Chapter 1
Design and Implementation of the D Editor *

(Extended Abstract)

Seema Hiranandani† Ken Kennedy† Chau-Wen Tseng‡ Scott Warren†

Abstract

This paper describes the design and implementation of the D editor, an intelligent interactive editor for data-parallel Fortran programs. It forms the centerpiece of the D system, a collection of integrated tools for programming languages such as Fortran D and High Performance Fortran. By cooperating with the underlying compiler, the D editor provides novel information on partitioning, parallelism, and communication based on compile-time analysis at the level of the original Fortran program. It uses color coding and a collection of graphical displays to help users zoom in on portions of the program containing sequentialized code or expensive communication. Although the current implementation of the D editor is only a browser, its powerful program analysis and presentation features make it a valuable tool for helping scientists to fully exploit advanced programming languages such as High Performance Fortran.

1 Introduction

When MIMD distributed-memory machines such as the Intel Paragon and Thinking Machines CM-5 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 [10].

HPF is based on a number of existing languages such as Fortran 90, CM Fortran, Fortran D [11] and Vienna Fortran [5]. 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 and other parallel architectures [4, 5, 7, 11, 12, 16].

Unfortunately, it is not clear how users should modify their programs when automatic parallelization fails. On shared-memory machines, interactive programming tools such as

*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, as well as an NSF CISE Postdoctoral Fellowship in Experimental Science.

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

‡Center For Integrated Systems, Stanford University, Stanford, CA 94305

```
{* Seqential 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(100,25)
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(100,25)
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
```

FIG. 1. *SOR*

the ParaScope Editor [8, 14, 15], Sigmacs [17], and Tiny [18] can provide sophisticated analyses and transformations to assist users. Forge90 [1] and the Vienna Fortran Compilation System [5] can also provide some interactive display capabilities for distributed-memory machines. However, these tools do not handle sophisticated compiler optimizations such as those for exploiting pipeline parallelism.

1.1 Feedback for HPF

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 decomposed 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. When the kernels are partitioned according to the *owner computes* rule and 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 [11]. If the array `A` is distributed block-wise by rows instead, the situation is reversed.

It should be evident from this example that determining the performance of a program 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 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, we are developing the D System, a collection of tools designed to help scientists to use languages like HPF. It is designed to support 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 D System contains an interprocedural analysis framework [9], the Fortran D compiler [11], and tools for performing automatic data decomposition [2], data-race detection, static performance estimation [3, 13], 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/HPF programs by providing feedback and guidance at the source level, based on compilte-time analysis and measurement of the performance of the compiler output. To achieve this end the D editor must be able to explain to the user how changes in the original Fortran D program will affect both the output code and its performance, especially changes to the data decomposition. Although the current implementation of the D editor is only a browser, its powerful program analysis and presentation features have made it a valuable tool for interactively displaying the results of compile-time analysis. Future enhancements will provide additional helpful advice and transformation capabilities.

In previous work we described the design, implementation, and evaluation of a prototype Fortran D compiler [11, 12]. Here we presents the design and implementation of the prototype D editor demonstrated at Supercomputing '93. Some of our original contributions are to discover:

- The information an interactive programming tool needs from the underlying compiler.
- Techniques for processing compiler information so they make sense to the user.
- Methods of displaying information to the user in an accessible and natural manner.
- A useful work model for using an interactive programming tool to improve the performance of HPF programs on distributed-memory machines.

2 Fortran D Compiler Interface

To ease the burden on programmers, the D editor provides feedback and guidance at the Fortran D/HPF language level. However, since Fortran D programs undergo many transformations during compilation, it would be very difficult for the D editor to simply examine the source and output program. Instead, the D editor requests information directly from the Fortran D compiler through an interface.

Figure 2 illustrates how the D editor and Fortran D compiler interact. Users employ the D editor to create and modify 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. When the user is satisfied, the Fortran D

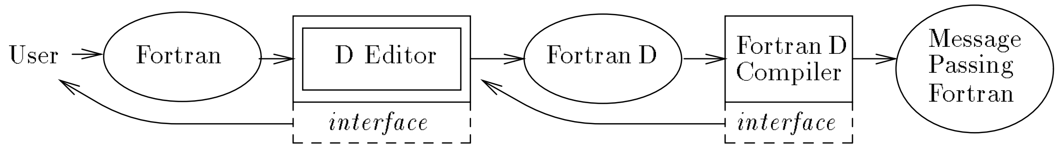

FIG. 2. *D Editor and Fortran D Compiler Interaction*

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.

2.1 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.

2.2 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. The manner in which data is partitioned in a Fortran D/HPF program crucially affects performance. It is thus important that the compiler interface provides information on how data is partitioned at each point in the program.

