Experiences Using the Parascope Editor

Mary Hall Tim Harvey
Ken Kennedy Nathaniel McIntosh
Kathryn McKinley Jeffrey Oldham
Michael Paleczny Gerald Roth

CRPC-TR91173
September, 1991

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

Revised April, 1992.

Technical Paper

Experiences Using the ParaScope Editor

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

Department of Computer Science, Rice University, Houston, Texas 77251–1892

Abstract

The ParaScope Editor is an interactive parallel programming tool for developing scientific Fortran programs. In addition to editing, it assists the knowledgeable user by displaying the results of sophisticated program analyses and by providing a set of powerful interactive transformations. This paper summarizes the experiences of a number of scientific programmers and tools experts using the ParaScope Editor. We describe existing useful features as well as new functionality and paradigms that should be incorporated. These results offer insights and application for the designers of a variety of programming tools.

1 Introduction

The complexity of constructing parallel programs is a significant obstacle to widespread use of parallel computers. In the process of writing a parallel program, the programmer must consider the implications of concurrency on the correctness of their algorithm. Ideally, the programmer could be freed from this concern by an intelligent compiler that can automatically convert a sequential program to an equivalent shared-memory parallel program. Although a substantial amount of research has been devoted to automatic parallelization, such systems have not established an acceptable level of success [ABC+87, AK84, EB91a, KKLW80, SH91].

Automatic parallelization depends on dependence analysis, which identifies a conservative set of potential race conditions that make parallelization illegal. In general, an automatic system cannot parallelize a loop or apply program transformations if doing so would violate a dependence and potentially change the program's semantics. The conservative nature of dependence analysis sometimes prohibits the compiler from uncovering parallelism that may be obvious to the programmer. Unfortunately, in a batch system, there is no satisfactory mechanism for the programmer to communicate this information to the compiler.

*Presenting and corresponding author: Phone: 1-(713)-527-8101 x2732, FAX: 1-(713)-285-5136, mpal@cs.rice.edu

The ParaScope Editor (PED) serves this purpose by enabling interaction between the compiler and the programmer [BKK+89, KMT91a, KMT91b]. The compiler provides the results of extensive analysis, but the enormous volume of information computed may reduce its usefulness to the programmer. PED acts as a sophisticated filter by displaying program dependences in meaningful ways. Furthermore, the tool assists the programmer with parallelization and the application of powerful source transformations that have proven themselves in practice. The programmer refines conservative assumptions by determining which dependences are valid and by selecting transformations to be applied. The editor then updates dependence information and source code to reflect the programmer's actions.

The ParaScope Editor, under construction since 1987, has reached a level of maturity that has enabled it to be used to parallelize many large, production Fortran 77 codes. One of the methods we used to evaluate its effectiveness was a two day workshop in July 1991, sponsored by the Center for Research on Parallel Computation (CRPC), an NSF Science and Technology Center. Eight researchers from research laboratories and industry attended. Each participating organization contributed at least one Fortran program to be parallelized using PED. Assisted by a member of the ParaScope research group, the attendees introduced parallelism into their codes.[1]

This paper presents the experiences of ParaScope Editor users from the workshop and from two other experiences. From these experiences, we derived valuable information about the process of constructing parallel programs and how our tool can be improved to better support this process. To parallelize complete applications, programmers need more extensive analysis, such as more precise array analysis across procedure boundaries. They also desire more assistance in navigating through their applications, *e.g.*, locating important loops and subroutines and viewing information across procedure calls. They can understand and use this information more effectively when displayed as an annotation on the program source, rather than in a separate display. These results are clearly applicable to other programming tools assisting in the construction of shared-memory parallel programs. Moreover, tools exposing compiler information to the programmer are useful aids in achieving efficiency on other modern architectures such as superscalar and distributed-memory machines.

The paper's next section briefly describes the ParaScope Editor. In Section 3, we present a work model for parallelizing a sequential Fortran code in PED. The following section then summarizes the experiences of PED users, particularly workshop participants' experiences. Section 5 describes research planned for the ParaScope Editor inspired by the observations from Section 4. In Sections 6 and 7, we discuss related work and conclude the paper.

[1]A similar workshop at Rice on PTOOL provided some of the inspiration for the current version of the ParaScope Editor [BBC+88, HHLS90].

2 The ParaScope Editor

The ParaScope Editor supports the "exploratory parallel programming" paradigm: the user converts a sequential Fortran program into an efficient parallel version by repeatedly discovering and exploiting opportunities for parallelism [KMT91a, KMT91b]. In the process, the system analyzes the program, the user interprets the results, and the system helps the user change the program. Using PED, a programmer selects a particular loop to parallelize, and PED provides a list of dependences that may prevent parallelization. The user may apply source code transformations, satisfying dependences or introducing parallelism, or override the conservative dependence analysis. The editor automatically and incrementally updates the dependence information after applying a transformation.

This section briefly describes PED's features. In particular, we describe *dependence analysis*, which is static analysis of memory accesses to disprove race conditions. We also present the three distinct forms of editing supported by PED: *source editing*, *dependence editing* and *variable classification*. We conclude the section with a brief outline of a new user interface currently being integrated into the existing tool.

2.1 Dependence Analysis

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 the *source* S_1 must be executed before the *sink* S_2, while, in a *data dependence* $S_1 \delta S_2$, the same memory location is accessed twice, with one access writing a new value. If these accesses are not synchronized, a *race condition* occurs. A dependence is *loop-carried*, as opposed to *loop-independent*, if the source and sink of the dependence occur on different loop iterations. These dependences prevent the loop's parallelization because the memory accesses corresponding to the dependence may occur out of order if the different loop iterations executed in parallel.

