ProDev™ WorkShop: ProMP User's Guide

007–2603–007

COPYRIGHT

© 1993, 1999, 2001 – 2002 Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND

The electronic (software) version of this document was developed at private expense; if acquired under an agreement with the USA government or any contractor thereto, it is acquired as "commercial computer software" subject to the provisions of its applicable license agreement, as specified in (a) 48 CFR 12.212 of the FAR; or, if acquired for Department of Defense units, (b) 48 CFR 227-7202 of the DoD FAR Supplement; or sections succeeding thereto. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy 2E, Mountain View, CA 94043-1351.

TRADEMARKS AND ATTRIBUTIONS

Silicon Graphics, SGI, the SGI logo, IRIX, and Origin are registered trademarks and ProDev and OpenMP are trademarks of Silicon Graphics, Inc.

MIPSpro is a trademark of MIPS Technologies, Inc., and is used under license by Silicon Graphics, Inc. ToolTalk is a trademark of Sun Microsystems, Inc. UNIX and the X device are registered trademarks of The Open Group in the United States and other countries.

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

Record of Revision

Version	Description
	1993 Original Printing.
2.9	April 1999 This release adds support for programs written in C and Fortran 90.
006	November 2001 This version supports the ProMP 2.9 release.
007	September 2002 This version supports the ProMP 2.9.2 release.

Contents

About This Guide . . . . **xxiii**
Related Publications xxiii
Obtaining Publications xxiv
Conventions xxv
Reader Comments xxv

1. Getting Started With ProMP **1**
Compiling a Program for ProMP Use 2
Generating Other Reports 2
OpenMP and PCF Directive Support 3
Reading Files With the Parallel Analyzer View 3

2. Tutorial: Examining Loops for FORTRAN 77 . . . . **5**
Compiling the Sample Code 6
Using the Main Window 7
Using the Loop List Display 9
Loop List Information Fields 10
Loop List Icons 10
Sorting and Filtering the Loop List 11
Sorting the Loop List 11
Filtering the Loop List 11
Viewing Detailed Information About Code and Loops 14
Using the Transformed Loops View 20
Transformed Loops View Description 21

Selecting Transformed Loops 22
Examples of Simple Loops 24
Simple Parallel Loop 24
Serial Loop 25
Explicitly Parallelized Loop 25
Fused Loops 27
Loop That Is Eliminated 28
Examining Loops With Obstacles to Parallelization 28
Obstacles to Parallelization: Carried Data Dependence 29
Unparallelizable Carried Data Dependence 29
Parallelizable Carried Data Dependence 29
Multi-line Data Dependence 30
Reductions 31
Obstacles to Parallelization: I/O Operations 32
Obstacles to Parallelization: Unstructured Control Flow 32
Obstacles to Parallelization: Subroutine Calls 33
Unparallelizable Loop With a Subroutine Call 33
Parallelizable Loop With a Subroutine Call 34
Obstacles to Parallelization: Permutation Vectors 34
Unparallelizable Loop With a Permutation Vector 34
Parallelizable Loop With a Permutation Vector 34
Obstacles to Parallelization Messages 35
Examining Nested Loops 39
Doubly Nested Loop 39
Interchanged Doubly Nested Loop 40
Triply Nested Loop With an Interchange 40
Modifying Source Files and Compiling 41

Making Changes 41
Adding C$OMP PARALLEL DO Directives and Clauses 42
Adding New Assertions or Directives With the Operations Menu 45
Deleting Assertions or Directives 46
Applying Requested Changes 46
Viewing Changes With gdiff 47
Modifying the Source File Further 48
Updating the Source File 49
Examining the Modified Source File 50
Added Assertion 50
Deleted Assertion 50
Examples Using OpenMP Directives 50
Explicitly Parallelized Loops: C$OMP DO 51
Loops With Barriers: C$OMP BARRIER 53
Critical Sections: C$OMP CRITICAL 55
Single-Process Sections: C$OMP SINGLE 55
Parallel Sections: C$OMP SECTIONS 55
Examples Using Data Distribution Directives 56
Distributed Arrays: C$SGI DISTRIBUTE 56
Distributed and Reshaped Arrays: C$SGI DISTRIBUTE_RESHAPE 58
Prefetching Data From Cache: C*$* PREFETCH_REF 59
Exiting From the omp_demo.f Sample Session 60
3. Tutorial: Examining Loops for Fortran 90 Code 61
Compiling the Sample Code 61
Demonstrating Array Statement Transformations 62
Transforming an Array Statement into a DO Loop 62

Transforming an Array Statement in Nested DO Loops 63
Transforming an Array Statement into a Subroutine 65
Exiting From the Session 66

4. Tutorial: Examining Loops for C Code 69

Compiling the Sample Code 70
Examples of Simple Loops 70
Simple Parallel Loop 71
Serial Loop 71
Explicitly Parallelized Loop 72
Fused Loops 74
Loop That Is Eliminated 74
Examining Loops With Obstacles to Parallelization 74
Obstacles to Parallelization: Carried Data Dependence 75
Unparallelizable Carried Data Dependence 75
Parallelizable Carried Data Dependence 77
Multi-line Data Dependence 78
Reductions 78
Obstacles to Parallelization: I/O Operations 79
Obstacles to Parallelization: Function Calls 79
Obstacles to Parallelization: Permutation Vectors 80
Unparallelizable Loop With a Permutation Vector 80
Parallelizable Loop With a Permutation Vector 80
Examining Nested Loops 80
Nested Loop 81
Doubly Nested Loop 81
Triple Nested Loop 81

Modifying Source Files and Compiling 82
Making Changes 83
Adding `#pragma omp parallel for` Directives and Clauses 83
Adding New Assertions or Directives With the Operations Menu 85
Deleting Assertions or Directives 87
Updating the Source File 89
Examining the Modified Source File 89
Added Assertion 89
Deleted Assertion 89
Examples Using OpenMP Directives 90
Explicitly Parallelized Loops: `#pragma omp for` 90
Loops With Barriers: `#pragma omp barrier` 91
Critical Sections: `#pragma omp critical` 92
Single-Process Sections: `#pragma omp single` 92
Parallel Sections: `#pragma omp sections` 93
Examples Using Data Distribution Directives 93
Distributed Arrays: `#pragma distribute` 94
Distributed and Reshaped Arrays: `#pragma distribute_reshape` 96
Prefetching Data From Cache: `#pragma prefetch_ref` 97
Exiting From the Sample Session 98

5. Using WorkShop With Parallel Analyzer View 99
Starting the Parallel Analyzer View 99
Using the Parallel Analyzer With Performance Data 100
Effect of Performance Data on the Source View 102
Sorting the Loop List by Performance Cost 102
Exiting From the linpackd Sample Session 104

6. Parallel Analyzer View Reference 107
Parallel Analyzer View Main Window 107
Parallel Analyzer View Menu Bar 109
Admin Menu 110
Icon Legend... Option 112
Launch Tool Submenu 112
Project Submenu 113
Views Menu 116
Fileset Menu 116
Update Menu 117
Configuration Menu 118
Operations Menu 119
Help Menu 122
Keyboard Shortcuts 123
Loop List Display 123
Status and Performance Experiment Lines 124
Loop List 125
Loop Display Controls 126
Search Loop List Field 127
Sort Option Button 127
Show Loop Types Option Button 128
Filtering Option Button 128
Loop Display Buttons 129
Loop Information Display 129
Highlight Buttons 130
Loop Parallelization Controls in the Loop Information Display 131
Loop Parallelization Status Option Button 131

MP Scheduling Option Button: Directives for All Loops 133
MP Chunk Size Field 134
Obstacles to Parallelization Information Block 134
Assertions and Directives Information Blocks 135
Compiler Messages 136
Views Menu Options 136
Parallelization Control View 136
Common Features of the Parallelization Control View 137
C$OMP PARALLEL DO and C$OMP DO Directive Information 138
MP Scheduling Option Button: Clauses for One Loop 142
Variable List Option Buttons 142
Variable List Storage Labeling 143
Transformed Loops View 144
PFA Analysis Parameters View 145
Subroutines and Files View 146
Loop Display Control Button Views 148
Source View and Parallel Analyzer View - Transformed Source 148

Appendix A. Examining Loops Containing PCF Directives 151
Setting Up the dummy.f Sample Session 151
Compiling the Sample Code 151
Starting the Parallel Analyzer View 152
Examples Using PCF Directives 152
Explicitly Parallelized Loops: C$PAR PDO 153
Loops With Barriers: C$PAR BARRIER 154
Critical Sections: C$PAR CRITICAL SECTION 156
Single-Process Sections: C$PAR SINGLE PROCESS 157
Parallel Sections: C$PAR PSECTIONS 157

Exiting From the dummy.f Sample Session 158

Index 159

Figures

Figure 2-1 Parallel Analyzer View Main Window 9
Figure 2-2 Loop Display Controls 11
Figure 2-3 Show Loop Types Option Button 12
Figure 2-4 Filtering Option Button 13
Figure 2-5 Subroutines and Files View 14
Figure 2-6 Source View 15
Figure 2-7 Transformed Source Window 17
Figure 2-8 Loop Information Display Without Performance Data 19
Figure 2-9 Transformed Loops View for Loop Olid 1 21
Figure 2-10 Transformed Loops in Source Windows 23
Figure 2-11 Explicitly Parallelized Loop 26
Figure 2-12 Source View of C$OMP PARALLEL DO Directive 27
Figure 2-13 Parallelizable Data Dependence 30
Figure 2-14 Highlighting on Multiple Lines 31
Figure 2-15 Requesting a C$OMP PARALLEL DO Directive 42
Figure 2-16 Parallelization Control View After Choosing `C$OMP PARALLEL DO...` . . 44
Figure 2-17 Effect of Changes on the Loop List 45
Figure 2-18 Deleting an Assertion 46
Figure 2-19 Run gdiff After Update 48
Figure 2-20 Build View of Build Manager 49
Figure 2-21 Loops Explicitly Parallelized Using C$OMP DO 52
Figure 2-22 Loops Using C$OMP BARRIER Synchronization 54
Figure 2-23 `C$SGI DISTRIBUTE` Directive and Text Field 57

Figure 3-1 Array Statement into `DO` Loop 63
Figure 3-2 Loop 22 64
Figure 3-3 Loop 23 65
Figure 3-4 Array Statement into a Subroutine 66
Figure 4-1 Explicitly Parallelized Loop 73
Figure 4-2 Obstacles to Parallelization 77
Figure 4-3 Creating a Parallel Directive 84
Figure 4-4 Parallelization Control View 85
Figure 4-5 Adding an Assertion 86
Figure 4-6 Deleting an Assertion 88
Figure 4-7 Loops Explicitly Parallelized Using `#pragma omp for` 91
Figure 4-8 `#pragma distribute` Directive and Text Field 95
Figure 5-1 Parallel Analyzer View — Performance Data Loaded 101
Figure 5-2 Source View for Performance Experiment 102
Figure 5-3 Sort by Performance Cost 103
Figure 5-4 Loop Information Display With Performance Data 104
Figure 6-1 Parallel Analyzer View Main Window 109
Figure 6-2 Output Text File Selection Dialog 111
Figure 6-3 Project Submenu and Windows 115
Figure 6-4 Operations Menu and Submenus 119
Figure 6-5 Loop List Display 124
Figure 6-6 Loop List with Column Headings 125
Figure 6-7 Loop Display Controls 127
Figure 6-8 Loop Information Display 130
Figure 6-9 Loop Parallelization Controls 131
Figure 6-10 MP Chunk Size Field Changed 134

Figure 6-11 Obstacles to Parallelization Block 135
Figure 6-12 Assertion Information Block and Options (n32 and n64 Compilation) 135
Figure 6-13 Parallelization Control View 137
Figure 6-14 Parallelization Control View With C$OMP PARALLEL DO Directive 139
Figure 6-15 Parallelization Control View With C$OMP DO Directive 140
Figure 6-16 Transformed Loops View 144
Figure 6-17 PFA Analysis Parameters View 146
Figure 6-18 Subroutines and Files View 147
Figure 6-19 Original and Transformed Source Windows 149
Figure A-1 Explicitly Parallelized Loops Using C$PAR PDO 154
Figure A-2 Loops Using C$PAR BARRIER Synchronization 156

Tables

Table 6-1 Parallel Analyzer View Keyboard Shortcuts 123

Examples

Example 2-1 Simple Parallel Loop 25

Example 2-2 Serial Loop 25

Example 2-3 Fused Loop 27

Example 2-4 Eliminated Loop 28

Example 2-5 Unparallelizable Carried Data Dependence 29

Example 2-6 Parallelizable Carried Data Dependence 30

Example 2-7 Multi-line Data Dependence 31

Example 2-8 Reduction 32

Example 2-9 Input/Output Operation 32

Example 2-10 Unstructured Control Flow 33

Example 2-11 Unparallelizable Loop With Subroutine Call 33

Example 2-12 Parallelizable Loop With Subroutine Call 34

Example 2-13 Unparallelizable Loop With Permutation Vector 34

Example 2-14 Parallelizable Loop With Permutation Vector 35

Example 2-15 Doubly Nested Loop 39

Example 2-16 Interchanged Doubly Nested Loop 40

Example 2-17 Triply Nested Loop With Interchange 40

Example 2-18 Explicitly Parallelized Loop Using `C$OMP DO` 51

Example 2-19 Loops Using `C$OMP BARRIER` 53

Example 2-20 Critical Section Using `C$OMP CRITICAL` 55

Example 2-21 Single-Process Section Using `C$OMP SINGLE` 55

Example 2-22 Parallel Sections Using `C$OMP SECTIONS` 56

Example 2-23 Distributed Array Using `C$SGI DISTRIBUTE` 58

Example 2-24 Distributed and Reshaped Array Using C$SGI DISTRIBUTE_RESHAPE . 59
Example 2-25 Prefetching Data From Cache Using C*$* PREFETCH_REF 60
Example 4-1 C: simple parallel loop 71
Example 4-2 C: serial loop 72
Example 4-3 C: explicitly parallelized loop 72
Example 4-4 C: fused loops 74
Example 4-5 C: eliminated loop 74
Example 4-6 C: nested loop 81
Example 4-7 C: doubly nested loop 81
Example 4-8 C: triple nested loop 82
Example 4-9 C: explicitly parallelized loops 90
Example 4-10 C: loops with barriers 92
Example 4-11 C: critical sections 92
Example 4-12 C: single process sections 93
Example 4-13 C: parallel sections 93
Example 4-14 C: distributed arrays 96
Example 4-15 C: distributed and reshaped arrays 97
Example 4-16 C: prefetching data from cache 97
Example 6-1 Explicitly Parallelized Loop Using C$PAR PDO 153
Example 6-2 Loops Using C$PAR BARRIER 155
Example 6-3 Critical Section Using C$PAR CRITICAL SECTION 157
Example 6-4 Single-Process Section Using C$PAR SINGLE PROCESS 157
Example 6-5 Parallel Section Using C$PAR PSECTIONS 158

Procedures

Procedure 2-1 Filtering the Loop List by Parallelization State 12

Procedure 2-2 Filtering the Loop List by Loop Origin 13

Procedure 2-3 Sorting by Subroutine 13

Procedure 2-4 Viewing Original Source 15

Procedure 2-5 Viewing Transformed Source 16

Procedure 2-6 Navigating the Loop List 17

Procedure 2-7 Selecting a Loop for Analysis 18

Procedure 2-8 Using the Loop Information Display 19

About This Guide

This publication documents the ProDev WorkShop ProMP product, version 2.9.2, which runs on IRIX systems. The ProMP product is a companion to the WorkShop suite of tools. It is used to analyze a program that has been parallelized; before using ProMP, you must first compile your program with an auto-parallelizing option. After the compiler generates an output file, you can use ProMP to analyze that file.

ProMP is integrated with WorkShop to let you examine a program's loops in conjunction with a performance experiment on either a single processor or multiprocessor run.

Note: This product was formerly called WorkShop Pro MPF.

The following topics are discussed in this guide:

- Chapter 1, "Getting Started With ProMP", page 1, describes (in general) how to use ProMP.
- Chapter 2, "Tutorial: Examining Loops for FORTRAN 77", page 5, describes how to analyze FORTRAN 77 loops.
- Chapter 3, "Tutorial: Examining Loops for Fortran 90 Code", page 61, describes how to analyze Fortran 90 loops.
- Chapter 4, "Tutorial: Examining Loops for C Code", page 69, describes how to analyze C loops.
- Chapter 5, "Using WorkShop With Parallel Analyzer View", page 99, describes how ProMP can be used with other WorkShop tools.
- Chapter 6, "Parallel Analyzer View Reference", page 107, is a reference guide to the ProMP product.

Related Publications

The following documents contain additional information that may be helpful:

- *SpeedShop User's Guide*

- *C Language Reference Manual*
- *MIPSpro C++ Programmer's Guide*
- *MIPSpro Fortran 90 Commands and Directives Reference Manual*
- *MIPSpro Fortran Language Reference Manual, Volume 1*
- *MIPSpro Fortran Language Reference Manual, Volume 2*
- *MIPSpro Fortran Language Reference Manual, Volume 3*
- *MIPSpro Fortran 77 Language Reference Manual*
- *MIPSpro Fortran 77 Programmer's Guide*
- *ProDev WorkShop: ProMP User's Guide*
- *ProDev WorkShop: Debugger User's Guide*
- *ProDev WorkShop: Debugger Reference Manual*
- *ProDev WorkShop: Static Analyzer User's Guide*
- *ProDev WorkShop: Overview*

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at: `http://docs.sgi.com`. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.
- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing `man` *title* on a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
...	Ellipses indicate that a preceding element can be repeated.
GUI	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, menus, toolbars, icons, buttons, boxes, fields, and lists.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  `http://docs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Parkway, M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Chapter 1

Getting Started With ProMP

This chapter describes how to run the ProDev WorkShop ProMP parallel analyzer. It contains the following sections:

- "Compiling a Program for ProMP Use", page 2.
- "Reading Files With the Parallel Analyzer View", page 3.

Note: This product was formerly called WorkShop Pro MPF.

ProMP requires the following software versions (or later versions):

- IRIX system software version 6.2
- MIPSpro FORTRAN 77, release 7.2.1
- MIPSpro Fortran 90, release 7.3
- MIPSpro C, release 7.3
- ToolTalk 1.1
- WorkShop 2.0

To determine what software is installed on your system, enter the following at the shell prompt:

```
% versions
```

The process of using the parallel analyzer involves two steps:

1. Compiling a program with the necessary options. See "Compiling a Program for ProMP Use", page 2 for details.
2. Reading the compiled files. See "Reading Files With the Parallel Analyzer View", page 3 for details.

For a more detailed introduction to the parallel analyzer view, follow one of the following tutorials:

- Chapter 2, "Tutorial: Examining Loops for FORTRAN 77", page 5.
- Chapter 3, "Tutorial: Examining Loops for Fortran 90 Code", page 61.

- Chapter 4, "Tutorial: Examining Loops for C Code", page 69.
- Chapter 5, "Using WorkShop With Parallel Analyzer View", page 99.
- Appendix A, "Examining Loops Containing PCF Directives", page 151.

Compiling a Program for ProMP Use

Before starting the parallel analyzer to analyze your source (in this case, Fortran source), run your compiler with the appropriate auto-parallelizing option. For the tutorials presented in subsequent chapters, makefiles are provided. You can adapt these to your specific source or enter one of the following commands:

```
% f90 -apo keep -O3 sourcefile.f
% f77 -apo keep -O3 sourcefile.f
```

The compiler generates its usual output files and an analysis file (`sourcefile.anl`), which the parallel analyzer reads.

The command-line options have the following effects:

- `-apo keep`: saves a `.anl` file, which has necessary information for the parallel analyzer view.
- `-O3`: sets the compiler for aggressive optimization. The optimization focuses on maximizing code performance, even if that requires extending the compile time or relaxing language rules.

See the *MIPSpro Fortran 77 Programmer's Guide*, *MIPSpro Fortran 90 Commands and Directives Reference Manual* and the `f90`(1) or `f77`(1) man pages for more information.

Note: The `cvpav` command assumes that the `-apo keep` option was used on each of the Fortran source files named in a single executable or file specifying several executables. If this is not the case, a warning message is posted, and the unprocessed files are marked by an error icon within the parallel analyzer's subroutines and files view, see "Subroutines and Files View", page 146.

Generating Other Reports

While they are not part of the parallel analyzer view, other parallelization reports can be generated using the following command-line options:

- `-apo list`: produces a `.l` file, a listing of those parts of the program that can run in parallel and those that cannot.
- `-mplist`: generates the equivalent parallelized program in a `.w2f.f` file.

These reports are text files that can be used for analysis.

OpenMP and PCF Directive Support

The MIPSpro Fortran compilers support OpenMP directives, unless you are compiling with the `-o32` option. If you put OpenMP directives in your `o32` code, they are treated as comments rather than being interpreted. For more information on OpenMP directives, see the documentation for your compiler system, or the OpenMP Architecture Review Board web site at the following URL: `http://www.openmp.org/`.

Although using OpenMP directives is recommended, the compilers still support PCF directives.

Reading Files With the Parallel Analyzer View

You can run the parallel analyzer view on any of the following objects:

- A source file
- An executable
- A list of files

To run the parallel analyzer view for one of these cases, enter one of the following commands:

```
% cvpav -f sourcefile.f
% cvpav -e executable
% cvpav -F fileset-file
```

The `cvpav` command reads information from all source files compiled into the application.

The parallel analyzer view has several other command line options, as well as several X Window System resources that you can set. See the `cvpav`(1) man page for more information.

Tutorial: Examining Loops for FORTRAN 77

This chapter presents an interactive sample session with the Parallel Analyzer View. The session demonstrates basic features of ProMP and illustrates aspects of parallelization and of the MIPSpro FORTRAN 77 compiler.

The sample session illustrates how to display code and basic loop information, as well as how to examine specific loops and apply directives and assertions.

The topics are introduced in this chapter by going through the process of starting the Parallel Analyzer View and stepping through the loops and routines in the sample code. The chapter is most useful if you perform the operations as they are described.

For more details about the Parallel Analyzer View interface, see Chapter 6, "Parallel Analyzer View Reference", page 107.

To use the sample sessions, note the following:

- `/usr/demos/ProMP` is the demonstration directory
- `ProMP.sw.demos` must be installed

The sample session discussed in this chapter uses the following source files in the directory `/usr/demos/ProMP/omp_tutorial`:

- `omp_demo.f_orig`
- `omp_dirs.f_orig`
- `omp_reshape.f_orig`
- `omp_dist.f_orig`

The source files contain many `DO` loops, each of which illustrates an aspect of the parallelization process.

The `/usr/demos/ProMP/omp_tutorial` directory also includes a `Makefile` to compile the source files.

The following topics are covered in this tutorial:

- "Compiling the Sample Code", page 6
- "Using the Main Window", page 7

- "Using the Loop List Display", page 9
- "Sorting and Filtering the Loop List", page 11
- "Viewing Detailed Information About Code and Loops", page 14
- "Examples of Simple Loops", page 24
- "Examining Loops With Obstacles to Parallelization", page 28
- "Examining Nested Loops", page 39
- "Modifying Source Files and Compiling", page 41
- "Examples Using OpenMP Directives", page 50
- "Examples Using Data Distribution Directives", page 56
- "Exiting From the omp_demo.f Sample Session", page 60

Compiling the Sample Code

Prepare for the session by opening a shell window and entering the following:

```
% cd /usr/demos/ProMP/omp_tutorial
% make
```

These commands create the following files:

- omp_demo.f: a copy of the demonstration program created by combining the *.f_orig files, which you can view with the Parallel Analyzer View (or any text editor), and print
- omp_demo.m: a transformed source file, which you can view with the Parallel Analyzer View, and print
- omp_demo.l: a listing file
- omp_demo.anl: an analysis file used by the Parallel Analyzer View

After you have the appropriate files from the compiler, start the session by entering the cvpav(1) command. which opens the main window of the Parallel Analyzer View loaded with the sample file data (see Figure 2-1, page 9):

```
% cvpav -f omp_demo.f
```

If at any time during the tutorial you should want to restart from the beginning, do the following:

- Quit the Parallel Analyzer View by choosing **Admin > Exit** from the Parallel Analyzer View menu bar.
- Clean up the tutorial directory by entering the following command:

```
% make clean
```

This removes all of the generated files; you can begin again by using the `make` command.

Using the Main Window

The Parallel Analyzer View main window contains the following components, as shown in Figure 2-1, page 9:

- Main menu bar, which includes the following menus:
  - **Admin**
  - **Views**
  - **Fileset**
  - **Update**
  - **Configuration**
  - **Operations**
  - **Help**
- List of loops and control structures, which consists of the following:
  - Status information
  - Performance experiment information
  - Loop list
- Loop display controls, which are the following:
  - Search editable text field

- Sort option button (**Sort in Source Order**)
- Show loop types option button (**Show All Loop Types**)
- Filtering option button (**No Filtering**)
- **Source** and **Transformed Source** control buttons
- **Next Loop** and **Previous Loop** navigation buttons

- Loop information display

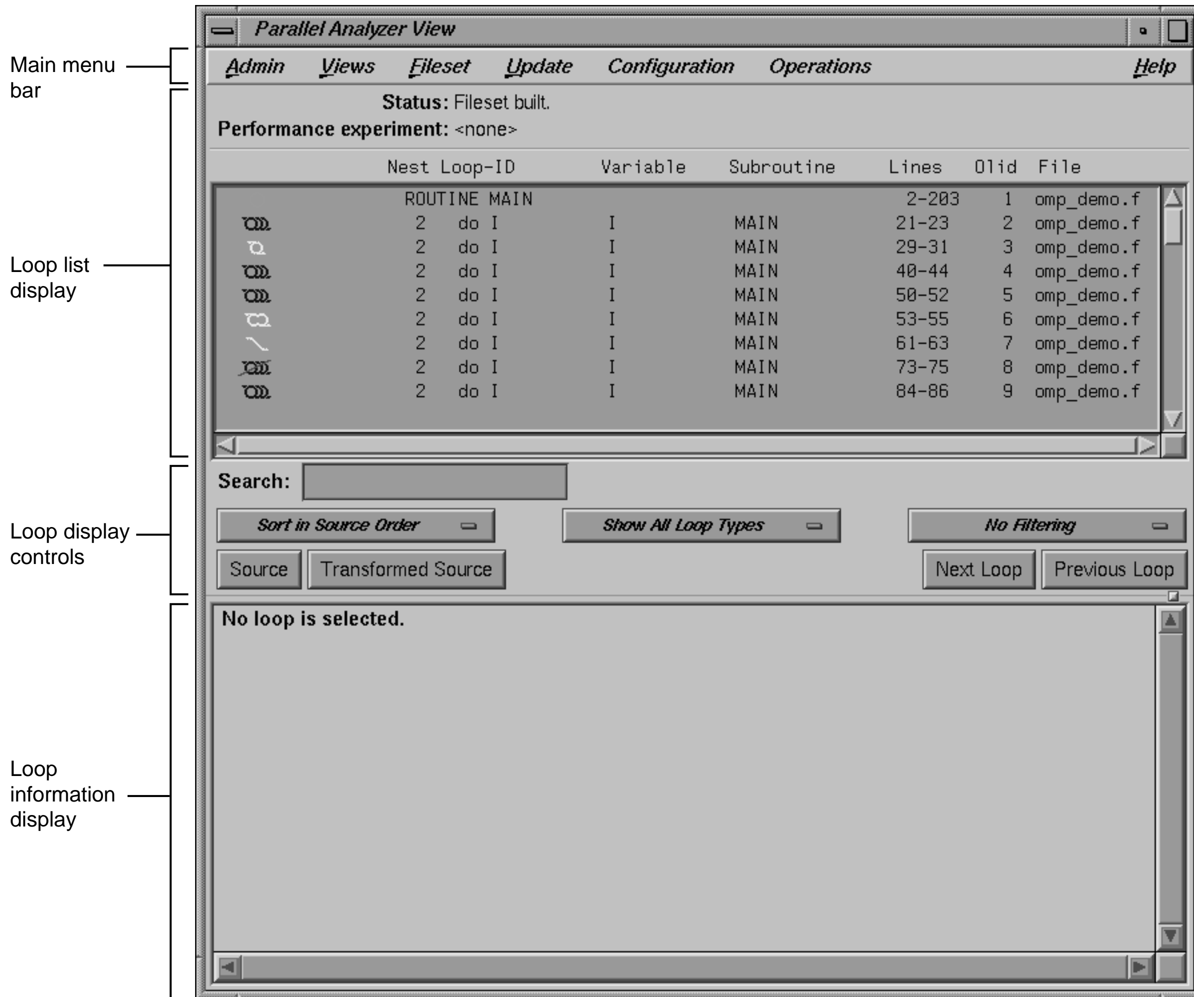


Figure 2-1 Parallel Analyzer View Main Window

Using the Loop List Display

The loop list display summarizes a program's structure and provides access to source code. Each line in the loop list contains an icon and a sequence of information fields about loops and routines in the program.

Loop List Information Fields

Each loop list entry contains the following fields:

- The icon symbolizes the status of the subroutine or loop.
- The nest field shows the nesting level for the loop.
- The loop-ID gives a description of the loop.
- The variable field indicates the loop index variable.
- The subroutine field contains the name of the subroutine in which the loop is located.
- The lines field displays the lines in the source code in which the loop is located.
- The Olid is the original loop ID, an internal identifier for the loop created by the compiler.
- The file field names the file in which the loop is located.

Loop List Icons

The icon at the start of each line summarizes briefly the following information:

- If the line refers to a subroutine or function.
- The parallelization status of the loop.
- OpenMP control structures.

To understand the meaning of the various icons, choose **Admin** > **Icon Legend...** .

To resize the loop list display and provide more room in the main window for loop information, use the adjustment button. The adjustment button is a small square below the **Previous Loop** button and just above the vertical scroll bar on the right side of the loop information display. In many of the following figures, the loop list is resized from its original configuration.

The loop list **Search** field allows you to find occurrences of any character in the loop list. You can search for subroutine names, a phrase (such as `parallel` or `region`), or Olid numbers. (See Figure 2-2, page 11.)

The search is not case sensitive; simply key in the string. To find subsequent occurrences of the same string, press the **Enter** key.

Sorting and Filtering the Loop List

This section describes the loop display control option buttons. They allow you to sort and filter the loop list, and so focus your attention on particular pieces of your code. As shown in Figure 2-1, page 9, the buttons are located in the main window, below the loop list display. Figure 2-2, page 11, shows all of the loop display controls.

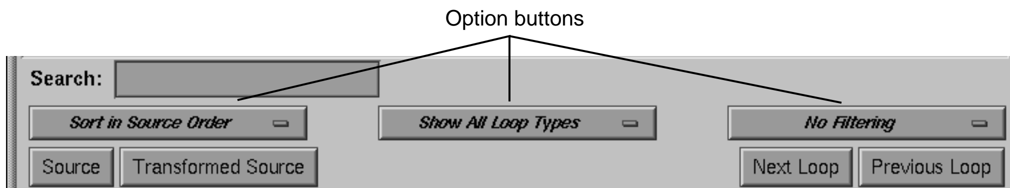


Figure 2-2 Loop Display Controls

Sorting the Loop List

You can sort the loop list either in the order of the source code, or by performance cost (if you are running the WorkShop performance analyzer). You usually control sorting with the sort option button, the left-most button below the **Search** field.

When loops are sorted in source order, the loop-ID is indented according to the nesting level of the loop. For the demonstration program, only the last several loops are nested, so you have to scroll down to see indented loop-IDs. For example, scroll down the loop list until you find a loop whose nest value, as shown in the loop list, is greater than 2.

When loops are sorted by performance cost, using **Sort by Perf.Cost** option button, the list is not indented. The sorting option is grayed out in the example because the performance analyzer is not currently running.

Filtering the Loop List

You may want to look at only some of the loops in a large program. The loop list can be filtered according to parallelization status and loop origin. The filter parameters are controlled by the two option buttons to the right of the sort option button.

Procedure 2-1 Filtering the Loop List by Parallelization State

Filtering according to parallelization state allows you to focus, for example, on loops that were not automatically parallelized by the compiler but that might still run concurrently if you add appropriate directives.

Filtering is controlled by the **Show Loop Types** option button centered below the loop list; the default setting is **Show All Loop Types**, as shown in Figure 2-3, page 12.

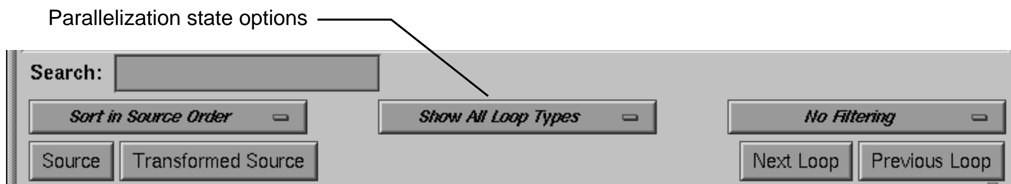


Figure 2-3 Show Loop Types Option Button

You can select according to the following states of loop parallelization and processing (which are displayed when you click the show loop types option button):

- **Show All Loop Types**, the default.
- **Show Unparallelizable Loops** displays loops that are running serially because they could not be parallelized.
- **Show Parallelized Loops** displays loops that were parallelized.
- **Show Serial Loops** displays loops that are best run serially.
- **Show PCF Directives** displays loops containing PCF directives.
- **Show OMP Directives** displays loops containing OpenMP directives.
- **Show Modified Loops** displays loops for which modifications have been requested.

The second, third, and fourth categories correspond to parallelization icons in the **Icon Legend...** window. Making modifications to loops is described in "Making Changes", page 41.

To see the effects of these options, choose them in turn by clicking on the option button and selecting each option. If you choose the **Show Modified Loops** option, a

message appears that no loops meet the filter criterion, because you have not made any modifications.

Procedure 2-2 Filtering the Loop List by Loop Origin

Another way to filter is to choose loops that come from a single file or a single subroutine or function using these steps:

1. Open a list of subroutines (or functions) and files from which to choose by selecting the **Views** > **Subroutines and Files View** option.

2. Choose the filter criterion from the filtering option button. This is the right-most option button in the Parallel Analyzer View window. The filter criterion is **No Filtering** by default. You can filter according to source file or subroutine.

   To place filtering information in the editable text field that appears above the option button (Figure 2-4, page 13), you can do one of the following:

   - Enter the file or subroutine name in the text box that appears when you select **Filter by Subroutine** or **Filter by File**.

   - Choose the file or subroutine of interest in the Subroutines and Files View.

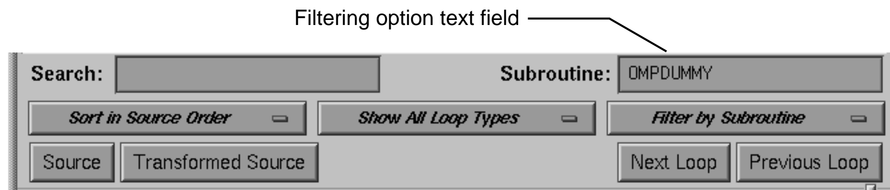


Figure 2-4 Filtering Option Button

Procedure 2-3 Sorting by Subroutine

The following procedure describes filtering the loop list by subroutine.

1. Open the subroutines and files view by choosing **Views** > **Subroutines and Files View**. The window opens and lists the subroutines and files in the file set. (See Figure 2-5, page 14.)

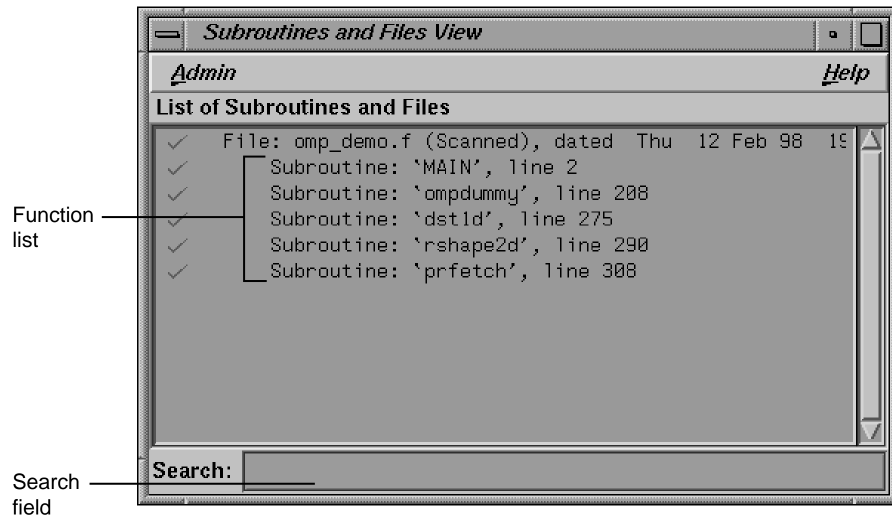


Figure 2-5 Subroutines and Files View

2. Choose **Filter by Subroutine** from the filtering option button.
3. Double-click the line for the subroutine `OMPDUMMY()` in the list of the **Subroutines and Files View** window. The name appears in the **Subroutine** filtering option text field (Figure 2-4, page 13), and the loop list is recreated according to the filter criteria.
4. You can also try choosing **Filter by File** with the filtering option button, but this is not very useful for this single-file example.
5. When you are done, display all of the loops in the sample source file again by choosing **No Filtering** with the option button.

Close the **Subroutines and Files View** by choosing its **Admin** > **Close** option.

Viewing Detailed Information About Code and Loops

This section describes how to examine original and transformed source, and the details of loops in the loop list.

Procedure 2-4 Viewing Original Source

Click the **Source** button on the lower left corner of the loop display controls to bring up the Source View window, shown in Figure 2-6, page 15.

Colored brackets mark the location of each loop in the file; you can click on a bracket to choose a loop in the loop list.

Note that the bracket colors vary as you scroll up and down the list. These colors correspond to different parallelization icons and indicate the parallelization status of each loop. The bracket colors indicate which loops are parallelized, which are unparallelizable, and which are left serial. The exact correspondence between colors and icons depends on the color settings of your monitor.