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, data decompositions are propagated between procedures. The Fortran D compiler applies interprocedural analysis to determine the data decompositions reaching each reference to a distributed array. For the D editor we have also extended the analysis to provide users links back to the actual Fortran D/HPF data decomposition statements that affect a given variable.

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, runtime)	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, runtime)	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

2.3 Parallelism

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

Because messages may serve 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 is 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.

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, but processors are able to work concurrently in only half the cases. To determine whether parallelism is exploited, the D editor must also consider the effects of synchronization caused by messages.

2.3.1 Cross-Processor Dependences Shared-memory parallelizing compilers detect parallelism by calculating data dependences present in a program. 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 computation partition, parallelism is only affected by synchronization between nodes. To avoid reporting internalized data dependences, the Fortran D compiler applies algorithms to distinguish *cross-processor dependences* [11]. These dependences are calculated by examining the pattern of data accesses of both endpoints of the dependence to determine whether they occur on separate processors. Cross-processor dependences can also be used to point out references and *cross-processor loops* that cause synchronization.

2.3.2 Parallel and Pipelined Loops Using information from cross-processor dependences, the Fortran D compiler can classify computations as parallel, pipelined, or sequential [11]. 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.

2.4 Communication

The third and final class of information concerns the communication a Fortran D/HPF program will generate. Because messages can be expensive, it has an important effect on program performance. The interface obtains information during the communication analysis and optimization phase of the Fortran D compiler. All array references are examined to determine if they cause any communication. For references that cause communication, augmented regular section descriptors (RSDs) containing detailed information about the communication are constructed and attached at the individual reference and at the loop where the communication is inserted. The D editor will later extract this information and translate it into text containing the details on the messages.

3 D Editor Interface

3.1 Graphical Display

In this section, we describe how the D editor presents its data in a natural, easily understood manner. The D editor display is derived from the display of the ParaScope editor [8, 14], but contains significantly more information. In order for the user to understand the performance implications of his or her program, 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.

3.1.1 Overview Pane The D editor display is separated into five panes. The first component 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.

3.1.2 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.

3.1.3 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. The message type of a message describes both its location and effect on parallelism with respect to a loop. Messages labeled *outside* take place once for the entire loop nest and are placed outside the loop. Messages labeled *inside* take place once per iteration and are placed inside the loop. Both message types allow execution to occur fully in parallel. In comparison, messages labeled *pipelined* 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.

3.1.4 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 full specification of the decomposition. Arrays that are not partitioned are listed without alignment and distribution specifications.

3.1.5 Source Pane Finally, a separate *source 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, variable references that cause communication are colored red, yellow, or green depending on their 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.

