Preliminary Experiences with the Fortran D Compiler

Seema Hiranandani
Ken Kennedy
Chau-Wen Tseng

CRPC-TR 93307
April 1993

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

Revised September 1993. To appear in *Supercomputing '93*, Portland, OR, November 1993.

Preliminary Experiences with the Fortran D Compiler

Seema Hiranandani
seema@cs.rice.edu

Ken Kennedy
ken@cs.rice.edu

Chau-Wen Tseng
tseng@cs.stanford.edu

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

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Abstract

Fortran D is a 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, multi-reductions, 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. Better analysis, run-time support, and flexibility are required for the prototype compiler to be useful for a wider range of programs.

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, shifting the burden of 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 processors [15].

The success of HPF hinges on the development of compilers that can provide performance satisfactory to users. The goal of our research is to 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 [13, 16, 17]. This paper describes our preliminary experiences with that compiler. Its major contributions include 1) advanced compilation techniques needed for complex loop nests, 2) empirical evaluation of the prototype Fortran D compiler, and 3) identifying necessary improvements for the compiler.

In the remainder of this paper, we briefly introduce the Fortran D language and compiler, then use case studies to illustrate a number of compilation problems and their solutions. We describe experiments comparing the compiler against hand-optimized codes and the CM Fortran compiler. Results are evaluated and used to point out directions for future research. We conclude with a comparison with related work.

This research was supported by the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008. This work was also sponsored by DARPA under contract #DABT63-91-K-0005 & DABT63-92-C-0038, the state of Texas under contract #1059, and the Keck Foundation.

2 Background

2.1 Fortran D Language and Compilation

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. Alignment and distribution statements can be executed, permitting dynamic data decomposition. Details of the language are presented elsewhere [10].

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 partitions computation across processors using the "owner computes" rule—where each processor only computes values of data it owns [6, 12, 25]. It performs a large number of communication and parallelism optimizations based on data dependence. Details of the compilation process are presented elsewhere [13, 16, 17, 27].

2.2 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 [9]. It utilizes existing tools for performing dependence analysis, program transformations, and interprocedural analysis. 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, unbuffered messages
- broadcasts, collective communications
- SUM, PROD, MIN, MAX, MINLOC, MAXLOC reductions
- fine-grain and coarse-grain pipelining (granularity specified by user via compile-time flag)
- relax "owner computes" rule for reductions, private variables
- nonlocal storage in overlaps & buffers

- loop bounds reduction, guard introduction
- global ↔ local index conversion
- interprocedural reaching decompositions & overlaps
- common blocks
- I/O (performed by processor 0)
- generation of calls to the Intel NX/2 and TMC CMMD message-passing libraries

The prototype compiler accepts a subset of Fortran D. Alignment offsets, multidimensional distributions, and dynamic data decomposition are not yet supported. For simplicity, the prototype compiler requires that all array sizes, loop bounds, and number of processors in the target machine be compile-time constants (though triangular loops are supported). 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 and our focus on exploring research directions, especially compile-time optimizations.

3 Compilation Case Studies

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

- Provide robust translation of global/local loop bounds and index variables. Reindex 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.
- Parallelize multidimensional reductions and multi-reductions. Use array kill analysis to eliminate communication for multi-reductions performed by replicated array variables.

3.1 SHALLOW

We begin with SHALLOW, 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 1 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 second dimension block-wise, resulting in a block of columns being assigned 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.

```
{* Original Fortran D Program *}
PROGRAM SHALLOW
 REAL u(n,n),v(n,n),p(n,n)
 REAL unew(n,n),pnew(n,n),vnew(n,n)
 REAL psi(n,n),pold(n,n),uold(n,n),vold(n,n)
 REAL cu(n,n),cv(n,n),z(n,n),h(n,n)
 DECOMPOSITION d(n,n)
 ALIGN u,v,p,unew,pnew,vnew,psi WITH d
 ALIGN 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
 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) = 0.5*(p(i+1,j)+p(i,j))*u(i+1,j)
    cv(i,j+1) = 0.5*(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))
  {* 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))
end
```

Figure 1: SHALLOW: Weather Prediction Benchmark

3.1.1 Statement Groups

We begin by describing a useful construct called *statement groups*. Realistic 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. The Fortran D compiler uses *iteration sets* to represent the set of loop iterations that will be executed by each processor; they are calculated for each statement using the owner computes rule during partitioning analysis. The partition is later instantiated by modifying loop bounds to the union of all iteration sets of statements in a loop, then inserting explicit guards for statements that are executed only on a subset of those iterations.

