Proceedings of Supercomputing'94. ISSN 1063-9535.

The D Editor: A New Interactive Parallel Programming Tool

Seema Hiranandani† Ken Kennedy* Chau-Wen Tseng‡ Scott Warren*
seema@sgi.com ken@cs.rice.edu tseng@cs.stanford.edu scott@cs.rice.edu

†*Silicon Graphics Computer Systems, Mountain View, CA 94039-7311*
**Department of Computer Science, Rice University, Houston, TX 77251-1892*
‡*Computer Systems Laboratory, Stanford University, Stanford, CA 94305-4070*

Abstract

Fortran D and High Performance Fortran are languages designed to support efficient data-parallel programming on a variety of parallel architectures. The goal of the D Editor is to provide a tool that allows scientists to use these languages efficiently. The D Editor combines analyses for shared-memory machines and compiler optimizations for distributed-memory machines. By cooperating with the underlying compiler, it can provide novel information on partitioning, parallelism, and communication based on compile-time analysis at the level of the original Fortran program. The D Editor uses color coding and a collection of graphical displays to help the user to zoom in on portions of the program containing sequentialized code or expensive communication. The prototype implementation is useful for interactively displaying the results of compile-time analysis; however, it has a number of shortcomings that must be addressed. Future enhancements will provide additional advice and transformation capabilities. We believe the D Editor is representative of a new generation of tools that will be needed to assist scientists to fully exploit languages such as High Performance Fortran.

1 Introduction

When MIMD distributed-memory machines such as the Intel Paragon, Thinking Machines CM-5, and IBM SP-1 were first introduced, to achieve good performance scientists were required to write message-passing programs that dealt with separate address spaces, communication, and synchronization. The development process was tedious, error-prone, and the results were not very portable. To solve this problem, researchers, vendors, and users together informally created High Performance Fortran (HPF), a version of Fortran designed to provide a simple yet efficient data-parallel programming model across a wide range of modern architectures [16].

HPF was based on a number of existing languages such as Fortran 90 [1], CM Fortran [25], Fortran D [11], and Vienna Fortran [7]. It simplifies programming by providing a global name space for the user, along with a variety of useful data-parallel array operations. One of the novel features found in HPF are annotations that allow users or automatic tools to specify *data placement*, the partitioning of data onto processors.

Using these annotations, in many cases advanced compilers can automatically generate efficient programs for MIMD distributed-memory machines [17] or even networks of workstations [2]. However, experience with parallelizing compilers for shared-memory machines has shown that no automatic system will work perfectly in all (some say most) cases. When automatic systems fail due to imperfect analysis or incorrect heuristics, compiler feedback [27] or interactive parallelization systems [13, 21] can prove helpful.

1.1 Feedback for High Performance Fortran

We believe that user feedback and interaction will be very important for languages such as HPF, because on massively parallel systems small mistakes may cause large degradations in performance. What is needed is a way to help the user understand how changes to the source code affect performance.

Unfortunately, this task is much harder for HPF than for previous languages. The reason is that HPF compilers transform the program to a great extent, especially on distributed-memory machines where the output programs operate in local name spaces and perform explicit communication through messages. This transformation process makes it quite difficult to determine how the program can be improved and relate it back to the user in the context of the original program. In particular, a small change in the manner data is partitioned in an HPF program will likely result in major changes in both the form of the compiler-generated code and its resulting performance.

For instance, consider the kernels in Figure 1 performing Successive-Over-Relaxation (SOR), a technique for solving partial differential equations. The array `A` is distributed block-wise by columns in both kernels. The computation traverses columns of `A` in the first kernel and rows of `A` in the second. Assume that the computation is assigned to processors according to the *owner computes* rule, where each processor calculates values only for data it owns. Once communication is introduced, the first kernel will be sequentialized (unless additional program transformations such as strip-mining and loop interchange are applied by the compiler). In comparison, the second kernel will execute mostly in parallel, exploiting pipeline parallelism [17]. If the array `A` is distributed block-wise by rows instead, the situation is reversed.

All four versions of SOR are shown in Figure 2. Arrows represent execution order; communication is required at the boundaries. With a naive compiler, the kernels in Figure 2(a) and (c) are sequentialized because each processor must complete most of its local computation before it can communicate data needed by the next processor. The kernels in Figure 2(b) and (d) are able to exploit pipeline parallelism because their execution order allows data needed by other processors to be computed and communicated earlier, rather than at the very end.

It should be evident from this example that determining performance is difficult at the source HPF level for non-experts. Examining the message-passing output does not help much, and even profiling information will only report poor performance, not suggest ways to improve the program. It will thus be much easier for scientists using languages such as HPF if programming tools can