PED's *dependence analyzer* conservatively locates dependences. In some cases, it produces a superset of the actual dependences in the program. To perform dependence analysis, pairs of references are tested to determine if they could access the same memory location. The analyzer applies a hierarchical suite of tests on each pair of references to determine if the references can access the same memory location [GKT91]. One of the distinguishing features of PED's analyzer is the use of sophisticated interprocedural information when loops contain procedure calls. The analysis techniques include more established techniques such as side-effect analysis [CKT86a] and constant propagation [CCKT86], as well as *regular section analysis*, *i.e.*, determining sections of arrays affected by procedure calls [CK87, HK90]. Symbolic analysis, exposing information about symbolic expressions when their values are unknown, will soon be available.

2.2 Source Editing Features

An intelligent, hybrid text and structure editor supports PED's editing functions. Fortran syntax and type checking, performed interactively, provide the user with immediate notification of syntactic and type-related semantic errors. In addition to standard graphical editing functions, *e.g.*, scrolling and search/replace, the editor uses its understanding of Fortran to provide a number of advanced features. *Template-based editing* allows the user to select from a menu of Fortran language constructs to build a program in a structured fashion. *View filtering*, a key interface feature, provides a means to selectively display certain types of language or textual constructs [EE68]. For example, source code view filter predicates can test whether a line contains a specific word, whether the line contains a syntax or semantic error or whether the line is a specification statement, an executable statement, a subprogram header or a loop header.

2.3 Dependence Viewing and Dependence Editing

The dependence display contains a table of dependences carried by the selected loop, showing each dependence's source and sink variable references and characterizations, such as exactness, required by program transformations. An *exact dependence* has been proven *feasible* by the analyzer, *i.e.*, the race condition will occur on a possible execution path through the program, while a *conservative dependence* exists when the analyzer cannot disprove a possible execution path's existence. A conservative dependence is *false* when the analyzer cannot disprove its existence, but the pair of variable references do not access the same memory location.

When a dependence is selected, the text of the source and sink are identified in the editing window. View filtering allows the programmer to display only certain dependences in the loop, such as all dependences on a particular variable. Users may also *edit* the dependence information by deleting dependences which may have arisen from the analyzer's overly conservative assumptions. Dependences can be deleted individually, or by describing them using the view filtering mechanism. In some cases, dependence deletion will eliminate all loop-carried dependences, allowing for loop parallelization.

2.4 Variable Classification

A *private variable* is defined and used during the same loop iteration. Thus, the variable's value during one loop iteration is independent of its value during other iterations. Correctly identifying these variables aids loop parallelization. The compiler can locate private variables, such as loop index variables, using KILL analysis [ASU86, Cal88], which, like dependence analysis, is conservative. A *shared variable*'s value during one loop iteration does affect its value during other iterations.

The variable display allows the user another mechanism to communicate with the compiler. The variable pane, displaying a table of variables, shows each variable's name, dimension, common block if applicable, and shared or private status. In addition, the variable's definitions reaching the loop or uses outside the loop are presented to understand its use in the rest of the program. The user, using his knowledge of the program, may edit a variable's shared or private status, possibly enabling parallelization by eliminating remaining loop dependences.