To aid 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; explicit guards do not 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.

One use of statement groups is to guide guard generation. For instance, consider the example in Figure 2. As-

```
     {* Fortran D Program *}      {* Iteration Sets *}
     DISTRIBUTE a(BLOCK)
     do i = 1,n                     [ 1 : n ]
S1     a(i) = ...                   [ 1 : n/P ]
S2     if (...)                     [ 1 : n/P ]
S3       a(i) = ...                 [ 1 : n/P ]
S4     a(i-1) = ...                 [ 2 : (n/P) + 1 ]
S5     t = ...                      [ 1 : n ]
     enddo
```

Figure 2: Example: Statement Groups

sume that array a is distributed block-wise across P processors. During partitioning analysis, iteration sets are calculated for each statement. According to the owner computes rule each processor executes the assignment at S_1 only on the $[1 : n/P]$ iterations that assign to local data. The same iteration set is assigned to S_3 and then to S_2 since it encloses only S_3. Because these three statements have the same iteration set, they are placed in the same statement group. S_4 receives a slightly different iteration set, and S_5 is assigned all iterations. The i loop is marked as nonuniform because it encloses statements with different iteration setS, then is assigned the union of the iteration sets for S_1–S_5. During code generation, guards will be generated for the first and second statement groups since they are not executed by all processors on local loop iterations.

3.1.2 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 require the creation of several individual point-to-point messages between boundary processors to transfer data required. Putting statements into groups can reduce the number of guards inserted, since all statements in a group share the same guard.

3.1.3 Loop Distribution

Another application of statement groups is to guide *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.

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, except for additional boundary conditions.

3.1.4 Inter-loop Communication Optimizations

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 prototype Fortran D compiler applies message coalescing

```
{* Original Fortran D Program *}
SUBROUTINE DISPER
 LOGICAL lsat(256)
 DOUBLE PRECISION ddx(256,8,8),ddy(256,8,8)
 DOUBLE PRECISION ddz(256,8,8),pmfr(256,8,8,4,5)
 DOUBLE PRECISION 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)) / (0.5*(ddy(i1+1,i2,i3)
       + ddy(i1-1,i2,i3)) + ddy(i1,i2,i3))
      ...
     endif
     endif
     ...
end
```

Figure 3: DISPER: Oil Reservoir Simulation

and aggregation only within a single loop nest. Its output for SHALLOW thus missed many opportunities to combine 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.

3.2 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 processors.

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 in multidimensional arrays [20]. 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.

3.2.1 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 different 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 prototype Fortran D compiler, unnecessary guards and communication in the compiler output were corrected by hand.

3.3 DGEFA

DGEFA, written by Jack Dongarra *et al.* at Argonne National Laboratory, is a key subroutine in LINPACK and 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 4 shows the original program, Figure 5 shows 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 [27]. In addition, we found using statement groups to guide guard generation and identifying MAX/MAXLOC reductions to be necessary.

3.3.1 Guard Generation

DGEFA also demonstrates how statement groups may be used to guide guard generation. During compilation, the Fortran D compiler partitions the statements of the loop body into five statement groups. In Figure 4, 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, an assignment to a replicated array that is executed by all processors.

The third statement group (S_6–S_7) calculates multipliers. Analysis shows that like the first group, it is executed by only one processor on each iteration of the k loop. The fourth group (S_8–S_9) calculates the remaining submatrix. Iterations of the inner j loop are partitioned, but all processors execute at least some iterations of j on each k loop iteration (except for boundary conditions). The fifth and final group (S_{10}) is another assignment to a replicated array that is executed by all processors.

Because loop k contains a variety of iteration sets, it is non-uniform. Its iterations are executed by all processors, and explicit guards are introduced for the first and third statement groups. Note that the third and fourth statement groups contain assignments to t, a replicated scalar. However, the Fortran D compiler determines that t is a private variable with respect to the k loop. With additional analysis, the compiler discovers that it does not need to replicate the assignment on all processors [16].

3.3.2 MINLOC/MAXLOC Reductions

Putting statements S_1 through S_4 in the same statement group requires detecting it as a reduction. The Fortran D compiler recognizes reductions through simple pattern matching. It finds the MAX/MAXLOC reduction in DGEFA 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 uses 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 example also demonstrates 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.

