Interactive Parallel Programming Using the ParaScope Editor

Ken Kennedy

Kathryn S. McKinley

Chau-Wen Tseng

CRPC-TR 90096

October 1990

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

In *IEEE Transactions of Parallel and Distributed Systems*, 2(3):329–341, July 1991.

Interactive Parallel Programming Using the ParaScope Editor *

Ken Kennedy Kathryn McKinley Chau-Wen Tseng

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

July 1991

Abstract

The ParaScope project is developing an integrated collection of tools to help scientific programmers implement correct and efficient parallel programs. The centerpiece of this collection is the ParaScope Editor, an intelligent interactive editor for parallel Fortran programs. The ParaScope Editor reveals to users potential hazards of a proposed parallelization in a program. It also provides a variety of powerful interactive program transformations that have been shown useful in converting programs to parallel form. In addition, the ParaScope Editor supports general user editing through a hybrid text and structure editing facility that incrementally analyzes the modified program for potential hazards. The ParaScope Editor is a new kind of program construction tool – one that not only manages text, but also presents the user with information about the correctness of the parallel program under development. As such, it can support an exploratory programming style in which users get immediate feedback on their various strategies for parallelization.

Keywords: Parallel Programming, Parallelism Detection, Dependence Analysis, Transformations, Environments, Interactive, Editor

1 Introduction

The widespread availability of affordable parallel machines has increasingly challenged the abilities of programmers and compiler writers alike. Programmers, eager to use new machines to speed up existing sequential scientific codes, want maximal performance with minimal effort. The success of automatic vectorization has led users to seek a similarly elegant software solution to the problem of programming parallel computers. A substantial amount of research has been conducted on whether sequential Fortran 77 programs can be automatically converted without user assistance to execute on shared-memory parallel machines [1, 2, 6, 7, 41, 49]. The results of this research have been both promising and disappointing. Although such systems can successfully parallelize many interesting programs, they have not established a level of success that will make it possible to avoid explicit parallel

*This research is supported by the National Science Foundation under grants CDA-8619893 and ASC-8518578, IBM, and the Cray Research Foundation. Published in IEEE Transactions of Parallel and Distributed Systems, 2(3):329–341, July 1991.

programming by the user. Hence, research has turned increasingly to the problem of supporting parallel programming.

Systems for automatic detection of parallelism are based on the analysis of *dependences* in a program, where two statements depend on each other if the execution of one affects the other. The process of calculating dependences for a program is known as *dependence analysis*. A dependence crossing between regions that are executed in parallel may correspond to a *data race*, indicating the existence of potential nondeterminism in the program. In general, an automatic parallelization system cannot be allowed to make any transformation which introduces a data race or changes the original semantics of the program.

The systems for automatic detection of parallelism in Fortran suffer from one principal drawback: the inaccuracy of their dependence analysis. The presence of complex control flow, symbolic expressions, or procedure calls are all factors which limit the dependence analyzer's ability to prove or disprove the existence of dependences. If it cannot be proven that a dependence does not exist, automatic tools must be conservative and assume a dependence, lest they enable transformations that will change the meaning of the program. In these situations, the user is often able to solve the problem immediately when presented with the specific dependence in question. Unfortunately, in a completely automatic tool the user is never given this opportunity[1].

To address this problem, we developed PTOOL, an interactive browser which displays the dependences present in a program [5]. Within PTOOL, the user selects a specific loop and is presented with what the analyzer believes are the dependences preventing the parallelization of that loop. The user may then confirm or delete these dependences based on their knowledge of the underlying algorithms of the program. Although PTOOL is effective at helping users understand the parallelism available in a given Fortran program, it suffers because it is a browser rather than an editor [32]. When presented with dependences, the user frequently sees a transformation that can eliminate a collection of dependences, only to be frustrated because performing that transformation requires moving to an editor, making the change, and resubmitting the program for dependence analysis. Furthermore, PTOOL cannot help the user perform a transformation correctly.

The ParaScope Editor, PED, overcomes these disadvantages by permitting the programmer and tool each to do what they do best: the tool builds dependences, provides expert advice, and performs complex transformations, while the programmer determines which dependences are valid and selects transformations to be applied. When transformations are performed, PED updates both the source and the dependence information quickly and correctly. This format avoids the possibility of the user accidentally introducing errors into the program. As its name implies, the ParaScope Editor is based upon a source editor, so it also supports arbitrary user edits. The current version reconstructs dependences incrementally after any of the structured transformations it provides and for simple edits, such as the deletion or addition of an assignment statement. For arbitrary unstructured edits with a broader scope, batch analysis is used to reanalyze the entire program.

The current prototype of PED is a powerful tool for exploring parallel programs: it presents the program's data and control relationships to the user and indicates the effectiveness of program transformations in eliminating impediments to parallelism. It also permits arbitrary program changes

[1] Several automatic parallelization systems (for example, see [44]) provide a directive that instructs the compiler to ignore all dependences. The use of broad directives like this is unsound because of the danger that the user will discard real dependences with the false ones, leading to errors that are hard to detect.

through familiar editing operations. PED supports several styles of parallel programming. It can be used to develop new parallel codes, convert sequential codes into parallel form, or analyze existing parallel programs. In particular, PED currently accepts and generates Fortran 77,IBM parallel Fortran [35], and parallel Fortran for the Sequent Symmetry [45]. The Parallel Computing Forum is developing PCF Fortran [43]. PCF Fortran defines a set of parallel extensions that a large number of manufacturers are committed to accepting, obviating the current need to support numerous Fortran dialects. These extensions will be supported when they emerge.

The remainder of this paper is organized as follows. Section 2 discusses the evolution of the ParaScope Parallel Programming Environment and in particular the ParaScope Editor. Section 3 outlines the program analysis capabilities of PED, and Section 4 describes the manner in which dependence information is displayed and may be modified. A survey of the program transformations provided by PED appears in Section 5. Issues involving interactive programming in PED are discussed in Section 6. Section 7 summarizes related research, and Section 8 offers some conclusions.

2 Background

PED is being developed in the context of the ParaScope project [16], a parallel programming environment based on the confluence of three major research efforts at Rice University: $\mathbb{R}^n$, the Rice Programming Environment [22]; PFC, a Parallel Fortran Converter [7]; and PTOOL, a parallel programming assistant [5]. All of these are major contributors to the ideas behind the ParaScope Editor, so we begin with a short description of each. Figure 1 illustrates the evolution of ParaScope.

2.1 The $\mathbb{R}^n$ Programming Environment

PED enjoys many advantages because it is integrated into the $\mathbb{R}^n$ Programming Environment. Begun in 1982, the $\mathbb{R}^n$ Programming Environment project pioneered the use of interprocedural analysis and optimization in a program compilation system. To accomplish this, it has built a collection of tools that collaborate to gather information needed to support interprocedural analysis while preparing a program for execution. Included in this collection is a source editor for Fortran that combines the features of text and structure editing, representing programs internally as abstract syntax trees. Also available are a whole program manager, a debugger for sequential and parallel programs, interprocedural analysis and optimizations, and an excellent optimizing scalar compiler.

