Analysis and Transformation in the ParaScope Editor

Ken Kennedy
Kathryn S. McKinley
Chau-Wen Tseng

CRPC-TR 90106
December 1990

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

In *Proceedings of the 1991 International Conference, on Supercomputing*, Cologne, Germany, June 1991.

Analysis and Transformation in the ParaScope Editor *

Ken Kennedy Kathryn S. McKinley Chau-Wen Tseng

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

Abstract

The ParaScope Editor is a new kind of interactive parallel programming tool for developing scientific Fortran programs. It assists the knowledgeable user by displaying the results of sophisticated program analyses and by providing editing and a set of powerful interactive transformations. After an edit or parallelism-enhancing transformation, the ParaScope Editor incrementally updates both the analyses and source quickly. This paper describes the underlying implementation of the ParaScope Editor, paying particular attention to the analysis and representation of dependence information and its reconstruction after changes to the program.

1 Introduction

The ParaScope Editor is a tool designed to help skilled users interactively transform a sequential Fortran 77 program into a parallel program with explicit parallel constructs, such as those in PCF Fortran [40]. In a language like PCF Fortran, the principal mechanism for the introduction of parallelism is the *parallel loop*, which specifies that its iterations may be run in parallel according to any schedule. The fundamental problem introduced by such languages is the possibility of non-deterministic execution. This can happen if two different iterations of a parallel loop both reference the same memory location, where at least one of the references writes to the location. For example, consider converting the following sequential loop into a parallel loop.

```
DO i = 1, 100
  A(i) = A(50) + 1
ENDDO
```

Because the parallel loop does not order its iterations, the value of each `A(i)` will depend on how early iteration 50 executes in the parallel execution schedule. Hence, the results may differ each time the program is executed. This kind of anomaly, usually called a *data race*, precludes the parallelization of the above loop. In the literature of compilation for parallel execution, a potential data race is referred to as a *loop-carried dependence* [4, 35]. Without explicit synchronization, only loops with no carried dependences may be safely run in parallel.

Automatic parallelizers use this principle by constructing a *dependence graph* for the entire program and then parallelizing every loop that does not carry a dependence. Unfortunately, a parallelizer is often forced to make conservative assumptions about whether dependences exist because of complex subscripts or the use of unknown symbolics. As a result, automatic systems miss many loops that could be parallelized. This weakness has led most researchers to conclude that automatic systems, by themselves, are not powerful enough to find all of the parallelism in a program.

However, the analysis performed by automatic systems can be extremely useful to the programmer during the parallelization process. The ParaScope Editor (PED) is based upon this observation. It is designed to support an interactive parallelization process in which the user examines a particular loop and its dependences. To safely parallelize a loop, the user must either determine that each dependence shown is not valid (because of some conservative assumption made by the system), or transform the loop to eliminate valid dependences. After each transformation, PED reconstructs the dependence graph so that the user may determine the level of success achieved and apply additional transformations if desired.

Clearly a tool with this much functionality is bound to be complex. PED incorporates a complete source editor and supports dependence analysis, dependence display, and a large variety of program transformations to enhance parallelism. Previous work has described the usage and motivation of the ParaScope Editor [8, 23, 33] and the ParaScope parallel programming environment [14]. In this paper, we focus on the implementation of PED's analysis and transformation features. Particular attention is paid to the representation of dependences, the construction of the dependence graph, and how dependences are used and incrementally reconstructed for

*This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center, by IBM Corporation, and the Cray Research Foundation.

each program transformation in an efficient and flexible manner.

We begin in Section 2 with a brief overview of dependence. Section 3 describes our work model. Section 4 presents PED's internal representations, and Section 5 examines the analysis strategy and algorithms in PED. Sections 6 and 7 explain PED's support for transformations and details an interesting subset. Section 8 summarizes PED's editing capabilities. We discuss related work in Section 9 and conclude.

2 Dependence

At the core of PED is its ability to analyze *dependences* in a program. Dependences describe a partial order between statements that must be maintained to preserve the meaning of the original sequential program. A dependence between statement S_1 and S_2, denoted $S_1 \delta S_2$, indicates that S_1, the *source*, must be executed before S_2, the *sink*. There are two types of dependence, data and control dependence, which are described below.

2.1 Data Dependence

A *data dependence*, $S_1 \delta S_2$, indicates that S_1 and S_2 read or write a common memory location in a way that requires their execution order to be preserved. There are four types of data dependence [35]:

True (flow) dependence occurs when S_1 writes a memory location that S_2 later reads.

Anti dependence occurs when S_1 reads a memory location that S_2 later writes.

Output dependence occurs when S_1 writes a memory location that S_2 later writes.

Input dependence occurs when S_1 reads a memory location that S_2 later reads.[1]

2.2 Control Dependence

Intuitively, a *control dependence*, $S_1 \delta_c S_2$, indicates that the execution of S_1 directly determines whether S_2 will be executed. The following formal definitions of control dependence and the postdominance relation are taken from the literature [22].

Def: x is *postdominated* by y in G_f, the control flow graph, 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 postdominates every node between x and y on p, and
2. y does not postdominate x.

[1] Input dependences do not restrict statement order.

2.3 Loop-Carried and Loop-Independent Dependence

Dependences are also characterized as either being loop-carried or loop-independent [3, 4]. Consider the following loop:

```
      DO i = 2, n
S1      A(i) = ...
S2      ... = A(i)
S3      ... = A(i-1)
      ENDDO
```

The true dependence $S_1 \delta S_2$ is *loop-independent* because it exists regardless of the surrounding loops. Loop-independent dependences, whether data or control, occur within a single iteration of the loop and 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 only affect other statements in the same iteration and not in any other iterations. However, loop-independent dependences do affect statement order within a loop iteration. Interchanging statements S_1 and S_2 violates the loop-independent dependence and changes the meaning of the program.

By comparison, the true dependence $S_1 \delta S_3$ is *loop-carried* because the source and sink of the dependence occur on different iterations of the loop: S_3 reads a memory location that was written to by S_1 on the previous iteration. Loop-carried dependences are important because they inhibit loops from executing in parallel without synchronization. When there are nested loops, the *level* of any carried dependence is the outermost loop on which it first arises [3, 4].

2.4 Dependence Testing

Determining the existence of data dependence between array references is more difficult than for scalars, because the subscript expressions must be considered. The process of differentiating between two subscripted references in a loop nest is called *dependence testing*. To illustrate, consider the problem of determining whether or not there exists a dependence from statement S_1 to S_2 in the following loop nest:

```
      DO i1 = L1, U1
         DO i2 = L2, U2
           ...
             DO in = Ln, Un
S1              A(f1(i1, ..., in), ..., fm(i1, ..., in)) = ...
S2              ... = A(g1(i1, ..., in), ..., gm(i1, ..., in))
           ...
      ENDDO
```