3.4 ERLEBACHER

ERLEBACHER is a 13 procedure, 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 computation phase followed by a forward and backward substitution phase. Figures 6 and 7 illustrate the core computation and substitution phases in 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

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 [17]. Consider the computation in the Z dimension, shown in Figure 6. The Fortran D compiler first distributes the loops enclosing statements S_1–S_4 because they belong to two distinct statement groups. Message vectorization then extracts all communication outside of each loop nest.

Finally, the Fortran D compiler applies vector message pipelining. It is particularly effective here because it moves

```
    {* 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
```

Figure 4: DGEFA: Gaussian Elimination with Pivoting

the *send* C_3 and C_4 before the *recv* 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–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-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 communication is required.

```
{* 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$1, n/4
        t = a(l, j)
        if (l .NE. k) then
          a(l, j) = a(k, j)
          a(k, j) = t
        endif
        do i = k+1, n
          a(i, j) = a(i, j) + t * dp$buf1(i-k)
        enddo
      enddo
    endif
  enddo
  ipvt(n) = n
end
```

Figure 5: DGEFA: Compiler Output

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 7, 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.

3.4.3 Array Kills

If the multi-reduction does not cross any distributed dimensions, no information is transferred between processors. A processor can then evaluate some of the reductions

```
    {* 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))
     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))
    end

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

Figure 6: ERLEBACHER: Computation in Z Dimension

comprising the multi-reduction using local data. Loop bounds reduction is sufficient to partition the reduction; no communication is needed. For instance, a multi-reduction is performed in the Y dimension solution step of ERLEBACHER, shown in Figure 8. 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 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 accessed only on iterations executed locally. In other words, each processor only uses values of *tot* that itselfs computes; values computed by other processors are not needed. 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

Because the computational wavefront traverses across processors in the Z dimension, the Fortran D compiler must efficiently exploit pipeline parallelism [16]. In Figure 7,

```
   {* Original Fortran D Program *}
   SUBROUTINE TRIDVPK
    REAL a(n),b(n),c(n),d(n),e(n),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)
    do j = 1,n
     do i = 1,n
S2   f(i,j,n) = (f(i,j,n)-tot(i,j))*b(n)
    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)
    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)
   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)
    global-sum tot(1:n,1:n)
    if (my$p .EQ. 3) then
     do j = 1,n
      do i = 1,n
       f(i,j,n$) = (f(i,j,n$)-tot(i,j))*b(n)
     do j = 1,n
      do i = 1,n
       f(i,j,n$-1) = f(i,j,n$-1)-e(n-1)*f(i,j,n$)
     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$p+1
      do k = ub$,1,-1
       k$ = k + off$0
       do i = i$,i$+8
        f(i,j,k) = f(i,j,k)-c(k$)*f(i,j,k+1)
                 - e(k$)*r$buf1(j*n+i-n)
      if (my$p .GT. 0)
       send f(i$:i$up, j, 1) to my$p-1
   end
```

Figure 7: ERLEBACHER: Solution Phase in Z Dimension

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 coarse-grain pipeline parallelism, the compiler interchanges the k loop inwards and strip-mines the enclosing i loop.

This example demonstrates two additional features of the Fortran D compiler. First, note that moving the k loop innermost would convert column-wise array accesses into row-wise accesses, resulting in poor data locality. To avoid these situations the prototype compiler leaves the two innermost loops in the original order when applying coarse-grain pipelining. Second, since the non-