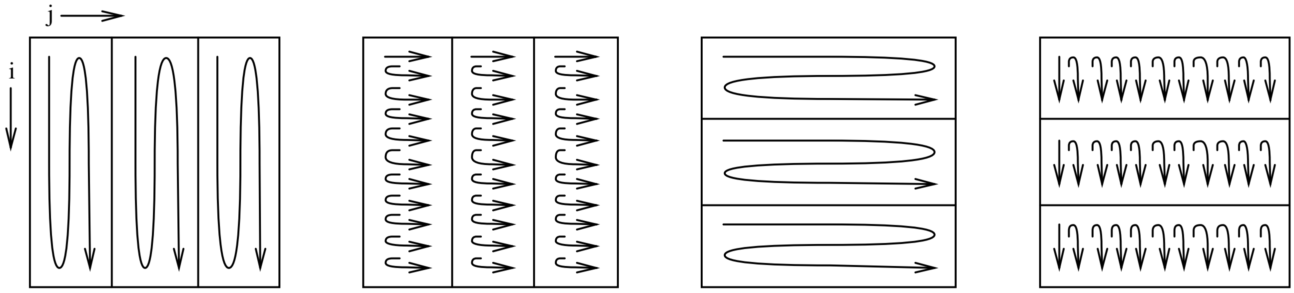
This research was sponsored by ARPA under contract #DABT63-91-K-0005 & DABT63-92-C-0038 and the IBM Corporation. Additional support was provided by the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008. Chau-Wen Tseng was supported in part by an NSF CISE Postdoctoral Fellowship in Experimental Science.

```
{* Sequential SOR *}
REAL A(100,100)
DISTRIBUTE A(:,BLOCK)
do j=2,99
  do i=2,99
    A(i,j) = 0.7*(A(i-1,j)+A(i,j-1)+
        A(i+1,j)+A(i,j+1))+0.3*A(i,j)
  enddo
enddo
          ⇓
{* Compiler Output for 4 Processors *}
REAL A(100,0:26)
myid = myproc()      {* 0...3 *}
if (myid > 0) send A() to P_left
if (myid < 3) recv A() from P_right
if (myid > 0) recv A() from P_left
do j=lb1,ub1
  do i=2,99
    A(i,j) = 0.7*(A(i-1,j)+A(i,j-1)+
        A(i+1,j)+A(i,j+1))+0.3*A(i,j)
  enddo
enddo
if (myid < 3) send A() to P_right
```

```
{* Pipelined SOR *}
REAL A(100,100)
DISTRIBUTE A(:,BLOCK)
do i=2,99
  do j=2,99
    A(i,j) = 0.7*(A(i-1,j)+A(i,j-1)+
        A(i+1,j)+A(i,j+1))+0.3*A(i,j)
  enddo
enddo
          ⇓
{* Compiler Output for 4 Processors *}
REAL A(100,0:26)
myid = myproc()      {* 0...3 *}
if (myid > 0) send A() to P_left
if (myid < 3) recv A() from P_right
do i=2,99
  if (myid > 0) recv A() from P_left
  do j=lb1,ub1
    A(i,j) = 0.7*(A(i-1,j)+A(i,j-1)+
        A(i+1,j)+A(i,j+1))+0.3*A(i,j)
  enddo
  if (myid < 3) send A() to P_right
enddo
```

Figure 1: SOR

DISTRIBUTE A(:,BLOCK)	DISTRIBUTE A(:,BLOCK)	DISTRIBUTE A(BLOCK,:)	DISTRIBUTE A(BLOCK,:)
do j =	do i =	do i =	do j =
do i =	do j =	do j =	do i =
A(i,j) =	A(i,j) =	A(i,j) =	A(i,j) =



(a) (b) (c) (d)

Figure 2: Execution Order of Four Versions of SOR

statically or dynamically distinguish when performance is poor, and offer advice on how the program may be enhanced.

1.2 The D System

To address this lack of understanding, the D System project at Rice University is developing a collection of tools designed to help scientists to use languages like HPF. The D System supports Fortran D, a precursor and contributor to HPF that contains a subset of features found in HPF as well as support for capabilities not currently in HPF. The system contains FIAT, an interprocedural analysis framework [15], the Fortran D compiler [17], and tools for performing automatic data decomposition [3], data-race detection, static performance estimation [4, 20], and performance profiling. All these components will be integrated in the D Editor, the core of the D system.

The goal of the D Editor is to help users develop efficient Fortran D or HPF programs by providing feedback and guidance at the source level, based on analysis and measurement of performance of the compiler output. To achieve this end the D Editor must be able to explain to the user how changes in the high-level program will affect both the output code and its performance, especially changes to the data decomposition.

1.3 Contributions

In previous work, we described the design, implementation, and evaluation of a prototype Fortran D compiler [14, 17, 18, 19]. The original contributions of this paper are to describe:

- A work model for using an interactive programming tool to improve the performance of HPF programs on distributed-memory machines.
- The information an interactive programming tool needs from the underlying compiler.
- Techniques used to process compiler information so it makes sense to the user.
- Methods used in the prototype editor to display information to the user in an accessible and natural manner.

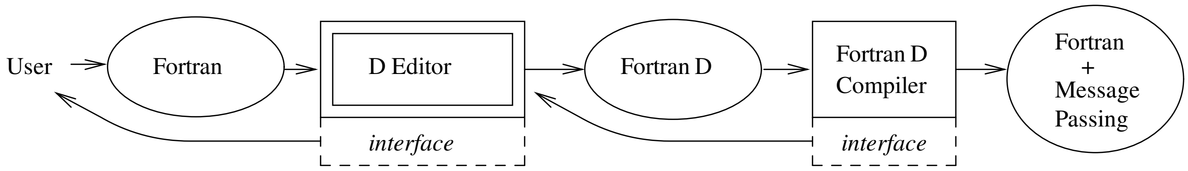


Figure 3: D Editor and Fortran D Compiler Interaction

In this paper we present the prototype D Editor, focusing on the interaction between the editor and the Fortran D compiler. We begin with a review of the Fortran D language and compiler, then describe the work model for using the D Editor. We list the information the compiler must provide to the editor at each level of the program and show how this information is processed and displayed, as well as additional options the editor provides the user. We discuss limitations of the prototype editor and possible solutions and future extensions. We conclude with a discussion of related work.

2 Background

2.1 Fortran D Language

In Fortran D, the DECOMPOSITION statement declares an abstract problem or index domain. The ALIGN statement maps each array element onto the decomposition. The DISTRIBUTE statement groups elements of the decomposition and aligned arrays, mapping them to a parallel machine. Each dimension is distributed in a BLOCK, CYCLIC, or BLOCK_CYCLIC manner; the symbol ":" marks dimensions that are not distributed. Because the alignment and distribution statements are executable, dynamic data decomposition is possible. The complete language is described in detail elsewhere [11].

Apart from minor syntactic differences, HPF provides data decomposition specifications virtually identical to those in Fortran D. Our experiences with the Fortran D compiler and D Editor are thus immediately applicable to HPF. HPF provides additional array intrinsics, procedure interfaces, and external interfaces not found in Fortran D. We continue to call our target language Fortran D to emphasize that it is a research version of HPF designed to support research on interprocedural compilation and irregular computations.