Let α and β be vectors of n integer indices within the ranges of the upper and lower bounds of the n loops. There is a dependence from S_1 to S_2 if and only if there exist α and β such that α is lexicographically less than or equal to β and the following system of *dependence equations* is satisfied:

$$f_k(\alpha) \;=\; g_k(\beta) \;\; \forall k,\; 1 \le k \le m$$

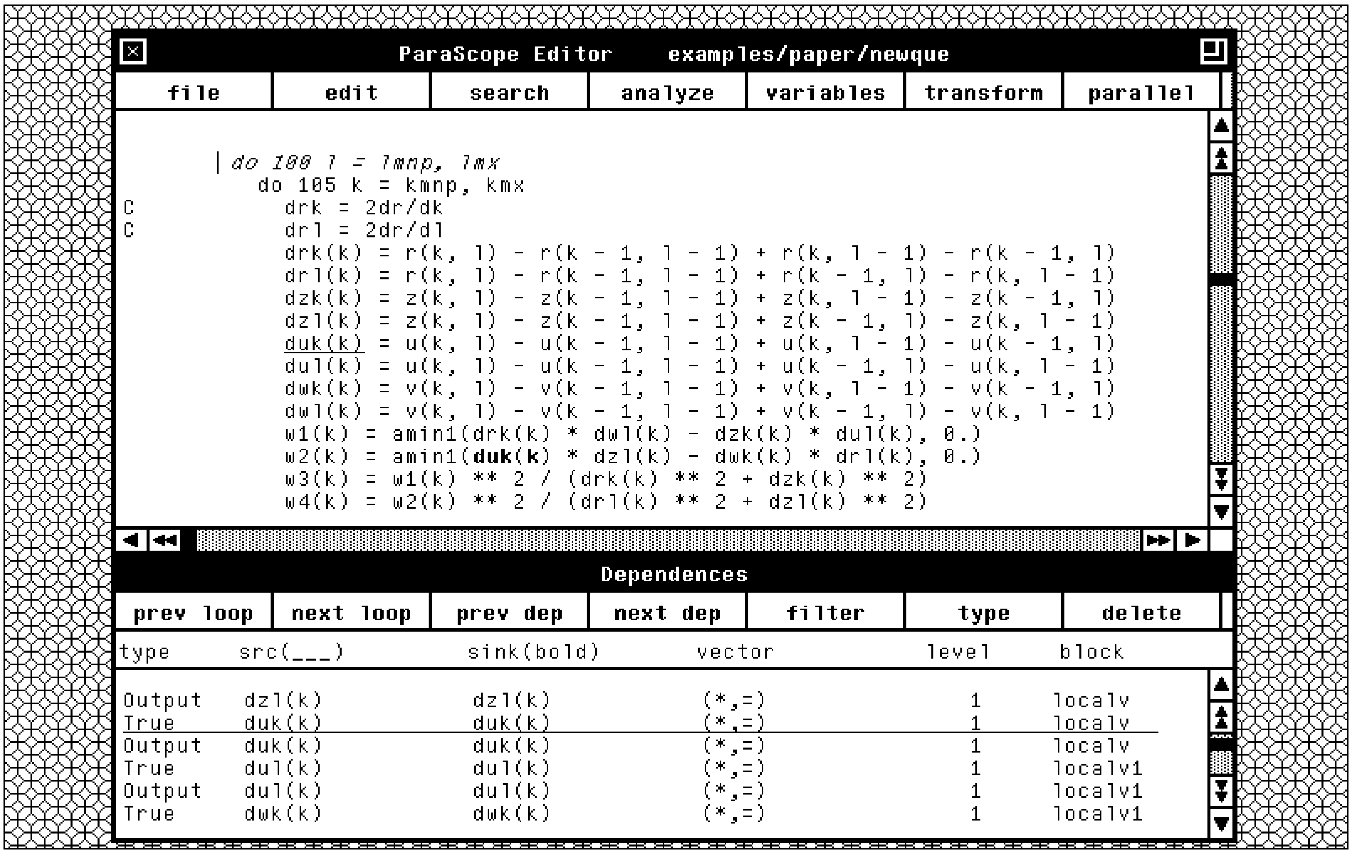

Figure 1: PED User Interface

2.5 Distance and Direction Vectors

Distance and direction vectors may be used to characterize data dependences by their access pattern between loop iterations. If there exists a data dependence for $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n)$, then the *distance vector* $\mathbf{D} = (D_1, \ldots, D_n)$ is defined as $\beta - \alpha$. The *direction vector* $\mathbf{d} = (d_1, \ldots, d_n)$ of the dependence is defined by the equation:

$$d_i = \begin{cases} < & \text{if } \alpha_i < \beta_i \\ = & \text{if } \alpha_i = \beta_i \\ > & \text{if } \alpha_i > \beta_i \end{cases}$$

The elements are always displayed left to right, from the outermost to the innermost loop in the nest. For example, consider the following loop nest:

```
DO i = 1, n
   DO j = 1, m
      DO k = 1, l
         A(i+1, j, k-1) = A(i, j, k) + C
      ENDDO
   ENDDO
ENDDO
```

The distance and direction vectors for the true dependence between the definition and use of array `A` are $(1, 0, -1)$ and $(<, =, >)$, respectively. Since several different values of α and β may satisfy the dependence equations, a set of distance and direction vectors may be needed to completely describe the dependences arising between a pair of array references.

Direction vectors, introduced by Wolfe [55], are useful for calculating loop-carried dependences. A dependence is carried by the outermost loop for which the element in the direction vector is not an '='. Additionally, direction vectors are used to determine the safety and profitability of loop interchange [4, 55]. Distance vectors, first used by Kuck and Muraoka [38, 43], are more precise versions of direction vectors that specify the actual number of loop iterations between two accesses to the same memory location. They are utilized by transformations to exploit parallelism [9, 39, 54, 56] and the memory hierarchy [12, 24].

3 Work Model

PED is designed to exploit *loop-level* parallelism, which comprises most of the usable parallelism in scientific codes when synchronization costs are considered [18]. In the work model best supported by PED, the user first selects a loop for parallelization. PED then displays all of its carried dependences. The user may sort or filter the dependences to help discover and delete dependences that are due to overly conservative dependence analysis. PED also provides a set of intelligent program transformations that can be used to eliminate dependences.

PED's user interface is shown in Figure 1. The *program pane* in the top half of the window displays a loop from the subroutine *newque* in SIMPLE; a two dimensional Lagrangian hydrodynamics program with heat diffusion produced by Lawrence Livermore National Laboratory. The outer loop is selected, which causes its header to be italicized. The *dependence pane* at the bottom of the window shows dependences carried by the selected loop. The true dependence on variable `duk` is selected, which causes it to be underlined in the dependence pane. The dependence is also reflected in the program pane by underlining its source and emboldening its sink. Buttons across the top of each pane invoke various PED features, such as transformations and program analysis.

4 Internal Representations

This section describes the three major internal representations used by PED: the abstract syntax tree and its associated structures, the dependence graph, and loop information.

4.1 Abstract Syntax Tree

In PED and throughout the ParaScope programming environment, the program is represented using an *abstract syntax tree* (AST). Because the AST is a public structure, PED does not change the structure of an AST node nor use any fields within the AST. However, PED obviously needs to associate information such as data and control dependences with elements of the AST. This is done by attaching to the AST array an array of the same length, called a *side array*.

PED uses the side array to hold pointers into a variable sized *info array*. Elements of the info array hold pointers to information PED associates with AST nodes such as level vectors, reference lists, loop information, and subscript text. They also point to shadow expressions containing the results of symbolic analysis; they are discussed in Section 5.4. These structures are described in full below and illustrated for a sample program in Figure 2.

4.2 Dependence Graph

PED uses a statement dependence graph to represent control and data dependences in the program. The dependence graph is made up of *dependence edges*, which are connected using *level vectors* and *reference lists*. The level vectors provide the means for quickly collecting dependence edges on a statement pertaining to a specific loop level, and the reference lists provide the means for finding all the dependences associated with a particular variable reference.