```
  {* 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)
   do k = 1,n
    do i = 1,n
S2   f(i,n,k) = (f(i,n,k) - tot(i,k))*b(n)
  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$) + d(j)*f(i,j,k)
   global-concat tot(1:n,1:n)
   do k = 1,n$
    k$ = k + off$0
    do i = 1,n
     f(i,n,k) = (f(i,n,k) - tot(i,k$))*b(n)
  end
```

Figure 8: ERLEBACHER: Solution Phase in Y Dimension

local accesses caused by $f(i,j,n)$ are communicated via a vectorized broadcast, to properly access data in the 1D buffer array the compiler must replace the reference with *r$buf1*$(j*n+i-n)$.

4 Empirical Evaluation of Compiler

To evaluate the status of the prototype Fortran D compiler, 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 Code

We begin by comparing the output of the Fortran D compiler against hand-optimized code 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.

Speedups for different problem and machine sizes are graphically displayed in Figure 9, 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 compiler-generated programs, respectively. Each line represents the speedup for a given problem size.

4.1.1 Results for Stencil Kernels

The hand-optimized stencil kernels are taken from a previous study evaluating the effect of different communication & parallelism optimizations on overall performance [17]. 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.

We found that the code generated for the inner product in Livermore 3 was identical to the hand-optimized version, 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 [5, 17].

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 differences to be due to complications with the scalar i860 node compiler in the parameterized hand-optimized version. We also observed superlinear speedups for some kernels, probably caused by the increase in total cache size with multiple processors.

4.1.2 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 distributed-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.

4.1.3 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 by hand. We also created a hand-optimized version of DISPER by applying aggressive inter-loop message aggregation and unbuffered messages. However, since communication overhead is small, the hand-optimized version only yielded minor improvements (1–3%) for the problem size tested.

4.1.4 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 4, 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

Problem Size	Proc	Fortran D time	Fortran D speedup	Hand-Optimized time	Hand-Optimized speedup	Hand / FortD
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: iPSC/860 Timings for Shallow (in seconds)

Problem Size	Proc	Fortran D time	Fortran D speedup	Hand-Optimized time	Hand-Optimized speedup	Hand / FortD
256 × 8 × 8 × 4	1	*estimated sequential time = 39.0*				
	4	9.971	3.91	10.22	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: iPSC/860 Timings for Disper (in seconds)

Problem Size	Proc	Fortran D time	Fortran D speedup	Hand-Optimized time	Hand-Optimized speedup	Hand / FortD
256 × 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 × 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	145.8	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: iPSC/860 Timings for Dgefa (in seconds)

Problem Size	Proc	Fortran D time	Fortran D speedup	Hand-Optimized time	Hand-Optimized speedup	Hand / FortD
64 × 64 × 64	1	*sequential time = 1.577*				
	2	0.858	1.84	0.805	1.96	0.94
	4	0.721	2.19	0.586	2.69	0.81
	8	0.657	2.40	0.448	3.52	0.68
	16	0.539	2.93	0.311	5.07	0.53
