Interprocedural Compilation of Fortran D
(extended version)

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

Rice COMP TR 96-264
June, 1996

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251-1892

A shorter version of this paper will appear in the Journal of Parallel and Distributed Computing, special issue on "Compilation Techniques for Distributed Memory Systems," 1996.

Interprocedural Compilation of Fortran D
(extended version)

Mary W. Hall
mhall@isi.edu
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

Seema Hiranandani
seema@mti.sgi.com
Silicon Graphics Computer Systems
2011 North Shoreline Blvd, Bldg #10
Mountain View, CA 94039

Ken Kennedy
ken@cs.rice.edu
Dept. of Computer Science
Rice University
Houston, TX 77251

Chau-Wen Tseng
tseng@cs.umd.edu
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Fortran D is a version of Fortran extended with data decomposition specifications. It is designed to provide a machine-independent programming model for data-parallel applications and has heavily influenced the design of High Performance Fortran (HPF). In previous work we described Fortran D compilation algorithms for individual procedures. This paper presents an interprocedural approach to analyze data & computation partitions, optimize communication, support dynamic data decomposition, and perform other tasks required to compile Fortran D programs. Our algorithms are designed to make interprocedural compilation efficient. First, we collect summary information after edits to solve important data-flow problems in a separate interprocedural propagation phase. Second, for non-recursive programs we compile procedures in reverse topological order to propagate additional interprocedural information during code generation. We thus limit compilation to a single pass over each procedure body. Third, recompilation analysis ensures that only procedures which have been affected by program changes need to be recompiled, preserving the benefits of separate compilation. We also perform optimizations across procedure boundaries by delaying instantiation of the computation partition, communication, and dynamic data decomposition. Empirical results show that interprocedural optimization is crucial in achieving acceptable performance for a common application code.

The authors performed this research while at Rice University. 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 ARPA under contract #DABT63-91-K-0005 & DABT63-92-C-0038, the state of Texas under contract #1059, and the Keck Foundation.

1 Introduction

MIMD distributed-memory machines such as the Intel Paragon and IBM SP-2 are difficult to program, since users must write message-passing programs that deal with separate address spaces, communication, and synchronization. To address this problem, we developed Fortran D, 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. Fortran D has contributed to the development of High Performance Fortran (HPF), an emerging Fortran standard for programming parallel architectures [28]. Because the HPF INHERIT attribute allows procedures to inherit the data decomposition of its callers in the same manner as in Fortran D, the interprocedural analyses and optimizations presented in this paper are also directly applicable to HPF.

To evaluate the Fortran D and HPF programming model, we implemented a prototype Fortran D compiler in the context of the ParaScope programming environment [8]. Previous work describes the implementation of the prototype compiler [25]; experimental results show that for many programs, the Fortran D compiler produces output that approaches the quality of hand-optimized code [26, 37]. However, it requires deep analysis since programs for distributed-memory machines must explicitly communicate data between processors. The Fortran D compiler must know both *when* a computation may be performed and *where* the data and computation is located. It is thus severely restricted by the limited program context available at procedures. This limitation is unfortunate since procedures are desirable for modularity, readability, code reuse, and maintainability.

Interprocedural analysis and optimization algorithms have been developed for scalar and shared-memory parallelizing compilers, but are seldom implemented due to their complexity and expense. For distributed-memory compilers, interprocedural analysis and optimization can no longer be considered a luxury, since the cost of making conservative assumptions at procedure boundaries is unacceptably high. Interprocedural compilation of data-placement languages such as Fortran D or HPF is complex, because it encompasses all existing interprocedural analysis in shared-memory compilers and requires solving additional new problems. In this paper, we present a comprehensive approach to interprocedural compilation of Fortran D that is both powerful and efficient.

This paper makes a number of original contributions. Previous researchers have developed efficient interprocedural analyses for shared-memory compilers [8], provided some interprocedural analysis for distributed-memory compilers [12], and proposed language extensions in place of interprocedural analysis [7]. However, we are the first to develop for distributed-memory compilers 1) efficient interprocedural analyses and 2) interprocedural optimizations. In particular, this paper is the first to:

- Design and implement an efficient one-pass interprocedural compilation system for distributed-memory machines (Section 3).
- Develop interprocedural partitioning and communication optimizations (Section 3).
- Present techniques for optimizing dynamic data decomposition. (Section 4).
- Empirically demonstrate the importance of interprocedural compilation. By achieving an order of magnitude improvement in the performance of DGEFA, a linear algebra computation, we establish that interprocedural analysis and optimization can be vital for distributed-memory compilers (Section 6).

In addition, we have also developed techniques for optimizing dynamic data decomposition and performing recompilation analysis using our interprocedural compilation framework [17].

In this paper we present a number of designs and algorithms; however, only the augmented call graph, reaching decompositions, and overlap offset analyses have been implemented in the prototype Fortran D compiler. In remaining sections of the paper, we review distributed-memory compilation and interprocedural analysis, then present efficient interprocedural analyses and optimizations for distributed-memory compilers. Next we discuss optimizations for dynamic data decomposition, recompilation analysis, and empirical results from case studies. We finish with a comparison with related work and conclusions.

2 Background

We begin this section with a brief introduction of the Fortran D language. We illustrate how Fortran D programs are compiled, then present the efficient 3-phase interprocedural compilation framework in ParaScope.

2.1 Fortran D Language

In Fortran D, the DECOMPOSITION statement declares an abstract problem or index domain. The ALIGN statement maps each array element onto the decomposition. The DISTRIBUTE statement groups elements of the decomposition and aligned arrays, mapping them to a parallel machine. Each dimension is distributed in a BLOCK, CYCLIC, or BLOCK_CYCLIC manner; the symbol ":" marks dimensions that are not distributed. Because the alignment and distribution statements are executable, dynamic data decomposition is possible; compiler analysis is required to determine whether data layout for each array may be changed at run time [17]. As in 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.