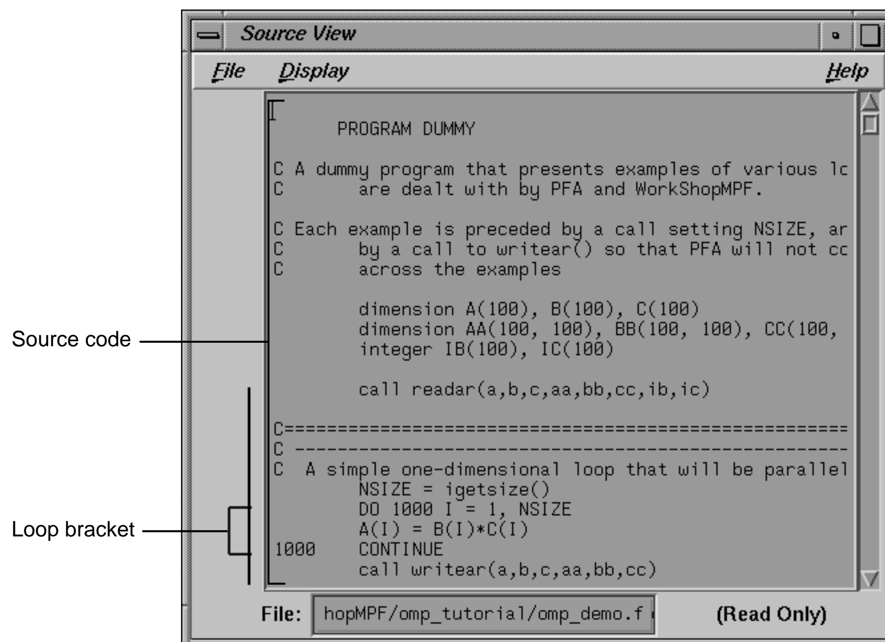


Figure 2-6 Source View

You can search the source listing by using one of the following:

- The File menu in the Source View.

- The keyboard shortcut **`Ctrl+s`** when the cursor is in the Source View.

You can locate a loop in the source code, click on its colored bracket in the Source View, and see more information about the loop in the loop information display.

Leave the Source View window open, because subsequent steps in this tutorial refer to the window.

Note: This window may also be used by the WorkShop Debugger and Performance Analyzer, so it remains open after you close the Parallel Analyzer View.

Procedure 2-5 Viewing Transformed Source

The compiler transforms loops for optimization and parallelization. The results of these transformations are not available to you directly, but they are mimicked in a file that you can examine. Each loop may be rewritten into one or more transformed loops, it may be combined with others, or it may be optimized away.

Click the **Transformed Source** button in the loop display controls (see Figure 2-2, page 11). A window labeled Parallel Analyzer View – Transformed Source opens, as shown in Figure 2-7, page 17.

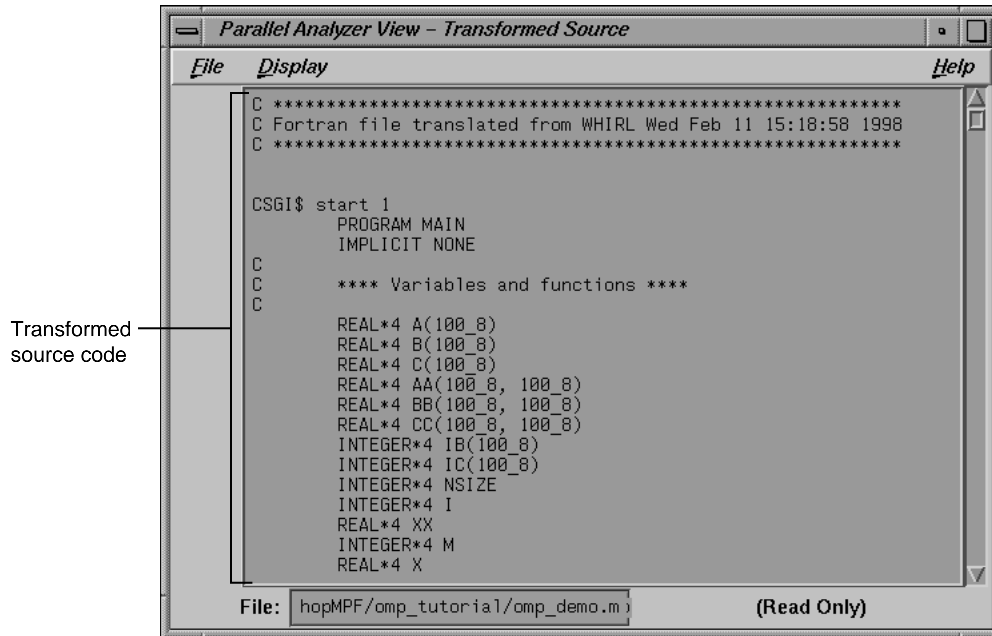


Figure 2-7 Transformed Source Window

Scroll through the Transformed Source window, and notice that it too has brackets that mark loops; the color correspondence is the same as for the Source View.

The bracketing color selection for the transformed source does not always distinguish between serial loops and unparallelizable loops; some unparallelizable loops may have the bracket color for a serial loop.

Leave the Transformed Source window open; subsequent steps in this tutorial refer to the window. You should have three windows open:

- Parallel Analyzer View
- Source View
- Transformed Source

Procedure 2-6 Navigating the Loop List

You can locate a loop in the main window by one of the following methods:

- Scrolling through the loop list using one of these:
  - Scroll bar.
  - Page Up and Page Down keys (the cursor must be over the loop list).
  - **Next Loop** and **Previous Loop** buttons.
- Searching for the Olid number using the Search field.

Procedure 2-7 Selecting a Loop for Analysis

To get more information about a loop, select it by one of the following methods:

- Double-click the line of text in the loop list (but not the icon).
- Click the loop bracket in either of the source viewing windows.

Selecting a loop has a number of effects on the different windows in the Parallel Analyzer View. Not all of the windows in the figure are open at this point in the tutorial; you can open them from the Views menu.

- In the Parallel Analyzer View, information about the selected loop appears in the previously empty loop information display.
- In the Source View, the original source code of the loop appears and is highlighted.
- In the Transformed Source, the first of the loops into which the original loop was transformed appears and is highlighted in the window. A bright vertical bar also appears next to each transformed loop that came from the original loop.
- The Transformed Loops View shows information about the loop after parallelization.
- The PFA Analysis Parameters View (`o32` code only) shows parameter values for the selected loop.

Try scrolling through the loop list and double-clicking various loops, and scrolling through the source displays and clicking the loop brackets to select loops. Notice that when you select a loop, a check mark appears to the left of the icon in the loop list, indicating that you have looked at it.

Scroll to the top of the loop list in the main view and double-click the line for the first loop, Olid 1.

Close the Transformed Loops View and the PFA Analysis Parameters View, if you have opened them.

Procedure 2-8 Using the Loop Information Display

The loop information display occupies the portion of the main view below the loop display controls. Initially, the display shows only **No loop is selected**. After a loop or subroutine is selected, the display contains detailed information and controls for requesting changes to your code.

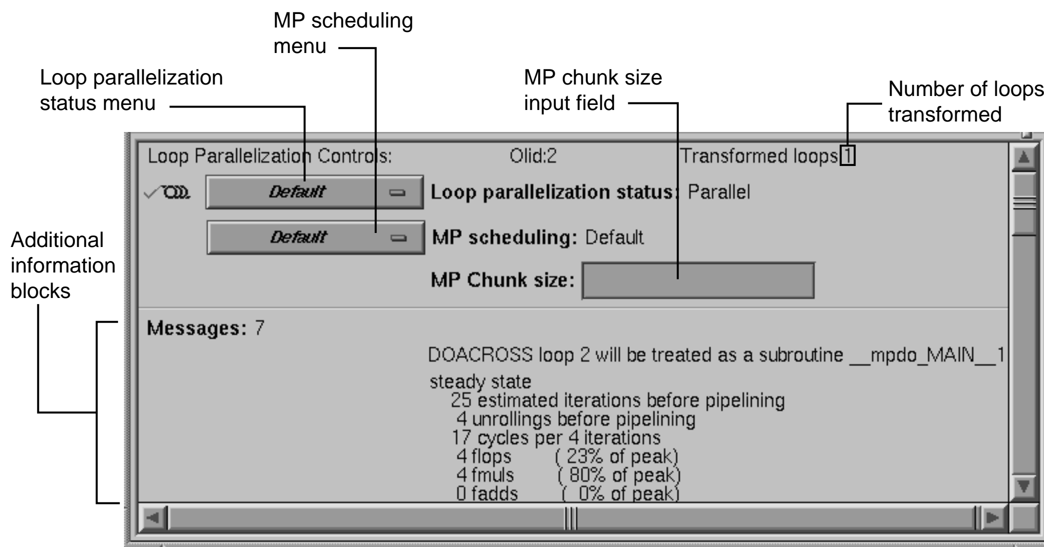


Figure 2-8 Loop Information Display Without Performance Data

The first line in the loop information display shows the Loop Parallelization Controls. The following are displayed when no performance information is available:

- On the first line is the loop Olid and the number of transformed loops derived from the selected loop.
- The next three lines display two option buttons and an editable text field.
  - The top button controls the loop's parallelization status (see "Loop Parallelization Status Option Button", page 131).
  - The second button controls the loop's multiprocessor scheduling. It is shown for all loops but is applicable to parallel loops only; for more information see "MP Scheduling Option Button: Directives for All Loops", page 133.

- The MP Chunk size editable text field lets you select the scheduling *chunk size*. For more information on the MP Chunk size, see "MP Chunk Size Field", page 134.

When the Parallel Analyzer View is run with a performance experiment, by invoking SpeedShop, an additional block (see Figure 5-4, page 104) appears above the parallelization controls. It gives performance information about the loop.

Up to five blocks of additional information may appear in the loop information display below the first separator line. These blocks list, when appropriate, the following information:

- Obstacles to parallelization
- Assertions made
- Directives applied
- Messages
- Questions the compiler asked (`o32` only)

Some of these lines may be accompanied by highlight buttons, represented by small light bulb icons. When you click one of these buttons, it highlights the relevant part of the code in the Source View and the Transformed Source windows.

The loop information display shows directives that apply to an entire subroutine when you select the line with the subroutine's name. If you select Olid 1, you see that there are no global directives in the main program. However, if you find subroutine `dst1d()`, you will see a directive that applies to it (see "Distributed Arrays: `C$SGI DISTRIBUTE` ", page 56).

The loop information display shows loop-specific directives when you select a loop. The lines for assertions and directives may have option buttons accompanying them that provide capabilities, such as, deleting a directive.

Using the Transformed Loops View

To see detailed information about the transformed loops derived from a particular loop, pull down the **Views** > **Transformed Loops View** option.

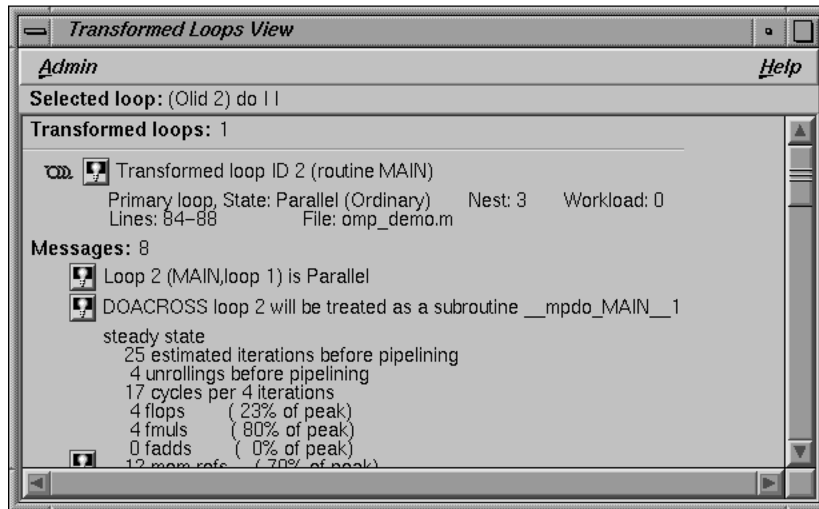


Figure 2-9 Transformed Loops View for Loop Olid 1

Transformed Loops View Description

The Transformed Loops View contains information about the loops into which the currently selected original loop was transformed. Each transformed loop has a block of information associated with it; the blocks are separated by horizontal lines.

The first line in each block contains:

- A parallelization status icon.
- A highlight button. It highlights the transformed loop in the Transformed Source window and the original loop in the Source View.
- The identification number of the transformed loop.

The next two lines describe the transformed loop. The first provides the following information:

- Whether it is a *primary* loop or *secondary* loop. A primary look is transformed from the selected original loop. A secondary loop is transformed from a different original loop, but it incorporates some code from the selected original loop.
- Parallelization state.

- Whether it is an ordinary loop or *interchanged* loop (see the Glossary).
- Nesting level.
- Workload.

The second line displays the location of the loop in the transformed source.

Any messages generated by the compiler are below the description lines. To the left of the message lines are highlight buttons, and left-clicking them highlights in the Source View the part of the original source that relates to the message. Often it is the first line of the original loop that is highlighted, since the message refers to the entire loop.

Selecting Transformed Loops

You can also select specific transformed loops. When you click a highlight button in the Transformed Loop View, the highlighting of the original source typically changes color, although for loop Olid 1 the highlighted lines do not (see Figure 2-10, page 23). For loops with more extensive transformations, the set of highlighted lines is different when you select from the Transformed Loops View (for example, see "Fused Loops", page 27).

Transformed loops can also be selected by clicking the corresponding loop brackets in the Transformed Source window.

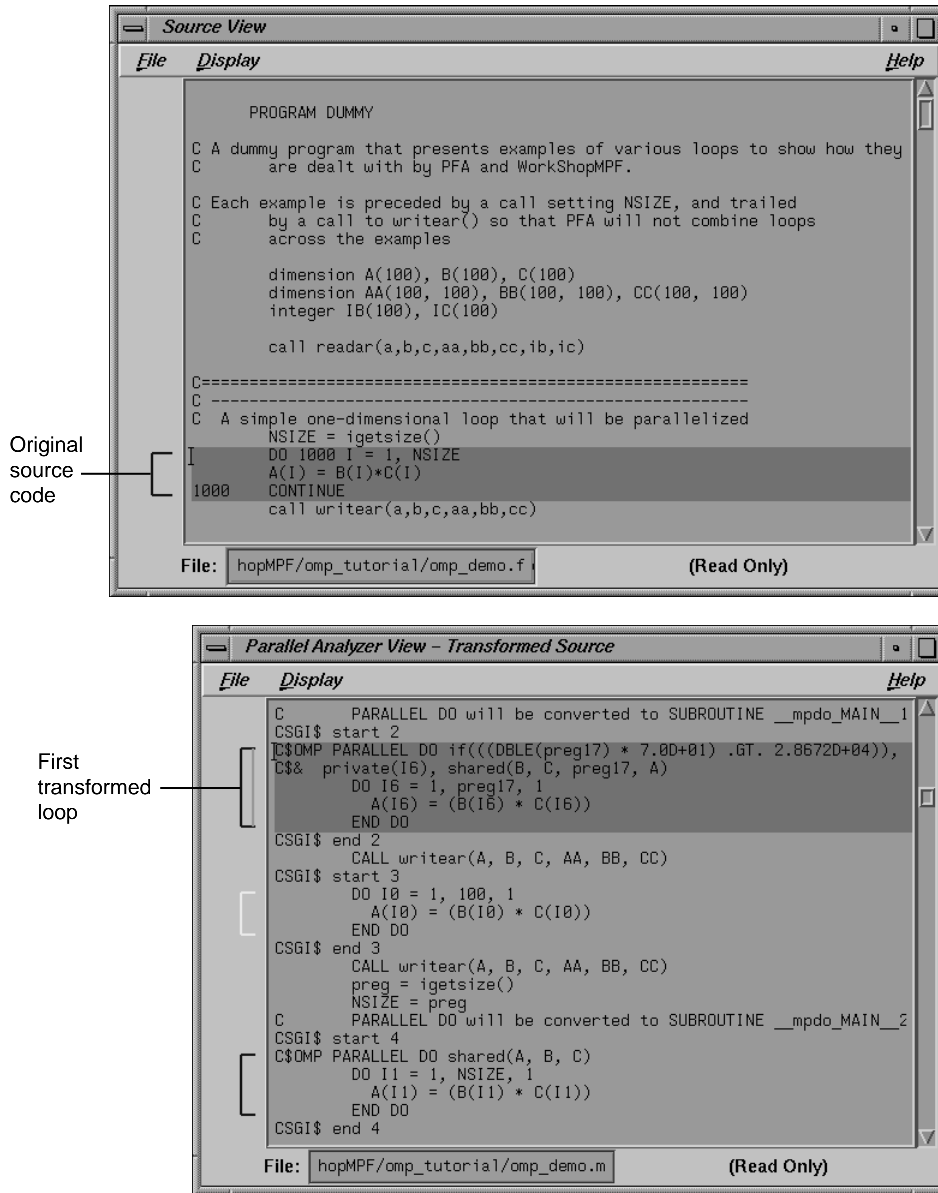


Figure 2-10 Transformed Loops in Source Windows

You can either leave the Transformed Loops View open or close it by selecting its **Admin** > **Close** menu item. When looking at subsequent loops, you might find it useful to see the information in the Transformed Loops View.

Examples of Simple Loops

Now that you are familiar with the basic features in the Parallel Analyzer View user interface, you can start examining, analyzing, and modifying loops.

The loops in this section are the simplest kinds of Fortran loops:

- "Simple Parallel Loop", page 24.
- "Serial Loop", page 25.
- "Explicitly Parallelized Loop", page 25.
- "Fused Loops", page 27.
- "Loop That Is Eliminated", page 28.

Two other sections discuss more complicated loops:

- "Examining Loops With Obstacles to Parallelization", page 28.
- "Examining Nested Loops", page 39.

Note: The loops in the next sections are referred to by their Olid. Changes to the Parallel Analyzer View, such as, the implementation of updated OpenMP standards, may cause the Olid you see on your system to differ from that in the tutorial. Example code, which you can find in the Source View, is included in the tutorial to clarify the discussion.

Simple Parallel Loop

Scroll to the top of the list of loops and select loop Olid 2. This loop is a simple loop: computations in each iteration are independent of each other. It was transformed by the compiler to run concurrently. Notice in the Transformed Source window the directives added by the compiler.

Example 2-1 Simple Parallel Loop

```
      DO 1000 I = 1, NSIZE
          A(I) = B(I)*C(I)
1000   CONTINUE
```

Move to the next loop by clicking the **Next Loop** button.

Serial Loop

Olid 2 is a simple loop with too little content to justify running it in parallel. The compiler determined that the overhead of parallelizing would exceed the benefits; the original loop and the transformed loop are identical.

Example 2-2 Serial Loop

```
      DO 1100 I = 1, NSIZE
        A(I) = B(I)*C(I)
1100  CONTINUE
```

Move to the next loop by clicking the **Next Loop** button.

Explicitly Parallelized Loop

Loop Olid 3 is parallelized because it contains an explicit `C$OMP PARALLEL DO` directive in the source, as is shown in the loop information display (Figure 2-11, page 26). The compiler passes the directive through to the transformed source.

The loop parallelization status option button is set to **C$OMP PARALLEL DO...**, and it is shown with a highlight button. Clicking the highlight button brings up both the Source View (Figure 2-12, page 27), if it is not already opened, and the Parallelization Control View, which shows more information about the parallelization directive.

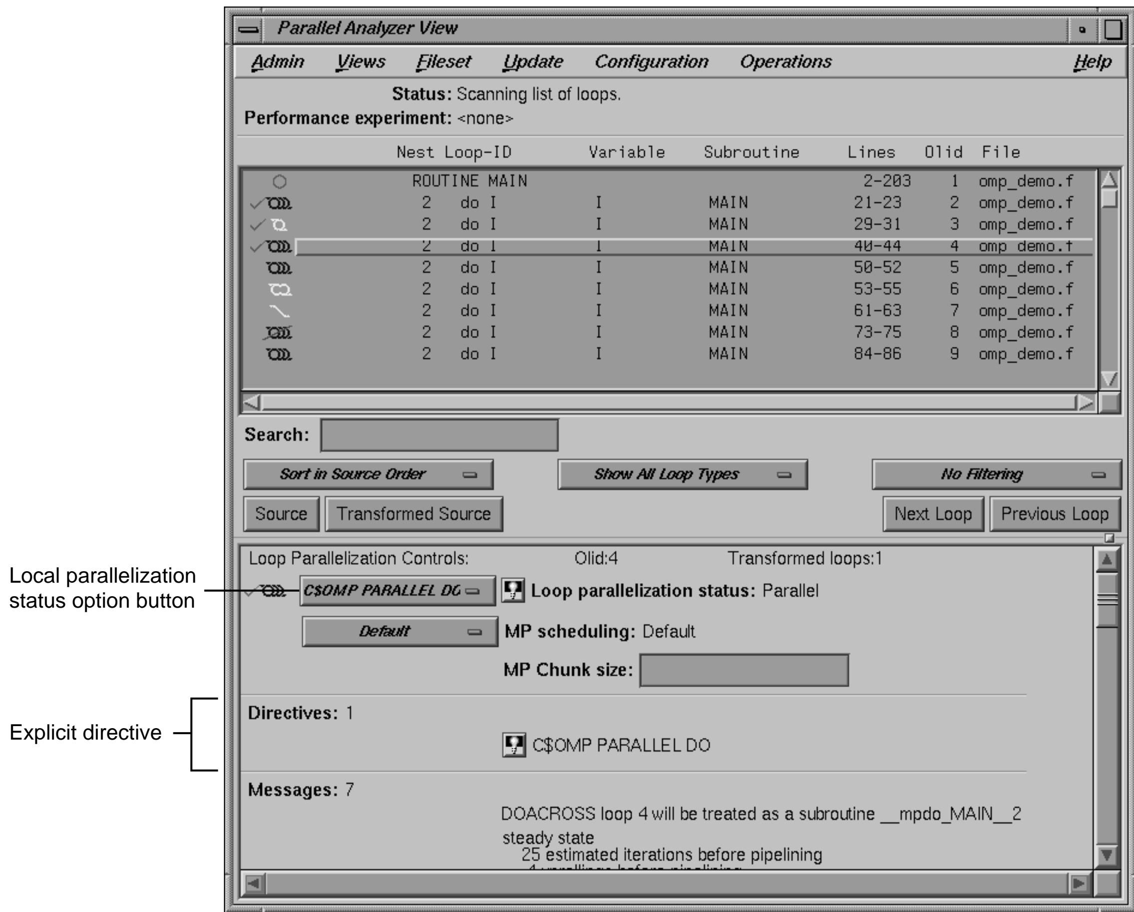


Figure 2-11 Explicitly Parallelized Loop

If you clicked on the highlight button, close the Parallelization Control View. (Using the Parallelization Control View is discussed in "Adding `C$OMP PARALLEL DO` Directives and Clauses", page 42.)

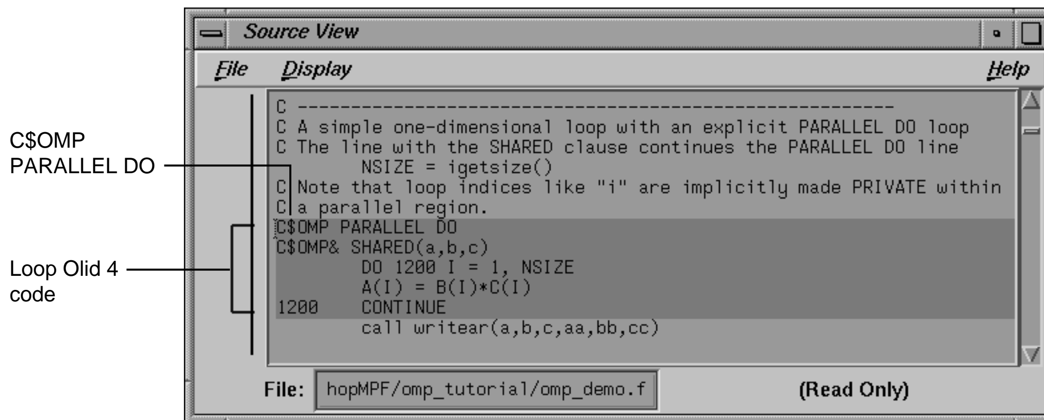


Figure 2-12 Source View of C$OMP PARALLEL DO Directive

Close the Source View and move to the next loop by clicking the **Next Loop** button.

Fused Loops

Loops Olid 5 and Olid 6 are simple parallel loops that have similar structures. The compiler combines these loops to decrease overhead. Note that loop Olid 6 is described as fused in the loop information display and in the Transformed Loops View; it is incorporated into the parallelized loop Olid 5. If you look at the Transformed Source window and select Olid 5 and Olid 6, the identical lines of code are highlighted for each loop.

Example 2-3 Fused Loop

```
        DO 1300 I = 1, NSIZE
        A(I) = B(I) + C(I)
1300    CONTINUE
        DO 1350 I = 1, NSIZE
        AA(I,NSIZE) = B(I) + C(I)
1350    CONTINUE
```

Move to the next loop by clicking **Next Loop** twice.

Loop That Is Eliminated

Loop Olid 7 is an example of a loop that the compiler can eliminate entirely. The compiler determines that the body is independent of the rest of the loop. It moves the body outside of the loop, and eliminates the loop. The transformed source is not scrolled and highlighted when you select Olid 7 because there is no transformed loop derived from the original loop.

Example 2-4 Eliminated Loop

```
        DO 1500 I = 1, NSIZE
        XX = 10.0
1500    CONTINUE
```

Move to the next loop, Olid 8, by clicking the **Next Loop** button. This loop is discussed in "Unparallelizable Carried Data Dependence", page 29.

Examining Loops With Obstacles to Parallelization

There are a number of reasons why a loop may not be parallelized. The loops in the following parts of this section illustrate some of these reasons, along with variants that allow parallelization:

- "Obstacles to Parallelization: Carried Data Dependence", page 29
- "Obstacles to Parallelization: I/O Operations", page 32
- "Obstacles to Parallelization: Unstructured Control Flow", page 32
- "Obstacles to Parallelization: Subroutine Calls", page 33
- "Obstacles to Parallelization: Permutation Vectors", page 34

These loops are a few specific examples of the obstacles to parallelization recognized by the compiler. The final part of this section, "Obstacles to Parallelization Messages", page 35, contains two tables that list all of the messages generated by the compiler that concern obstacles to parallelization.

Obstacles to Parallelization: Carried Data Dependence

Carried data dependence typically arises when recurrence of a variable occurs in a loop. Depending on the nature of the recurrence, parallelizing the loop may be impossible. The following loops illustrate four kinds of data dependence:

- "Unparallelizable Carried Data Dependence", page 29
- "Parallelizable Carried Data Dependence", page 29
- "Multi-line Data Dependence", page 30
- "Reductions", page 31

Unparallelizable Carried Data Dependence

Loop Olid 8 is a loop that cannot be parallelized because of a data dependence; one element of an array is used to set another in a recurrence.

Example 2-5 Unparallelizable Carried Data Dependence

```
        DO 2000 I = 1, NSIZE-1
        A(I) = A(I+1)
2000    CONTINUE
```

If the loop were nontrivial (if `NSIZE` were greater than two) and if the loop were run in parallel, iterations might execute out of order. For example, iteration 4, which sets `A(4)` to `A(5)`, might occur after iteration 5, which resets the value of `A(5)`; the computation would be unpredictable.

The loop information display lists the obstacle to parallelization. Click the highlight button that accompanies it. Two kinds of highlighting occur in the Source View:

- The relevant line that has the dependence
- The uses of the variable that obstruct parallelization; only the uses of the variable within the loop are highlighted

Move to the next loop by clicking **Next Loop**.

Parallelizable Carried Data Dependence

Loop Olid 9 has a structure similar to loop Olid 8. Despite the similarity however, Olid 9 may be parallelized.

Example 2-6 Parallelizable Carried Data Dependence

```
C*$*ASSERT DO (CONCURRENT)
        DO 2100 I = 1, NSIZE
        A(I) = A(I+M)
2100    CONTINUE
```

Note that the array indices differ by offset `M`. If `M` is equal to `NSIZE` and the array is twice `NSIZE`, the code is actually copying the upper half of the array into the lower half, a process that can be run in parallel. The compiler cannot recognize this from the source, but the code has the assertion `C*$* ASSERT DO (CONCURRENT)` so the loop is parallelized.

Click the highlight button to show the assertion in the Source View.

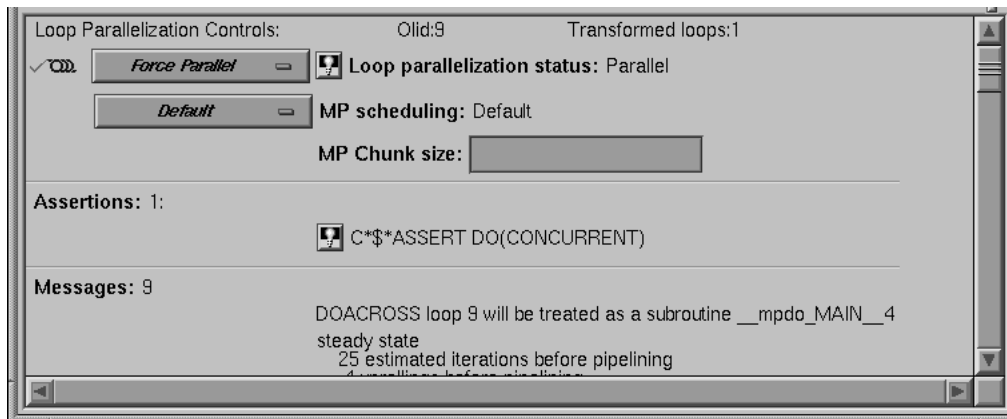


Figure 2-13 Parallelizable Data Dependence

Move to the next loop by clicking the **Next Loop** button.

Multi-line Data Dependence

Data dependence can involve more than one line of a program. In loop Olid 10, a dependence similar to that in Olid 9 occurs, but the variable is set and used on different lines.

Example 2-7 Multi-line Data Dependence

```
        DO 2200 I = 1, NSIZE-1
        B(I) = A(I)
        A(I+1) = B(I)
2200    CONTINUE
```

Click the highlight button on the obstacle line.

In the Source View, highlighting shows the dependency variable on the two lines (see the figure that follows). Of course, real programs, typically, have far more complex dependences than this.

Move to the next loop by clicking **Next Loop.**

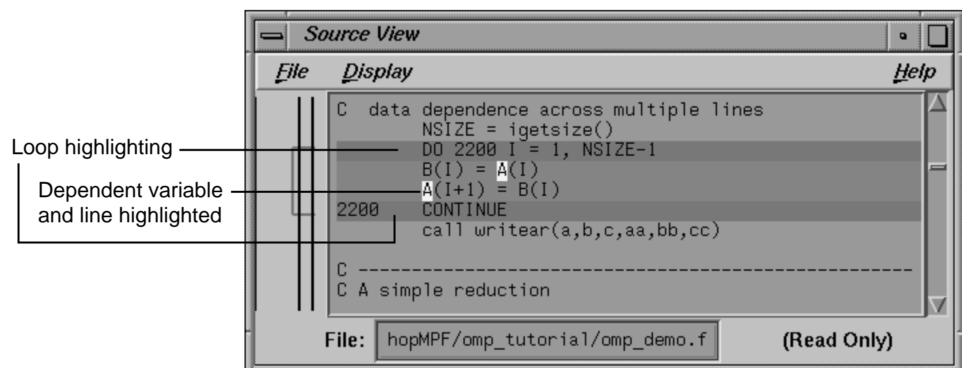


Figure 2-14 Highlighting on Multiple Lines

Reductions

Loop Olid 11 shows a data dependence that is called a reduction: the variable responsible for the data dependence is being accumulated or **reduced** in some fashion. A reduction can be a summation, a multiplication, or a minimum or maximum determination. For a summation, as shown in this loop, the code could accumulate partial sums in each processor and then add the partial sums at the end.

Example 2-8 Reduction

```
        DO 2300 I = 1, NSIZE
        X =  B(I)*C(I) + X
2300    CONTINUE
```

However, because floating-point arithmetic is inexact, the order of addition might give different answers because of roundoff error. This does not imply that the serial execution answer is correct and the parallel execution answer is incorrect; they are equally valid within the limits of roundoff error. With the `-O3` optimization level, the compiler assumes it is OK to introduce roundoff error, and it parallelizes the loop. If you do not want a loop parallelized because of the difference caused by roundoff error, compile with the `-OPT:roundoff=0` or `1` option.

Move to the next loop by clicking **Next Loop**.

Obstacles to Parallelization: I/O Operations

Loop Olid 12 has an input/output (I/O) operation in it. It cannot be parallelized because the output would appear in a different order depending on the scheduling of the individual CPUs.

Example 2-9 Input/Output Operation

```
        DO 2500 I = 1, NSIZE
        print 2599, I, A(I)
2599        format("Element A(",I2,") = ",f10.2)
2500    CONTINUE
```

Click the button indicating the obstacle, and note the highlighting of the print statement in the Source View.

Move to the next loop by clicking **Next Loop**.

Obstacles to Parallelization: Unstructured Control Flow

Loop Olid 13 has an unstructured control flow: the flow is not controlled by nested `if` statements. Typically, this problem arises when `goto` statements are used; if you can get the branching behavior you need by using nested `if` statements, the compiler can better optimize your program.

Example 2-10 Unstructured Control Flow

```
        DO 2600 I = 1, NSIZE
        A(I) = B(I)*C(I)
        IF (A(I) .EQ. 0) GO TO 2650
2600    CONTINUE
```

Because the `goto` statement is essential to the program's behavior, the compiler cannot determine how many iterations will take place before exiting the loop. If the compiler parallelized the loop, one thread might execute iterations past the point where another has determined to exit.

Click the highlight button in the Obstacles to Parallelization information block in the loop information display, next to the unstructured control flow message. Note that the line with the exit from the loop is highlighted in the Source View.

Move to the next loop by clicking **Next Loop**.

Obstacles to Parallelization: Subroutine Calls

Unless you make an assertion, a loop with a subroutine call cannot be parallelized; the compiler cannot determine whether a call has side effects (such as creating data dependencies.)

Unparallelizable Loop With a Subroutine Call

Loop Olid 14 is unparallelizable because there is a call to a subroutine, `RTC()`, and there is no explicit assertion to parallelize.

Example 2-11 Unparallelizable Loop With Subroutine Call

```
        DO 2700 I = 1, NSIZE
        A(I) = B(I) + RTC()
2700    CONTINUE
```

Click the highlight button on the obstacle line; note the highlighting of the line containing the call and the highlighting of the subroutine name.

Move to the next loop by clicking the **Next Loop button**.

Parallelizable Loop With a Subroutine Call

Although loop Olid 15 has a subroutine call in it similar to that in Olid 14, it can be parallelized because of the assertion that the call has no side effects that will prevent concurrent processing.

Example 2-12 Parallelizable Loop With Subroutine Call

```
C*$*ASSERT CONCURRENT CALL
        DO 2800 I = 1, NSIZE
        A(I) = B(I) + FOO()
2800    CONTINUE
```

Click the highlight button on the assertion line in the loop information display to highlight the line in the Source View containing the assertion.

Move to the next loop by clicking **Next Loop**.

Obstacles to Parallelization: Permutation Vectors

If you specify array index values by values in another array (referred to as a **permutation vector**), the compiler cannot determine if the values in the permutation vector are distinct. If the values are distinct, loop iterations do not depend on each other and the loop can be parallelized; if they are not, the loop cannot be parallelized. Thus, without an assertion, a loop with a permutation vector is not parallelized.

Unparallelizable Loop With a Permutation Vector

Loop Olid 16 has a permutation vector, `IC(I)`, and cannot be parallelized.

Example 2-13 Unparallelizable Loop With Permutation Vector

```
        DO 3200 I = 1, NSIZE-1
        A(IC(I)) = A(IC(I)) + DELTA
3200    CONTINUE
```

Move to the next loop by clicking the **Next Loop** button.

Parallelizable Loop With a Permutation Vector

An assertion, `C*$* ASSERT PERMUTATION`, that the index array, `IB(I)` is indeed a permutation vector has been added before loop Olid 17. Therefore, the loop is parallelized.

Example 2-14 Parallelizable Loop With Permutation Vector

```
C*$*ASSERT PERMUTATION(ib)
        DO 3300 I = 1, NSIZE
        A(IB(I)) = A(IB(I)) + DELTA
3300    CONTINUE
```

Move to the next loop, Olid 18, by clicking **Next Loop**. This loop is discussed in "Doubly Nested Loop", page 39.

Obstacles to Parallelization Messages

All of the messages that can be found in an Obstacles to Parallelization information block are found in the following lists. Because they include specific loop and line information, messages that appear in the loop information display differ slightly from those in the tables.

`Loop doesn't have parallelization directive`

Auto-parallelization is off. Loop doesn't contain a parallelization directive.

`Loop is preferred serial; insufficient work to justify parallelization`

Could have been parallelized, but preferred serial. The compiler determined there was not enough work in the loop to make parallelization worthwhile.

`Loop is preferred serial; parallelizing inner loop is more efficient`

Could have been parallelized, but preferred serial. The compiler determined that making an inner loop parallel would lead to faster execution.

`Loop has unstructured control flow`

Might be parallelizable. There is a `goto` statement or other unstructured control flow in the loop.

`Loop was created by peeling the last iteration of a parallel loop`

Might be parallelizable. Loop was created by peeling off the final iteration of another loop to make that loop go parallel. Compiler did not try to parallelize this peeled, last iteration.

`User directive specifies serial execution for loop`

Might be parallelizable. Loop has a directive that it should not be parallelized.

`Loop can not be parallelized; tiled for reshaped array instead`

Might be parallelizable. The loop has been tiled because it has reshaped arrays, or is inside a loop with reshaped arrays. The compiler does not parallelize such loops.

`Loop is nested inside a parallel loop`

Might be parallelizable. Loop is inside a parallel loop. Therefore, the compiler does not consider it to be a candidate for parallelization.

`Loop is the serial version of parallel loop`

Might be parallelizable. The loop is part of the serial version of a parallelized loop. This may occur when a loop is in a routine called from a parallelized loop; the called loop is effectively nested in a parallel loop, so the compiler does not parallelize it.

`Tough upper bounds`

Could not have gone parallel. Loop could not be put in standard form, and therefore could not be analyzed for parallelization. Standard form is:

```
for (i = lb; i <= ub; i++)
```

`Indirect ref`

Could not have gone parallel. Loop contains some complex memory access that is too difficult to analyze.

The following table lists the Obstacles to Parallelization block messages that deal with dependence issues (such as those involving scalars, arrays, missing information, and finalization).

`Loop has carried dependence on scalar variable`

Problem with scalars. The loop has a carried dependence on a scalar variable.

`Loop scalar variable is aliased precluding auto parallelization`

Problem with scalars. A scalar variable is aliased with another variable, e.g. a statement equivalencing a scalar and an array.

`Loop can not determine last value for variable`

Problem with scalars. A variable is used out of the loop, and the compiler could not determine a unique last value.

`Loop carried dependence on array`

Problem with arrays. The loop carries an array dependence from one array member to another array member.

`Call inhibits auto parallelization`

Problem with missing dependence information. A call in the loop has no dependence information, and is assumed to create a data dependence.

`Input-output statement`

Problem with missing dependence information. The compiler does not parallelize loops with input or output statements.

`Insufficient information in array`

Problem with missing dependence information.

Array has no dependence information.

`Insufficient information in reference`

Problem with missing dependence information. Unnamed reference has no dependence information.

`Loop must finalize value of scalar before it can go parallel`

Problem with finalization. Value of scalar must be determined to parallelize loop.