	32	0.613	2.57	0.315	5.00	0.51
96 × 96 × 96	1	*estimated sequential time = 5.3*				
	4	1.517	3.49	1.151	4.60	0.76
	8	1.431	3.70	0.917	5.78	0.64
	16	1.481	3.58	0.813	6.52	0.55
	32	1.334	3.97	0.720	7.36	0.54
128 × 128 × 128	1	*estimated sequential time = 12.6*				
	8	2.738	4.60	1.905	6.61	0.70
	16	2.705	4.65	1.584	7.95	0.58
	32	2.533	4.97	1.347	9.35	0.53

Table 4: iPSC/860 Timings for Erlebacher (in seconds)

on optimizations described in the literature [11, 22]. 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 rarely occur.

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.

The hand-optimized version of Dgefa 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.

4.1.5 Results for ERLEBACHER

As we have seen, Erlebacher requires global communication and contains computation wavefronts that sequentialize parts of the computation. During compilation the Fortran D compiler 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 two optimizations by hand. First, we used interprocedural array kill analysis to eliminate unnecessary global concatenation for local multi-reductions in the X and Y sweeps. Second, we hand-tuned the granularity of coarse-grain pipelining, selecting strip sizes of 16 and 24 rather than the default size of 8. These optimizations yielded speedups of 5–9, improving performance by up to 50% over the Fortran D compiler-generated code.

4.1.6 Analysis of Results

By generating output for Shallow and Disper that virtually matched their hand-optimized versions, the Fortran D compiler has demonstrated its effectiveness for parallel stencil computations, despite not producing the most efficient communication. The 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.

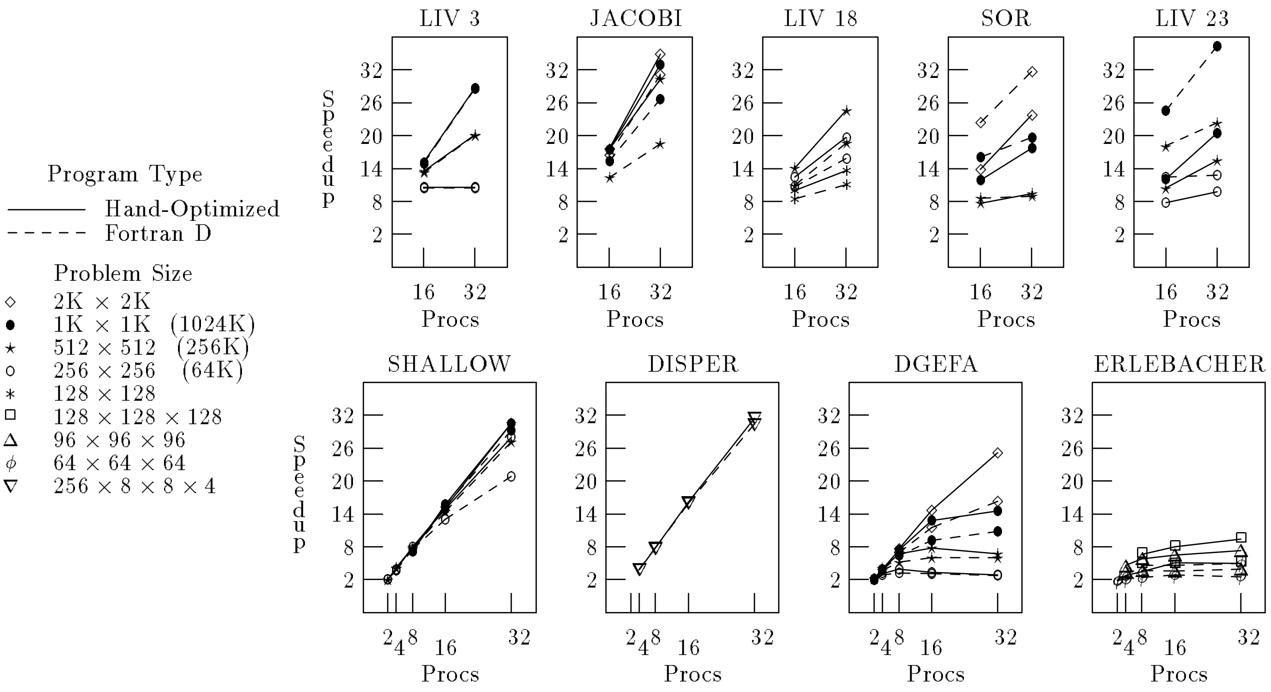

Figure 9: Speedups for Stencils & Programs (iPSC/860)

Instead, the focus should be on improving the flexibility and robustness of the Fortran D compiler, as discussed in Section 5.

In comparison, there is considerable room for improvement when compiling communication-intensive codes such as linear algebra and pipelined computations. Results for DGEFA and ERLEBACHER show that the prototype Fortran D compiler only attains limited speedups. Noticeable performance gains can be achieved through advanced communication optimizations; they are important because communication is performed much more frequently than in parallel stencil computations. The effect of these optimizations on overall execution time increases 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 [17].

To summarize, the Fortran D compiler performs extensive analysis for stencil computations and is able to achieve good speedups. More complex linear algebra and pipelined computations require sophisticated optimizations that are not fully incorporated into the prototype compiler, hence we see only modest speedups.

4.2 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. When combined, the four vector units on each node can outperform the Sparc by a factor of six or more.

In our experiment, 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 *final*, 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*().

4.2.1 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 [27]. 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 were 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 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, and the node-level CMF compiler was insufficiently robust to perform experiments. Fortran D message-passing programs are thus forced to rely on the Sparc processor. For the purpose of comparison, we pro-