2.2 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 Fortran D compiler applies a compilation strategy based on the "owner computes" rule—where each processor only computes values of data it owns locally [25].

We briefly describe each major step of the compilation process below, details are presented elsewhere [25, 37]. In our discussion, 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), data structures that compactly represent rectangular array sections and their higher dimension analogs [23]. We describe RSDs using Fortran 90 triplet notation.

1. **Analyze Program.** The Fortran D compiler performs scalar dataflow analysis, symbolic analysis, and dependence testing to determine the type and level of all data dependences.
2. **Partition data.** The compiler analyzes Fortran D data decomposition specifications to determine the decomposition of each array in a program. Alignment and distribution statements are used to calculate the array section owned by each processor.
3. **Partition computation.** The compiler partitions computation according to the owner computes rule. The left-hand side (*lhs*) of each assignment statement in a loop nest is used to calculate the set of loop iterations that cause a processor to assign to local data. This represents the work that must be performed by the processor. The compiler relaxes the owner computes rule for reduction operations and assignments to private variables.
4. **Analyze communication.** Once the work partition is computed, it is used to calculate for each right-hand side (*rhs*) reference to a distributed array the nonlocal data accessed by

each processor. References that result in nonlocal accesses are marked since they require communication to be inserted. RSDs are built for the sections of data to be communicated.

5. **Optimize communication.** The compiler examines each marked nonlocal reference, using results of data decomposition, symbolic and dependence analysis to determine the legality of optimizations to reduce communication costs. Communication from multiple references are combined where possible.

6. **Manage storage.** The compiler collects the extent and type of nonlocal data accesses represented by RSDs to calculate the storage required for nonlocal data. For RSDs representing array elements contiguous to the local array section, the compiler reserves storage using *overlaps* created by extending the local array bounds. Otherwise temporary buffers or hash tables are used for storing instances of nonlocal data.

7. **Generate code.** Finally, the Fortran D compiler *instantiates* the communication, data and computation partition determined previously, generating the *single-program, multiple-data* (SPMD) program with explicit message-passing that executes directly on the nodes of the distributed-memory machine. Array and loop bounds are reduced and guards are introduced to instantiate the data and computation partitions. RSDs representing nonlocal data accesses are used to generate buffer routines, *send* and *recv* calls, and collective communication routines. The compiler applies run-time resolution techniques to references not analyzable at compile time.

2.3 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.

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 [25]. As can be seen from the last third of Figure 1, run-time resolution incurs much greater overhead 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.

2.4 Interprocedural Support in ParaScope

The prototype Fortran D compiler described in this paper was developed as part of the ParaScope programming environment, a predecessor to the D System Tools currently under development at

Rice University [1]. ParaScope was designed as a programming environment for scientific Fortran programmers. It has fostered research on aggressive optimization of scientific codes for both scalar and shared-memory machines [8]. Its pioneering work on incorporating interprocedural optimization in an efficient compilation system has also contributed the development of the Convex Applications compiler [30].

Through careful design, the compilation process in ParaScope preserves separate compilation of procedures to a large extent. An *interprocedural compilation framework*, called FIAT, provides an implementation platform for the interprocedural analyses in ParaScope, including those implemented as part of the Fortran D compiler [20]. FIAT collects information from other tools in the environment 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 compilation system uses the following 3-phase approach [8, 16, 20]:

1. **Local Analysis.** When a procedure is edited, ParaScope calculates and stores summary information concerning all local interprocedural effects for the 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 when each procedure is edited, 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 [19]. 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 changes, thus reducing recompilation costs [6, 10]. This process is described in greater detail in Section 5.

Interprocedural analyses of scalar variables have been developed for ParaScope using FIAT, including REF, MOD, ALIAS and CONSTANTS. The same framework has been used to solve a number of other interprocedural problems important to parallelization, including interprocedural symbolic and RSD analysis [22]. 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 [9].

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 & 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 by no means straightforward. They required careful design and are described in the rest of the paper.

Interprocedural Propagation	Code Generation
Call graph ↓	Local iteration sets ↑
Loop structure ↓	Nonlocal index sets ↑
Array aliasing & reshaping ↓	Overlaps ↕
Scalar & array side effects ↑	Buffers ↑
Symbolics & constants ↕	Live decompositions ↑
Reaching decompositions ↓	Loop-invariant decomps ↑

Table 1 Interprocedural Fortran D Data-flow Problems

3 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, since we need to solve many new interprocedural problems in addition to the ones computed by shared-memory parallelizing compilers. The list of interprocedural data-flow problems that must be solved by the Fortran D compiler is shown in Table 1. The loop structure of a program may be assembled directly from summary information, other interprocedural problems are labeled ↓, ↑, or ↕ depending on whether they are 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.

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 4 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.

In this article and our compiler implementation we use regular section descriptors (RSD) to describe sets of computation and data. More general descriptors such as integer polyhedra have also been shown feasible [3, 4], and may be used in place of RSDs in our interprocedural compilation framework. For simplicity, we also assume there is no recursion and arrays are not reshaped across procedure boundaries; more powerful algorithms would be needed to handle these cases.

3.1 Augmented Call Graph

Interprocedural compilation relies on the call graph, an underlying program representation where nodes represent procedures in the program and edges between nodes represent call sites. The Fortran D compiler requires the *augmented call graph* (ACG) for information about interprocedural loop nesting. Conceptually, the ACG is simply a call graph augmented with loop nodes and nesting edges.

The ACG also contains mappings from actuals to formals (and vice versa) at call sites. These mappings enable translation of data-flow information across calls, such as the *Translate* function in Figure 5. Translation must support array reshaping across procedure boundaries, such as linearization, delinearization and reduction in dimensions.

The call graph analysis algorithm used in ParaScope is presented elsewhere [18]. Constructing the augmented call graph involves annotating the nodes in the call graph with templates for their loops and recording the placement of call sites within the loop structure. Each loop is annotated with its bounds, step and index variable. Because bounds and step sizes may be symbolic, the annotation may be a *jump function* which describes the value as a linear function of input variables to the procedure [13]. The jump functions may be replaced with more precise information following the results of interprocedural constant propagation and symbolic analysis.

