Experiences Using the Parascope Editor: an Interactive Parallel Programming Tool

Mary W. Hall *Timothy J. Harvey*
Ken Kennedy *Nathaniel McIntosh*
Kathryn S. McKinley *Jeffery D. Oldham*
Michael Paleczny *Gerald Roth*

CRPC-TR93297
February 1993

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

To appear in the Proceedings of the Symposium on the Principles and Practice of Parallel Programming, San Diego, CA, May 1993.

Experiences Using the ParaScope Editor: an Interactive Parallel Programming Tool *†

Mary W. Hall‡ Timothy J. Harvey§ Ken Kennedy§ Nathaniel McIntosh§
Kathryn S. McKinley¶ Jeffrey D. Oldham‡ Michael H. Paleczny§ Gerald Roth§

Abstract

The ParaScope Editor is an interactive parallel programming tool that assists knowledgeable users in developing scientific Fortran programs. It displays the results of sophisticated program analyses, provides a set of powerful interactive transformations, and supports program editing. This paper summarizes experiences of scientific programmers and tool designers using the ParaScope Editor. We evaluate existing features and describe enhancements in three key areas: user interface, analysis, and transformation. Many existing features prove crucial to successful program parallelization. They include interprocedural array side-effect analysis and program and dependence view filtering. Desirable functionality includes improved program navigation based on performance estimation, incorporating user assertions in analysis and more guidance in selecting transformations. These results offer insights for the authors of a variety of programming tools and parallelizing compilers.

1 Introduction

The complexity of constructing parallel programs is a significant obstacle to widespread use of parallel computers. In the process of writing a parallel program, programmers must consider the implications of concurrency on the correctness of their algorithms. Ideally, an advanced compiler could free the programmer from this concern by automatically converting a sequential program into an equivalent shared-memory parallel program. Although a substantial amount of research has been devoted to automatic parallelization, such systems are not consistently successful [1, 9, 6, 30, 33]. When automatic systems fail, the ParaScope Editor (PED) assists in parallelizing Fortran applications by combining programmer expertise with extensive analysis and program transformations [3, 27, 28].

This paper reports on the utility of PED as an interactive parallel programming tool in the ParaScope parallel programming environment [10]. Through user evaluation of PED, we assess existing functionality and suggest enhancements. The critiques confirm the usefulness of tools like PED and offer new insights for compiler writers and parallel programming tool designers in three key areas.

User Interface. We describe a new user interface that was designed and implemented based on previous evaluations [17, 25]. This interface enhances the original [28], unifying improved navigation and viewing of the program, its dependences, and information about its variables. User evaluation confirms the appropriateness of the interface but exposes the need for new features such as navigation assistance based on performance estimation and a facility enabling users to assert information about variable values.

Program Analysis. Dependence analysis specifies a partial order of a program's statements that preserves its meaning [5, 15]. For safety, the compiler must assume a dependence exists if it cannot prove otherwise. In ParaScope, analysis of interprocedural and intraprocedural constants, symbolics and array sections improve the precision of its dependence analysis [19, 27]. (We assume the reader is familiar with these analyses.) User evaluation confirms they are indispensable for discovering parallelism. In addition, more advanced symbolic analysis and exploiting user assertions in all analyses are essential when parallelizing many programs.

Transformations. Program transformations introduce, discover, and exploit parallelism without changing the meaning of the program. PED provides users with the same source-to-source transformations and accompanying analysis employed by ParaScope's compilers [8, 22], allowing users to apply transformations di-

*To appear in the Proceedings of the *Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May, 1993

†This research was supported by The Center for Research and Parallel Computation (CRPC) at Rice University, under NFS Cooperative Agreement Number CCR-9120008.

‡Center for Integrated Systems, Stanford University, Palo Alto, CA 94305. §Department of Computer Science, Rice University, Houston, TX 77251-1892. ¶ENSMP (CRI), 35, rue Saint Honoré, F-77305 Fontainebleau Cedex, France.

name	description & contributor	lines	procedures
spec77	weather simulation code Steve Poole, IBM Kingston & Lo Hsieh, IBM Palo Alto	5600	67
neoss	thermodynamics code Mary Zosel, Lawrence Livermore National Laboratory	350	5
nxsns	quantum mechanics code John Engle, Lawrence Livermore National Laboratory	1400	11
dpmin	molecular mechanics and dynamics program Marcia Pottle, Cornell Theory Center	5000	52
slab2d	2-D severe storm fluid flow prototype Roy Heimbach, National Center for Supercomputing Applications	550	9
slalom	benchmark program Roy Heimbach, National Center for Supercomputing Applications	1200	13
pueblo3d	hydrodynamics benchmark program Ralph Brickner, Los Alamos National Laboratory	4000	50
arc3d	3-D hydrodynamics code Doreen Cheng, NASA Ames Research Center	3600	25

Table 1: Analyzed and Parallelized Programs.

rectly. User evaluation reveals that access to transformations is not sufficient. Users found it difficult to determine which transformations to attempt and request that the system suggest a transformation or compound transformations.

This paper is organized into 4 more sections, related work and conclusions. In the next section, we present our evaluation methodology. Sections 3 through 5 discuss the three key areas in detail, briefly describing existing functionality, evaluating it and outlining obstacles to parallelization and proposed solutions.

