zu(j, k) = zu(j, k) + s * (za(j, k) * (zz(j, k) - zz(j+1, k))
        - za(j-1, k)* (zz(j, k) - zz(j-1, k)) - zb(j, k)
        * (zz(j, k) - zz(j, k-1)) + zb(j, k+1) * (zz(j, k) - zz(j, k+1)))
  forall (k=2:N-1, j=2:N-1)
    zv(j, k) = zv(j, k) + s * (za(j, k) * (zr(j, k) - zr(j+1, k))
        - za(j-1, k) * (zr(j, k) - zr(j-1, k)) - zb(j, k+1) * (zr(j, k)
        - zr(j, k-1)) + zb(j, k+1) * (zr(j, k) - zr(j, k+1)))
  forall (k=2:N-1, j=2:N-1)
    zr(j, k) = zr(j, k) + t * zu(j, k)
  forall (k=2:N-1, j=2:N-1)
    zz(j, k) = zz(j, k) + t * zv(j, k)

{* SOR *}
  do j = 4,2*(N-1)
    forall (i=max(2,j-N+1):min(N-1,j-2))
      a(i,j-i) = 0.175*(a(i-1,j-i)+a(i+1,j-i)+a(i,j-i-1)
        + a(i, j-i+1)) + 0.3 * a(i, j-i)
  enddo

{* Livermore 23 (Implicit Hydrodynamics) *}
  do j = 4,2*(N-1)
    forall (k=max(2,j-N+1):min(N-1,j-2))
      za(k,j-k) = za(k,j-k)+.175*((za(k,j-k+1)*zr(k,j-k)
          * zb(k, j-k) + za(k+1,j-k)*zu(k,j-k)
          + za(k-1, j-k) * zv(k, j-k)) - za(k,j-k))
  enddo

{* Gassian Elimination *}
  indx = 0
  do k = 1,N
    iTmp = maxloc(abs(a(:,k)), MASK = indx .EQ. 0)
    indxRow = iTmp(1)
    maxNum = a(indxRow,k)
    indx(indxRow) = k
    fac = a(:,k) / maxNum
    row = a(indxRow,:)
    forall (i = 1:N, j = 1:N+1, (indx(i).EQ.0) .AND. (j.GE.k))
      a(i,j) = a(i,j) - fac(i) * row(j)
  enddo
```

Figure 9 CM Fortran Versions of Kernels

Program	Problem Size	Sequential Execution Sparc	Fortran D + CMMD Sparc	CM Fortran Sparc	CM Fortran Vector	CM Fortran / Fortran D Sparc	CM Fortran / Fortran D Vector
Livermore 3	$64K$	0.005	0.002	0.018	0.005	9.92	3.19
Inner	$256K$	0.020	0.007	0.032	0.006	4.67	0.88
Product	$1024K$	0.079	0.027	0.098	0.007	3.63	0.27
Jacobi	512×512	0.877	0.027	0.236	0.045	8.74	1.67