As an example, the Fortran D program in Figure 2 produces the ACG shown in Figure 3. 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.

3.2 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. We call this the *reaching decompositions* problem. Reaching decomposition analysis was first implemented in the SUPERB compiler [12]. Here we present novel techniques employed by the Fortran D compiler that can calculate reaching decomposition information in the presence of dynamic data decomposition, using only one analysis pass over each procedure body.

In Fortran D and HPF, procedures may 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.

```
{* Fortran D Program *}
PROGRAM P1
  REAL X(100)
  PARAMETER (n$proc = 4)
  DISTRIBUTE X(BLOCK)
  call F1(X)
end

SUBROUTINE F1(X)
  REAL X(100)
  do i = 1,95
S1  X(i) = ℱ(X(i+5))
  enddo
end
```

```
{* Fortran D Compiler Output *}
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
```

```
{* Run-time Resolution 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 1 Intraprocedural Fortran D Compilation Example

```
   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 = 1,95
       Z(k,i) = ℱ(Z(k+5,i))
     enddo
   end
```

Figure 2 Reaching Decomposition Example

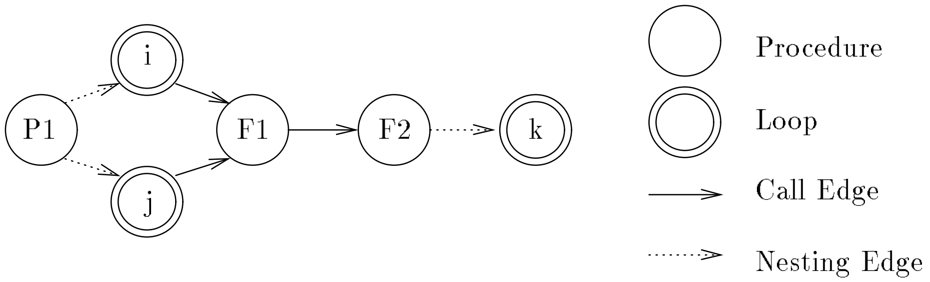
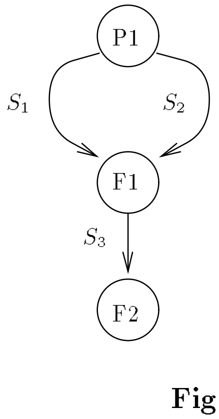


Figure 3 Augmented Call Graph



$\textsc{LocalReaching}(S_1) = \{\, \langle\, (block, :), X \,\rangle \,\}$

$\textsc{LocalReaching}(S_2) = \{\, \langle\, (:, block), Y \,\rangle \,\}$

$\textsc{LocalReaching}(S_3) = \{\, \langle\, \top, Z \,\rangle \,\}$

$\textsc{Reaching}(\mathrm{P1}) = \emptyset$

$\textsc{Reaching}(\mathrm{F1}) = \textsc{LocalReaching}(S_1) \cup \textsc{LocalReaching}(S_2)$

$\textsc{Reaching}(\mathrm{F2}) = \textsc{LocalReaching}(S_3)$

Figure 4 Reaching Decompositions

Reaching Decompositions Calculation. Because of dynamic data decomposition, an array may be decomposed differently within the body of a procedure. We thus need to first summarize the decomposition of each array at every call site. Locally, reaching decompositions may be computed in the same manner as *reaching definitions*, with each decomposition treated as a "definition." Interprocedural reaching decompositions is a *flow-sensitive* data-flow problem [5] since dynamic data decomposition is affected by control flow. However, the restriction on the scope of dynamic data decomposition in Fortran D means that the reaching decompositions for a procedure are 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 dynamic remapping 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 5. During local analysis, we calculate the decompositions that reach each call site C. Formally,

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

$\textsc{LocalReaching}$ 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 $\textsc{LocalReaching}$ to calculate $\textsc{Reaching}(P)$, the set of decompositions reaching a procedure P from its callers. Formally,

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

The function *Translate* maps actual parameters in the $\textsc{LocalReaching}$ 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. The translation function may fail to produce a meaningful decomposition reaching a procedure in the presence of *array reshapes*, whereby the programmer has passed an actual array of different dimension and/or size to the corresponding formal. While some reshapes can be supported, those in which distributed dimensions are reshaped may force the compiler to resort to run-time resolution for accesses to such arrays.

$\textsc{Reaching}(P)$ is computed as the union of the translated $\textsc{LocalReaching}$ sets for all calls to P. We then update all $\textsc{LocalReaching}$ 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 $\textsc{Reaching}(P)$. This phase 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 $\textsc{LocalReaching}$ for each statement in a procedure, taking $\textsc{Reaching}$ into account.

Reaching Decompositions Example. Figure 4 illustrates the reaching decomposition calculation for the program in Figure 2. During the local analysis phase, $\textsc{LocalReaching}$ 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. $\textsc{LocalReaching}(S_3)$ is set to the element $\langle \top, Z\rangle$ since the decomposition inherited by procedure F1 that reaches Z is unknown during the local analysis phase.

During the interprocedural propagation phase, the call graph is constructed and $\textsc{Reaching}$ sets are computed top-down for program P1 and procedures F1 and F2. $\textsc{Reaching}$(P1) is the empty set, since P1 has no callers. $\textsc{Reaching}$(F1) is calculated as the union of $\textsc{LocalReaching}$ 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.

3.3 Procedure Cloning

The Fortran D compiler can generate much more efficient code if there is only a single decomposition reaching an array. *Procedure cloning* creates multiple copies of a procedure customized for different calling context [9]. 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. Cloning to ensure unique reaching decompositions was first applied in SUPERB, but only distinguished between local, standard, and non-standard partitions for array parameters [12]. Here we show how to apply cloning for different data decompositions and filter out unnecessary clones.

The procedure cloning algorithm is presented in Figure 5. 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 [8]. Thus, the value of APPEAR is readily available from interprocedural scalar side-effect analysis [5]. We also define a function *Filter*(R, V), where R is a set of $\langle$ decomposition, variable $\rangle$ pairs, and V is a set of filter variables. *Filter*() 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 [9]. Although this should occur rarely if ever, cloning may be disabled when a threshold program growth has been exceeded, forcing run-time resolution instead.

3.4 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. The data partition determines the portion of the data assigned to each processor. 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 compiler delays local instantiation of the computation partition as much as possible. This enables the reduction of computation partitioning costs by using loop bounds reduction or by merging guards across procedure boundaries. The compiler first forms the union of all iteration sets for statements in the procedure. If local statements have different iteration sets for a loop, explicit guards are introduced to ensure statements are executed on the appropriate loop iterations.

If statements have the same iteration set for a loop in the procedure, guards may be avoided by reducing loop bounds instead. If the loop is outside the procedure, the compiler simply saves the unioned iteration set and passes it to the caller(s), delaying instantiation of the computation partition. Later, when compiling each caller, it may be possible to reduce loop bounds to the local iteration set for the call. Otherwise explicit guards are introduced for the call to ensure it is executed only on iterations in the local iteration set. The partitioning algorithm is shown in Figure 5.

Computation Partitioning Example. We illustrate the partitioning process for the code in Figure 6. Cloning has been applied to F1, producing F1$row and F1$col. 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.

3.5 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 with true dependences that are either carried by loops within P or cross between loop nests in P; their presence may be determined by examining local code and results of interprocedural RSD analysis. 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 [25]. Communication for loop-carried dependences is inserted within 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.

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.

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 [23]. During code generation, the 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 that is either loop-independent or 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 5.

Another advantage of delayed instantiation of partitioning and communication is that it allows both to be customized by the calling procedure. Once a procedure has been cloned to ensure unique reaching decompositions for each parameter accessed, the remaining partitioning and communication analysis for the procedure can be shared by all of its callers, since only components local to the procedure are computed. In other words, each procedure determines its own local partitioning and communication requirements, leaving it to the caller to satisfy the remaining requirements at each call site.

Communication Optimization Example. We illustrate the analysis and optimization techniques used to generate communication for Figure 6. 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 read 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 read 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.

3.6 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 6 if the compiler cannot delay instantiation across procedure boundaries, but must immediately instantiate both the computation and communication partition. 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 is much less efficient, as shown in Figure 6.

This example also points out limitations for language extensions designed to avoid interprocedural analysis. Language features such as *interface blocks* [28] 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 6.1.

3.7 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 variable subscripts during local analysis. These offsets are propagated in the interprocedural analysis phase to estimate the maximal overlaps needed for each array. Analysis must iterate over the call graph because an overlap estimate imposed by accesses to a variable may impact overlaps at other procedures that are not ancestors in the call graph; however, this iterative propagation is efficient as it does not require examination of procedure bodies. 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 5.

Overlap Example. For instance, the overlaps required for X and Y in Figure 6 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.

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. 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 7 shows how parameterized overlaps may be generated for the program in Figure 2. 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.

Reaching Decompositions Algorithm