$\mathbb{R}^n$ is written in C and runs under X Windows. It is a mature environment that has been distributed in both source and executable form to many external sites. One of the goals of $\mathbb{R}^n$ has been a consistent user interface that is easy to learn and use. As with any large system consisting of many independent and related portions, another goal has been to create a modular, easily modified implementation. The resulting environment is well suited for integrating and extending our research in parallel programming. ParaScope includes and builds on all of the functionality of the $\mathbb{R}^n$ project.

2.2 PFC

The PFC project was begun in 1979 with the goal of producing an automatic source to source vectorizer for Fortran. It is written in PL/1 and runs on an IBM mainframe. In recent years the project has focused on the more difficult problem of automatically parallelizing sequential code. PFC performs

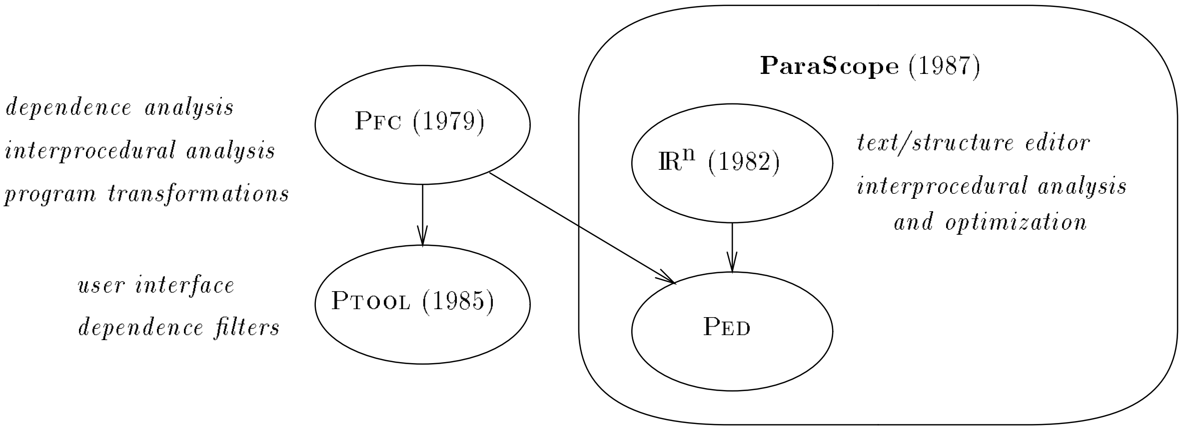

Figure 1 Evolution of ParaScope

data dependence analysis [8, 12, 13, 56], interprocedural side effect analysis [24] and interprocedural constant propagation [17]. More recently an implementation of *regular section analysis* [19, 31], which determines the subarrays affected by procedure calls, has been completed. This analysis significantly improves the precision ofPFC's dependence graph, because arrays are no longer treated as single units across procedure calls. PFC also performs *control dependence* analysis [27], which describes when the execution of one statement directly determines if another will execute.

The analyses performed inPFC result in a *statement dependence graph* that specifies a partial ordering on the statements in the program. This ordering must be maintained to preserve the semantics of the program after parallelization. The dependence graph is conservative in that it may include dependences that do not exist, but cannot be eliminated because of imprecise dependence analysis. In being conservative,PFC guarantees that only safe transformations are applied, but many opportunities for parallelism may be overlooked. A special version ofPFC has been modified to export the results of control and data dependence, dataflow, symbolic, and interprocedural analysis in the form of an ASCII file for use by PTOOL and PED.

PFC concentrates on discovering and enhancing loop level parallelism in the original sequential program. Although loops do not contain all the possible parallelism in a program, there are several reasons for focusing on them. In scientific and numerical applications, most computation occurs in loops. Also, separate iterations of a loop usually offer portions of computation that require similar execution times, and often provide enough computation to keep numerous processors occupied.

PFC has had many successes. It was influential in the design of several commercial vectorization systems [47], and it has successfully found near-optimal parallelism for a selected set of test cases [18]. However, it has not been successful enough to obviate the need for explicit parallel programming. In large complex loops, it tends to find many spurious race conditions, any one of which is sufficient to inhibit parallelization. Therefore, we have also turned our attention to the use of dependence information to support the parallel programming process.

2.3 PTOOL

PTOOL is a program browser that was developed to overcome some of the limitations of automatic parallelizing systems by displaying dependences in Fortran programs. It is in use as a debugging and analysis tool at various sites around the country, such as the Cornell National Supercomputing Facility. PTOOL was developed at the request of researchers at Los Alamos National Laboratory who wanted to debug programs parallelized by hand. It uses a dependence graph generated by PFC to examine the dependences that prevent loops from being run in parallel. To assist users in determining if loops may be run in parallel, PTOOL also classifies variables as shared or private.

When examining large scientific programs, users frequently found an overwhelming number of dependences in large loops, including spurious dependences due to imprecise dependence analysis [32]. To ameliorate this problem, a number of improvements were made in PFC's dependence analysis. In addition, a dependence filtering mechanism was incorporated in the PTOOL browser which could answers complex queries about dependences based on their characteristics. The user could then use this mechanism to focus on specific classes of dependences. PED incorporates and extends these abilities (see Section 4).

2.4 The ParaScope Editor

The ParaScope Editor is an interactive tool for the sophisticated user. Programmers at Rice, Los Alamos, and elsewhere have indicated that they want to be involved in the process of parallel programming. They feel the output of automatic tools is confusing, because it does not easily map to their original code. Often sequential analysis may be invalidated by the parallel version or, even worse, be unavailable. This complicates the users' ability to improve the modified source. They want their program to be recognizable; they want to be in control of its parallelization; and they want to be able to tailor codes for general usage as well as for specific inputs. PED is intended for users of this type.

PED is an interactive editor which provides users with all of the information available to automatic tools. In addition, PED understands the dependence graph and parallel constructs, and can provide users with both expert advice and mechanical assistance in making changes and corrections to their programs. PED will also update both the source and dependence graph incrementally after changes. In PED, a group of specific transformations provide the format for modifying programs in a structured manner. When changes takes this structured form, updates are incremental and immediate. PED also supports general arbitrary edits. When program changes are unstructured, source updates are done immediately and dependence analysis of the program is performed upon demand.

3 Program Analysis

3.1 Control and Data Dependences

In sequential languages such as Fortran, the execution order of statements is well defined, making for an excellent program definition on which to build the dependence graph. The statement dependence graph describes a partial order between the statements that must be maintained in order to preserve the semantics of the original sequential program. A dependence between statement S_1 and S_2, denoted $S_1 \delta S_2$, indicates that S_2 depends on S_1, and that the execution of S_1 must precede S_2.

There are two types of dependence, control and data. A *control dependence*, $S_1 \delta_c S_2$, indicates that the execution of S_1 directly determines if S_2 will be executed at all. The following formal definitions of control dependence and the post-dominance relation are taken from the literature [26, 27].

> **Def:** x is *post-dominated* by y in G_f if every path from x to STOP contains y, where STOP is the exit node of G_f.
>
> **Def:** Given two statements $x, y \in G_f$, y is *control dependent* on x if and only if:
>
> 1. $\exists$ a non-null path p, from x to y, such that y post-dominates every node between x and y on p, and
> 2. y does not post-dominate x.

A *data dependence*, $S_1 \delta S_2$, indicates that S_1 and S_2 use or modify a common variable in a way that requires their execution order to be preserved. There are three types of data dependence [39]:

- **True (flow) dependence** occurs when S_1 stores a variable S_2 later uses.
- **Anti dependence** occurs when S_1 uses a variable that S_2 later stores.
- **Output dependence** occurs when S_1 stores a variable that S_2 later stores.