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	11.4	3.65
Inner	$256K$	0.020	0.006	0.032	0.006	5.33	1.07
Product	$1024K$	0.079	0.024	0.098	0.007	4.08	0.32
Jacobi	512×512	0.877	0.024	0.236	0.045	9.83	1.87
Iteration	$1K \times 1K$	3.525	0.093	0.766	0.079	8.24	0.85
	$2K \times 2K$	14.14	0.362	2.834	0.159	7.83	0.44
Livermore 18	128×128	0.457	0.019	0.165	0.100	8.68	5.26
Explicit	256×256	1.861	0.054	0.332	0.132	6.15	2.44
Hydrodynamics	512×512	7.554	0.199	0.994	0.163	4.99	0.82
	256×256	1.297	0.029	0.409	0.185	14.1	6.38
Shallow	512×512	5.210	0.129	1.363	0.256	10.6	1.98
	$1K \times 1K$	20.88	0.520	6.159	0.408	11.8	0.78
Successive	512×512	0.376	0.053	17.04	7.559	321	143
Over	$1K \times 1K$	1.519	0.126	116.1	27.39	921	217
Relaxation	$2K \times 2K$	6.134	0.353	209.9	128.6	595	364
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	0.539	10.65	3.604	19.7	6.69
Dgefa	512×512	40.61	2.680	104.8	56.50	39.1	21.1
	$1K \times 1K$	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: CM-5 Timings for Kernels and Programs (in seconds, using 32 processors)

vide 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.

Results indicate that when utilizing only Sparc processors, the CM Fortran compiler produces code that is significantly slower than the corresponding message-passing programs generated by the Fortran D compiler. The difference is pronounced for small data sizes. The CM Fortran compiler fared best on data-parallel computations such as Jacobi, Livermore 18, and Shallow (4–14 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.

4.2.2 Analysis of Results

Direct comparisons are somewhat misleading, since the CM Fortran compiler (2.1 *beta*) directly generates Sparc code instead of using the Sparc Fortran 77 compiler. Additional factors also affect performance, as evidenced by the fact that for small to medium problems the Fortran D compiler is actually faster than the CM Fortran compiler using vector units. First, the code generated by the CM Fortran compiler uses virtual processes, causing extensive run-time calculation of addresses and much unnecessary data movement even for purely local computation. Second, it utilizes a *host-node* model, where a host processor synchronizes global computation on each node. In comparison, the Fortran D compiler utilizes a *hostless* model, eliminating global host-to-node synchronization. Finally, few communication optimizations are performed at compile-time.

We note that it is not completely fair to compare a research tool like the Fortran D compiler against a commercial product like the CM Fortran compiler. Because it is a product, the CM Fortran compiler must be able to accept all legal programs and generate correct code. When targeting distributed-memory machines, guaranteeing correctness for complex computations and data decompositions is quite difficult. In comparison, we were able to concentrate on compile-time optimizations by limiting the range of programs accepted by the Fortran D compiler. The CM Fortran compiler is also handicapped because it is designed to generate code that can be executed on any number of processors, whereas the prototype Fortran D compiler targets a fixed number of processors at compile-time.

Nonetheless, our experiments prove that severe performance penalties result if important compile-time decisions are postponed until run-time. Because the CM Fortran compiler for the CM-5 is relatively new (though it is the most mature commercial compiler available), its performance is sure to improve. We hope that our experiences with the prototype Fortran D compiler will aid the development of future CM Fortran and HPF compilers. However, we believe that scientists will need to be patient, since effective HPF compilers will take time.