2 Methodology

The experiences presented in this paper result primarily from a workshop held in July, 1991 at Rice University. Eight researchers from industrial sites and government laboratories attended. Each contributed a Fortran code to be parallelized with PED. Table 1 lists the workshop attendees with their programs. Additional evaluations result from two studies of parallel programming tools: one by Joseph Stein, a visiting scientist at Syracuse University [35], and another by Katherine Fletcher of Rice University and Doreen Cheng at NASA Ames Research Center [16].

At the workshop, we divided the attendees into groups of one or two people and assigned a researcher familiar with PED to assist each group. After a morning of introductory talks and a PED demonstration, the user groups parallelized their application with PED for the next two half days. They followed the work model described in Section 3.1.

We collected feedback from the workshop through two sources. First, the assistants recorded observations from the hands-on sessions with their user groups: which features were used, any difficulties encountered, and participants' comments. Each group also presented their experiences. We concluded with a lengthy discussion involving all attendees.

The contents of this paper are derived from the assistants' accounts and the discussions as well as talks by Stein and Fletcher on their experiences.

3 User Interface

3.1 Background

The user interface communicates PED's capabilities to programmers. Repeated evaluation of PED [17] and its predecessor PTOOL [2, 25] by compiler writers, application programmers and human interface experts has led to its current design. Although many of the features in the new PED interface are found in the original [28], it improves upon the original by providing more consistent and powerful features.

PED employs three user interface techniques to manage and present the detailed information computed by the system: (1) a book metaphor with *progressive disclosure*, (2) user-controlled *view filtering*, and (3) *power steering* for complex actions. The book metaphor portrays a Fortran program as an *electronic book* with analysis results annotating the source, analogous to a book's marginal notes, footnotes, appendices, and indices [37]. Progressive disclosure presents details incrementally as they become relevant [34]. View filtering emphasizes or conceals parts of the book as specified by a user [14]. Power steering automates repetitive or error-prone tasks, providing unobtrusive assistance while leaving the user in control.

The layout of a PED window is shown in Figure 1. The large area at the top is the *source pane* displaying the Fortran text. Two footnotes beneath it, the *dependence pane* and the *variable pane,* display dependence and variable information. All three panes operate similarly. Scroll bars can be used to bring other portions of the display into view, predefined or user-defined view filters may be applied to customize the display, the mouse may be used to make a selection, and menu choices or keyboard input may be used to edit the displayed infor-

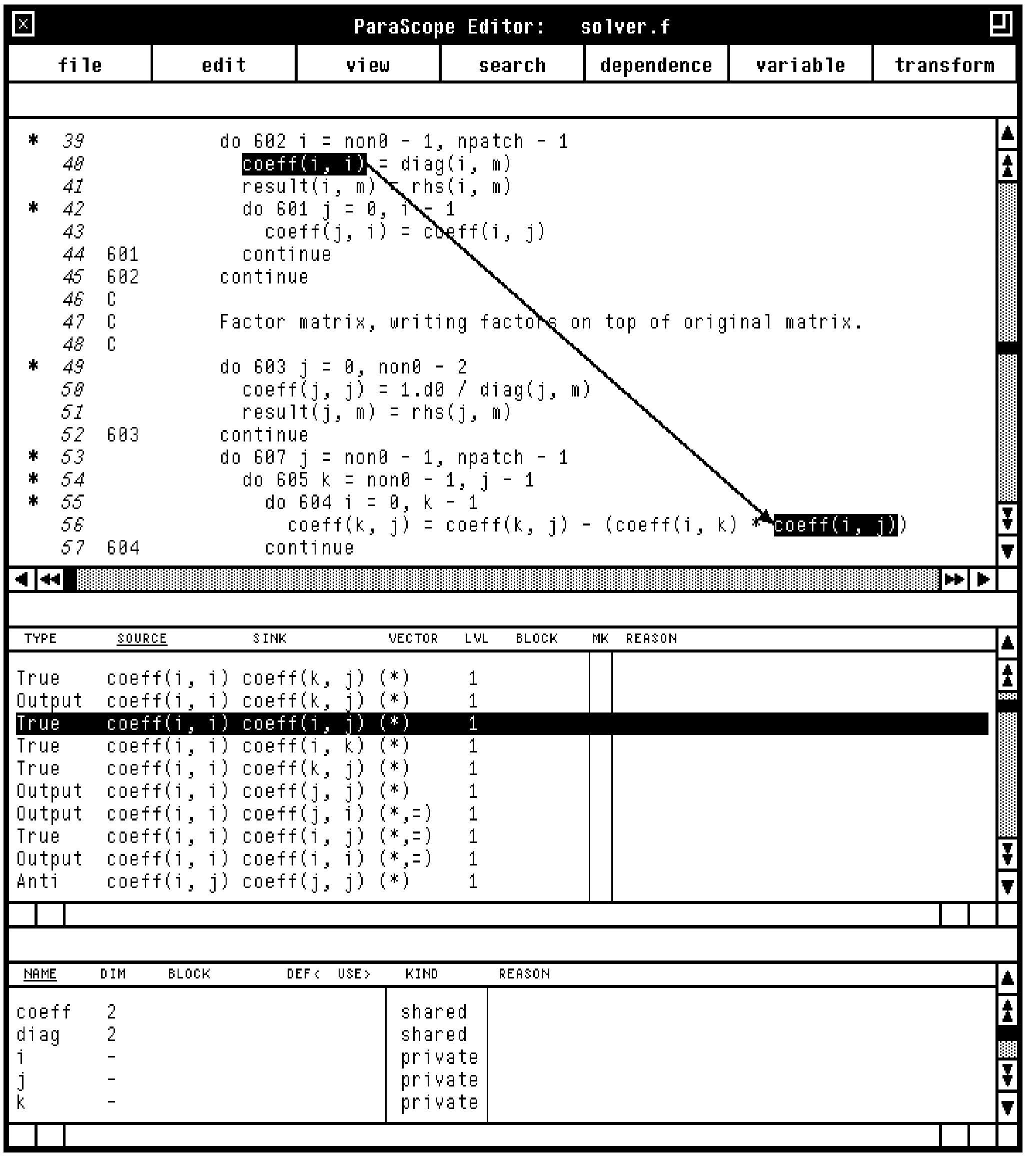

Figure 1: The ParaScope Editor.