`Loop must finalize value of array before it can go parallel`

Problem with finalization. Value of array must be determined to parallelize loop.

`Scalar may not be assigned in final iteration`

Problem with finalization. The compiler needed to finalize the value of a scalar to parallelize the loop, but it couldn't because the value is not always assigned in the last iteration of the loop.

The following code is an example. The variable *s* poses a problem; the `if` statement makes it unclear whether the variable is set in the last iteration of the loop.

```
subroutine fun02(a, b, n, s)
    integer a(n), b(n), s, n
    do i = 1, n
      if (a(i) .gt. 0) then
        s = a(i)
      end if
      b(i) = a(i) + s
    end do
    end
```

`Array may not be assigned in final iteration`

Problem with finalization. The compiler needed to finalize the value of an array to parallelize the loop, but it couldn't because the values are not always assigned in the last iteration of the loop.

The following is an example. The variable *b* poses a problem when the compiler tries to parallelize the *i* loop; it is not set in the last iteration.

```
subroutine fun04(a, b, n)
 integer i, j, k, n
 integer b(n), a(n,n,n)
 do i = 1, n
   do j = i + 3, n
```

```
c*$* no fusion
            do k = 1, n
              b(k) = k
            end do
            do k = 1, n
              a(i,j,k) = a(i,j,k) + b(k)
            end do
          end do
        end do
        end
```

Examining Nested Loops

The loops in this section illustrate more complicated situations, involving nested and interchanged loops.

Doubly Nested Loop

Loop Olid 18 is the outer loop of a pair of loops and it runs in parallel. The inner loop runs in serial, because the compiler knows that one parallel loop should not be nested inside another. However, you can force parallelization in this context by inserting a `C$OMP PARALLEL DO` directive with the `C$SGI&NEST` clause. For example, see "Distributed and Reshaped Arrays: `C$SGI DISTRIBUTE_RESHAPE`", page 58.

Example 2-15 Doubly Nested Loop

```
        DO 4000   I = 1,NSIZE
          DO 4010   J = 1,NSIZE
            AA(J,I) = BB(J, I)
4010      CONTINUE
4000    CONTINUE
```

Click **Next Loop** to move to the inner loop, Olid 19.

Notice that when you select the inner loop that the end-of-loop continue statement is not highlighted. This happens for all interior loops and is a compiler error that disrupts line numbering in the Parallel Analyzer View. Be careful if you use the Parallel Analyzer View to insert a directive for an interior loop; check that the directive is properly placed in your source code.

Click **Next Loop** again to select the outer loop of the next nested pair.

Interchanged Doubly Nested Loop

The outer loop, Olid 20, is shown in the loop information display as a serial loop inside a parallel loop. The original interior loop is labelled as parallel, indicating the order of the loops has been interchanged. This happens because the compiler recognized that the two loops can be interchanged, and that the CPU cache is likely to be more efficiently used if the loops are run in the interchanged order. Explanatory messages appear in the loop information display.

Example 2-16 Interchanged Doubly Nested Loop

```
        DO 4100   I = 1,NSIZE
          DO 4110   J = 1,NSIZE
            AA(I,J) = BB(I, J)
4110      CONTINUE
4100    CONTINUE
```

Move to the inner loop, Olid 21, by clicking the **Next Loop** button.

Click **Next Loop** once again to move to the following triply-nested loop.

Triply Nested Loop With an Interchange

The order of Olid 22 and Olid 23 has been interchanged. As with the previous nested loops, the compiler recognizes that cache misses are less likely.

Example 2-17 Triply Nested Loop With Interchange

```
        DO 5000   I = 1,NSIZE
          DO 5010   J = 1,NSIZE
            CC(I,J) = 0.
            DO 5020   K = 1,NSIZE
              CC(I,J) = CC(I,J) + AA(I,K)* BB(K,J)
5020        CONTINUE
5010      CONTINUE
5000    CONTINUE
```

Double-click on Olid 22, Olid 23, and Olid 24 in the loop list and note that the loop information display shows that Olid 22 and Olid 24 are serial loops inside a parallel loop, Olid 23.

Because the innermost serial loop, Olid 24, depends without recurrence on the indices of Olid 22 and Olid 23, iterations of loop Olid 22 can run concurrently. The compiler

does not recognize this possibility. This brings us to the subject of the next section, the use of the Parallel Analyzer View tools to modify the source.

Return to Olid 22, if necessary, by using the **Previous Loop** button.

Modifying Source Files and Compiling

So far, the discussion has focused on ways to view the source and parallelization effects. This section discusses controls that can change the source code by adding directives or assertions, allowing a subsequent pass of the compiler to do a better job of parallelizing your code.

You control most of the directives and some of the assertions available from the Parallel Analyzer View with the Operations menu.

You control most of the assertions and the more complex directives, `C$OMP DO` and `C$OMP PARALLEL DO`, with the loop parallelization status option button. (See Figure 2-15, page 42.)

There are two steps to modifying source files:

1. Make changes using the Parallel Analyzer View controls, discussed in "Making Changes", page 41
2. Modify the source and rebuild the program and its analysis files, discussed in "Applying Requested Changes", page 46.

Making Changes

You make changes by one of the following actions:

- Add or delete assertions and directives using the Operations menu or the Loop Parallelization Controls.
- Add clauses to or modify directives using the Parallelization Control View.
- Modify the PFA analysis parameters in the PFA Analysis Parameters View (`o32` only.)

You can request changes in any order; there are no dependencies implied by the order of requests. The following changes are discussed in this section:

- "Adding `C$OMP PARALLEL DO` Directives and Clauses", page 42

• "Adding New Assertions or Directives With the Operations Menu", page 45
• "Deleting Assertions or Directives", page 46

Adding `C$OMP PARALLEL DO` Directives and Clauses

Loop Olid 22, shown in Figure 2-15, is a serial loop nested inside a parallel loop. It is not parallelized, but its iterations could run concurrently.

To add a `C$OMP PARALLEL DO` directive to Olid 22, do the following:

1. Make sure loop Olid 22 is selected.
2. Click on the loop parallelization status option button (Figure 2-15) and choose **C$OMP PARALLEL DO...** to parallelize Olid 22.

This sequence requests a change in the source code, and opens the Parallelization Control View (Figure 2-15). You can now look at variables in the loop and attach clauses to the directive, if needed.

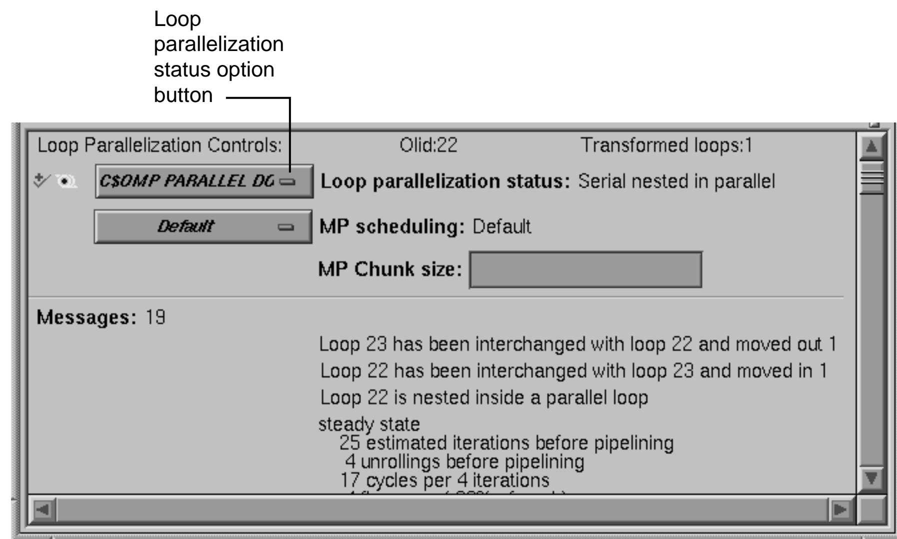


Figure 2-15 Requesting a C$OMP PARALLEL DO Directive

Figure 2-16, page 44, shows information presented in the Parallelization Control View for a `C$OMP PARALLEL DO` directive. (For the `C$OMP DO` directive, see "Parallelization Control View", page 136):

- The selected loop.
- Condition for parallelization editable text field.
- MP scheduling option button.
- MP Chunk size editable text field.
- `PRIVATE`, `SHARED`, `DEFAULT`, `FIRSTPRIVATE`, `LASTPRIVATE`, `COPYIN`, `REDUCTION`, `AFFINITY`, `NEST`, and `ONTO` clause windows.
- A list of all the variables in the loop, each with an icon indicating whether the variable was read, written, or both; these icons are introduced in "Loop List Icons", page 10.

In the list of variables, each variable has a highlight button to indicate in the Source View its use within the loop; click some of the buttons to see the variables highlighted in the source view. After each variable's name, there is a descriptor of its storage class: Automatic, Common, or Reference. (See "Variable List Storage Labeling", page 143.)

You can add clauses to the directive by placing appropriate parameters in the text fields, or using the options menus.

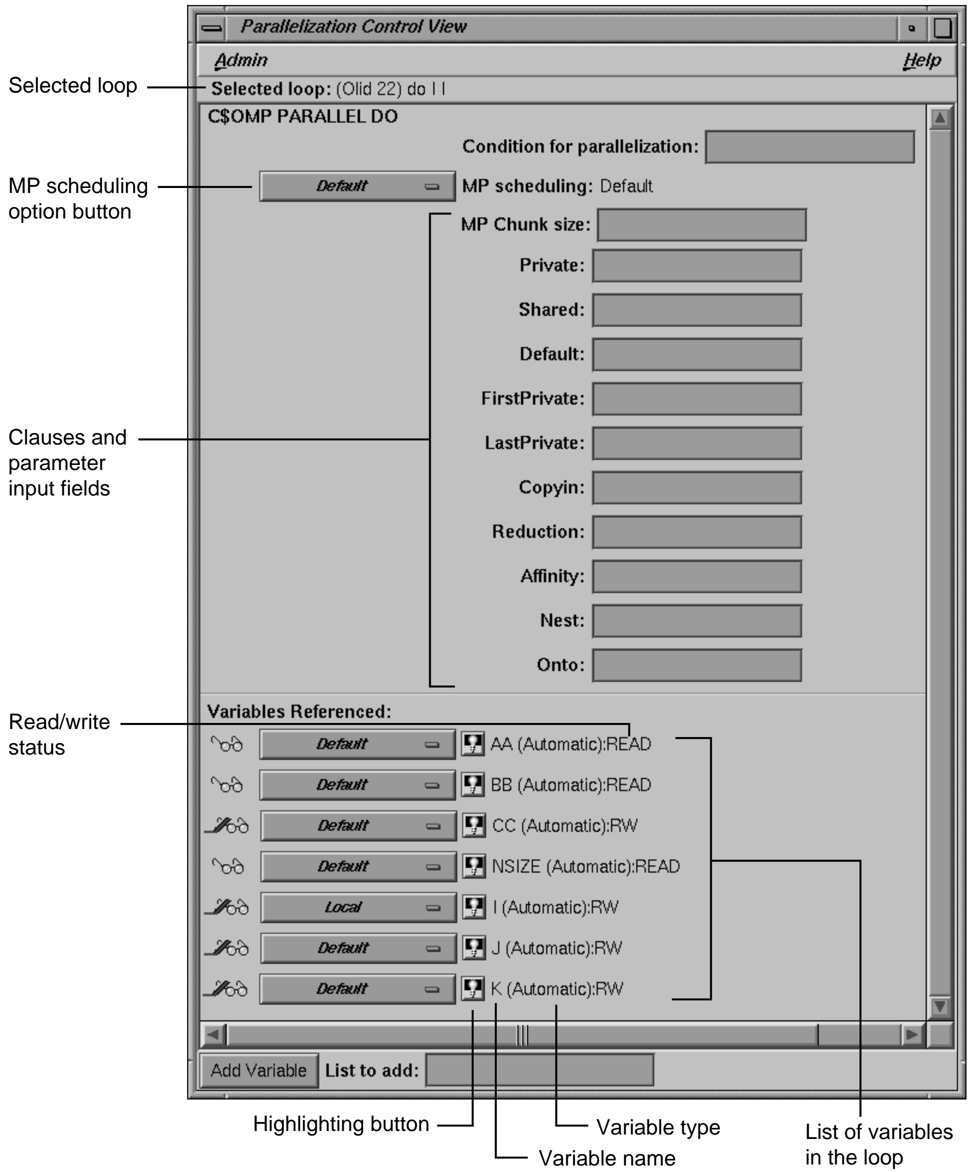


Figure 2-16 Parallelization Control View After Choosing `C$OMP PARALLEL DO...`

Notice that in the loop list, there is now a red plus sign next to this loop, indicating that a change has been requested (see Figure 2-17).

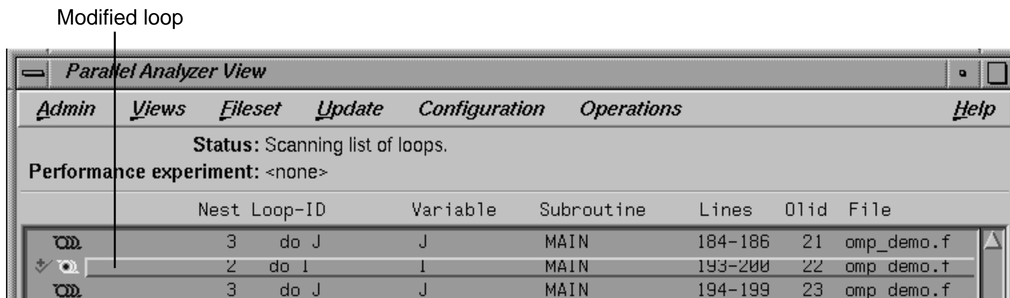


Figure 2-17 Effect of Changes on the Loop List

Close the Parallelization Control View by using its **Admin > Close** option.

Adding New Assertions or Directives With the Operations Menu

To add a new assertion to a loop, do the following:

1. Find loop Olid 14 (introduced in Example 2-11, page 33) either by scrolling the loop list or by using the search feature of the loop list. (Go to the Search field and enter **`14`**.)

2. Double-click the highlighted line in the loop list to select the loop.

3. Pull down **Operations > Add Assertion >** `C*$*ASSERT CONCURRENT CALL` to request a new assertion.

This adds an assertion, `C*$* ASSERT CONCURRENT CALL`, that says it is safe to parallelize the loop despite the call to `RTC()`, which the compiler thought might be an obstacle to parallelization. The loop information display shows the new assertion, along with an **Insert option** button to indicate the state of the assertion when you modify the code.

The procedure for adding directives is similar. To start, choose **Operations > Add Directive**.

Deleting Assertions or Directives

Move to the next loop, Olid 15 (shown in Example 2-12, page 34).

To delete an assertion, follow these steps:

1. Find the assertion `C*$* ASSERT CONCURRENT CALL` in the loop information display.
2. Select its **Delete** option button.

Figure 2-18 shows the state of the assertion in the information display. A similar procedure is used to delete directives.

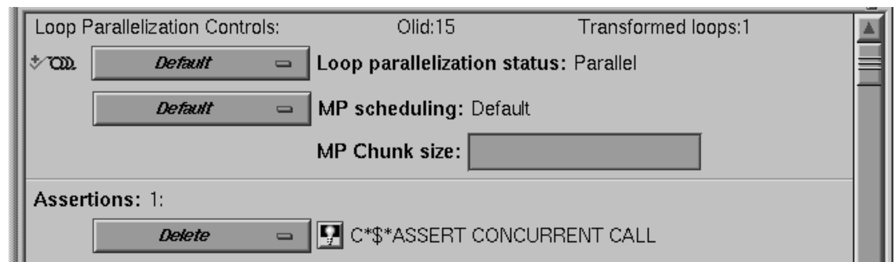


Figure 2-18 Deleting an Assertion

From this point, the next non-optional step in the tutorial is at the beginning of "Updating the Source File", page 49.

Applying Requested Changes

Now you have requested a set of changes. Using the controls in the Update menu, you can update the file. These are the main actions that the Parallel Analyzer View performs during file modification:

1. Generates a `sed` script to accomplish the following steps.
   - Rename the original file to have the suffix `.old`.
   - Run `sed` on that file to produce a new version of the file, in this case `omp_demo.f`.

2. Depending on how you set the two checkboxes in the Update menu, the Parallel Analyzer View then does one of the following:

   - Spawns the WorkShop Build Manager to rerun the compiler on the new version of the file.

   - Opens a `gdiff` window or an editor, allowing you to examine changes and further modify the source before running the compiler. When you quit `gdiff`, the editing window opens if you have set the checkboxes for both windows. When you quit these tools, the Parallel Analyzer View spawns the WorkShop Build Manager.

3. After the build, the Parallel Analyzer View rescans the files and loads the modified code for further interaction.

Viewing Changes With gdiff

By default, the Parallel Analyzer View does not open a `gdiff` window. To open a `gdiff` window that shows the requested changes to the source file before compiling the modified code, toggle the checkbox in **Update > Run gdiff After Update** (Figure 2-19, page 48).

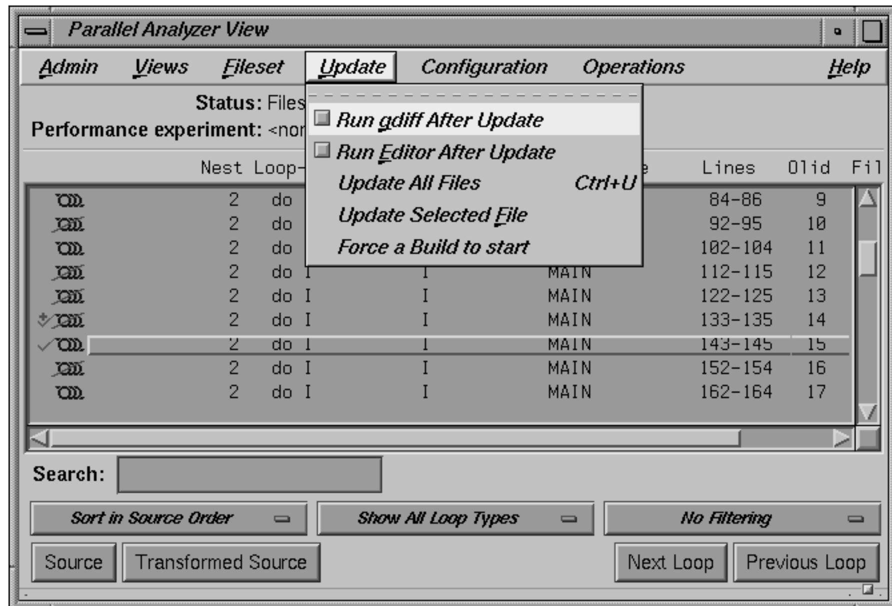


Figure 2-19 Run gdiff After Update

If you always wish to see the `gdiff` window, you can set the resource in your `.Xdefaults` file:

```
cvpav*gDiff: True
```

Modifying the Source File Further

After running the `sed`script, to make additional changes before compiling the modified code, open an editor by toggling the **Update > Run Editor After Update** checkbox. An `xwsh` window with `vi` running in it opens with the source code ready to be edited.

If you always prefer to run the editor, you can set the resource in your `.Xdefaults` file:

```
cvpav*runUserEdit: True
```

If you prefer a different window shell or a different editor, you can modify the resource in your `.Xdefaults` file and change from `xwsh` or `vi` as you prefer. The

following is the default command in the .Xdefault, which you can edit for your preference:

```
cvpav*userEdit: xwsh -e vi %s +%d
```

In the above command, the +%d tells vi at what line to position itself in the file and is replaced with **1** by default. (You can omit the +%d parameter if you wish.) The edited file's name either replaces any explicit %s, or if the %s is omitted, its filename is appended to the command.

Updating the Source File

Choose **Update > Update All Files** to update the source file to include the changes requested in this tutorial. (See .) Alternatively, you can use the keyboard shortcut for this operation, **Ctrl+U**, with the cursor anywhere in the main view.

If you have set the checkbox and opened the gdiff window or an editor, examine the changes or edit the file as you wish. When you exit these tools, the Parallel Analyzer View spawns the WorkShop Build Manager (Figure 2-20).

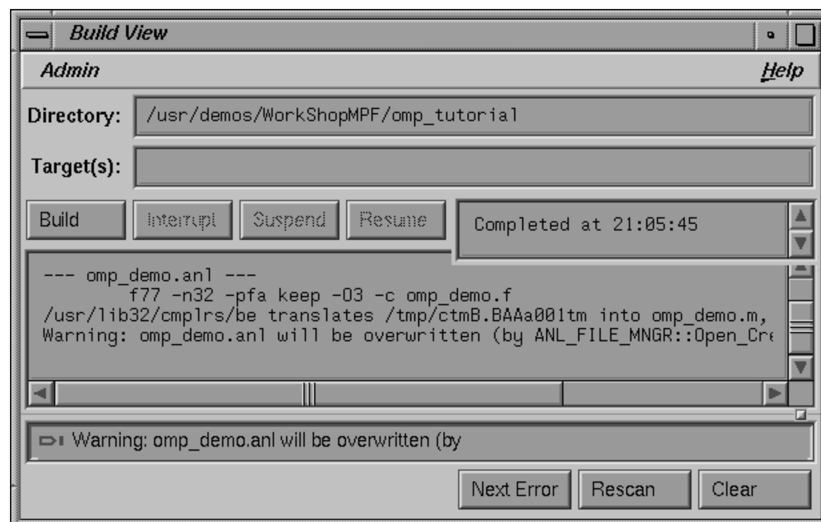


Figure 2-20 Build View of Build Manager

Note: If you edited any files, verify when the Build Manager comes up that the directory shown is the directory in which you are running the sample session; if not, change it.

Click the **Build** button in the Build Manager window, and the Build Manager reprocesses the changed file.

Examining the Modified Source File

When the build completes, the Parallel Analyzer View updates to reflect the changes that were made. You can now examine the new version of the file to see the effect of the requested changes.

Added Assertion

Scroll to Olid 14 to see the effect of the assertion request made in "Adding New Assertions or Directives With the Operations Menu", page 45. Notice the icon indicating that loop Olid 14, which previously was unparallelizable because of the call to `RTC()`, is now parallel.

Double-click the line and note the new loop information. The source code also has the assertion that was added.

Move to the next loop by clicking the **Next Loop** button.

Deleted Assertion

Note that the assertion in loop Olid 15 is gone, as requested in "Deleting Assertions or Directives", page 46, and that the loop no longer runs in parallel. Recall that the loop previously had the assertion that `foo()` was not an obstacle to parallelization.

Examples Using OpenMP Directives

This section examines the subroutine `ompdummy()`, which contains four parallel regions and a serial section that illustrate the use of OpenMP directives:

- "Explicitly Parallelized Loops: `C$OMP DO`", page 51

- "Loops With Barriers: `C$OMP BARRIER`", page 53
- "Critical Sections: `C$OMP CRITICAL`", page 55
- "Single-Process Sections: `C$OMP SINGLE`", page 55
- "Parallel Sections: `C$OMP SECTIONS`", page 55

For more information on OpenMP directives, see your compiler documentation or the OpenMP Architecture Review Board Web site: `http://www.openmp.org`.

Go to the first parallel region of `ompdummy()` by scrolling down the loop list, or using the Search field and entering **`parallel`**.

To select the first parallel region, double-click the highlighted line in the loop list, Olid 72.

Explicitly Parallelized Loops: `C$OMP DO`

The first construct in subroutine `ompdummy()` is a parallel region containing two loops that are explicitly parallelized with `C$OMP DO` directives. With this construct in place, the loops can execute in parallel, that is, the second loop can start before all iterations of the first complete.

Example 2-18 Explicitly Parallelized Loop Using `C$OMP DO`

```
C$OMP PARALLEL SHARED(a,b)
C$OMP DO SCHEDULE(DYNAMIC, 10-2*2)
        DO 6001 I=-100,100
           A(I) = I
6001   CONTINUE
C$OMP DO SCHEDULE(STATIC)
        DO 6002 I=-100,100
           B(I) = 3 * A(I)
6002   CONTINUE
C$OMP END PARALLEL
```

Notice in Figure 2-21, page 52, that the controls in the loop information display are now labelled Region Controls. The controls now affect the entire region. The **Keep option** button and the highlight buttons function the same way they do in the Loop Parallelization Controls.

Figure 2-21 Loops Explicitly Parallelized Using C$OMP DO

Click **Next Loop** twice to step through the two loops. Notice in the Source View that both loops contain a `C$OMP DO` directive.

Click **Next Loop** to step to the second parallel region.

Loops With Barriers: `C$OMP BARRIER`

The second parallel region, Olid 75, contains a pair of loops that are identical to the previous example except for a barrier between them. Because of the barrier, all iterations of the first `C$OMP DO` loop must complete before any iteration of the second loop can begin.

Example 2-19 Loops Using `C$OMP BARRIER`

```
C$OMP PARALLEL SHARED(A,B)
C$OMP DO SCHEDULE(STATIC, 10-2*2)
        DO 6003 I=-100,100
           A(I) = I
6003   CONTINUE
C$OMP END DO NOWAIT
C$OMP BARRIER
C$OMP DO SCHEDULE(STATIC)
        DO 6004 I=-100,100
           B(I) = 3 * A(I)
6004   CONTINUE
C$OMP END PARALLEL
```

Click **Next Loop** twice to view the barrier region. (See Figure 2-22, page 54.)

Figure 2-22 Loops Using C$OMP BARRIER Synchronization

Click **Next Loop** twice to go to the third parallel region.

Critical Sections: **C$OMP CRITICAL**

Click **Next Loop** to view the first of the two loops in the third parallel region. This loop contains a critical section.

Example 2-20 Critical Section Using `C$OMP CRITICAL`

```
C$OMP DO
       DO 6005 I=1,100
C$OMP CRITICAL(S3)
           S1 = S1 + I
C$OMP END CRITICAL(S3)
6005  CONTINUE
```

Click **Next Loop** to view the critical section. The critical section uses a named locking variable (*S3*) to prevent simultaneous updates of *S1* from multiple threads. This is a standard construct for performing a reduction.

Move to the next loop by using **Next Loop**.

Single-Process Sections: **C$OMP SINGLE**

This loop has a single-process section, which ensures that only one thread can execute the statement in the section. Highlighting in the Source View shows the begin and end directives.

Example 2-21 Single-Process Section Using `C$OMP SINGLE`

```
       DO 6006 I=1,100
C$OMP SINGLE
           S2 = S2 + I
C$OMP END SINGLE
6006  CONTINUE
```

Click **Next Loop** to view information about the single-process section.

Move to the final parallel region in `ompdummy()` by clicking the **Next Loop** button.

Parallel Sections: **C$OMP SECTIONS**

The fourth and final parallel region of `ompdummy()` provides an example of parallel sections. In this case, there are three parallel subsections, each of which calls a

function. Each function is called exactly once, by a single thread. If there are three or more threads in the program, each function may be called from a different thread. The compiler treats this directive as a single-process directive, which guarantees correct semantics.

Example 2-22 Parallel Sections Using `C$OMP SECTIONS`

```
C$OMP PARALLEL SHARED(A,C) PRIVATE(I,J)
C$OMP SECTIONS
        call boo
C$OMP SECTION
        call bar
C$OMP SECTION
        call baz
C$OMP END SECTIONS
C$OMP END PARALLEL
```

Click **Next Loop** to view the entire `C$OMP SECTIONS` region.

Click **Next Loop** to view a `C$OMP SECTION` region.

Move to the next subroutine by clicking **Next Loop** twice.

Examples Using Data Distribution Directives

The next series of subroutines illustrate directives that control data distribution and cache storage. The following three directives are discussed:

- "Distributed Arrays: `C$SGI DISTRIBUTE` ", page 56
- "Distributed and Reshaped Arrays: `C$SGI DISTRIBUTE_RESHAPE`", page 58
- "Prefetching Data From Cache: `C*$* PREFETCH_REF`", page 59

Distributed Arrays: `C$SGI DISTRIBUTE`

When you select the subroutine `dst1d()`, a directive is listed in the loop information display that is global to the subroutine. The directive, `C$SGI DISTRIBUTE`, specifies placement of array members in distributed, shared memory. (See Figure 2-23, page 57.)

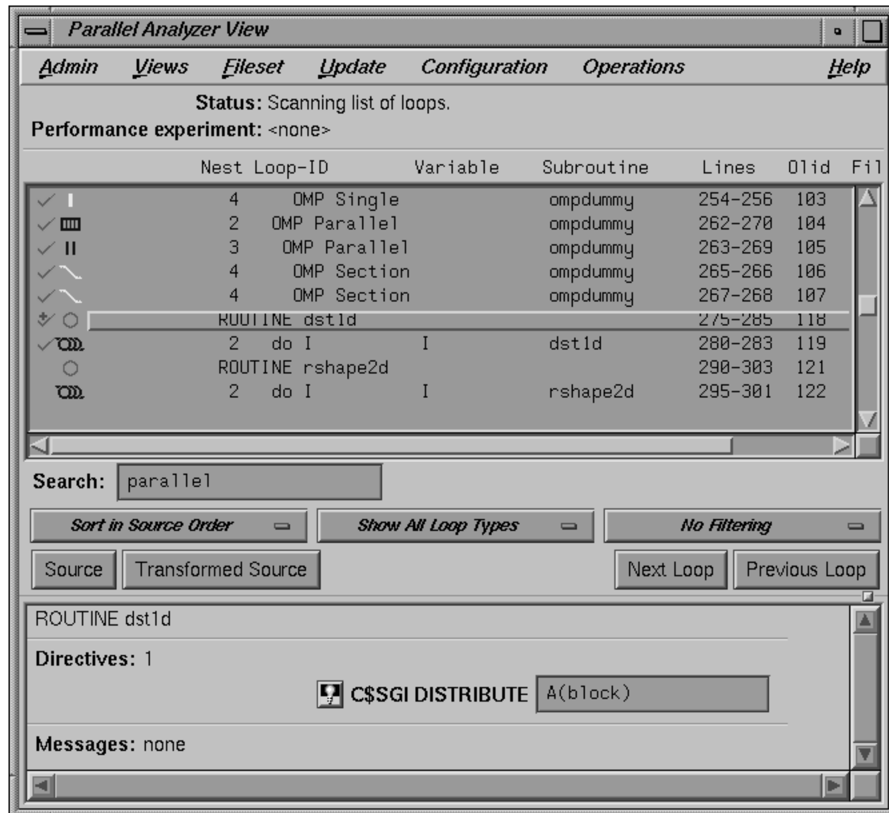


Figure 2-23 `C$SGI DISTRIBUTE` Directive and Text Field

In the editable text field adjacent to the directive name is the argument for the directive, which in this case distributes the one-dimensional array *a(m)* among the local memories of the available processors. To highlight the directive in the Source View, click the highlight button.

Click **Next Loop** to move to the parallel loop.

The loop has a `C$OMP PARALLEL DO` directive, which works with `C$SGI DISTRIBUTE` to ensure that each processor manipulates locally stored data.

Example 2-23 Distributed Array Using `C$SGI DISTRIBUTE`

```
        subroutine dst1d(a)

        parameter (m=10)
        real a(m)
C$DISTRIBUTE a(BLOCK)
C$OMP PARALLEL DO
        do i=1,m
          a(i)= i
        end do

        return
```

You can highlight the `C$OMP PARALLEL DO` directive in the Source View with either of the highlight buttons in the loop information display. If you use the highlight button in the Loop Parallelization Controls, the Parallelization Control View presents more information about the directive and allows you to change the `C$OMP PARALLEL DO` clauses. In this example, it confirms what you see in the code: that the index variable `i` is local.

Click **Next Loop** again to view the next subroutine.

Distributed and Reshaped Arrays: `C$SGI DISTRIBUTE_RESHAPE`

When you select the subroutine `rshape2d()`, the subroutine's global directive is listed in the loop information display. The directive, `C$SGI DISTRIBUTE_RESHAPE`, also specifies placement of array members in distributed, shared memory. It differs from the directive `C$SGI DISTRIBUTE` in that it causes the compiler to reorganize the layout of the array in memory to guarantee the desired distribution. Furthermore, the unit of memory allocation is not necessarily a page.

In the text field adjacent to the directive name is the argument for the directive, which in this case distributes the columns of the two-dimensional array *b(m,m)* among the local memories of the available processors. To highlight the directive in the Source View, click the highlight button.

Click the **Next Loop** button to move to the parallel loop.

The loop has a `C$OMP PARALLEL DO` directive (Example 2-24), which works with `C$SGI DISTRIBUTE_RESHAPE` so that each processor manipulates locally stored data.

Example 2-24 Distributed and Reshaped Array Using `C$SGI DISTRIBUTE_RESHAPE`

```
        subroutine rshape2d(b)
        parameter (m=10)
        real b(m,m)

C$DISTRIBUTE_RESHAPE b(*,BLOCK)
C$OMP PARALLEL DO
C$SGI&NEST (i,j)
        do i=1,m
           do j=1,m
              b(i,j)= i*j
           end do
        end do
        return
```

If you use the highlight button in the Loop Parallelization Controls, the Parallelization Control View presents more information. In this example, it confirms what you see in the code: that the index variable *i* is local, and that the nested loop can be run in parallel.

If the code had not had the `C$SGI&NEST` clause, you could have inserted it by supplying the arguments in the text field in the Parallelization Control View. You can use the `C$SGI&NEST` clause to parallelize nested loops only when both loops are fully parallel and there is no code between either the `do-i` and `do-j` statements or the `enddo-i` and `enddo-j` statements.

Click **Next Loop** to move to the nested loop. Notice that this loop has an icon in the loop list and in the loop information display indicating that it runs in parallel.

Click **Next Loop** to view the next subroutine, `prfetch()`.

Prefetching Data From Cache: `C*$* PREFETCH_REF`

Click **Next Loop** to go to the first loop in `prfetch()`. The compiler switched the order of execution of the nested loops, Olid 128 and 129. To see this, look at the Transformed Source view.

Example 2-25 Prefetching Data From Cache Using `C*$* PREFETCH_REF`

```
      subroutine prfetch(a, b, n)

      integer*4 a(n, n), b(n, n)
      integer i, j, n

      do i=1, n
         do j=1, n
C*$*PREFETCH_REF = b(i,j), STRIDE=2,2 LEVEL=1,2 KIND=rd, SIZE=4
              a(i,j) = b(i,j)
           end do
        end do
```

Click **Next Loop** to move to the nested loop. The list of directives in the loop information display shows `C*$* PREFETCH_REF` with a highlight button to locate the directive in the Source View. The directive allows you to place appropriate portions of the array in cache.

Exiting From the `omp_demo.f` Sample Session

This completes the first sample session.

Quit the Parallel Analyzer View by choosing **Admin** > **Exit**.

Not all windows opened during the session close when you quit the Parallel Analyzer View. In particular, the Source View remains open because all the tools interoperate, and other tools may share the Source View window. You must close the Source View independently.

To clean up the directory, so that the session can be rerun, enter the following in your shell window to remove all of the generated files:

```
% make clean
```

Tutorial: Examining Loops for Fortran 90 Code

This chapter presents an interactive tutorial using the Fortran 90 compiler. It illustrates how the MIPSpro compiler transforms Fortran 90 arrays into loops.

Analyzing a Fortran 90 program is very similar to analyzing a FORTRAN 77 program. See the previous chapter for reference information that applies to both compilers.

The following topics are discussed in this tutorial:

- "Compiling the Sample Code", page 61.
- "Demonstrating Array Statement Transformations", page 62.

Before starting this sample session, make sure `ProMP.sw.demos` is installed. The sample session uses the source file `f90_tutorial_f90_orig` in the directory `/usr/demos/ProMP/f90_tutorial`. The file `Makefile` compiles the source file.

The source file contains array statements, each of which exemplifies an aspect of the parallelization process.

Compiling the Sample Code

Prepare for the session by entering the following in a shell window:

```
% cd /usr/demos/ProMP/f90_tutorial
% make
```

These commands create the following files:

- `f90_tutorial.f90`: a copy of the demonstration program created by copying `f90_tutorial.f90_orig`.
- `f90_tutorial.m`: a transformed source file, which you can view with the **Parallel Analyzer View** and print
- `f90_tutorial.l`: a listing file.
- `f90_tutorial.anl`: an analysis file used by the **Parallel Analyzer View**

After you have created the files, start the session by entering the `cvpav`(1) command. The command opens the main window of the **Parallel Analyzer View** and loads the sample file data.

```
% cvpav -f f90_tutorial.f90
```

Open the **Source View** window by clicking the **Source** button once the main window opens.

Demonstrating Array Statement Transformations

This section demonstrates the following transformations:

- "Transforming an Array Statement into a DO Loop", page 62.
- "Transforming an Array Statement in Nested DO Loops", page 63.
- "Transforming an Array Statement into a Subroutine", page 65.

Transforming an Array Statement into a DO Loop

To continue the tutorial begun in the last section, go to loop 5 in the **Parallel Analyzer View** window and double-click the highlighted line in the loop list. First double-click the **Source** button, and then double-click the **Transformed Source** button.

Notice in the **Transformed Source** window that the following array statement has been transformed into a DO loop:

```
logical*1 l(12),r,r1

l = .true.
```

The **Transformed Loops View** window (see Figure 3-1, page 63) identifies line 40 from the source as a Fortran 90 array statement. It notes that a loop was generated but indicates that the loop array statement was not made parallel because it contains too little work.

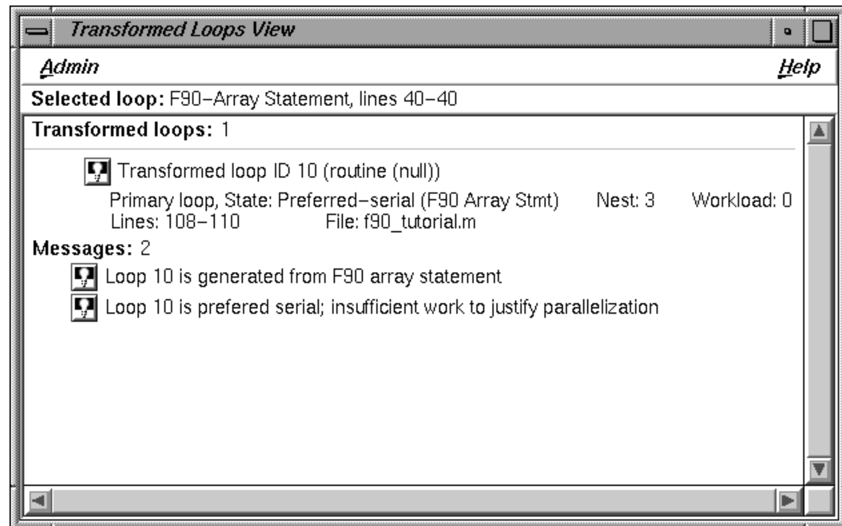


Figure 3-1 Array Statement into DO Loop

Transforming an Array Statement in Nested DO Loops

Pull down the **Show All Loop Types** menu and click on **Show Fortran 90 Array Stmts**. Only the Fortran 90 arrays statements that were transformed into DO loops are displayed.