5 Compiler Improvements

Our preliminary experiences show that the prototype Fortran D compiler has achieved considerable success in generating efficient code for stencil computations, but needs to improve its optimization of linear algebra and pipelined codes. We find, however, that the compiler must become much more flexible before it can become a successful machine-independent programming model.

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

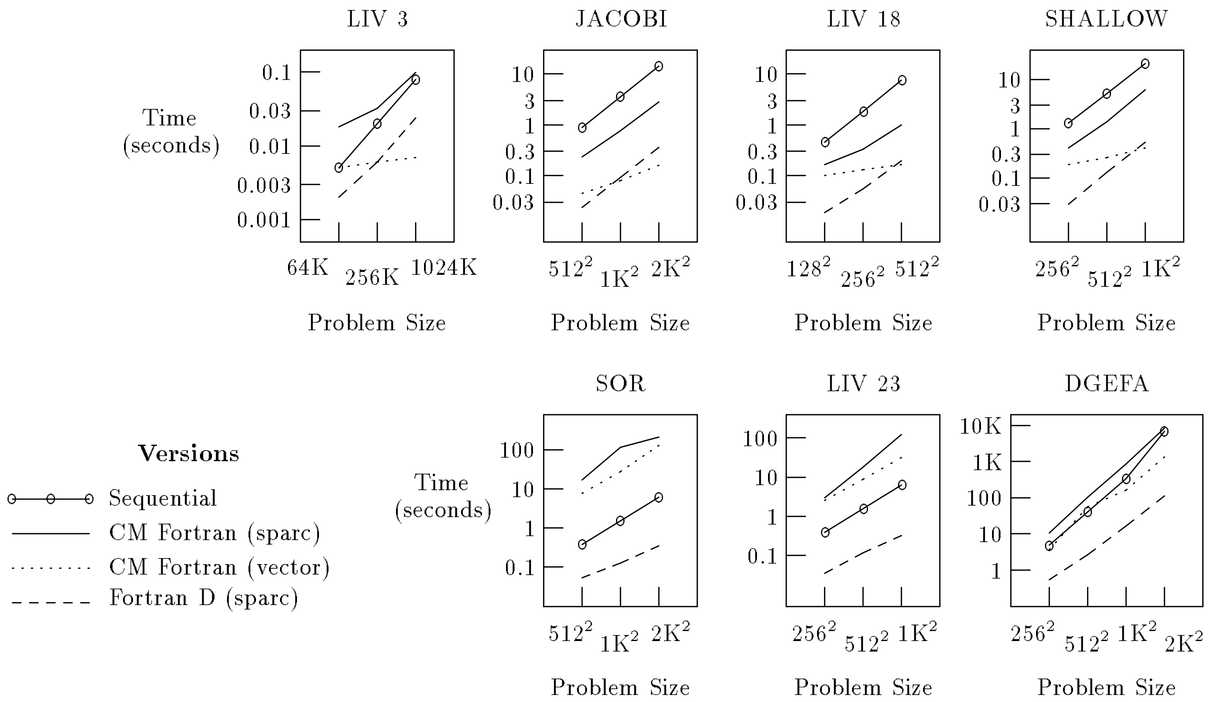

Figure 10: Execution Times (32 Processor Thinking Machines CM-5)

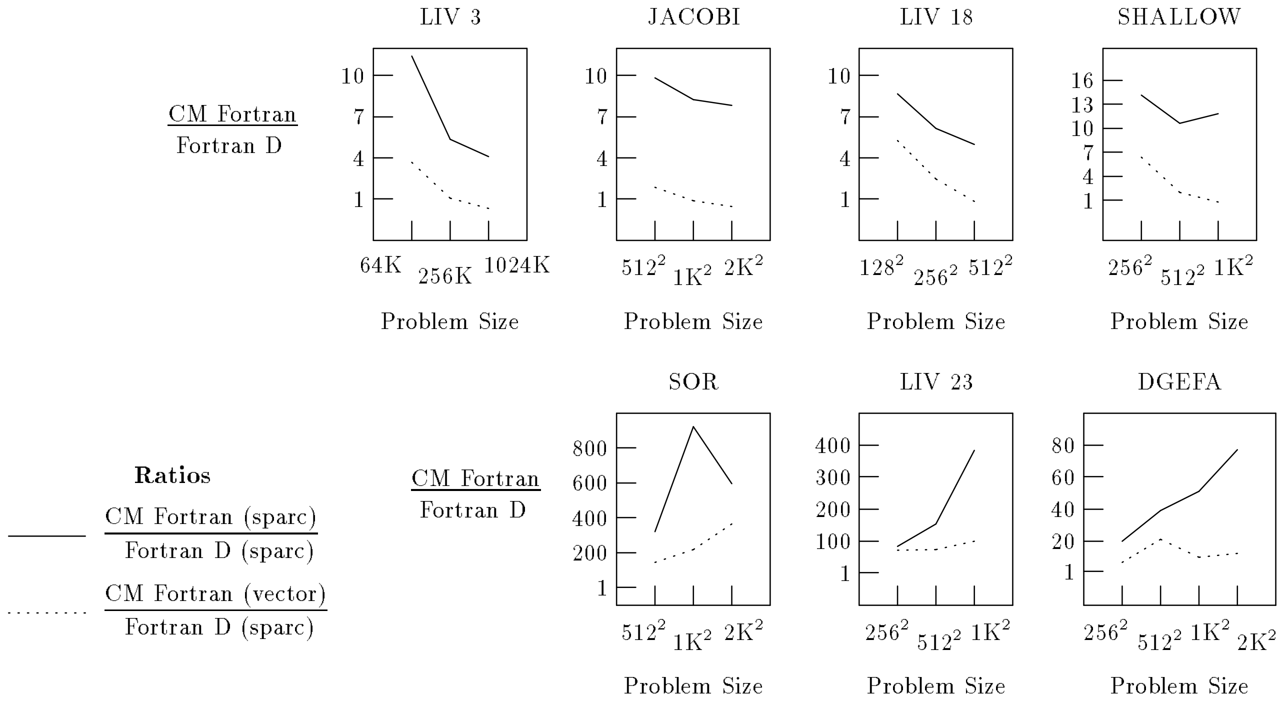

Figure 11: CM Fortran/Fortran D Comparisons (32 Processor Thinking Machines CM-5)

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 handle. The causes for this inflexibility can be categorized as follows.

5.1 Improved Analysis

The lack of symbolic analysis in the prototype 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 real programs, constant propagation, forward expression folding, and auxiliary induction variable substitution all need to be performed before the Fortran D compiler can proceed.

The prototype compiler 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 prototype compiler before many existing programs can be considered.

5.2 Run-time Support

Another problem with the Fortran D compiler is that it relies almost completely on compile-time analysis. The only run-time support it requires are simple 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, partitioning, and communication.

While this approach is necessary for advanced optimizations and generating efficient code, it limits the Fortran D compiler. Real programs, frequently contain components such as indirect references that cannot be easily analyzed at compile-time. In these 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 needs 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 Additional Features

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 currently 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.

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

6 Related Work

The Fortran D compiler is a second-generation distributed-memory compiler that incorporates and extends features from previous compilation systems [6, 12, 19, 21, 25]. Compared with other contemporary systems [2, 3, 7, 8, 18, 24], The Fortran D compiler is less flexible but performs deeper compile-time analysis, many more advanced optimizations, requires fewer language extensions, and relies on less run-time support.

Few researchers have published experimental results for large programs. Pingali & Rogers apply message vectorization, message pipelining, and reduction recognition in ID NOUVEAU to parallelize SIMPLE [25]. Koelbel & Mehrotra are able to parallelize ADI integration in KALI by implicitly applying dynamic data decomposition between computation phases [19]. Olander & Schnabel show that DINO programs can be significantly improved through iteration reordering and pipelining [23].

Bromley *et al.* develop optimizations in CM Fortran compiler for stencils on the CM-2 [4]. By inserting calls to hand-coded microcode routines that apply *unroll-and-jam*, they avoid unnecessary intra-processor data motion, insert communication only for nonlocal data, and improve register usage. The resulting compiler achieves significant improvements in execution speed for a finite-difference seismic model. Hatcher *et al.* demonstrate that DATAPARALLEL C can achieve speedups for large scientific applications on MIMD architectures [14].