2.2 Fortran D Compiler

To validate the feasibility and usefulness of the Fortran D and HPF programming model, we have designed and implemented a prototype Fortran D compiler for MIMD distributed-memory machines. Preliminary results show that for many programs, it is able to produce output that approaches the quality of hand-optimized code [18]. Given a data decomposition, the prototype compiler can automatically translate sequential programs into efficient parallel programs using the owner computes rule. The two major steps it performs in targeting MIMD distributed-memory machines are partitioning the data and computation across processors, then introducing communication for nonlocal accesses where needed. The compiler applies the following passes in order: analyze program, partition data, partition computation, analyze communication, optimize communication, manage storage, and generate code. Details are presented elsewhere [14, 17, 19, 26].

2.3 The ParaScope Programming Environment

The Fortran D compiler is implemented as a part of ParaScope, a programming environment that pioneered research on interprocedural optimization in an efficient compilation system [8, 9]. Its interprocedural compilation framework has been encapsulated into a system called FIAT (Framework for Interprocedural Analysis and Transformation) [15] and used to solve a number of interprocedural problems in both ParaScope and the Stanford SUIF compiler.

A key component of the ParaScope environment is the ParaScope Editor (PED), an intelligent interactive editor for shared-memory machines [13, 21, 22]. The ParaScope Editor provides the results of sophisticated program analyses and a variety of powerful program transformations that have been shown useful in converting programs to parallel form. The ParaScope Editor also supports general user editing through a hybrid text and structure editing facility that incrementally analyzes the modified program for potential hazards. The D Editor is built on top of the ParaScope Editor and possesses all of its analysis capabilities. In addition, the D Editor incorporates additional features required for supporting languages such as Fortran D and HPF.

3 Work Model

We begin by describing how we believe an advanced programming tool can be used to analyze and tune the performance of a program. This work model was developed as a group effort in discussions with other members of the D System group at Rice University. First, Fortran D or HPF data decomposition specifications are inserted by the user or automatic tools. The resulting program is loaded into a graphical tool; analysis is performed to statically calculate parallelism and communication. A summary of the program consisting of subroutines and loop headers is then presented. Sections of low parallelism or high communication are highlighted.

The user selects one such region, causing the tool to zoom in and provide detailed information. The program text in the region is displayed. References causing nonlocal accesses are colored, with the color indicating its effect on parallelism. Additional information on data dependences, messages and variable data mappings are also provided. If the user is dissatisfied, he or she may attempt to improve the program's performance by making changes to the data decomposition or program structure. The tool can then present the updated result and allow the user to evaluate the effect on either parallelism or communication.

As the description of the work session shows, the user is able to quickly zoom in on portions of the program causing communication and restricting parallelism. The key here is that since the tool presents results using static analysis at the source level, feedback should be both quick and easy to understand.

4 Fortran D Compiler Interface

Now that we have sketched the features we hope to provide in the D Editor, we can show how the prototype is constructed. To ease the burden on programmers, the D Editor needs to provide feedback and guidance at the Fortran D or HPF language level. However, since Fortran D programs undergo major transformations during compilation, it would be very difficult for the editor to simply examine the source and output program. Instead, we

Program	*Information Type*	*Source*
Reference Level	Location of data decomposition specifications affecting reference	compute
	Messages caused by reference	record
	Alignment, distribution, and decomposition of array	available
	Overlaps for array	available
Statement Level	Statement type (parallel, replicated, single-processor, reduction)	compute
	Cross-processor dependences caused by statement	compute
	Nonlocal references in statement	record
	Messages caused by statement	record
	Does statement assign to a private variable?	available
	List of uses of assignment to private variable	available
Loop Level	Loop type (replicated, parallel, pipelined, sequential, single-processor)	compute
	Cross-processor dependences carried by loop	record
	Nonlocal references in loop	record
	Statements with nonlocal references	record
	Lists of messages caused by loop (independent, carried-all, carried-part)	record
	All loop statements executed identically?	available
	Number of iterations executed by each processor	available
Procedure Level	Messages caused by procedure	record
	Nonlocal references	record
	Statements with nonlocal references	record
	Lists of loops by parallelism type (replicated, parallel, pipelined, sequential)	record
	Lists of loops by message type (shift, broadcast, gather, reduction, run-time)	record
Program Level	Messages in program	record
	Nonlocal references	record
	Statements with nonlocal references	record
	All loops and procedures in program	record
Message Level	Message type (independent, carried-all, carried-part, reduction)	record
	Communication type (shift, broadcast, gather, reduction, inspector, run-time)	record
	Reduction type (sum, prod, min, max, minloc, maxloc, logical AND, OR)	record
	List of processors sending/receiving data	record
	List of array sections sent/received	record
	List of array sections communicated in message	available
	Location of message in program	available
	Loop level of message	available

Table 1: Fortran D Compiler Interface to the D Editor

have designed the D Editor so that it can request information directly from the Fortran D compiler through an interface. This section describes the design and implementation of the interface between the compiler and the D Editor; the manner in which the editor displays the information is presented later in Section 5.

Figure 3 illustrates how the D Editor and Fortran D compiler are designed to interact. Users employ the D Editor to examine Fortran D programs. The Fortran D compiler analyzes and optimizes these input programs, providing feedback to the D Editor through the compiler interface. The D Editor processes this information and presents it to the user in an easily accessible manner through its graphical interface. Future versions of the editor will provide editing and transformation capabilities, but these features are not available in current prototype. When the user is satisfied, the Fortran D compiler produces a message-passing program that can be compiled using native Fortran compilers and linked with the underlying run-time system to create executables.

4.1 Information Presented

The Fortran D compiler interface provides information about the program at several levels: individual variable references, statements, loops, procedures, and the entire program. Information on communication is encapsulated in message objects that can be queried for additional information.

Table 1 lists the information provided by the Fortran D compiler interface at each level of the program. In the column labeled *Source*, we describe the method used to collect the information gathered by the interface. Items labeled *compute* require the most effort. This group represents new information that must be calculated by additional analysis in the Fortran D compiler; examples include classification of statements and loops by degree of parallelism. New algorithms were designed to compute the appropriate information and are described later.

A second group of data labeled *record* represents information produced but not normally recorded by the compiler. Examples include the list of messages caused by a particular statement or reference. The compiler does not maintain such information, since it only determines where messages need to be inserted, not exactly where they are caused. The D Editor, however, needs the information to provide more feedback to the user. To support the D Editor, the compiler has been modified to either record information in this category when it is produced or dynamically collect and cache it the first time it is requested. The final group of data labeled *available* is produced and retained by the compiler as part of its standard analysis. This category includes the data decomposition for each array and nonlocal references.