Dependences are also characterized by either being *loop-carried* or *loop-independent* [4, 8]. Consider the following:

```
        DO I = 2, N
S1          A(I) = ...
S2          ...  = A(I)
S3          ...  = A(I-1)
        ENDDO
```

The dependence, $S_1 \delta S_2$, is a loop-independent true dependence, and it exists regardless of the loop constructs surrounding it. Loop-independent dependences, whether data or control, are dependences that occur in a single iteration of the loop and in themselves do not inhibit a loop from running in parallel. For example, if $S_1 \delta S_2$ were the only dependence in the loop, this loop could be run in parallel, because statements executed on each iteration affect only other statements in the same iteration, and not in any other iterations.

In comparison, $S_1 \delta S_3$ is a loop-carried true dependence. Loop-carried dependences are dependences that cross different iterations of some loop, and they constrain the order in which iterations of that loop may execute. For this loop-carried dependence, S_3 uses a value that was created by S_1 on the previous iteration of the `I` loop. This prevents the loop from being run in parallel without explicit synchronization. When there are nested loops, the level of any carried dependence is the outermost loop on which it first arises [4, 8].

3.2 Dependence Analysis

A major strength of PED is its ability to display dependence information and utilize it to guide structured transformations. Precise analysis of both control and data dependences in the program is thus very important. PED's dependence analyzer consists of four major components: the dependence driver, scalar dataflow analysis, symbolic analysis, and dependence testing.

The *dependence driver* coordinates other components of the dependence analyzer by handling queries, transformations, and edits. *Scalar dataflow analysis* constructs the control flow graph and postdominator tree for both structured and unstructured programs. Dominance frontiers are computed for each scalar variable and used to build the *static single assignment* (SSA) graph for each procedure [25]. A coarse dependence graph for arrays is constructed by connecting {Defs} with {Defs $\cup$ Uses} for array variables in each loop nest in the program.

Symbolic analysis determines and compares the values of expressions in programs. When possible, this component eliminates or characterizes symbolic expressions used to determine loop bounds, loop steps, array subscript expressions, array dimensions, and control flow. Its main goal is to improve the precision of dependence testing. The SSA graph provides a framework for performing constant propagation [53], auxiliary induction variable detection, expression folding, and other symbolic analysis techniques.

Detecting data dependences in a program is complicated by array references, since it is difficult to determine whether two array references may ever access the same memory location. A data dependence exists between these references only if the same location may be accessed by both references. *Dependence testing* is the process of discovering and characterizing data dependences between array references. It is a difficult problem which has been the subject of extensive research [8, 12, 13, 56]. Conservative data dependence analysis requires that if a dependence cannot be disproven, it must be assumed to exist. *False dependences* result when conservative dependences do not actually exist. The most important objective of the dependence analyzer is to minimize false dependences through precise analysis.

PED applies a dependence testing algorithm that classifies array references according to their *complexity* (number of loop index variables) and *separability* (no shared index variables). Fast yet exact tests are applied to simple separable subscripts. More powerful but expensive tests are held in reserve for the remaining subscripts. In most cases, results can be merged for an exact test [29].

PED also characterizes all dependences by the flow of values with respect to the enclosing loops. This information is represented as a hybrid distance/direction vector, with one element per enclosing loop. Each element in the vector represents the distance or direction of the flow of values on that loop. The hybrid vector is used to calculate the *level* of all loop-carried dependences generated by an array reference pair. The dependence information is used to refine the coarse dependence graph (constructed during scalar dataflow analysis) into a precise statement dependence graph.

The statement dependence graph contains control and data dependences for the program. Since PED focuses on loop-level parallelism, the dependence graph is designed so that dependences on a particular loop may be collected quickly and efficiently. The dependences may then be displayed to the user, or analyzed to provide expert advice with respect to some transformation. The details of the dependence analysis techniques in PED are described elsewhere [37].

3.3 Interprocedural Analysis

The presence of procedure calls complicates the process of detecting data dependences. Interprocedural analysis is required so that worst case assumptions need not be made when calls are encountered. Interprocedural analysis provided in ParaScope discovers aliasing, side effects such as variable definitions and uses, and interprocedural constants [17, 24]. Unfortunately, improvements to dependence analysis are limited because arrays are treated as monolithic objects, and it is not possible to determine whether two references to an array actually access the same memory location.

To improve the precision of interprocedural analysis, array access patterns can be summarized in terms of *regular sections* or *data access descriptors*. These abstractions describe subsections of arrays such as rows, columns, and rectangles that can be quickly intersected to determine whether dependences exist [11, 19, 31]. By distinguishing the portion of each array affected by a procedure, regular sections provide precise analysis of dependences for loops containing procedure calls.

3.4 Synchronization Analysis

A dependence is *preserved* if synchronization guarantees that the endpoints of the dependence are always executed in the correct order. Sophisticated users may wish to employ *event synchronization* to enforce an execution order when there are loop-carried dependences in a parallel loop. In these cases, it is important to determine if the synchronization preserves all of the dependences in the loop. Otherwise, there may exist race conditions.

Establishing that the order specified by certain dependences will always be maintained has been proven Co-NP-hard. However, efficient techniques have been developed to identify dependences preserved in parallel loops by *post* and *wait* event synchronization [20, 21, 52]. PED utilizes these techniques in a transformation that determines whether a particular dependence is preserved by event synchronization in a loop. Other forms of synchronization are not currently handled in PED. We intend to expand our implementation and include a related technique that automatically inserts synchronization to preserve dependences.

3.5 Implementation Status

Though the implementation of dependence analysis in PED has made much progress, several parts are still under construction. Underlying structures such as the control flow graph, postdominator tree, and SSA graphs have been built, but are not yet fully utilized by the dependence analyzer. PED propagates constants, but does not currently perform other forms of symbolic analysis. Most dependence tests have been implemented, but work remains for the Banerjee-Wolfe and symbolic tests. Interprocedural analysis of aliases, side effects, and constants is performed by the ParaScope environment, but is not integrated with PED's dependence analysis. This integration is underway as part of a larger implementation encompassing both interprocedural symbolic and regular section analysis.

To overcome these gaps in the current implementation of dependence analysis, PED can import on demand dependence information from PFC. When invoked with a special option, PFC utilizes its more mature dependence analyzer to produce a file of dependence information. PED then converts the dependence file into its own internal representation. This process is a temporary expedient which will be unnecessary when dependence analysis in PED is complete.

4 The Dependence Display

This section describes how PED's interface allows users to view and modify the results of program analysis. The persistent view provided by PED appears in Figure 2. The PED window is divided into two panes, the *text pane* and the *dependence pane*. The text pane is on top and consists of two parts; a row of buttons under the title bar, and a large area where Fortran source code is displayed.[2] The buttons in the text pane provide access to functions such as editing, saving, searching, syntax checking, and program transformations.

Directly below the text pane is the dependence pane in which dependences are displayed. It also has two parts: buttons for perusing loops and dependences, and a larger pane for detailed dependence descriptions. The dependence pane shows the results of program analysis to users. Since PED focuses on loop level parallelism, the user first selects a loop for consideration. Regardless of whether the loop is parallel or sequential, the analysis assumes sequential semantics and the dependences for that loop are displayed.