```
{* Local analysis phase *}
for each procedure P do
   initialize decomposition of all variables to ⊤
   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) =
         ⋃_{P invoked at C} Translate(LocalReaching(C))
   clone P if multiple decompositions found
   for each call site C in P do
      for each element ⟨⊤, V⟩ ∈ LocalReaching(C) do
         replace with ⟨D, V⟩ ∈ 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
```

Procedure Cloning Algorithm

```
partition calls C invoking P into {π_1 ... π_n} such that
   Filter( Translate(LocalReaching(C)),Appear(P))
   is equal ∀ calls C in each partition π_i
if n > 1 then   {* multiple partitions created *}
   for each π_i ∈ {π_1 ... π_n} do
      create clone P_i of P
      calculate Reaching(P_i) =
            ⋃_{C in π_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
```

Data and Computation Partitioning Algorithm

```
{* 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
```

Communication Analysis and Optimization Algorithm

```
{* 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
```

Overlap Estimation Algorithm

```
{* 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
repeat
   for each procedure P do (in reverse topological order)
      for each array variable V in P do
         merge local overlap offsets and offsets from calls
         propagate overlap offset to callers
      endfor
   endfor
   for each procedure P do (in topological order)
      for each array variable V in P do
         propagate overlap offset estimates to callees
      endfor
   endfor
until estimates stablize
{* 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 5 Fortran D Interprocedural Compilation Algorithms

Interprocedural Compilation

```
  {* Fortran D Program *}
  PROGRAM P1                        SUBROUTINE F1(Z,i)
    REAL X(100,100),Y(100,100)        REAL Z(100,100)
    PARAMETER (n$proc = 4)            do k = 1,95
    ALIGN Y(i,j) with X(j,i)            Z(k,i) = F(Z(k+5,i))
    DISTRIBUTE X(BLOCK,:)             enddo
    do i = 1,100                    end
      call F1(X,i)
    enddo
    do j = 1,100
      call F1(Y,j)
    enddo
  end

  {* Fortran D Compiler Output *}
  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)*25,95)-(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
```

Version with Immediate Instantiation

```
  {* Naive Compiler Output *}
  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)*25,95)-(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 6 Interprocedural Fortran D Compilation Example

```
  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 7 Parameterized Overlaps

4 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 run-time library routines that can be invoked to remap arrays for different data decompositions. Routines such as *map-to-block*() and *map-to-cyclic*() can determine the data decomposition of arrays at run-time. They either map an array in place to the desired decomposition if necessary, or simply return if the array is already appropriately partitioned. 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.

4.1 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 8. 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 9a. 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 9b.

