Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines

Mary W. Hall
Seema Hiranandani
Ken Kennedy
Chau-Wen Tseng

CRPC-TR 91195
November 1991

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

In *Proceedings of Supercomputing '92*,
Minneapolis, MN, November 1993.

Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines *

Mary W. Hall — *mhall@cs.stanford.edu*
Seema Hiranandani — *seema@cs.rice.edu*
Ken Kennedy — *ken@cs.rice.edu*
Chau-Wen Tseng — *tseng@cs.rice.edu*

Center for Integrated Systems, 4070
Stanford University
Stanford, CA 94305

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

Abstract

Algorithms exist for compiling Fortran D for MIMD distributed-memory machines, but are significantly restricted in the presence of procedure calls. This paper presents interprocedural analysis, optimization, and code generation algorithms for Fortran D that limit compilation to only one pass over each procedure. This is accomplished by collecting summary information after edits, then compiling procedures in reverse topological order to propagate necessary information. Delaying instantiation of the computation partition, communication, and dynamic data decomposition is key to enabling interprocedural optimization. Recompilation analysis preserves the benefits of separate compilation. Empirical results show that interprocedural optimization is crucial in achieving acceptable performance for a common application.

1 Introduction

Fortran D is an enhanced version of Fortran that allows the user to specify *data placement*—the partitioning of data onto processors. Its goal is to provide a machine-independent programming model for data-parallel applications that shifts the burden of machine-dependent optimizations to the compiler. Preliminary results show that the Fortran D compiler produces programs that closely approach the quality of hand-written code. However, it requires deep analysis because it must know both *when* a computation may be performed and *where* the data and computation is located. The compiler is thus severely restricted by the limited program context available at procedures. This limitation is unfortunate since procedures are desirable for programming style, modularity, readability, code reuse, and maintainability.

Interprocedural analysis and optimization algorithms have been developed for scalar and parallelizing compilers, but are seldom implemented. We show that interprocedural analysis and optimization can no longer be considered a luxury, since the cost of making conservative assumptions at procedure boundaries is unacceptably high when compiling data-placement languages such as Fortran D. The major contribution of this paper is to demonstrate efficient interprocedural Fortran D compilation techniques. We have begun implementing these techniques in the current compiler prototype.

In the remainder of this paper, we briefly introduce the Fortran D language and illustrate how Fortran D programs are compiled. We illustrate the need for interprocedural compilation and show how the Fortran D compiler is integrated into the ParaScope interprocedural framework. We present analysis, optimization, and code generation algorithms in detail for a number of interprocedural problems, then provide the overall interprocedural compilation algorithm. Recompilation tests are described that preserve the benefits of separate compilation. A case study of DGEFA is used to demonstrate the effectiveness of interprocedural analysis and optimization. We conclude with a comparison with related work.

2 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. The complete language is described in detail elsewhere [15]. As in High Performance Fortran (HPF), each array is implicitly aligned with a default decomposition. This feature allows arrays to be distributed or aligned with other arrays directly without explicit DECOMPOSITION or ALIGN statements if desired.

3 Intraprocedural 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 [22, 23]:

1. **Analyze Program.** Symbolic and data dependence analysis is performed.
2. **Partition data.** Fortran D data decomposition specifications are analyzed to determine the decomposition of each array.
3. **Partition computation.** The compiler partitions computation across processors using the "owner computes" rule—where each processor only computes values of data it owns [7, 29, 33].
4. **Analyze communication.** Based on the computation partition, references that result in nonlocal accesses are marked.

*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.

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 [2, 33].
6. **Manage storage.** "Overlaps" [33] or buffers 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 represented by regular section descriptors (RSDs) [20]. We describe RSDs using Fortran 90 triplet notation.

3.1 Compilation Example

We illustrate the Fortran D compilation process for procedure F1 in Figure 1 onto a machine with four processors. If interprocedural analysis determines that array X in procedure F1 is distributed blockwise, the compiler generates efficient code for F1 as follows.

First, data partitioning assigns the local index set [1:25] to each processor for X. Computation partitioning applies the owner computes rule to the *lhs* $X(i)$, yielding the local iteration set [1:25]. Communication analysis finds that the *rhs* $X(i+5)$ accesses the index set [6:30]. Subtracting off the local index set reveals the nonlocal index set [26:30]. The lack of true dependences on S_1 allows this to be vectorized outside the i loop. Storage management selects overlaps, expanding the local bounds of X to [1:30].

During code generation, the compiler first sets *my$p* to the local processor number, an integer between 0 and 3. It instantiates the data and computation partition by modifying the program text to reduce the array bounds for X to [1:30] and the i loop bounds to [1:25]. An expression is generated for *ub*$1, the upper loop bound, to handle boundary conditions. Finally, the compiler instantiates the RSD for the nonlocal index set by inserting guarded calls to *send* and *recv* routines outside of the i loop. The resulting code is shown in Figure 2.

Without interprocedural analysis, the Fortran D compiler cannot locally determine the data decomposition of X in F1. It is forced to generate code using *run-time resolution* techniques to explicitly calculate the ownership and communication for each reference [7, 29, 33]. As can be seen from Figure 3, run-time resolution produces code that is much slower than the equivalent compile-time generated code. Not only does the program have to explicitly check every variable reference, it generates a message for each nonlocal access. One of the prime goals for interprocedural compilation is to avoid resorting to run-time resolution.

```
PROGRAM P1                        SUBROUTINE F1(X)
  REAL X(100)                       REAL X(100)
  PARAMETER (n$proc = 4)            do i = 1,95
  DISTRIBUTE X(BLOCK)          S1     X(i) = F(X(i+5))
  call F1(X)                        enddo
end                               end
```

Figure 1: Simple Fortran D Program

```
SUBROUTINE F1(X)
  REAL X(30)
  my$p = myproc() {* 0...3 *}
  ub$1 = min((my$p+1)*25,95)-(my$p*25)
  if (my$p .GT. 0) send X(1:5) to my$p-1
  if (my$p .LT. 3) recv X(16:30) from my$p+1
  do i = 1,ub$1
    X(i) = F(X(i+5))
  enddo
end
```

Figure 2: Fortran D Compiler Output

```
SUBROUTINE F1(X)
  REAL X(100)
  my$p = myproc() {* 0...3 *}
  do i = 1,95
    if (my$p .EQ. owner(X(i+5))) then
      send X(i+5) to owner(X(i))
    endif
    if (my$p .EQ. owner(X(i))) then
      recv X(i+5) from owner(X(i+5))
      X(i) = F(X(i+5))
    endif
  enddo
end
```

Figure 3: Run-time Resolution

4 Interprocedural Support in ParaScope

ParaScope is a programming environment for scientific Fortran programmers. It has fostered research on aggressive optimization of scientific codes for both scalar and shared-memory machines [6]. Its pioneering work on incorporating interprocedural optimization in an efficient compilation system has also contributed to the development of the Convex Applications compiler [26]. Through careful design, the compilation process in ParaScope preserves separate compilation of procedures to a large extent. Tools in the environment cooperate so that a procedure only needs to be examined once during compilation. Additional passes over the code can be added if necessary, but should be avoided since experience has shown that examination of source code dominates analysis time. The existing compilation system uses the following 3-phase approach [6, 12, 17]:

1. **Local Analysis.** At the end of an editing session, ParaScope calculates and stores summary information concerning all local interprocedural effects for each procedure. This information includes details on call sites, formal parameters, scalar and array section uses and definitions, local constants, symbolics, loops and index variables. Since the initial summary information for each procedure does not depend on interprocedural effects, it only needs to be collected after an editing session, even if the program is compiled multiple times or if the procedure is part of several programs.

2. **Interprocedural Propagation.** The compiler collects local summary information from each procedure in the program to build an *augmented call graph* containing loop information [18]. It then propagates the initial information on the call graph to compute interprocedural solutions.

3. **Interprocedural Code Generation.** The compiler directs compilation of all procedures in the program based on the results of interprocedural analysis.