Iteration	$1K \times 1K$	3.525	0.103	0.766	0.079	7.44	0.77
	$2K \times 2K$	14.14	0.362	2.834	0.159	7.83	0.44
Livermore 18	128×128	0.457	0.022	0.165	0.100	7.50	4.55
Explicit	256×256	1.861	0.062	0.332	0.132	5.35	2.13
Hydrodynamics	512×512	7.554	0.223	0.994	0.163	4.46	0.73
	256×256	1.297	0.031	0.409	0.185	13.2	5.97
SHALLOW	512×512	5.210	0.141	1.363	0.256	9.67	1.82
	$1K \times 1K$	20.88	0.520	6.159	0.408	11.8	0.78
Successive	512×512	0.376	0.060	17.04	7.559	284	126
Over	$1K \times 1K$	1.519	0.130	116.1	27.39	893	211
Relaxation	$2K \times 2K$	6.134	0.364	209.9	128.6	577	353
Livermore 23	256×256	0.389	0.035	2.897	2.516	82.7	71.9
Implicit	512×512	1.562	0.118	18.19	8.686	154	73.6
Hydrodynamics	$1K \times 1K$	6.252	0.320	122.7	31.59	383	98.7
	256×256	4.791	0.561	10.65	3.604	19.0	6.42
DGEFA	512×512	40.61	2.779	104.8	56.50	37.7	20.3
	$1K \times 1K$	337.1	16.82	856.9	162.1	50.9	9.64
	$2K \times 2K$	6809	109.8	8449	1365	76.8	12.4

Table 5 TMC CM-5 Execution Times (for 32 processors, in seconds)

When comparing Sparc versions of each program, our measurements indicate the current CM Fortran compiler produces code that is significantly slower than the corresponding message-passing programs generated by the Fortran D compiler. The difference is especially pronounced for small data sizes, and even intrinsic functions such as DOTPRODUCT yield very poor performance. The CM Fortran compiler fared best on data-parallel computations such as Jacobi, Livermore 18, and SHALLOW (4–13 times slower). It appears to handle pipelined computations and Gaussian elimination poorly (20+ times slower), even when expressed in a form that contains vector parallelism. Only when the CM Fortran compiler is able to exploit CM-5 vector units does it match the performance of the Fortran D compiler, exceeding it on larger problem sizes.