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*. 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 [31]. 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 the presence of aliasing [31]. Even without aliasing, interprocedural live variable analysis can be expensive since it requires bidirectional propagation over the call graph, 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 4.4.

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. 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 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 10.

Live Decompositions Example. Consider how live decompositions are calculated in Figure 8. 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:

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

Figure 8 Dynamic Data Decomposition Example

```
{* No Optimization *}
  do k = 1,T
S4  map-to-cyclic(X)
    call F1(X)
S5  map-to-block(X)
S6  map-to-cyclic(X)
    call F1(X)
S7  map-to-block(X)
  enddo
  call F2(X)
```
(9a)

```
{* Live Decompositions *}
  do k = 1,T
S8  map-to-cyclic(X)
    call F1(X)
    call F1(X)
S9  map-to-block(X)
  enddo
  call F2(X)
```
(9b)

```
{* Loop-invariant Decomps *}
map-to-cyclic(X)
do k = 1,T
  call F1(X)
  call F1(X)
enddo
map-to-block(X)
call F2(X)
```
(9c)

```
{* Array Kills *}
map-to-cyclic(X)
do k = 1,T
  call F1(X)
  call F1(X)
enddo
mark-as-block(X)
call F2(X)
```
(9d)

Figure 9 Dynamic Data Decomposition Optimizations

```
{* 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 10 Live Decompositions Algorithm

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

$$
\begin{aligned}
\text{DecompUse(F2)} &= \{\ X\ \} \\
\text{DecompKill(F2)} &= \emptyset \\
\text{DecompBefore(F2)} &= \emptyset \\
\text{DecompAfter(F2)} &= \emptyset
\end{aligned}
$$

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 9a. 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 9b.

4.2 Loop-invariant Decompositions

In addition to eliminating non-live decompositions and coalescing identical live decompositions, we can also hoist loop-invariant decompositions out of loops to reduce remapping. For instance, consider the mapping routines remaining in Figure 9b. 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:

- 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 9b, 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 9c.

4.3 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 8 kills all values in array X. We can then eliminate the map-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 9d.

4.4 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 [5]. In Fortran 77, aliases arise through parameter passing, either between

1) 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

2) Interprocedural propagation phase
construct call graph, augment with loop information,
calculate aliasing, symbolics, scalar and array
side effects, reaching decompositions, cloning,
overlap offsets

3) 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 11 Complete Interprocedural Compilation Algorithm

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 [31]. 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.

4.5 Interprocedural Compilation Algorithm

The full interprocedural Fortran D compilation algorithm is shown in Figure 11. It integrates Fortran D compilation techniques with the interprocedural analysis and optimization framework of ParaScope.

5 Recompilation Analysis and Tests

5.1 Recompilation Analysis

Interprocedural compilation of Fortran D can be made more practical by following the ParaScope approach for limiting recompilation in the presence of interprocedural optimization [6, 10]. 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 3.

Recompilation mimics the interprocedural compilation algorithm presented in Figure 11. 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. Alternatively, changes in interprocedural information may make some clones obsolete, causing their retraction. 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 needs to apply 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 procedure has been marked, it is compiled in the usual manner, producing new interprocedural information to be tested.

5.2 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 [6]. 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:

$$\mathit{Filter}(\textsc{Reaching}(oldP), \textsc{Appear}(P)) \neq \mathit{Filter}(\textsc{Reaching}(P), \textsc{Appear}(P))$$

Filter and Appear are described in Section 3.2. 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 [6, 16]. The compiler can maintain 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.

6 Empirical Results

We have implemented the augmented call graph, interprocedural reaching decompositions, and overlap offsets in the prototype Fortran D compiler. Large programs can thus be automatically parallelized by the compiler with very little user input. The prototype Fortran D compiler automatically propagates reaching decompositions and overlap offsets into each subroutine, followed by local compilation [37]. However, we have not implemented interprocedural optimizations. These optimizations are thus applied manually in the following case studies. For the DGEFA example we also assume the compiler can analyze array reshaping across procedure boundaries based on Fortran sequence and storage association.

6.1 Compilation Strategies for DGEFA

We first demonstrate the effectiveness of interprocedural optimization using the routine DGEFA from Linpack, a linear algebra library. DGEFA is also a major component in Linpackd, 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 12. 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. We applied interprocedural compilation by hand, generating three versions of the program shown in Figure 12. In the *run-time resolution* version, lack of decomposition information implies that processors must determine ownership and communication for individual array elements. In the *interprocedural analysis* version, we assume that reaching decomposition information 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 RSD 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.

6.1.1 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*(). Results are tabulated in Figure 13. 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 "$*$".

Timings are also displayed graphically in Figure 13. Each graph represents a single problem size. Execution times are plotted logarithmically along the Y-axis in seconds. The number of processors is plotted along the X-axis. Results for perfect or ideal speedup are included for purposes of comparison. 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. Run-time resolution is thus too expensive to employ except for very infrequently executed sections of the program.

Simplified Sequential Version of DGEFA

```
{* Gaussian Elimination w/ 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
```

Compiled Versions of DGEFA

```
{* Run-time Resolution *}
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

{* Interprocedural Analysis *}
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

{* Interprocedural Optimization  *}
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 12 DGEFA Example

Prob Size	P	Run-time Resolution *time*	Run-time Resolution *speedup*	Interprocedural Analysis *time*	Interprocedural Analysis *speedup*	Interprocedural Optimization *time*	Interprocedural Optimization *speedup*
	1	*sequential time* = 0.84 seconds					
200	2	125	−149	8.7	−10.4	.57	1.47
×	4	104	−124	7.8	−9.3	.45	1.87
200	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
	1	*sequential time* = 7.16 seconds					
400	2	1050	−147	53	−7.4	4.0	1.79
×	4	841	−117	54	−7.5	2.6	2.75
400	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
	1	*sequential time* = 64.5 seconds					
800	2	8394	−130	358	−5.6	32.8	1.97
×	4	*	*	400	−6.2	18.4	3.51
800	8	*	*	467	−7.2	11.7	5.51
	16	*	*	499	−7.7	9.0	7.17
	32	*	*	511	−7.9	8.1	7.96