Another important aspect of the compilation system is what happens on subsequent compilations. In an interprocedural system, a module that has not been edited since the last compile may require recompilation if it has been indirectly affected by changes to some other module. Rather than recompiling the entire program after each change, ParaScope performs *recompilation analysis* to pinpoint modules that may have been affected by program

Interprocedural Propagation	Code Generation
Call graph $\downarrow$	Local iteration sets $\uparrow$
Loop structure $\downarrow$	Nonlocal index sets $\uparrow$
Array aliasing & reshaping $\downarrow$	Overlaps $\updownarrow$
Scalar & array side effects $\uparrow$	Buffers $\uparrow$
Symbolics & constants $\updownarrow$	Live decompositions $\uparrow$
Reaching decompositions $\downarrow$	Loop-invariant decomps $\uparrow$

Table 1: Interprocedural Fortran D Dataflow Problems

changes, thus reducing recompilation costs [5, 13]. This process is described in greater detail in Section 8.

ParaScope computes interprocedural REF, MOD, ALIAS and CONSTANTS. Implementations are underway to solve a number of other important interprocedural problems, including interprocedural symbolic and RSD analysis. ParaScope also contains support for inlining and cloning, two interprocedural transformations that increase the context available for optimization. *Inlining* merges the body of the called procedure into the caller. *Cloning* creates a new version of a procedure for specific interprocedural information [10, 12].

Existing interprocedural analysis in ParaScope is useful for the Fortran D compiler, but it is not sufficient. The compiler must also incorporate analysis to understand the partitioning of data and computation and to apply communication optimizations. In order to use the above 3-phase approach, additional interprocedural information is collected during code generation and propagated to other procedures in the program. These extensions are described in the rest of the paper.

5 Interprocedural Compilation

As we have seen, interprocedural compilation of Fortran D is needed to generate efficient code in the presence of procedure calls. The Fortran D compilation process is complex. The list of interprocedural data-flow problems that must be solved by the Fortran D compiler is shown in Table 1. Each problem is labeled $\downarrow$, $\uparrow$, or $\updownarrow$ depending on whether it is computed top-down, bottom-up, or bidirectional, respectively. We have carefully structured the Fortran D compiler to perform compilation in a single pass over each procedure for programs without recursion. It has three key points. The first two support compilation in a single pass, the third improves the effectiveness of interprocedural optimization:

- Certain interprocedural data-flow problems are computed first because their solutions are needed to enable code generation. In particular, reaching decompositions information is needed to determine the data partition, the initial step in compiling Fortran D. These problems are solved by gathering local information during editing and computing solutions during interprocedural propagation.

- Other interprocedural data-flow problems depend on data produced only during code generation. For instance, local iteration and nonlocal index sets required for optimizations are calculated as part of local Fortran D compilation. While we could introduce additional local analysis and interprocedural propagation phases to solve these problems, it is much more efficient to combine their calculation with code generation. This approach is possible because the set of problems we want to compute during interprocedural code generation are all bottom-up. By visiting procedures in reverse topological order, the results of analysis for each procedure are available when compiling its callers. Only overlaps need to be handled separately.

- *Delayed instantiation* of the computation partition, communication, and dynamic data decomposition enables optimization across procedure boundaries. In other words, guards, messages, and calls to data remapping routines are not inserted immediately when compiling a procedure. Instead, where legal they are stored and passed to the procedure's callers, delaying their insertion. This technique provides the flexibility needed to perform interprocedural optimization.

```
    PROGRAM P1
      REAL X(100,100),Y(100,100)
      PARAMETER (n$proc = 4)
      ALIGN Y(i,j) with X(j,i)
      DISTRIBUTE X(BLOCK,:)
      do i = 1,100
S1      call F1(X,i)
      enddo
      do j = 1,100
S2      call F1(Y,j)
      enddo
    end

    SUBROUTINE F1(Z,i)
      REAL Z(100,100)
S3    call F2(Z,i)
    end

    SUBROUTINE F2(Z,i)
      REAL Z(100,100)
      do k = 1,100
        Z(k,i) = F(Z(k+5,i))
      enddo
    end
```

Figure 4: Example Fortran D Program

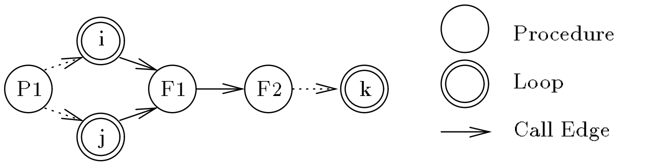


Figure 5: Augmented Call Graph

The remainder of this section presents interprocedural solutions required by the Fortran D compiler and shows how interprocedural information is used during code generation. Section 6 describes additional interprocedural analysis and optimization for efficiently supporting dynamic data decomposition. The overall algorithm is then presented. For clarity, each problem and solution is described separately, even though the compilation process uses the 3-phase ParaScope approach described in the previous section.

5.1 Augmented Call Graph

Most interprocedural problems are solved on the call graph, where nodes represent procedures and edges represent call sites. Since the Fortran D compiler also requires information about interprocedural loop nesting, it uses the *augmented call graph* (ACG) [18]. Conceptually, the ACG is simply a call graph plus *loop nodes* that contain the bounds, step, and index variable for each loop, plus *nesting edges* that indicate which nodes directly encompass other nodes.

For instance, the Fortran D program in Figure 4 produces the ACG shown in Figure 5. The ACG shows that program P1 has two loops, i and j, both of which contain calls to F1. F1 calls F2, which in turn contains loop k. Annotations stored in the ACG show that the formal parameter i in F1 and F2 is actually the index variable for a loop in P1 that iterates from 1 to 100 with a step of 1.

The ACG also contains representations of the formal and actual parameters and their dimensions associated with each procedure and call site. This information is used by interprocedural analysis to translate data-flow sets across calls, mapping formals to actuals and vice versa. An example of this translation is the *Translate* function in Figure 6.

Translation must also deal with *array reshaping* across procedure boundaries. Interprocedural symbolic analysis used in conjunction with *linearization* and *delinearization* of array references can discover standard reference patterns that may be compiled efficiently [4, 17, 20].

5.2 Reaching Decompositions

To effectively compile Fortran D programs, it is vital to know the data decomposition of a variable at every point it is referenced in the program. In Fortran D, procedures inherit the data decompositions of their callers. For each call to a procedure, formal parameters inherit the decompositions of the corresponding actual parameters passed at the call, and global variables retain their decomposition from the caller. A variable's decomposition may also be changed at any point in the program, but the effects of decomposition specifications are limited to the scope of the current procedure and its descendants in the call graph.

Reaching Decompositions Calculation. To determine the decomposition of distributed arrays at each point in the program, the compiler calculates *reaching decompositions*. Locally, it is computed in the same manner as *reaching definitions*, with each decomposition treated as a "definition" [1]. Interprocedural reaching decompositions is a *flow-sensitive* data-flow problem [3, 11] since dynamic data decomposition is affected by control flow. However, the restriction on the scope of dynamic data decomposition in Fortran D means that reaching decompositions for a procedure is only dependent on control flow in its callers, not its callees. The effect of data decomposition changes in a procedure can be ignored by its callers, since it is "undone" upon procedure return.

By taking advantage of this restriction, interprocedural reaching decompositions may be solved in one top-down pass over the call graph using the algorithm in Figure 6. During local analysis, we calculate the decompositions that reach each call site C. Formally,

> LocalReaching$(X) = \{\, \langle D, V\rangle \mid D$ is the set of decomposition specifications reaching actual parameter or global variable V at point X $\}$.

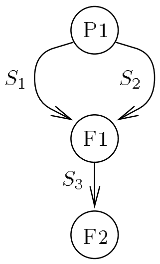
LocalReaching may include elements of the form $\langle \top, V\rangle$ if V may be reached by a decomposition inherited from a caller. $\top$ serves as a placeholder. During interprocedural propagation, we use the call graph and LocalReaching to calculate Reaching(P), the set of decompositions reaching a procedure P from its callers. Formally,

> Reaching$(P) = \{\, \langle D, V\rangle \mid D$ is the set of decomposition specifications reaching formal parameter or global variable V at procedure P $\}$.