2.5 Transformations

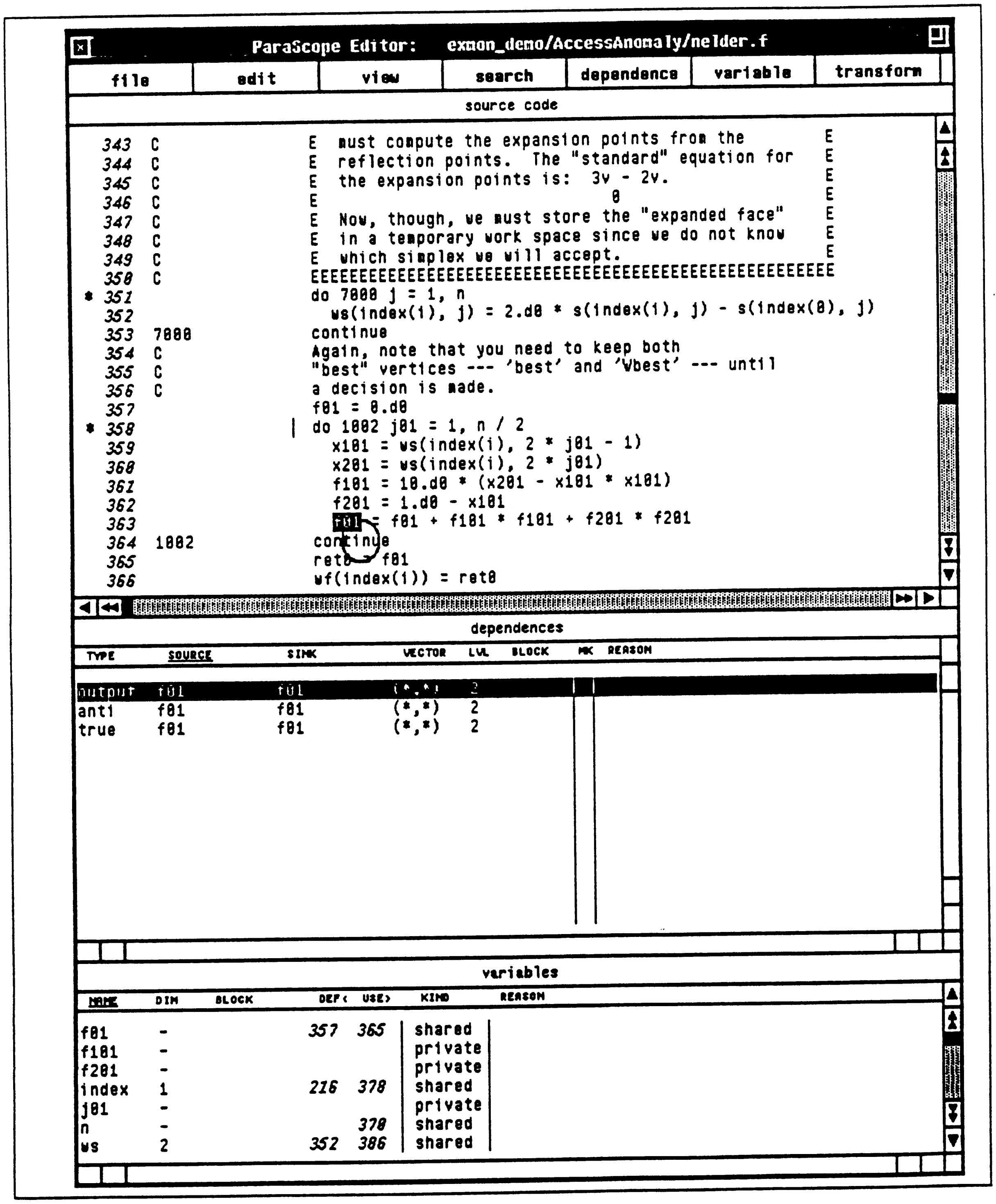
PED provides a number of interactive structured transformations enhancing or exposing a subroutine's parallelism. Transformations are applied according to a *power steering* paradigm: the user specifies the transformations to be made, and the system provides advice and carries out the mechanical details. The system advises whether the transformation is *applicable* (makes syntactic sense), *safe* (preserves the semantics of the program) and *profitable* (contributes to parallelization). The transformations' complexity makes their correct application difficult and tedious. Thus, power steering provides safe, profitable and correct application of transformations, incrementally updating the dependence information.

PED supports a large set of transformations proven useful for introducing, discovering, and exploiting parallelism and for enhancing memory hierarchy use. These transformations are described in detail in the literature [AC72, ACK87, AK87, CCK90, KKLW84, KM90, KMT91b, Lov77, Wol86]. Figure 1 shows a taxonomy of the transformations supported in PED.

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 with other transformations to structure computations in a way that enables useful parallelism to be introduced.

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	**Loop 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**

Figure 1 A taxonomy of the transformations supported in PED.



```
343  C          E  must compute the expansion points from the        E
344  C          E  reflection points.  The "standard" equation for   E
345  C          E  the expansion points is:  3v - 2v.                E
346  C          E                                0                   E
347  C          E  Now, though, we must store the "expanded face"    E
348  C          E  in a temporary work space since we do not know    E
349  C          E  which simplex we will accept.                     E
350  C          EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
* 351           do 7000 j = 1, n
352               ws(index(i), j) = 2.d0 * s(index(i), j) - s(index(0), j)
353  7000       continue
354  C          Again, note that you need to keep both
355  C          "best" vertices --- 'best' and 'Vbest' --- until
356  C          a decision is made.
357             f01 = 0.d0
* 358         | do 1002 j01 = 1, n / 2
359               x101 = ws(index(i), 2 * j01 - 1)
360               x201 = ws(index(i), 2 * j01)
361               f101 = 10.d0 * (x201 - x101 * x101)
362               f201 = 1.d0 - x101
363               f01 = f01 + f101 * f101 + f201 * f201
364  1002       continue
365             ret0 = f01
366             wf(index(i)) = ret0
```

```
TYPE     SOURCE   SINK     VECTOR   LVL   BLOCK   MK   REASON

output   f01      f01      (*,*)    2
anti     f01      f01      (*,*)    2
true     f01      f01      (*,*)    2
```

```
NAME     DIM   BLOCK     DEF<   USE>   KIND      REASON

f01      -               357    365    shared
f101     -                             private
f201     -                             private
index    1               216    370    shared
j01      -                             private
n        -                      370    shared
ws       2               352    386    shared
```

Figure 2 The ParaScope Editor.

Dependence breaking transformations eliminate specific dependences that inhibit parallelism. They introduce new storage, eliminating storage-related dependences, and convert loop-carried dependences to loop-independent dependences.

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.

2.6 The New PED User Interface

Early in PED's development, a group consisting of experts in compilers, applications and human interfaces designed a more powerful and effective interface to the program information [FKMW90]. The new interface, implemented concurrently with changes to the editor's analysis and transformations, is in the process of being integrated with the rest of PED. Demonstrated but not used during the workshop, the interface addressed many of the workshop attendees' suggestions.

In the new interface, color and graphics convey dependences more visually. For example, red arrows appear in the source code between the source and the sink of a dependence appearing in the dependence pane. With the view filtering mechanism, the programmer may view an individual dependence or collection of dependences in this way, such as all dependences involving a particular variable or a particular statement. View filtering has been extended to filter information in the variable pane as well.