4.2 Data Decomposition

The main factors affecting performance on MIMD distributed-memory machines are the data decomposition, parallelism, and communication. To be useful, the D Editor must relate parallelism and communication back to the structure of the data decomposition and computation for the original program, using information from the compiler. Here we describe data collected for data decompositions; later sections describe the interface for the other two classes.

Recall that the Fortran D compiler partitions computation among the processors using the *owner computes* rule. The way computation is partitioned in turn determines both parallelism and communication. As a result, the manner in which data is partitioned in a Fortran D or HPF program crucially affects performance. It is important that the compiler interface provides information on how data is partitioned at each point in the program, as well as which HPF statements are responsible.

This process is complicated by the fact that Fortran D supports dynamic data decomposition, allowing data decompositions to be changed at different points in the program. Furthermore, in Fortran D, data decompositions are propagated between procedures. For each procedure invocation, the formal parameters of the called procedure inherit the decompositions of the corresponding actual parameters passed at the call [14]. Global variables retain their decomposition from the caller. Since a program may span multiple modules (files), the actual Fortran D statements that describe the data decomposition for the reference may reside in another module.

Interprocedural Reaching Decompositions. *Reaching decompositions* refers to the problem of determining the data decomposition of a variable at a given point in the program [14]; the Fortran D compiler solves this problem using FIAT [15]. Since FIAT's interprocedural analysis strategy is based upon a demand-driven paradigm, the compiler interface is able to obtain the solution to the reaching decomposition problem at any node in the call graph by demanding the annotation that corresponds to our problem. If the solution was computed earlier, then the system returns the annotation without recomputing it. Details on how the interprocedural reaching decomposition solution is computed are discussed elsewhere [14].

After the interprocedural reaching decomposition annotation is computed by FIAT, the compiler performs local reaching decomposition to determine if an array's decomposition has been re-defined within the procedure. The resulting details on the decomposition, alignment, and distribution that reach every reference in the procedure are then stored.

Location of Fortran D Statements. A desirable feature for the D Editor is to point the user to the actual Fortran D or HPF data decomposition statements that affect a given variable. To provide this information, the compiler interface computes a mapping from each array reference to the Fortran D statements that specify the decomposition of the array reference. The compiler computes during the local phase a unique ID for each data decomposition statement and the context of the module containing the statement. There is a bi-directional one-to-one mapping from the ID to the statement that it refers to. This information is propagated interprocedurally when reaching decompositions are computed and stored for each reference in the procedure. Given the context and the ID, the mapping to the actual Fortran D statements may be found. Additional information may be readily calculated, including the size, name, and declaration of the decomposition, inter and intra-dimensional alignment, and the distribution chosen.

4.3 Parallelism

The second important class of information provided by the D Editor is parallelism. The Fortran D compiler interface calculates parallelism information at both the statement and loop level; the loop level information is more direct and significant. The compiler interface also gathers information about data dependences that may impede parallelism.

Because Fortran D and HPF compilers generate SPMD programs with explicit messages serving as synchronization, techniques for determining the amount of parallelism in a program are somewhat different from standard methods for shared-memory programs. For shared-memory programs, the primary criteria for exploiting parallelism was partitioning the computation so that each processor performs a portion of the work. However, this condition no longer implies parallel execution for SPMD programs, because messages may act as synchronization to sequentialize all or part of the computation.

```
    REAL A(256,256,256)
    DISTRIBUTE A(BLOCK,:,:)
    do 10 k=2,N
     do 10 j=1,N
      do 10 i=1,N
       A(i,j,k)  = F(A(i,j,k),A(i,j,k-1))
 10 continue
```

Figure 4: Internalized Dependences

We showed an example of this phenomenon for the SOR example in Figure 1. The computation is always partitioned and executed on multiple processors due to the owner computes rule; however, without advanced compiler transformations processors are able to work concurrently in only half the cases. In this section we describe how the compiler interface determines whether parallelism is exploited by considering the effects of synchronization.

Cross-Processor Dependences. Shared-memory parallelizing compilers detect parallelism by calculating the data dependences present in a program. A true (flow) data dependence represents a definition (write) that must take place before a corresponding use (read). It is the fundamental constraint on execution order. In addition, a true dependence is said to be *carried* by a loop if its source (write reference) and sink (read reference) take place on different iterations of the loop. A loop can only be executed in parallel if it does not carry any dependences.

Data dependence information is provided by traditional programming tools such as the ParaScope Editor [21]. Because the D Editor is built on the ParaScope Editor, it can calculate and display data dependences in the same manner. Unfortunately, simply reporting occurrences of parallel loops is insufficient for determining the amount of parallelism exploited for languages such as Fortran D and HPF. For a given partition, parallelism is only affected by synchronization between nodes; data dependences local to a node no longer affect large-scale parallelism.

For instance, consider the example shown in Figure 4. The only data dependence in the loop nest is a true dependence between iterations of the `k` loop. Standard analysis would thus label the `k` loop as sequential and the `j` and `i` loops as parallel. However, the D Editor can and needs to analyze additional information. From Fortran D statements we find array `A` is distributed block-wise in the first dimension. Since the data dependence occurs only between elements of the third dimension, the endpoints of the dependence are on the same processor. The dependence is thus *internalized* and does not affect node-level parallelism [3]. The entire loop nest is thus executed in parallel without synchronization.

In order to avoid reporting these internalized data dependences, the Fortran D compiler applies algorithms to distinguish dependences causing synchronization between nodes as *cross-processor dependences* [17]. They are calculated by examining the pattern of data accesses of both endpoints of the dependence to determine whether they occur on separate processors. Non-time-step loops that can carry cross-processor dependences are labeled *cross-processor loops*; they help guide program transformations to exploit pipeline parallelism. Cross-processor dependences can also be used to point out specific references that cause synchronization.