The current loop can be set by using the *next loop* and *previous loop* buttons in the dependence pane, or by using the mouse to select the loop header in the text pane. The loop's header and footer are then displayed in italics for the user. PED will display all the dependences for the current loop, or one or more of the loop-carried, loop-independent, control, or private variable[3] dependences. The type of dependences to be displayed can be selected using the *view* button. The default view displays just the loop-carried dependences, because only they represent race conditions that may lead to errors in a parallel loop.

Although many dependences may be displayed, only one dependence is considered to be the current dependence. The current dependence can be set in the dependence pane by using the *next dependence* and *previous dependence* buttons, or by direct selection with the mouse. For the convenience of the user the current dependence is indicated in both panes. In the dependence pane, the current dependence is underlined; in the text pane, the source reference is underlined and the sink reference is emboldened.

For each dependence the following information is displayed in the dependence pane: the dependence type (control, true, anti, or output), the source and sink variable names involved in the dependence (if any), a hybrid dependence vector containing direction and/or distance information (an exclamation point indicates that this information is exact), the loop level on which the dependence first occurs, and the common block containing the array references. As we will show in the next section, dependences that are of interest can be further classified and organized to assist users in concentrating on some important group of dependences.

4.1 The Dependence Filter Facility

PED has a facility for further filtering classes of dependences out of the display or restricting the display to certain classes. This feature is needed because there are often too many dependences for the user to effectively comprehend. For example, the filtering mechanism permits the user to hide any dependences that have already been examined, or to show only the class of dependences that the

[2] The code displayed is a portion of the subroutine *newque* from the code SIMPLE, a two dimensional Lagrangian hydrodynamics program with heat diffusion, produced by Lawrence Livermore National Laboratory.

[3] Private variables are discussed in Section 4.2.

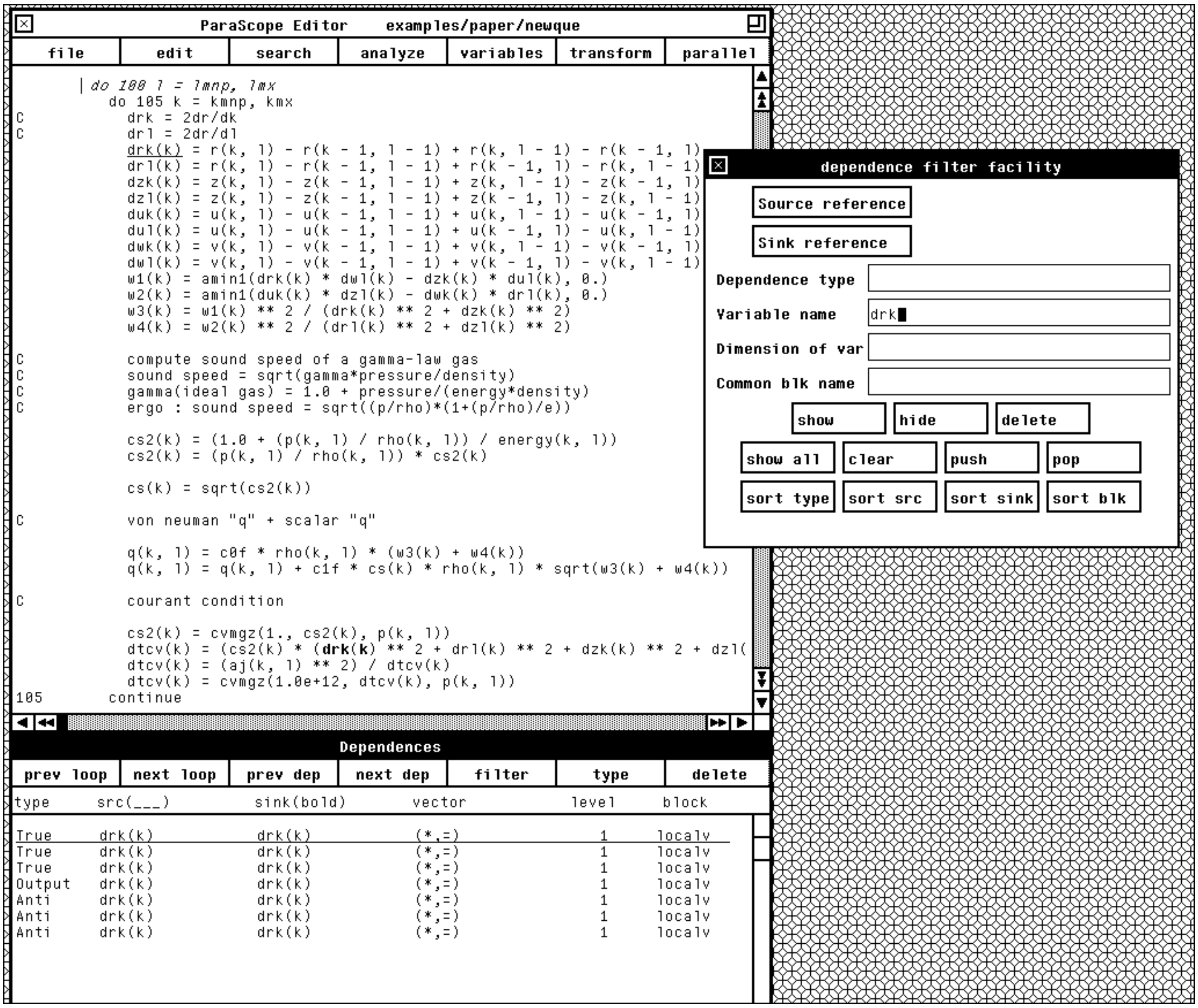

Figure 2 PED Dependence Display and Filter

user wishes to deal with at the moment. When an edge is hidden, it is still in the dependence graph, and all of the transformation algorithms still consider it; it is simply not visible in the dependence display. Users can also delete dependences that they feel are false dependences inserted as the result of imprecise dependence analysis. When an edge is deleted, it is removed from the dependence graph and is no longer considered by the transformation algorithms.

The dependence filter facility is shown in Figure 2. A class of dependences is specified by a *query*. Queries can be used to select sets of dependences according to specific criteria. Valid query criteria are the names of variables involved in dependences, the names of common blocks containing variables of interest, source and sink variable references, dependence type, and the number of array dimensions. All of the queries, except for the source and sink variable references, require the user to type a string into the appropriate query field. The variable reference criteria are set when the user selects the variable reference of interest in the text pane, and then selects the *sink reference* or *source reference* buttons, or both.

Once one or more of the query criteria have been specified, the user can choose to *show* or *hide* the matching dependences. With the *show* option all of the dependences in the current dependence list whose attributes match the query become the new dependence list. In Figure 2, we have selected *show* with the single query criterion: the variable name, `drk`. With the *hide* option all of the dependences in the current list whose characteristics match the query are hidden from the current list, and the remaining dependences become the new list. All of the criteria can be set to empty by using the *clear* button.

The user can also *push* sets of dependences onto a stack. A *push* makes the set of dependences matching the current query become the current database for all subsequent queries. A *pop* returns to the dependence database that was active at the time of the last *push*. Multiple *pushes* and corresponding *pops* are supported. A *show all* presents all the dependences that are part of the current database.