A new *dependence marking* feature allows the programmer to designate dependences as "accepted," "rejected," or "pending," with an opportunity for the programmer to provide a reason for the designation. Thus, the system differentiates dependences that have already been considered, and programmer-supplied comments remind the programmer why the designation was made. A sample screen from the new interface appears in Figure 2.

3 The PED Work Model

Our work model to parallelize Fortran programs consists of four distinct steps: profiling the code, importing the code, analyzing the code, and examining the analysis information to make modifications that expose parallelism.

3.1 Profiling the Code

First, run-time profiling pinpoints the routines in which the program spends most of its time. This information suggests the important loops in the program and the ones that should be parallelized if possible. Many systems provide profiling facilities, although at present, PED does not. For most of the workshop participants, the standard UNIX profiler provided sufficient information.

3.2 Importing Source Code

All code is entered into the ParaScope database in order to convert it into ParaScope's internal abstract syntax tree in preparation for analysis. During parsing, the code is automatically checked for syntactic correctness. If there are errors, the user corrects them before analysis can be performed. These errors are fairly common since the language recognized by Fortran 77 compilers varies greatly. The editor facilitates error correction by emboldening incorrect lines and annotating them with a message.

After all of the program's subroutines have been imported, they must be gathered together in a *composition*, a specification of the modules comprising an entire executable program [CKT$^+$86b]. During the building of the composition, the composition editor checks procedure calls to ensure that actual and formal parameters are consistent in number and type, and that each procedure is defined by exactly one module. If the composition editor finds any serious errors, the user edits the erroneous source to fix the error before any dependence analysis takes place.

3.3 Analyzing the Code

Once a composition has been built and syntactic errors have been removed, the user requests dependence analysis. PED includes a preliminary implementation of dependence analysis. In addition, dependence analysis calculated by PFC may be obtained in PED [AK84]. PFC, a research tool developed at Rice beginning in 1979, automatically converts sequential Fortran to parallel or vector form. The tool is more mature and provides more precise and robust dependence information; for the most part, PFC analysis was used during the workshop. When PED's analyzer is completely implemented, analysis will be performed by invoking ParaScope's *program compiler* to perform dependence analysis on the whole program [CKT86a, CKT$^+$86b, Hal91, Tor85].

3.4 Exposing Parallelism

The previous three steps in the work model are useful and necessary for the final stage, using the analysis generated thus far inside PED to expose parallelism. At this point, the user invokes PED on subroutines that profiling has indicated as a target for attention. The loops in these subroutines are prime candidates for parallelization because of their significant effect on execution time. The user may use editor buttons to iterate through each loop in the module or the mouse to select a particular loop. Either way, the current loop's dependences are displayed.To expose parallelism, the user may either delete dependences, reclassify variables or perform source transformations.

Dependences that can be eliminatedusually result from conservative assumptions made during dependence analysis. Using knowledge about the subroutine, the user can often delete conservative dependences and locate variables that can be made private. Again, conservative analysis may mark a variable as shared although the user knows its values during different loop iterations are independent.

Dependences that prevent parallelism can sometimes be avoided by transforming the source inside PED. PED provides a number of structured source transformations that can be performed automatically. The user need only select the transformation desired. PED then determines the safety of the transformation, the expected profitability, and the degree to which the code should be modified (for example, in loop unrolling, the editor computes the most profitable number of iterations to unroll).

If dependences still exist after all of the above techniques have been tried, the loop is probably not parallelizable. At this point, the user can completely rewrite this section of code or continue to examine other loops.

4 User Observations

4.1 Users and Their Programs

For the workshop, eight researchers from research laboratories and industry each brought at least one Fortran program to analyze. The participants and their areas of interest are listed in the table below. Both Doreen Cheng from NASA Ames Research Center and Marcia Pottle from Cornell Theory Center were familiar with PED and other parallelization tools. Doreen had evaluated many parallelization tools including Forge [For90], an interactive parallelization tool produced by Pacific Sierra; the Cray fpp autotasker [Cra], an automatic parallelization tool for Cray computers; and PED [CP91]. Marcia has taught training classes for parallel programming tools. After attending a similar PTOOL workshop at Rice several years previously, she developed a PTOOL presentation given at several user training workshops.

Mary Zosel	Lawrence Livermore National Laboratory	parallel architectures and performance
John Engle	Lawrence Livermore National Laboratory	parallel programming environments and compilation
Ralph Brickner	Los Alamos National Laboratory	Connection Machine applications and communication library
Doreen Cheng	NASA Ames Research Center	evaluating parallelization tools
Roy Heimbach	National Center for Supercomputing Applications	software for parallel programming development
Marcia Pottle	Technology Integration Group, Cornell Theory Center	mainframe coprocessing and parallelism tools
Lo Hsieh	IBM Palo Alto Scientific Center	designing parallel debugging tools
Steve Poole	IBM Kingston	evaluating parallelization tools

Table 1 PED Workshop Participants, Their Affiliations and Their Interests.

The participants contributed the small to medium-sized Fortran programs shown in Table 2. Two programs neoss and nxsns were Fortran II codes containing Cray compiler directives to force parallelization. The code for spec77 included both a vector version for IBM 3090 mainframes and a hand-coded parallel version. The program arc3d is a three-dimensional version of a two-dimensional Perfect benchmark program.