The function *Translate* maps actual parameters in the LocalReaching set of a call to formal parameters in the called procedure. Global variables are simply copied, and actual parameters are replaced by the corresponding formal parameters. Reaching(P) is computed as the union of the translated LocalReaching sets for all calls to P. We then update all LocalReaching sets in P that contain $\top$. Each element $\langle \top, V\rangle$ is expanded to $\langle D, V\rangle$, where D is the set of decompositions for variable V in Reaching(P). This step propagates decompositions along paths in the call graph. During code generation the compiler needs to determine which decomposition reaches each variable reference. It repeats the calculation of LocalReaching for each procedure, taking Reaching into account.

{∗ *Local analysis phase* ∗}
for each procedure P **do**
  initialize decomposition of all variables to $\top$
  for each call site C in P **do**
    calculate LocalReaching(C)
  endfor
endfor
{∗ *Interprocedural propagation phase* ∗}
for each procedure P **do** (in topological order)
  calculate Reaching$(P) =$
    $\bigcup_{P \text{ invoked at } C}$ *Translate*(LocalReaching(C))
  clone P if multiple decompositions found
  for each call site C in P **do**
    for each element $\langle \top, X\rangle \in$ LocalReaching(C) **do**
      replace with $\langle D, X\rangle \in$ Reaching(P)
    endfor
  endfor
endfor
{∗ *Interprocedural code generation phase* ∗}
for each procedure P **do** (in reverse topological order)
  calculate LocalReaching for all variables in P
endfor

Figure 6: Reaching Decompositions Algorithm



LocalReaching$(S_1) = \{\, \langle (block, :), X \rangle \,\}$
LocalReaching$(S_2) = \{\, \langle (:, block), Y \rangle \,\}$
LocalReaching$(S_3) = \{\, \langle \top, Z \rangle \,\}$
Reaching(P1) $= \emptyset$
Reaching(F1) $=$ LocalReaching$(S_1) \cup$ LocalReaching(S_2)
Reaching(F2) $=$ LocalReaching(S_3)

Figure 7: Reaching Decompositions

Reaching Decompositions Example. Figure 7 illustrates the reaching decomposition calculation for the program in Figure 4. During the local analysis phase, LocalReaching sets are computed for the call sites S_1, S_2 and S_3. The results for S_1 and S_2 contain the decompositions that reach the actual parameter at the call site. At the first call site S_1, the actual parameter X is distributed row-wise. At the second call site S_2, Y is distributed column-wise. LocalReaching(S_3) is set to the element $\langle \top, Z\rangle$ since the decomposition inherited by procedure F1 reaches Z.

During the interprocedural propagation phase, the call graph is constructed and Reaching sets are computed top-down for program P1 and procedures F1 and F2. Reaching(P1) is the empty set, since P1 has no callers. Reaching(F1) is calculated as the union of LocalReaching for the call sites S_1 and S_2. The *Translate* function maps the decomposition of the actual parameters X and Y at the call sites to the formal Z, resulting in $\langle \{(:, block), (block, :)\}, Z\rangle$. $\top$ for Z in LocalReaching(S_3) is replaced with these column and row distributions from Reaching(F1). Since S_3 is the only call site invoking F2, the resulting data decompositions are also assigned to Reaching(F2). Finally, during local code generation the interprocedural reaching decompositions in Reaching are used to calculate the decomposition for each local variable.

Procedure Cloning. The Fortran D compiler can generate much more efficient code if there is only a single de-

partition calls C invoking P into $\{\pi_1 \dots \pi_n\}$ such that
  Filter(*Translate*(LocalReaching(C)),Appear(P))
  is equal $\forall$ calls C in each partition π_i
if $n > 1$ **then** {* *multiple partitions created* *}
  for each $\pi_i \in \{\pi_1 \dots \pi_n\}$ **do**
    create clone P_i of P
    calculate Reaching(P_i) =
      $\bigcup_{C \ in \ \pi_i}$ *Translate*(LocalReaching(C))
    for each call C in π_i **do**
      replace P with P_i as endpoint of edge
        representing C in call graph
    endfor
  endfor
endif

Figure 8: Procedure Cloning Algorithm

composition reaching an array. We assume that cloning or run-time techniques will be applied locally to ensure that each array has a unique decomposition within each procedure. Procedure cloning may still be necessary if calls to procedure P provide different decompositions for variables that appear in P or its descendants. The procedure cloning algorithm is presented in Figure 8. We define Appear(P) to be the set of formal parameters and global variables appearing in procedure P or its descendants. Formally,

$$\text{Appear}(P) = \text{Gmod}(P) \cup \text{Gref}(P).$$

Gmod and Gref represent the variables modified or referenced by a procedure or its descendants [11]. The value of Appear is readily available from interprocedural scalar side-effect analysis [3, 12]. We also define a function *Filter*(R, V) that removes from R all decompositions elements $\langle D, X\rangle$ where $X \notin V$, returning the remaining decomposition elements.

In the algorithm we partition the calls to P so that calls providing the same decompositions can share the same clone. We use *Filter* to remove reaching decompositions that are not in Appear. This step avoids unnecessary cloning that would expose decompositions for unreferenced variables. A clone of P is produced for each partition, resulting in a unique decomposition for each variable accessed. For instance, the compiler creates two copies of procedure F1 and F2 because they possess two different reaching decompositions for Z. Edges in the call graph are updated appropriately for the clone. In pathological cases, cloning can result in an exponential growth in program size [10]. Under these circumstances, cloning may be disabled when a threshold program growth has been exceeded, forcing run-time resolution instead.

5.3 Partitioning Data and Computation

Recall that a major responsibility of the Fortran D compiler is to partition the data and computation across processors. Reaching decompositions calculated in LocalReaching are translated into *distribution functions* that compute the data partition for each variable. Once the data partition is calculated, it is used with loop information in the ACG to derive the computation partition via the owner computes rule. In the Fortran D compiler, the data and computation partition are represented by local index and iteration sets, respectively. The computation partition is instantiated by modifying the program text to reduce loop bounds and/or introduce explicit guards.

When compiling a procedure, the Fortran D compiler delays local instantiation of the computation partition as much as possible. It first forms the union of all iteration sets for statements in the procedure. Bounds are reduced for loops local to the procedure. Guards are introduced for loops outside the procedure only if local statements have different iteration sets for those loops. Otherwise the compiler simply saves the unioned iteration set, using it to instantiate the computation partition later when compiling the callers. Delayed instantiation enables the compiler to reduce computation partitioning costs by using loop bounds reduction or by merging guards across procedure boundaries. The partitioning algorithm is shown in Figure 9.

{* *Interprocedural code generation phase* *}
for each procedure P **do** (in reverse topological order)
  for each variable V in P **do**
    calculate local index set for V
  endfor
  for each assignment statement S in P **do**
    construct iteration set for S
  endfor
  for each call to Q at site C in P **do**
    assign iteration set of Q to C
  endfor
  instantiate local data and computation partitions
  collect union of all iteration sets in P for callers
endfor

Figure 9: Data and Computation Partitioning Algorithm

Computation Partitioning Example. We illustrate the partitioning process for the code in Figure 4. For simplicity, we assume that procedure F1 contains the k loop. Cloning has already been applied to F1, producing F1$row and F1$col as shown in Figure 10. The compiler computes the local index set for Z to be [1:25,1:100] in F1$row and [1:100,1:25] in F1$col. Disregarding boundary conditions, applying the owner computes rule results in the local iteration sets [1:25,1:100] and [1:95,1:25] for the assignments to $Z(k, i)$ at S_3 and S_4, respectively. Since these are the only computation statements in F1$row and F1$col, they become the iteration sets for the entire procedures as well.