The following is the array statement in the source:

```
logical*8 l(3,12)
.
.
.
l = .true.
```

Because the array has two dimensions, two nested DO loops are generated. Double-click first on loop 22, then on loop 23. They are the two new loops generated from the array statement. The **Transformed Loops View** window gives information on each loop. (See Figure 3-2, page 64 for loop 22 and Figure 3-3, page 65 for loop 23.)

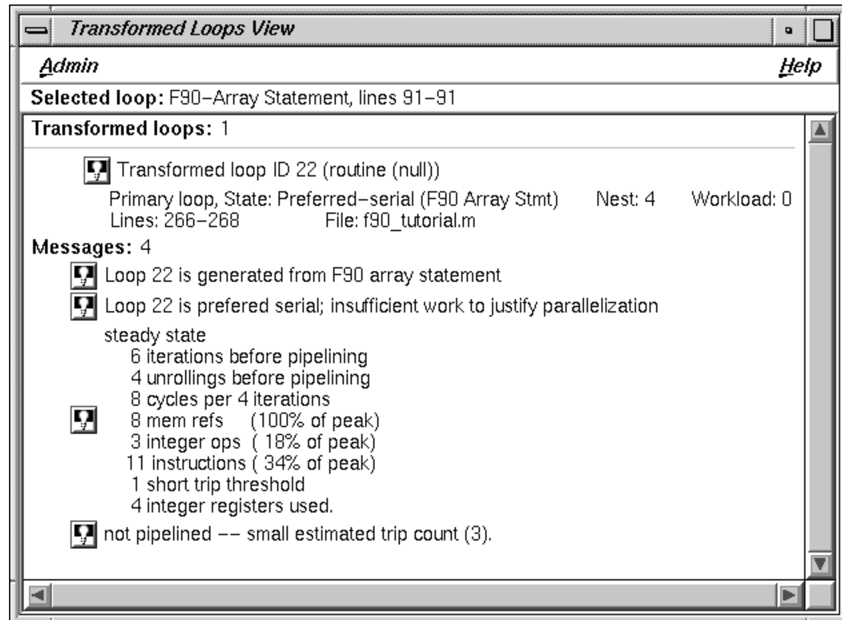


Figure 3-2 Loop 22

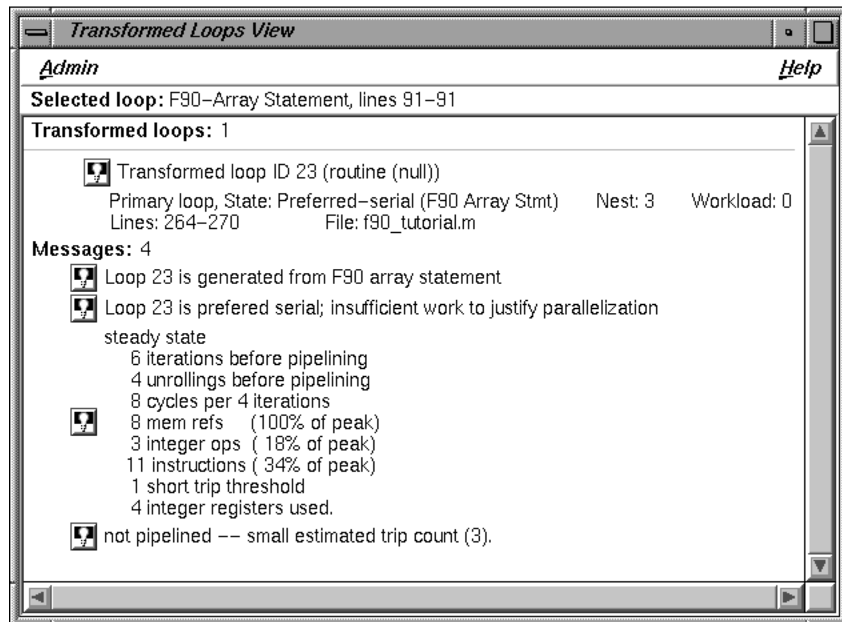


Figure 3-3 Loop 23

Transforming an Array Statement into a Subroutine

Click on loop 26. Notice in the **Transformed Source** window how the following sliced array statement is transformed into an `OMP PARALLEL DO` statement, which will itself be converted into a subroutine:

```
r(:il*2:2) = all(l,id)
```

The **Transformed Loops View** (see Figure 3-4, page 66) shows the process of converting first to a parallel loop and then to a subroutine:

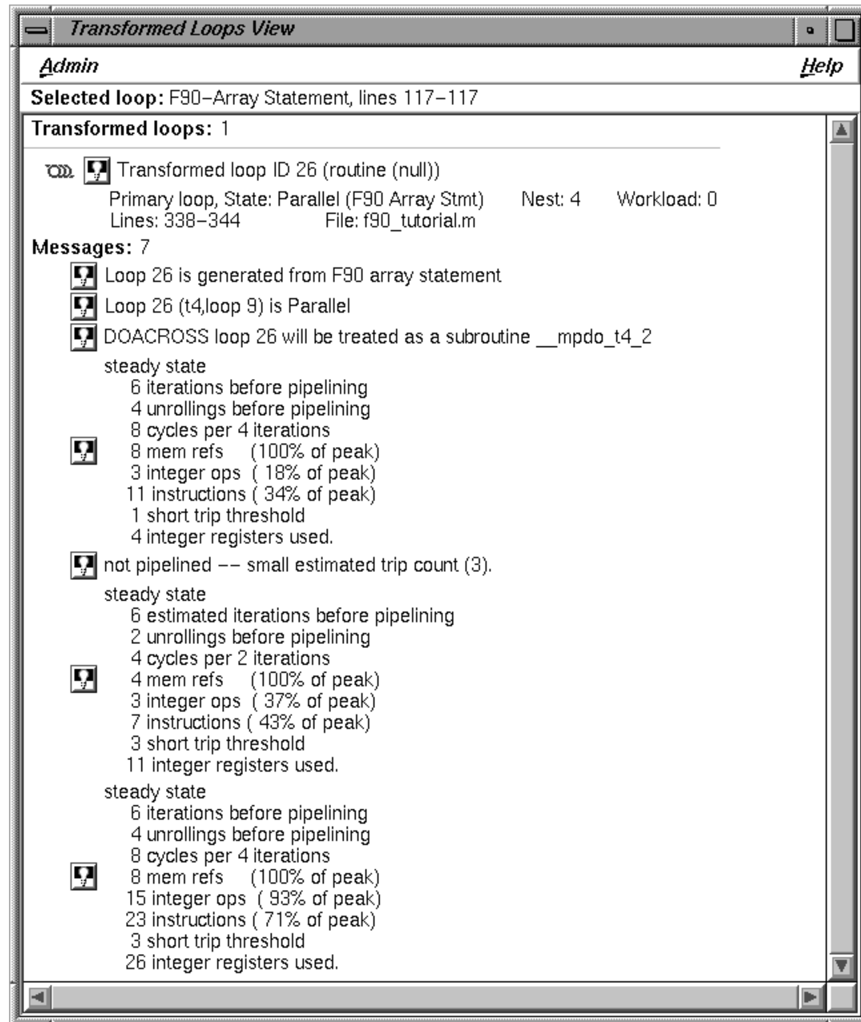


Figure 3-4 Array Statement into a Subroutine

Exiting From the Session

This completes the session. Quit the **Parallel Analyzer View** by choosing **Admin > Exit** and close any windows that may still be open.

To clean up the directory so that the session can be rerun, enter the following in your shell window:

```
% make clean
```

Tutorial: Examining Loops for C Code

This chapter presents another interactive sample session with the **Parallel Analyzer View**. The session illustrates aspects of the MIPSpro C compiler.

Analyzing a C program is very similar to analyzing a Fortran program. See Chapter 1, "Getting Started With ProMP", page 1, for reference information that applies to both languages.

The following sections are included in this tutorial:

- "Compiling the Sample Code", page 70
- "Examples of Simple Loops", page 70.
- "Examining Loops With Obstacles to Parallelization", page 74.
- "Examining Nested Loops", page 80.
- "Modifying Source Files and Compiling", page 82.
- "Examples Using OpenMP Directives", page 90.
- "Examples Using Data Distribution Directives", page 93.

The topics are introduced in this chapter by going through the process of starting the ProMP Parallel Analyzer View and stepping through the loops and routines in the sample code. The chapter is most useful if you perform the operations as they are described.

For more details about the ProMP interface, see Chapter 6, "Parallel Analyzer View Reference", page 107.

To use the sample sessions discussed in this guide, note the following:

- `/usr/demos/ProMP` is the demonstration directory
- `ProMP.sw.demos` must be installed

The sample session discussed in this chapter uses the `c_tutorial.c_orig` file in the directory `/usr/demos/ProMP/c_tutorial`. The source file contains many loops, each of which exemplifies an aspect of the parallelization process.

The directory `/usr/demos/ProMP/c_tutorial` also includes `Makefile` to compile the source files.

Compiling the Sample Code

Prepare for the session by opening a shell window and entering the following:

```
% cd /usr/demos/ProMP/c_tutorial
% make
```

These commands create the following files:

- `c_tutorial.c` from `c_tutorial.c_orig`
- `c_tutorial.m`: a transformed source file, which you can view with the **Parallel Analyzer View**, and print
- `c_tutorial.l`: a listing file
- `c_tutorial.anl`: an analysis file used by the **Parallel Analyzer View**

After you have the appropriate files from the compiler, start the session by entering the `cvpav`(1) command, which opens the main window of the **Parallel Analyzer View** loaded with the sample file data:

```
% cvpav -f c_tutorial.c
```

If at any time during the tutorial you should want to restart from the beginning, do the following:

- Quit the **Parallel Analyzer View** by choosing **Admin > Exit** from the menu bar.
- Clean up the tutorial directory by entering the following command:

  ```
  % make clean
  ```

Examples of Simple Loops

The loops in this section are the simplest kinds of C loops:

- "Simple Parallel Loop", page 71.
- "Serial Loop", page 71.
- "Explicitly Parallelized Loop", page 72.
- "Fused Loops", page 74.
- "Loop That Is Eliminated", page 74.

Two other sections discuss more complicated loops:

- "Examining Loops With Obstacles to Parallelization", page 74.
- "Examining Nested Loops", page 80.

Note: The loops in the next sections are referred to by their Olid numbers. Changes to the **Parallel Analyzer View**, such as, the implementation of updated OpenMP standards, may cause the Olid numbers you see on your system to differ from those in the tutorial. The Olid numbers in the tutorial are not in the same order as in the program. Example code, which you can find in the **Source View**, is included in the tutorial to clarify the discussion.

Simple Parallel Loop

Scroll to the top of the list of loops and select loop Olid 5, either by advancing by using the **Next Loop** and **Previous Loop** buttons or by double-clicking the line at the top of the display.

Example 4-1 C: simple parallel loop

```
nsize = sizeof(a);
for (i = 0; i < nsize; i++) {
  a[i] = b[i]*c[i];
}
```

This is a simple loop; computations in each iteration are independent of each other. It was transformed by the compiler to run concurrently. Notice in the **Transformed Source** window the directives added by the compiler.

Move to the next loop by selecting Olid 6.

Serial Loop

Olid 6 is a simple loop with too little content to justify running it in parallel. The compiler determined that the overhead of parallelizing would exceed the benefits; the original loop and the transformed loop are identical.

Example 4-2 C: serial loop

```
nsize = ARRAYSIZE;
for (i = 0; i < ARRAYSIZE; i++) {
  a[i] = b[i]*c[i];
}
```

Move to the Olid 2 loop.

Explicitly Parallelized Loop

Loop Olid 2 is parallelized because it contains an explicit `#pragma omp parallel for` directive in the source, as shown in the Loop Parallelization Controls area of the window (see Figure 4-1, page 73). The compiler passes the directive through to the transformed source.

Example 4-3 C: explicitly parallelized loop

```
#pragma omp parallel for shared(a,b,c)
        for (i = 0; i < nsize; i++)
                a[i] = b[i]*c[i];
```

The loop parallelization status option button is set to **#pragma omp parallel for...**, and it is shown with a highlight button. Clicking the highlight button brings up both the **Source View** and the **Parallelization Control View**, which shows more information about the parallelization directive.

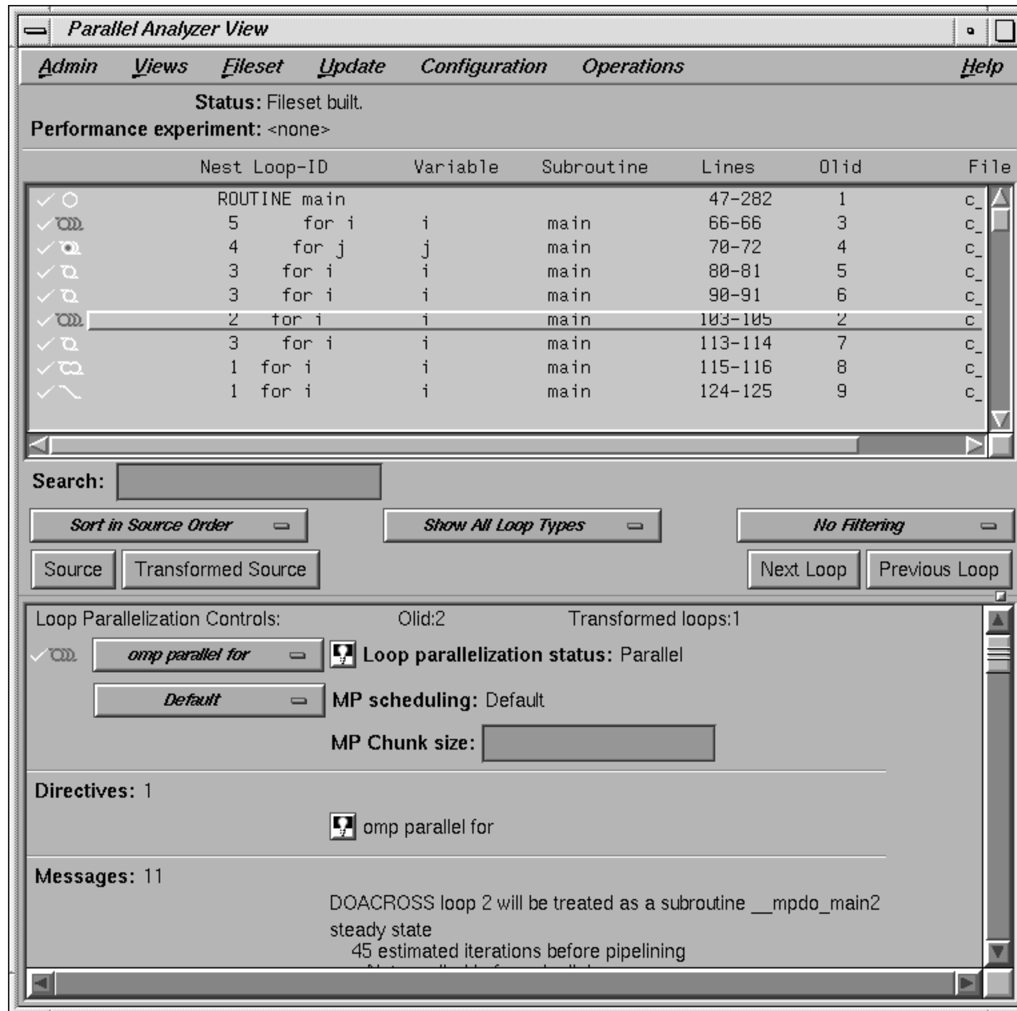


Figure 4-1 Explicitly Parallelized Loop

If you clicked on the highlight button, close the **Parallelization Control View**. (Using the **Parallelization Control View** is discussed in "Adding `#pragma omp parallel for` Directives and Clauses", page 83.) Close the **Source View** and move to the next loop by clicking the **Next Loop** button.

Fused Loops

Loops Olid 7 and Olid 8 are simple parallel loops that have similar structures. The compiler combines these loops to decrease overhead. Note that loop Olid 8 is described as fused in the loop information display, and in the **Transformed Loops View**, it is incorporated into Olid 7. If you look at the **Transformed Source** window and select Olid 7 and Olid 8, the same lines of code are highlighted for each loop.

Example 4-4 C: fused loops

```
nsize = sizeof(a);
for (i = 0; i < nsize; i++)
        a[i] = b[i]+c[i];
for (i = 0; i < nsize; i++)
        a[i] = b[i]+c[i];
```

Move to the next loop by clicking **Next Loop** twice.

Loop That Is Eliminated

Loop Olid 9 is an example of a loop that the compiler can eliminate entirely. The compiler determines that the body is independent of the rest of the loop. It moves the body outside of the loop and eliminates the loop. The transformed source is not scrolled and highlighted when you select Olid 9 because there is no transformed loop derived from the original loop.

Example 4-5 C: eliminated loop

```
nsize = sizeof(a);
for (i = 0; i < nsize; i++)
        xx = 10.0;
```

Move to the next loop, Olid 10, by clicking the **Next Loop** button. This loop is discussed in "Unparallelizable Carried Data Dependence", page 75.

Examining Loops With Obstacles to Parallelization

There are a number of reasons why a loop may not be parallelized. The loops in the following sections illustrate some of the reasons, along with variants that allow parallelization:

- Carried Data Dependence, see"Obstacles to Parallelization: Carried Data Dependence", page 75.
- Input/Output Operations, see "Obstacles to Parallelization: I/O Operations", page 79.
- Function Calls, see "Obstacles to Parallelization: Function Calls", page 79.
- Permutation Vectors, see "Obstacles to Parallelization: Permutation Vectors", page 80.

These loops are a few specific examples of the obstacles to parallelization recognized by the compiler.

Messages that appear in the graphical user interface offer further tips on obstacles to parallelization. See "Obstacles to Parallelization Messages", page 35 for two tables that list messages generated by the compiler that concern obstacles to parallelization.

Obstacles to Parallelization: Carried Data Dependence

Carried data dependence typically arises when recurrence of a variable occurs in a loop. Depending on the nature of the recurrence, parallelizing the loop may be impossible. The following loops illustrate four kinds of data dependence:

- Unparallelizable Carried Data Dependence, see "Unparallelizable Carried Data Dependence", page 75.
- Parallelizable Carried Data Dependence, see "Parallelizable Carried Data Dependence", page 77.
- Multi-line Data Dependence, see "Multi-line Data Dependence", page 78.
- Reductions, see "Reductions", page 78.

Unparallelizable Carried Data Dependence

Loop Olid 10 is a loop that cannot be parallelized because of a data dependence; one element of an array is used to set another in a recurrence.

```
nsize = sizeof(a);
for (i = 0; i < nsize -1; i++)
        a[i] = a[i+1];
```

If the loop were nontrivial (if `nsize` were greater than two) and if the loop were run in parallel, iterations might execute out of order. For example, iteration 4, which sets `a[4]` to `a[5]`), might occur after iteration 5, which resets the value of `a[5]`; the computation would be unpredictable.

The loop information display in Figure 4-2, page 77, lists the obstacle to parallelization.

Click the highlight button that accompanies it. Two kinds of highlighting occur in the **Source View**:

- The relevant line that has the dependence.
- The uses of the variable that obstruct parallelization; only the uses of the variable within the loop are highlighted.

Move to the next loop by clicking **Next Loop**.

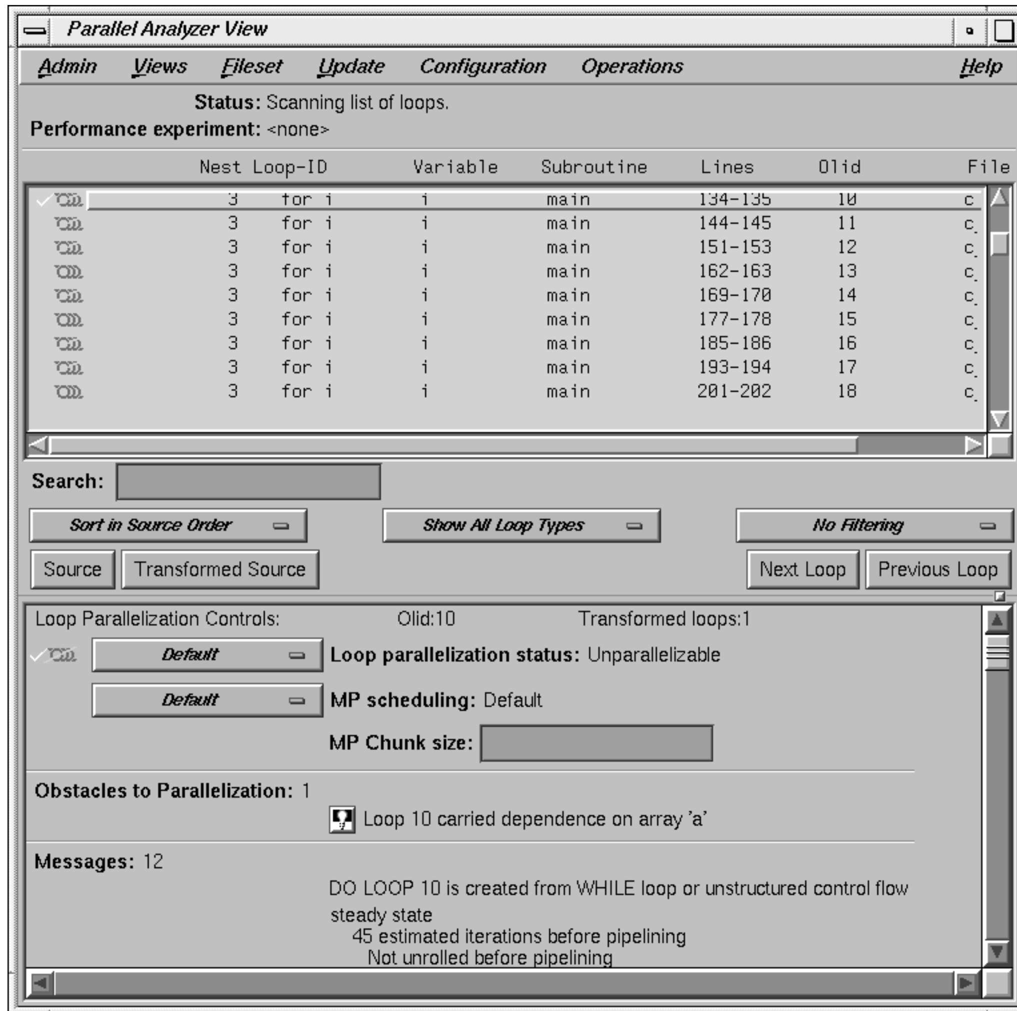


Figure 4-2 Obstacles to Parallelization

Parallelizable Carried Data Dependence

Loop Olid 11 has a structure similar to loop Olid 10. Despite the similarity, however, Olid 11 can be parallelized.

```
nsize = sizeof(a);
#pragma concurrent
for (i = 0; i < nsize ; i++)
        a[i]= a[i+m];
```

Note that the array indices differ by offset `m`. If `m` is equal to `nsize` and the array is twice `nsize`, the code is actually copying the upper half of the array into the lower half, a process that can be run in parallel. The compiler cannot recognize this from the source, but the code has the assertion `#pragma concurrent`, so the loop is parallelized.

Click the highlight button to show the assertion in the **Source View**.

Move to the next loop by clicking the **Next Loop** button.

Multi-line Data Dependence

Data dependence can involve more than one line of a program. In loop Olid 12, a dependence similar to that in Olid 11 occurs, but the variable is set and used on different lines.

```
nsize = sizeof(a);
for (i = 0; i < nsize-1; i++) {
        b[i] = a[i];
        a[i+1] = b[i];
}
```

Click the highlight button on the obstacle line.

In the **Source View**, highlighting shows the dependency variable on two lines. Of course, real programs usually have far more complex dependences than this.

Move to the next loop by clicking **Next Loop.**

Reductions

Loop Olid 13 shows a data dependence that is called a reduction: the variable responsible for the data dependence is being accumulated or *reduced* in some fashion. A reduction can be a summation, a multiplication, or a minimum or maximum determination. For a summation, as shown in this loop, the code could accumulate partial sums in each processor and then add the partial sums at the end.

```
nsize = array_size;
x = 0;
for (i = 0; i < nsize; i++)
        x =  b[i]*c[i] + x;
```

However, because floating-point arithmetic is inexact, the order of addition might give different answers due to roundoff error. This does not imply that the serial execution answer is correct and the parallel execution answer is incorrect; they are equally valid within the limits of roundoff error. With the `-O3` optimization level, the compiler assumes it is permissible to introduce roundoff error, and it parallelizes the loop. If you do not want a loop parallelized because of the difference caused by roundoff error, compile with the `-OPT:roundoff=0` or `-OPT:roundoff=1` option.

Move to the next loop by clicking **Next Loop**.

Obstacles to Parallelization: I/O Operations

Loop Olid 14 has an input/output (I/O) operation in it. It cannot be parallelized because the output would appear in a different order, depending on the scheduling of the individual CPUs.

```
for (i = 0; i < nsize; i++)
        printf( "Element A[%d] = %f\n",i,a[i]);
```

Click the button indicating the obstacle and note the highlighting of the print statement in the **Source View**.

Move to the next loop by clicking **Next Loop**.

Obstacles to Parallelization: Function Calls

Unless you make an assertion, a loop with a function call cannot be parallelized; the compiler cannot determine whether a call has side effects, such as creating data dependencies.

Although loop Olid 15 has a function call, it can be parallelized. You can add an assertion that the call has no side effects that will prevent concurrent processing.

```
nsize = sizeof(ARRAYSIZE);
#pragma concurrent call
for (i = 0; i < nsize; i++)
        a[i] = b[i] + foo();
```

Click the highlight button on the assertion line in the loop information display to highlight the line in the **Source View** containing the assertion.

Move to the next loop by clicking **Next Loop**.

Obstacles to Parallelization: Permutation Vectors

If you specify array index values by values in another array (referred to as a *permutation vector*), the compiler cannot determine if the values in the permutation vector are distinct. If the values are distinct, loop iterations do not depend on each other, and the loop can be parallelized; if they are not distinct, the loop cannot be parallelized. Without an assertion, a loop with a permutation vector is not parallelized.

Unparallelizable Loop With a Permutation Vector

Loop Olid 16 has a permutation vector, `ic[i]`, and cannot be parallelized.

```
for (i = 0; i < nsize-1; i++)
        a[ic[i]] = a[ic[i]] + DELTA;
```

Move to the next loop by clicking the **Next Loop** button.

Parallelizable Loop With a Permutation Vector

An assertion, `#pragma permutation(ib)`, that the index array `ib[i]` is indeed a permutation vector has been added before loop Olid 17. Therefore, the loop is parallelized.

```
#pragma permutation(ib)
for (i = 0; i < nsize; i++)
        a[ib[i]] = a[ib[i]] + DELTA;
```

Move to the next loop, Olid 18, by clicking **Next Loop**. This loop is discussed in "Nested Loop", page 81.

Examining Nested Loops

The loops in this section illustrate more complicated situations, involving nested and interchanged loops.

Nested Loop

Loop Olid 18 is the outer loop of a pair of loops, and it runs in parallel. The inner loop runs in serial because the compiler knows that one parallel loop should not be nested inside another. However, you can force parallelization in for the inner loop by inserting a `#pragma omp parallel for` directive in front of the outer loop. For example, see "Distributed and Reshaped Arrays: `#pragma distribute_reshape`", page 96.

Example 4-6 C: nested loop

```
for (i = 0; i < nsize; i++) {
        for (j = 0; i < nsize; i++)
              aa[j][i] = bb[j][i];
```

Click **Next Loop** to move to Olid 19.

Doubly Nested Loop

The inner loop, Olid 20, is shown in the loop information display as a serial loop inside a parallel loop. Olid 19 is labelled as parallel. Explanatory messages appear in the loop information display.

Example 4-7 C: doubly nested loop

```
nsize = array_size;
for (i = 0; i < nsize; i++) {
        for (j = 0; j < nsize; j++)
                aa[i][j] = bb[i][j];
}
```

Move to the inner loop, Olid 20, by clicking the **Next Loop** button. Click **Next Loop** once again to move to the following triple-nested loop.

Triple Nested Loop

The following triple-nested loop, with Olids 21, 22, and 23, is transformed into two serial loops executing under parallel loop Olid 21:

Example 4-8 C: triple nested loop

```
for (i = 0; i < nsize; i++) {
        for (j = 0; j < nsize; j++)  {
                cc[i][j] = 0.0;
                for (k = 0; k < nsize; k++)
                        cc[i][j] = cc[i][j] + aa[i][k] * bb[k][j];
        }
}
```

Double-click on Olid 21, Olid 22, and Olid 23 in the loop list and note that the loop information display shows that Olid 22 and Olid 23 are serial loops inside a parallel loop, Olid 21.

Because the innermost serial loop, Olid 23, depends without recurrence on the indices of Olid 21 and Olid 22, iterations can run concurrently. The compiler does not recognize this possibility. This brings us to the subject of the next section, the use of the **Parallel Analyzer View** tools to modify the source.

Return to Olid 21, if necessary, by using the **Previous Loop** button.

Modifying Source Files and Compiling

So far, the discussion has focused on ways to view the source and parallelization effects. This section discusses controls that can change the source code by adding directives or assertions, allowing a subsequent pass of the compiler to do a better job of parallelizing your code.

You control most of the directives and some of the assertions available from the **Parallel Analyzer View** with the **Operations** menu . You control most of the assertions and the more complex directives, `#pragma omp for` and `#pragma omp parallel for`, with the loop parallelization status option button (see Figure 4-3, page 84).

There are two steps to modifying source files:

1. Making changes using the **Parallel Analyzer View** controls, discussed in the next subsection, "Making Changes", page 83.
2. Modifying the source and rebuilding the program and its analysis files, discussed in "Updating the Source File", page 89.

Making Changes

You make changes by one of the following actions:

- Adding or deleting assertions and directives using the **Operations** menu or the Loop Parallelization Controls.
- Adding clauses to or otherwise modifying directives using the **Parallelization Control View** window.
- Modifying the PFA analysis parameters in the **PFA Analysis Parameters View** (`o32` only.)

You can request changes in any order; there are no dependencies implied by the order of requests.

The following changes are discussed:

- "Adding `#pragma omp parallel for` Directives and Clauses", page 83.
- "Adding New Assertions or Directives With the Operations Menu", page 85.
- "Deleting Assertions or Directives", page 87.

Adding `#pragma omp parallel for` Directives and Clauses

Loop Olid 22, shown in "Triple Nested Loop", page 81, is a serial loop nested inside a parallel loop. It is not parallelized, but its iterations could run concurrently.

To add a `#pragma omp parallel for` directive to Olid 22, do the following:

1. Make sure loop Olid 22 is selected.
2. Click on the loop parallelization status option button (see Figure 4-3, page 84) and choose **omp parallel for** to parallelize Olid 22.

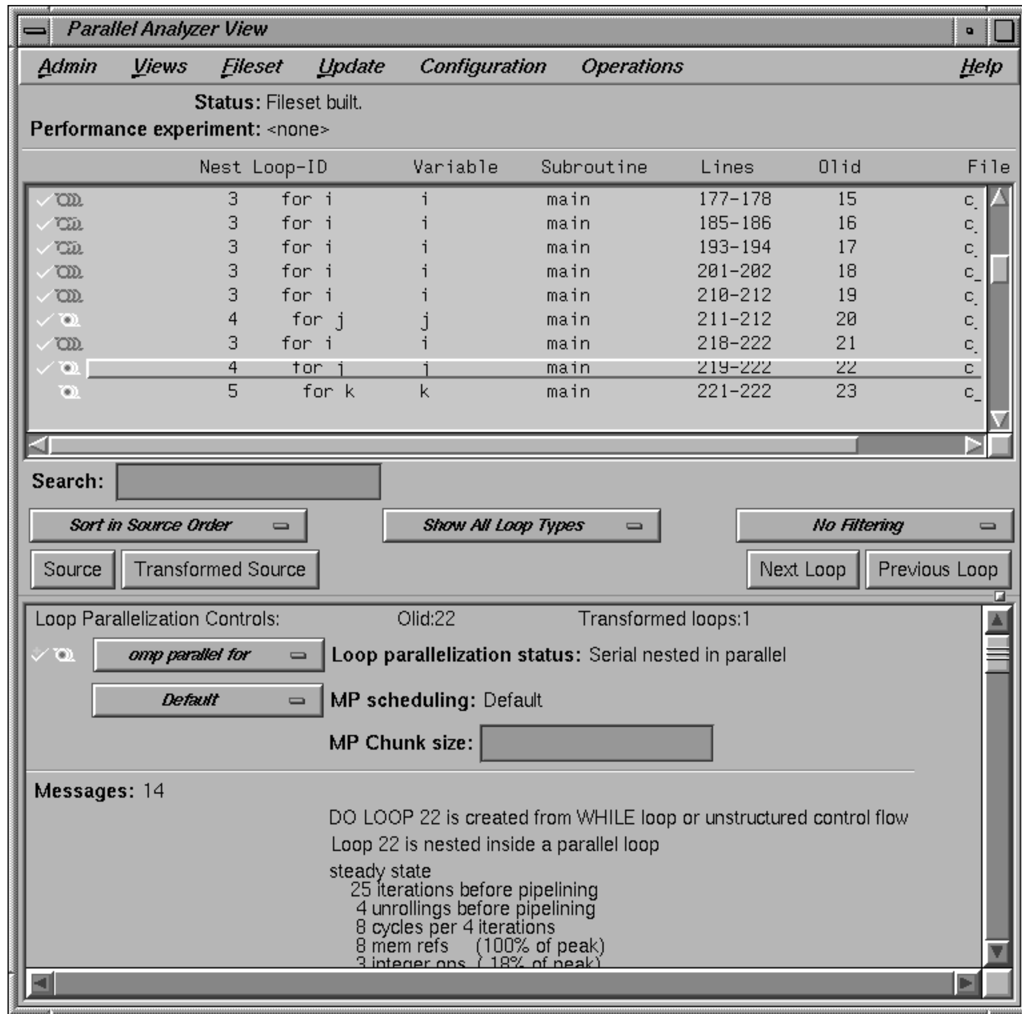


Figure 4-3 Creating a Parallel Directive

This sequence requests a change in the source code and opens the **Parallelization Control View** (see Figure 4-4, page 85). You can now look at variables in the loop and attach clauses to the directive, if needed.

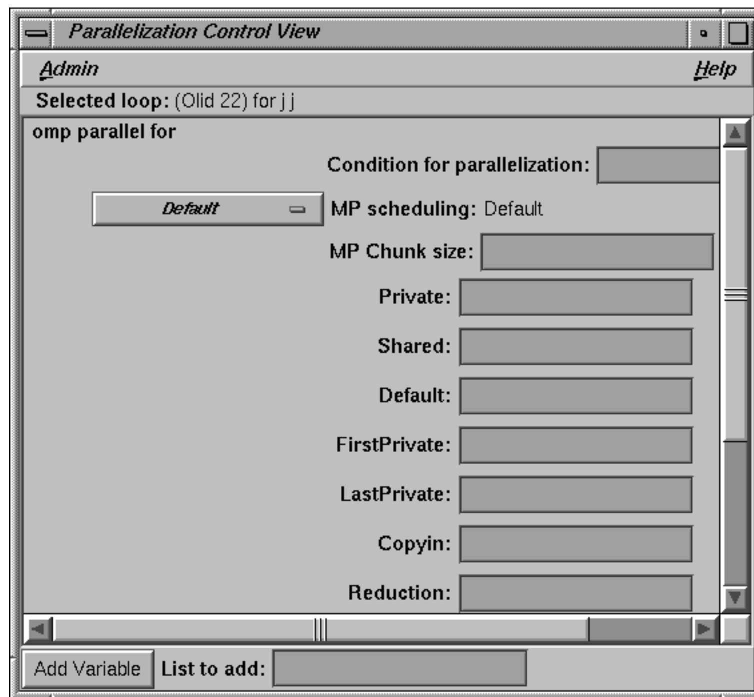


Figure 4-4 Parallelization Control View

Notice that in the loop list there is now a red plus sign next to this loop, indicating that a change has been requested.

Close the **Parallelization Control View** by using its **Admin > Close** option.

Adding New Assertions or Directives With the Operations Menu

To add a new assertion to a loop, do the following:

1. Find loop Olid 15 either by scrolling the loop list or by using the search feature. (Go to the Search field and enter `15`.)
2. Double-click the highlighted line in the loop list to select it.
3. Pull down **Operations > Add Assertion > ASSERT CONCURRENT CALL** to request a new assertion.

This adds the assertion `#pragma assert concurrent call`. The assertion indicates that it is safe to parallelize the loop despite the call to the function `foo`, which the compiler considers a possible obstacle to parallelization.

The loop information display shows the new assertion, along with an **Insert** button to indicate the state of the assertion when you modify the code. (See Figure 4-5.)

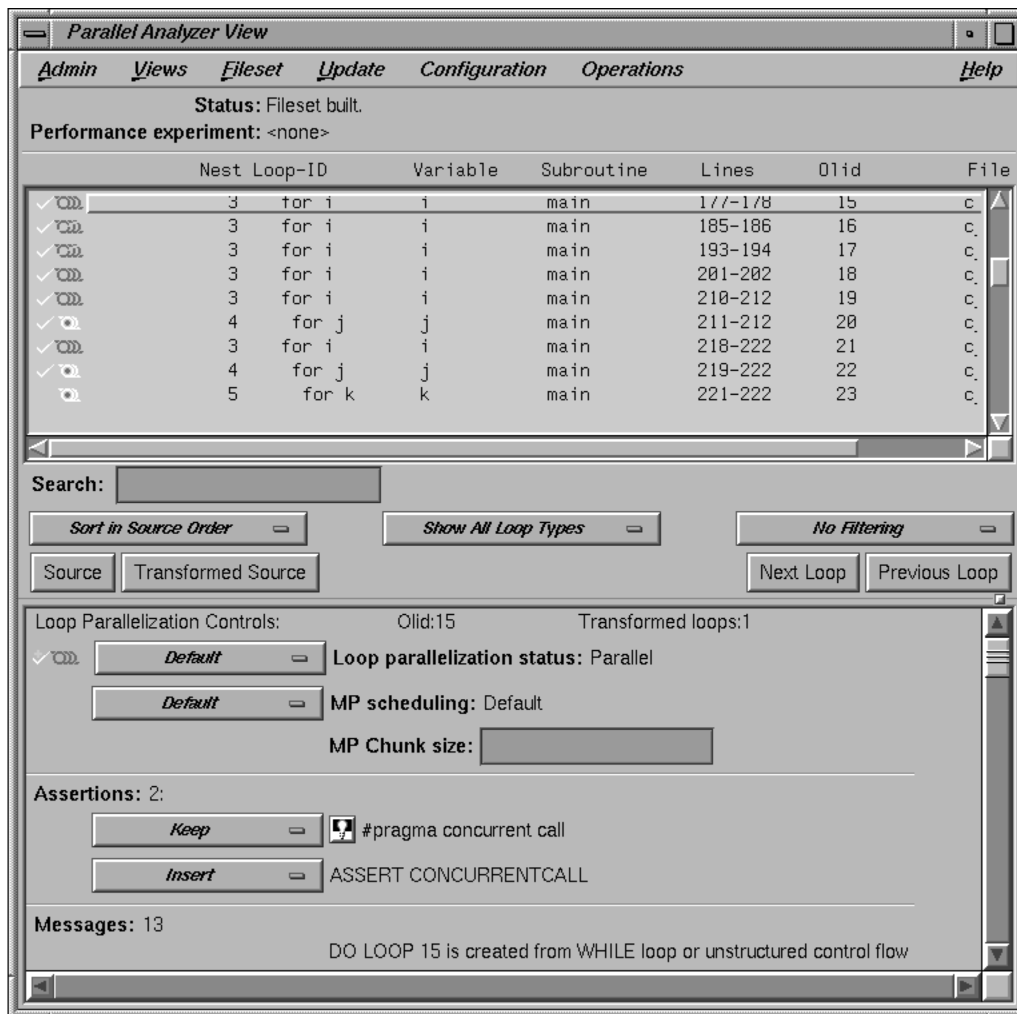


Figure 4-5 Adding an Assertion

The procedure for adding OpenMP directives is similar. To start, choose **Operations > Add OMP Directive**.