The dependence list can be sorted by source reference, sink reference, dependence type, or common block. Any group of dependences can be selected and deleted from the database by using the *delete* button. Delete is destructive, and a removed dependence will no longer be considered in the transformation algorithms nor appear in the dependence display. In Section 6, we discuss the implications of dependence deletion.

4.2 Variable Classification

One of the most important parts of determining whether a loop can be run in parallel is the identification of variables that can be made private to the loop body. This is important because private variables do not inhibit parallelism. Hence, the more variables that can legally be made private, the more likely it is that the loop may be safely parallelized.

The *variable classification* dialog, illustrated in Figure 3, is used to show the classification of variables referenced in the loop as *shared* or *private*. Initially, this classification is based upon dataflow analysis of the program. Any variable that is:

- defined before the loop and used inside the loop,
- defined inside the loop and used after the loop, or
- defined on one iteration of the loop and used on another

is assumed to be shared. In each of these cases, the variable must be accessible to more than one

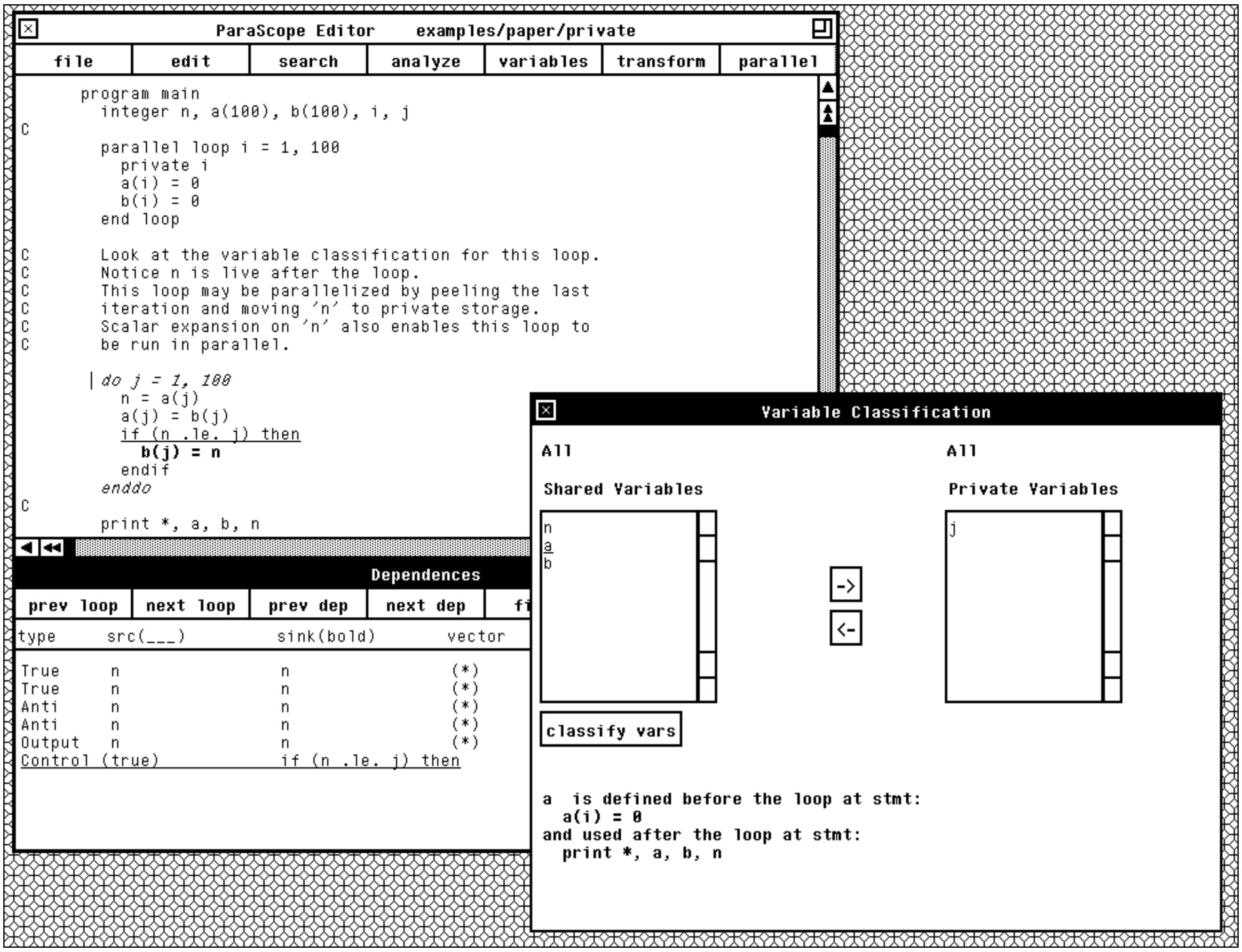

Figure 3 PED Variable Classification

iteration. All other variables are assumed to be private. To be accessible by all iterations, shared variables must be allocated to global storage. Private variables must be allocated for every iteration of a parallel loop, and may be put in a processor's local storage. Notice in Figure 3, the first loop is parallel. The induction variable, `i`, is declared as private, because each iteration needs to have its own copy.

Consider the second loop in Figure 3, but assume `n` is not live after the loop. Then the values of `n` are only needed for a single iteration of the loop. They are not needed by any other iteration, or after the execution of the loop. Each iteration of the loop must have its own copy of `n`, if the results of executing the loop in parallel are to be the same as sequential execution. Otherwise, if `n` were shared, the problem would be that one iteration might use a value of `n` that was stored by some other iteration. This problem inhibits parallelism, if `n` cannot be determined to be private.

In Figure 3, the variable classification dialog displays the shared variables in the left list, and the private variables in the right list for the current loop. If the current loop were transformed into a parallel loop, variables in the shared list would be implicitly specified to be in global storage, and the variables in the private list would be explicitly included in a *private* statement. The user can select variables from either list with the mouse. Once a variable is selected the reason for its classification will be displayed. Notice in Figure 3, the variable `a` is underlined, indicating that it is selected, and the reason it must be in shared-memory is displayed at the bottom of the pane.

Users need not accept the classification provided by PED. They can transfer variables from one list to the other by selecting a variable and then selecting the *arrow* button that points to the other list. When users transfer variables from one list to another, they are making an assertion that overrides the results of dependence analysis, and there is no guarantee that the semantics of the sequential loop will be the same as its parallel counterpart.

Usually programmers will try to move variables from shared to private storage to increase parallelism and to reduce memory contention and latency. To assist users in this task, the *classify vars* button supports further classification of the variable lists. This mechanism helps users to identify shared variables that may be moved to private storage by using transformations, or by correcting conservative dependences.

5 Structured Program Transformations

PED provides a variety of interactive structured transformations that can be applied to programs to enhance or expose parallelism. In PED transformations are applied according to a *power steering* paradigm: the user specifies the transformation to be made, and the system provides advice and carries out the mechanical details. A single transformation may result in many changes to the source, which if done one at a time may leave the intermediate program either semantically incorrect, syntactically incorrect, or both. Power steering avoids incorrect intermediate stages that may result if the user were required to do code restructuring without assistance. Also, because the side effects of a transformation on the dependence graph are known, the graph can be updated directly, avoiding any unnecessary dependence analysis.