mation. The Fortran text, the dependence information, and the variable information are each described below in terms of the information displayed, view filtering, and editing. We also discuss the work model best supported by PED and used during the workshop.

Fortran Source Code. Fortran source code is displayed in pretty-printed form in the source pane. At the left are marginal annotations showing the ordinal number of each source line and asterisks ($*$) which indicate the start of each loop. The ordinal line numbers for the statements in the *current loop* are highlighted with color (not visible in Figure 1). Source view filtering can be used to highlight or conceal source lines, based on either the text or the underlying semantics of each line. For example, source view filter predicates can test attributes of a line such as if it contains certain text, if it is a loop header, or if it is erroneous.

The source pane allows arbitrary editing of the program using mixed text and structure editing techniques [36]. The user has the full freedom to edit character by character or to use the power steering afforded by template-based structure editing. Incremental parsing occurs in response to edits, and the user is immediately informed of any syntactic or semantic errors.

Dependence Information. The editor reveals program dependences in a tabular list in the dependence

	used	improve	like	dislike
user interaction				
dependence deletion	⋆⋆⋆⋆⋆⋆			⋆⋆⋆
variable classification	⋆⋆⋆⋆			
access to analysis		⋆⋆⋆		
navigation				
program	⋆⋆⋆⋆⋆⋆⋆	⋆⋆⋆⋆⋆⋆⋆	⋆⋆	
dependence	⋆⋆⋆⋆⋆⋆⋆	⋆⋆⋆⋆⋆	⋆⋆	⋆
view filtering	⋆		⋆	
other				
detect interface error	⋆⋆⋆			
help	⋆⋆	⋆	⋆⋆	
teaching tool			⋆⋆	

⋆: one group of users out of a possible seven
used: feature was used
improve: current implementation should be improved
like: liked current implementation of feature
dislike: did not like current implementation of feature

Table 2: User Interface Evaluation.

pane. The display shows each dependence's source and sink variable references and characteristics such as dependence distance. PED uses progressive disclosure of dependences based on the current loop. When the user expresses interest in a particular loop by selecting its marker in the source pane, the selected loop's dependences immediately appear in the dependence pane. The user may select a range of dependences in the dependence pane to examine that are then highlighted. Simultaneously, the source pane graphically displays selected dependences using red arrows from source variable references to sink variables. In Figure 1, the user has selected a single dependence to be displayed, which is reflected in the source and dependence panes.

Dependence view filter predicates can test the computed and user-controlled attributes of a dependence, such as its source and sink variable references and line numbers, its type, loop nesting level, *mark* and *reason*. The filtered dependence list may then be displayed graphically all at once, one dependence at a time, or according to line numbers in the source.

The dependence pane also permits an important form of editing known as *dependence marking*. The system marks each dependence as either *proven*, *pending*, *accepted* or *rejected*. If PED proves a dependence exists with an exact dependence test [19], the dependence is marked as *proven*; otherwise it is marked *pending*. Users may sharpen PED's dependence analysis by marking a pending dependence as *accepted* or *rejected*. Rejected dependences are disregarded when PED considers the safety of a parallelizing transformation, but they remain in the system so the user can reconsider them at a later time. If desired, the user can edit the dependence's *reason* attribute to attach a comment explaining the classification decision. A *Mark Dependences* dialog box provides power steering for dependence marking by allowing the user to classify in one step an entire set of dependences that satisfy a chosen predicate.

Variable Information. The variable pane displays a list of variables with progressive disclosure similarly based on the current loop. The attributes of a variable include its name, definitions and uses outside the current loop, dimension, common block if any, and shared or private status. View filter predicates can test any of these attributes.

The variable pane also supports editing in the form of *variable classification*. Locating variables that may legally be made private to a loop body can greatly reduce the number of spurious dependences in a loop (see Section 4.1). Analysis automatically locates many such variables, but it is conservative and may classify variables as shared that may legally be made private. With variable classification, the user may edit a variable's *shared/private* attribute to correct overly conservative classifications for arrays and scalars. As with dependence editing the user may also edit a variable's *reason* field, attaching a comment regarding the variable's classification. A *Classify Variables* dialog box (analogous to *Mark Dependences*) provides power steering for variable classification.

Work Model. The following work model was used by the workshop participants. Users select a loop for consideration and examine any parallelism inhibiting dependences. If they are the result of overly conservative assumptions, the user employs dependence deletion and variable classification to increase the precision of analysis. If necessary, they perform transformations to expose parallelism. PED supports this model by displaying the dependences and variables associated with a selected loop and relating them back to the source code. During the workshop, the attendees augmented this work model with program execution profiles to help them focus on the most computationally intensive loops in their program.