Deleting Assertions or Directives

Move to loop Olid 17 (shown in "Parallelizable Loop With a Permutation Vector", page 80).

To delete an assertion, follow these steps:

1. Find the assertion `#pragma permutation(ib)` in the loop information display.
2. Select its **Delete** option button.

Figure 4-6, page 88, shows the state of the assertion in the information display. A similar procedure is used to delete directives.

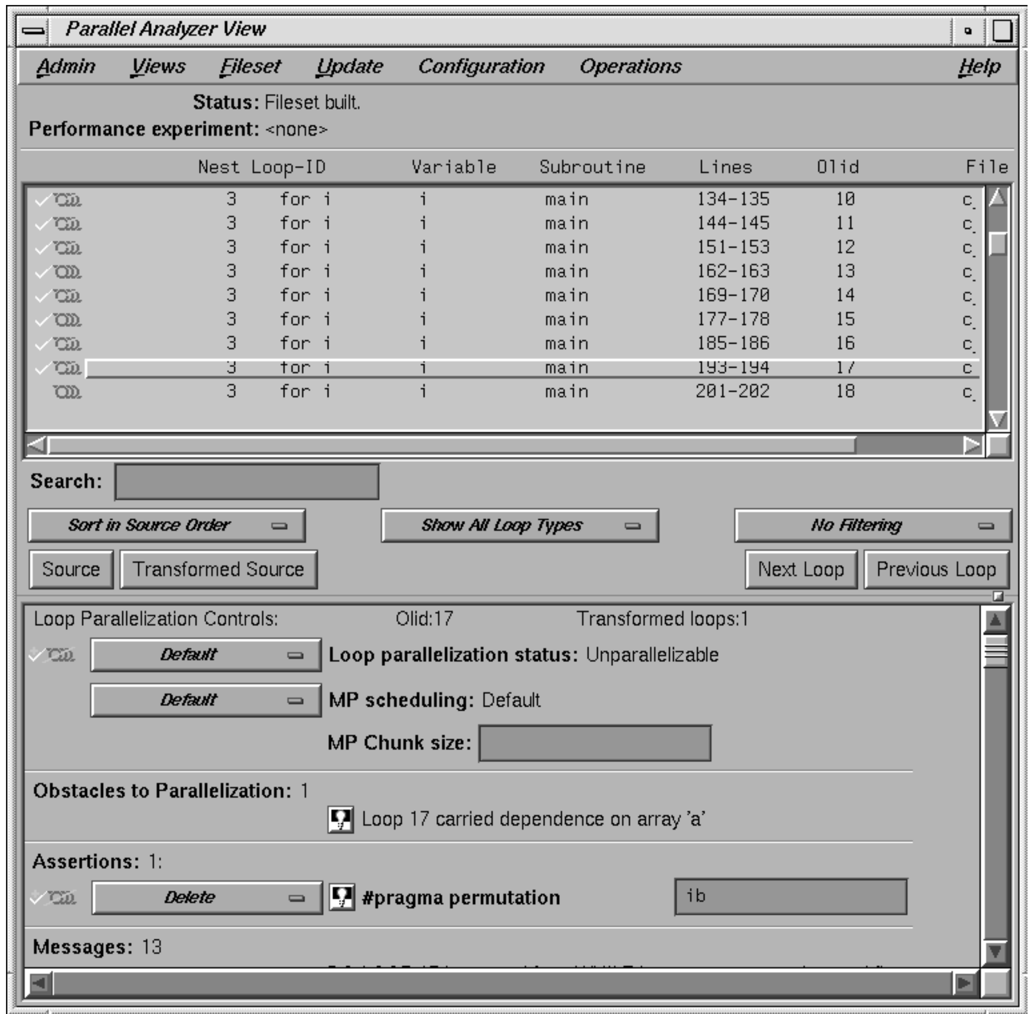


Figure 4-6 Deleting an Assertion

For information on applying changes and viewing the changes in a `gdiff` window, see "Updating the Source File", page 89.

Updating the Source File

Choose **Update > Update All Files** to update the source file to include the changes made in this tutorial. Alternatively, you can use the keyboard shortcut for this operation, **`Ctrl+U`**, with the cursor anywhere in the main view.

If you have set the checkbox and opened the `gdiff` window or an editor, examine the changes or edit the file as you wish. When you exit these tools, the **Parallel Analyzer View** spawns the WorkShop Build Manager.

Note: If you edited any files, verify when the Build Manager comes up that the directory shown is the one in which you are running the sample session; if the directory is different, change it.

Click the **Build** button in the Build Manager window, and the Build Manager will reprocess the changed file.

Examining the Modified Source File

When the build completes, the **Parallel Analyzer View** updates to reflect the changes. You can now examine the new version of the file to see the effect of the requested changes.

Added Assertion

Scroll to Olid 15 to see the effect of the assertion request made in "Adding New Assertions or Directives With the Operations Menu", page 85. Notice the icon indicating that loop Olid 15, which previously was unparallelizable because of the call to the function `foo`, is now parallel.

Double-click the line and note the new loop information. The source code also has the assertion that was added.

Move to the next loop by clicking the **Next Loop** button.

Deleted Assertion

Note that the assertion in loop Olid 16 is gone, as requested in "Deleting Assertions or Directives", page 87, and that the loop no longer runs in parallel. Recall that the loop previously had the assertion that `ib` was not an obstacle to parallelization.

Examples Using OpenMP Directives

This section examines the function `omp_demo`, which contains parallel regions and a serial section that illustrate the use of OpenMP directives:

- "Explicitly Parallelized Loops: `#pragma omp for`", page 90.
- "Loops With Barriers: `#pragma omp barrier`", page 91.
- "Critical Sections: `#pragma omp critical`", page 92.
- "Single-Process Sections: `#pragma omp single`", page 92.
- "Parallel Sections: `#pragma omp sections`", page 93.

For more information on OpenMP directives, see the compiler documentation or the OpenMP Architecture Review Board Web site: `http://www.openmp.org`.

Go to the first parallel region of `omp_demo` by scrolling down the loop list or using the Search field and entering **`parallel`**.

To select the first parallel region, double-click the highlighted line in the loop list, Olid 53.

Explicitly Parallelized Loops: `#pragma omp for`

The `omp_demo` function declares a parallel region containing three loops, the third of which is nested in the second. The first two loops are explicitly parallelized with `#pragma omp for` directives.

Example 4-9 C: explicitly parallelized loops

```
#pragma omp parallel shared(a,b)
{
#pragma omp for schedule(dynamic,10-2*2)
        for (i=0; i < ARRAYSIZE; i++)
           a[i] = i;
```

```
#pragma omp for schedule(static)
        for (i=0; i < ARRAYSIZE; i++) {
           b[i] = 3 * a[i];
           a[i] = b[i] * a[i];
           for (j = 0; j < ARRAYSIZE; j++)
                c[j][i] = a[i] + b[j];
        }
}
```

Notice in Figure 4-7 that the controls in the loop information display are now labelled `Region Controls`. The controls now affect the entire region. The **Keep** option button and the highlight buttons function the same way as in the Loop Parallelization Controls.

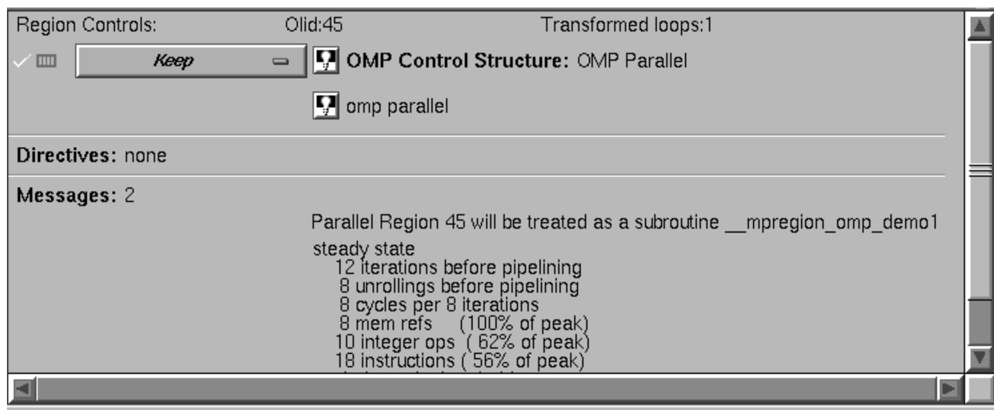


Figure 4-7 Loops Explicitly Parallelized Using `#pragma omp for`

Notice in the **Source View** that both loops contain a `#pragma omp parallel for` directive. Click **Next Loop** to step to the second parallel region.

Loops With Barriers: `#pragma omp barrier`

Olid 58 contains a pair of loops with a barrier between them. Because of the barrier, all iterations of the first `for` loop must complete before any iteration of the second loop can begin.

Example 4-10 C: loops with barriers

```
#pragma omp parallel shared(a,b)
{
#pragma omp for schedule(static, 10-2*2) nowait
        for (i=0; i < ARRAYSIZE; i++)
                a[i] = i;
#pragma omp barrier
#pragma omp for schedule(static)
        for (i=0; i< ARRAYSIZE; i++)
           b[i] = 3 * a[i];
} /*omp end parallel */
```

Click **Next Loop** twice to go to the third parallel region.

Critical Sections: `#pragma omp critical`

Click **Next Loop** to view the first of the two loops in the third parallel region. This loop contains a critical section.

Example 4-11 C: critical sections

```
#pragma omp for
       for (i = 0; i < ARRAYSIZE; i++) {
#pragma omp critical(s3)
{
           s1 = s1 + i;
}
        }
```

Click **Next Loop** twice to view the critical section. The critical section uses a named locking variable (`s3`) to prevent simultaneous updates of `s1` from multiple threads. This is a standard construct for performing a reduction.

Move to the next loop by using **Next Loop**.

Single-Process Sections: `#pragma omp single`

This loop has a single process section, which ensures that only one thread will execute the statement in the section. Highlighting in the **Source View** shows the begin and end directives.

Example 4-12 C: single process sections

```
       for (i=0; i <ARRAYSIZE; i++) {
#pragma omp single
           s2 = s2 + i;
        }

} /* omp end parallel */
```

Move to the final parallel region in `omp_demo` by clicking the **Next Loop** button.

Parallel Sections: `#pragma omp sections`

The fourth parallel region of `omp_demo` provides an example of parallel sections.

In this case, there are three parallel subsections, each of which calls a function. Each function is called once by a single thread. If there are three or more threads in the program, each function may be called from a different thread. The compiler treats this directive as a single-process directive, which guarantees correct semantics.

Example 4-13 C: parallel sections

```
#pragma omp sections
{
        dst1d(n,a);
#pragma omp section
        rshape2d(n,c);
#pragma omp section
        baz();
} /* omp sections */
```

Click **Next Loop** to view the entire `#pragma omp sections` region. Click **Next Loop** to view a `#pragma omp section` region. Move to the next subroutine by clicking **Next Loop** twice.

Examples Using Data Distribution Directives

The next series of functions illustrates directives that control data distribution and cache storage. The following topics are described in this section:

- Distributed Arrays: `#pragma distribute`, see "Distributed Arrays: `#pragma distribute`", page 94.
- Distributed and Reshaped Arrays: `#pragma distribute_reshape`, see "Distributed and Reshaped Arrays: `#pragma distribute_reshape`", page 96.
- Prefetching Data From Cache: `#pragma prefetch_ref`, see "Prefetching Data From Cache: `#pragma prefetch_ref`", page 97.

Distributed Arrays: `#pragma distribute`

When you select the function `dst1d()`, a parallelized loop icon is listed in the loop information display. The `#pragma distribute` directive specifies placement of array members in distributed, shared memory. (See Figure 4-8, page 95.)

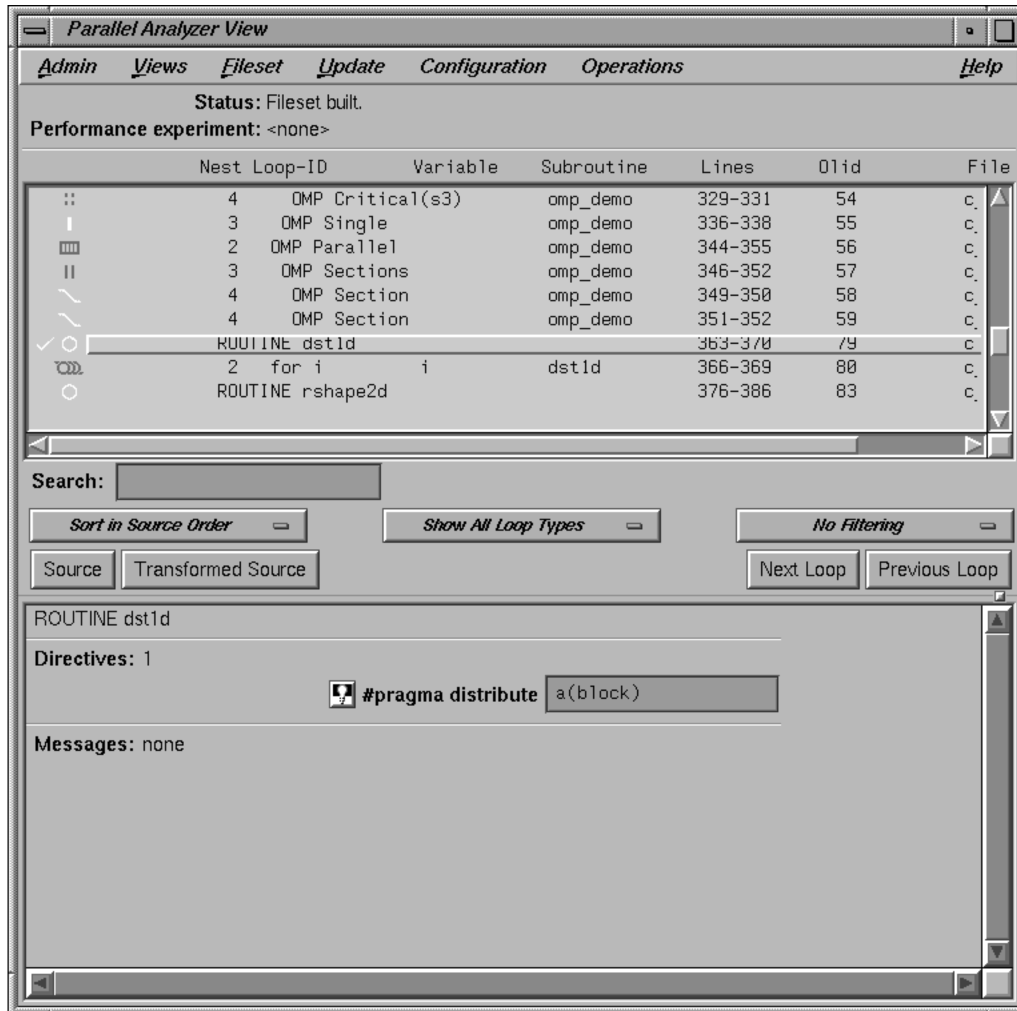


Figure 4-8 `#pragma distribute` Directive and Text Field

In the editable text field adjacent to the directive name is the argument for the directive, which in this case distributes the one-dimensional array `a` among the local memories of the available processors. To highlight the directive in the **Source View**, click the highlight button.

Click **Next Loop** to move to the parallel loop.

The loop has a `#pragma parallel for` directive, which works with `#pragma distribute` to ensure that each processor manipulates locally stored data.

Example 4-14 C: distributed arrays

```
void dst1d(int m,int a[m])
{
int i;
#pragma distribute a[block]
#pragma omp for
        for (i=1; i < m; i++)
           a[i] = i;

}
```

You can highlight the `#pragma parallel for` directive in the **Source View** with either of the highlight buttons in the loop information display. If you use the highlight button in the Loop Parallelization Controls, the **Parallelization Control View** window presents more information about the directive and lets you to change the `#pragma parallel for` clauses. In this example, it confirms what you see in the code: that the index variable `i` is local.

Click **Next Loop** until the next function (`rshape2d`) is selected.

Distributed and Reshaped Arrays: `#pragma distribute_reshape`

When you select the function `rshape2d`, the function's global directive is listed in the loop information display. The `#pragma distribute_reshape` directive specifies placement of array members in distributed, shared memory. It differs from the `#pragma distribute` directive in that it causes the compiler to reorganize the layout of the array in memory to guarantee the desired distribution. Furthermore, the unit of memory allocation is not necessarily a page.

In the text field adjacent to the directive name is the argument for the directive, which in this case distributes the columns of the two-dimensional array `c` among the local memories of the available processors. To highlight the directive in the **Source View**, click the highlight button.

Click the **Next Loop** button to move to the parallel loop.

The loop has a `#pragma parallel for` directive (see the following example), which works with `#pragma distribute_reshape` to enable each processor to manipulate locally stored data.

Example 4-15 C: distributed and reshaped arrays

```
static void
rshape2d(int m, int c[m][m])
{
int i,j;
#pragma distribute_reshape c[*][block]
#pragma omp for
        for (i=1; i < m; i++) {
           for (j = 1; j < m; j++) {
              c[i][j] = i*j;
           }
        }

}
```

If you use the highlight button in the Loop Parallelization Controls, the **Parallelization Control View** presents more information. In this example, it confirms what you see in the code: that the index variable `i` is local.

For more information on the `#pragma distribute_reshape` directive, see the *C Language Reference Manual*.

Click **Next Loop** to move to the nested loop. Notice that this loop has an icon in the loop list and in the loop information display indicating that it does not run in parallel.

Click **Next Loop** to view the `prfetch` function.

Prefetching Data From Cache: `#pragma prefetch_ref`

Click **Next Loop** to go to the first loop in `prfetch()`.

Example 4-16 C: prefetching data from cache

```
static void
prfetch( int n, int a[n][n], int b[n][n])
{

      int i, j;
```

```
      for (i =0; i < n ; i++) {
         for (j =0; j < n ; j++) {
            a[i][j] = b[i][j];
#pragma prefetch_ref=b[i][j],stride=2,2 level=1,2 kind=rd, size=4
#pragma prefetch_ref=b[i][j],stride=2,2 level=1,2 kind=rd, size=4
         }
      }
}
```

Click **Next Loop** to move to the nested loop. The list of directives in the loop information display shows `#pragma prefetch_ref` with a highlight button to locate the directive in the **Source View**. The directive allows you to place appropriate portions of the array in cache.

Exiting From the Sample Session

This completes the sample session. Quit the **Parallel Analyzer View** by choosing **Admin > Exit**.

Not all windows opened during the session close when you quit the **Parallel Analyzer View**. In particular, the **Source View** remains open because all the tools interoperate, and other tools may share the **Source View** window. You must close the **Source View** separately.

To clean up the directory so that the session can be rerun, enter the following in your shell window:

```
% make clean
```

Using WorkShop With Parallel Analyzer View

This is a brief demonstration of the integration of ProDev ProMP and the WorkShop performance tools. WorkShop must be installed for this session to work.

This sample session examines LINPACK, a standard benchmark designed to measure CPU performance in solving dense linear equations. See the *SpeedShop User's Guide* for a tutorial analysis of LINPACK.

This tutorial assumes you are already familiar with the basic features of the **Parallel Analyzer View** discussed in previous chapters. You can also consult Chapter 6, "Parallel Analyzer View Reference", page 107, for more information.

Start by entering the following commands:

```
% cd /usr/demos/ProMP/linpack
% make
```

This updates the directory by compiling the source program `linpackd.f` and creating the necessary files. The performance experiment data is in the file `test.linpack.cp`.

After the directory has been updated, start the demo by typing:

```
% cvpav -e linpackd
```

Note that the flag is `-e`, not `-f` as in the previous sample session. The main window of the Parallel Analyzer View opens, showing the list of loops in the program.

Click the **Source** button to open the loop list and the Source View. Scroll briefly through the list. Note that there are many unparallelized loops, but there is no way to know which are important. Also note that the second line in the main view shows that there is no performance experiment currently associated with the view.

Starting the Parallel Analyzer View

Pull down **Admin > Launch Tool > Performance Analyzer** to start the Performance Analyzer.

The main window of the Performance Analyzer opens; it is empty. A small window labeled Experiment: also opens at the same time. This window is used to enter the name of an experiment. For this session, use the installed prerecorded experiment.

Click on the **test.linpack.cpu** directory in the directory display area, then click on the **bbcounts** experiment name.

The Performance Analyzer shows a busy cursor and fills its main window with the list of functions in `main()`. The Parallel Analyzer recognizes that the Performance Analyzer is active, and posts a busy cursor with a Loading Performance Data message. When the message goes away, performance data will have been imported by the Parallel Analyzer.

For more information about the Performance Analyzer and how it affects the user interface, see the *ProDev WorkShop: Performance Analyzer User's Guide*.

Using the Parallel Analyzer With Performance Data

Once performance data has been loaded in the Parallel Analyzer View, several changes occur in the main window, as shown in Figure 5-1.

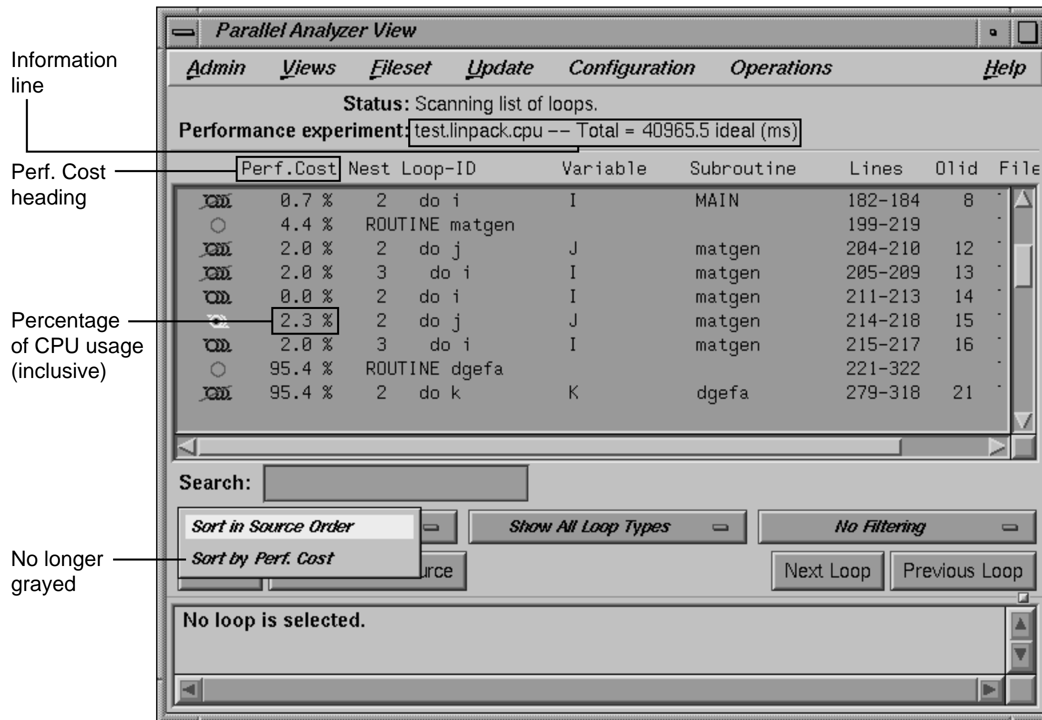


Figure 5-1 Parallel Analyzer View — Performance Data Loaded

- A new column, **Perf. Cost**, appears in the loop list next to the icon column. The values in this column are inclusive: each reflects the time spent in the loop and in any nested loops or functions called from within the loop.
- The Performance experiment line, in the main view below the menu bar, now shows the name of the performance experiment and the total cost of the run in milliseconds.
- The **Sort by Perf.Cost** option of the sort option button is now available.
- In the Source View, three columns appear to the left of the loop brackets. (These columns may take a few moments to load.) They reflect the measured performance data:
  - **Exq Count**: the number of times the line has been executed

– **Excl Ideal(ms)**: exclusive, ideal CPU time in milliseconds

– **Incl Ideal(ms)**: inclusive, ideal CPU time in milliseconds

Effect of Performance Data on the Source View

To see the effect of the performance data on the Source View, select Olid 30, which is in subroutine `daxpy()`. The Source View appears as shown in Figure 5-2.

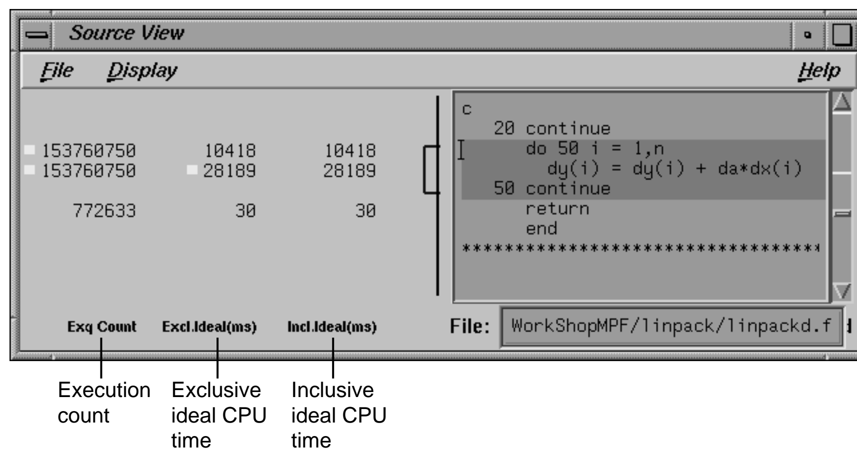


Figure 5-2 Source View for Performance Experiment

Sorting the Loop List by Performance Cost

Choose the **Sort by Perf.Cost** sort option. Note that the third most expensive loop listed, Olid 30 of subroutine `daxpy()`, represents approximately 94% of the total time. (See Figure 5-3, page 103.)

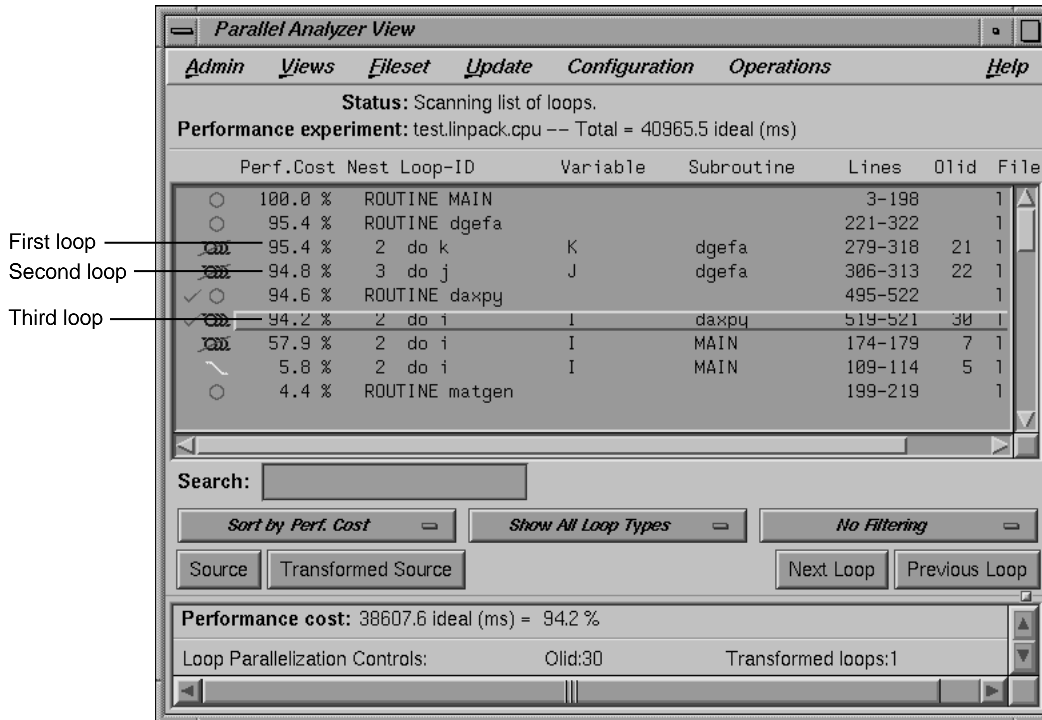


Figure 5-3 Sort by Performance Cost

The first of the high-cost loops, Olid 21 in subroutine `dgefa()`, contains the second most expensive loop (Olid 22) nested inside it. This second loop calls `daxpy()`, which contains Olid 30—the heart of the LINPACK benchmark. Olid 30 performs the central operation of scaling a vector and adding it to another vector. It was parallelized by the compiler. Note the `C$OMP PARALLEL DO` directive that appears for this loop in the Transformed Source View.

The loop following `daxpy()` uses approximately 58% of the CPU time. This loop is the most frequent caller of `dgefa()`, and so of Olid 30.

Double-click Olid 30. Note that the loop information display contains a line of text listing the performance cost of the loop, both in time and as a percentage of the total time. (See Figure 5-4, page 104.)

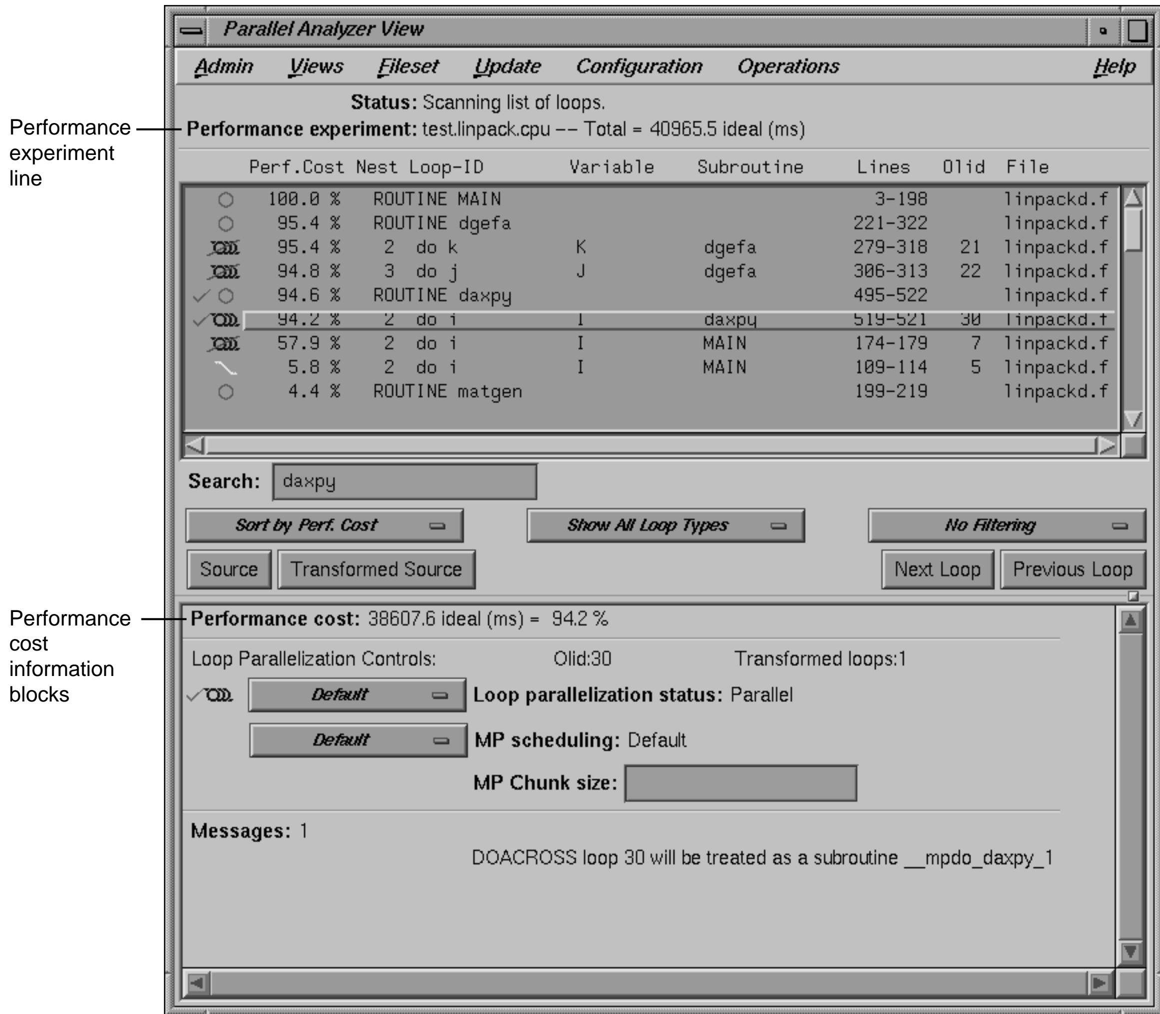


Figure 5-4 Loop Information Display With Performance Data

Exiting From the linpackd Sample Session

This completes the second sample session.

Close all windows—those that belong to the Parallel Analyzer View as well as those that belong to the Performance Analyzer and the Source View—by selecting the option **Admin > Project > Exit** in the Parallel Analyzer View.

You don't need to clean up the directory, because you haven't made any changes in this session.

If you experiment and do make changes, when you are finished you can clean up the directory and remove all generated files by entering the following in your shell window:

```
% make clean
```

Parallel Analyzer View Reference

This chapter describes in detail the function of each window, menu, and display in the ProDev ProMP Parallel Analyzer View's user interface. It contains the following main sections:

- "Parallel Analyzer View Main Window", page 107
- "Parallel Analyzer View Menu Bar", page 109
- "Loop List Display", page 123
- "Loop Display Controls", page 126
- "Loop Information Display", page 129
- "Views Menu Options", page 136
- "Loop Display Control Button Views", page 148

Parallel Analyzer View Main Window

The main window is displayed when the Parallel Analyzer View begins. It consists of the following elements, shown in Figure 6-1, page 109:

- Main menu bar, containing these menus:
  - "Admin Menu", page 110.
  - "Views Menu", page 116.
  - "Fileset Menu", page 116.
  - "Update Menu", page 117.
  - "Configuration Menu", page 118.
  - "Operations Menu", page 119.
  - "Help Menu", page 122.
- Loop list display, which has the following members:
  - "Status and Performance Experiment Lines", page 124.

  - "Status and Performance Experiment Lines", page 124.
  - "Loop List", page 125.

- Loop display controls, consisting of the following:

  - "Search Loop List Field", page 127.
  - "Sort Option Button", page 127, "Show Loop Types Option Button", page 128, and "Filtering Option Button", page 128.
  - "Loop Display Buttons", page 129.
- "Loop Information Display", page 129.

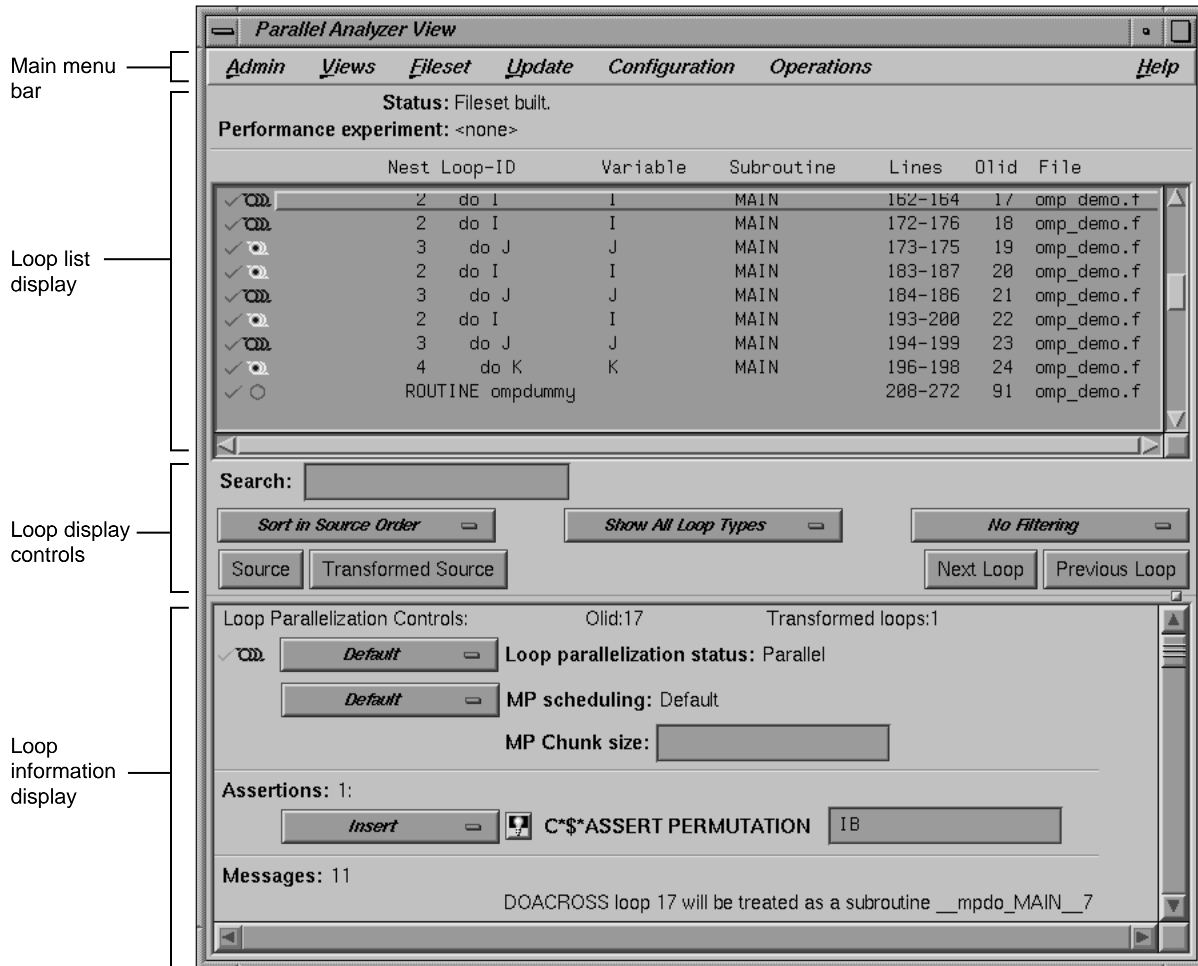


Figure 6-1 Parallel Analyzer View Main Window

Parallel Analyzer View Menu Bar

This section describes the menus found in the menu bar located at the top of the Parallel Analyzer View main window.