Program	Description	Contributors	Lines	Modules
neoss	thermodynamics code	Mary Zosel, John Engle	350	5
nxsns	quantum mechanics code	Mary Zosel, John Engle	1400	11
pueblo3d	hydrodynamics benchmark program	Ralph Brickner	4000	50
arc3d	3-D hydrodynamics code	Doreen Cheng	3600	25
slab2d	2-D severe storm fluid flow prototype	Roy Heimbach	550	9
slalom	benchmark program	Roy Heimbach	1200	13
dpmin	molecular mechanics and dynamics program	Marcia Pottle	5000	52
spec77	weather simulation code	Lo Hsieh, Steve Poole	5600	67

Table 2 Programs Analyzed and Parallelized During the Workshop.

Two independent researchers also evaluated PED. Joseph Stein, a physicist from Hebrew University, visited the CRPC at Syracuse University for a year to learn about parallel programming tools. Also, Katherine Fletcher, a graduate student at Rice University, spent several weeks at NASA Ames Research Center working with Doreen evaluating ParaScope, Forge, and the Cray fpp autotasker [CP91].

Joseph compared the ability of a large number of tools, including PFC and PED, to parallelize the following programs:

- hyd: a 500-line hydrodynamic code that simulates a shock tube;
- dyna: a 4000-line 1-dimensional astrophysical code;
- tfss: a 4000-line code that calculates thermodynamic parameters for a 2-dimensional Thomas-Fermi quasi-molecule in hot plasma;
- kwech: a small code to compress ASCII data.

Katherine and Doreen tested PED on a fast Fourier transform code with 770 lines of source, a structural mechanics code with 720 lines, and a computational fluid dynamics code with 3600 lines. Trying to generate fast Cray parallel source code, they evaluated the tools' dependence analysis, quality of the user interface, and functionality. Joseph and Katherine's experiences are incorporated with the others' comments.

4.2 User Interface

Working with users enabled us to evaluate the strengths and weaknesses of the current PED user interface. The primary strength is that PED reflects the results of compiler analysis with visual annotations in the source code pane. Katherine contrasted PED's interface with Forge's, which provides a number of discrete functions with separate interfaces that are difficult to relate to each other or the source code. The workshop participants easily learned how to use the dependence editing facility. The view filtering capabilities, which filter classes or instances of dependences and source level instructions, allowed the participants to concentrate when the loop carried a large number of dependences. Finally, the easily accessed on-line help facility was found to be especially useful for new users.

While the workshop participants were comfortable with editing dependences, they had difficulty finding dependences in the source pane. They wanted dependences graphically displayed and better navigation support, problems addressed by the new user interface [FKMW90]. However, the users requested graphical presentations of the program's call graph, interprocedural search/replace and interprocedural dependence navigation, features that are not available in the new interface.

4.3 Analysis

The workshop participants were quite pleased with the dependence analyzer. Dependence analysis supports source transformations. The participants agreed that the dependence information available was excellent, and they found the fact classifying dependences as exact or conservative was extremely helpful.

In particular, they found the information and precision provided by interprocedural analysis to be very useful. Interprocedural regular section analysis, which determines the subsections of arrays used and defined across procedure calls [HK91], was essential in determining if loops containing calls could be made parallel.

The dependence analyzer can be improved. Array kill analysis would eliminate a number of false dependences and could be used to determine if arrays can be made private. For example, in Figure 3, array TMP is completely redefined, *i.e.*, killed, on each iteration of the outer loop. Thus, the user may parallelize the outer loop and treat TMP as a private array.

Reduction operations that sum all elements of an array, or find the maximum or minimum value of an array, occur frequently in scientific codes. PED's current inability to recognize and parallelize such reduction operations was problematic for several users. Analysis of index arrays and auxiliary induction variables needs to be improved.

Much of the analysis is for internal use and is not intended for displaying. Several people expressed interest in having better access to this existing analysis. An example is constant propagation. Users wanted

```
DO I=1, 1000
   DO J=1, 100
      TMP(J) = B(J,I) * C(J,I)
        ....
      D(J,I) = TMP(J) + E(J,I)
   ENDDO
ENDDO
```

Figure 3 Example of an array kill.

to know variable values, or ranges of possible values, when determined by symbolic analysis. This information would allow the user to know the number of iterations a loop would execute. Some information could be added to the variable pane, removing the need to check initial parameter statements in another module. Similarly, some users wanted internal dataflow information in order to browse definition-use chains.

4.4 Dependence Editing

Joseph Stein discovered dependences unnatural paradigm for understanding parallel programs. A loop may carry a large number of dependences. A large list is difficult for a programmer to work through, particularly if little information describing the dependence is provided. For example, it would be useful to know under which conditions two references result in a dependence. It may depend on the possible values of some input variable, and the user may be able to provide knowledge proving the dependence does not exist.

A further problem with the dependence paradigm is it fails to capture the reasons behind the user's belief that a dependence does not exist. While the user can add a comment accompanying the dependence, this information is not available to the system. Thus, after program edits and reanalysis, it is difficult for the system to map deleted dependences in the previous code to calculated dependences in the current version. As a result, many deleted dependences reappear and the programmer must delete them again.

Alternatively, Joseph suggests dependence deletion occur by using user-defined assertions. These assertions are added to the program text, and the analyzer uses the assertions to rule out dependences during reanalysis. He would also like the option to test the assertions at run-time, as a debugging strategy.

Joseph's suggestions were presented in the introductory talks at our workshop. During their hands-on sessions, most participants expressed a desire to utilize an assertion facility, and we are currently planning an implementation. It is interesting to note that many assertions deal with variables and aggregate treatment of arrays. While similar to dependences, the focus on variables provides a higher level paradigm.