4.2.1 Dependence Edges

Each data and control dependence in the program is represented as an explicit edge in the dependence graph. Dependences between pairs of variable references are *reference level* dependences. *Statement level* dependences arise due to input and output statements, branches out of loops, unanalyzed subroutine calls, and control dependences. An edge in the dependence graph is a data structure that describes the following significant features of the dependence.

- Type of the dependence: true, anti, output, input, control, i/o, exit, or call
- Level of the loop carrying the dependence
- AST indices for source/sink statements or references
- Pointers to the source/sink level vectors and reference lists
- Pointer to a hybrid direction/distance vector
- *Interchange preventing* and *interchange sensitive* flags indicating the safety and profitability of loop interchange

PED uses a hybrid direction/distance vector to store the results of dependence testing. Each element of the vector can represent a dependence distance or direction. Dependence edges are organized for the user interface using a higher level data abstraction, called the *edge list*. The edge list provides the user a configurable method of filtering, sorting, and selecting dependences [8, 33].

4.2.2 Level Vectors

Dependence edges hold most of the dependence information for a program, but level vectors provide the glue which links them together and to the AST. Every executable statement in a loop nest involved with a dependence has a level vector. A level vector contains entries for each loop nesting level in which the statement is contained. Every entry points to the list of dependences carried at its level. Level vectors therefore provide quick and efficient access to all of the dependences at a particular loop level for each statement. They are used extensively by PED's loop based transformations and user interface.

Because of their high frequency and simplicity, PED treats dependence edges resulting from scalar variables differently. Dependences between a pair of scalar variable references occur for all commonly nested loops. To avoid duplicate edges, only a single edge is stored along with its deepest nesting level. These scalar edges have a separate entry in the level vector. Queries for dependences carried on level k thus cause two entries in the level vector to be searched—the entry at level k and the entry for scalar dependences.

4.2.3 Reference Lists

Every variable reference in a loop nest involved in a dependence has a reference list. A reference list points to all dependences that have a given reference as the source or sink. They are useful for transformations such as scalar expansion and array renaming, as well as for reference based user queries.

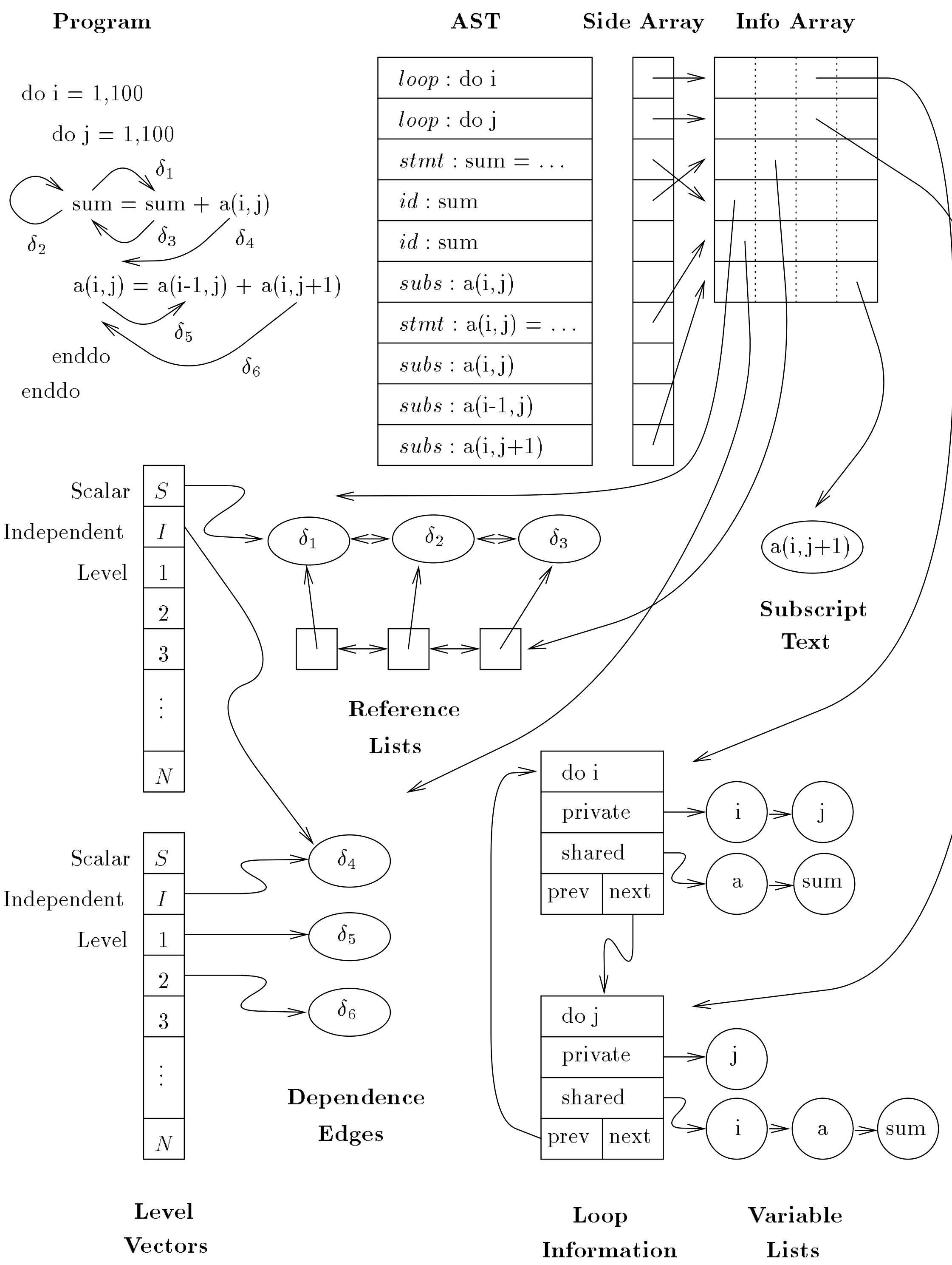

Figure 2: Ped Internal Representations

4.3 Loop Information and Variable Lists

Every loop in the program has a structure known as *loop info* that contains the following information.

- AST index for the loop header
- Level (depth) of the loop in a nest
- Pointers to the loop info for the previous and next loops
- Pointers to the shared and private variable lists
- Flag for parallelized loops

The shared and private variable lists record the status of all variables accessed in the loop body. Private variables are defined and used only within an iteration of the loop. All other variables are shared.

5 Program Analysis

A major strength of PED is its ability to analyze dependences. PED's dependence analyzer faces three major challenges:

Precision Conservative data dependence analysis requires that if a dependence cannot be disproven, it must be assumed to exist. But if these dependences do not actually exist, PED may be inhibited from exploiting the parallelism available in a program. The most important objective of the dependence analyzer is to minimize these *false dependences* through precise analysis.

Efficiency In order for PED to be truly useful, the dependence analyzer must provide precise results in a reasonable amount of time. PED's dependence analyzer gains efficiency by utilizing fast algorithms for analyzing common simple cases, holding more powerful but expensive algorithms in reserve.

Incrementality Historically, dependence analysis has been performed in batch mode as a single pass through the entire program. In an interactive tool or parallelizing compiler, batch reanalysis after each program transformation has proven to be unacceptably slow. PED's dependence analyzer supports *incremental dependence analysis*, a technique to limit reanalysis which results in significantly faster updates after program changes. In general, incremental algorithms must be both quick and precise. Otherwise, users will prefer batch analysis. PED achieves efficiency by applying a tiered set of incremental responses based on the scope of the program change. PED's incremental analysis strategy and algorithms are described in Sections 6.1 and 8.1.