Within each menu, the names of some options are followed by keyboard shortcuts, which you can use instead of the mouse for faster access to these options. For a summary, see "Keyboard Shortcuts", page 123.

You can tear off a menu from the menu bar, so that it is displayed in its own window with each menu command visible at all times, by selecting the dashed line at the top of the menu (the first item in each of the menus). Submenus can also be torn off and displayed in their own window.

Admin Menu

The Parallel Analyzer View Admin menu contains file-writing commands, other administrative commands, and commands for launching and manipulating other WorkShop application views.

The commands in the Admin menu have the following effects:

- Save as Text: saves the complete loop information for all files and subroutines in the current session in a plain ASCII file. Choosing **Admin > Save as Text** brings up a File Selection dialog, which lets you choose where to save the file and what name to call it. (See Figure 6-2, page 111.)

  The default directory is the one from which you invoked the Parallel Analyzer View; the default filename is `Text.out`. The Parallel Analyzer View asks for confirmation before overwriting an existing file.

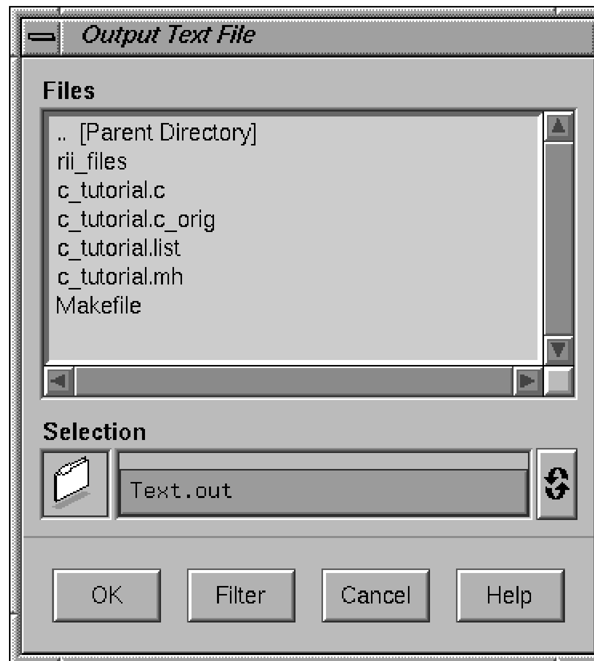


Figure 6-2 Output Text File Selection Dialog

- **Icon Legend**: provides an explanation of the graphical icons used in several of the views. Shortcut: `Ctrl+S`. See "Icon Legend… Option", page 112.
- **Iconify**: stows all the open windows belonging to a given invocation of the Parallel Analyzer View as icons in the style of the window manager you are using.
- **Raise**: brings all open windows in the current session to the foreground of the screen, in front of other windows. The command also opens any previously iconified windows belonging to the invocation of the Parallel Analyzer View and brings them to the foreground. Shortcut: `Ctrl+R`.
- **Launch Tool**: opens various WorkShop tools. See "Launch Tool Submenu", page 112.
- **Project**: controls project windows. See "Project Submenu", page 113.
- **Exit**: quits the current session of the Parallel Analyzer View, closing all windows.

If you have not updated source files and have pending requests for changes, a dialog box asks if it is OK to discard the changes. Click **OK** only if you want to **discard** any changes; otherwise, click **Cancel** to update the files.

Icon Legend... Option

This Admin menu option opens the Parallelization Icon Legend which provides the meanings of the icons that appear in various views, such as the following:

- Parallel Analyzer View, shown in Figure 2-1, page 9
- Transformed Loops View, shown in Figure 6-16, page 144
- Subroutines and Files View, shown in Figure 6-18, page 147
- Parallelization Control View, shown in Figure 6-13, page 137

Launch Tool Submenu

The Admin menu's Launch Tool submenu contains commands for launching other WorkShop tools, as well as new sessions of the Parallel Analyzer.

To work properly with the other WorkShop tools, the files in the current fileset must have been loaded into the Parallel Analyzer from an executable. There are two ways to do this:

- Use the **-e** option on the command line.
- Choose the **Fileset > Add File** menu option.

If you launch Workshop tools from a session not based on an executable, the tools start without arguments.

The following options launch applications from the Launch Tool submenu:

- **Build Analyzer**: launches the Build Manager, a utility that lets you compile software without leaving the WorkShop environment. For more information, see the *ProDev WorkShop: Debugger User's Guide*.
- **Debugger**: launches the WorkShop Debugger, a source-level debugging tool that provides special windows for displaying program data and execution status. For more information, see the *ProDev WorkShop: Debugger User's Guide*
- **Parallel Analyzer**: launches another session of the Parallel Analyzer.

- **Performance Analyzer**: launches the Performance Analyzer, a utility that collects performance data and allows you to analyze the results of a test run. For more information, see the *ProDev WorkShop: Performance Analyzer User's Guide*.
- **Static Analyzer** : launches the Static Analyzer, a utility that allows you to analyze and display source code written in C, C++, or Fortran. For more information, see the *ProDev WorkShop: Static Analyzer User's Guide*.
- **Tester**: launches the Tester, a UNIX-based software quality assurance tool set for dynamic test coverage over any set of tests. For more information, see the *ProDev WorkShop: Tester User's Guide*.

If any of these tools is not installed on your system, the corresponding menu item is grayed out.

If the file `/usr/lib/WorkShop/system.launch` is absent (that is, if you are running the Parallel Analyzer View without WorkShop 2.0 installed), the entire Launch Tool submenu is grayed out.

Project Submenu

The Project submenu of the Admin menu contains commands that affect all the windows containing WorkShop or ProDev ProMP applications that have been launched to manipulate a single executable. The set of windows is a WorkShop **project**. The Project submenu and windows that you can open from it are shown in Figure 6-3, page 115.

The Project submenu commands are as follows:

- **Iconify**: stows all the windows in the current project as icons, in the style of the window manager you are using.
- **Raise**: brings all open windows in the current project to the foreground of the screen, in front of other windows. The command also opens any previously iconified windows in the current project and brings them to the foreground.
- **Remap Paths...**: lets you modify the set of mappings used to redirect references to filenames located in your code to their actual locations in your file system. However, if you compile your code on one tree and mount it on another, you may need to remap the root prefix to access the named files.
- **Project View...**: launches the WorkShop Project View, a tool that helps you manage project windows.

- **Exit**: quits the current project, closing all windows, including those of related open applications. Thus the Source View closes, as well as, for example, the Parallel Analyzer.

  If you have not updated source files and have pending requests for changes, a dialog box asks if it is OK to discard the changes. Click **OK** only if you want to discard any changes; otherwise, click **Cancel** and update the files.

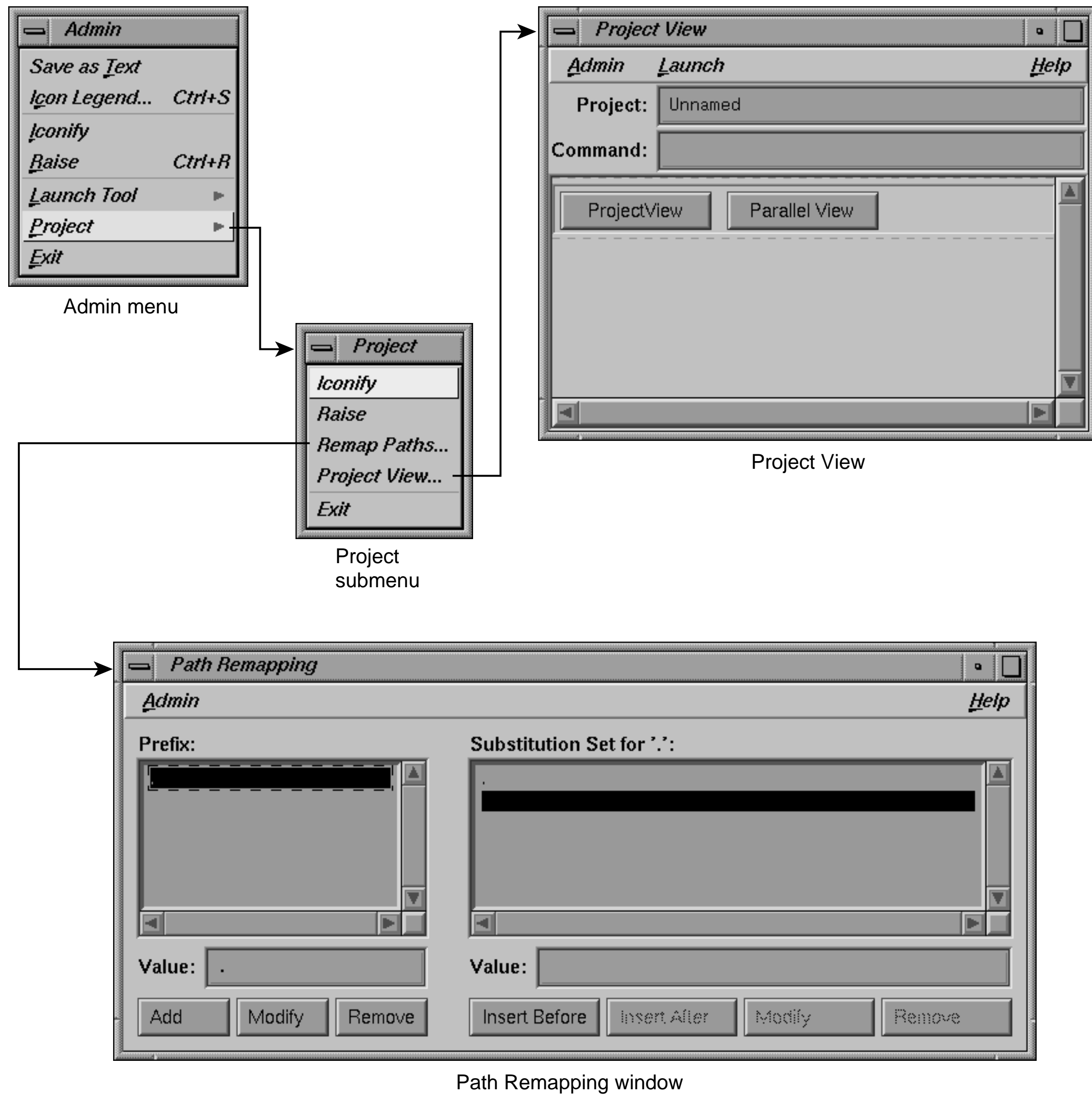


Figure 6-3 Project Submenu and Windows

Views Menu

The Views menu of the Parallel Analyzer View contains commands for launching a variety of secondary windows, or **views**, that provide specific sets of information about, and tools to apply to, selected loops.

The options in the Views menu have the following effects:

- **Parallelization Control View**: opens a Parallelization Control View for the loop currently selected from the loop list display. Shortcut: **`Ctrl+P`**. For more information on this view, see "Parallelization Control View", page 136.
- **Transformed Loops View**: opens a Transformed Loops View for the loop currently selected from the loop list display. Shortcut: **`Ctrl+T`**. For more information on this view, see "Transformed Loops View", page 144.
- **PFA Analysis Parameters View**: opens the PFA Analysis Parameters View, which provides a means of modifying a variety of PFA parameters. Shortcut: **`Ctrl+A`**. This view is further described in "PFA Analysis Parameters View", page 145.
- **Subroutines and Files View**: opens the Subroutines and Files View, which provides a complete list of subroutine and file names being examined within the current session of the Parallel Analyzer View. Shortcut: **`Ctrl+F`**. This view is further described in "Subroutines and Files View", page 146.

Fileset Menu

The Fileset menu contains commands for manipulating the files displayed by the Parallel Analyzer View. A *fileset* is a list of source filenames contained in an ASCII file, each on a separate line.

The options in the Fileset menu have the following effects:

- **Rescan All Files**: the Parallel Analyzer View checks and updates all the source files loaded into its current session so they match the versions of those files in the file system. The Parallel Analyzer View rereads only the files it needs to.
- **Delete All Files**: removes all files from the current session of the Parallel Analyzer View. You can then add new files using the Add File, Add Files from Fileset, or Add Files from Executable options, described below.
- **Delete Selected File**: deletes a selected file from the current session of the Parallel Analyzer View. To select a file for deletion, open the Subroutines and Files View and double-click the desired filename.

- **Add File**: adds a new source file to the current session of the Parallel Analyzer View. Selecting this command brings up a File Selection dialog that lets you select a Fortran source file.

  Before you can select a given source file, you must compile it to create the `.anl` file needed by the Parallel Analyzer View. (See "Compiling a Program for ProMP Use", page 2.)

  If the current session is based on an executable, you cannot add files to it until you have deleted the executable's fileset. (See the Add Files from Executable option, described below.)

- **Add Files from Fileset**: lets you add a list of new source files to the current session of the Parallel Analyzer View. Choosing this command brings up a File Selection dialog as it does for the Add File option. If you select a file containing a fileset list, all Fortran source files in the list are loaded into the current session (other files in the list are ignored).

  If the current session is based on an executable, you cannot add files to it until you have deleted the executable's fileset.

- **Add Files from Executable**: imports all the Fortran source files listed in the symbol table of a compiled Fortran application. This command works only if there are no files in the current session of the Parallel Analyzer View when the command is selected from the menu. Selecting this command brings up a File Selection dialog as it does for the Add File option. Other WorkShop applications can also operate on files imported from an executable.

Update Menu

The Parallel Analyzer View Update menu contains commands for placing requested changes to directives and assertions in your Fortran source code.

The options in the Update menu have the following effects:

- **Run gdiff After Update**: sets a checkbox that causes a `gdiff` window to open after you have updated changes to your source file. This window illustrates in a graphical manner the differences between the unchanged source and the newly updated source.

If you always wish to see the gdiff window, you may set the resource in your .Xdefaults file:

```
cvpav*gDiff: True
```

For more information on using gdiff, see the man page for gdiff(1).

- **Run Editor After Update**: sets a checkbox that opens an xwsh shell window with the vi editor on the updated source file.

  If you always wish to run the editor, you can set the resource in your .Xdefaults file:

  ```
  cvpav*runUserEdit: True
  ```

  If you prefer a different window shell or a different editor, you can modify the resource in your .Xdefaults file and change from xwsh or vi as you prefer. The following is the default command in the .Xdefaults, which you can edit for your preference:

  ```
  cvpav*userEdit: xwsh -e vi %s +%d
  ```

  In the above command, the **+%d** tells vi at what line to position itself in the file and is replaced with **1** by default (you can also omit the **+%d** parameter if you wish). The edited file's name either replaces any explicit **%s**, or if the **%s** is omitted, the filename is appended to the command.

- **Update All Files**: writes to the appropriate source files all changes to loops requested during the current session of the Parallel Analyzer View. Shortcut: **Ctrl+U**.

- **Update Selected File**: writes to a selected file changes to loops requested during the current session of the Parallel Analyzer View. You choose a file for updating by double-clicking in the Subroutines and Files View the line corresponding to the desired filename. (See also "Subroutines and Files View", page 146.)

- **Force a Build to Start**: performs the Update All Files option and starts a build.

Configuration Menu

The Configuration menu allows you to choose between having the Parallel Analyzer View use OpenMP or PCF directives.

The options are the following:

- **OpenMP**: causes the Parallel Analyzer View to use OpenMP directives.
- **PCF**: causes the Parallel Analyzer View to use PCF directives.

Operations Menu

The Parallel Analyzer View Operations menu contains commands for adding assertions and directives to loops, and removing pending changes to source files (Figure 6-4). The general effects of the Operations menu options are to prepare a set of requested changes to your source code. For information on how these changes are subsequently performed see "Update Menu", page 117.

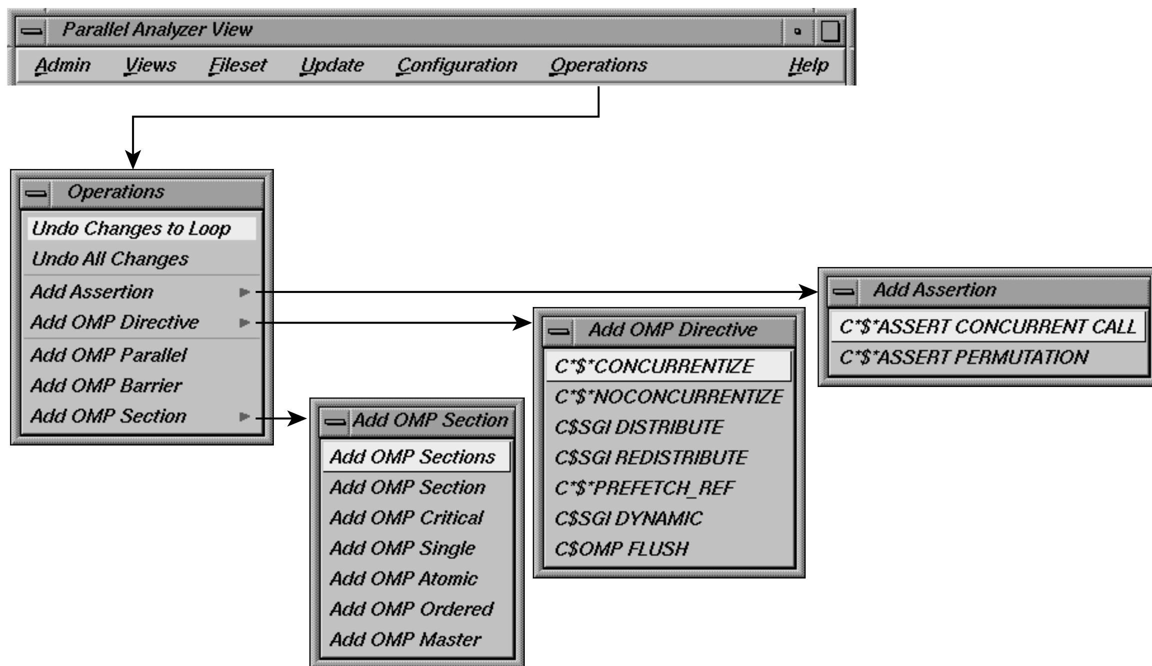


Figure 6-4 Operations Menu and Submenus

The Operations menu is one of two points in the Parallel Analyzer View where you can add assertions and directives. The other point is discussed in "Loop

Parallelization Controls in the Loop Information Display", page 131. These two menus focus on different aspects of the parallelization task:

- The Operations menu focuses on automatic parallelization directives, which may be inserted in code by the MIPSpro Auto-Parallelizing Option, and memory distribution.
- The parallelization controls in the loop information display focus on manual (that is, not automatic) parallelization controls, which you can insert to further parallelize your code.

The assertions and directives you can add from the Operations menu are described in the following lists.

- `C*$* ASSERT CONCURRENT CALL`: ignore dependences in subroutine calls that would inhibit parallelizing.
- `C*$* ASSERT PERMUTATION (`*array_name*`)`: Array *array_name* is a permutation array.
- `C*$* CONCURRENTIZE`: selectively override `C*$* NOCONCURRENTIZE`: typically inserted during automatic parallelization.
- `C*$* NOCONCURRENTIZE`: do not parallelize file subroutine (depending on placement). Typically inserted during automatic parallelization.
- `C$SGI DISTRIBUTE`, `C$SGI REDISTRIBUTE`: distribute array storage among processors. For Origin2000 systems.
- `C*$* PREFETCH_REF`: load data into cache. May be used with nonconcurrent code.
- `C$SGI DYNAMIC`: allow run-time array redistribution. For Origin2000 systems.
- `C$OMP FLUSH`: identifies synchronization points at which the implementation is required to provide a consistent view of memory.

The options in the Operations menu have the following specific effects:

- **Undo Changes to Loop**: removes pending changes to the currently selected loop. Changes that have already been written to the source file using the Update menu commands cannot be undone.
- **Undo All Changes**: removes pending changes to all the loops in the current fileset. Changes that have already been written to the source file using the Update menu commands cannot be undone.

- **Add Assertion**: opens the Add Assertion menu which allows you to add the following assertions:
  - C*$*ASSERT CONCURRENT CALL
  - C*$*ASSERT PERMUTATION
- **Add OMP Directive**: opens the Add OMP Directive menu which allows you to add these directives:
  - C*$* CONCURRENTIZE
  - C*$* NOCONCURRENTIZE
  - C$SGI DISTRIBUTE (formerly C*$* DISTRIBUTE)
  - C$SGI REDISTRIBUTE (formerly C*$* REDISTRIBUTE)
  - C*$* PREFETCH_REF
  - C$SGI DYNAMIC (formerly C*$* DYNAMIC)
  - C$OMP FLUSH
- **Add OMP Parallel**: allows you to add the C$OMP PARALLEL directive. The directive defines a parallel region, that is a block of code that is to be executed by multiple threads in parallel.
- **Add OMP Barrier**: allows you to add the C$OMP BARRIER synchronization directive. This directive causes each thread to wait at the designated point until all have reached it.
- **Add OMP Section**: opens the Add OMP Section submenu whose seven options allow you to add the OpenMP synchronization directives shown below.
  - **Add OMP Sections**: C$OMP SECTIONS specifies that the enclosed sections of code are to be divided among threads in a team.
  - **Add OMP Section**: C$OMP SECTION: delineates a section within C$OMP SECTIONS.
  - **Add OMP Critical**: C$OMP CRITICAL: restrict access to enclosed code to one thread at a time.
  - **Add OMP Single**: C$OMP SINGLE: only one thread executes the enclosed code

- **Add OMP Atomic**: `C$OMP ATOMIC`: update memory location atomically, not simultaneously.
- **Add OMP Ordered**: `C$OMP ORDERED`: execute enclosed code in same order as sequential execution.
- **Add OMP Master**: `C$OMP MASTER`: specify code to be executed by master thread.

To use the **Add OMP Section** option, do the following:

1. Bring up the **Source View**.
2. Using the mouse, sweep out a range of lines for the new construct.
3. Invoke the appropriate menu item to add the new construct.

When you add a new OMP Section construct, the list is redrawn with the new construct in place, and the new construct is selected. Brackets defining the new constructs are **not** added to the file loop annotations.

The Parallel Analyzer does not enforce any of the semantic restrictions on how parallel regions and or sections must be constructed. When you add nested regions or constructs, be careful that they are properly nested: they must each begin and end on distinct lines. For example, if you add a parallel region and a nested critical section that end at the same line, the terminating directives are not in the correct order.

Help Menu

The **Help** menu contains commands that allow you to access online information and documentation for the Parallel Analyzer View.

The options in the **Help** menu have the following effects:

- **On Version...** : opens a window containing version number information for the Parallel Analyzer View.
- **On Window...** : invokes the Help Viewer, which displays a descriptive overview of the current window or view and its graphical user interface.
- **On Context** : invokes context-sensitive help. When you choose this option, the normal mouse cursor (an arrow) is replaced with a question mark. When you click on graphical features of the application with the left mouse, or position the

cursor over the feature and press the **F1** key, the Help Viewer displays information on that context.

- **Index...** : invokes the Help Viewer and displays the list of available help topics, which you can browse alphabetically, hierarchically, or graphically.

Keyboard Shortcuts

Table 6-1, page 123, lists the keyboard shortcuts available in the Parallel Analyzer View:

Table 6-1 Parallel Analyzer View Keyboard Shortcuts

Shortcut	Menu	Menu Option
Ctrl+S	Admin	Icon Legend...
Ctrl+R	Admin	Raise
Ctrl+P	Views	Parallelization Control View
Ctrl+T	Views	Transformed Loops View
Ctrl+A	Views	PFA Analysis Parameters View
Ctrl+F	Views	Subroutines and Files View
Ctrl+U	Update	Update All Files

Loop List Display

This section describes the loop list display and the various option buttons and fields that manipulate the information shown in the loop list display, shown in Figure 6-5, page 124.

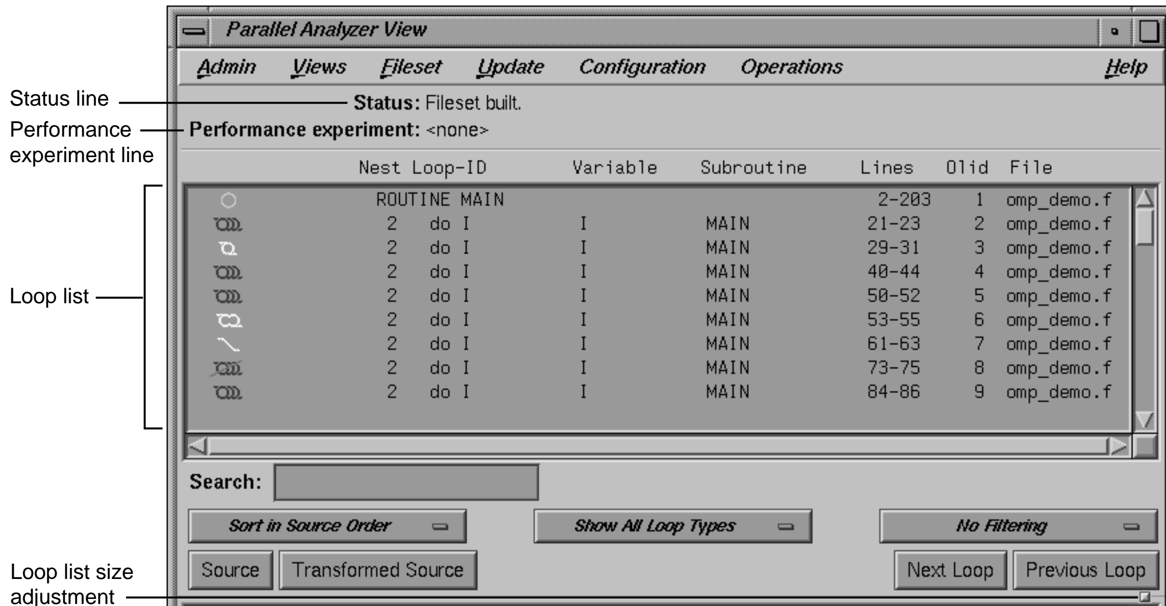


Figure 6-5 Loop List Display

You can resize the loop list to change the number of loops displayed; use the adjustment button: a small square below the **Previous Loop** button.

Status and Performance Experiment Lines

The Status line displays messages about the current status of the loop list, providing feedback on manipulations of the current fileset.

The Performance experiment line is meaningful if you run the WorkShop Performance Analyzer. The line displays the name of the current experiment directory and the type of experiment data, as well as total data for the current caliper setting in the Performance Analyzer. (See "Launch Tool Submenu", page 112, for information on invoking the Performance Analyzer from the Parallel Analyzer View.) If the Performance Analyzer is not being used, the performance experiment line displays <none>.

Loop List

The loop list lets you select and manipulate any Fortran DO loop contained in the source files loaded into the Parallel Analyzer View. Information about the loops is displayed in columns in the list; the headings of the columns are shown at the top of Figure 6-6 and described below.

Column headings | Nesting level | Loop index variable | Subroutine name | Location in code | Loop identifier | Filename

Nest	Loop-ID	Variable	Subroutine	Lines	Olid	File
ROUTINE MAIN				2-203	1	omp_demo.f
2	do I	I	MAIN	21-23	2	omp_demo.f
2	do I	I	MAIN	29-31	3	omp_demo.f
2	do I	I	MAIN	40-44	4	omp_demo.f
2	do I	I	MAIN	50-52	5	omp_demo.f
2	do I	I	MAIN	53-55	6	omp_demo.f
2	do I	I	MAIN	61-63	7	omp_demo.f
2	do I	I	MAIN	73-75	8	omp_demo.f
2	do I	I	MAIN	84-86	9	omp_demo.f

Figure 6-6 Loop List with Column Headings

The columns in the loop list contain the following information about each loop, from left to right:

- **Parallelization icon**: Indicates the parallelization status of each loop. The meaning of each icon is described in the Icon Legend dialog box. (See "Icon Legend… Option", page 112.) When a loop is displayed in the loop information display (by double-clicking the loop’s row), a green check mark is placed to the left of the icon to indicate that it has been examined. If any changes are made from within the loop information display, a red plus sign is placed above the check mark.

- **Perf. Cost** (not shown in Figure 6-6, page 125): The performance cost is displayed when the WorkShop Performance Analyzer is launched on the current fileset. (See "Launch Tool Submenu", page 112.) The loops can be sorted by Perf. Cost via the sort option button. (See "Sort Option Button", page 127.)

When performance cost is shown, each loop's execution time is displayed as a percentage of the total execution time. This percentage includes all nested loops, subroutines, and function calls.

- **Nest**: The nesting level of the given loop.
- **Loop-ID**: An ID for each loop in the list display. The ID is displayed indented to the right to reflect the loop's nesting level when the list is sorted in source order, and unindented otherwise.
- **Variable**: The name of the loop index variable.
- **Subroutine**: The name of the Fortran subroutine in which the loop occurs.
- **Lines**: The lines in the source file that make up the body of the loop.
- **Olid**: Original loop id is a unique internal identifier for the loops generated by the compiler. Use this value when reporting bugs.
- **File**: The name of the Fortran source file that contains the loop.

To *highlight* a loop in the list, click the left mouse anywhere in a loop's row; typing unique text from the row into the Search field does the same thing. (See "Search Loop List Field", page 127.)

To *select* a loop, double-click on its row; this will bring up detailed information in the loop information display below the loop list display. (See "Loop Information Display", page 129.) Selecting a loop affects other displays.

Loop Display Controls

The loop display controls are shown in Figure 6-7, page 127, and are discussed in the next sections.

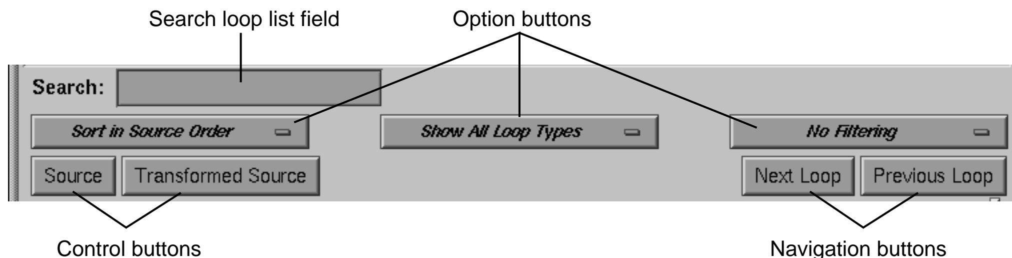


Figure 6-7 Loop Display Controls

Search Loop List Field

You can use the search loop list editable text field, shown at the top left of Figure 6-7, page 127, to find a specific loop in the loop list display. The Parallel Analyzer View matches any text typed into the field to the first instance of that text in the loop list, and highlights the row of the list in which the text occurs. The search field matches its text against the contents of each column in the loop list.

As you type into the field, the list highlights the first entry that matches what you have already typed, scrolling the list if necessary. If you press **`Enter`**, the highlight moves to the next match. If no match is found, the system beeps, and pressing **`Enter`** positions the highlight at the top of the list again.

Sort Option Button

The sort option button is the left-most option button under the loop list search field shown in Figure 6-7. It controls the order in which the loops are displayed in the loop list display.

The choices in the sort option button have the following effects:

- **Sort in Source Order**: orders the loops as they appear in the source file.
- **Sort by Perf.Cost**: orders the loops by their performance cost (from greatest to least) as calculated by the Workshop Performance Analyzer. This is the default setting. You must invoke the Performance Analyzer from the current session of the Parallel Analyzer View to make use of this option. See "Launch Tool Submenu", page 112, for information on how to open the Performance Analyzer from the current session of the Parallel Analyzer View.

Show Loop Types Option Button

The show loop types option button is the center option button under the loop list search field shown in Figure 6-7, page 127. It controls what kind of loops are displayed for each file and subroutine in the loop list.

The options in the show loop types button have the following effects:

- **Show All Loop Types**: default setting.
- **Show Unparallelizable Loops**: show only loops that could not be parallelized, and thereby run serially.
- **Show Parallelized Loops**: show only loops that are parallelized.
- **Show Serial Loops**: show only loops that are preferably serial.
- **Show Modified Loops**: show only loops with pending changes.
- **Show OMP Directives**: show only loops containing OMP directives.

Filtering Option Button

The filtering option button is the right-most option button under the loop list search field shown in Figure 6-7, page 127. It lets you display only those loops contained within a given subroutine or source file.

The button choices have the following effects:

- **No Filtering**: the default setting; lists all loops and routines.
- **Filter by Subroutine**: lets you enter a subroutine name into a filtering editable text field that appears above the option button. Only loops contained in that subroutine are displayed in the loop list.
- **Filter by File**: lets you enter a Fortran source filename into a filtering editable text field that appears above the option button. Only loops contained in that file are displayed in the loop list.

To place the name of a subroutine or file in the appropriate filter text field, you can double-click on a line in the Subroutines and Files View. If the appropriate type of filtering is currently selected, the loop list is rescanned.

Loop Display Buttons

The loop display controls (Figure 6-7, page 127) include two control buttons:

- **Source**: Opens the Source View window, with the source file containing the loop currently selected (double-clicked) in the loop list. The body of the loop is highlighted within the window. If no loop is selected, the last selected file is loaded; if no file is selected, the first file in the fileset is loaded.

  For more information on the Source View window, see "Source View and Parallel Analyzer View - Transformed Source", page 148.

- **Transformed Source**: Opens a Parallel Analyzer View - Transformed Source window, with the compiled source file containing the loop currently selected (double-clicked) in the loop list. The body of the loop is highlighted within the window. If no loop is selected, the last selected file is loaded; if no file is selected, the first file in the fileset is loaded.

  For more information on the Transformed Source window, see "Source View and Parallel Analyzer View - Transformed Source", page 148.

The loop display controls also include two navigation buttons:

- **Next Loop**: Selects the next loop in the loop list. The information in the loop information display and all other windows is updated accordingly. If no loop is currently selected, clicking on the button selects the first loop.

- **Previous Loop**: Selects the previous loop in the loop list. The information in the loop information display and all other windows is updated accordingly. If no loop is currently selected, clicking on the button selects the first loop.

Loop Information Display

The loop information display provides detailed information on various loop parameters, and allows you to alter those parameters to incorporate the changes into the Fortran source. The display is divided into several information blocks displayed in a scrolling list as shown in Figure 6-8, page 130.

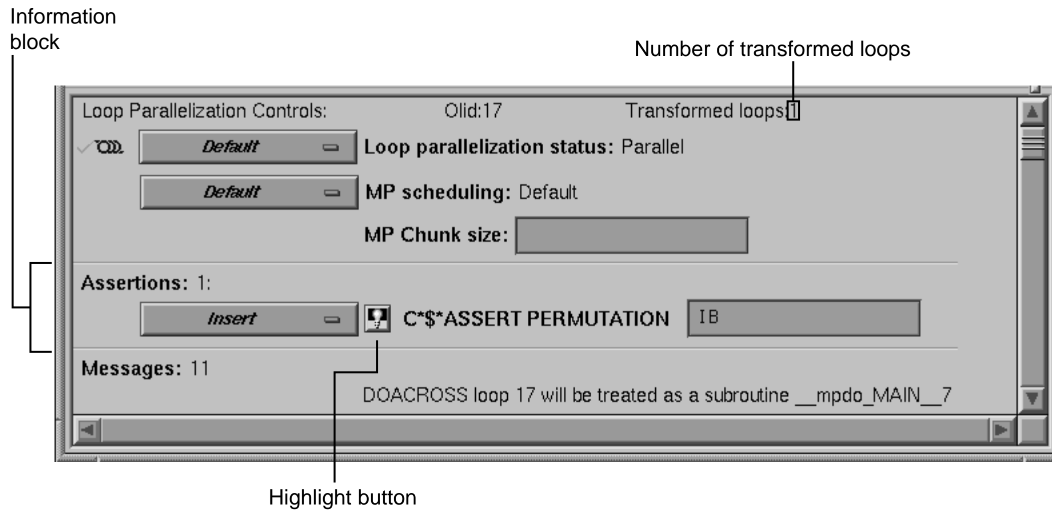


Figure 6-8 Loop Information Display

Each of these sections and the information it contains is described in detail below. The display is empty when no loop has been selected.

Highlight Buttons

A highlight button (light bulb, see Figure 6-8, page 130) appears as a shortcut to more information related to text in the display. Clicking the button does one or both of the following:

- Highlights the loop and the relevant line(s) in a Source View window. (See "Source View and Parallel Analyzer View - Transformed Source", page 148.)
- If a directive appears in the options menu next to it, the highlight button presents details about directive clauses in a Parallelization Control View. (See "Parallelization Control View", page 136.)

If directives or assertions with highlight buttons are also listed below the Loop Parallelization Controls, these buttons highlight the same piece of code as the corresponding button in the Loop Parallelization Controls, but they do not activate the Loop Parallelization Control View.

Loop Parallelization Controls in the Loop Information Display

The first line of the Loop Parallelization Controls section shows the Olid of the selected loop and, on the far right, how many transformed loops were derived from the selected loop.

Controls for altering the parallelization of the selected loop are shown in Figure 6-9. The controls in this section allow you to place parallelization assertions and directives in your code. Recall that you have similar controls available through the Operations menu. (See "Operations Menu", page 119.)