4.5 Transformations

Overall, the workshop participants seemed impressed by PED's support of a large number of powerful program transformations. They appreciated the automatic approach that frees the user from the tedious details of performing a transformation while indicating the applicability, safety, and profitability of a transformation. However, the participants were not without suggestions for improvements.

One complaint that was echoed by most of the workshop participants was that, although PED has many useful and powerful transformations, there was no guidance in selecting which transformations to apply. As a result, only a few of the available transformations were actually used during the workshop. It would be desirable for PED to provide a list of suggested transformations on demand.

A few additional transformations should be incorporated, including *array expansion* and *reductions*. Just as a scalar may be expanded into a one-dimensional array, any array should be able to be expanded into an array with one more dimension. More importantly, PED should locate reduction operations, followed by either automatic or user-driven handling of these reductions when it is possible for them to be removed from a loop. Such a transformation could be explicit or implicit, simply by marking the reduction statement, hiding its dependences, and transforming the code before output.

Transformations to assist the conversion from Fortran II and Fortran 66 code to structured Fortran 77 syntax would have been useful. Simply changing `IF-GOTO` constructs into the appropriate `IF-THEN-ELSE` or `DO-ENDDO` would significantly aid the user in understanding code written in other versions of Fortran. Also, since the dependence analyzer can become confused by some of the old syntax, such transformations could improve dependence detection.

4.6 Profiling

Many of the workshop groups examined run-time performance profiles to identify the principal computational procedures. This focused the parallelization efforts on less than five procedures within each program, each generally containing only one or two loop nests.

Doreen Cheng, having analyzed arc3d using FORGE, brought some of FORGE's profiling information to the workshop. It included such things as the number of iterations of each loop during the profiling run and the total amount of time spent in each loop. We agreed that such information would be very useful to have available in PED, especially with navigational support to automatically visit the most important loops or other annotated locations.

Although they found FORGE's profiling information useful, Katherine and Doreen disagreed with the assumption made by FORGE's code generator that parallelism should be introduced whenever possible. This assumption does not consider that the overhead associated with parallelizing a loop may significantly increase the execution time of the loop. They suggested the addition of either compile-time performance estimation or run-time tests, in order to introduce only profitable parallelism.

To effectively parallelize a program for a specific architecture, the new PED must have some facility for estimating the program's performance of a program on the target machine. Performance estimation requires that the user specify a target and it's performance model to be used in deciding which transformations to suggest to the user. Over the past year, we have been experimenting with performance estimation following this guideline, and we are currently incorporating it into PED.

5 Improvements and Additions: A Summary

As a result of the workshop, we have developed design goals for a new version of PED, which should be substantially more usable than the current tool. In addition to the completion of projects that are in progress, such as the complete dependence analysis implementation, the recognition of reductions and the production of a manual, we believe the following features should be included:

5.1 Improved Analysis

An important part of determining whether a loop can be run in parallel is identifying variables that can be made private to the loop body. The dependence analysis performed by PFC during the workshop identified private scalar variables using KILL analysis. The consensus of the workshop was that it is also necessary to identify private array variables. Other experience also suggests that KILL information on arrays, both interprocedural and intraprocedural, is important in parallelizing existing applications [EB91b, SH91].

Computing KILL information is more complex for arrays than for scalar values. In practice, some simple cases arise in loops that are easier to detect than the general case. For example, many initialization loops have the property that they define every element of an array `A`, and no element is referenced before its definition. In this case, the compiler can conclude that `A` is killed by the loop. If the loop either contains a procedure call or is called from within a loop where the KILL information would be useful, then the compiler needs interprocedural KILL information. We plan to explore methods for approximating both kinds of KILL sets, based on interprocedural regular section analysis.

Proper handling of reduction operations is very important in any parallelizing compiler or tool. We plan to add this capability to PED in two steps. First, PED will identify reduction operations by analyzing cycles in the dependence graph. All dependences that originate from a reduction will have a special indicator signifying that they can be removed by a reduction transformation. View filtering will be enhanced to allow these dependences to be masked out. Second, when the user requests a reduction transformation, PED will privatize the scalar involved in the reduction, and then generate a separate loop to combine the results of all the private copies. Alternatively, the reduction transformation will be applied implicitly when a user attempts to parallelize a loop whose only loop-carried (*i.e.*, parallel preventing) dependences arise from a reduction operation.

Currently being implemented in PED is a robust symbolic analysis system. This system will provide more precise dependence testing of array subscripts that contain symbolic expressions. The analysis will also be used in answering user queries on the value of variables and expressions.

Finally, regular section analysis will be made more precise. Not only will the analysis be enhanced by using improved algorithms and the symbolic analyzer, but the regular sections will be marked as exact or conservative. Exact regular sections will be very important in performing KILL analysis for arrays. In addition, interprocedural dependence analysis will be markedly improved by using more precise regular sections.

5.2 Guiding Program Transformation

In addition to dependence analysis and transformations, PED users need guidance to know where to focus their attention when introducing parallelism. Performance information, supplied by performance estimation, profiling or performance visualization, identifies computationally intensive loops. As described in section 4.6, most users relied upon sequential profile information to find these loops. Internally, PED will have access to a static performance estimation facility based on the *training set* methodology [KMM92, BFKK91]. Although the static estimate is less precise than using run-time profiling, static estimation eliminates the need for cooperation between the run-time system and the compiler, and the results are independent of input data sets. With performance estimates annotating the loops in the program representation, PED will direct the user to the important loops in the program.