Based on examining the assembly code output of the CM Fortran compiler, we believe its poor performance is due to the fact that the current CM Fortran compiler generates code for executing virtual processes on each node. This mode of execution requires extensive run-time calculation of addresses and results in much unnecessary data movement. In addition, the CM Fortran compiler generates code that can be executed on any number of processors, whereas the prototype Fortran D compiler targets a specific number of processors at compile-time.

Because the CM Fortran compiler for the CM-5 is relatively new (though it is the most mature commercial compiler available), we hesitant to draw too many conclusions. However, it is clear from our results that severe performance penalties may result if important compile-time decisions are postponed until run-time. Improvements in an upcoming release of the CM Fortran compiler will allow more meaningful comparisons with the Fortran D compiler in the future.

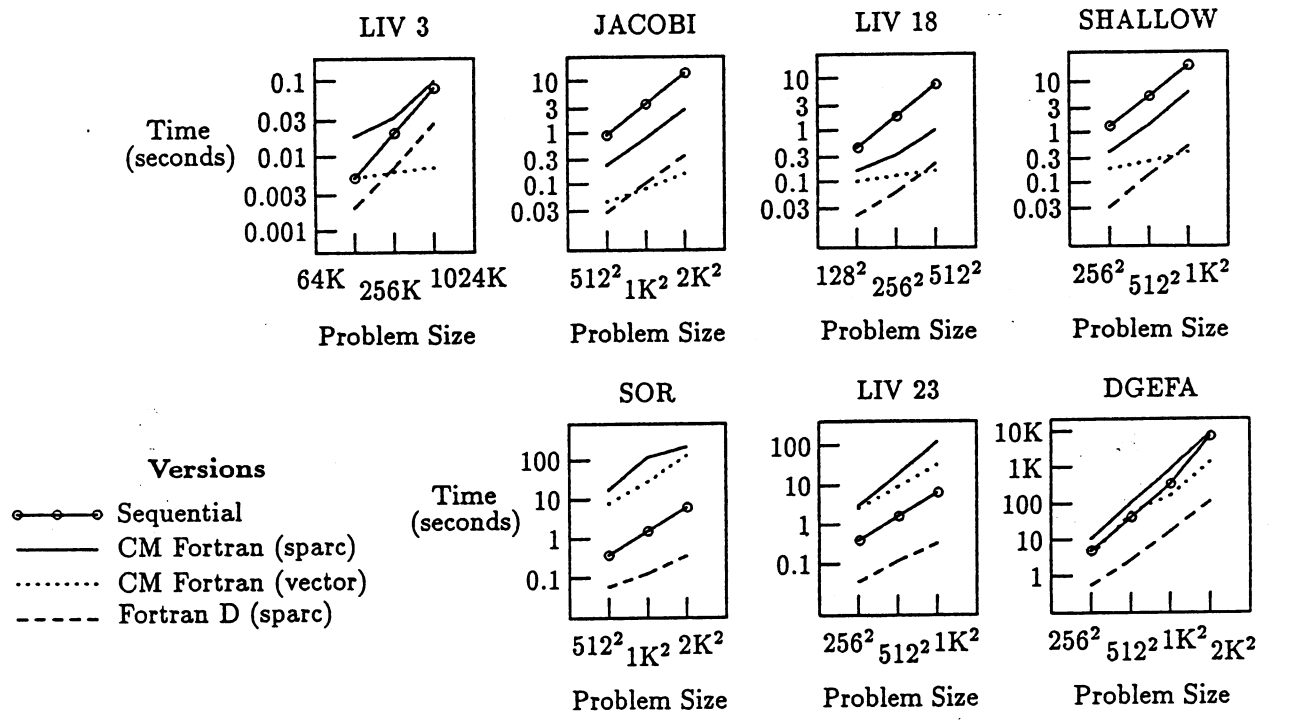

Figure 10 Execution Times (32 Processor Thinking Machines CM-5)

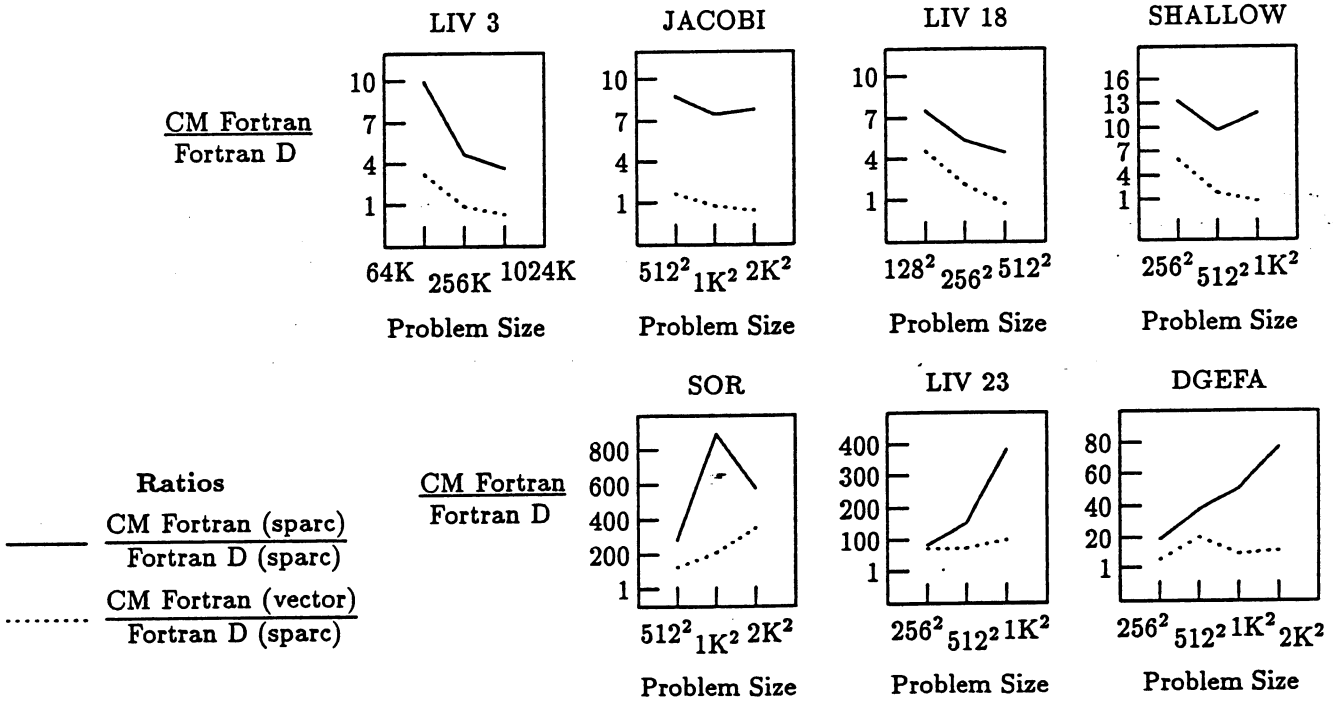

Figure 11 CM Fortran/Fortran D Comparisons (32 Processor Thinking Machines CM-5)

5 Lessons and Implications

Our preliminary experiences have helped us evaluate the current Fortran D compiler prototype. The initial results, though encouraging, point out a number of areas that require additional work. We summarize our appraisal of the Fortran D compiler with these observations:

- It has achieved considerable success in generating efficient code for stencil computations.