5.1 Dependence Analyzer Overview

The architecture of PED's *dependence analyzer* is shown in Figure 3. It consists of four major components: the dependence driver, scalar dataflow analysis, symbolic analysis, interprocedural analysis, and dependence testing. The following sections examine each in more detail.

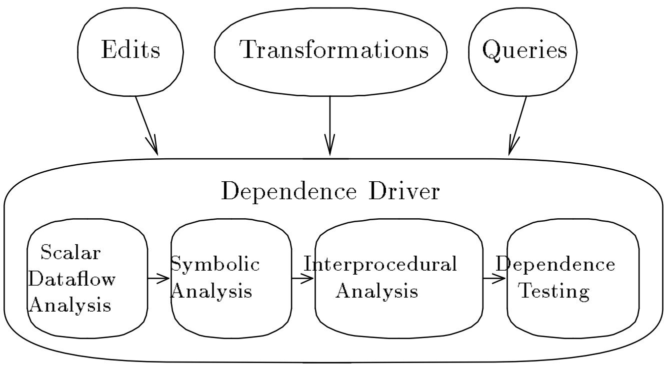

Figure 3: Dependence Analyzer Architecture

5.2 Dependence Analysis Driver

The dependence analysis driver insulates and coordinates the internal phases of the dependence analyzer from the programming environment. It serves four important functions: coordination, query management, change notification, and dependence updates.

First, the driver *coordinates* each of the internal phases of the analyzer. This is especially important for incremental analysis. It also invokes syntax and type checking after edits to ensure that the input program is correct before continuing program analysis. *Query management* is achieved by hiding the internal representation of the dependence graph and enforcing a standard interface for all queries about dependence information. Queries are insulated from the dependence graph by the driver, allowing lazy dependence updates after a series of edits.

The driver is also responsible for *change notification.* It receives notice of program changes from the editor and user interface, determines the scope of changes, and decides on an appropriate update strategy. After updates have been performed, the driver must also notify other parts of the environment. Finally, the dependence driver also supports an interface for direct *dependence updates.* This interface enables efficient incremental updates of dependence information after structured transformations with known effects.

5.3 Scalar Dataflow Analysis

Scalar dataflow analysis computes dataflow information, control dependences, and data dependences for scalar variables. It also provides a framework for the later phases of the dependence analyzer. PED first constructs the control flow graph and postdominator tree. It then computes dominance frontiers for each scalar variable and uses them to build the *static single assignment* (SSA) graph for each procedure [21]. Edges in the SSA graph correspond to precise true dependences for scalar variables.

Next, PED constructs a coarse dependence graph for array variables in each loop nest by connecting {Defs} with {Defs $\cup$ Uses}. These edges are later refined through dependence testing to construct dependence edges. The same technique is used to build a set of coarse anti and output dependences for scalar variables in loop nests. More precise anti and output dependences may be calculated for scalars using techniques similar to those used in building the SSA graph, but we do not find it necessary. PED also inserts loop-carried dependences for unanalyzed procedure calls, input and output statements, and branches out of loops.

The scalar dataflow analysis phase also calculates information used to determine whether variables in a loop may be safely made private. If a scalar variable is not live outside a specific loop and does not have loop-carried true dependences, it may be made private to eliminate loop-carried storage (anti and output) dependences. PED currently assumes that all arrays are live outside of loops. Transformations such as loop distribution also need control dependences, which are calculated from the postdominator tree (see Sections 2.2 and 7.3).

All internal dependence representations and underlying structures are complete in the current implementation of PED. The only remaining task is to compute live ranges and dependences for scalars using the SSA graph.

5.4 Symbolic Analysis

Symbolic analysis is the set of techniques used to determine and compare the values of expressions in programs. Symbolic analysis improves the precision of interprocedural analysis and dependence testing. When possible, this phase eliminates or characterizes symbolic expressions used to determine loop bounds, loop steps, array subscript expressions, array dimensions, and control flow.

The SSA graph for scalars, produced by scalar dataflow analysis, provides the framework for symbolic analysis. Data flow along SSA edges is analyzed to deduce constraints and relationships on symbolic expressions, which may be in turn propagated. Studies have shown that symbolic analysis is essential for precise analysis of scientific programs [25, 27, 49]. The symbolic analyzer performs the following steps.

Constant propagation uses the *sparse conditional constant* algorithm to eliminate as many symbolics as initially feasible [53].

Auxiliary induction variable substitution replaces auxiliary induction variables by functions of loop index variables.

Expression folding propagates symbolic expressions along edges in the SSA graph.

Loop invariant expression detection detects symbolics that may be eliminated by the symbolic expression simplifier.

Reduction recognition examines the SSA graph to detect opportunities for using reduction operators.

Unlike automatic parallelization tools, PED does not make transformations to the program for purposes of analysis. Accordingly, the results of symbolic analysis are stored in *shadow expressions*, annotations attached to expressions in the program. If the dependence testing phase desires the value of a program expression, it uses a shadow expression if one exists.

Only constant propagation has been implemented in the current version of PED. We are adding shadow expressions and other parts of the symbolic analyzer as part of a larger effort that also encompasses interprocedural symbolic analysis.

5.5 Interprocedural Analysis

The presence of procedure calls complicates the process of analyzing dependences. Interprocedural analysis is required so that worst case assumptions need not be made when calls are encountered. ParaScope performs conventional interprocedural analysis that discovers constants, aliasing, flow insensitive side effects such as REF and MOD, and flow sensitive side effects such as USE and KILL [15, 20]. However, improvements are limited because arrays are treated as monolithic objects, making it impossible to determine whether two references to an array actually access the same memory location.

To provide more precise analysis, array accesses can be summarized in terms of *regular sections* or *data access descriptors* that describe subsections of arrays such as rows, columns, and rectangles [7, 16, 28]. Local symbolic analysis and interprocedural constants are required to build accurate regular sections. Once constructed, regular sections may be quickly intersected during interprocedural analysis and dependence testing to determine whether dependences exist.

We are integrating existing ParaScope interprocedural analysis and transformations such as *inlining* and *cloning* into PED [19, 20]. The implementation of regular sections is also under way.

5.6 Dependence Testing

The dependence testing phase refines the coarse dependence graph for array variables created by scalar analysis and sharpened by interprocedural analysis. PED classifies subscripts in a pair of array references according to two orthogonal criteria: *complexity*, which refers to the number of indices appearing within the subscript; and *separability*, which describes whether a given subscript interacts with other subscripts for the purpose of dependence testing.

Using this classification scheme, appropriate dependence tests are selected and applied to each subscript position in a pair of array references. Dependence edges are eliminated if dependence between the references can be disproved. Otherwise, dependence testing characterizes the dependences with a minimal set of hybrid distance/direction vectors. This dependence information is vital for guiding transformations. PED's dependence tests are discussed in detail elsewhere [25].

Most dependence tests have been implemented in the current version of PED; we are in the process of extending them to handle symbolic expressions, complex iteration spaces, and regular sections.

5.7 Analysis of Synchronization

In a sophisticated parallel program, the user may wish to employ complex synchronization. Typically, synchronization constructs are used to ensure that a dependence is satisfied. When synchronization is present, it is important to eliminate any *preserved dependences* so that the user will not need to consider them further.