3.2 Evaluation

This section describes the evaluation of PED's user interface, as summarized in Table 2. The table lists a number of features in its rows and responses in the columns. Each of the five workshop groups, along with Fletcher and Stein, is represented by an asterisk, for a total of seven possible asterisks. Because the implementation of the interface described above was not robust at the time of the workshop, it was demonstrated but not used by the participants. Here, we address only issues that apply to both interfaces and are not addressed in the current implementation. This structure allows us to focus on user suggestions for improvement.

User Interaction. Users were comfortable with selecting a loop and then evaluating its dependences and variable annotations. Specifically, one user commented that having all information available in one window annotating the source code was preferable to the functional-based approach of Forge [31], which requires the programmer to bring up a series of windows to access the results of program analysis.

To correct overly conservative analysis, users performed both dependence deletion and variable reclassification to reflect their perception of the true program state. Variable reclassification proved to be the right level of abstraction. When users were able to determine a variable could be made private to a loop, changing the variable's classification and seeing the resulting decrease in dependences was effective. However, deleting individual dependences was too tedious for most users. One user disliked dependences as a paradigm and desired a higher level of abstraction based on variables. We discussed simplifying dependence deletion by introducing variable-based user assertions that the system would then use to automatically eliminate dependences (see Section 3.3).

Three users requested more access to all the analysis computed by the system in order to enhance their understanding of the program and provide insight into the reasons the system must assume dependences exist. For instance, one user felt that displaying array sections that summarize the portion of an array accessed by a loop, similar to those used in interprocedural analysis (see Section 4.1), would provide an intuitive representation of the accesses within the loop. Two other users requested that the system display constant values or ranges for loop bounds and subscript expressions. In general, they wanted access to any information the system uses to compute dependences in order to correct overly conservative analysis.

Program Navigation. Existing features in PED assist in navigating the source code. They include view filtering and annotations in the display. Source view filtering was not widely used during the workshop primarily because it was unfamiliar to the assistants. However, one group defined filters based on labels to help them understand a procedure's control flow (see Section 5.3). One user commented that they would have liked to apply view filtering to examine the procedure's loop structure; this feature is one of the predefined view filters. Another requested that the display annotations include a loop nesting depth indicator.

All users requested more assistance in locating the most computation-intensive procedures and loops in the program in order to better target their parallelization efforts. For this purpose, the users relied on external tools to profile their codes. They used the Unix utility *gprof* from Sun and Cray executions to derive procedure-level profiles. One user brought loop-level profiling available from Forge [31], which shows the loop structure of a procedure, with the number of iterations executed and its percentage of execution time. The users requested that similar profiling or static performance estimation be integrated into PED to help focus user attention on the loops where effective parallelization would have the highest payoff. ParaScope now includes a static performance estimator used to predict the relative execution time of loops and subroutines in parallel programs [26].

Several users wanted a graphical representation of the call graph, rather than the current textual presentation [21]. A visual program representation provides a much needed "big picture" when working with a large or unfamiliar program. Short cuts to accessing other procedures via this representation were also requested.

Dependence Navigation. In navigating dependences, the users requested a number of features subsumed by the current implementation. First, they needed to visit dependence endpoints quickly rather than having to scroll through the source. They wanted view filtering capabilities to examine in the source and dependence panes all dependences associated with a particular definition. They were comfortable with use-definition chains and requested a filter mechanism for viewing them.

The users also wanted additional navigational features to examine dependences with at least one endpoint outside the current loop, in another loop or even another procedure. Displaying dependences with endpoints outside the selected loop requires straightforward extensions to PED. For example, if a variable is used or defined outside the current loop, the variable display reveals that information. To find the reference(s) at the workshop, the users were forced to search the program. Now the user need only select the line number of interest in the USE or DEF field (see Figure 1), and the appropriate reference appears in the source pane.

Dependence endpoints spanning multiple procedures pose a more difficult problem. Because interprocedural side-effect information summarizes accesses, one endpoint may correspond to several reads or writes of an array in multiple procedure bodies. To fully support dependence navigation, PED must be able to display other procedures while iterating over all the endpoints corresponding to a dependence.

Other. Another ParaScope tool, the *Composition Editor*, compares a procedure definition to calls invoking it, ensuring the parameter lists agree in number and type [11, 21]. These types of errors exist in production codes because most compilers do not perform cross-procedure comparisons. Several mismatched parameters between a procedure call and its declaration as well as type errors were detected and subsequently corrected using this analysis. One user requested further analysis to ensure that common block declarations have the same shape in every procedure in which they appear and to perform static array bounds checking.

Two participants were eager to use PED as a teaching tool for parallel programmers at their institutions. They felt PED's tutorial nature could help explain parallel programming to novices. Two users found the interactive help facility useful, but one wanted it to provide more information. One user wanted the ability to print the program, dependences, and variable information.

3.3 Obstacles to Parallelization

User Assertions. Explicit deletion of dependences fails to capture the user's reason for eliminating a dependence. A user deletes a dependence or a group of dependences because of some additional knowledge about the program that the system is unable to detect statically. This information is usually at a higher level than a specific dependence. Users would prefer to specify a high-level assertion and then have the system respond by deleting associated dependences. They proposed adding a user assertion language.

To support user assertions, we first need to define an assertion language based on three important concepts. (1) Assertions should express program properties that are natural to a user. (2) Assertions should provide information to the system that is useful in eliminating dependences. (3) It should be possible for the system to verify the correctness of the assertions at run time.