Parallel and Pipelined Loops. Using information from cross-processor dependences and loops, the Fortran D compiler can classify computations as parallel, pipelined, or sequential [17]. Using our definitions, parallel loop nests contain no cross-processor loops, sequential computations have an outermost cross-processor loop, and pipelined computations have a cross-processor loop in an inner position. For the D Editor we classify any loop enclosed in a cross-processor loop to be sequential.

Note that unlike shared-memory compilers, we no longer consider the amount of parallelism of each loop separately. Instead, we calculate the parallelism for each loop nest together as a single unit. For instance, if all the computation enclosed in a loop is executed in parallel, we do not mark it as a sequential loop even if the iterations of the loop are executed sequentially. Conversely, we may mark a loop with no data dependences as sequential or pipelined depending on outer loops, even if the loop could potentially be executed in parallel. Our classification system is motivated by the desire to only bring loops that actually sequentialize computation to the user's attention.

In addition to classifying loops based on their contributions to parallelism, we also label a loop as partitioned if its iterations are spread out and executed on multiple processors.

Parallel Statements. Users may also desire to know how individual statements will be executed in the resulting program. The compiler interface classifies statements as parallel, pipelined, or sequential by finding the deepest loop enclosing each statement. The statement is then assigned the same classification as the loop. In addition, the compiler provides information on how statements have been partitioned; whether its instances are executed by one processor, partitioned across all processors, or replicated on all processors. By classifying loops and statements as parallel, pipelined or sequential, the user is able to quickly find loops and statements that constitute bottlenecks in the program.

4.4 Communication

The third and final class of information concerns the communication a Fortran D or HPF program will generate. Because communication can be expensive, it has an important effect on program performance. Unlike information on data decompositions or parallelism, the compiler interface does not need to perform extra record-keeping or analysis to provide information on messages. Message-passing libraries require precise information on data to be communicated, so the compiler already collects information needed by the D Editor. The interface simply needs to gather the information used to generate each message.

The interface constructs message information during the communication analysis and optimization phases of the Fortran D compiler. All array references are examined to determine if they cause any communication. When communication is found, detailed information about the message is constructed and attached at both the individual reference and at the loop where it is inserted. The D Editor will later extract this information and translate it into text containing the details on the messages.

For each message, the compiler interface provides information on its type, size, location, and destination. With respect to a loop, a message is classified as *independent* if it is communicated once before the loop, *carried-all* if it is executed by all processors at the beginning of each iteration of the loop, and *carried-part* if it is executed before and after the loop to synchronize pipelined computations [17]. Figure 5 shows examples of the three message types and references that cause each message. Note that the message type is selected based on the fundamental ordering constraint imposed by data dependence of the program, and is not an artifact of the Fortran D compiler.

```
      REAL A(256), B(256), C(256)
      DECOMPOSITION D(256)
      ALIGN A, B, C WITH D
      DISTRIBUTE D(BLOCK)
      do 10 k = 1,M
        do 20 i = 1,N
          B(i) = A(i-1)+B(i+1)
          C(i) = C(i-1)
20      continue
10    continue
                ⇓            {* After Compilation *}
      if (...) send & recv A      {* Independent *}
      do 10 k = 1,M
        if (...) send & recv B    {* Carried-all *}
        if (...) recv C           {* Carried-part *}
        do 20 i = 1,N
          B(i) = A(i-1)+B(i+1)
          C(i) = C(i-1)
20      continue
        if (...) send C           {* Carried-part *}
10    continue
```

Figure 5: Message Types

The communication type of a message may be either a simple point-to-point send/receive or one of a number of collective communication patterns such as broadcast, gather, or reduction. If the communication is for a reduction, the type of the reduction is determined. In addition to the message and communication type, the compiler interface can also easily determine the position of the message in the code, the size and location data being sent, and the processors participating in the communication. For architectures that require message data to be contiguous, the amount of buffering required can also be calculated.

5 D Editor Interface

Now that the sources of compiler information are clear, we describe how the D Editor displays this data in a natural, easily understood manner. There are several reasons why a Fortran D program may not perform well on a parallel machine, but it is usually the case that there is either insufficient parallelism inherent in the program or communication costs are very high. The optimizations performed by the Fortran D compiler are aimed largely at exploiting parallelism and reducing communication overhead.

The D Editor display is designed to help the user understand the implications of the compiler optimizations on the behavior of the program at the level of the original Fortran D program. We first examine features of the graphical display, then show how it works on an example program. The D Editor display is derived from the display of the ParaScope Editor [13, 21, 22], but contains significantly more information. Figure 6 provides an example of the user interface of the D Editor for different versions of SOR. The display is separated into five panes.

To assist users in understanding the performance implications of their programs, we have provided a coloring scheme that characterizes references, statements, dependences and loops based on their impact on communication and parallelism. The coloring scheme is analogous to the colors of the traffic light. Code highlighted in red indicate computation that has been sequentialized. Yellow implies partial parallelism is exploited using pipelining. Green implies the computation is performed completely in parallel. Communication is colored in the same manner to indicate its effect on parallelism. The color generally corresponds to the expense of communication as well. Code that is not colored (black) does not affect either communication or parallelism.