Burns *et al.* developed techniques for guiding the use of unbuffered messages on the Alliant CAMPUS/800 using data dependence information [5]. They show that unbuffered messages improve overall performance for a collection of hand-parallelized scientific programs. Their studies validate the effectiveness of selected compiler optimizations for complete programs.

Rühl performed studies on a variety of parallel architectures, demonstrating excellent speedups for the OXYGEN compiler [26]. Amarasinghe & Lam use precise data-flow information for arrays from *last-write-trees* in SUIF to avoid over-communication [1]. They report speedups for Gaussian elimination without pivoting.

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 much 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.

Our experiences show that the prototype Fortran D 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.

8 Acknowledgements

We are grateful to Uli Kremer for providing results for DISPER and our referees for their comments. We wish to thank Geoffrey Fox for use of the TMC CM-5 at Syracuse University. 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.

References

[1] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

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

[3] T. Brandes. Efficient data parallel programming without explicit message passing for distributed memory multiprocessors. Internal Report AHR-92-4, High Performance Computing Center, GMD, September 1992.

[4] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[5] C. Burns, R. Kuhn, and E. Werme. Low copy message passing on the Alliant CAMPUS/800. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

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

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

[8] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. *Journal of Parallel and Distributed Computing*, 16(2):79–91, October 1992.

[9] 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*, 81(2):244–263, February 1993.

[10] 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.

[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] 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.

[14] P. Hatcher, M. Quinn, R. Anderson, A. Lapadula, B. Seevers, and A. Bennett. Architecture-independent scientific programming in Dataparallel C: Three case studies. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[20] U. Kremer and Marcelo Ramé. Compositional oil reservoir simulation in Fortran D: A feasibility study on Intel iPSC/860. Technical Report TR93-209, Dept. of Computer Science, Rice University, September 1993.

[21] 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.

[22] 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.

[23] D. Olander and R. Schnabel. Preliminary experience in developing a parallel thin-layer Navier Stokes code and implications for parallel language design. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[24] 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.

[25] K. Pingali and A. Rogers. Compiling for locality. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, June 1990.

[26] R. Rühl. Evaluation of compiler-generated parallel programs on three multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

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