During code generation, the bounds of local loop k are reduced in F1$row, but not for F1$col. The iteration sets for F1$row and F1$col are stored and assigned to the call sites at S_1 and S_2 when compiling P1. This causes the bounds of the j loop enclosing S_2 to be reduced from [1:100] to [1:25], based on the iteration set calculated for F1$col. The result is shown in Figure 10.

5.4 Communication Analysis and Optimization

Once they are calculated, local iteration sets (representing the computation partition) may be used to compute nonlocal accesses. Communication is generated only for nonlocal references in procedure P that cause true dependences carried by loops within P. This may be determined from RSDs and local code. Messages for other nonlocal references will be added when P's callers are later compiled. Communication is instantiated by modifying the text of the program to insert *send* and *recv* routines or collective communication primitives.

To see how this strategy works, first recall that message vectorization uses the level of the *deepest* loop-carried true dependence to combine messages at outer loop levels [2, 33]. Communication for loop-carried dependences is inserted at the beginning of the loop that carries the dependence. Communication for loop-independent dependences is inserted in the body of the loop enclosing both the source and sink of the dependence. If both loop-carried and loop-independent dependences exist at the same level, the loop-independent dependence takes priority [22].

```
  PROGRAM P1
    REAL X(30,100), Y(100,25)
    my$p = myproc() {* 0...3 *}
    if (my$p .GT. 0) send X(1:5,1:100) to my$p-1
    if (my$p .LT. 3) recv X(26:30,1:100) from my$p+1
    do i = 1,100
S1    call F1$row(X,i)
    enddo
    do j = 1,25
S2    call F1$col(Y,j)
    enddo
  end
  SUBROUTINE F1$row(Z,i)
    REAL Z(30,100)
    ub$1 = min((my$p+1)*25,99)-(my$p*25)
    do k = 1,ub$1
S3    Z(k,i) = F(Z(k+5,i))
    enddo
  end
  SUBROUTINE F1$col(Z,i)
    REAL Z(100,25)
    do k = 1,95
S4    Z(k,i) = F(Z(k+5,i))
    enddo
  end
```

Figure 10: Interprocedural Fortran D Compiler Output

Because the program is compiled in reverse topological order, local dependence analysis augmented with interprocedural RSDs representing array uses and definitions can precisely detect all loop-independent dependences and dependences carried by loops *within* the procedure, but not all dependences carried on loops *outside* the procedure. This imprecision is not a problem since the Fortran D compiler delays instantiation of communication for nonlocal references in any case to take advantage of additional opportunities to apply message vectorization, coalescing, aggregation, and other communication optimizations [23].

For interprocedural compilation, the Fortran D compiler first performs interprocedural dependence analysis. References within a procedure are put into RSD form, but merged only if no loss of precision will result. The resulting RSDs may be propagated to calling procedures and translated as definitions or uses to actual parameters and global variables [20]. During code generation, the Fortran D compiler uses intraprocedural algorithms to calculate nonlocal index sets, using the deepest true dependence to determine the loop level for vectorizing communication. If a nonlocal reference is the sink of a true dependence carried by a loop in the current procedure, communication must be generated within the procedure. Otherwise the nonlocal index set is marked and passed to the calling procedure, where its level and location may be determined more accurately and optimizations applied. The algorithm for optimizing communication is shown in Figure 11.

Communication Optimization Example. We illustrate the analysis and optimization techniques used to generate communication for Figure 10. First, the Fortran D compiler uses the local iteration sets calculated for statements S_3 and S_4 to determine the nonlocal index sets for the *rhs* $Z(k+5,i)$. In procedure F1$col, the local iteration set [1:95,1:25] yields the accesses [6:100,1:25]. Since the local index set for Z is [1:100,1:25], all accesses are local and no communication is required.

In procedure F1$row, the local iteration set [1:25,1:100] yields the accesses [6:30,1:100]. Subtracting the local index set produces the nonlocal index set [26:30,1:100]. The compiler determines that communication does not need to generated locally because $Z(k+5,i)$ has no true dependences carried by the local k loop. Instead, it computes the nonlocal index set [26:30,i] for Z and saves it for use when compiling the caller.

When compiling P1, the Fortran D compiler translates the nonlocal index set for Z into a reference to X, the actual parameter for the call to procedure F1$row at S_1. Interprocedural dependence analysis based on RSDs shows that it has no true dependence carried on the i loop either. The compiler thus vectorizes the message outside the i loop, resulting in the nonlocal index set [26:30,1:100]. Guarded messages are generated to communicate this data between processors.

```
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
  for each rhs reference V in P do
    compare with lhs to determine type of communication
    if communication is needed then
      use dependence information to calculate commlevel
      build RSD representing data to be communicated
      insert RSD at loop at commlevel if local
    endif
  endfor
  for each call site in P do
    insert RSDs from call at commlevel if local to P
  endfor
  for each loop in P do
    merge RSDs at loop if no precision is lost
    aggregate RSDs for messages to the same processor
  endfor
  instantiate communication for RSDs at local loops
  collect remaining RSDs for callers
endfor
```

Figure 11: Communication Analysis and Optimization

5.5 Optimization vs. Language Extensions

An important point demonstrated in the previous sections is how delayed instantiation of the computation partition and communication is key to interprocedural optimization. For instance, consider the code generated for Figure 4 if the compiler cannot delay instantiation across procedure boundaries, but must immediately instantiate both the computation and communication partition. For simplicity, again assume that procedure F1 contains the k loop. When compiling F1$row, the Fortran D compiler would need to insert messages inside the procedure to communicate nonlocal data accessed. This code would result in a hundred messages for X[26:30,i], one for each invocation of F1$row, rather than a single message for X[26:30,1:100] in P1. In addition, the compiler would need to introduce explicit guards in F1$col to partition the computation, rather than simply reducing the bounds of the j loop in P1. The resulting program, shown in Figure 12, is much less efficient than the code in Figure 10.

This example also points out limitations for language extensions designed to avoid interprocedural analysis. Language features such as *interface blocks* [32] require the user to specify information at procedure boundaries. These features impose additional burdens on the programmer, but can reduce or eliminate the need for interprocedural analysis. However, current language extensions are insufficient for interprocedural optimizations. This may significantly impact performance for certain computations, as we show in Section 9.

```
  PROGRAM P1
    REAL X(30,100), Y(100,25)
    do i = 1,100
S1    call F1$row(X,i)
    enddo
    do j = 1,100
S2    call F1$col(Y,j)
    enddo
  end
  SUBROUTINE F1$row(Z,i)
    REAL Z(30,100)
    my$p = myproc() {* 0...3 *}
    if (my$p .GT. 0) send X(1:5,i) to my$p-1
    if (my$p .LT. 3) recv X(26:30,i) from my$p+1
    ub$1 = min((my$p+1)*25,99)-(my$p*25)
    do k = 1,ub$1
S3    Z(k,i) = F(Z(k+5,i))
    enddo
  end
  SUBROUTINE F1$col(Z,i)
    REAL Z(100,25)
    if ((i .GT. 0) .AND. (i .LT. 25)) then
      do k = 1,95
S4      Z(k,i) = F(Z(k+5,i))
      enddo
    endif
  end
```

Figure 12: Program with Immediate Instantiation

5.6 Overlap Calculation

The Fortran D compiler uses overlaps and buffers to store nonlocal data fetched from other processors. The number and sizes of temporary buffers required may be propagated up the call graph during code generation as each procedure is compiled. At the top level, the total number and size of buffers is known and can be allocated. Calculating the overlap regions needed for each array is more difficult. The problem is that multidimensional arrays must be declared to have consistent sizes in all but the last dimension, or else inadvertent array reshaping will result. Since using overlaps changes the size of array dimensions, the size of an overlap region must be the same across all procedures. This restriction prevents the use of any single-pass algorithms.