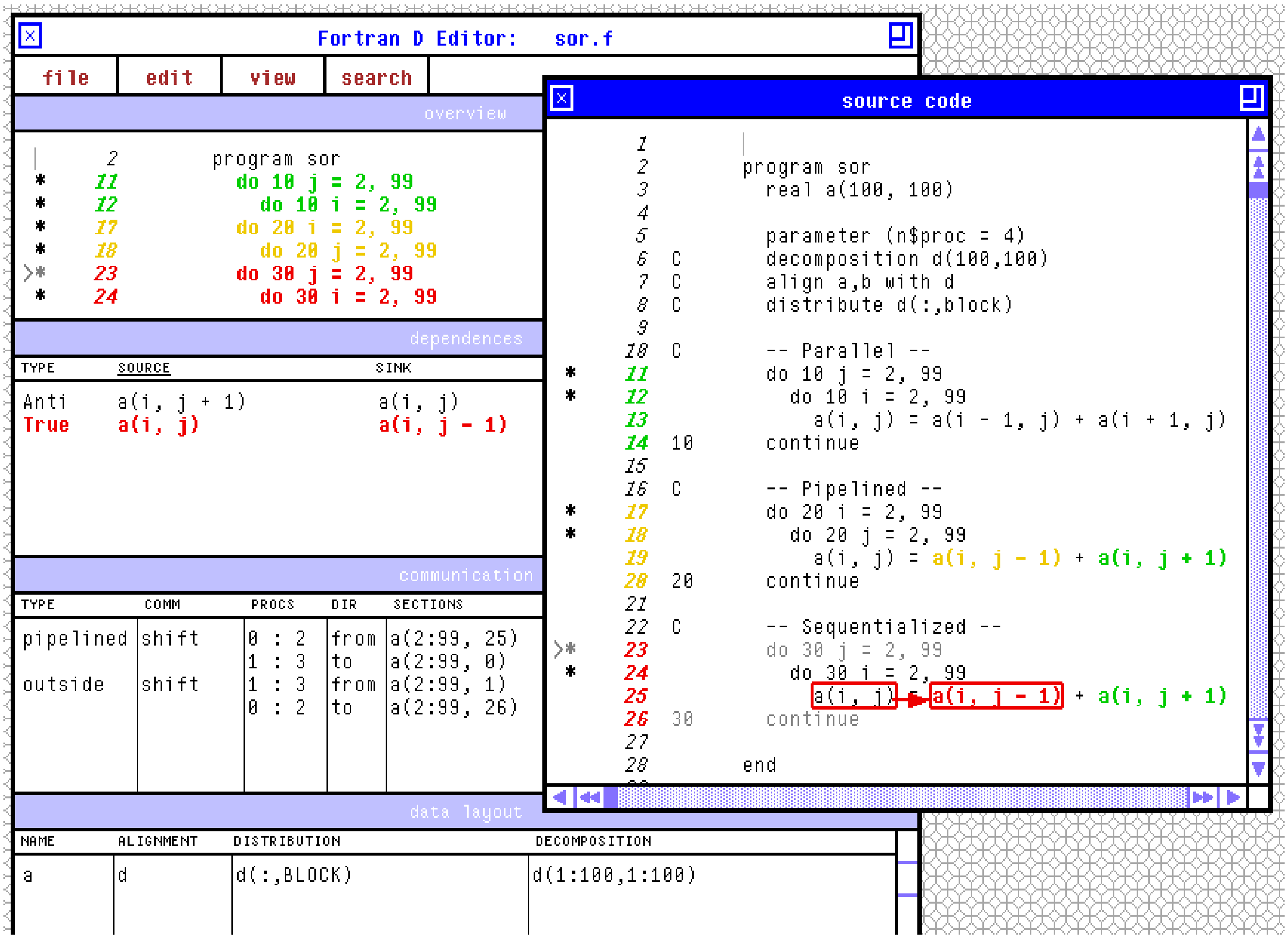


Figure 6: D Editor with SOR in Session

5.1 Overview Pane

The first component of the D Editor display is the *overview pane*; it provides a summary of the procedures and loops in the program. Procedures and loop headers are displayed and colored based on their parallelism. The overview pane is useful for quickly finding sequentialized computation or loops causing communication. The user can inspect the loop by selecting it in the overview pane. The D Editor marks the currently selected loop with the > symbol in the overview pane; it determines which dependences, messages, and arrays are displayed in the other panes.

In Figure 6, the overview pane in the top left corner summarizes the three loop nests present in the example SOR program. The color of the three loop nests (green, yellow, red) show that they are parallel, pipelined, and sequentialized, respectively. The > symbol is next to the loop at line 23, indicating the third loop nest has been selected by the user.

5.2 Dependence Pane

Positioned immediately below the overview pane, the *dependence pane* displays the data dependences carried on the selected loop. These dependences are classified as true (flow), anti, or output dependences. Cross-processor dependences point out values that are defined and used on different processors; they are marked red, yellow, or green depending on their effect on parallelism. Dependences selected in the dependence pane are displayed as arrows in the source pane. The D Editor allows the user to selectively display arrows for individual dependences, all cross-processor dependences, or all loop-carried dependences.

Because the Fortran D compiler uses the owner computes rule, only a single processor writes to each array element. There are thus no colored (*i.e.,* cross-processor) output dependences. There are also no red-colored anti-dependences; they never impede parallelism since message vectorization will automatically move communication caused by cross-processor anti-dependences out of loop nests. As a result, all cross-processor dependences colored red in the D Editor are true dependences.

In the SOR example, dependences for the selected loop `j` (the first loop in third loop nest, at line 23) are displayed in the dependence pane. There are two data dependences carried on loop `j`. The first is an anti-dependence between `a(i,j)` and `a(i+1,j)`. Since the communication caused by this anti-dependence is extracted out of the loop nest completely by message vectorization, it has no effect on parallelism and is not colored (black). In comparison, the true data dependence between `a(i,j)` and `a(i-1,j)` results in communication that sequentializes the entire loop nest. It is therefore colored red, alerting users to its significance.

5.3 Communication Pane

Next we have the *communication pane*, which displays all communication associated with the selected loop. For each message caused by the loop, the D Editor displays its location, type, and other information in two lines. Messages of type *independent* are labeled *outside* because they take place once for the entire loop nest and are placed outside the loop. Messages of type *carried-all* are labeled *inside* because they take place once per iteration and are placed inside the loop. Recall that both message types allow execution to occur fully in parallel. In comparison, messages of type *carried-part* are labeled *pipelined* because they cause either pipelined or sequentialized execution, depending on whether the loop carrying the message is outermost.

The communication type (*e.g.,* shift, broadcast, reduction) of each message is displayed after the message type, followed by the sets of sending/receiving processors and section of data communicated. If the communication is a global reduction, the type of the reduction (*e.g.,* plus, times, min, max) is shown. By examining the communication pane, the user can thus quickly determine the number, type, and sizes of the messages executed at the level of a particular loop.

In Figure 6, the communication pane shows two messages occurring at the level of the `j` loop at line 23. The first message corresponds to the communication caused by the reference `a(i-1,j)`. The message is labeled *pipelined* because it causes processors to execute in order, normally inducing pipelined execution. However, since the messages are inserted at the level of the outermost loop, in this case the computation is actually sequentialized. In comparison, the second message corresponding to the reference `a(i+1,j)` is inserted outside the entire loop nest and does not impede parallel execution.