For example, the assertion language could enable users to specify the value or range of values of a variable using familiar Fortran syntax. Two simple but very useful assertions encountered during evaluation were specifying relationships between two symbolic variables and the properties of *index arrays*, arrays used in subscript expressions of another array. Consider the following fragment from *pueblo3d*.

```
DO M = 1, 3
   MCN = ICN (M, IR)
   ...
   DO I = ISTRT(IR), IENDV(IR)
      ...
      ... = UF(I + MCN, 3)
      UF(I, M) = ...
ENDDO
```

This construction appears in 10 loop nests in *pueblo3d* and several of these consume the majority of the total execution time. MCN represents "my current neighbor" and is used to index linearized three-dimensional arrays. Relative to the above loop structure, this program ensures that MCN > IENDV(IR) - ISTRT(IR) and therefore, there are no loop-carried dependences on UF. As in this case, arriving at an appropriate and useful assertion may require some careful thought, but the system will be offering some assistance as well (see Section 4.3).

4 Program Analysis

4.1 Background

Discovering and evaluating potential parallelism in a program requires extensive program analysis. In particular, dependence analysis provides the fundamental paradigm for detecting parallelism. A loop that contains no *loop-carried* dependences may execute its iterations in parallel. Further, dependences are used to prove the safety of program transformations. This section briefly describes the dependence analysis and supporting analyses that are currently available in Ped [27].

Dependence Analysis. Ped detects data and control dependences. *Data dependences* are located by testing pairs of references in a loop. A hierarchical suite of tests is used, starting with inexpensive tests, to prove or disprove that a dependence exists [19]. *Control dependences* explicitly represent how control decisions affect statement execution [15].

Supporting Analysis. Scalar data-flow analysis, including def-use chains, constant propagation and symbolic analysis, provides additional information about the values and relationship of variables. They can vastly improve the precision of dependence analysis [19, 20]. Def-use chains expose dependences among scalar variables as well as linking all accesses to each array for dependence testing. A critical contribution of scalar data-flow analysis is recognizing scalars that are *killed*, or redefined, on every iteration of a loop and may be made private, thus eliminating dependences. Constant propagation can locate constant-valued loop bounds, step sizes and subscript expressions. Symbolic analysis locates auxiliary induction variables, loop-invariant expressions and equivalent expressions. It also performs expression simplification on demand.

Interprocedural Data-flow Analysis. One of the distinguishing features of Ped's dependence information is the incorporation of an extensive suite of interprocedural analysis techniques that determine the effects of procedure calls on variables. Interprocedural constants are inherited from a procedure's callers and directly incorporated into the intraprocedural constants. *Flow-insensitive side-effect analysis*, including Mod and Ref analysis, describes the variables that may be accessed on some control flow path through the procedure [4]. *Flow-sensitive side-effect* analysis, such as Kill analysis, describes accesses that occur on every possible control flow path [7]. *Regular section analysis* is also used to describe more precisely, when possible, the side-effects to portions of arrays [24].

4.2 Evaluation

Table 3 demonstrates the importance of existing analysis and the need for additional analysis for the programs described in Table 1.

Dependence Analysis. The *dependence* entry in Table 3 indicates whether dependence analysis locates parallel loops in each program. For all of the programs, the system is able to automatically detect many parallel loops. Small, inner parallel loops are almost always detected. However, outer loop parallelism is essential to achieving measurable performance improvements for applications programs on many parallel architectures, and it too often goes undetected. We discuss impediments and solutions for parallel loop detection in Section 4.3.

Scalar Kill Analysis. As illustrated by the *scalar kills* entry, almost all of the programs contain a loop

	spec77	neoss	nxsns	dpmin	slab2d	slalom	pueblo3d	arc3d
dependence	U	U	U	U	U	U	U	U
scalar kills		U	U	U	U	U	U	U
sections	U	U	U	U	U			U
array kills	N		N	N	N	N	N	N
reductions		N	N	N		N		N
index arrays				N		N	N	

U: existing analysis was used. *N:* additional analysis was needed.

Table 3: Analysis Used or Needed During Workshop.

that becomes parallelizable following scalar privatization. In the program nxsns, interprocedural scalar KILL analysis reveals a scalar variable is killed in a procedure invoked inside a loop. Experience using PTOOL, PED's predecessor, also suggests interprocedural scalar KILL analysis is useful in eliminating spurious dependences [25].

Interprocedural Side-effect Analysis. The *sections* entry indicates that scalar side-effect analysis or regular section analysis reduces the number of dependences on a loop containing a procedure call in six of the programs. Of the two remaining programs, one does not contain loops with procedure calls and analysis failed on the other. In spec77 and nxsns, interprocedural side-effect analysis reveals that loops containing procedure calls can safely execute in parallel.

4.3 Obstacles to Parallelization

There are several areas where existing analysis in PED is not sufficient to detect parallelism, but the users together with the workshop assistants were able to discover it. In many cases, more precise analysis can detect and eliminate overly conservative dependences. In others, static analysis will probably never be sufficient and user assertions are needed.

Array Kill Analysis. For loops in seven of the programs, array kill analysis would eliminate important dependences, revealing parallelism. In slab2d and arc3d, automatic privatization of one or more killed arrays is sufficient to prove that loops may be safely executed in parallel. Frequently, a temporary array is assigned and used in an inner loop and its value does not carry across iterations of the outer loop. In arc3d, an array is killed inside a procedure invoked in a loop, so interprocedural array kill analysis is required. To perform array privatization in slab2d, kill analysis must be combined with loop transformations. Because the need for array kill analysis has been discussed previously [6, 33], we do not elaborate further here.