A simple algorithm can compile all procedures and record overlaps used, then perform a second pass over procedures in order to make overlap declarations uniform. To eliminate a second pass over the program, the Fortran D compiler tries to estimate the number and sizes of overlaps by storing constant offsets that appear in array variables subscripts during local analysis. These offsets are propagated in the interprocedural analysis phase to estimate the maximal overlaps needed for each array. Code generation then determines what overlaps are actually needed. The estimate may be updated incrementally if it has not been used in previously compiled procedures. Otherwise the compiler may choose to either utilize buffers or go back and modify array declarations in those procedures. The algorithm for calculating overlaps is described in Figure 13.

Overlap Example. For instance, the overlaps required for X and Y in Figure 10 are calculated as follows. In the local analysis phase, the reference $Z(k+5,i)$ results in the overlap offset Z({+5},0). Interprocedural propagation of overlap offsets translates these offsets for the formal parameter Z to the actual parameters X and Y, discovering that this is the maximum offset for both arrays. Using the results of reaching decomposition analysis, the compiler determines that the first dimension of X and the second dimension of Y are distributed. The overlap offset ({+5},0) yields for X the estimated overlap region [26:30,100]. No overlap is needed for Y since the offset in the distributed dimension is zero. During code generation these overlaps are discovered to be both necessary and sufficient.

```
{* Local analysis phase *}
for each procedure P do
   for each array reference R to variable V do
      mark overlap offset in each dimension
   endfor
endfor
{* Interprocedural propagation phase *}
calculate reaching decompositions
for each procedure P do (in reverse topological order)
   for each array variable V in P do
      merge local overlap offsets and those from calls
      propagate overlap offset to callers
   endfor
endfor
propagate resulting overlap offset estimates down ACG
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
   for each array variable V in P do
      determine actual overlap needed for V
      if actual overlap is greater than estimated then
         use buffer instead, or modify previous procedures
      endif
      mark overlap estimate as used by P
   endfor
   instantiate local overlaps
   collect actual overlap offsets for callers
endfor
```

Figure 13: Overlap Calculation Algorithm

```
PROGRAM P1                  SUBROUTINE F1(X,Xlo,Xhi)
  REAL X(30)                  REAL X(Xlo:Xhi)
  call F1(X,1,30)             do i = 1,25
end                             X(i) = F(X(i+5))
                              enddo
                            end
```

Figure 14: Parameterized Overlaps

Overlap Alternatives. The overlap estimation algorithm is not very precise, but unfortunately is hard to improve without significantly more effort during local analysis. Empirical results will be needed to establish its accuracy in practice. The difficulty posed by overlaps may motivate other storage methods altogether. When analysis is known to be imprecise, the Fortran D compiler may choose to store nonlocal data in buffers instead of overlaps. Using buffers requires additional work by the compiler to separate loop iterations accessing nonlocal data, but this is necessary in any case to perform *iteration reordering*, a communication optimization designed to overlap communication with computation [23]. If the overlap region is noncontiguous, using buffers also has the advantage of eliminating the need to unpack nonlocal data.

Alternatively, the Fortran D compiler can rely on Fortran's ability to specify array dimensions at run time. By adding additional arguments to a procedure, the compiler can produce *parameterized overlaps* for array parameters. Since the extent of all overlaps are known after compiling the main program, they may simply be specified as compile-time constants and passed as arguments to procedures. For instance, Figure 14 shows how parameterized overlaps may be generated for the program in Figure 1. Unfortunately only overlaps for array formal parameters may be parameterized. Overlaps for global arrays found in Fortran common blocks must be determined statically at compile time using the algorithm previously described.

6 Optimizing Dynamic Data Decomposition

As stated previously, users can dynamically change data decompositions in Fortran D. This feature is desirable because phases of a computation may require different data decompositions to reduce data movement or load imbalance. Fortran D assumes the existence of a collection of library routines that can be invoked to remap arrays for different data decompositions. It is the task of the compiler to determine where calls to these mapping routines must be inserted to map affected arrays when executable ALIGN and DISTRIBUTE statements are encountered.

We show that straightforward placement of mapping routines may produce highly inefficient code. In comparison, an interprocedural approach can yield significant improvements. Additional language support is insufficient, because optimization must be performed across procedure boundaries. As with communication and partitioning optimizations, the key to enabling interprocedural optimization is delayed instantiation of dynamic data decomposition. In other words, the Fortran D compiler waits to insert data mapping routines in the callers rather than in the callee.

6.1 Live Decompositions

Because the cost of remapping data can be very high, we would like to recognize and eliminate unnecessary remapping where possible. For instance, consider the calls to procedure F1 at S_1 and S_2 in Figure 15. Array X is originally distributed block-wise, but is redistributed cyclically in F1. If no optimizations are performed, the compiler inserts mapping routines before each call to F1, as displayed in Figure 16a. This code causes array X to be mapped four times for each iteration of loop k. The same problems result if delayed instantiation is not used, because calls to mapping routines are inserted in F1 instead of P1. Analysis can show that the mapping routine for X at S_5 is *dead*, because X is not referenced before it is remapped at S_6. A more efficient version of the program would map array X just twice, before and after the calls to F1, as in Figure 16b.

We pose a new flow-sensitive data-flow problem to detect and eliminate such redundant mappings. We define *live decompositions* to be the set of data decomposition specifications that may reach some array reference aligned with the decomposition. The Fortran D compiler treats each ALIGN or DISTRIBUTE statement as a number of *definitions*, one for each array affected by the statement. A reference to one of these arrays constitutes a *use* of the definition for that array. With this model, the Fortran D compiler can calculate live decompositions in the same manner as *live variables* [1]. Array mapping calls that are not live may be eliminated.

One approach would be to calculate live decompositions during interprocedural propagation. During local analysis, we would collect summary information representing control flow and the placement of data decomposition specifications. We would then need to compute the solution on the *supergraph* formed by combining local control flow graphs with the call graph, taking care to avoid paths that do not correspond to possible execution sequences [27]. To avoid this complexity, we choose instead to compute live decompositions during code generation, when control flow information is available.

Live Decompositions Calculation. Interprocedural live variable analysis has been proven Co-NP-complete in

<pre>   PROGRAM P1
      REAL X(100)
      DISTRIBUTE X(BLOCK)
      do k = 1,T
S1   call F1(X)
S2   call F1(X)
      enddo
      call F2(X)
    end</pre>
	<pre> SUBROUTINE F1(X)
	      REAL X(100)
	      DISTRIBUTE X(CYCLIC)
	      ... = X(...)
	    end
	    SUBROUTINE F2(X)
	      REAL X(100)
	S3 X(...) = ...
	    end</pre>

Figure 15: Dynamic Data Decomposition Example

the presence of aliasing [27]. Even without aliasing, interprocedural live variable analysis can be expensive since it requires bidirectional propagation, causing a procedure to be analyzed multiple times. We rely on two restrictions to make the live decompositions problem tractable for the Fortran D compiler. First, the scope of dynamic data decomposition is limited to the current procedure and its descendants. Second, Fortran D disallows dynamic data decomposition for aliased variables, as discussed in Section 6.4.

By inserting mapping routines in the callers rather than in the callee, we can solve live decompositions in one pass by compiling in reverse topological order during the interprocedural code generation phase. The key insight is that due to Fortran D scoping rules, we know all local dynamic data decompositions are dead at procedure exit. To determine whether they are live within a procedure, we only need information about the procedure's descendants. The compiler cannot determine locally whether calls to mapping routines to restore inherited data decompositions are live, but these mapping calls may be collected and passed to the callers. By delaying their instantiation, we eliminate the need for information about the procedure's callers.