Performance estimation can also guide the user in transformation selection. The current PED uses a paradigm that might be referred to as "power steering." The user selects a transformation and the system then performs that transformation correctly. The system provides no assistance to the user in the form of suggestions about which transformations to try. To correct this deficiency we plan to build a prototype automatic source-to-source parallelizer for shared-memory multiprocessors. Using performance estimation and other interprocedural information to guide transformation selection, the parallelizer will apply a combination of interprocedural transformations and parallelism-enhancing transformations to produce an initial parallel program [McK92, KMM91, KM92]. If the user requires additional performance they may interactively transform the code using information collected during automatic code generation. The compiler's loop-based algorithms and static performance estimates will also be available as program annotations to help find the best combination of transformations to apply.

In addition to performance estimation, we are also investigating incorporating *performance visualization* into ParaScope, to present profiling information to the user in a visual manner. We have incorporated one such package into ParaScope, upshot by Rusty Lusk at Argonne National Laboratories. We are also collaborating with Dan Reed at University of Illinois at Urbana-Champaign to interface with the Pablo system. These systems require that ParaScope provide an interface for users to indicate what code to visualize and then instrument the source code with calls that trace the events of interest at run-time.

5.3 Assertions

In the current PED, the user explicitly deletes dependences. This mechanism is awkward because it does not convey to PED the reasons behind the user's belief that a dependence does not exist. Two problems with this paradigm that became visible during the workshop are: redeletion of dependences after general editing and reanalysis, and tedious deletion of multiple dependences for the same reason.

A better system would be to have assertions that annotate the source code and eliminate dependences. Under this scheme, the user would be prompted to produce a reason for eliminating dependences, this reason could often be tested at run time. Further, by making the assertions part of the program, the system may derive the information necessary to recover dependence deletions following edits. The user is held responsible for maintaining the validity of the assertions.

Joseph Stein suggested that we incorporate the following user assertions in PED:

- variable_is_defined($\langle s \rangle$): scalar s is always defined by the current loop.

- variable_in_range ($\langle s, r\rangle$): range of a scalar s is r.
- array_is_defined($\langle a, r\rangle$): range r of array a is defined.
- array_is_undefined($\langle a, r\rangle$): initial values for array a in range r are not needed by the current loop.
- error_exit: placed before a loop exit, this suggests a path that should not be considered by analysis.

These assertions provide a higher level interface to dependence deletion. Through them the user can convey to PED the reason for deleting a dependence. This information may then be used automatically to refine other dependence tests, avoiding repetitive deletion by the user.

We intend to build a facility in ParaScope to enable users to make assertions that can be incorporated into dependence testing. The assertion language should include information about variables, such as value ranges, and will use the new symbolic analysis system being built for ParaScope. In addition, the language will allow a description of the interprocedural information used to support the annotation. This is discussed further in the following section.

5.4 Supporting Whole Programs

One of the distinguishing features of Parascope is the extensive interprocedural analysis information provided. We would like to further exploit this advantage by enabling PED to be invoked on a module in the context of a containing program. The power of this mechanism is illustrated with three examples.

Interprocedural navigation. PED's approach of associating all analysis information with its corresponding source code provides a convenient way of making changes when parallelism-inhibiting code is located. A particularly useful feature is the ability to navigate between endpoints of a dependence, a feature added in PED's new interface. However, it is still only possible to navigate within the body of the module.

With the program context and interprocedural information available, the tool can navigate dependences that reach other procedures. This feature allows the user to see the actual statements causing the dependence, greatly simplifying code understanding and changes. This same approach could be used in displaying other interprocedural information, such as the previous definition or next use of a variable.

Interprocedural transformations. ParaScope's existing interprocedural compilation system provides inline substitution as a source transformation to replace a call with the body of the invoked procedure [CHT91]. An prototype version of ParaScope contains an implementation of procedure cloning, another source transformation that replicates a procedure and reassociates its callers in order to tailor optimizations on the procedure body to different calling environments [CHK92]. Recent research has established the usefulness of interprocedural transformations other than inlining and cloning [HKM91, McK92]. In particular, moving a loop into or out of a procedure invocation may enable application of parallelizing transformations such as loop interchange and loop fusion.

Since all of these transformations cross procedure boundaries, it is difficult for the user to apply them. The program context provides the necessary information to apply the transformations automatically. Moreover, interprocedural information can indicate whether these transformations will enable further parallelization; the system can often detect what transformations are applicable and guide the programmer in their use.

Reanalysis after global changes. PED's current incremental analysis is designed to support updates in response to changes during a single editing session. This approach does not address the more difficult issue of updating after global program changes. The user might make an editing decision based on interprocedural information when designating a loop as parallel, adding assertions, or applying transformations.

The compiler can sometimes verify a loop designated as parallel if the sequential version contains no loop-carried dependences. However, in most cases it is impossible for the system to understand what interprocedural information might have been used by the programmer. For this purpose, we could include in the assertion language assumptions about interprocedural information. In this way, the user informs the system what interprocedural information is important, and the system warns the user when this information changes. In determining the safety of program transformations, the system may rely on interprocedural information to make decisions. The system should record any interprocedural information used to prove transformation safety. Then on a subsequent editing session, the system compares the current interprocedural information with that stored from the previous session and warns of any changes that might have invalidated transformations. The programmer is responsible for determining the effects of these changes on the meaning of their program.