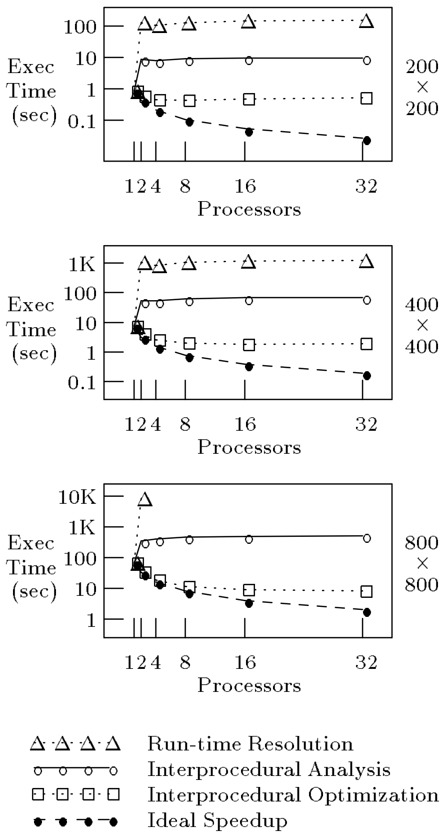


Figure 13 Execution Times and Speedups for DGEFA (Intel iPSC/860)

Surprisingly, the code produced with interprocedural analysis is five to ten times more expensive than the sequential program, worsening as the number of processors increases. Unlike run-time resolution, its performance improves for larger problem sizes. However, even 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.

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.

6.2 Compilation of ERLEBACHER

For our second example we consider ERLEBACHER, an 11 procedure, 800 line program successfully compiled by the Fortran D compiler. ERLEBACHER is a program for solving partial differential equations written by Thomas Eidson at the Institute for Computer Applications in Science and Engineering. It performs tridiagonal solves using ADI integration, resulting in computational wavefronts across all three dimensions of the major data arrays. To convert ERLEBACHER into a Fortran D program, we only needed to insert four lines of directives in the main program, as shown in Figure 15.

```
{* Fortran D Program *}
PROGRAM ERLEBACHER
  COMMON /dvars/ U(N,N,N),DUX(N,N,N),DUY(N,N,N),DUZ(N,N,N)
  COMMON /ivars/ X(N),Y(N),Z(N)
  COMMON /grids/ DX,DY,DZ
  COMMON /times/ DT,DTX,DTY,DTZ
  DOUBLE PRECISION DT,DTX,DTY,DTZ

  {* Fortran D Statements *}
  PARAMETER (n$proc = 4)
  DECOMPOSITION D(N,N,N)
  ALIGN U,DUX,DUY,DUZ WITH D
  DISTRIBUTE D(:,:,BLOCK)

  CALL setgrid
  CALL dotest
  CALL prntout
end
```

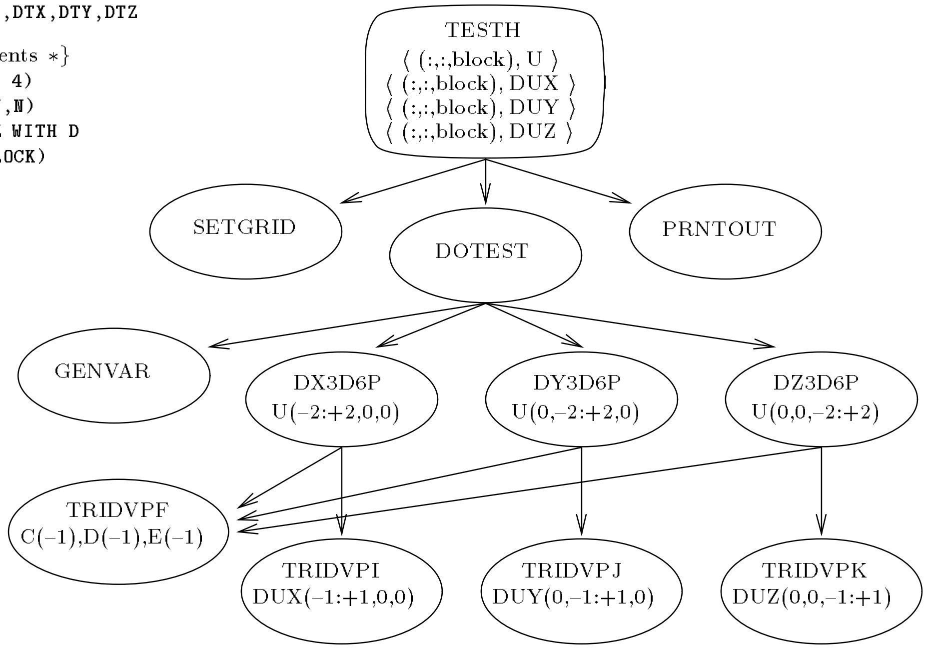


Figure 14 ERLEBACHER and its Annotated Call Graph

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

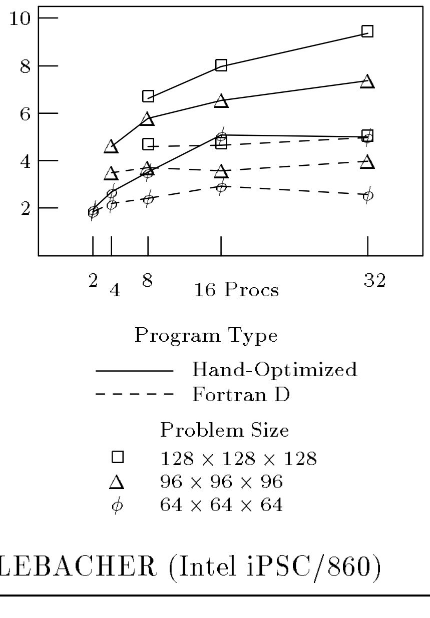


Figure 15 Execution Times and Speedups for ERLEBACHER (Intel iPSC/860)

Here we describe how the Fortran D compiler automatically compiled ERLEBACHER. During local analysis, the compiler first computes local reaching decomposition information and overlap offset estimates. In the main program, the local decomposition of the arrays U, DUX, DUY, and DUZ are found to be identically aligned and distributed (:,:,BLOCK) at all three call sites. For all other procedures, the compiler sets LocalReaching to $\top$, since no local data decomposition specifications were found. All incoming decompositions are thus propagated to callees.