The basic live decompositions algorithm works as follows. We calculate during code generation the following summary sets for each procedure:

- DECOMPUSE(P) = $\{ X \mid X \in$ APPEAR(P) and may use some decomposition reaching P $\}$
- DECOMPKILL(P) = $\{ X \mid X \in$ APPEAR(P) and must be dynamically remapped when P is invoked $\}$
- DECOMPBEFORE(P) = $\{ \langle D, X\rangle \mid X \in$ APPEAR(P) and must be mapped to decomposition D before P $\}$
- DECOMPAFTER(P) = $\{ \langle D, X\rangle \mid X \in$ APPEAR(P) and must be mapped to decomposition D after P $\}$

DECOMPUSE and DECOMPKILL are calculated through local data-flow analysis. They provide interprocedural information for computing live decompositions. DECOMPBEFORE consists of all variables X that need to be mapped before invoking P. DECOMPAFTER consists of all variables X that are mapped in P to some new decomposition, and thus must be remapped when returning from P. Together DECOMPBEFORE and DECOMPAFTER represent dynamic data decompositions from P whose instantiation have been delayed.

We calculate live decompositions by simply propagating uses backwards through the local control flow graph for each procedure [1]. A data decomposition statement is live with respect to a variable X only if there is some path between it and a reference to X that is not killed by another decomposition statement or by DECOMPKILL of an intervening call. Summary sets describe the effect of each procedure call encountered. Formal parameters of P in DECOMPUSE and DECOMPKILL are translated and treated as

```
{* No Optimization *}
   do k = 1,T
S4   map-block-to-cyclic(X)
     call F1(X)
S5   map-cyclic-to-block(X)
S6   map-block-to-cyclic(X)
     call F1(X)
S7   map-cyclic-to-block(X)
   enddo
   call F2(X)
```
(16a)

```
{* Live Decompositions *}
   do k = 1,T
S8   map-block-to-cyclic(X)
     call F1(X)
     call F1(X)
S9   map-cyclic-to-block(X)
   enddo
   call F2(X)
```
(16b)

```
{* Loop-invariant Decomps *}
   map-block-to-cyclic(X)
   do k = 1,T
     call F1(X)
     call F1(X)
   enddo
   map-cyclic-to-block(X)
   call F2(X)
```
(16c)

```
{* Array Kills *}
   map-block-to-cyclic(X)
   do k = 1,T
     call F1(X)
     call F1(X)
   enddo
   mark-as-block(X)
   call F2(X)
```
(16d)

Figure 16: Dynamic Data Decomposition Optimizations

references to actual parameters. DecompBefore and DecompAfter are translated and treated as decompositions affecting variables in P. Decompositions that are dead may be removed. In addition, we can *coalesce* live decompositions if they are identical and their live ranges overlap. All live decompositions except the first may then be eliminated. The live decomposition algorithm is presented in Figure 17.

Live Decompositions Example. Consider how live decompositions are calculated in Figure 15. The Fortran D compiler proceeds in reverse topological order, so we begin with either F1 or F2. For procedure F1, local live and reaching decomposition analysis shows that no incoming decompositions are used. The local redistribution of X to *cyclic* kills the incoming decomposition for X, and requires that X be distributed to *cyclic* before F1 and back to *block* after F1. Since there are no local data decompositions for F2, the incoming decomposition is used for the reference to X. No decompositions are killed in F2 or needed before or after F2. The resulting information is produced:

$$
\begin{aligned}
\text{DecompUse(F1)} &= \emptyset \\
\text{DecompKill(F1)} &= \{\ X\ \} \\
\text{DecompBefore(F1)} &= \{\ \langle (\mathit{cyclic}), X \rangle\ \} \\
\text{DecompAfter(F1)} &= \{\ \langle (\mathit{block}), X \rangle\ \} \\
\text{DecompUse(F2)} &= \{\ X\ \} \\
\text{DecompKill(F2)} &= \emptyset \\
\text{DecompBefore(F2)} &= \emptyset \\
\text{DecompAfter(F2)} &= \emptyset
\end{aligned}
$$

When we compile the main program body P1, we translate all summary sets in terms of local variables. The DecompBefore and DecompAfter sets correspond to potential calls to mapping routines, equivalent to the program shown in Figure 16a. Local live decomposition analysis discovers that there are no uses of the *block* decomposition for X at S_5, allowing it to be eliminated. Local reaching decomposition analysis can then determine that the *cyclic* decompositions for X at S_4 and S_6 are identical. They may then be coalesced, eliminating S_6 to achieve the program shown in Figure 16b.

6.2 Loop-invariant Decompositions

In addition to eliminating non-live decompositions and coalescing identical live decompositions, we can also hoist loop-

```
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
   for each call site in P do
      Translate DecompAfter, DecompBefore,
      DecompUse, DecompKill to actual parameters
   endfor
   calculate local live decompositions
   eliminate dead decompositions
   coalesce identical decompositions
   for each variable X ∈ Appear(P) do
      if original decomposition may reach X then
         add X to DecompUse
      if X must be assigned a decomposition then
         add X to DecompKill
      if X is assigned a decomposition D before
            it uses its inherited decomposition then
         add ⟨D, X⟩ to DecompBefore
      if X is locally assigned a decomposition D that
         differs from the inherited decomposition D' then
         add ⟨D', X⟩ to DecompAfter
   endfor
endfor
```

Figure 17: Live Decompositions Algorithm

invariant decompositions out of loops to reduce remapping. For instance, consider the mapping routines remaining in Figure 16b. If we can hoist the mapping routines, each remapping then occurs once rather than on each iteration of the loop. There are two situations where a decomposition that is live and loop-invariant with respect to variable X may be hoisted out of a loop. They vary slightly from the requirements for loop-invariant code motion [1]:

- If the decomposition is not used within the loop for X, it may be moved after the loop. We verify this condition by comparing LocalReaching and DecompUse for all statements in the loop.
- If the decomposition is the only one used within the loop for X, it may be moved prior to the loop. We verify this condition by checking that no other decompositions reach any occurrences of X.

In the program in Figure 16b, the mapping routine at S_9 is not used within the loop and can be moved after the loop. Now the mapping routine at S_8 is the only decomposition reaching all references to X in the loop, so it can be hoisted to a point preceding the loop, producing the desired program shown in Figure 16c.

6.3 Array Kills

Array kill analysis may be used to determine when the values of an array are *live*. An array whose values are not live does not need to be remapped by physically copying values between processors. Instead, it may be remapped in place by simply marking it as possessing the new decomposition. For instance, suppose that array kill analysis determines that statement S_3 in Figure 15 kills all values in array X. We can then eliminate the cyclic-to-block mapping routine preceding the call to F2, notifying the run-time system instead if necessary. This optimization results in the program shown in Figure 16d.

6.4 Aliasing

Two variables X and Y are *aliased* at some point in the program if X and Y may refer to the same memory location [3]. In Fortran 77, aliases arise through parameter passing, either between reference parameters of a procedure if the same memory location is passed to both formals, or between a global and formal to which it is passed.

Aliasing affects dynamic data decomposition because a variable may be remapped indirectly through one of its aliases. Unfortunately, precise alias analysis is computationally intractable [27]. As a result, the compiler cannot efficiently prove that a decomposition that has been applied to a variable holds for a possible alias. The compiler would have to evaluate reaching decompositions for a variable and all of its potential aliases, reverting to run-time resolution if multiple decompositions reach an access to the variable.

To eliminate the efficiency problems and avoid certain confusing program semantics associated with aliasing, Fortran D requires that a variable and its alias cannot have different reaching decompositions that are live at the same point in the program. This requirement is similar to the specification in the Fortran 77 standard that makes it illegal to write to aliased variables. As a result, the compiler can ignore aliasing when analyzing decompositions since it is illegal to construct a program where remapping a variable's alias changes the decomposition reaching an access to the variable.

Since it is possible to construct a syntactically correct but illegal program, the compiler should warn the programmer of situations where aliasing might cause undefined behavior. We can test the reaching decompositions for each possible alias of a variable at a decomposition statement, warning the programmer if the alias has a different decomposition that is live. Only a warning is produced since the imprecision of alias and live analysis may signal problems in a legal program.