In order to provide its users with flexibility, PED differentiates between *safe, unsafe,* and *inapplicable* transformations. An inapplicable transformation cannot be performed because it is not mechanically possible. For example, loop interchange is inapplicable when there is only a single loop.

Transformations are safe when they preserve the sequential semantics of the program. Some transformations always preserve the dependence pattern of the program and therefore can always be safely applied if mechanically possible. Others are only safe when the dependence pattern of the program is of a specific form.

An unsafe transformation does not maintain the original program's semantics, but is mechanically possible. When a transformation is unsafe, users are often given the option to override the system advice and apply it anyway. For example, if a user selects the *parallel* button on a sequential loop with loop-carried dependences, PED reminds the user of the dependences. If the user wishes to ignore them, the loop can still be made parallel. This override ability is extremely important in an interactive tool, where the user is being given an opportunity to apply additional knowledge that is unavailable to the tool.

To perform a transformation, the user selects a sequential loop and chooses a transformation from the menu. Only transformations that are *enabled* may be selected. Transformations are enabled based on the control flow contained in the selected loop. All the transformations are enabled when a loop and any loops nested within it contain no other control flow; most of the transformations are enabled when they contain structured control flow; and only a few are enabled when there is arbitrary control flow.

Once a transformation is selected, PED responds with a diagnostic. If the transformation is safe, a profitability estimate is given on the effectiveness of the transformation. Additional advice, such as a suggested number of iterations to skew, may be offered as well. If the transformation is unsafe, a warning explains what makes the transformation unsafe. If the transformation is inapplicable, a diagnostic describes why the transformation cannot be performed. If the transformation is applicable, and the user decides to execute it, the user selects the *do <transformation name>* button. The Fortran source and the dependence graph are then automatically updated to reflect the transformed code.

The transformations are divided into four categories: reordering transformations, dependence breaking transformations, memory optimizing transformations, and a few miscellaneous transformations. Each category and the transformations that PED currently supports are briefly described below.[4]

5.1 Reordering Transformations

Reordering transformations change the order in which statements are executed, either within or across loop iterations, without violating any dependence relationships. These transformations are used to expose or enhance loop level parallelism in the program. They are often performed in concert with other transformations to structure computations in a way that allows useful parallelism to be introduced.

- **Loop distribution** partitions independent statements inside a loop into multiple loops with identical headers. It is used to separate statements which may be parallelized from those that must be executed sequentially [36, 37, 39]. The partitioning of the statements is tuned for vector or parallel hardware as specified by the user.

[4] The details of PED's implementation of several of these transformations appear in [37].

- **Loop interchange** interchanges the headers of two perfectly nested loops, changing the order in which the iteration space is traversed. When loop interchange is safe, it can be used to adjust the granularity of parallel loops [8, 37, 56].
- **Loop skewing** adjusts the iteration space of two perfectly nested loops by shifting the work per iteration in order to expose parallelism. When possible, PED computes and suggests the optimal skew degree. Loop skewing may be used with loop interchange in PED to perform the wavefront method [37, 54].
- **Loop reversal** reverses the order of execution of loop iterations.
- **Loop adjusting** adjusts the upper and lower bounds of a loop by a constant. It is used in preparation for loop fusion.
- **Loop fusion** can increase the granularity of parallel regions by fusing two contiguous loops when dependences are not violated [3, 42].
- **Statement interchange** interchanges two adjacent independent statements.

5.2 Dependence Breaking Transformations

The following transformations can be used to break specific dependences that inhibit parallelism. Often if a particular dependence can be eliminated, the safe application of other transformations is enabled. Of course, if all the dependences carried on a loop are eliminated, the loop may then be run in parallel.

- **Scalar expansion** makes a scalar variable into a one-dimensional array. It breaks output and anti dependences which may be inhibiting parallelism [40].
- **Array renaming**, also known as node splitting [40], is used to break anti dependences by copying the source of an anti dependence into a newly introduced temporary array and renaming the sink to the new array [8]. Loop distribution may then be used to separate the copying statement into a separate loop, allowing both loops to be parallelized.
- **Loop peeling** peels off the first or last k iterations of a loop as specified by the user. It is useful for breaking dependences which arise on the first or last k iterations of the loop [3].
- **Loop splitting**, or index set splitting, separates the iteration space of one loop into two loops, where the user specifies at which iteration to split. For example, if `do i = 1, 100` is split at 50, the following two loops result: `do i = 1, 50` and `do i = 51, 100`. Loop splitting is useful in breaking *crossing dependences*, dependences that cross a specific iteration [8].

5.3 Memory Optimizing Transformations

The following transformations adjust a program's balance between computations and memory accesses to make better use of the memory hierarchy and functional pipelines. These transformations are useful for scalar and parallel machines.

- **Strip mining** takes a loop with step size of 1, and changes the step size to a new user specified step size greater than 1. A new inner loop is inserted which iterates over the new step size. If the minimum distance of the dependences in the loop is less than the step size, the resultant inner loop may be parallelized. Used alone the order of the iterations is unchanged, but used in concert with loop interchange the iteration space may be *tiled* [55] to utilize memory bandwidth and cache more effectively [23].

- **Scalar replacement** takes array references with consistent dependences and replaces them with scalar temporaries that may be allocated into registers [14]. It improves the performance of the program by reducing the number of memory accesses required.

- **Unrolling** decreases loop overhead and increases potential candidates for *scalar replacement* by unrolling the body of a loop [3, 37].

- **Unroll and Jam** increases the potential candidates for *scalar replacement* and pipelining by unrolling the body of an outer loop in a loop nest and fusing the resulting inner loops [14, 15, 37].

5.4 Miscellaneous Transformations

Finally PED has a few miscellaneous transformations.

- **Sequential $\leftrightarrow$ Parallel** converts a sequential DO loop into a parallel loop, and vice versa.
- **Statement addition** adds an assignment statement.
- **Statement deletion** deletes an assignment statement.
- **Preserved dependence?** indicates whether the current selected dependence is preserved by any post and wait event synchronization in the loop.
- **Constant replacement** performs global constant propagation for each procedure in the program, using the *sparse conditional constant* algorithm [53]. Any variable found to have a constant value is replaced with that value, increasing the precision of subsequent dependence analysis.

5.5 Example

The following example is intended to give the reader the flavor of this type of transformational system. Consider the first group of nested loops in Figure 4. Let S_1 be the first assignment statement involving the array `D`, and S_2 be the second assignment statement involving the array `E`. There are two loop-carried dependences, $S_1 \delta S_1$ and $S_2 \delta S_2$. The first is a true dependence on `D` carried by the `I` loop in the first subscript position, and the second is a true dependence on `E` carried by the `J` loop in the second subscript position. We notice that both loops are inhibited from running in parallel by different dependences which are not involved with each other. To separate these independent statements, we consider distributing the loops. Distribution on the inner loop results in the message shown in Figure 4. The message indicates what the results of performing distribution on this loop would be. The execution of distribution results in the following code.

