Developer Magic™: WorkShop Pro MPF User's Guide

Document Number 007-2603-002

CONTRIBUTORS

Written by Marty Itzkowitz, Robert M. Reimann, Carol Geary and
Douglas B. O'Morain
Revised by Leif Wennerberg
Illustrated by Douglas B. O'Morain, Carol Geary, and Leif Wennerberg
Production by Kirsten Pekarek
Engineering contributions by Marty Itzkowitz, and Zaineb Asaf

Developer Magic™: WorkShop Pro MPF User's Guide
Document Number 007-2603-002

Contents

List of Figures ix

List of Tables xi

About This Guide xiii
What This Guide Contains xiv
What You Should Know Before Reading This Guide xiv
Recommended Reading xiv
Conventions xv

1. Getting Started With the Parallel Analyzer View 1
Setting Up Your System 1
Required Software 1
Verifying Currently Installed Software 2
Installing WorkShop Pro MPF 2
Running the Parallel Analyzer View: General Features 2
Compiling a Program for Parallel Analyzer View 3
PCF Directive Support 3
Reading Files With the Parallel Analyzer View 4
Tutorials 4

2. Examining Loops, Modifying Source Code 5
Setting Up the dummy.f Sample Session 6
Restarting the Tutorial 6
Compiling the Sample Code 7
Starting the Parallel Analyzer View 7

Using the Loop List Display 10
Loop List Information Fields 10
Loop List Icons: The Icon Legend 10
Resizing the Loop List Display 12
Searching the Loop List Display 12
Sorting and Filtering the Loop List 12
Sorting the Loop List 13
Filtering the Loop List 13
Filtering the Loop List by Parallelization State 14
Filtering the Loop List by Loop Origin: 15
Filtering by Loop Origin: Details for Sorting by Subroutine 16
Viewing Detailed Information About Code and Loops 17
Viewing Original and Transformed Source 17
Viewing Original Source 17
Viewing Transformed Source 19
Navigating the Loop List 20
Selecting a Loop for Analysis 21
Using the Loop Information Display 23
Loop Parallelization Controls 23
Additional Loop Information and Controls 24
Using the Transformed Loops View 25
Transformed Loops View Description 25
Selecting Transformed Loops 26
Examples of Simple Loops 28
Simple Parallel Loop 28
Serial Loop 28
Explicitly Parallelized Loop 29
Fused Loops 30
Loop That Is Optimized Away 30

Examples of Loops With Obstacles to Parallelization 31
Carried Data Dependence 35
Unparallelizable Carried Data Dependence 36
Parallelizable Carried Data Dependence 37
Multi-line Data Dependence 38
Reductions 38
Input-Output Operations 39
Unstructured Control Flows 39
Subroutine Calls 39
Unparallelizable Loop With a Subroutine Call 40
Parallelizable Loop With a Subroutine Call 40
Permutation Vectors 40
Unparallelizable Loop With a Permutation Vector 40
Parallelizable Loop With a Permutation Vector 40
Examples of Nested Loops 41
Doubly Nested Loop 41
Interchanged Doubly Nested Loop 41
Triply Nested Loop With an Interchange 42
Modifying Source Files and Compiling 42
Requesting Changes 42
Directives and Assertions 43
Adding a C$DOACROSS Directive and Clauses 43
Adding a New Assertion or Directive With the Operations Menu 46
Deleting an Assertion or a Directive 48
Applying Requested Changes 48
Viewing Changes With gdiff 49
Modifying the Source File Further 49
Updating the Source File 50
Examining the Modified Source File 51
New Assertion 51
Deleted Assertion 51

Examples With PCF Directives 52
Explicitly Parallelized Loops: C$PAR PDO 52
Loops With Barriers: C$PAR BARRIER 54
Critical Sections: C$PAR CRITICAL SECTION 55
Single-Process Sections: C$PAR SINGLE PROCESS 55
Parallel Sections: C$PAR PSECTIONS 55
Examples With Data Distribution Directives 56
Distributed Arrays: C$DISTRIBUTE 56
Distributed and Reshaped Arrays: C$DISTRIBUTE_RESHAPE 57
Prefetching Data From Cache: C*$*PREFETCH_REF 58
Exiting From the dummy.f Sample Session 58