The Fortran D compiler also calculates initial overlap offset estimates for each procedure. Local overlap offsets of (−2:+2) are found in the first, second, and third dimensions of U in procedures DX3D6P, DX3D6P, and DX3D6P, respectively. Local overlap offset estimates of (−1:+1) are found in the first dimension of DUX in TRIDVPI, second dimension of DUY in TRIDVPJ, and third dimension of TRIDVPK. Additionally, an overlap offset of (−1) is found for B, C, and D in TRIDVPF.

During interprocedural propagation, the Fortran D compiler first builds the augmented call graph and annotates it with local information, as shown in Figure 14. No procedure invocations are found to be enclosed in loops. Reaching decompositions are propagated for all array parameters; they turn out to be unique for all procedures, so cloning is not necessary. The resulting information shows that all 3D arrays are identically aligned and distributed block-wise in the third dimension (:,:,BLOCK) for the entire program.

Next, overlap offset estimates are propagated for each array to find the maximum extent of nonlocal accesses in each dimension. Offsets for arrays B, C, and D are not needed because those arrays are not distributed. Similarly, overlap offsets for the first two dimensions of U, DUX and DUY may be discarded because those dimensions are local. For U the compiler finds the maximum estimated overlap offset is (0,0,–2:+2), for DUZ it finds (0,0,–1:+1). These overlap offsets are stored and used during code generation. Finally, the procedures are compiled in a bottom-up order during code generation. The estimated overlap offsets are found to be correct. There are no opportunities for applying interprocedural partitioning or communication optimizations in ERLEBACHER.

Without reaching decomposition information, the compiler would have to generate inefficient run-time resolution code. With interprocedural analysis, the Fortran D compiler can achieve limited speedup. The performance of the output program can be improved with two additional hand optimizations. Interprocedural array kill information can be used to eliminate global concatenates for replicated arrays, and the granularity of pipeline parallelism may be adjusted for the iPSC/860 architecture [26]. The execution times and speedups for the original and hand-optimized codes are shown in Figure 15.

7 Related Work

Previous researchers have studied efficient interprocedural analysis for shared-memory machines or compiling for distributed-memory machines, but usually not in combination [17]. Others have focused on efficiently solving classic data-flow problems interprocedurally [11, 32] or precise interprocedural analyses of array accesses [14, 15, 22, 23, 24, 27]. Several recent distributed-memory compilation systems employ integer polyhedra instead of RSDs for greater flexibility [3, 4].

Few other distributed-memory 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 [36]. Array parameters are then copied to buffers of the expected form at run-time if needed, eliminating the need for interprocedural analysis. C* [34] and Dataparallel C [21] specify parallelism through the use of parallel functions. Arguments to procedures in Id Nouveau [33] and Kali [29] 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 [35]. The user labels parameters as IN or

OUT to indicate whether their values are used and/or defined.

The systems most similar to the Fortran D compiler are SUPERB and its successor, VIENNA FORTRAN. SUPERB is the first distributed-memory compiler that performs interprocedural analysis [12]. It analyzes parameter passing to classify each formal parameter of a procedure as unpartitioned or having a standard/nonstandard partition. A clone is built 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. Compared with the Fortran D compiler, reaching decomposition analysis and cloning is simplified in SUPERB because it does not provide dynamic data decomposition and only supports generalized BLOCK distributions. The efficiency of its interprocedural analysis is unknown but appears to require multiple passes.

VIENNA FORTRAN provides data distribution specifications similar to Fortran D and permits dynamic data decomposition. Arrays are copied at procedure boundaries if redistribution takes place. It provides extensive language features not present in Fortran D. In VIENNA FORTRAN, users may specify additional attributes for each distributed array [7]. *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.

The efficiency of the interprocedural analysis in VIENNA FORTRAN is not discussed; no interprocedural optimizations are supported. In comparison, we show how to perform interprocedural analyses and optimizations in the Fortran D compiler in one pass. There are no published performance results for the VIENNA FORTRAN compiler, even for individual procedures. We are thus unable to compare the efficiency and effectiveness of its interprocedural system with that of the Fortran D compiler.

Agrawal and Saltz have developed an interprocedural partial redundancy elimination algorithm within the Fortran D compilation system and applied it to optimizing the placement of communication preprocessing statements used for supporting irregular accesses [2]. Their work focuses on one aspect of interprocedural compilation of Fortran D, namely communication optimization, and solves it for a more general class of applications, including recursive programs.

HPF adopted the ALIGN and DISTRIBUTE data decomposition model from Fortran D, as well as its restriction that prevents dynamic data decomposition in a procedure from becoming visible in its callers [28]. It allows procedures to inherit the data decomposition of its callers in the same manner as in Fortran D, but in addition extends Fortran 90 interface blocks to permit the user to specify data decompositions for procedure arguments. HPF thus supports separate compilation to a much greater extent than Fortran D. Our work on interprocedural compilation of Fortran D is directly applicable to HPF compilers, since it shows we can relax the need for explicitly declaring data decompositions at procedure boundaries, and also potentially improve performance by performing optimizations across procedure boundaries. A commercial compiler developer, Applied Parallel Research (APR), has extended HPF by including interprocedural reaching decomposition analysis in their FORGExplorer/xHPF compilation system.