Reductions. Five of the programs contain sum reductions which go unrecognized by PED. For example, computing the sum of all the elements of an array. Because addition is associative, the additions do not need to be performed in order and so the loop can be parallelized after restructuring the accumulation of the sum. The need and methods for recognizing reductions are well known and we do not elaborate here. However, transforming reductions in an interactive setting is complicated by the property that efficient execution requires an architecture-specific approach.

Symbolic Expressions. Static analysis cannot derive information about certain symbolic expressions, such as variables read from an input file or index arrays used in subscript expressions. Symbolic terms in subscript expressions are a key limiting factor in precise dependence analysis. One study found that over 50% of the array references in some numerical packages contained at least one unknown symbolic term [20]. The *index arrays* entry in Table 3 demonstrates that three programs contained index arrays in subscript expressions that prevented parallelization.

We are using a three-pronged approach to improving the precision of dependence information in the presence of symbolics: (1) sophisticated symbolic analysis; (2) *partial evaluation*, or compiling the program with all or part of an input data set [18]; and (3) incorporating user assertions to eliminate dependences (described from the user interface perspective in Section 3.3).

The following program fragment from the routine *filter3d* in *arc3d* demonstrates the type of advanced interprocedural symbolic analysis that would improve program parallelization.

```
      DO 15 N = 1, 5
        DO 16 J = 1, JM
          DO 16 K = 2, KM
            WR1(J,K) = Q(JPL,K,L,N)-Q(J,K,L,N)
   16   CONTINUE

        DO 76 K = 2, KM
          WR1(JMAX,K) = WR1(JM,K)
   76   CONTINUE
        ... = WR1(J,K)
        ...
   15 CONTINUE
```

In the initialization routine, the assignment JM = JMAX -1 occurs, and this relation holds for the rest of the program. Given this symbolic relationship and array kill analysis, the DO 15 loop may be safely parallelized by privatizing WR1 and two other arrays. The ability to detect and propagate this type of relationship can greatly improve the precision of dependence analysis.

In other cases, symbolic values are read, making it virtually impossible for static analysis to determine actual dependences. Consider the following fragment from the program *dpmin*.

```
      DO 300 N = 1, NBA
         I3 = IT(N)
         J3 = JT(N)
         K3 = KT(N)
         ...
         F(I3 + 1) = F(I3 + 1) - DT1
         F(I3 + 2) = F(I3 + 2) - DT2
         F(I3 + 3) = F(I3 + 3) - DT3
         F(J3 + 1) = F(J3 + 1) - DT4
         F(J3 + 2) = F(J3 + 2) - DT5
         F(J3 + 3) = F(J3 + 3) - DT6
         F(K3 + 1) = F(K3 + 1) - DT7
         F(K3 + 2) = F(K3 + 2) - DT8
         F(K3 + 3) = F(K3 + 3) - DT9
  300 CONTINUE
```

The arrays IT(N), JT(N) and KT(N) are read from a file, so the system assumes dependences connect all references to F.

To assist the user in deriving assertions that eliminate spurious dependences, the system may be able to derive *breaking conditions* that eliminate a particular dependence or class of dependences. In the above, a breaking condition for loop-carried dependences between instances of F(I3+1) is that IT(N) is a permutation array (*i.e* all values for I3+1 are unique). While possible, it would take significantly more analysis for the system to derive breaking conditions to eliminate all dependences on F and parallelize the loop. The system must recognize that if IT(N) is a function satisfying the constraint IT(I) + 3 $\leq$ IT(I+1) (similarly for JT and KT), IT(NBA) + 3 $\leq$ JT(1) and JT(NBA) + 3 $\leq$ KT(1), all dependences may be eliminated.

A similar approach is being pursued by Pugh and Wonnacott [32]. They derive relational constraints on variables during dependence testing using a variant of integer programming, and these are presented to the user in their implementation.

5 Program Transformation

5.1 Background

PED supports a large set of transformations proven useful for introducing, discovering, and exploiting parallelism and for enhancing memory hierarchy use [27]. Figure 2 shows a taxonomy of the transformations available in PED. Transformations are applied according to a *power steering* paradigm: the user specifies the transformations to be made, and the system provides advice and carries out the mechanical details. The system advises whether the transformation is *applicable* (is syntactically correct), *safe* (preserves the semantics of the program) and *profitable* (contributes to parallelization). The complexity of many transformations makes correct application difficult and tedious by hand. Thus, power steering provides safe, profitable and correct application of transformations and incremental updates of dependence information to reflect the modified program.

Reordering transformations change the order in which statements are executed, either within or across loop iterations. They expose or enhance loop-level parallelism and improve data locality. *Dependence breaking* transformations eliminate storage-related dependences that inhibit parallelism. They often introduce new storage and convert loop-carried dependences to loop-independent dependences. *Memory optimizing* transformations expose reuse of memory locations in registers or cache.

5.2 Evaluation

Table 4 lists the transformations used for parallelizing each program. The rows describe the existing transformations used and the additional transformations needed.

It is notable that only a few of PED's transformations were used. The most commonly used transformation was scalar expansion, which transforms a scalar into an array to eliminate loop-carried dependences. Loop unrolling was the only other transformation used more than once. For the most part, the transformations in the table are commonly used in vectorization; however, in one example loop interchange of an imperfect loop nest was required.

As compared with the previous section, users were much better at reproducing analysis not provided by the system than at determining which transformations could improve parallelization. The users and their assistants mentioned that selecting among the large number of transformations is too overwhelming. It was not clear to them which transformations to attempt for a given loop nest.