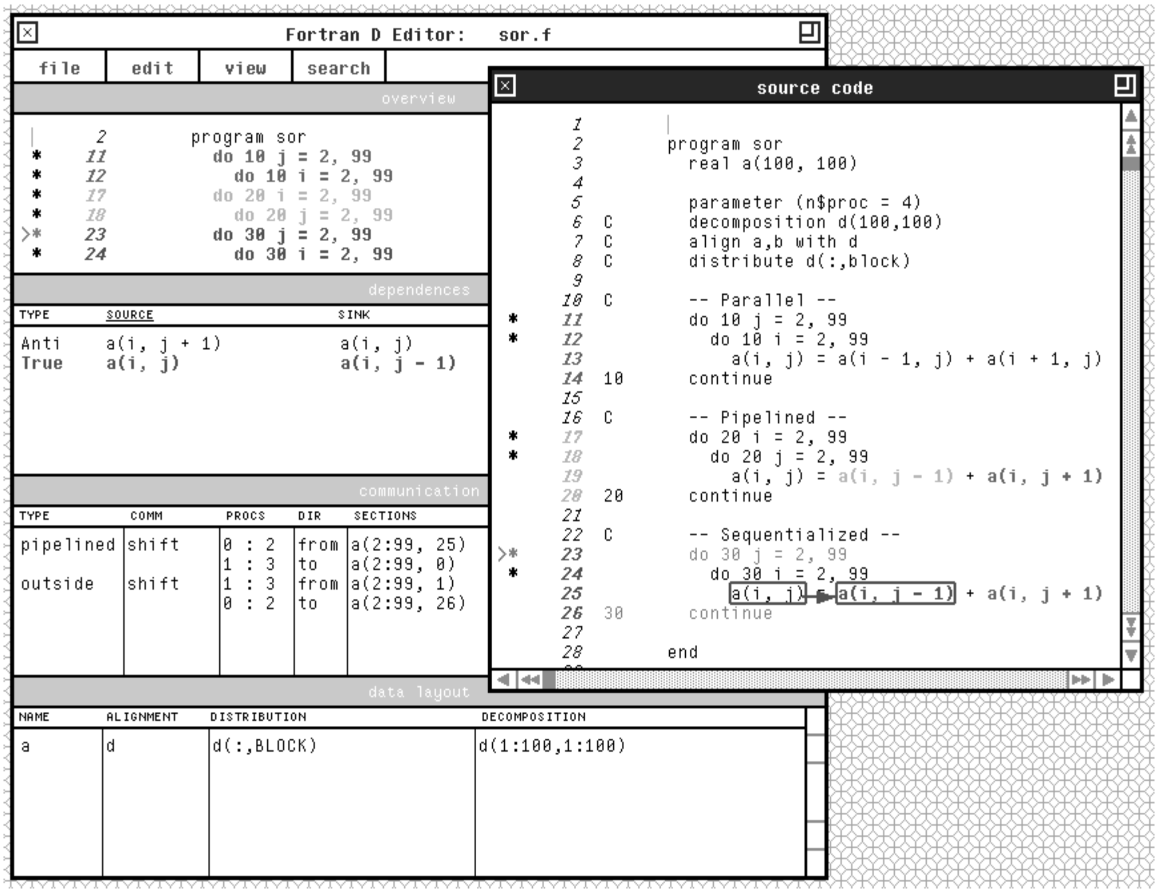

FIG. 3. *D Editor with SOR in Session*

3.2 SOR Example

To demonstrate how the D editor is used, we reconsider our SOR example. Figure 3 displays the user interface of the D editor for different versions of SOR (though the figure is black and white, the D editor display uses color). The overview pane in the top left corner summarizes the three loop nests present. The color of the loop nests (green, yellow, and red) show that they are parallel, pipelined, and sequentialized, respectively. The `>` symbol is next to loop `j` at line 23, indicating the third loop nest has been selected by the user.

The dependence pane displays the two dependences carried by the `j` loop. The first is an anti-dependence; since it has no effect on parallelism it is not colored (*i.e.,* black). In comparison, the true data dependence between `a(i,j)` and `a(i-1,j)` results in communication that sequentializes the loop nest. It is therefore colored red, alerting users to its significance.

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*; however, since the loop is outermost the computation is actually sequentialized. In comparison, the second message (corresponding to the reference `a(i+1,j)`) is inserted outside the loop nest and does not impede parallel execution. The communication type of both messages corresponds to a shift between pairs of logically neighboring processors. The sending and receiving processors are listed, as well as the actual portions of each array communicated. The data layout pane shows that the array `a` is identically aligned with the decomposition `d`, which is distributed block-wise in the third dimension.

The source pane displays the entire SOR example program. The line numbers 11–14 are colored green, indicating the first loop nest can be executed in parallel. The line numbers 17–20 are colored yellow, signifying the second 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 causing 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 numbers 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. The red arrow from `a(i,j)` to `a(i,j-1)` at line 25 shows the endpoints of the cross-processor dependence sequentializing the computation. The reference `a(i,j+1)` at line 25 is colored green. As before, it causes communication but does not affect parallelism.

3.3 Work Model

With the powerful presentation capabilities of the D editor, users can quickly track down performance bottlenecks in their parallel programs. Once a program has been analyzed, users can zoom in on loop nests displayed in red in the overview pane; these loops correspond to portions of the computation that restrict parallelism. The communication and source panes then provide detailed information on the impediments to parallelism.

If the user is dissatisfied after examining the parallelism and communication estimates, he or she may attempt to improve the program performance by making changes to the data decomposition or program structure and saving the program. The D editor can then reanalyze the program and present the updated results, allowing the user to evaluate the effect on either parallelism or communication. The key point is that since the D editor presents results using static analysis at the source level, feedback is both quick and easy to understand.

4 Conclusions

Languages such as Fortran D and High Performance Fortran can potentially make it much easier for scientists to exploit the power of MIMD distributed-memory machines. However, evaluating the performance of these programs can be difficult. In this paper we described the design and implementation of the prototype D editor, an intelligent interactive editor that statically evaluates the parallelism and data movement of parallel Fortran programs.

We listed 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. A color-coded presentation system allows users to quickly find performance bottlenecks in their program. 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.

The current implementation of the D editor is a valuable tool for interactively displaying the results of compile-time analysis. Researchers at Rice are working on enhancements to the D editor to provide interactive program transformations [15], automatic data decomposition [2], performance statistics [3, 13], and additional graphics interfaces. We believe the D editor is representative of a new generation of tools that will be needed to assist scientists programming scalable parallel machines.

5 Acknowledgements

We are grateful to the ParaScope and Fortran D research groups for providing the underlying software infrastructure for the D editor.

References

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

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

[3] 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.

[4] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

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

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

[16] 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.

[17] 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.

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