Establishing that the order specified by certain dependences will always be observed has been shown to be co-NP-hard, but techniques have been developed to identify dependences that are satisfied by existing synchronization under restricted circumstances [17, 52]. The current implementation of PED can determine if event style synchronization is sufficient to protect a particular dependence.

5.8 Utilizing External Analysis

To overcome gaps in the current implementation of dependence analysis, PED may import dependence information from PFC, the Rice system for automatic vectorization and parallelization [4]. PFC's dependence analyzer is more mature and contains symbolic analysis, interprocedural regular sections and constants, as well as control and data dependence analysis. PFC produces a file of dependence information that PED converts into its own internal representation. This process is a temporary expedient which will become unnecessary when dependence analysis in PED is complete.

6 Transformations

PED provides a variety of interactive, structured transformations that enhance or expose parallelism in programs. These 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. Therefore the user is relieved of the responsibility of making tedious and error prone program changes.

PED evaluates each transformation invoked according to three criteria: applicability, safety, and profitability. A transformation is *applicable* if it can be mechanically performed. For example, loop interchange is inapplicable for a single loop. A transformation is *safe* if it preserves the meaning of the original sequential program. Some transformations are always safe, others require a specific dependence pattern. Finally, PED classifies a transformation as *profitable* if it can determine that the transformation directly or indirectly improves the parallelism of the resulting program.

To perform a transformation, the user makes a program selection and invokes the desired transformation. If the transformation is inapplicable, PED responds with a diagnostic message. If the transformation is safe, PED advises the user as to its profitability. For parameterized transformations, PED may also suggest a parameter value. The user may then apply the transformation. For example, see loop skewing and unroll and jam in Sections 7.2 and 7.4.

If the transformation is unsafe or unprofitable, PED responds with a warning explaining the cause. In these cases the user may decide to override the system advice and apply the transformation anyway. For example, if a user decides to parallelize a loop with loop-carried dependences, PED will warn the user of the dependences but allow the loop to be made parallel. This override ability is extremely important in an interactive tool, since it allows the user to apply knowledge unavailable to the tool. The program AST and dependence information are automatically updated after each transformation to reflect the transformed source.

PED supports a large set of transformations that have proven useful for introducing, discovering, and exploiting parallelism. PED also supports transformations for enhancing the use of the memory hierarchy. These transformations are described in detail in the literature [2, 4, 12, 32, 33, 36, 41, 56]. We classify the transformations in PED as follows.

Reordering Transformations

Loop Distribution	Loop Interchange
Loop Skewing	Loop Reversal
Statement Interchange	Loop Fusion

Dependence Breaking Transformations

Privatization	Scalar Expansion
Array Renaming	Loop Peeling
Loop Splitting	Alignment

Memory Optimizing Transformations

Strip Mining	Scalar Replacement
Loop Unrolling	Unroll and Jam

Miscellaneous Transformations

Sequential $\leftrightarrow$ Parallel	Loop Bounds Adjusting
Statement Addition	Statement Deletion

Reordering transformations change the order in which statements are executed, either within or across loop iterations. They are safe if all program dependences in the original program are preserved. Reordering transformations are used to expose or enhance loop-level parallelism. They are often performed in concert with other transformations to structure computations in a way that allows useful parallelism to be introduced.

Dependence breaking transformations are used to break specific dependences that inhibit parallelism. They may introduce new storage to eliminate storage-related anti or output dependences, or convert loop-carried dependences to loop-independent dependences, often enabling the safe application of other transformations. If all the dependences carried on a loop are eliminated, the loop may then be run in parallel.

Memory optimizing transformations adjust a loop's balance between computations and memory accesses to make better use of the memory hierarchy and functional pipelines. These transformations have proven to be extremely effective for both scalar and parallel machines.

6.1 Incremental Analysis

A significant advantage of structured transformations is that their effects are known in advance. In particular, few transformations affect global dataflow or symbolic information. PED can thus perform updates very efficiently. Some transformations may require partial reanalysis, while others may directly update the existing dependence graph. Below, we classify safe transformations based on their PED update algorithms.

None
Statement Interchange, Loop Bounds Adjusting

Move edges
Loop Interchange, Array Renaming

Modify edges
Loop Distribution, Loop Skewing,
Loop Reversal, Alignment, Privatization

Delete edges
Scalar Expansion, Statement Deletion

Add edges
Strip Mining, Scalar Replacement

Redo dependence testing
Loop Peeling, Loop Splitting

Redo dependence analysis for loop nest
Loop Fusion, Loop Unrolling,
Unroll and Jam, Statement Addition

7 Example Transformations

Although many of the algorithms for applying these transformations have appeared elsewhere, our implementation gives profitability advice and performs incremental updates of dependence information. Rather than describe all these phases for each transformation, we have chosen to examine only a few interesting transformations in detail. We discuss loop interchange, loop skewing, loop distribution, and unroll and jam. The purpose, mechanics, and safety of these transformations are presented, followed by their profitability estimates, user advice, and incremental dependence update algorithms.

7.1 Loop Interchange

Loop interchange is a key transformation that modifies the traversal order of the iteration space for the selected loop nest [4, 55]. It has been used extensively in vectorizing and parallelizing compilers to adjust the granularity of parallel loops and to expose parallelism [4, 36, 56]. PED interchanges pairs of adjacent loops. Loop permutations may be performed as a series of pairwise interchanges. PED supports interchange of triangular or skewed loops. It also interchanges hexagonal loops that result after skewed loops are interchanged.

Safety Loop interchange is safe if it does not reverse the order of execution of the source and sink of any dependence. PED determines this by examining the direction vectors for all dependences carried on the outer loop. If any dependence has a direction vector of the form $(<,>)$, interchange is unsafe. These dependences are called *interchange preventing*. They are precomputed and stored as a flag in the dependence edge. Each dependence edge carried on the outer loop is examined. If any one of these has the interchange preventing flag set, PED advises the user that interchange is unsafe.

Profitability PED judges the profitability of loop interchange by calculating which of the loops will be parallel after the interchange. A dependence carried on the outer loop will move inward if it has a direction vector of the form $(<,=)$. These dependences are called *interchange sensitive*. They are also precomputed and stored in a flag on each dependence edge. PED examines each dependence edge on the outer loop to determine where it will be following interchange. It then checks for dependences carried on the inner loop as well; they move outward following interchange. Depending on the result, PED advises the user that neither, one, or both of the loops will be parallel after interchange.

Update Updates after loop interchange are very quick. Dependence edges on the interchanged loops are moved directly to the appropriate loop level based on their interchange sensitive flags. All the dependences in the loop nest then have the elements in their direction vector corresponding to the interchanged loops swapped, *e.g.,* $(<,=)$ becomes $(=,<)$. Finally, the interchange flags are recalculated for dependences in the loop nest.

7.2 Loop Skewing

Loop skewing is a transformation that changes the shape of the iteration space to expose parallelism across a wavefront [31, 39, 43, 56]. It can be applied in conjunction with loop interchange, strip mining, and loop reversal to obtain effective loop-level parallelism in a loop nest [9, 54, 57]. All of these transformations are supported in PED.

Loop skewing is applied to a pair of perfectly nested loops that both carry dependences, even after loop interchange. Loop skewing adjusts the iteration space of these loops by shifting the work per iteration, changing the shape of the iteration space from a rectangle to a parallelogram. This is illustrated in Figure 4. Skewing changes dependence distances for the inner loop so that all dependences are carried on the outer loop after loop interchange. The inner loop can then be safely parallelized.