- It needs to improve its optimization of pipelined and linear algebra codes.
- It must become much more flexible before it can become a successful machine-independent programming model. Symbolic information and run-time support must be added.

In addition, our experiences confirm that the nature of the computation is the overriding factor in determining the success of the Fortran D compiler. We discuss each point in greater detail in the following sections.

5.1 Parallel Stencil Computations

By generating output for SHALLOW and DISPER that virtually matched optimized hand-optimized versions, the Fortran D compiler has demonstrated its success for parallel stencil computations. This is despite the fact that the compiler is not producing the most efficient communication, since it does not yet support unbuffered messages. The Fortran D compiler succeeds because it does a sufficiently good job that communication costs become a minor part of the overall execution time. In particular, scalability is excellent because performance improves as the problem size increases. Implementing additional optimizations is desirable for achieving good speedup for small programs or many processors, but is not crucial. Instead, the focus should be on improving the flexibility and robustness of the Fortran D compiler, as discussed in section 5.3.

5.2 Pipelined and Linear Algebra Computations

In comparison, there is considerable room for improvement when compiling communication-intensive codes such as pipelined and linear algebra computations. Results for DGEFA and ERLEBACHER show that the current Fortran D compiler prototype only attains limited speedups. It can achieve noticeable performance gains by applying advanced communication optimizations. These optimizations are important because communication is performed much more frequently than in parallel stencil computations. Their effect on overall execution time gain in importance as the problem size and number of processors increases. In particular, the Fortran D compiler will need to use information from training sets and static performance estimation to select an efficient granularity for coarse-grain pipelining.

5.3 Increase Flexibility

Finally, when evaluating its overall performance, we find that the most serious problem facing the prototype Fortran D compiler is its lack of flexibility. In the course of conducting our study, we were unable to apply the Fortran D compiler to a large number of standard benchmark programs, despite the fact they contained dense-matrix computations that should have been acceptable to the compiler. Even programs that were written in a "clean" data-parallel manner required fairly extensive rewriting to eliminate programming artifacts that the prototype proved unable to compile. The causes for this inflexibility can be categorized as follows:

5.3.1 Immature Symbolic and Interprocedural Analysis

The lack of symbolic analysis in the current Fortran D compiler proved to be a major stumbling block. Unlike parallelizing compilers for shared-memory machines, simply providing precise dependence information was insufficient for the Fortran D compiler. The compiler performs deep analysis that requires knowledge of all

subscript expressions and loop bounds in the program. For most programs, constant propagation, forward expression folding, and auxiliary induction variable substitution all need to be performed before the Fortran D compiler can proceed.

The current prototype is also inhibited by missing pieces in interprocedural analysis. It does not understand formal parameters that represent subarrays in the calling procedure or multiple entry points. Both symbolic and interprocedural analysis need to be completed and integrated with the Fortran D compiler before many existing programs can be considered.

5.3.2 Lack of Run-time Support

Another problem with the current compiler prototype is its reliance on compile-time analysis. The only run-time support it requires are routines for packing and unpacking non-contiguous array elements into contiguous message buffers. The compiler attempts to calculate at compile-time all information, including ownership, communication, and partitioning. While this approach is necessary for advanced optimizations and generating efficient code, it limits the Fortran D compiler to computations it can completely analyze.