6 Related Work

The interactive tool FORGE also assists conversion of a sequential Fortran program into a parallel version. Distributed by Pacific Sierra, the tool generates a run time profile and then performs dependence analysis. Using the profile information, the user assists the compiler by removing false dependences and parallelizing loops. In the resulting parallel version, all parallel loops are converted to DO ALL loops.

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 [SG90], PAT [SA88], MIMDIZER [Hil90], and SUPERB [ZBG88], 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 [GGJ+89].

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 [SAS90]. Rather than analyzing the effects of existing synchronization, PAT can instead insert synchronization to preserve specific dependences. 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 [KZBG88].

7 Conclusions

This paper has reported experiences using the ParaScope Editor, an interactive tool for constructing shared-memory parallel programs. As a result of these efforts, we learned what features provided by the tool were important and exposed further areas of improvement.

Several interesting research issues came to light during this process. In particular, we learned that making assertions about variable values is more natural and useful than explicit dependence deletion. Second, transformations are difficult to use without more guidance about when they are applicable. Third, in addition to the extensive compiler analysis currently in ParaScope, array kill analysis and reduction recognition also need to be incorporated. The users also wanted access to more of the compiler analysis, presented in the context of the corresponding source code.

An interactive programming tool such as PED can be a powerful program development aid if it is able to filter the large volume of information computed by the compiler to a form that a knowledgeable user can exploit. Although the ParaScope Editor focuses on optimizing programs for shared-memory parallel architectures, the notion of exposing compiler analysis to the programmer can enhance performance tuning for a variety of architectures.

Acknowledgments

We are very grateful to all the workshop attendees and the other researchers who participated in evaluating the ParaScope Editor. We would also like to acknowledge the many researchers who made PED the useful research platform that it is today. We would especially like to thank Scott Warren for his contributions to the new PED user interface and Vasanth Bala, David Callahan, Paul Havlak, Carl Rosene and Chau-Wen Tseng for their efforts in the design and implementation of this tool. We also appreciate the valuable technical support provided by Joel Castellanos. This research was supported by the Center for Research on Parallel Computation (a National Science Foundation Science and Technology Center), DARPA, the state of Texas, IBM, Cray Research, and by a DARPA/NASA Research Assistantship in Parallel Processing, administered by the Institute for Advanced Computer Studies, University of Maryland.

References

[ABC+87] 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.

[AC72] F. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, 1972.

[ACK87] 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.

[AK84] 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.

[AK87] 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.

[ASU86] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[BBC+88] V. Balasundaram, D. Bäumgartner, D. Callahan, K. Kennedy, and J. Subhlok. PTOOL: A system for static analysis of parallelism in programs. Technical Report TR88-71, Dept. of Computer Science, Rice University, 1988.

[BFKK91] 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.

[BKK+89] 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.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[CCK90] 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.

[CCKT86] 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.

[CHK92] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, April 1992.

[CHT91] K. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[CK87] 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.

[CKT86a] 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.

[CKT+86b] K. Cooper, K. Kennedy, L. Torczon, A. Weingarten, and M. Wolcott. Editing and compiling whole programs. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, CA, December 1986.

[CP91] D. Cheng and D. Pase. An evaluation of automatic and interactive parallel programming tools. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[Cra] Cray fortran reference manual. Technical report.

[EB91a] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[EB91b] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.

[EE68] D. C. Engelbart and W. K. English. A research center for augmenting human intellect. In *Proceedings of AFIPS 1968 Fall Joint Computer Conference*, pages 395–410, December 1968.

[FKMW90] 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.

[For90] The forge user's guide version 7.01. Technical report, December 1990.

[GGJ+89] 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.

[GKT91] 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.

[Hal91] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, April 1991.

[HHLS90] 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.

[Hil90] R. Hill. MIMDizer: A new tool for parallelization. *Supercomputing Review*, 3(4):26–28, April 1990.

[HK90] P. Havlak and K. Kennedy. Experience with interprocedural analysis of array side effects. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HKM91] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[KKLW80] 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.

[KKLW84] 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.

[KM90] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[KM92] K. Kennedy and K. S. McKinley. Optimizing for parallelism and memory hierarchy. Technical Report TR92-175, Dept. of Computer Science, Rice University, January 1992.

[KMM91] K. Kennedy, N. McIntosh, and K. S. McKinley. Static performance estimation. Technical Report TR91-174, Dept. of Computer Science, Rice University, December 1991.

[KMM92] K. Kennedy, N. McIntosh, and K. S. McKinley. Static performance estimation in a parallelizing compiler. Technical report, Dept. of Computer Science, Rice University, 1992. available May 1992.

[KMT91a] 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.

[KMT91b] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[KZBG88] U. Kremer, H. Zima, H.-J. Bast, and M. Gerndt. Advanced tools and techniques for automatic parallelization. *Parallel Computing*, 7:387–393, 1988.

[Lov77] D. Loveman. Program improvement by source-to-source transformations. *Journal of the ACM*, 17(2):121–145, January 1977.

[McK92] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Rice University, April 1992.

[SA88] 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.

[SAS90] 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.

[SG90] 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.

[SH91] 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.

[Tor85] L. Torczon. *Compilation Dependences in an Ambitious Optimizing Compiler*. PhD thesis, Rice University, May 1985.

[Wol86] M. J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, August 1986.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.