Loop skewing of degree α is performed by adding α times the outer loop index variable to the upper and lower bounds of the inner loop, followed by subtracting the same amount from each occurrence of the inner loop index variable in the loop body. In the example below, the second loop nest results when the `j` loop in the first loop nest is skewed by degree 1 with respect to loop `i`.

```
DO i = 1, 100
   DO j = 2, 100
      A(i,j) = A(i-1,j) + A(i,j-1)
   ENDDO
ENDDO
DO i = 1, 100
   DO j = i+2, i+100
      A(i,j-i) = A(i-1,j-i) + A(i,j-i-1)
   ENDDO
ENDDO
```

Figure 4 illustrates the iteration space for this example.

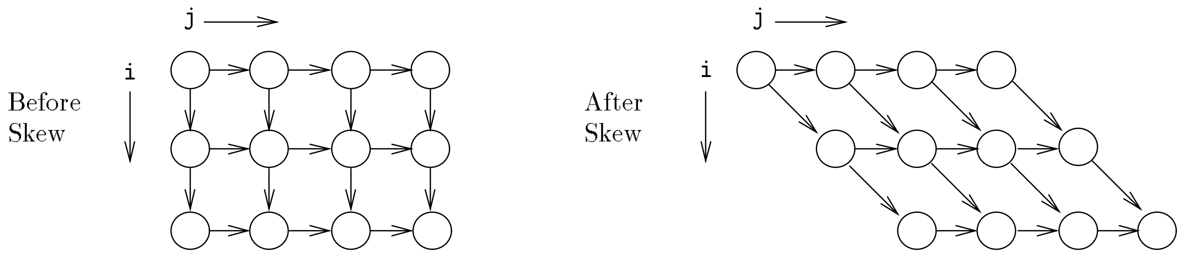

Figure 4: Effect of Loop Skew on Dependences and Iteration Space

For the original loop, dependences with distance vectors $(1,0)$ and $(0,1)$ prevent either loop from being safely parallelized. In the skewed loop, the distance vectors for dependences are transformed to $(1,1)$ and $(0,1)$. There are no longer any dependences within each column of the iteration space, so parallelism is exposed. However, to introduce the parallelism on the `i` loop requires a loop interchange.

Safety Loop skewing is always safe because it does not change the order in which array memory locations are accessed. It only changes the shape of the iteration space.

Profitability To determine if skewing is profitable, PED ascertains whether skewing will expose parallelism that can be made explicit using loop interchange and suggests the minimum skew amount needed to do so. This analysis requires that all dependences carried on the outer loop have precise distance vectors. Skewing is only profitable if:

1. $\exists$ dependences on the inner loop, and
2. $\exists$ at least one dependence on the outer loop with a distance vector (d_1, d_2), where $d_2 \leq 0$.

The interchange preventing or interchange sensitive dependences in (2) prevent the application of loop interchange to move all dependences to the outer loop. If they do not exist, at least one loop may already be safely parallelized, possibly by using loop interchange. The purpose of loop skewing is to change the distance vector to (d_1, d_2'), where $d_2' \geq 1$. In terms of the iteration space, loop skewing is needed to transform dependences that point down or downwards to the left into dependences that point downwards to the right. Followed by loop interchange, these dependences will remain on the outer loop, allowing the inner loop to be safely parallelized.

To compute the skew degree, we first consider the effect of loop skewing on each dependence. When skewing the inner loop with respect to the outer loop by an integer degree α, the original distance vector (d_1, d_2) becomes $(d_1, \alpha d_1 + d_2)$. So for any dependence where $d_2 \leq 0$, we want α such that $\alpha d_1 + d_2 \geq 1$. To find the minimal skew degree we compute

$$\alpha = \left\lceil \frac{1 - d_2}{d_1} \right\rceil$$

for each dependence, taking the maximum α for all the dependences; this is suggested as the skew degree.

Update Updates after loop skewing are also very fast. After skewing by degree α, the incremental update algorithm changes the original distance vectors (d_1, d_2) for all dependences in the nest to $(d_1, \alpha d_1 + d_2)$, and then updates their interchange flags.

7.3 Loop Distribution

Loop distribution separates independent statements inside a single loop into multiple loops with identical headers [4, 37]. It is used to expose partial parallelism by separating statements which may be parallelized from those that must be executed sequentially. It is a cornerstone of vectorization and parallelization.

In PED the user can specify whether distribution is for the purpose of vectorization or parallelization. If the user specifies vectorization, then each statement is placed in a separate loop when possible. If the user specifies parallelization, then statements are grouped together into the fewest loops such that the most statements can be made parallel. The user is presented with a partition of the statements into new loops, as well as an indication of which loops are parallelizable. The user may then apply or reject the distribution partition.

Safety To maintain the meaning of the original loop, the partition must not put statements that are involved in *recurrences* into different loops [32, 37]. Recurrences are calculated by finding strongly connected regions in the subgraph composed of loop-independent dependences and dependences carried on the loop to be distributed. Statements not involved in recurrences may be placed together or in separate loops, but the order of the resulting loops must preserve all other data and control dependences. PED always computes a partition which meets these criteria.

If there is control flow in the original loop, the partition may cause decisions that occur in one loop to be used in a later loop. These decisions correspond to loop-independent control dependences that cross between partitions. We use Kennedy and McKinley's method to insert new arrays, called *execution variables*, that record these "crossing" decisions [32]. Given a partition, this algorithm introduces the minimal number of execution variables necessary to effect the partition, even for loops with arbitrary control flow.

Profitability Currently PED does not change the order of statements in the loop during partitioning. This simplification improves the recognizability of the resulting program, but may reduce the parallelism uncovered. In particular, statements that fall lexically between statements in a recurrence will be put into the same partition as the recurrence. In addition, when the source of a dependence lexically follows the sink, these statements will be placed in the same partition. A more flexible partitioning algorithm is planned.

When distributing for vectorization, statements not involved in recurrences are placed in separate loops. When distributing for parallelization, they are partitioned as follows. A statement is added to the preceding partition only if it does not cause that partition to be sequentialized. Otherwise it begins a new partition. Consider distributing the following loop for parallelization.

```
     DO i = 2, n
S1     A(i) = ...
S2     ... = A(i-1)
     ENDDO
```

```
     DO i = 2, n
S1     A(i) = ...
     ENDDO
     DO i = 2, n
S2     ... = A(i-1)
     ENDDO
```

This loop contains only the loop-carried true dependence $S_1 \delta S_2$. Since there are no recurrences, S_1 and S_2 begin in separate partitions. S_1 is placed in a parallel partition, then S_2 is considered. The addition of S_2 to the partition would instantiate the loop-carried true dependence, causing the partition to be sequential. Therefore, S_2 is placed in a separate loop and both loops may be made parallel.

Update Updates can be performed quickly on the existing dependence graph after loop distribution. For each new loop PED also creates new loop info structures and attaches them to the AST. Data and control dependences between statements in the same partition remain unchanged. Data dependences carried on the distributed loop between statements placed in separate partitions are converted into loop-independent dependences (as in the above example).

Loop-independent control dependences that cross partitions are deleted and replaced as follows. First, loop-independent data dependences are introduced between the definitions and uses of execution variables representing the crossing decision. A control dependence is then inserted from the test on the execution variable to the sink of the original control dependence. The update algorithm is explained more thoroughly elsewhere [32].