5.3 Obstacles to Parallelization

To assist users in program parallelization, they requested more automated assistance for applying transformations and two additional transformations.

Transformation Guidance. While few transformations were performed on these programs, it was *not* be-

Figure 2: Transformation Taxonomy for PED.

Reordering		
Loop Distribution	Loop Interchange	Loop Fusion
Statement Interchange	Loop Skewing	Loop Reversal
Dependence Breaking		
Privatization	Array Renaming	Loop Peeling
Scalar Expansion	Loop Splitting	Loop Alignment
Memory Optimizing		
Strip Mining	Scalar Replacement	
Loop Unrolling	Unroll and Jam	
Miscellaneous		
Sequential $\leftrightarrow$ Parallel	Loop Bounds Adjusting	
Statement Addition	Statement Deletion	

	spec77	neoss	nxsns	dpmin	slab2d	slalom	pueblo3d	arc3d
loop distribution		U						
loop interchange							U	
loop fusion		U						
scalar expansion		U	U	U				
loop unrolling			U			U		
control flow		N	N	N				
interprocedural	N							

U: existing transformation was used. *N:* new transformation was needed.

Table 4: Transformations Used and Needed During the Workshop.

cause opportunities for transformation did not exist. Indeed, a substantial amount of research has demonstrated the value of loop transformations in exposing parallelism. Cursory re-examination of the programs reveals opportunities for many transformations that enable parallelization of outer loops. These opportunities include fusion and interchange in pueblo3d and distribution in dpmin and neoss. However, when the users were confronted with the selection of transformations, they did not know which ones to explore.

Several users want the transformation selection to include only those which are safe and profitable for the currently selected loop. This structure would save them from sifting through the entire list of transformations for each loop. As a simple extension to the current system, it could evaluate the safety of all the transformations for a particular loop on demand and present only the safe ones. However, determining what transformations are profitable is much more difficult. Profitability not only depends on machine specifics, but on subsequent transformations.

According to the users, transformation advice should incorporate the compiler's parallel code generation algorithms for a particular architecture. Ideally, a user would select the architecture and request parallelization at the loop, subroutine or program level. The system would then automatically perform parallelization or describe the impediments to a desired parallelization. Impediments would be presented in a systematic fashion based on the relative importance of a loop or subroutine. The user could evaluate any impediments and correct overly conservative assumptions, thus enabling *semi-automatic* parallelization. Several users stressed the importance of providing consistent analysis and parallelization algorithms between the compiler and interactive tool.

Complex Control Flow. Three programs, neoss, nxsns and dpmin, were written in dialects of Fortran that do not support structured *if* statements and that require *do* loops to execute at least one iteration regardless of loop bounds. Possibly to compensate for constructs lacking in the language, programmers introduced complex control flow involving *goto* statements. Consider the following loop with GOTOs from neoss and the structured equivalent produced by hand during the workshop.

```
      DO 50 K= ...                    original
        <b1>
        IF (DENV(K) - RES(NR+1)) 100, 10, 10
10      CONTINUE
        <b2>
        GOTO 101
100     <b3>
101     <b4>
50    CONTINUE
                           ⇓  structured version
      DO 50 K=...
        <b1>
        IF (DENV(K) .GE. RES(NR+1)) THEN
          <b2>
        ELSE
          <b3>
        ENDIF
          <b4>
50    CONTINUE
```

The *goto*s make it difficult for the users to understand the original loop. Other variations of *if-then-else* constructs formed with *goto*s also appear in these programs. However, users were able to further transform and parallelize a loop of this sort after control flow was simplified by hand.

To assist users in this process, the simplification of complex control flow can be automated by recognizing and substituting structured idioms for unstructured control-flow when appropriate. The need for this transformation is unique to an interactive setting. It is not necessary in completely automatic systems or internally in interactive tools because control dependence suffices to understand control flow regardless of the language constructs.

Interprocedural Transformations. The program spec77 has a number of loops containing procedure calls in the key procedure *gloop*. Interprocedural analysis indicates that the loops may be safely parallelized, but the loops have at most twelve iterations, limiting the number of possible parallel threads. The procedures invoked in these loops however contain outer loops with many more iterations that may also safely execute in parallel. A solution that combines the granularity of the outer loop with the parallelism of the loop in the procedure is to perform loop interchange across the procedure boundary [23]. In some cases, loops in *gloop* contained multiple calls so the loops of the called procedures were first fused before applying interchange.

In order to enable transformations such as loop interchange and loop fusion across procedure boundaries, we must be able to move a loop into or out of a procedure invocation. We call these interprocedural transformations *loop embedding* and *loop extraction*, respectively [23]. Steve Poole brought `spec77` to the workshop because he was familiar with this work and wanted to perform these transformations in PED. Embedding and extraction are not currently implemented in PED.

6 Related Work

A few other papers report on the effectiveness of existing automatic parallelizing compilers on large applications [9, 6, 12, 13, 33] and of interactive tools [9]. Two of these suggest compiler-programmer interaction to achieve parallelization [9, 33].

Blume and Eigenmann explore the effectiveness of KAP [29] applied to Perfect benchmark programs on an 8-processor Alliant [6]. Half of the programs demonstrate little or no improvement following parallelization. Using the transformation algorithms in KAP, scalar expansion is the only transformation that consistently improved performance on several codes. The compiler often fails to parallelize important loops, such as loops containing procedure calls, and sometimes parallelizes loops with insufficient granularity. These failures are in part because the compiler does not perform interprocedural analysis.