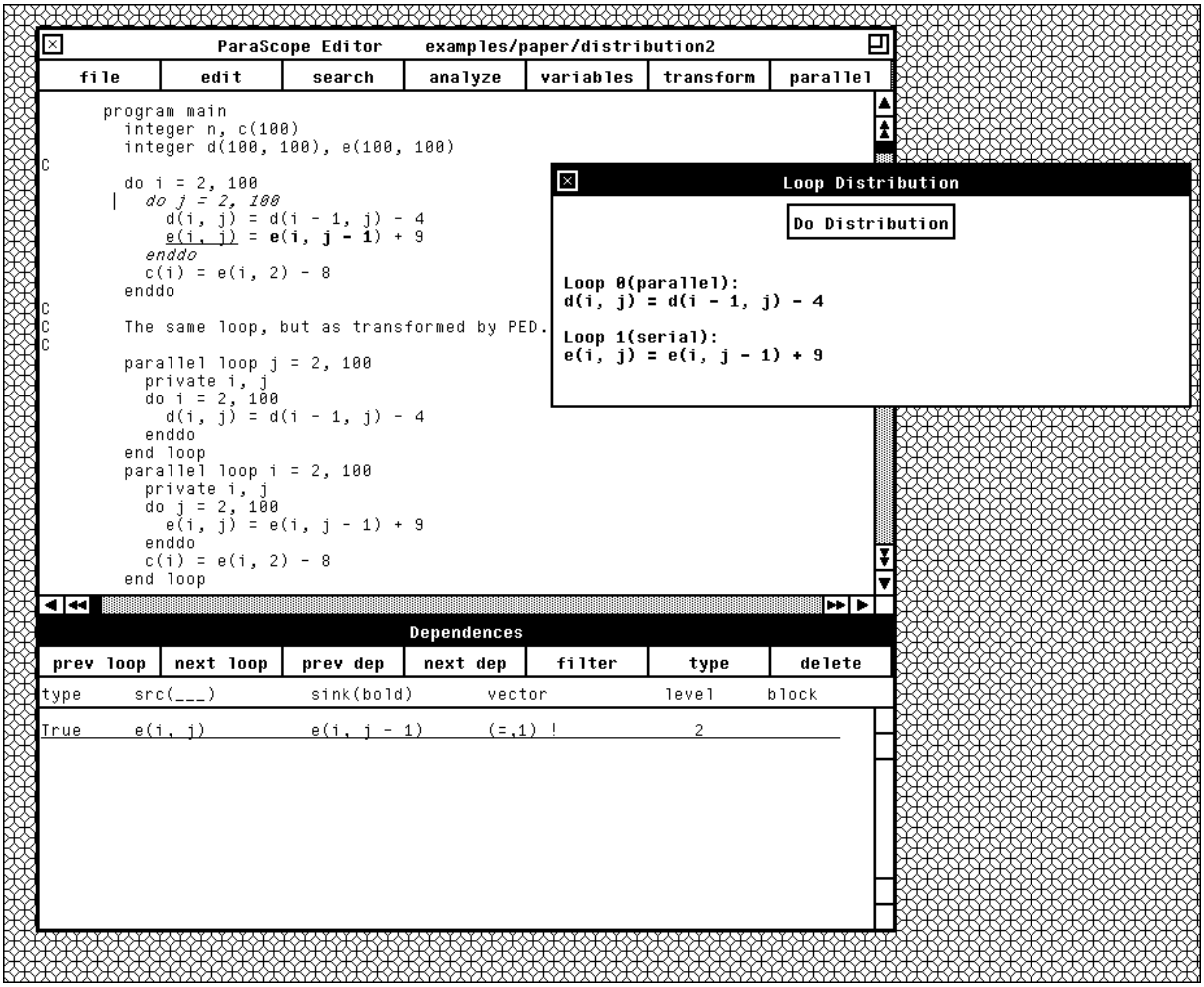

Figure 4 Loop Distribution

```
            DO I = 2, 100
               DO J = 2, 100
S1                D(I, J) = D(I - 1, J) - 4
               ENDDO
               DO J = 2, 100
S2                E(I, J) = E(I, J - 1) + 9
               ENDDO
S3             C(I) = E(I, 2) - 8
            ENDDO
```

Unfortunately, the dependence on S_1 carried by the I loop still inhibits the I loop parallelism for S_2. We perform distribution once again, this time on the outer loop.

```
            DO I = 2, 100
               DO J = 2, 100
S1                D(I, J) = D(I - 1, J) - 4
               ENDDO
            ENDDO
            DO I = 2, 100
               DO J = 2, 100
S2                E(I, J) = E(I, J - 1) + 9
               ENDDO
S3             C(I) = E(I, 2) - 8
            ENDDO
```

We have decided to distribute for parallelism in this example. So even though S_2 and S_3 are independent, the algorithm leaves them together to increase the amount of computation in the parallel loop. If we had selected vectorization they would have been placed in separate loops.

Continuing our example, notice the second I loop can now be run in parallel, and the inner J loop in the first nest can be run in parallel. To achieve a higher granularity of parallelism on the first loop nest, the user can interchange the loops, safely moving the parallelism to the outer loop. As can be seen in the second loop nest of Figure 4, we have safely separated the two independent statements and their dependences, achieving two parallel loops.

6 Relating User Changes to Analysis

In previous sections we discussed how users may direct the parallelization process by making assertions about dependences and variables, as well as by applying structured transformations. This section first briefly describes editing in PED. Then, the interaction between program changes and analysis is examined.

Editing is fundamental for any program development tool because it is the most flexible means of making program changes. Therefore, the ParaScope Editor integrates advanced editing features along with its other capabilities. PED supplies simple text entry and template-based editing with its underlying hybrid text and structure editor. It also provides search and replace functions, intelligent and customizable view filters, and automatic syntax and type checking.

Unlike transformations or assertions, editing causes existing dependence information to be unreliable. As a result, the transformations and the dependence display are disabled during editing because they rely on dependence information which may be out of date. After users finish editing, they can request the program be reanalyzed by selecting the *analysis* button. Syntax and type checking are performed first, and any errors are reported. If there are no errors, dependence analysis is performed. PED's analysis may be incremental when the scope of an edit is contained within a loop nest or is an insertion or deletion of a simple assignment statement. The details of incremental analysis after edits and transformations are discussed elsewhere [37].

The purpose of an edit may be error correction, new code development, or just to rearrange existing code. Unlike with transformations, where the correctness of pre-existing source is assumed, PED does not know the intent of an edit. Consequently, the user is not advised as to the correctness of the edit. Instead, the "new" program becomes the basis for dependence analysis and any subsequent changes. No editing history is maintained. Similarly, any transformations the user performs before an edit, whether safe or unsafe, are included in the new basis. However, if prior to editing the user made any assertions, analysis causes them to be lost.

For example, suppose the user knows the value of a symbolic. Based on this knowledge, the user deletes several overly conservative dependences in a loop and transforms it into a parallel loop. Later, the user discovers an error somewhere else in the program and corrects it with a substantial edit. The user then reanalyzes the program. In the current implementation, the parallel loop will remain parallel, but any deleted dependences will reappear and, as experience has shown, annoy users.

As a result, a more sophisticated mechanism is planned [28]. In the future, edges that are deleted by users will be marked, rather than removed from the dependence graph. Additionally, the time, date, user, and an optional user-supplied explanation will be recorded with any assertions. This mechanism will also support more general types of assertions, such as variable ranges and values which may affect many dependence edges. These records will be used during analysis to keep deleted dependences from reappearing. However, to prevent errors when edits conflict with assertions, users will be given an opportunity to reconsider any assertions which may have been affected by the edit. Users may delay or ignore this opportunity. With this mechanism, the assertions will also be available during execution and debugging. There, if an assertion is found to be erroneous, users can be presented with any anomalies which may have been ignored, overlooked, or introduced.