8 Conclusions

We believe that data-placement languages such as Fortran D and HPF are required to make large-scale parallel machines useful for scientific programmers. We show that interprocedural compilation is needed to fully exploit the benefits of data-placement languages. Among its other original contributions, this paper is the first to present 1) efficient algorithms for performing interprocedural analysis required by distributed-memory compilers, and 2) interprocedural optimizations for distributed-memory machines.

We describe in this paper how to design efficient interprocedural analysis, optimization, and code generation techniques 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 can preserve most of the benefits of separate compilation.

Experiments indicate that interprocedural communication optimization is essential for DGEFA, a version of Gaussian elimination with partial pivoting from Linpack. On the Intel iPSC/860, extracting communication across procedure boundaries improves performance by orders of magnitude. We have implemented interprocedural reaching decompositions and overlap offsets in the prototype Fortran D compiler, allowing it to compile entire scientific programs with very little user effort. Our interprocedural compilation techniques should prove useful for the commercial HPF compilers under development.

9 Acknowledgements

We wish to thank Kathryn McKinley, Uli Kremer, and Michael Wolfe for their helpful comments, and John Mellor-Crummey for pointing out an error in our overlap calculation algorithm. We are grateful to the ParaScope group for their assistance with the underlying software infrastructure. 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] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, and C.-W. Tseng. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, Fall 1994.

[2] G. Agrawal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

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

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[5] 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.

[6] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[7] 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.

[8] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[9] K. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, February 1993.

[10] 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.

[11] E. Duesterwald, R. Gupta, and M. Soffa. Demand-driven computation of interprocedural data-flow. In *Proceedings of the Twenty-second Annual ACM Symposium on the Principles of Programming Languages*, San Francisco, CA, January 1995.

[12] M. Gerndt. Array distribution in SUPERB. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece, June 1989.

[13] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[14] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[15] M. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Interprocedural analysis for parallelization. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[16] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Dept. of Computer Science, Rice University, April 1991.

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

[18] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

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

[20] M. W. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

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

[22] P. Havlak. *Interprocedural symbolic analysis*. PhD thesis, Rice University, Dept. of Computer Science, May 1994.

[23] 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.

[24] M. Hind, M. Burke, P. Carini, and S. Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.

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

[26] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[27] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[28] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[29] 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.

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

[31] 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.

[32] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the Twenty-second Annual ACM Symposium on the Principles of Programming Languages*, San Francisco, CA, January 1995.

[33] A. Rogers and K. Pingali. Compiling for distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):281–298, March 1994.

[34] 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.

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

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

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

Biographies

Mary Hall

Mary W. Hall has recently joined USC's Information Sciences Institute. Her research interests focus on compiler support for high-performance computing, particularly interprocedural analysis and automatic parallelization. Dr. Hall received a B.A. in computer science and mathematical sciences in 1985, an M.S. and Ph.D. in computer science in 1989 and 1991, respectively, all from Rice University. Prior to joining USC/ISI, Dr. Hall was a Visiting Assistant Professor and Senior Research Fellow in the Department of Computer Science at Caltech. She previously held positions as Research Scientist at Stanford University working with the SUIF compiler group and in the Center for Research on Parallel Computation at Rice University.

Seema Hiranandani

Seema Hiranandani graduated cum laude from Smith College in 1988 with a B.A. in Computer Science and was awarded the Bert Mendelson Prize for excellence in Computer Science at graduation. She continued her studies and earned an M.S. in Computer Science from the University of Massachusetts at Amherst in 1989. Her research interests are in the field of compilers and parallel computation. She has been a major contributor in the design and implementation of the Fortran D compiler developed at Rice University where she worked from 1990-1994. Currently she is a member of technical staff at Silicon Graphics Computer Systems.

Ken Kennedy

Ken Kennedy received a B.A. in mathematics from Rice University in 1967, an M.S. in mathematics from New York University in 1969, and a Ph.D. in computer science from New York University in 1971. Since 1971, he has been a faculty member at Rice University. Professor Kennedy served as Chair of the Department of Computer Science in 1984–88 and 1990–92. He has held a Noah Harding Professorship since 1985. Professor Kennedy is currently Director of the Center for Research on Parallel Computation, an NSF Science and Technology Center. He was named a fellow of the AAAS in 1994, and of the ACM and IEEE in 1995. He was elected to the National Academy of Engineering in 1990 and since 1995 he has been a member of the National Research Council's Commission on Physical Sciences, Mathematics and Applications. Since 1992, he has chaired the High Performance Fortran Forum, which is seeking to define an informal standard for machine-independent parallel programming extensions to Fortran 90.

Dr. Kennedy has published over one hundred technical articles and supervised more than twenty-five Ph.D. dissertations on programming support software for high-performance computer systems. He has supervised the construction of two substantial software systems for programming parallel machines: an automatic vectorizer for Fortran 77 and an integrated scientific programming environment. His current research focuses on extending techniques developed for automatic vectorization to programming tools for parallel computer systems and high-performance microprocessors. Through the Center for Research on Parallel Computation, he is seeking to develop new strategies for supporting architecture-independent parallel programming, especially in science and engineering.

Chau-Wen Tseng

Chau-Wen Tseng received the A.B. degree in computer science from Harvard University in 1986, and the M.S. and Ph.D. degrees in computer science from Rice University in 1992 and 1993. He served as a Research Associate at the Computer Systems Laboratory at Stanford University in 1993–95 and as a consultant at Silicon Graphics before taking a position as an Assistant Professor at the University of Maryland at College Park in 1995. Dr. Tseng's research interests are in the field of parallel and distributed systems, particularly compilers for high-performance architectures. He has designed and implemented compilers for both message-passing multiprocessors such as the Intel Paragon and scalable shared-memory multiprocessors such as the Stanford DASH. He is currently exploring interactions between compilers, run-time systems, and architectures for a new generation of software distributed-shared-memory systems and flexible shared-memory multiprocessors that combine the benefits of coherent shared-memory with message passing and extensible coherence protocols.