7.4 Unroll and Jam

Unroll and jam is a transformation that *unrolls* an outer loop in a loop nest, then *jams* (or *fuses*) the resulting inner loops [2, 13]. Unroll and jam can be used to convert dependences carried by the outer loop into loop independent dependences or dependences carried by some inner loop. It brings two accesses to the same memory location closer together and can significantly improve performance by enabling reuse of either registers or cache. When applied in conjunction with *scalar replacement* on scientific codes, unroll and jam has resulted in integer factor speedups, even for single processors [12]. Unroll and jam may also be applied to imperfectly nested loops or loops with complex iteration spaces. Figure 5 shows an example iteration space before and after unroll and jam of degree 1.

Before performing unroll and jam of degree α on a loop with step σ, we may need to use *loop splitting* to make the total number of iterations divisible by $\alpha + 1$ by separating the first few iterations of the loop into a preloop. We then create α additional copies of the loop body. All occurrences of the loop index variable in the i^{th} new loop body must be incremented by σi. The step of the loop is then increased to $\sigma(\alpha + 1)$.

In the following matrix multiply example, loop `i` is unrolled and jammed by one to bring together references to `B(k,j)`, resulting in the second loop nest. Unroll and jam may also be performed on loop `j` to bring together references to `A(i,k)`.

```
DO i = 1, 100
  DO j = 1, 100
    C(i,j) = 0.00
    DO k = 1, 100
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    ENDDO
  ENDDO
ENDDO
DO i = 1, 100, 2
  DO j = 1, 100
    C(i,j)   = 0.0
    C(i+1,j) = 0.0
    DO k = 1, 100
      C(i,j)   = C(i,j)   + A(i,k)   * B(k,j)
      C(i+1,j) = C(i+1,j) + A(i+1,k) * B(k,j)
    ENDDO
  ENDDO
ENDDO
```

Safety To determine safety, an alternative formulation of unroll and jam is used. Unroll and jam is equivalent to strip mining the outer loop by the unroll degree, interchanging the strip mined loop to the innermost position, and then completely unrolling the strip mined loop. Since strip mining and loop unrolling are always safe, we only need to determine whether we can safely interchange the strip mined loop to the innermost position.

PED determines this by searching for interchange preventing dependences on the outer loop. Unroll and jam is unsafe if any dependence carried by the outer loop has a direction vector of the form $(<, >)$. Even if such a dependence is found, unroll and jam is still safe if the unroll degree is less than the distance of the dependence on the outer loop, since this dependence would remain carried by the outer loop. PED will either warn the user that unroll and jam is unsafe, or provide a range of safe unroll degrees.

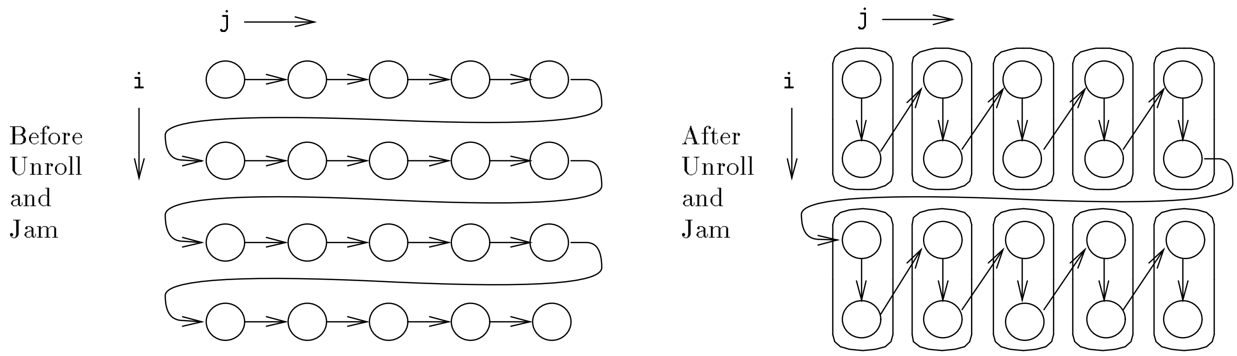

Figure 5: Effect of Unroll and Jam on Iteration Space

Unroll and jam of imperfectly nested loops changes the execution order of the imperfectly nested statements with respect to the rest of the loop body. Dependences carried on the unrolled loop with distance less than or equal to the unroll degree are converted into loop-independent dependences. If any of these dependences cross between the imperfectly nested statements and the statements in the inner loop, they inhibit unroll and jam. Specifically, the intervening statements cannot be moved and prevent fusion of the inner loops.

Profitability *Balance* describes the ratio between computation and memory access rates [13]. Unroll and jam is profitable if it brings the balance of a loop closer to the balance of the underlying machine. PED automatically calculates the optimal unroll and jam degree for a loop nest, including loops with complex iteration spaces [12].

Update An algorithm for the incremental update of the dependence graph after unroll and jam is described elsewhere [12]. However, we chose a different strategy. Since no global dataflow or symbolic information is changed by unroll and jam, PED rebuilds the scalar dependence graph for the loop nest and refines it with dependence tests. This update strategy proved much simpler to implement, and does not take noticeably longer to execute.

7.5 Implementation Status

All of the structured program transformations in this paper have been implemented in PED, including their corresponding incremental update algorithms. We are, as always, in the process of further extending PED's transformation capabilities. A major effort is underway to incorporate automatic parallelization strategies within PED in order to provide users with further assistance in the parallelization process [42].

8 Edits

Editing is fundamental for any program development tool because it is the most flexible means of making program changes. The ParaScope Editor therefore provides advanced editing features. When editing, the user has complete access to the functionality of the hybrid text and structure editor underlying PED, including simple text entry, template-based editing, search and replace functions, intelligent and customizable view filters, and automatic syntax and type checking.

Rather than reanalyze immediately after each edit, PED waits for a reanalyze command from the user. This avoids analyzing intermediate stages of the program that may be illegal or simply uninteresting to the user. Both transformations and the dependence display are disabled during an editing session, because they rely on dependence information that may be invalidated by the edits. Once the user prompts PED, the dependence driver invokes syntax and type checking. If errors are detected, the user is warned. Otherwise reanalysis proceeds.

8.1 Incremental Analysis

Unfortunately, incremental dependence analysis after edits is a very difficult problem. As we have already seen, precise dependence analysis requires utilization of several different kinds of information. In order to calculate precise dependence information, PED may need to incrementally update the control flow, control dependence, SSA, and call graphs, as well as recalculate live range, constant, symbolic, interprocedural, and dependence testing information.

Several algorithms for incremental analysis can be found in the literature; *e.g.,* dataflow analysis [47, 58], interprocedural analysis [10, 46], interprocedural recompilation analysis [11], as well as dependence analysis [45]. However, few of these algorithms have been implemented and evaluated in an interactive environment. Rather than tackle all these problems at once, we chose a simple yet practical strategy for the current implementation of PED. First, the scope of each program change is evaluated. Incremental analysis is applied only when it may be profitable, otherwise batch dependence analysis is invoked. PED will apply incremental dependence analysis when the following situations are detected:

No update needed Many program edits fall into this category. It is trivial to determine that changes to comments or whitespace do not require reanalysis. Other cases include changes to arithmetic expressions that do not disturb control flow or symbolic analysis. For instance, changing the assignment `a(i)=b(i)` to `a(i)=b(i)+1` does not affect dependence information one whit.