It turns out real programs contain many components that cannot be easily analyzed at compile-time, such as indirect references, complex control flow, and scalar computations. These occur fairly frequently in initializations and boundary condition calculations. In many cases the Fortran D compiler was forced to abort, despite being able to compile the important kernel computations in the program.

What the Fortran D compiler must provide are methods of utilizing run-time support, trading performance for greater flexibility in non-critical regions of the program. The compiler can either apply run-time resolution or demand more support from the run-time library to calculate ownership, partitioning, and communication at run-time. Since in most cases the code affected is executed infrequently, the expense of run-time methods should not significantly impact overall execution time.

5.3.3 Immature Fortran D Compiler

A major part of the problem lies with the immaturity of the Fortran D compiler itself. There are a number of dense-matrix computations that it is not able to analyze and compile efficiently. For instance, the prototype compiler does not handle non-unit loop steps or subscript coefficients. It is thus unable to compile Red-Black SOR or multigrid computations, both of which possess constant step sizes greater than one. Computations such as Fast Fourier Transform (FFT), linear recurrences, finite-element, n-body problems, and banded tridiagonal solvers all possess regular but specialized data access patterns that the Fortran D compiler needs to recognize and efficiently support. In addition, run-time support for irregular and sparse computations must also be added. Only when these obstacles are overcome can the Fortran D compiler serve as a credible general-purpose programming model.

5.3.4 Dusty Decks

Finally, the Fortran D compiler cannot compile a number of "dusty deck" Fortran programs that were originally written for sequential or vector machines. These programs contain programming constructs that the compiler does not understand, such as linearized arrays, loops formed by backward GOTO statements, and storing and using constants in arrays. Dusty deck programs have proven to be very challenging for even shared-memory vectorizing and parallelizing compilers. Because of the deep analysis required, they are even more difficult for distributed-memory compilers. It is not a goal of the Fortran D compiler to be able to automatically parallelize these programs for distributed-memory machines. Requiring users to

Program	Data Size	Computation Amount	Computation Type	Communication Messages	Communication Size	Hand-Optimized Fortran D
DGEFA	$O(n^2)$	$O(n^3)$	Pivot	$O(n)$	$O(n)$	0.65–0.91
ERLEBACHER	$O(n^3)$	$O(n^3)$	Pipeline	$O(n)$	$O(n)$	0.50–0.59
SHALLOW	$O(n^2)$	$O(n^2)$	Parallel	$O(1)$	$O(n)$	0.95–0.99
DISPER	$O(n^4)$	$O(n^4)$	Parallel	$O(1)$	$O(n)$	0.97–1.03

Table 6 Inherent Communication and Parallelism in Applications

program in Fortran 90 can help prevent such poor programming practices, and is the approach taken by High Performance Fortran. However, as shown by the poor performance of the CM Fortran compiler, Fortran 90 syntax does not eliminate the need for advanced compile-time analysis and optimization.

5.4 Nature of Applications

We close by considering implications for future success of the Fortran D approach to data-parallel programming. We believe that problems with the immaturity of symbolic analysis, interprocedural analysis, run-time support, and the Fortran D compiler can be solved in the short term. No breakthroughs are required, simply a major investment in development effort.

Problems with dusty deck codes will remain. Simply adding data decomposition specifications to existing sequential, vector, or parallel programs does not ensure they will be compiled by the Fortran D compiler. Users of massively-parallel machines will eventually recognize that current compiler technology cannot automatically extract parallelism from such codes. They should be willing to rewrite important programs *once* in a "clean" machine-independent form using either Fortran 77 or Fortran 90, if advanced compiler techniques will ensure these programs can be executed efficiently across a wide range of machine architectures.

Compilation technology and dusty deck codes, however, are not the key limitations confronting the Fortran D compiler. Instead, it is the amount of parallelism and communication present in the input Fortran D program. Our experiences show that this is the most significant factor determining the success of the Fortran D compiler. Because the Fortran D compiler does not change the input algorithm or data decomposition, there is an inherent amount of communication and parallelism in a Fortran D program. The nature of the application thus dictates the performance achievable by the Fortran D compiler.