7 Interprocedural Compilation Algorithm

The full interprocedural Fortran D compilation algorithm is shown in Figure 18. It integrates Fortran D compilation techniques with the interprocedural analysis and optimization framework of ParaScope.

```
{* Local analysis phase *}
for each procedure P do
   calculate information for: augmented call graph,
      scalar and array side effects, symbolics,
      reaching decompositions, overlap offsets
endfor
{* Interprocedural propagation phase *}
construct call graph, augment with loop information,
calculate aliasing, symbolics, scalar and array
   side effects, reaching decompositions, cloning,
   overlap offsets
{* Interprocedural code generation phase *}
for each procedure P do (in reverse topological order)
   translate information from call sites in P
   update local loop and subscript information
   perform scalar data-flow analysis, symbolic analysis,
         dependence testing, variable classification
   partition data and computation
   analyze and optimize communication
   calculate number, size, and type of overlaps & buffers
   calculate live, loop-invariant decompositions
   generate code, collect information for callers
endfor
```

Figure 18: Interprocedural Compilation of Fortran D

8 Recompilation Analysis

The Fortran D compiler will follow the ParaScope approach for limiting recompilation in the presence of interprocedural optimization [5, 13]. Recompilation analysis is used to limit recompilation of a program following changes, an important component to maintaining the advantages of separate compilation. Briefly stated, modules only need to be recompiled if they have been edited or if they have been optimized using interprocedural information that is no longer valid.

To determine whether recompilation is needed, the compiler records the interprocedural information used by a compilation. In subsequent compilations, it compares interprocedural information used in the previous compilation with what has been computed in the current compilation. The Fortran D compiler needs to record scalar data-flow analysis results and array side-effects, as well as reaching and live decompositions, overlap offsets, local iteration sets, and nonlocal index sets. The complete list of problems is shown in Table 1 in Section 5.

Recompilation mimics the interprocedural compilation algorithm presented in Figure 18. Local analysis is applied to edited procedures, then interprocedural propagation is performed. Following an initial test to discover which modules have been edited since the previous compilation, we apply *recompilation tests* to interprocedural data-flow information for each module and its call sites. The compiler must also ensure that cloning applied to expose reaching decompositions is still valid; it may decide to form more clones at this time. As soon as one recompilation test fails, the module is marked as needing recompilation.

In the bottom-up pass over the program, if the current node has not been marked for recompilation, the compiler applies recompilation tests on the iteration sets, nonlocal index sets and RSDs at each call site. Depending on the results, some procedures are marked for recompilation. If the current procedures has been marked, it is compiled in the usual manner, producing new interprocedural information to be tested.

8.1 Recompilation Tests

Recompilation tests ensure that interprocedural information used to compile a procedure conservatively approximates the current information. A simple test just verifies that the old information is equal to the new information. However, safe tests that generate less recompilation are possible if we consider how the information will be used. Improved recompilation tests for many scalar data-flow problems are described by Burke and Torczon [5]. To give the flavor of the recompilation tests, we describe the test for reaching decompositions. Let *oldP* be the representation of *P* from the previous compilation. The procedures needing recompilation are those for which the following is true:

$$\begin{aligned}&\mathit{Filter}(\text{R}\textsc{eaching}(\mathit{oldP}),\text{A}\textsc{ppear}(P)) \neq\\ &\mathit{Filter}(\text{R}\textsc{eaching}(P),\text{A}\textsc{ppear}(P))\end{aligned}$$

Filter and APPEAR are described in Section 5.2. They are used to determine whether differences in reaching decompositions actually affect optimization. The test thus marks a procedure *P* for recompilation only if the decomposition reaching a variable appearing in *P* or its descendants changes.

Recompilation tests for other Fortran D interprocedural data-flow problems are simpler. Callers must be recompiled if the local iteration or nonlocal index sets of a procedure have changed, since the callers' guards, loop bounds, or communication may be affected. Similarly, modifications to live or loop-invariant decomposition information requires recompilation of the caller. Changes in array section analysis may affect array kill information, requiring recompilation if array remapping routines were affected in the caller. If overlap offsets for a procedure change but do not exceed the original assigned overlaps, recompilation is not necessary. However, if the new overlap offset is greater than the overlap allocated during code generation, every procedure referencing the array will need to be recompiled to reflect the new overlap offset, not just the callers.

A little more work is needed to calculate the extent of recompilation in the presence of cloning based on reaching decompositions [5, 17]. The compiler maintains a mapping from procedures in the call graph to the list of compiled clones for that procedure. For a procedure that has been cloned, the recompilation test can be applied to all the clones in order to find a match for the procedure. It must also pass recompilation tests for other interprocedural problems.

9 Empirical Results

9.1 Compilation Strategies for DGEFA

This section demonstrates the effectiveness of interprocedural optimization using the routine DGEFA from Linpack, a linear algebra library [14]. DGEFA is also a major component in the Linpack Benchmark Program. DGEFA uses Gaussian elimination with partial pivoting to factor a double-precision floating-point array. A simplified version is shown in Figure 19. DGEFA relies on three other Linpack routines: IDAMAX, DSCAL, and DAXPY. Since arrays are stored in *column-major* order in Fortran, DGEFA performs operations column-wise to provide data locality.

To reduce both communication and load imbalance, we choose a column-wise cyclic distribution of array A. We focus on DAXPY because it performs the majority of the computation. Because the techniques discussed in this paper have not yet been implemented in the Fortran D compiler, we applied them by hand, generating three versions of the program. In the *run-time resolution* version shown in Figure 20, lack of decomposition information implies that processors must determine ownership and communication for individual array elements. In the *interprocedural analysis* program displayed in Figure 21, we assume that reaching decomposition is provided for DAXPY through analysis or language extensions. This information allows us to vectorize messages inside the procedure.

Finally, in the version created by *interprocedural optimization*, interprocedural array section analysis can determine that DAXPY reads a column of A starting at $A(k+1,k)$ and defines a column of A starting at $A(k+1,j)$. Dependence analysis discovers that the two columns never intersect, since $k < j \leq n$, proving that no true dependences are carried by the j loop. Message vectorization can then insert communication outside the j loop altogether, avoiding redundant communication. In addition, we utilize *broadcast* rather than *send*, since the same column is required by all processors. The resulting program is shown in Figure 22.

9.2 Measured Execution Times

For our measurements we used a 32 node Intel iPSC/860 with 8 Meg of memory per node. Each program was compiled under -O4 using Release 2.0 of *if77*, the iPSC/860 compiler. We timed the program for several problem sizes and numbers of processors using *dclock*(). The results are shown in Table 2. Execution time is presented in seconds. We define speedup in the table as follows, given parallel execution time T_{par} and sequential execution time T_{seq}. If $T_{par} < T_{seq}$, speedup is T_{seq}/T_{par}. Otherwise speedup is calculated as $-T_{par}/T_{seq}$. In some cases programs using run-time resolution sent more messages than could be handled by the iPSC/860, causing the program to deadlock. These programs are marked with "*".