Eigenmann et al. present novel approaches for improving parallelization in four Perfect programs [12, 13]. These techniques include run-time dependence testing and aggressive use of synchronization to guard critical sections.

Singh and Hennessy examine parallelization of three programs on an 8-processor Alliant [33]. They observe that certain programming styles interfere with compiler analysis. These include linearized arrays and specialized use of the boundary elements in an array. To aid the compiler in selecting appropriate loops to parallelize, they suggest user access to profiling information and assertion facilities that allow specifying ranges of symbolic variables.

Cheng and Pase consider 25 programs running on an 8 processor Cray Y-MP, using Cray fpp, KAP/Cray and Forge to introduce parallelism [9]. Most of the parallel versions demonstrate a speedup of less than 2 over their vector counterparts. When using Forge, the only interactive tool, they offer two suggestions. First, the user should be given insight about what loops to parallelize, either through profiling or performance estimation. Second, they want the system to query for unknown scalar variable values and use these assertions in analysis to eliminate dependences.

Because PED contains some of the features recommended by these studies, our evaluation reveals how they work in practice. For example, most of these studies find interprocedural analysis to be essential, but missing from the compilers under investigation. In PED, interprocedural analysis is found effective, but more advanced analysis such as interprocedural symbolic propagation is also needed.

Our evaluation is also distinguished because it examines the interactive parallelization process with outside users. The existence of an advanced interactive tool allowed us to go beyond the comments provided by the studies of automatic parallelizers to investigate the appropriate level for compiler and tool interaction. For instance, users deleted dependences in PED, as suggested previously [25, 33], but requested higher-level assertions and guidance. We also uncovered the need for new features such as access to compiler transformation algorithms and control flow simplification.

7 Conclusion

In an interactive system, the combination of user expertise with the sophisticated analysis and transformations used by parallelizing compilers requires powerful mechanisms and careful engineering. Through user evaluation of the ParaScope Editor, an interactive parallel programming tool, we have established several essential user requirements. Users prefer that the user interface tie the compiler analysis to the source code and provide facilities such as view filtering of source code and analysis results, navigational assistance, and transformation guidance. To assist users in refining program analysis for use in compilation, a facility that communicates high-level information to the tool in the form of assertions is needed. Advanced analysis, such as interprocedural array section analysis, symbolic analysis and array privatization, can substantially reduce the amount of work the user must perform. By continuing extensive evaluations, improvements and additions, the ParaScope Editor aspires to meet user requirements.

8 Acknowledgements

We would like to thank all of the researchers who participated in this study. We also thank the members of the ParaScope programming environment group, past and present, who participated in the implementation of PED and the infrastructure upon which it is built. We especially appreciate Scott Warren's implementation of the new user interface.

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[2] J. R. Allen, D. Bäumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986. IEEE Computer Society Press.

[3] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

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

[5] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, October 1966.

[6] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[7] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[8] S. Carr. *Memory-Hierarchy Management*. PhD thesis, Dept. of Computer Science, Rice University, September 1992.

[9] D. Cheng and D. Pase. An evaluation of automatic and interactive parallel programming tools. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[10] 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.

[11] K. Cooper, K. Kennedy, L. Torczon, A. Weingarten, and M. Wolcott. Editing and compiling whole programs. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, CA, December 1986.

[12] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua. Restructuring Fortran programs for Cedar. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[13] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[14] D. C. Engelbart and W. K. English. A research center for augmenting human intellect. In *Proceedings of AFIPS 1968 Fall Joint Computer Conference*, San Francisco, CA, December 1968.

[15] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[16] K. Fletcher. Experience with ParaScope at NASA Ames Research Center. Presentation at the ParaScope Editor Workshop, July 1991.

[17] K. Fletcher, K. Kennedy, K. S. McKinley, and S. Warren. The ParaScope Editor: User interface goals. Technical Report TR90-113, Dept. of Computer Science, Rice University, May 1990.

[18] G. Goff. Practical techniques to augment dependence analysis in the presence of symbolic terms. Technical Report TR92-194, Dept. of Computer Science, Rice University, October 1992.

[19] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[20] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high-performance parallelizing compilers. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

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

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

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

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

[25] L. Henderson, R. Hiromoto, O. Lubeck, and M. Simmons. On the use of diagnostic dependency-analysis tools in parallel programming: Experiences using PTOOL. *The Journal of Supercomputing*, 4:83–96, 1990.

[26] 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.

[27] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[28] 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.

[29] Kuck & Associates, Inc. *KAP User's Guide*. Champaign, IL 61820, 1988.

[30] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Dept. of Computer Science, Rice University, April 1992.

[31] Pacific-Sierra Research. *Forge User's Guide, version 7.01*, December 1990.

[32] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[33] J. Singh and J. Hennessy. An empirical investigation of the effectiveness of and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, Tokyo, Japan, April 1991.

[34] D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the Star user interface. *BYTE*, 7(4):242–282, April 1982.

[35] J. Stein. On outer-loop parallelization of existing, real-life Fortran-77 programs. Colloquium at Rice University, July 1991. In collaboration with M. Paul and G.C. Fox.

[36] R. C. Waters. Program editors should not abandon text oriented commands. *ACM SIGPLAN Notices*, 17(7):39–46, July 1982.

[37] N. Yankelovitch, N. Meyrowitz, and A. van Dam. Reading and writing the electronic book. *IEEE Computer*, 18(10):15–29, October 1985.