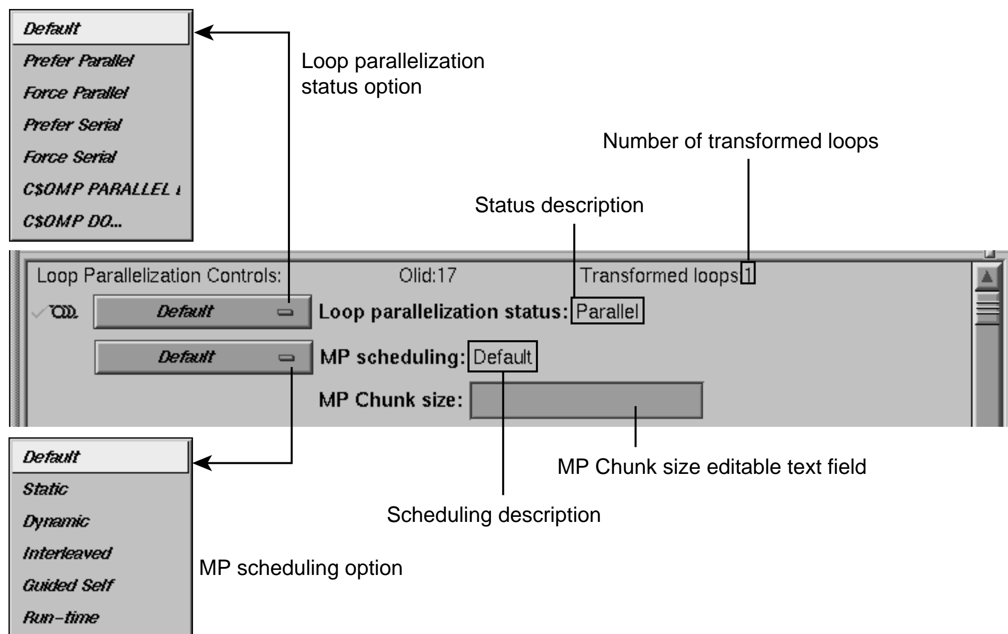


Figure 6-9 Loop Parallelization Controls

Loop Parallelization Status Option Button

The loop parallelization status option button (shown in Figure 6-9, page 131) lets you alter a loop's parallelization scheme. To the right of the option button is the Loop parallelization status field, a description of the current loop status as implemented in the transformed source. A small highlight button appears to the left of this description if the status was set by a directive.

The loop parallelization status option button choices are described in the following list:

- **Default**: always selects the parallelization scheme that the compiler picked for the selected loop.
- **Prefer Parallel**: adds the assertion `C*$*ASSERT DO PREFER` (CONCURRENT).
- **Force Parallel**: adds the assertion `C*$*ASSERT DO` (CONCURRENT).
- **Prefer Serial**: adds the assertion `C*$ASSERT DO PREFER` (SERIAL).
- **Force Serial**: adds the assertion `C*$*ASSERT DO` (SERIAL).
- **C$OMP PARALLEL DO...** : adds the OpenMP directive `C$OMP PARALLEL DO`. Selecting this item opens the Parallelization Control View. See "Parallelization Control View", page 136, for more information.
- **C$OMP DO...**: launches the Parallelization Control View, which allows you to manipulate the scheduling clauses for the OpenMP `C$OMP DO` directive and to set each of the referenced variables as either region-default or last-local.

  A `C$OMP DO` must be within a parallel region, although the tool does not enforce this restriction. If one is added outside of a region, the compiler reports an error.

A menu choice is grayed out if you are looking at a read-only file, if you invoked `cvpav` with the `-ro True` option, or if the loop comes from an included file. So in some cases you are not allowed to change the menu setting.

The following list describes the assertions and directives that you control from the loop parallelization status option button.

- `C*$* ASSERT DO (CONCURRENT)`: parallelize the loop; ignore possible data dependences.
- `C*$* ASSERT DO PREFER (CONCURRENT)`: attempt to parallelize the selected loop. If not possible, try each nested loop.
- `C*$* ASSERT DO (SERIAL)`: do not parallelize the loop.
- `C*$* ASSERT DO PREFER (SERIAL)`: do not parallelize the loop.
- `C$OMP PARALLEL DO`: parallelize the loop, ignore automatic parallelizer.
- `C$OMP DO`: assign each loop iteration to a different thread, ignore automatic parallelizer.

MP Scheduling Option Button: Directives for All Loops

The MP scheduling option button (Figure 6-9, page 131) lets you alter a loop's scheduling scheme by changing `C$MP_SCHEDTYPE` modes and values for `C$CHUNK`. For those modes that require a chunk size, there is a editable text field to enter the value. (See "MP Chunk Size Field", page 134.)

These directives affect the current loop and all subsequent loops in a source file. For control over a single loop, use the `C$OMP PARALLEL DO` directive clause. (See "MP Scheduling Option Button: Clauses for One Loop", page 142.)

The button choices are as follows:

- **Default**: always selects the scheduling scheme that the compiler picked for the selected loop.
- **Static** : divides iterations of the selected loop among the processors by dividing them into contiguous pieces and assigning one to each processor.
- **Dynamic**: divides iterations of the selected loop among the processors by dividing them into pieces of size `C$CHUNK`. As each processor finishes a piece, it enters a critical section to grab the next piece. This scheme provides good load balancing at the price of higher overhead.
- **Interleaved**: divides the iterations into pieces of size `C$CHUNK` and interleaves the execution of those pieces among the processors. For example, if there are four processors and `C$CHUNK` = 2, then the first processor executes iterations 1-2, 9-10, 17-18,…; the second processor executes iterations 3-4, 11-12, 19-20,…; and so on.
- **Guided Self**: divides the iterations into pieces. The size of each piece is determined by the total number of iterations remaining. The idea is to achieve good load balancing while reducing the number of entries into the critical section by parceling out relatively large pieces at the start and relatively small pieces toward the end.
- **Run-time**: lets you specify the scheduling type at run time.

To the right of the MP scheduling option button is the MP scheduling field, a description of the current loop scheduling scheme as implemented in the transformed source. A highlight button appears to the left of this description if the scheduling scheme was set by a directive.

MP Chunk Size Field

Below the MP scheduling description is the MP Chunk size editable text field, a field that allows you to set the `C$CHUNK` size for the scheduling scheme you select.

When you change an entry in the field, the upper right corner of the field turns down, indicating the change (Figure 6-10). To toggle back to the original value, left-click the turned-down corner (changed-entry indicator). The corner unfolds, leaving a fold mark. If you click again on the fold mark, you can toggle back to the changed value. You can enter a new value at any time; the field remembers the original value, which is always displayed after you click on the changed-entry indicator.

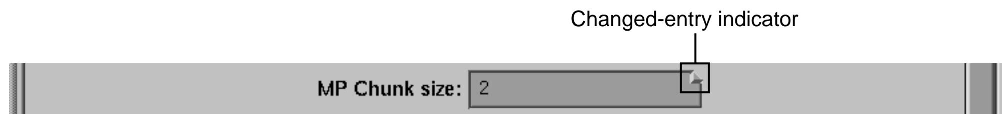


Figure 6-10 MP Chunk Size Field Changed

Be aware of the following when you use the MP Chunk size field:

- Your entry should be syntactically correct, although it is not checked.
- Like any other editable text field, the background color changes when you cannot make edits. This can happen if you are looking at a read-only file, if you invoked `cvpav` with the **-ro True** option, if the loop comes from an included file, or in some other cases.

Obstacles to Parallelization Information Block

Obstacles to parallelization are listed when the compiler discovers aspects of a loop's structure that make it impossible to parallelize. They appear in the loop information display below the parallelization controls.

Figure 6-11, page 135, illustrates a message describing an obstacle. The message has a highlight button directly to its left to indicate the troublesome line(s) in the Source View window, and opens the window if necessary. If appropriate, the referenced variable or function call is highlighted in a contrasting color.

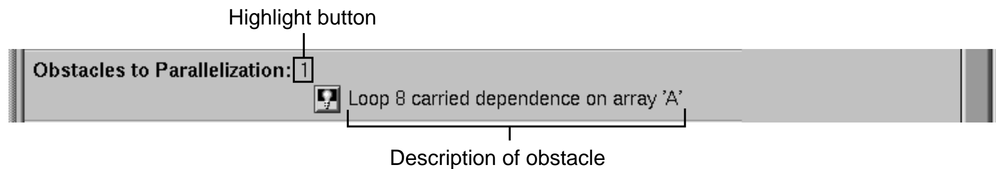


Figure 6-11 Obstacles to Parallelization Block

Assertions and Directives Information Blocks

The loop information display lists any assertions and directives for the selected loop along with highlight buttons. When you left-click the highlight button to the left of an assertion or directive, the Source View window shows the selected loop with the assertion or directive highlighted in the code.

Recall that assertions and directives are special Fortran source comments that tell the compiler how to transform Fortran code for multiprocessing. Directives enable, disable, or modify features of the compiler when it processes the source. Assertions provide the compiler with additional information about the source code that can sometimes improve optimization.

Some assertions or directives appear with an information block option button that allows you to **Keep** or **Delete it**. (If you compile o32, you can also **> Reverse it**.) Figure 6-12 shows an assertion block and its option button.

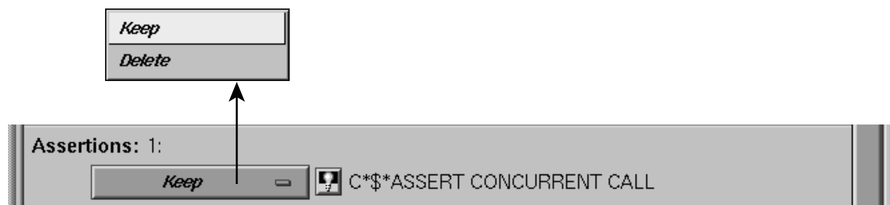


Figure 6-12 Assertion Information Block and Options (n32 and n64 Compilation)

Assertions and directives that govern loop parallelization or scheduling do not have associated option buttons; those functions are controlled by the loop parallelization status option button and the MP scheduling option button. (See "Loop Parallelization Controls in the Loop Information Display", page 131.)

Compiler Messages

The Loop information display also shows any messages generated by the compiler to describe aspects of the loops created by transforming original source loops. As an example, the loop information display in Figure 6-8, page 130, shows there are 11 messages present although only one is shown. Some messages have associated buttons that highlight sections of the selected loop in the Source View window.

Views Menu Options

The views in this section are launched from the Views menu in the main menu bar of the Parallel Analyzer View. All of the views discussed in this section contain the following in their menu bars:

- Admin menu: This menu contains a single Close command that closes the corresponding view.
- Help menu: This menu provides access to the online help system. (See "Help Menu", page 122, for an explanation of the commands in this menu.)

Parallelization Control View

The Parallelization Control View shows parallelization controls (directives and their clauses), where applicable, and all the variables referenced in the selected loop, OpenMP construct, or subroutine. It can be opened by either of two ways.

- Selecting the **Views > Parallelization Control View** option. Figure 6-13, page 137, shows the Parallelization Control View when it is launched from the Views menu with the Default loop parallelization status option button; this is the display for loops without directives.
- Selecting **C$OMP PARALLEL DO...** or **C$OMP DO...** in the loop parallelization status option button (Figure 6-14, page 139, and Figure 6-15, page 140). This approach provides controls for clauses you can append to these directives.

Features that appear no matter which method is used to open the Parallelization Control View are discussed under "Common Features of the Parallelization Control View", page 137. Features that appear only when the view is opened from the loop parallelization status option button with **C$OMP PARALLEL DO...** or **C$OMP DO...** selected are discussed in the following:

- "`C$OMP PARALLEL DO` and `C$OMP DO` Directive Information", page 138
- "MP Scheduling Option Button: Clauses for One Loop", page 142
- "Variable List Option Buttons", page 142

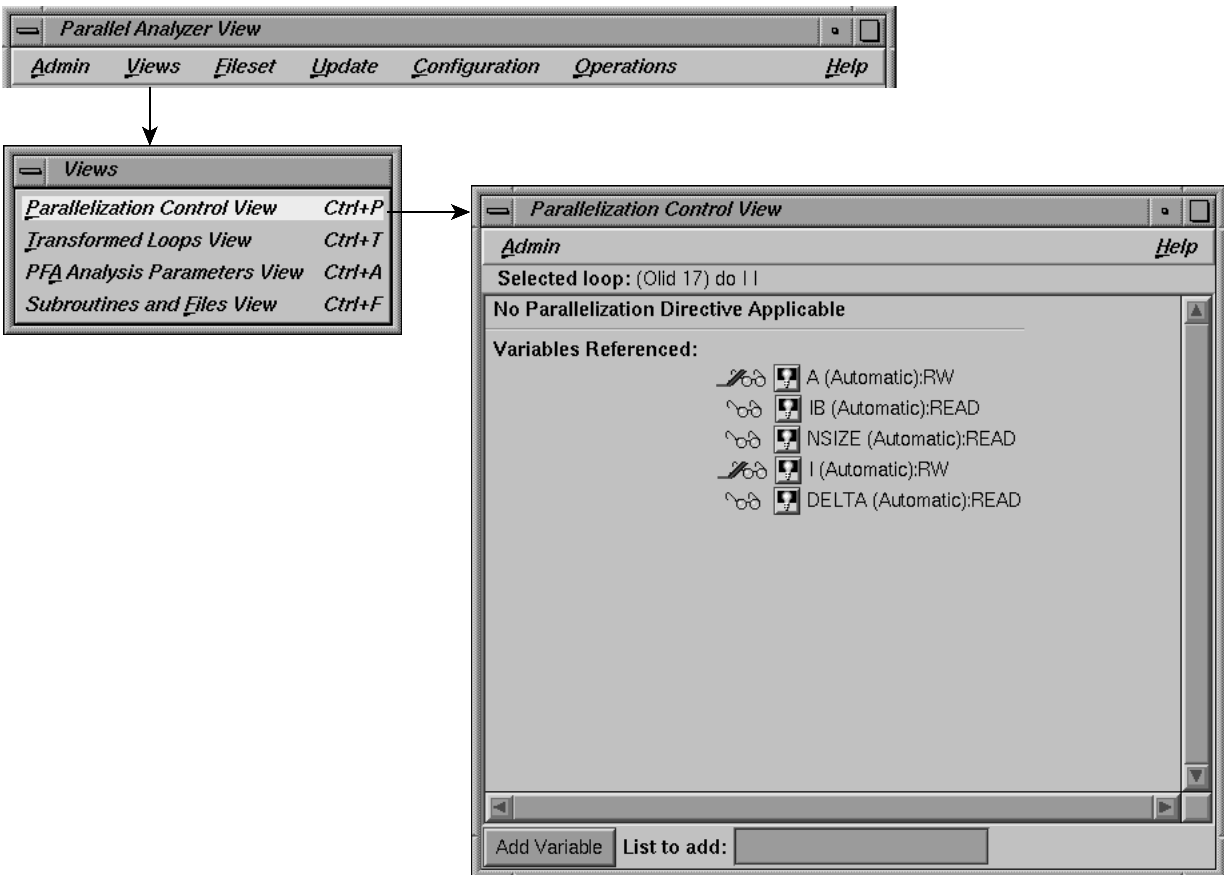


Figure 6-13 Parallelization Control View

Common Features of the Parallelization Control View

Independently of how you open the Parallelization Control View, these elements appear in the window (Figure 6-13, page 137):

- **Selected loop**: Contains the Olid of the loop, and the information about the loop from the Loop-ID and Variable columns of the loop list.
- **Directive information section**: If a directive is applicable to the loop, this section lists directive, clauses, and parameter values. (See "C$OMP PARALLEL DO and C$OMP DO Directive Information", page 138.)
- **Variables Referenced**: The listing has two icons for each variable. They allow you to highlight the variable in the Source View and to determine the variable's read/write status; see "Icon Legend… Option", page 112, for an explanation of these icons.

  For discussion of added option buttons that appear if the view is opened from the loop parallelization status option button when **C$OMP PARALLEL DO…** or **C$OMP DO…** is selected, see "Variable List Option Buttons", page 142.
- **Add Variable**: Located at the bottom of the window frame, this button allows you to add new variables to a loop.
- **List to add**: Located at the bottom of the window frame, this editable text field allows you to indicate the variables you wish to add to the loop. You may enter multiple variables, with each variable name separated by a space or comma.

`C$OMP PARALLEL DO` and `C$OMP DO` Directive Information

Option buttons and editable text fields in addition to those described in "Common Features of the Parallelization Control View", page 137, are available if you open the Parallelization Control View from the loop parallelization status option button with either **C$OMP PARALLEL DO…** or **C$OMP DO…** selected. (See Figure 6-14, page 139, and Figure 6-15, page 140.)

There are two additional option buttons available:

- **MP scheduling option button**: This button allows you to alter a loop's scheduling scheme by changing the `C$MP_SCHEDTYPE` clause. See "MP Scheduling Option Button: Clauses for One Loop", page 142, for further information. This is the same button shown in Figure 6-9, page 131.
- **Synchronization construct option button** (**C$OMP DO…** only): This button allows you to set the `NOWAIT` clause at the end of the `C$OMP END DO` directive to avoid the implied `BARRIER`.

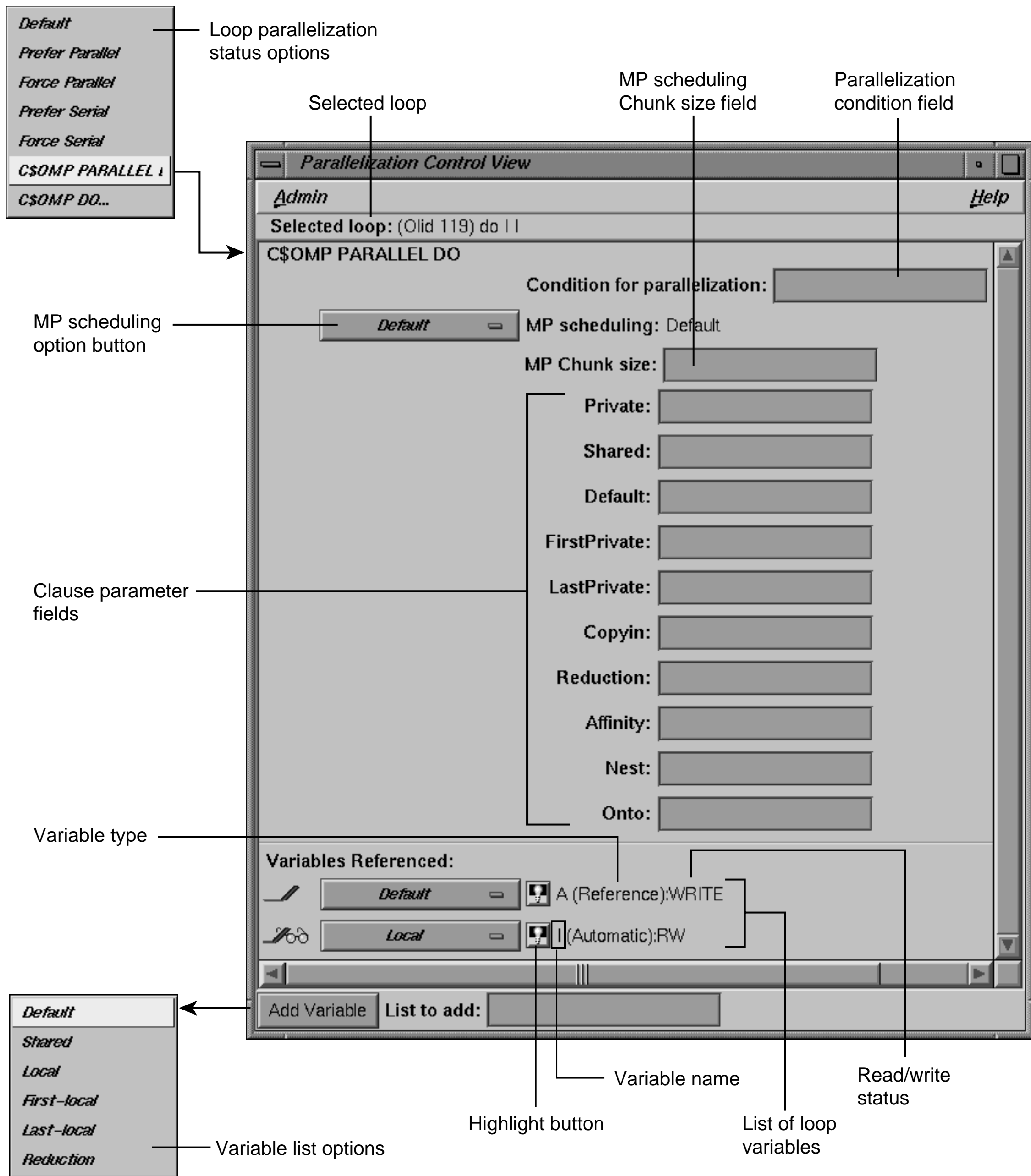


Figure 6-14 Parallelization Control View With `C$OMP PARALLEL DO` Directive

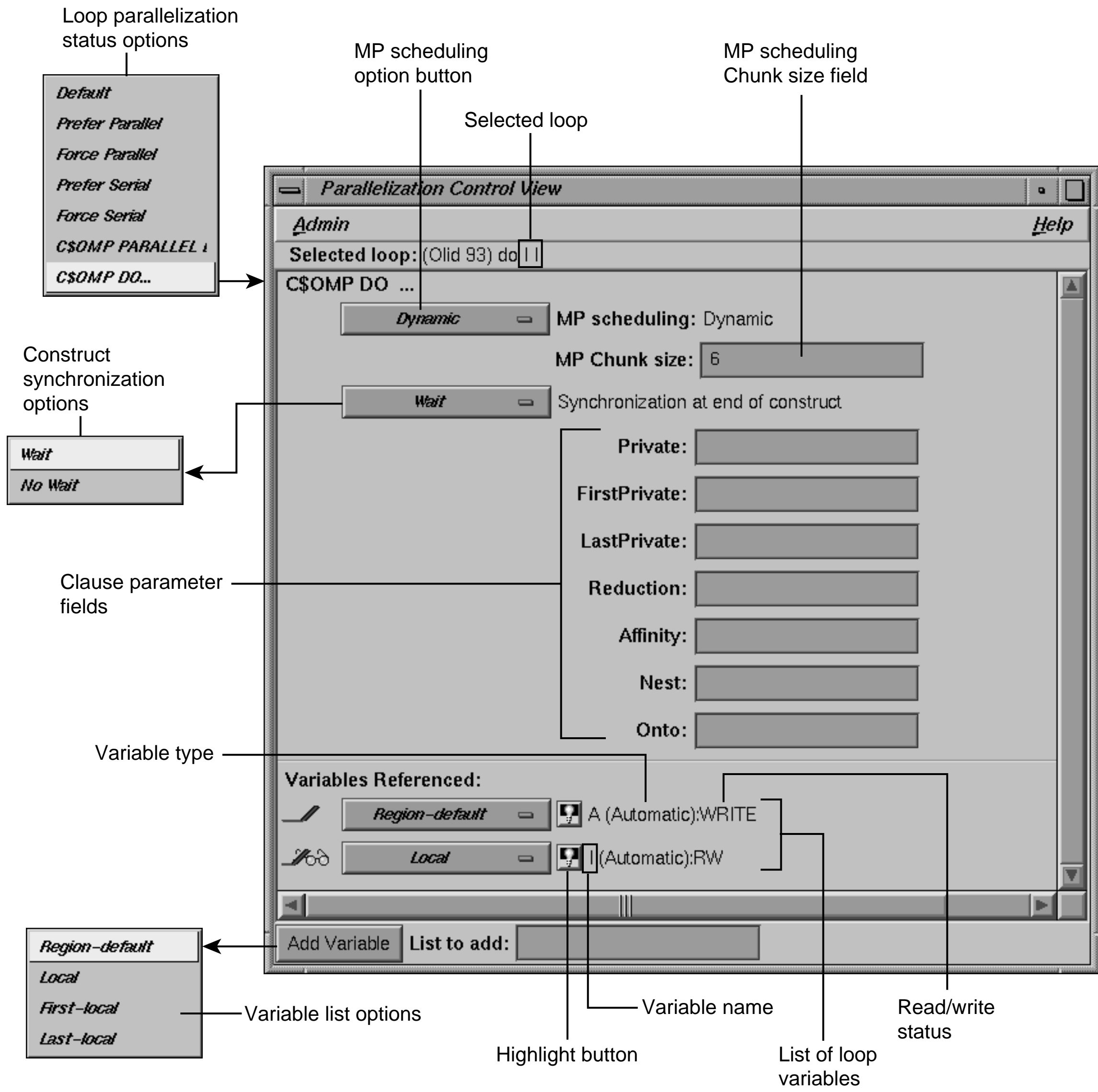


Figure 6-15 Parallelization Control View With `C$OMP DO` Directive

The following is a list of additional editable text fields that allow you to specify clauses for the C$OMP PARALLEL DO or C$OMP DO directives.

- **Condition for parallelization**: Allows you to enter a Fortran conditional statement, for example, NSIZE .GT. 83. (**C$OMP PARALLEL DO...** only.)

  This statement determines the circumstances under which the loop will be parallelized. The upper right corner of the field changes when you type in the field. Your entry must be syntactically correct; it is not checked.

- **MP Chunk size**: Allows you to set the C$CHUNK size for the scheduling scheme you select. For further information, see "MP Chunk Size Field", page 134.
- **Private**: Declares the variables in a list to be PRIVATE to each thread in a team.
- **Shared**: Makes variables that appear in a list shared among all the threads in a team. All threads within a team access the same storage area for SHARED data. (**C$OMP PARALLEL DO...** only.)
- **Default**: Allows you to specify a PRIVATE, SHARED, or NONE scope attribute for all variables in the lexical extent of any parallel region. Variables in THREADPRIVATE common blocks are not affected by this clause. (**C$OMP PARALLEL DO...** only.)
- **Firstprivate**: Provides a superset of the functionality provided by the PRIVATE clause.
- **Lastprivate**: Provides a superset of the functionality provided by the PRIVATE clause.
- **Copyin**: Applies only to common blocks that are declared as THREADPRIVATE. (**C$OMP PARALLEL DO...** only.)

  A COPYIN clause on a parallel region specifies that the data in the master thread of the team be copied to the thread private copies of the common block at the beginning of the parallel region.

- **Reduction**: Performs a reduction on the variables that appear in a list with an operator (+, *, -, .AND., .OR., .EQV., or .NEQV.), or an intrinsic (MAX, MIN, IAND, IOR, or IEOR).
- **Affinity**: Allows you to specify the parameters for the affinity scheduling clause. The two types of affinity scheduling are described below. For more details and syntax, see the *MIPSpro Fortran 77 Programmer's Guide*.

  - Data affinity scheduling, which assigns loop iterations to processors according to data distribution.
  - Thread affinity scheduling, which assigns loop iterations to designated processors.

- **Nest**: Allows you to specify parameters in this clause for concurrent execution of nested loops. You can use the `NEST` clause to parallelize nested loops only when there is no code between either the opening `DO` statements or the closing `ENDDO` statements. For more details and syntax, see the *MIPSpro Fortran 77 Programmer's Guide*.
- **Onto**: Allows you to specify parameters for this clause to determine explicitly how processors are assigned to array variables or loop iteration variables. For more details and syntax, see the *MIPSpro Fortran 77 Programmer's Guide*

MP Scheduling Option Button: Clauses for One Loop

The Parallelization Control View contains an MP scheduling option button if it is opened from the loop parallelization status option button with either **C$OMP PARALLEL DO...** or **C$OMP DO...** selected.

The options that appear have the same names as those for the MP scheduling option button in the loop information display, shown in Figure 6-9, page 131. However, the option button in the Parallelization Control View affects the `C$MP_SCHEDTYPE` and `C$CHUNK` clauses in the `C$OMP PARALLEL DO` directive, and so affects only the currently selected loop. Recall that the MP scheduling option button in the loop information display affects the placement of the `C$MP_SCHEDTYPE` and `C$CHUNK` directives and thus all subsequent loops.

Except for this difference in scope, the effects of both option buttons are the same; for a description, see "MP Scheduling Option Button: Directives for All Loops", page 133.

Variable List Option Buttons

If the Parallelization Control View is opened from the loop parallelization status option button when either **C$OMP PARALLEL DO...** or **C$OMP DO...** is selected, each variable listed in the lower portion of the view appears with an option button. The menu allows you to append a clause to the directive, enabling you to control how the processors manage the variable. It is an addition to the highlight and read/write icons discussed in "Common Features of the Parallelization Control View", page 137.

Note: The highlight button may not indicate in the Source View all the occurrences relevant to a variable subject to a OpenMP directive; you may need to select the entire parallel region in which the variable occurs.

If the view is opened from the loop parallelization status option button when **C$OMP PARALLEL DO…** is selected, the following are the variable list option button choices (Figure 6-14, page 139):

- **Default**: uses the control established by the compiler.
- **Shared**: one copy of the variable is used by all threads of the MP process.
- **Local**: each processor has its own copy of the variable.
- **Last-local**: similar to Local, except the value of the variable after the loop is as the logically last iteration would have left it.
- **Reduction** : a sum, product, minimum, or maximum computation of the variable can be done partially in each thread and then combined afterwards.

If the view is opened from the loop parallelization status option button when **C$OMP DO…** is selected, the following are the variable list option button choices (Figure 6-15, page 140):

- **Region-default**: uses the control established by the compiler for the parallel region.
- **Local**: Each processor has its own copy of the variable.
- **First-local**: similar to Local, except the value of the variable after the loop is as the logically first iteration would have left it.
- **Last-local**: similar to Local, except the value of the variable after the loop is as the logically last iteration would have left it.

Variable List Storage Labeling

In parentheses after each variable name in the list of variables is a word indicating the storage class of the variable. There are three possibilities:

- **Automatic**: The variable is local to the subroutine, and is allocated on the stack.
- **Common**: The variable is in a common block.

- **Reference**: The variable is a formal argument, or dummy variable, local to the subroutine.

Transformed Loops View

The Transformed Loops View contains information about how a loop selected from the loop list is rewritten by the compiler into one or more *transformed loops*.

To open this view, choose **Views > Transformed Loops View** (see Figure 6-16).

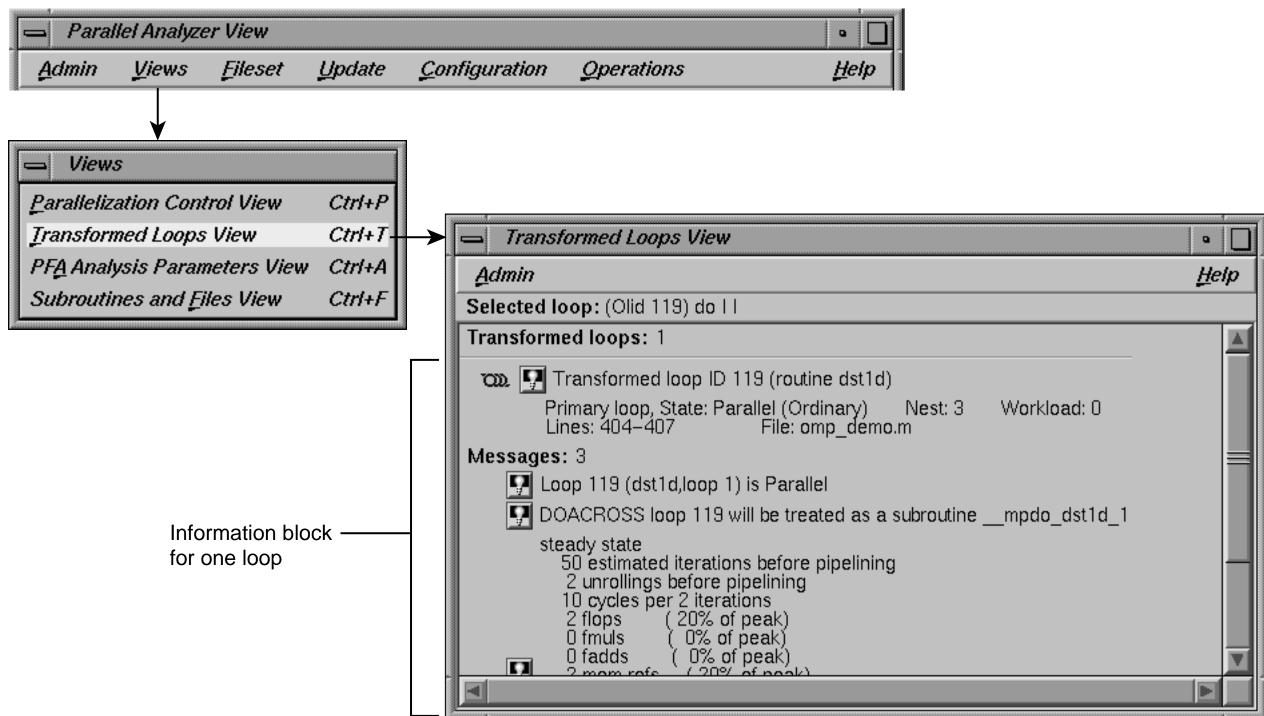


Figure 6-16 Transformed Loops View

Loop identifying information appears on the first line below the window menu, and below that is an indication of how many transformed loops were created.

Each transformed loop is displayed in its own section of the Transformed Loops View in an information block.

- The first line in each block for contains:
  - A parallelization status icon
  - A highlighting button (highlights the loop in the Transformed Source window and in the original loop in the Source View)
  - The Olid number of the transformed loop
- The next line describes the transformed loop, providing information such as the following:
  - Whether it is a *primary* loop or *secondary* loop (whether it is directly transformed from the selected original loop or transformed from a different original loop, but incorporates some code from the selected original loop)
  - Parallelization state
  - Whether it is an ordinary loop or interchanged loop
  - Its nesting level
- The last line in the loop's information block displays the location of the loop in the transformed source.

Any messages generated by the compiler are below the loop information blocks. To the left of the message lines are highlight buttons. Left-clicking them highlights in the Transformed View the part of the original source that relates to the message. Often it is the first line of the original loop that is highlighted, since the message refers to the entire loop.

PFA Analysis Parameters View

If you compile with `o32`, you can use the PFA Analysis Parameters View, which contains a list of PFA execution parameters accompanied by fields into which you can enter new values. If you compile with `n32` or `n64`, these parameters have no effect and this view is not useful.

To open this view, choose **Views > PFA Analysis Parameters View** in the main window. (See Figure 6-16, page 144.)

When you update a source file, any PFA parameters you alter are changed for that file (Figure 6-17). When you change a parameter, the upper right corner of the field window turns down, as discussed in "MP Chunk Size Field", page 134.

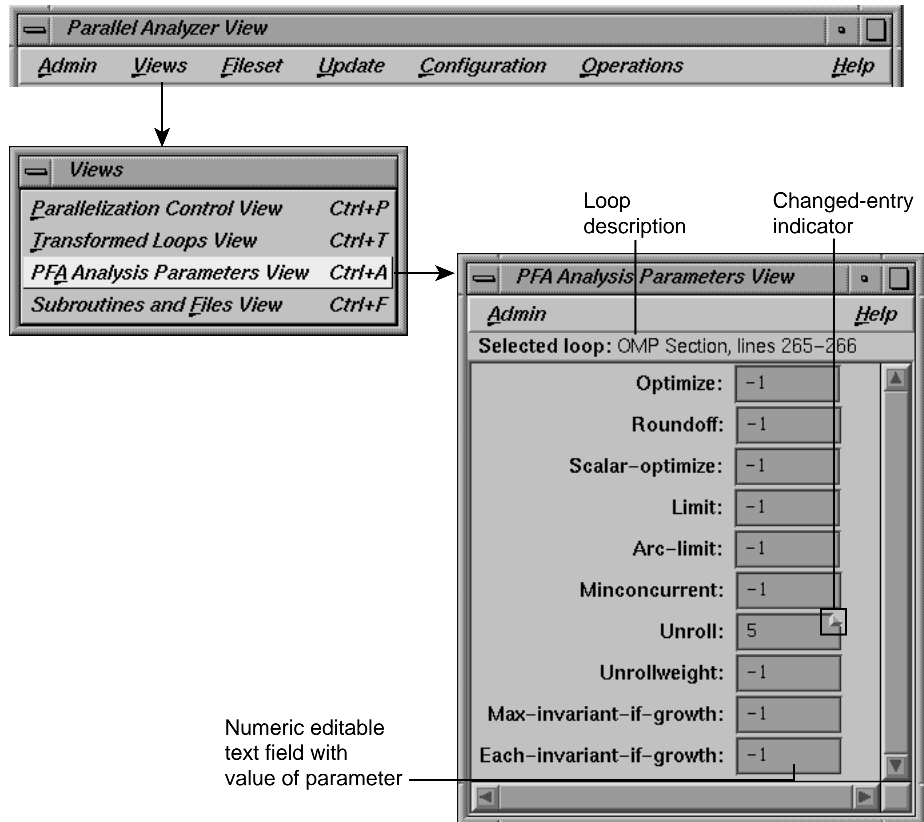


Figure 6-17 PFA Analysis Parameters View

Subroutines and Files View

The Subroutines and Files View contains a list from the file(s) in the current session of the Parallel Analyzer View (Figure 6-18, page 147). Below each filename in the list is an indented list of the Fortran subroutines it contains. Each item in the list is accompanied by icons to indicate file or subroutine status:

- A green check mark appears to the left of the file or subroutine name if the file has been scanned correctly or the subroutine has no errors.
- A red plus sign is above the green check mark shows if any changes have been made to loops in the file using the Parallel Analyzer View.
- A red international **not** symbol replaces the check mark if an error occurred because a file could not be scanned or a subroutine had errors.

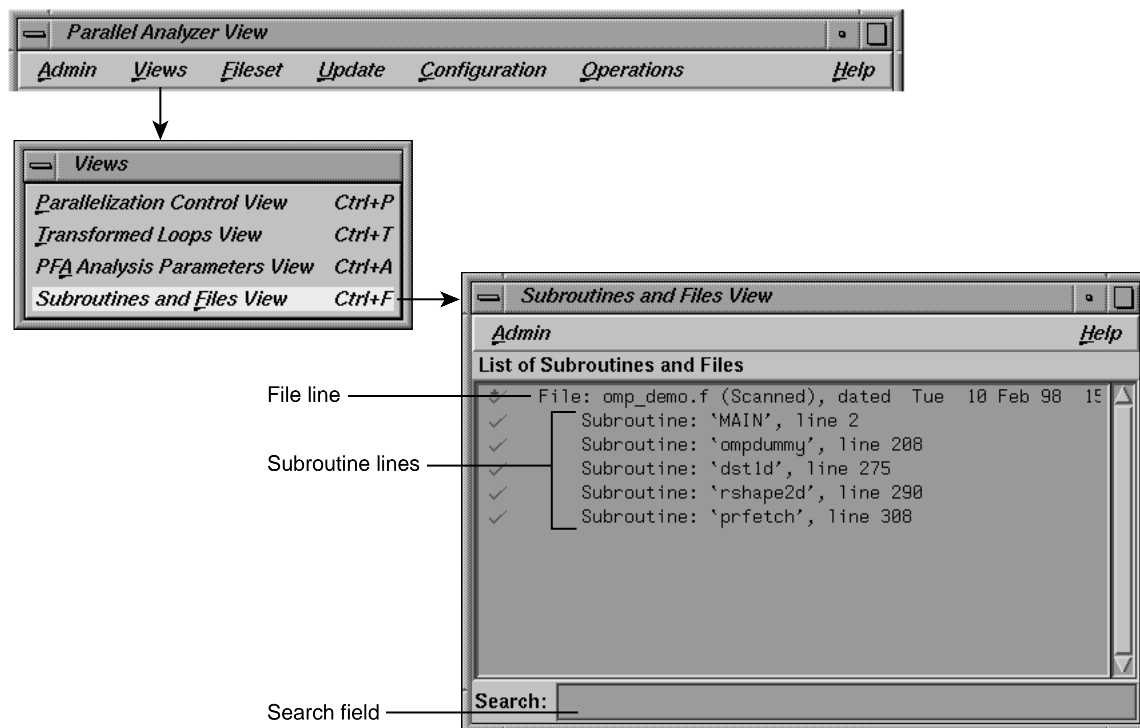


Figure 6-18 Subroutines and Files View

The Search field in the Parallel Analyzer View uses the subroutine and file names listed in the Subroutines and Files View as a menu for search targets; see "Search Loop List Field", page 127.