We make several observations. First, run-time resolution produces code that is over a hundred times slower than the sequential program. Its performance is not affected by problem size, and degrades as the number of processors increases. Even with interprocedural analysis, the code is five to ten times more expensive than the sequential program and worsens as the number of processors increases. Unlike run-time resolution, its performance improves for larger problem sizes. However, for an 800×800 array, approximately the largest double-precision array possible on a single processor, the resulting code is still five times slower than the equivalent sequential program. Only interprocedural optimization produces positive speedups. After interprocedural optimization we observe a speedup of 8 on 32 processors. Further speedup is limited by the small problem sizes.

```
{* Gaussian Elimination with Partial Pivoting *}
SUBROUTINE DGEFA(n,a,IPVT)
  INTEGER n,IPVT(n),j,k,l
  DOUBLE PRECISION A(n,n), t
  do k = 1, n-1
    l = IDAMAX(n-k+1,A(k,k),1) + k - 1
    IPVT(k) = l
    if (l .NE. k) then
      t = A(l,k)
      A(l,k) = A(k,k)
      A(k,k) = t
    endif
    t = -1.0d0/A(k,k)
    call DSCAL(n-k,t,A(k+1,k))
    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
      call DAXPY(n-k,t,A(k+1,k),A(k+1,j))
    enddo
  enddo
  IPVT(n) = n
end
{* Find Maximum Element in Vector *}
INTEGER FUNCTION IDAMAX(n,DX)
  DOUBLE PRECISION DX(n),dmax
  INTEGER i,ix,n
  dmax = DABS(DX(1))
  do i = 2,n
    if (DABS(DX(i)) .GT. dmax) then
      idamax = i
      dmax = DABS(DX(i))
    endif
  enddo
end
{* Scale a Vector by a Constant *}
SUBROUTINE DSCAL(n,da,DX)
  DOUBLE PRECISION da,DX(n)
  INTEGER i,n
  do i = 1,n
    DX(i) = da*DX(i)
  enddo
end
{* Constant times Vector plus Vector *}
SUBROUTINE DAXPY(n,da,DX,DY)
  DOUBLE PRECISION DX(n),DY(n),da
  INTEGER i,n
  do i = 1,n
    DY(i) = DY(i) + da*DX(i)
  enddo
end
```

Figure 19: Simplified Sequential Version of DGEFA

```
SUBROUTINE DAXPY(n,da,DX,DY)
  do i = 1,n
    if (own(DX(i)) .AND. .NOT. own(DY(i))) then
      send DX(i) to owner(DY(i)))
    endif
    if (own(DY(i)) .AND. .NOT. own(DX(i))) then
      recv DX(i) from owner(DX(i)))
    endif
    if (own(DY(i))) then
      DY(i) = DY(i) + da*DX(i)
    endif
  enddo
end
```

Figure 20: DGEFA: Run-time Resolution

```
SUBROUTINE DAXPY(n,da,DX,DY)
  if (own(DX(1)) .AND. .NOT. own(DY(1))) then
    send DX(1:n) to owner(DY(1)))
  endif
  if (own(DY(1)) .AND. .NOT. own(DX(1))) then
    recv DX(1:n) from owner(DX(1)))
  endif
  if (own(DY(1))) then
    do i = 1,n
      DY(i) = DY(i) + da*DX(i)
    enddo
  endif
end
```

Figure 21: DGEFA: Interprocedural Analysis

```
SUBROUTINE DGEFA(n,a,IPVT)
  do k = 1, n-1
    if (own(A(k+1,k))) then
      broadcast A(k+1:n,k)
    else
      recv A(k+1:n,k) from owner(A(k+1,k))
    endif
    do j = k+1, n
      call DAXPY(n-k,t,A(k+1,k),A(k+1,j))
    enddo
  enddo
end
SUBROUTINE DAXPY(n,da,DX,DY)
  do i = 1,n
    DY(i) = DY(i) + da*DX(i)
  enddo
end
```

Figure 22: DGEFA: Interprocedural Optimization

Problem Size	P	Run-time Resolution		Interprocedural Analysis		Interprocedural Optimization	
		time	*speedup*	*time*	*speedup*	*time*	*speedup*
200 × 200	1	*sequential time* = 0.84 seconds					
	2	125	−149	8.7	−10.4	.57	1.47
	4	104	−124	7.8	−9.3	.45	1.87
	8	129	−154	9.0	−10.7	.43	1.95
	16	144	−171	9.6	−11.4	.46	1.83
	32	152	−181	9.4	−11.2	.52	1.62
400 × 400	1	*sequential time* = 7.16 seconds					
	2	1050	−147	53	−7.4	4.0	1.79
	4	841	−117	54	−7.5	2.6	2.75
	8	1047	−146	63	−8.8	2.0	3.58
	16	1177	−164	67	−9.4	1.8	3.98
	32	1256	−175	68	−9.5	1.9	3.77
800 × 800	1	*sequential time* = 64.5 seconds					
	2	8394	−130	358	−5.6	32.8	1.97
	4	*	*	400	−6.2	18.4	3.51
	8	*	*	467	−7.2	11.7	5.51
	16	*	*	499	−7.7	9.0	7.17
	32	*	*	511	−7.9	8.1	7.96

Table 2: Performance of DGEFA for Intel iPSC/860

Our empirical results show that interprocedural compilation can improve performance by several orders of magnitude for an important application. We do not expect interprocedural optimization to be required in all cases, but for many computations it can make a significant difference.

10 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 [24], Booster [28], Callahan-Kennedy [7], MIMDizer [21], and Superb in that the compilation process is based on a decomposition of the data in the program.

Few other compilation systems have discussed interprocedural issues, especially interprocedural optimization. The CM Fortran compiler utilizes user-defined *interface blocks* to specify a data partition for each procedure [32]. Array parameters are then copied to buffers of the expected form at run-time if needed, eliminating the need for interprocedural analysis. C* [30] and Dataparallel C [19] specify parallelism through the use of parallel functions. Arguments to procedures in Id Nouveau [29] and Kali [25] are labeled with their expected incoming data partition. The user must ensure that the procedure is called only with the appropriately decomposed arguments. Distributed array parameters to *composite procedures* in Dino cause their values to be communicated to the appropriate processors [31]. The user labels parameters as IN or OUT to indicate whether their values are used and/or defined.

Superb performs interprocedural data-flow analysis of parameter passing to classify each formal parameter of a procedure as unpartitioned or having a standard/nonstandard partition [16, 33]. A clone is produced for each possible combination of classification of the procedure parameters. For local compilation, Superb modifies procedures so that arrays are always accessed according to their true number of dimensions, inserting additional parameters where necessary for newly created subscripts.

Vienna Fortran [9] provides data distribution specifications similar to Fortran D. Dynamic data decomposition is permitted; arrays are copied at procedure boundaries if redistribution takes place. Vienna Fortran allows the user to specify additional attributes for each distributed array [8]. *Restore* forces an array to be restored to its decomposition at procedure entry. *Notransfer* causes remapping to be performed logically, rather than actually copying the values in the array. *Nocopy* guarantees that its formal and actual parameters have the same data decomposition. No copies take place, but an error results if different decompositions are encountered. We attempt to achieve the same benefits in the Fortran D compiler through interprocedural analysis and optimization.

11 Conclusions

We believe that data-placement languages such as Fortran D are required to make large-scale parallel machines useful for scientific programmers. This paper shows that interprocedural compilation is needed to fully exploit the benefits of data-placement languages. Efficient interprocedural analysis, optimization, and code generation techniques can be designed that require only one pass over the program. Delaying instantiation of the computation partition, communication, and dynamic data decomposition is key to improving interprocedural optimization. Recompilation analysis preserves the benefits of separate compilation. We have completed reaching decompositions and are implementing the other interprocedural optimizations in the prototype Fortran D compiler. Once finished, we intend to empirically measure the effectiveness of interprocedural analysis and optimization for real scientific programs.

12 Acknowledgements

We wish to thank Kathryn McKinley, Uli Kremer, and our referees for their helpful comments. We are also grateful to the ParaScope group for their assistance. 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. The content of this information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[2] 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.

[3] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, January 1979.

[4] P. Briggs, K. Cooper, M. W. Hall, and L. Torczon. Goal-directed interprocedural optimization. Technical Report TR90-147, Dept. of Computer Science, Rice University, December 1990.

[5] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, to appear.

[6] 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.

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

[8] B. Chapman, P. Mehrotra, and H. Zima. Handling distributed data in Vienna Fortran procedures. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[9] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[10] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, April 1992.

[11] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[12] 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.

[13] K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[14] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide.* SIAM Publications, Philadelphia, PA, 1979.

[15] 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.

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

[17] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, April 1991.

[18] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

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

[20] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[21] R. Hill. MIMDizer: A new tool for parallelization. *Supercomputing Review*, 3(4):26–28, April 1990.

[22] 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.

[23] 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.

[24] 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.

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

[26] J. Loeliger, R. Metzger, M. Seligman, and S. Stroud. Pointer target tracking: An empirical study. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[27] E. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[28] 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.

[29] 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.

[30] 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.

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

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

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