Delete dependence edges Removal of an array reference may be handled simply by deleting all edges involving that reference.

Add dependence edges Addition of an array reference may be handled by scanning the loop nest for occurrences of the same variable, performing dependence tests between the new reference and any other references, and adding the necessary dependence edges.

Redo dependence testing Changes to loop bounds or array subscript expressions require dependence testing to be performed on all affected array variables.

Redo local symbolic analysis Some types of program changes do not affect the scalar dependence graph, but may require symbolic analysis to be reapplied. For instance, changing the assignment `j=j+1` to `j=j+2`, where `j` is an auxiliary induction variable, requires redoing symbolic analysis and dependence testing.

Redo local dependence analysis Changes such as the modification of control flow or variables involved in symbolic analysis require significant updates best handled by redoing dependence analysis. However, the nature of the change may allow the reanalysis to be limited to the current loop nest or procedure. In these cases, the entire program does not need to be reanalyzed.

8.2 Implementation Status

Editing is fully supported in PED, but difficulties with the underlying editor currently require batch dependence analysis to be performed at the end of an editing session. However, the incremental framework is in place.

9 Related Work

Several other research groups are also developing advanced parallel programming tools. PED's analysis and transformation capabilities compare favorably to automatic parallelization systems such as Parafrase, PTRAN, and of course PFC. Our work on interactive parallelization bears similarities to SIGMACS, PAT, and SUPERB.

PED has been greatly influenced by the Rice Parallel Fortran Converter (PFC), which has focused on the problem of automatically vectorizing and parallelizing sequential Fortran [4]. PFC has a mature dependence analyzer which performs data dependence analysis, control dependence analysis, interprocedural constant propagation [15], interprocedural side effect analysis of scalars [20], and interprocedural array section analysis [16, 28]. PED expands on PFC's analysis and transformation capabilities and makes them available to the user in an interactive environment.

Parafrase was the first automatic vectorizing and parallelizing compiler [36]. It supports program analysis and performs a large number of program transformations to improve parallelism. In Parafrase, program transformations are structured in phases and are always applied where applicable. Batch analysis is performed after each transformation phase to update the dependence information for the entire program. Parafrase-2 adds scheduling and improved program analysis and transformations [44]. More advanced interprocedural and symbolic analysis is planned [27]. Parafrase-2 uses FAUST as a front end to provide interactive parallelization and graphical displays [26].

PTRAN is also an automatic parallelizer with extensive program analysis. It computes the SSA and *program dependence graphs*, and performs constant propagation and interprocedural analysis [22]. PTRAN introduces both task and loop parallelism, but currently the only other program transformations are variable privatization and loop distribution [1].

SIGMACS, a programmable interactive parallelizer in the FAUST programming environment, computes and displays call graphs, process graphs, and a statement dependence graph [26, 48]. In a process graph each node represents a task or a process, which is a separate entity running in parallel. The call and process graphs may be animated dynamically at run time. SIGMACS also performs several interactive program transformations, and is working on automatic updating of dependence information.

PAT is also an interactive parallelization tool [50]. Its dependence analysis is restricted to Fortran programs where only one write occurs to each variable in a loop. In addition, PAT uses simple dependence tests that do not calculate distance or direction vectors. Hence, it is incapable of applying loop level transformations such as loop interchange and skewing. However, PAT does support replication and alignment, insertion and deletion of assignment statements, and loop parallelization. It can also insert synchronization to protect specific dependences. PAT divides analysis into scalar and dependence phases, but does not perform symbolic or interprocedural analysis. The incremental dependence update that follows transformations is simplified due to its austere analysis [51].

SUPERB interactively converts sequential programs into data parallel SPMD programs that can be executed on the SUPRENUM distributed memory multiprocessor [59]. SUPERB provides a set of interactive program transformations, including transformations that exploit data parallelism. The user specifies a data partitioning, then node programs with the necessary *send* and *receive* operations are automatically generated. Algorithms are also described for incremental update of use-def and def-use chains following structured program transformations [34].

10 Conclusions

Our experience with the ParaScope Editor has shown that dependence analysis can be used in an interactive tool with acceptable efficiency. This efficiency is due to fast yet precise dependence analysis algorithms, and a dependence representation that makes it easy to find dependences and to reconstruct them after a change. To our knowledge, PED is the first tool to offer general editing with dependence reconstruction along with a substantial collection of useful program transformations.

PED's ability to analyze and display dependence information has made it into a powerful tool for the experienced parallel programmer. By reviewing the dependences carried by a particular parallel loop, the programmer can avoid the kinds of simple mistakes that may later take months to find and correct. In this paper we have shown how to keep the cost of providing this information low enough to make PED a practical interactive tool.

The analysis and representation of dependence in the ParaScope Editor have also proven very useful in the development of several other advanced tools, including a compiler [29] and data decomposition tools [5, 6] for distributed-memory machines, and an on-the-fly access anomaly detection system for shared-memory machines [30].

11 Acknowledgments

We would like to thank Vasanth Balasundaram, Preston Briggs, Keith Cooper, Paul Havlak, Marina Kalem, Rhonda Reese, Jaspal Subhlok, and Linda Torczon 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 PFC, $\mathbb{R}^n$, and ParaScope research groups, who have provided the software infrastructure upon which PED is built. Finally, we would like to thank our collaborators at Los Alamos National Laboratory, particularly Bob Hiramoto and Margaret Simmons, for numerous suggestions that have positively influenced the current design of PED.

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[10] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, 1990.

[11] M. Burke, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. Technical Report TR90-126, Dept. of Computer Science, Rice University, 1990.

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

[13] 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.

[14] 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.

[15] 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, July 1986.

[16] 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.

[17] 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.

[18] D. Chen, H. Su, and P. Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[19] K. Cooper, M. Hall, and L. Torczon. An experiment with inline substitution. Technical Report TR90-128, Dept. of Computer Science, Rice University, July 1990. To appear in *Software—Practice and Experience*.

[20] 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.

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

[22] 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.

[23] 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, 1990.

[24] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Confer-*

ence on Supercomputing. Springer-Verlag, Athens, Greece, June 1987.

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

[26] V. Guarna, D. Gannon, Y. Gaur, and D. Jablonowski. Faust: An environment for programming parallel scientific applications. In *Proceedings of Supercomputing '88*, Orlando, FL, November 1988.

[27] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high performance parallelizing compilers. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

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

[29] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Dept. of Computer Science, Rice University, February 1991. To appear in J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*, Elsevier, 1991.

[30] 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.

[31] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.

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

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

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

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

[36] 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.

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

[38] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[39] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

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

[41] D. Loveman. Program improvement by source-to-source transformations. *Journal of the ACM*, 17(2):121–145, January 1977.

[42] K. S. McKinley. Evaluating the potential for automatic parallel code generation. Technical Report TR90-148, Dept. of Computer Science, Rice University, December 1990.

[43] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971. Report No. 71-424.

[44] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[45] C. Rosene. *Incremental Dependence Analysis*. PhD thesis, Rice University, March 1990.

[46] B. Ryder and M. Carroll. An incremental algorithm for software analysis. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, CA, December 1986.

[47] B. Ryder and M. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[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, Brown University, May 1989.

[54] M. E. Wolf and M. Lam. Maximizing parallelism via loop transformations. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[55] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

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

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

[58] F. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[59] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.