Consider the classification of programs and their communication requirements in Table 6. It categorizes the programs we examined by their data, computation, and communication requirements, then compares the effectiveness of the Fortran D compiler against hand-optimized versions for the largest problems measured. As the amount of communication increases, the discrepancy between the Fortran D compiler and hand-optimized codes worsens. The table thus clearly demonstrates the correlation between the amount of communication performed and the success of the Fortran D compiler.

Our experiences with the prototype Fortran D compiler leads us to believe that within a few years, compilers for languages such as Fortran D or High Performance Fortran can match the performance of hand-optimized code for applications with high parallelism and low communication. However, prospects for programs with low parallelism or high communication remain unclear. These applications are much more sensitive to communication overhead, and it will be difficult for the compiler to automatically perform the optimizations programmers apply by hand to achieve high performance. Much additional research will be needed to develop automatic compilation techniques for these problems.

6 Related Work

The Fortran D compiler is a second-generation distributed-memory compiler that integrates and extends previous analysis and optimization techniques. It is similar to ASPAR [19], BOOSTER [24], Callahan-Kennedy [5], FORGE90 [2], P^{3}C [10], SUPERB [12, 32], and VIENNA FORTRAN [6] in that the compilation process is based on a decomposition of the data in the program. The Fortran D compiler can automatically detect and efficiently exploit parallelism in sequential Fortran 77 programs with very few language extensions. Other projects require the user to specify single assignment (CRYSTAL [22], ID NOUVEAU [26]), all parallel loops (KALI [21], MODULA-2* [25]), parallel functions (C* [27], DATAPARALLEL C [15], DINO [28]), parallel code blocks (OXYGEN [29], PANDORE [1]), or parallel array operations (CM FORTRAN [30], PARAGON [7]).

7 Conclusions

An efficient, portable, data-parallel programming model is required to make large-scale parallel machines useful for scientific programmers. We believe that Fortran D provides such a model for distributed-memory machines. This paper describes compiler techniques developed in response to problems posed by linear algebra computations, large subroutines, and whole programs.

The performance of the prototype Fortran D compiler is evaluated against hand-optimized programs on the Intel iPSC/860. Results show reasonable performance is obtained for stencil computations, though room for improvement exists for communication-intensive codes such as linear algebra and pipelined computations. The prototype significantly outperforms the CM Fortran compiler on the CM-5.

The compiler requires symbolic analysis, greater flexibility, and improved optimization of pipelined and linear algebra codes. We believe the Fortran D compilation approach will be competitive with hand-optimized programs for many data-parallel computations in the near future. However, additional effort is required before the compiler will be as effective for partially parallel computations requiring large amounts of communication.

References

[1] 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.

[2] Applied Parallel Research, Placerville, CA. *Forge 90 Distributed Memory Parallelizer: User's Guide*, version 8.0 edition, 1992.

[3] 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.

[4] 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.

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

[6] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[7] 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.

[8] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, To appear 1993.

[9] 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.

[10] E. Gabber, A. Averbuch, and A. Yehudai. Experience with a portable parallelizing Pascal compiler. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[11] G. Geist and C. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal of Scientific Stat. Computing*, 9:639–649, 1988.

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

[13] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[14] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[15] P. Hatcher and M. Quinn. *Data-parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

[16] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, January 1993.

[17] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[18] 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.

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

[20] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

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

[22] 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.

[23] M. Mu and J. Rice. Row oriented Gauss elimination on distributed memory multiprocessors. *International Journal of High Speed Computing*, 4(2):143–168, June 1992.

[24] E. Paalvast, H. Sips, and A. van Gemund. Automatic parallel program generation and optimization from data decompositions. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[25] M. Philippsen and W. Tichy. Compiling for massively parallel machines. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.

[26] 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.

[27] 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.

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

[29] R. Rühl 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.

[30] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, version 1.0 edition, February 1991.

[31] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

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