5.4 Data Layout Pane

At the bottom of the D Editor display is the *data layout pane*. Here the editor displays the data decomposition information for each array in the loop. For each variable, the D Editor lists the decomposition to which it is mapped. The alignment between the array and decomposition is shown next; the lack of alignment annotations means the array is aligned perfectly with the decomposition. The distribution of the decomposition is next, followed by the size and dimensions of the decomposition. Arrays that are not partitioned are listed without alignment and distribution specifications. Unfortunately, the prototype D Editor does not provide users with the ability to navigate between the variable and its corresponding Fortran D data decomposition statement. In the SOR example, the array `a` is shown to be identically aligned with the decomposition `d`, which is distributed block-wise in the third dimension.

5.5 Source Code Pane

Finally, a separate *source code pane* displays the actual program code. As in the overview pane, loops are colored based on their parallelism. However, only their line numbers are colored to allow greater emphasis on actual references in the program. The header of the selected loop is marked with the > symbol next to its line number. As mentioned earlier, data dependences selected in the dependence pane are displayed in the source pane with an arrow pointing from the source to the sink of the dependence.

In the source pane, every variable reference that causes messages to be inserted in the output program is colored red, yellow, or green depending on its effect on parallelism. As a result, the user can quickly determine the causes of nonlocal accesses in the program. A loop nest may contain a number of nonlocal references, only some of which cause the computation to be sequentialized. The hierarchical design of the D Editor, combined with its use of color coding, makes it easy for users to zoom in on the problematic portions of their code.

In the current implementation of the D Editor, the source pane scrolls to the loop selected by the user in the overview pane. Other navigational aids include the ability to automatically scroll to the source or sink of a dependence by selecting it in the dependence pane. We are working on adding editing capabilities that will allow the user to modify a section of his or her program and interactively analyze the effects of the changes on performance.

In Figure 6, the source pane displays the entire SOR example program. The line numbers 11–14 are colored green, indicating the loop nest can be executed in parallel. No reference enclosed by the loop is colored, so no communication is required for this loop nest. The line numbers 17–20 are colored yellow, signifying the loop nest is executed in a pipelined fashion. The reference `a(i,j-1)` at line 19 is colored yellow, showing that it is responsible for the cross-processor data dependence that causes pipelining. The reference `a(i,j+1)` at line 19 is colored green; indicating it causes communication but does not affect parallelism.

The third loop nest is the most interesting. The lines 23–26 are colored red, designating the computation in the loop nest as sequentialized. The reference `a(i,j-1)` at line 25 is colored red, betraying it as the impediment to parallel execution. Because the `j` loop at line 23 is also the selected loop (marked by the > symbol), cross-processor data dependences on the loop are displayed using colored arrows. The red arrow from `a(i,j)` to `a(i,j-1)` at line 25 shows the endpoints of the dependence. Since the arrow represents a true data dependence, values produced by the assignment to `a(i,j)` are used on later iterations of the `j` loop by the reference `a(i,j-1)`. Additionally, because the arrow is colored, the user knows that those values are consumed by processors other than the producing processor, with the color red signifying the communication sequentializes the computation in the loop nest. The reference `a(i,j+1)` at line 25 is colored green; as before, it causes communication but does not affect parallelism.

6 Prototype D Editor Evaluation

So far, we have described the status of the prototype D Editor demonstrated in November 1993 at Supercomputing'93. As we have seen, this implementation can be a valuable tool for interactively displaying the results of compile-time analysis. However, it also has a number of weaknesses and limitations. In this section we discuss some limitations of the current prototype and possible solutions, as well as some new features planned for the D Editor.

6.1 Limitations

Compiler Transformations. The prototype D Editor assumes that the underlying compiler does not perform program transformations such as loop interchange and strip-mining [21]. As a result, the information it presents may be inaccurate for an automatically transformed program. For instance, for kernels such as SOR the Fortran D compiler can apply loop interchange to convert sequential computations into pipelined computations; these loops should be colored yellow rather than red. The D Editor also does not provide information on the granularity of pipelining selected by the underlying compiler if strip-mining is enabled.

A possible solution to this problem is to extend the editor interface with a control panel for all the optimizations performed by the underlying compiler. Then the compiler interface can return the correct parallelism and granularity information depending on whether loop interchange and strip-mining are enabled in the control panel. As an added advantage, expert users can use the control panel to enable or disable various compiler optimizations and select the granularity for pipelining [17].

Unfortunately, providing a control panel for optimizations only solves part of the problem. Mapping performance and debugging information from the transformed program back to the original source is an open research problem; compiler assistance is likely to be essential.

Communication Information. Another limitation is that detailed communication information is supplied only for the currently selected loop. As a result, it is not possible to examine messages generated for all loops in the program at once, or for messages caused by nonlocal references not in loops. The overview pane also does not provide an indication of where communication occurs in the program. Since this information is available from the compiler interface, we can solve the problem by extending the current editor interface. One possibility is inserting icons into the overview and source panes to indicate the presence of communication; selecting an icon will cause messages represented by the icon to be displayed in detail in the communication pane.

Parallelism Classification. Some users were perplexed by the parallelism classification system used by the D Editor because they expected parallel loops to be executed in parallel and sequential loops to be executed on a single processor. Instead, the prototype editor classified loops based on whether the enclosed computation is executed concurrently. One possible solution is to only classify loops whose iterations are partitioned across multiple processors, marking them as parallel, pipelined, or sequential. All other loops are simply classified as replicated, indicating that they are executed on all processors. More experience is needed to determine what classification system is more understandable for users.

Multiple Subroutines. The current prototype is able to display only one subroutine at a time. As a result, it cannot relate information between procedures, particularly reaching decompositions. Adding navigation support between variable references and their Fortran D data decomposition statements would ameliorate this problem.

Editing. Editing is not permitted in the current D Editor, but can easily be enabled. After program modifications users will be required to save edits and hit the *reanalyze* button to update parallelism and communication information.