You can select items in the list for two purposes:

- To save changes to a selected file: click the filename and use the **Update > Update Selected File** option at the top of the Parallel Analyzer View main window. (See "Update Menu", page 117.)
- To select a file or subroutine for loop list filtering, discussed in "Filtering Option Button", page 128, double-click on it. The selected name appears in the filtering text field; if the item is appropriate for the selected filtering option, the loop list is rescanned.

At the bottom of the window is a Search editable text field, which you can use to search the list of files and subroutines.

Loop Display Control Button Views

These views are summoned by clicking on the **Source** and **Transformed Source loop** display control buttons.

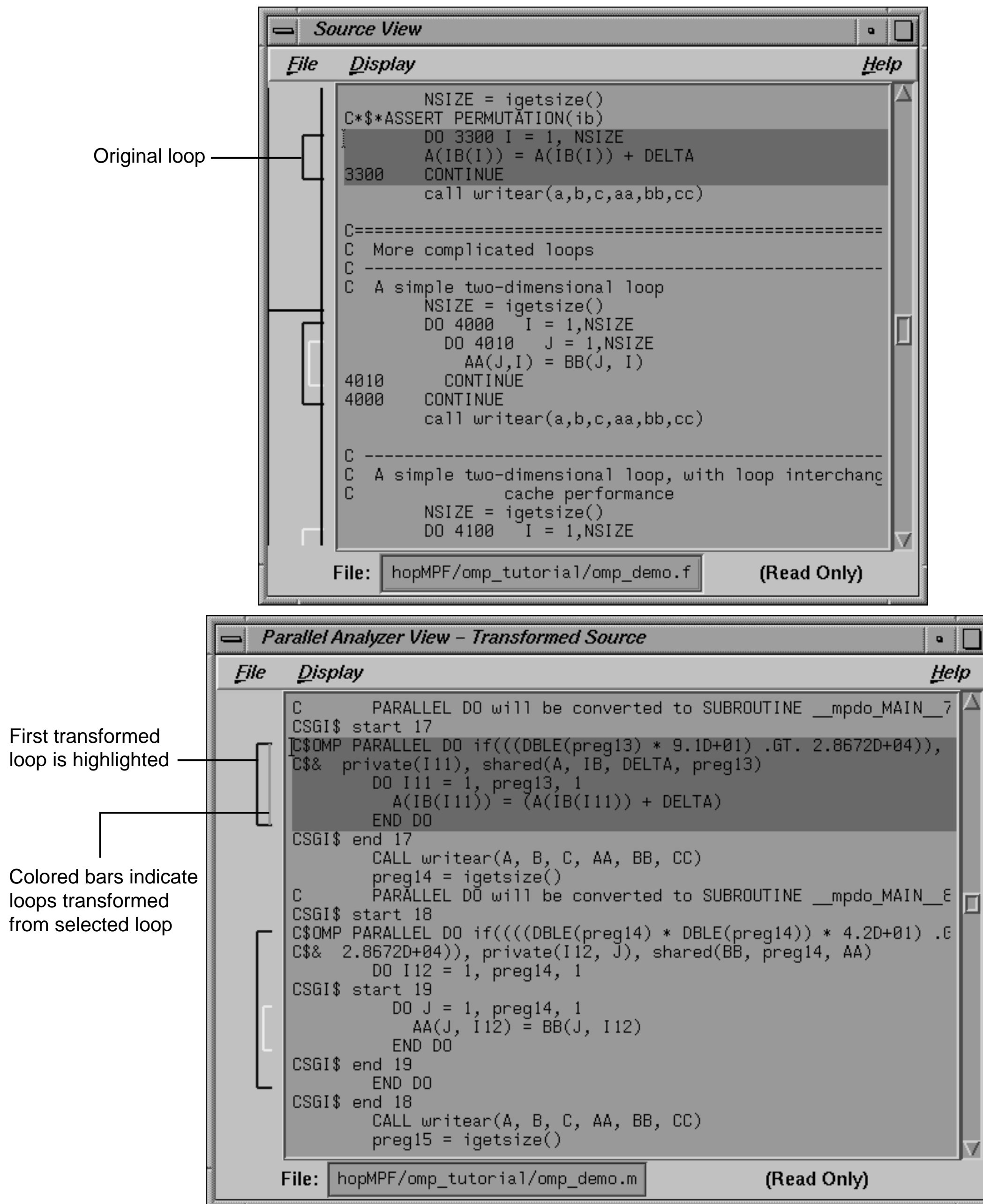
Source View and Parallel Analyzer View - Transformed Source

The Source View window and the Transformed Source window together present views of the source code before and after compiler optimization (Figure 6-19, page 149). The two windows use the WorkShop Source View interface.

Both the Source View and Transformed Source windows contain bracket annotations in the left margin that mark the location and nesting level of each loop in the source file. Clicking on a loop bracket to the left of the code chooses and highlights the corresponding loop.

In the Transformed Source window, an indicator bar (a vertical line in a different color) indicates each loop that was transformed from the selected original loop.

If the source windows are invoked from a session linked to the WorkShop Performance Analyzer (see "Launch Tool Submenu", page 112), any displayed sources files known to the Performance Analyzer are annotated with performance data.




```
        NSIZE = igetsize()
C*$*ASSERT PERMUTATION(ib)
        DO 3300 I = 1, NSIZE
        A(IB(I)) = A(IB(I)) + DELTA
3300    CONTINUE
        call writear(a,b,c,aa,bb,cc)

C=====================================================
C  More complicated loops
C ----------------------------------------------------
C  A simple two-dimensional loop
        NSIZE = igetsize()
        DO 4000   I = 1,NSIZE
          DO 4010   J = 1,NSIZE
            AA(J,I) = BB(J, I)
4010      CONTINUE
4000    CONTINUE
        call writear(a,b,c,aa,bb,cc)

C ----------------------------------------------------
C  A simple two-dimensional loop, with loop interchanc
C               cache performance
        NSIZE = igetsize()
        DO 4100   I = 1,NSIZE
```

```
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__7
CSGI$ start 17
C$OMP PARALLEL DO if(((DBLE(preg13) * 9.1D+01) .GT. 2.8672D+04)),
C$&  private(I11), shared(A, IB, DELTA, preg13)
        DO I11 = 1, preg13, 1
          A(IB(I11)) = (A(IB(I11)) + DELTA)
        END DO
CSGI$ end 17
        CALL writear(A, B, C, AA, BB, CC)
        preg14 = igetsize()
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__8
CSGI$ start 18
C$OMP PARALLEL DO if((((DBLE(preg14) * DBLE(preg14)) * 4.2D+01) .G
C$&  2.8672D+04)), private(I12, J), shared(BB, preg14, AA)
        DO I12 = 1, preg14, 1
CSGI$ start 19
          DO J = 1, preg14, 1
            AA(J, I12) = BB(J, I12)
          END DO
CSGI$ end 19
        END DO
CSGI$ end 18
        CALL writear(A, B, C, AA, BB, CC)
        preg15 = igetsize()
```

Figure 6-19 Original and Transformed Source Windows

Note: The File and Display menus shown in the Source View and Transformed Source windows are standard Source View menus, and are described in the *ProDev WorkShop: Debugger User's Guide*.

Examining Loops Containing PCF Directives

The content of this appendix is similar to that of "Examples Using OpenMP Directives", page 50, except it uses the older PCF (Parallel Computing Forum) directives instead of OpenMP directives.

Setting Up the dummy.f Sample Session

To use this sample session, note the following:

- `/usr/demos/ProMP` is the PCF demonstration directory
- `ProMP.sw.demos` must be installed

The sample session discussed in this chapter uses the following source files in the directory `/usr/demos/ProMP/tutorial`:

- `dummy.f_orig`
- `pcf.f_orig`
- `reshape.f_orig`
- `dist.f_orig`

The source files contain many `DO` loops, each of which exemplifies an aspect of the parallelization process.

The directory `/usr/demos/ProMP/tutorial` also includes `Makefile` to compile the source files.

Compiling the Sample Code

Prepare for the session by opening a shell window and entering the following:

```
% cd /usr/demos/ProMP/tutorial
% make
```

This creates the following files:

- `dummy.f`: a copy of the demonstration program created by combining the `*.f_orig` files, which you can view with the Parallel Analyzer View or a text editor, and print
- `dummy.m`: a transformed source file, which you can view with the Parallel Analyzer View, and print
- `dummy.l`: a listing file
- `dummy.anl`: an analysis file used by the Parallel Analyzer View

Starting the Parallel Analyzer View

Once you have created the appropriate files with the compiler, start the session by entering the following command, which opens the main window of the Parallel Analyzer View loaded with the sample file data:

```
% cvpav -f dummy.f
```

Open the Source View window by clicking the **Source** button after the Parallel Analyzer View main window opens.

Examples Using PCF Directives

This section discusses the subroutine `pcfdummy()`, which contains four parallel regions and a single-process section that illustrate the use of PCF directives:

- "Explicitly Parallelized Loops: `C$PAR PDO`", page 153
- "Loops With Barriers: `C$PAR BARRIER`", page 154
- "Critical Sections: `C$PAR CRITICAL SECTION` ", page 156
- "Single-Process Sections: `C$PAR SINGLE PROCESS`", page 157
- "Parallel Sections: `C$PAR PSECTIONS`", page 157

To go to the first explicitly parallelized loop in `pcfdummy()`, scroll down the loop list to Olid 92.

Select this loop by double-clicking the highlighted line in the loop list.

Explicitly Parallelized Loops: C$PAR PDO

The first construct in subroutine `pcfdummy()` is a parallel region, Olid 92, containing two loops that are explicitly parallelized with `C$PAR PDO` statements. (See Figure A-1, page 154.) With this construct, the second loop can start before all iterations of the first complete.

Example 6-1 Explicitly Parallelized Loop Using `C$PAR PDO`

```
C$PAR PARALLEL SHARED(A,B) LOCAL(I)
C$PAR PDO dynamic blocked(10-2*2)
        DO 6001 I=-100,100
           A(I) = I
6001   CONTINUE
C$PAR PDO static
        DO 6002 I=-100,100
           B(I) = 3 * A(I)
6002   CONTINUE
C$PAR END PARALLEL
```

Notice in the loop information display that the parallel region has controls for the region as a whole. The **Keep option** button and the highlight buttons function the same way they do in the Loop Parallelization Controls.

Click **Next Loop** twice to step through the two loops. You can see in the Source View that both loops contain a `C$PAR PDO` directive.

Click **Next Loop** to step to the second parallel region.

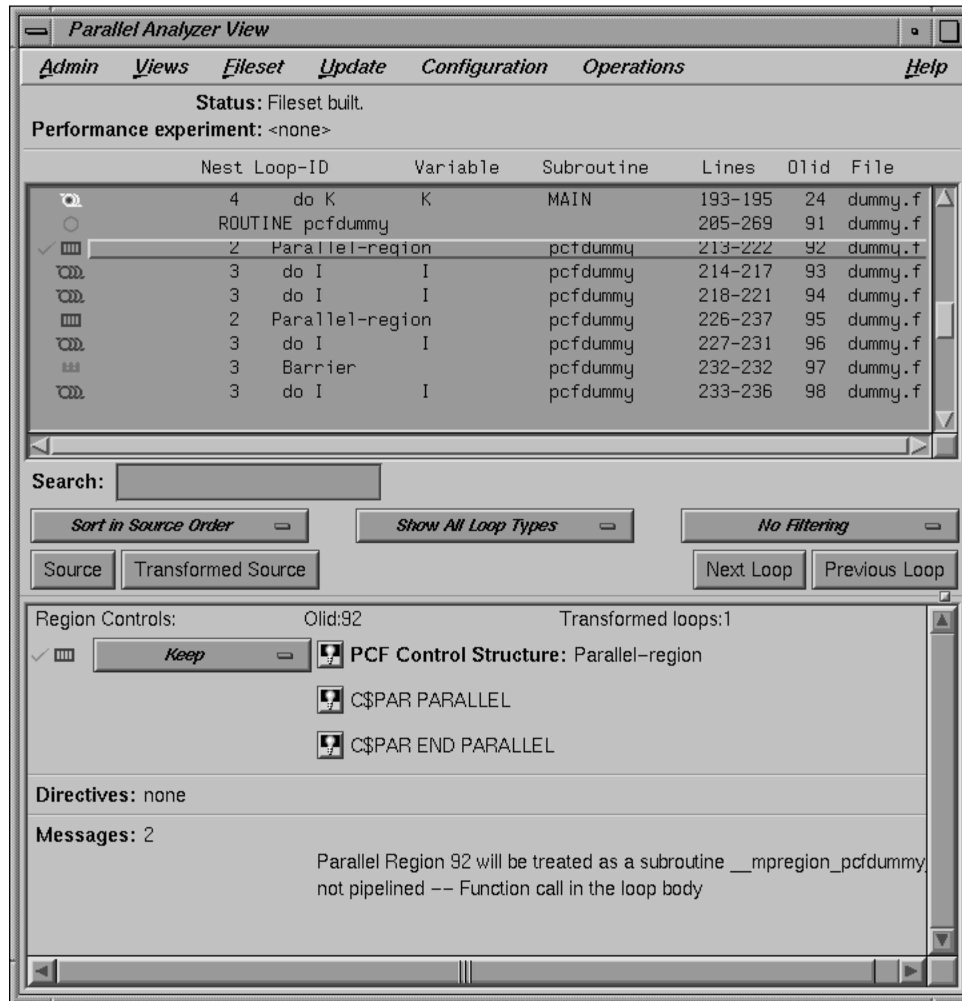


Figure A-1 Explicitly Parallelized Loops Using `C$PAR PDO`

Loops With Barriers: `C$PAR BARRIER`

The second parallel region, Olid 95, contains a pair of loops identical to the previous example, but with a barrier between them. Because of the barrier, all iterations of the first `C$PAR PDO` must complete before any iteration of the second loop can begin.

Example 6-2 Loops Using `C$PAR BARRIER`

```
C$PAR PARALLEL SHARED(A,B) LOCAL(I)
C$PAR PDO interleave blocked(10-2*2)
        DO 6003 I=-100,100
           A(I) = I
6003   CONTINUE
C$PAR END PDO NOWAIT
C$PAR barrier
C$PAR PDO static
        DO 6004 I=-100,100
           B(I) = 3 * A(I)
6004   CONTINUE
C$PAR END PARALLEL
```

Click **Next Loop** twice to view the barrier region. (See Figure A-2, page 156.)

Click **Next Loop** twice to go to the third parallel region.

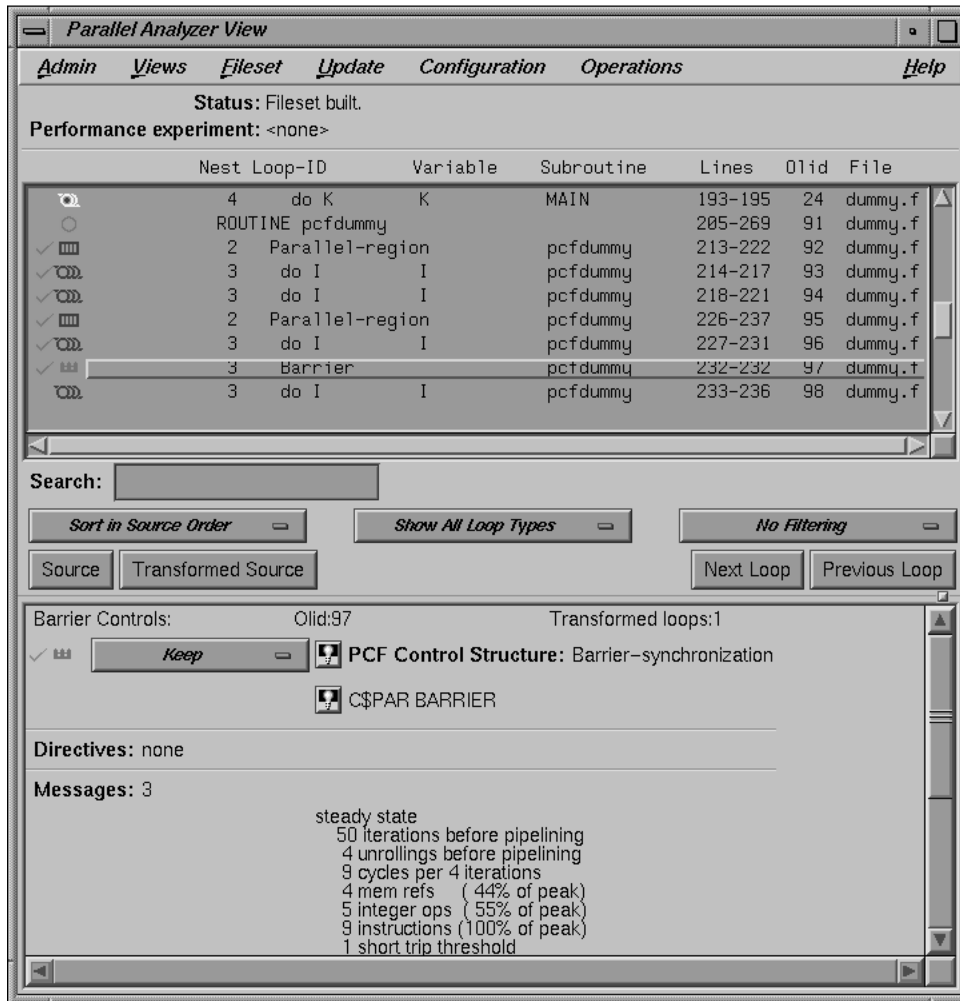


Figure A-2 Loops Using C$PAR BARRIER Synchronization

Critical Sections: `C$PAR CRITICAL SECTION`

Click **Next Loop** to view the first of the two loops in the third parallel region, Olid 100. This loop contains a critical section.

Example 6-3 Critical Section Using `C$PAR CRITICAL SECTION`

```
C$PAR PDO
       DO 6005 I=1,100
C$PAR CRITICAL SECTION (S3)
           S1 = S1 + I
C$PAR END CRITICAL SECTION
6005  CONTINUE
```

Click **Next Loop** to view the critical section.

The critical section uses a named locking variable (*S3*) to prevent simultaneous updates of *S1* from multiple threads. This is a standard construct for performing a reduction.

Move to the next loop by clicking **Next Loop**.

Single-Process Sections: `C$PAR SINGLE PROCESS`

Loop Olid 102 has a single-process section, which ensures that only one thread can execute the statement in the section. Highlighting in the Source View shows the begin and end directives.

Example 6-4 Single-Process Section Using `C$PAR SINGLE PROCESS`

```
       DO 6006 I=1,100
C$PAR SINGLE PROCESS
           S2 = S2 + I
C$PAR END SINGLE PROCESS
6006  CONTINUE
```

Click **Next Loop** to view information about the single-process section.

Move to the final parallel region in `pcfdummy()` by clicking **Next Loop**.

Parallel Sections: `C$PAR PSECTIONS`

The fourth and final parallel region of `pcfdummy()`, Olid 104, provides an example of parallel sections. In this case, there are three parallel subsections, each of which calls a function. Each function is called exactly once, by a single thread. If there are three or more threads in the program, each function may be called from a different

thread. The compiler treats this directive as a single-process directive, which guarantees correct semantics.

Example 6-5 Parallel Section Using `C$PAR PSECTIONS`

```
C$PAR PARALLEL shared(a,c) local(i,j)
C$PAR PSECTIONS
        call boo
C$PAR SECTION
        call bar
C$PAR SECTION
        call baz
C$PAR END PSECTIONS
C$PAR END PARALLEL
```

Click **Next Loop** to view the parallel section.

Exiting From the dummy.f Sample Session

This completes the PCF sample session.

Close the Source View window by choosing its **File > Close** option.

Quit the Parallel Analyzer View by choosing **Admin > Exit**.

To clean up the directory, enter the following in your shell window to remove all of the generated files:

```
% make clean
```

Index

A

Add assertion submenu
  in operations menu, 121
Add file
  option in fileset menu, 117
Add files from executable
  option in fileset menu, 117
Add files from fileset
  option in fileset menu, 117
Add omp atomic
  option in add omp section submenu, 122
Add omp barrier
  option in operations menu, 121
Add omp critical
  option in add omp section submenu, 121
Add omp directive
  option in operations menu, 121
Add omp master
  option in add omp section submenu, 122
Add omp ordered
  option in add omp section submenu, 122
Add omp parallel
  option in operations menu, 121
Add omp section
  option in add omp section submenu, 121
Add omp section submenu
  Add omp atomic option, 122
  Add omp critical option, 121
  Add omp master option, 122
  Add omp ordered option, 122
  Add omp section option, 121
  Add omp sections option, 121
  Add omp single option, 121
  in operations menu, 121
Add omp sections
  option in add omp section submenu, 121
Add omp single
  option in add omp section submenu, 121
Add variable button
  in parallelization control view, 138
adding an assertion, 45, 85
adjustment button
  resize loop list display, 10, 124
Admin menu
  Exit option, 112
  Icon legend… option, 111, 112
  Iconify option, 111
  in parallel analyzer view, 110
  in views menu options, 136
  Launch tool submenu, 111, 112
  Project submenu, 111
  Raise option, 111
  Save as text option, 110
AFFINITY clause, 43
  Parallelization control view and, 141
Affinity field
  in parallelization control view, 141
analyzing loops
  C, 69
  FORTRAN 77, 5
  Fortran 90, 61
apo keep command line option, 2
assert concurrent call
  adding, 86
  deleting, 87
assertions
  adding from loop parallelization controls, 131
  adding from operations menu, 119
  controlling, 41, 82
  deleting, 46, 87
Assertions information block
  in loop information display, 135
Automatic storage

variable list storage label, 143

B

barrier
OpenMP, 53, 91
PCF, 154
brackets
colors, 15
loop, 23
bugs, reporting, 126
Build analyzer
option in launch tool submenu, 112
Build manager, 46

C

C, 69
C$CHUNK variable, 133, 134
MP scheduling option button and, 133
Parallelization control view and, 141
C$MP_SCHEDTYPE variable, 133
MP scheduling option button and, 133
C$OMP barrier, 53, 121
C$OMP critical, 55
C$OMP do, 51, 132
C$OMP do…
option in loop parallelization status option button, 132
Parallelization control view and, 138
C$OMP flush, 121
C$OMP parallel, 121
C$OMP parallel do, 25, 132
adding, 42
C$SGI distribute and, 57
C$SGI&NEST and, 39
C$OMP parallel do…
option in loop parallelization status option button, 132
Parallelization control view and, 138
C$OMP sections, 55
C$OMP single, 55
C$PAR barrier, 154
C$PAR critical section, 156
C$PAR pdo, 153
C$PAR psections, 157
C$PAR single process, 157
C$SGI distribute, 57, 120, 121
C$SGI dynamic, 120, 121
C$SGI redistribute, 121
C*$* assert concurrent call, 34, 120, 121
adding, 45
deleting, 46
C*$* assert do (CONCURRENT), 30, 132
C*$* assert do (SERIAL), 132
C*$* assert do prefer (CONCURRENT), 132
C*$* assert do prefer (SERIAL), 132
C*$* assert permutation, 35, 120, 121
C*$* concurrentize, 120, 121
C*$* noconcurrentize, 120, 121
C*$* prefetch_REF, 59, 120, 121
cache
prefetching data from, 59, 97
caliper setting in performance analyzer, 124
changed-entry indicator, 134
check mark, 125
closing all windows, project submenu, exit option, 115
colors, brackets and icons, 15
command line options, 5
Common storage
variable list storage label, 144
compiler messages, 136
compiling
C, 70
Fortran 90, 61
compiling sample code
FORTRAN 77, 6
Condition for parallelization field
in parallelization control view, 141
Configuration menu

in parallel analyzer view, 118
OpenMP option, 119
PCF option, 119
COPYIN clause, 43
Parallelization control view and, 141
THREADPRIVATE directive and, 141
Copyin field
in parallelization control view, 141
critical section
OpenMP, 55, 92
PCF, 156
cvpav
compiling for, 2
opening editor, 49, 118
starting, 3

D

data dependence
carried
parallelizable, 30, 78
unparallelizable, 29, 75
multi-line, 31, 78
daxpy subroutine, linpackd session, 103
Debugger
option in launch tool submenu, 112
Default
C$MP_SCHEDTYPE mode, 133
option in loop parallelization status option button, 132
option in mp scheduling option button, 133
option in variable list option button, 143
DEFAULT clause, 43
Parallelization control view and, 141
Default field
in parallelization control view, 141
Delete all files
option in fileset menu, 116
Delete information block option button, 135
Delete selected file
option in fileset menu, 116
demonstration
OpenMP, 6
PCF, 152
demonstration directory
Fortran 90 sample session, 61
OpenMP sample session, 5, 69
PCF sample session, 151
dgefa subroutine, linpackd session, 103
directive information
in parallelization control view, 138
directives
adding from loop parallelization controls, 131
adding from mp scheduling option menu, 133
adding from operations menu, 119
controlling, 41, 82
deleting, 46, 87
OpenMP, 90
Directives information block
in loop information display, 135
distributed and reshaped array
C$SGI distribute_RESHAPE, 58
#pragma distribute_reshape, 96
distributed arrays, 56, 94
dst1d function, 94
dst1d subroutine, omp_demo.f session, 56
Dynamic
C$MP_SCHEDTYPE mode, 133
option in mp scheduling option button, 133

E

Exit
option in admin menu, 112
option in project submenu, 114
explicitly parallelized loop
OpenMP, 51, 90
PCF, 153

F

file
  update, 46
File loop list field, 126
Fileset menu
  Add file option, 117
  Add files from executable option, 117
  Add files from fileset option, 117
  Delete all files option, 116
  Delete selected file option, 117
  in parallel analyzer view, 116
Filter by file
  option in filtering option button, 128
Filter by subroutine
  option in filtering option button, 128
filtering
  by file, 14
  by parallelization state, 12
  option menus, 12
filtering option button
  Filter by file option, 128
  Filter by subroutine option, 128
  in loop display controls, 128
  No filtering option, 128
First-local
  option in variable list option button, 143
FIRSTPRIVATE clause, 43
  Parallelization control view and, 141
Firstprivate field
  in parallelization control view, 141
foo subroutine, omp_demo.f session, 50
Force a build to start
  option in update menu, 118
Force parallel
  option in loop parallelization status option button, 132
Force serial
  option in loop parallelization status option button, 132
FORTRAN 77, 5
FORTRAN 77 tutorial
  topics, 5
Fortran 90, 61
function call
  parallelizable, 79

G

gdiff, 47
Guided self
  option in mp scheduling option button, 133
  Scheduling, c$MP_SCHEDTYPE mode, 133

H

Help menu
  in parallel analyzer view, 122
  in views menu options, 136
  Index… option, 123
  On context option, 122
  On version… option, 122
  On window… option, 122
highlight button, 20, 130
  directives, 131
highlighting a loop, 126

I

Icon legend…
  dialog box, 112
  option in admin menu, 111, 112
Iconify
  option in admin menu, 111
  option in project submenu, 113
icons
  check mark, 18
  description, 112
  loop list, 10
Index…

option in help menu, 123
information blocks
Assertions, 135
Directives, 135
Obstacles to parallelization, 134
option buttons
Delete, 135
Keep, 135
Reverse, 135
input/output operation, 32, 79
Interleaved
C$MP_SCHEDTYPE mode, 133
option in mp scheduling option button, 133

K

Keep information block option button, 135
keyboard shortcuts, 123

L

Last-local
option in variable list option button, 143
LASTPRIVATE clause, 43
Parallelization control view and, 141
Lastprivate field
in parallelization control view, 141
Launch tool submenu
Debugger option, 112
Launch tool submenu
Build analyzer option, 112
in admin menu, 111, 112
Parallel analyzer option, 113
Performance analyzer option, 113
Static analyzer option, 113
Tester option, 113
light bulb button, 20
Lines loop list field, 126
LINPACK, 99
List to add field
in parallelization control view, 138
Local
option in variable list option button, 143
loop
complex, 39, 81
detailed information, 15
doubly nested, 39, 81
examining simple, 24, 70
explicitly parallelized, 25, 72
fused, 27, 74
information blocks, 20
optimized away, 28, 74
primary, 21
secondary, 21
serial, 25, 72
simple parallel, 25, 71
status, 125
transformed, 21
selecting, 23
with obstacles to parallelization, 28, 75
loop display controls, 126
buttons, 129
control button
Source, 129
Transformed source, 129
navigation button
Next loop, 129
Previous loop, 129
option button
filtering, 128
show loop types, 128
sort, 127
loop information display, 19
in parallel analyzer view, 129
Loop parallelization controls, 131
loop list, 125
column contents, 125
filtering, 12
in loop list display, 10
sorting, 11
loop list display, 10, 123

loop list, 10
loop list icons, 10
Loop parallelization controls, 19
in loop information display, 131
loop parallelization status option button, 131
MP chunk size field, 134
MP scheduling option button, 133
loop parallelization status option button
C$OMP do… option, 132
C$OMP parallel do… option, 42, 132
Default option, 132
Force parallel option, 132
Force serial option, 132
in loop parallelization controls, 131
#pragma omp parallel for… option, 84
Prefer parallel option, 132
Prefer serial option, 132
Loop-ID
loop list field, 10, 126

M

main window
menu bar, 110
make clean, 67, 98
OpenMP sample session, 7, 60, 70
PCF sample session, 158
performance session, 105
messages
obstacles to parallelization, 28, 75
modifying source files, 41, 82
MP chunk size field, 43
in loop parallelization controls, 134
in parallelization control view, 141
MP scheduling option button
Default option, 133
Dynamic option, 133
Guided self option, 133
in loop parallelization controls, 133
in parallelization control view, 142, 138
Interleaved option, 133
Run-time option, 133
Static option, 133
MP scheduling option menu, 133

N

NEST clause, 43
Parallelization control view and, 142
Nest field
in loop list, 10, 126
in parallelization control view, 142
nested loops, 39, 81
Next loop navigation button
in loop display controls, 129
No filtering
option in filtering option button, 128

O

O3
command line option, 2
optimization level, 32, 79
obstacles to parallelization, 28, 75
Obstacles to parallelization information block
dependence messages, 37
in loop information display, 134
Olid
loop list, 10
loop list field, 126
omp_demo, 90
ompdummy subroutine, omp_demo.f session, 50, 55
On context
option in help menu, 122
On version…
option in help menu, 122
On window…
option in help menu, 122
ONTO clause, 43

Parallelization control view and, 142
Onto field
in parallelization control view, 142
OpenMP
option in configuration menu, 119
OpenMP directives, 90
Operations menu
Add assertion submenu, 121
Add omp barrier option, 121
Add omp directive option, 121
Add omp parallel option, 121
Add omp section submenu, 121
in parallel analyzer view, 119
Undo all changes option, 121
Undo changes to loop option, 120
original loop id. See Olid, 10

P

Parallel analyzer
launching, 113
option in launch tool submenu, 112
Parallel Analyzer View
source view, 15
Parallel analyzer view
Admin menu, 110
Configuration menu, 118
Fileset menu, 116
Help menu, 122
loop information display, 129
menu bar, 110
OpenMP support, 3
Operations menu, 119
starting, 3
Update menu, 117
Views menu, 116
parallel analyzer view
compiling for, 2
Parallel Analyzer View - transformed source, 23
Parallel analyzer view - transformed source
Transformed source control button and, 148
parallel sections
OpenMP, 55, 93
PCF, 157
parallelization
status option menu, 12
Parallelization control view, 136
Add variable button, 138
brought up by a highlight button, 58, 96
C$CHUNK variable and, 141
C$OMP do… button and, 138
C$OMP parallel do… button and, 138
directive clauses
AFFINITY, 142
COPYIN, 141
DEFAULT, 141
FIRSTPRIVATE, 141
LASTPRIVATE, 141
NEST, 142
ONTO, 142
PRIVATE, 141
REDUCTION, 141
SHARED, 141
directive fields
Affinity, 141
Condition for parallelization, 141
Copyin, 141
Default, 141
Firstprivate, 141
Lastprivate, 141
MP chunk size field, 141
Nest, 142
Onto, 142
Private, 141
Reduction, 141
Shared, 141
directive information, 138
List to add field, 138
loop status option menu and, 132
MP scheduling option button, 138
one loop clauses, 142
option in views menu, 116

Selected loop field, 138
Synchronization construct option button, 138
variable list option button, 142
C$OMP do… option and, 143
C$OMP parallel do… option and, 143
Default option, 143
First-local option, 143
Last-local option, 143
Local option, 143
Reduction option, 143
Region-default option, 143
Shared option, 143
variable list storage labels
Automatic, 143
Common, 143
Reference, 144
Variables referenced section, 138
parallelization icon
in loop list, 125
PCF
option in configuration menu, 119
pcfdummy subroutine, dummy.f session, 152
Perf. cost loop list field, 125
performance
cost per loop, 125
Performance analyzer, 99
launching, 113
option in launch tool submenu, 113
performance experiment line, 124
Performance experiment line, 124
performance session
exiting, 105
starting, 99
permutation vector, 34, 80
parallelizable, 34, 80
unparallelizable, 34, 80
PFA analysis parameters view
in views menu, 145
option in views menu, 116
plus sign, 125
red, 125
#pragma concurrent, 78
#pragma concurrent call, 79
#pragma distribute, 94
#pragma omp barrier, 91
#pragma omp critical, 92
#pragma omp for, 90
#pragma omp parallel for, 72, 81
adding, 83
#pragma omp sections, 93
#pragma omp single, 93
#pragma parallel for
and #pragma distribute, 96
#pragma permutation, 80
#pragma prefetch_ref, 97
Prefer parallel
option in loop parallelization status option button, 132
Prefer serial
option in loop parallelization status option button, 132
Previous loop navigation button
in loop display controls, 129
prfetch function, 97
prfetch subroutine, omp_demo.f session, 59
PRIVATE clause, 43
Parallelization control view and, 141
Private field
in parallelization control view, 141
Project submenu, 113
Exit option, 115
Iconify option, 113
in admin menu, 111
Project view… option, 114
Raise option, 113
Remap paths… option, 113
Project view…
option in project submenu, 113

R

Raise

option in admin menu, 111
option in project submenu, 113
recurrence, 29, 75
Reduction
option in variable list option button, 143
reduction, 31, 78
REDUCTION clause, 43
Parallelization control view and, 141
Reduction field
in parallelization control view, 141
Reference storage
variable list storage label, 144
Region-default
option in variable list option button, 143
Remap paths…
option in project submenu, 113
Rescan all files
option in fileset menu, 116
resize loop list display, 10
Reverse information block option button, 135
round-off, 32
roundoff, 79
rshape2d function, 96
rshape2d subroutine, omp_demo.f session, 58
RTC subroutine, omp_demo.f session, 33, 45, 50
Run editor after update
option in update menu, 118
Run gdiff after update
option in update menu, 117
Run-time
C$MP_SCHEDTYPE mode, 133
option in mp scheduling option button, 133

S

sample session
Performance analyzer, 99
Save as text
option in admin menu, 110
Search field
in subroutines and files view, 147
loop list, 45, 85
editable text field, 127
searching source code, 15
sed, 46
Selected loop field
in parallelization control view, 138
selecting a loop, 18, 126
Shared
option in variable list option button, 143
SHARED clause, 43
Parallelization control view and, 141
Shared field
in parallelization control view, 141
Show all loop types
option in show loop types option button, 128
show loop types option button, 12
in loop display controls, 128
Show all loop types option, 128
Show modified loops option, 128
Show omp directives option, 128
Show parallelized loops option, 128
Show serial loops option, 128
Show unparallelizable loops option, 128
Show modified loops
option in show loop types option button, 128
Show omp directives
option in show loop types option button, 128
Show parallelized loops
option in show loop types option button, 128
Show serial loops
option in show loop types option button, 128
Show unparallelizable loops
option in show loop types option button, 128
single-process section
OpenMP, 55, 92
PCF, 157
software
required to install, 1
Sort by perf. cost
option in sort option button, 127
Sort in source order

option in sort option button, 127
sort option button
in loop display controls, 127
Sort by perf. cost option, 127
Sort in source order option, 127
sorting
by performance cost, 103, 125
Source control button, 15
in loop display controls, 129
Source view, 148
source files
examining modified, 50, 89
manipulating fileset, 116
modifying, 41, 82
undoing changes, 119
updating, 46, 49, 89, 117
viewing, 15
Source view, 23
opening, 129
Source control button and, 148
Static
C$MP_SCHEDTYPE mode, 133
option in mp scheduling option button, 133
Static analyzer
option in launch tool submenu, 113
Status line, 124
Subroutine and files view, 13
keyboard shortcut, 13
subroutine call
parallelizable, 34
unparallelizable, 33
Subroutine loop list field, 126
Subroutines and files view
filtering text field and, 129
in views menu, 146
option in views menu, 116
Search field, 147
Synchronization construct option button
in parallelization control view, 138

T

Tester
option in launch tool submenu, 113
transformed
source files, viewing, 16
Transformed loops view
in views menu, 144
option in views menu, 116
Transformed source
window, opening, 129
Transformed source control button, 16
in loop display controls, 129
in parallel analyzer view - transformed source, 148
turned-down corner of mp chunk size field, 134

U

Undo all changes
option in operations menu, 120
Undo changes to loop
option in operations menu, 120
unstructured control flow, 32
Update all files
option in update menu, 118
Update menu
Force a build to start option, 118
in parallel analyzer view, 117
Run editor after update option, 118
Run gdiff after update option, 118
Update all files option, 118
Update selected file option, 118
Update selected file
option in update menu, 118
updating files, 46, 47

V

Variable
  loop list, 10
variable list option buttons
  C$OMP do… option and, 143
  C$OMP parallel do… option and, 143
  Default option, 143
  First-local option, 143
  in parallelization control view, 142
  Last-local option, 143
  Local option, 143
  Reduction option, 143
  Region-default option, 143
  Shared option, 143
variable list storage labels
  Automatic, 143
  Common, 143
  Reference, 144
Variable loop list field, 126
Variables referenced section
  in parallelization control view, 138
versions command, 1
vi, 48
Views menu
  in parallel analyzer view, 116
  options menus
    Admin menu, 136
    Help menu, 136
  Parallelization control view option, 116
  PFA analysis parameters view option, 116
  Subroutines and files view option, 116
  Transformed loops view option, 116

W

windows, closing all, project submenu, exit option, 115
WorkShop, 99
  debugger, launching, 112
WorkShop build manager, 47, 49, 89

X

X resources, 5
.Xdefaults, 118
xwsh, 48