7 Related Work

Several other research groups are also developing advanced interactive parallel programming tools. PED is distinguished by its large collection of transformations, the expert guidance provided for each transformation, and the quality of its program analysis and user interface. Below we briefly describe SIGMACS [48], PAT [50],MIMDizer [46], and SUPERB [57], placing emphasis on their unique features.

SIGMACS is an interactive emacs-based programmable parallelizer in the FAUST programming environment. It utilizes dependence information fetched from a project database maintained by the *database server*. SIGMACS displays dependences and provides some interactive program transformations. Work is in progress to support automatic updating of dependence information after statement insertion and deletion. FAUST can compute and display call and process graphs that may be animated dynamically at run-time [30]. Each node in a process graph represents a task or a process, which is a separate entity running in parallel. FAUST also provides performance analysis and prediction tools for parallel programs.

PAT can analyze programs containing general parallel constructs. It builds and displays a statement dependence graph over the entire program. In PAT the program text that corresponds with a selected portion of the graph can be perused. The user may also view the list of dependences for a given loop. However, PAT can only analyze programs where only one write occurs to each variable in a loop. Like PED, incremental dependence analysis is used to update the dependence graph after structured transformations [51]. Rather than analyzing the effects of existing synchronization, PAT can instead insert synchronization to preserve specific dependences. Since PAT does not compute distance or direction vectors, loop reordering transformations such as loop interchange and skewing are not supported.

MIMDizer is an interactive parallelization system for both shared and distributed-memory machines. Based on FORGE,MIMDizer performs dataflow and dependence analysis to support interactive loop transformations. Cray microtasking directives may be output for successfully parallelized loops. Associated tools graphically display control flow, dependence, profiling, and call graph information. A history of the transformations performed on a program is saved for the user. MIMDizer can also generate communication for programs to be executed on distributed-memory machines.

Though designed to support parallelization for distributed-memory multiprocessors, SUPERB provides dependence analysis and display capabilities similar to that of PED. SUPERB also possesses a set of interactive program transformations designed to exploit data parallelism for distributed-memory machines. Algorithms are described for the incremental update of use-def and def-use chains following structured program transformations [38].

8 Conclusions

Programming for explicitly parallel machines is much more difficult than sequential programming. If we are to encourage scientists to use these machines, we will need to provide new tools that have a level of sophistication commensurate with the difficulty of the task. We believe that the ParaScope Editor is such a tool: it permits the user to develop programs with the full knowledge of the data relationships in the program; it answers complex questions about potential sources of error; and it correctly carries out complicated transformations to enhance parallelism.

PED is an improvement over completely automatic systems because it overcomes both the imprecision of dependence analysis and the inflexibility of automatic parallel code generation techniques by permitting the user to control the parallelization process. It is an improvement over dependence browsers because it supports incremental change while the user is reviewing potential problems with the proposed parallelization. PED has also proven to be a useful basis for the development of several other advanced tools, including a compiler [33] and data decomposition tool [9, 10] for distributed-memory machines, as well an on-the-fly access anomaly detection system for shared-memory machines [34].

We believe that PED is representative of a new generation of intelligent, interactive programming tools that are needed to facilitate the task of parallel programming.

Acknowledgments

We would like to thank Donald Baker, Vasanth Balasundaram, Paul Havlak, Marina Kalem, Ulrich Kremer, Rhonda Reese, Jaspal Subhlok, Scott Warren, and the PFC research group for their many contributions to this work. Their efforts have made PED the useful research tool it is today. In addition, we gratefully acknowledge the contribution of the $\mathbb{R}^n$ and ParaScope research groups, who have provided the software infrastructure upon which PED is built.

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] F. Allen, M. Burke, P. Charles, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for detecting useful parallelism. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[3] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[4] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*. PhD thesis, Rice University, April 1983.

[5] 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.

[6] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[7] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.

[8] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[9] 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.

[10] 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.

[11] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[12] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

[13] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[14] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[15] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[16] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[17] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[18] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of Supercomputing '88*, Orlando, FL, November 1988.

[19] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[20] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[21] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WA, May 1988.

[22] A. Carle, K. Cooper, R. Hood, L. Torczon K. Kennedy, and S. Warren. A practical environment for scientific programming. *Computer*, 20(11):75–89, November 1987.

[23] S. Carr and K. Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.

[24] 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.

[25] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.

[26] R. Cytron, J. Ferrante, and V. Sarkar. Experience using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[27] 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.

[28] 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.

[29] 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.

[30] V. Guarna, D. Gannon, D. Jablonowski, A. Malony, and Y. Gaur. Faust: An integrated environment for parallel programming. *IEEE Software*, 6(4):20–27, July 1989.

[31] P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[32] 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.

[33] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, to appear 1991.

[34] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[35] IBM. *Parallel Fortran Language and Library Reference*, first edition, February 1988. Document Number SC23-0431-0.

[36] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[37] 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.

[38] U. Kremer, H. Zima, H.-J. Bast, and M. Gerndt. Advanced tools and techniques for automatic parallelization. *Parallel Computing*, 7:387–393, 1988.

[39] D. Kuck. *The Structure of Computers and Computations, Volume 1*. John Wiley and Sons, New York, NY, 1978.

[40] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. Analysis and transformation of programs for parallel computation. In *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference*, pages 709–715, Chicago, IL, October 1980.

[41] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. The structure of an advanced retargetable vectorizer. In *Supercomputers: Design and Applications*, pages 163–178. IEEE Computer Society Press, Silver Spring, MD, 1984.

[42] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[43] B. Leasure, editor. *PCF Fortran: Language Definition, version 3.1*. The Parallel Computing Forum, Champaign, IL, August 1990.

[44] H. Levesque and J. Williamson. *A Guidebook to Fortran on Supercomputers*. Harcourt Brace Jovanovich, San Diego, CA, 1989.

[45] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems*. Sequent Technical Publications, San Diego, CA, 1989.

[46] R. Sawdayi, G. Wagenbreth, and J. Williamson. MIMDizer: Functional and data decomposition; creating parallel programs from scratch, transforming existing Fortran programs to parallel. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, to appear 1991.

[47] R.G. Scarborough and H.G. Kolsky. A vectorizing Fortran compiler. *IBM Journal of Research and Development*, 30(2):163–171, March 1986.

[48] B. Shei and D. Gannon. SIGMACS: A programmable programming environment. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[49] 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.

[50] K. Smith and W. Appelbe. PAT - an interactive Fortran parallelizing assistant tool. In *Proceedings of the 1988 International Conference on Parallel Processing*, St. Charles, IL, August 1988.

[51] K. Smith, W. Appelbe, and K. Stirewalt. Incremental dependence analysis for interactive parallelization. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[52] J. Subhlok. *Analysis of Synchronization in a Parallel Programming Environment*. PhD thesis, Rice University, August 1990.

[53] M. Wegman and K. Zadeck. Constant propagation with conditional branches. Technical Report CS-89-36, Dept. of Computer Science, Brown University, May 1989.

[54] M. J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, August 1986.

[55] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[56] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

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