Underlying Compiler. Finally, we found that the prototype editor is limited by the robustness of the underlying Fortran D compiler. As the compiler is also a research prototype, there are many programs it is unable to compile, reducing the applicability of the D Editor. This shortcoming will ease as the compiler improves.

6.2 Future Enhancements

Researchers in the D System group at Rice University are working on enhancements to the D Editor to provide additional advice and transformation capabilities. These new features either display new information or suggest actions to improve program performance.

Interactive Program Transformations. The ParaScope Editor provides a number of automatic structured program transformations targeting shared-memory machines, as well as advice as to the legality and profitability of each transformation [21]. We believe the same transformations can be useful in the D Editor. Legality may be determined in the same manner, but profitability may be quite different. The D Editor can use the same algorithms as the ParaScope Editor to incrementally update dependence information after transformations [22], but new techniques for updating parallelism, partitioning, and communication information are needed. Researchers are also studying the problem of incremental interprocedural analysis. New transformations to Fortran D or HPF data decomposition statements can also benefit the user.

Automatic Data Decomposition. The D Editor currently provides information for the data decomposition specified in the program. Instead of evaluating each data decomposition and presenting its effect on parallelism and communication, the D Editor can incorporate heuristics for automatically selecting data decompositions [3]. The resulting choices can then be presented to the user as a list of possible selections, or be used to suggest data decompositions that provide better overall performance.

Performance Statistics. Another useful feature for the D Editor is providing performance data. Ranking procedures and loops by expected or measured execution time would allow users to more easily focus on critical portions of the program. Execution times would also make it easier to evaluate the effectiveness of automatic data decomposition or program transformations. Performance statistics may either be collected at compile time through static performance estimation [4, 20] or at run time via profiling [10].

Graphic User Interfaces. Additional facilities for presenting analyses and performance data graphically can improve the usefulness of the D Editor. Information such as the program call graph, data decompositions, load balance, communication pattern, and recurrences (cycles in the data dependence graph) can all be displayed through a graphic interface for easier user comprehension. Integration with systems such as PABLO from Reed *et al.* at the University of Illinois would greatly simplify the construction of a graphic user interface.

7 Related Work

There is a large amount of work in both compiling for distributed-memory architectures and interactive programming environments. The D Editor is one of the first to combine both features in a single parallel programming tool. The Fortran D compiler is similar to CALLAHAN-KENNEDY [6], SUPERB [12], and KALI [23] in that the compilation process is based on the decomposition of data in the program. In comparison with these and other systems, the Fortran D compiler performs significantly more compile-time analysis and optimization and relies less on program transformations, language extensions or an extensive run-time system.

The D Editor is based on the ParaScope Editor [13, 21, 22]. They share many features, including the underlying hybrid text/structure editor, graphics utilities, and dependence display. The ParaScope Editor is distinguished by its incremental analysis and ability to provide guidance on a large number of interactive program transformations. SIGMACS is another interactive system targeting shared-memory systems; it can display call graphs, process graphs, and a statement dependence graph [24]. TINY is a system that provides precise data dependence analysis and program transformations for a core subset of Fortran [28].

FORGE 90 is an interactive parallelization system that performs data-flow and dependence analyses and also supports loop-level transformations [2]. Associated tools graphically display call graph, control flow, dependence, and profiling information. FORGE 90 can be used to generate parallel programs for both shared and distributed-memory machines. The VIENNA FORTRAN compilation system, a successor to SUPERB, can display the program after different stages of compilation to distributed-memory machines [7]. It incorporates a performance estimation system that relies on profiling information [10].

The FORTRAN 90D compiler is a closely related project that shares many of the design and implementation strategies of the Fortran D compiler [5]. It takes Fortran 90 as input, provides robust run-time support, but does not attempt automatic parallelization. The FORTRAN 90D compiler can also graphically display data decompositions in a program.

8 Conclusions

Languages such as Fortran D and High Performance Fortran can potentially make it much easier for scientists to program MIMD distributed-memory machines. However, evaluating the performance of a Fortran D program can be difficult. In this paper we introduce the D Editor, an intelligent interactive editor that statically evaluates the parallelism and data movement of parallel Fortran programs. It forms the centerpiece of the D System, an integrated collection of tools to help scientists use these new languages.

This paper describes the variety of information that must be collected and displayed by the D Editor. By cooperating with the underlying compiler, it can provide novel information on data partitioning, parallelism, and communication. Because information is collected through static program analysis and presented at the level of the original Fortran D program, the D Editor is both efficient and easy to use. Though the prototype editor contains a number of limitations that need to be corrected, it is valuable for experimenting with techniques to provide advanced user feedback in an interactive programming tool. We believe the D Editor is representative of a new generation of tools that will be needed to assist scientists programming scalable parallel machines.

9 Acknowledgements

We are grateful to the ParaScope and Fortran D research groups for providing the underlying software infrastructure for the D Editor. In particular, we wish to thank Alan Carle, Chuck Koelbel, and John Mellor-Crummey for participating in the early group discussions on the design of the D Editor and for their feedback on this paper; Vikram Adve and Elana Granston also contributed to later discussions on the D System. We acknowledge Debbie Campbell for her helpful comments.

References

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

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

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

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

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

[8] 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, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[10] T. Fahringer, R. Blasko, and H. Zima. Automatic performance prediction to support parallelization of Fortran programs for massively parallel systems. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[11] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

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

[13] M. W. Hall, T. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth. Experiences using the ParaScope Editor: an interactive parallel programming tool. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[14] 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.

[15] 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.

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

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

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

[19] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April 1994.

[20] K. Kennedy, N. McIntosh, and K. S. McKinley. Static performance estimation in a parallelizing compiler. Technical Report TR91-174, Dept. of Computer Science, Rice University, December 1991.

[21] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[22] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice & Experience*, 5(7):575–602, October 1993.

[23] 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.

[24] B. Shei and D. Gannon. SIGMACS: A programmable programming environment. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

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

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

[27] M. J. Wolfe. Semi-automatic domain decomposition. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[28] M. J. Wolfe. The Tiny loop restructuring research tool. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.