3. Using WorkShop With Parallel Analyzer View 59
Setting Up the linpackd Sample Session 59
Starting the Parallel Analyzer View 60
Starting the Performance Analyzer 60
Using the Parallel Analyzer With Performance Data 62
Effect of Performance Data on the Source View 63
Sorting the Loop List by Performance Cost 64
Exiting From the linpackd Sample Session 65

4. Parallel Analyzer View Reference 67
Main View Menu Bar 68
Admin Menu 69
Launch Tool Submenu 71
Project Submenu 73
Views Menu 75
Fileset Menu 76
Operations Menu 77
Update Menu 80
Help Menu 82
Keyboard Shortcuts 83

Loop List 84
Resizing the Loop List 84
Status and Performance Experiment Lines 84
Loop List Display 85
Loop List Search Field 86
Sort Option Menu 87
Show Loop Types Option Menu 87
Filtering Option Menu 88
Loop List Buttons 89
Loop Information Display 90
Highlighting Buttons 90
Loop Parallelization Controls in the Loop Information Display 91
Loop Parallelization Status Option Menu 92
MP Scheduling Option Menu: Directives for All Loops 94
MP Scheduling Chunk Size Field 95
Obstacles to Parallelization Display 95
Assertions and Directives Display 96
Compiler Messages 97
Other Views 97
Parallelization Control View 97
Common Features of the Parallelization Control View 98
Adding C$DOACROSS... or C$PAR PDO... Clauses 99
Parallelization Control View MP Scheduling Option Menu: Clauses for One Loop 103
Parallelization Control View Variable List: Option Menus 103
Parallelization Control View Variable List: Storage Labeling 104
Transformed Loops View 105
PFA Analysis Parameters View 106
Subroutines and Files View 108
Source View and Transformed Source Windows 109
Icon Legend 110

Index 113

List of Figures

Figure 2-1 Parallel Analyzer View Main Window 9
Figure 2-2 The “Icon Legend...” Window 11
Figure 2-3 Loop Display Controls 12
Figure 2-4 Parallelization-State Filter Options 14
Figure 2-5 Filter by Subroutine Option Menu and Text Field 15
Figure 2-6 Subroutines and Files View 16
Figure 2-7 Filter Option Menu 16
Figure 2-8 Source View 18
Figure 2-9 Transformed Source Window 19
Figure 2-10 Global Effects of Selecting a Loop (Olid 27) 22
Figure 2-11 Loop Information Display Without Performance Data 23
Figure 2-12 Highlighting Button 24
Figure 2-13 Transformed Loops View for Loop Olid 2 25
Figure 2-14 Transformed Loops in Source Windows 27
Figure 2-15 Explicitly Parallelized Loop 29
Figure 2-16 Source View of C$DOACROSS Directive 30
Figure 2-17 Obstacle to Parallelization 36
Figure 2-18 Parallelizable Data Dependence 37
Figure 2-19 Highlighting on Multiple Lines 38
Figure 2-20 Requesting a C$DOACROSS for Olid 22 44
Figure 2-21 Parallelization Control View for Loop Olid 22 After Choosing “C$DOACROSS...” 45
Figure 2-22 Effect of Changes on the Loop List Display 46
Figure 2-23 Adding an Assertion 47
Figure 2-24 Deleting an Assertion 48
Figure 2-25 Setting the Run Editor Toggle 49
Figure 2-26 Update All Files 50
Figure 2-27 Explicitly Parallelized Loops With C$PAR PDO 53
Figure 2-28 Loops With Barrier Synchronization 54

Figure 3-1 Starting the Performance Analyzer 61
Figure 3-2 Performance Data — Parallel Analyzer View 62
Figure 3-3 Source View for Performance Experiment 63
Figure 3-4 Sort by Performance Cost 64
Figure 3-5 Loop Information Display With Performance Data 65
Figure 4-1 Parallel Analyzer View Menu Bar and Pulldown Menus 68
Figure 4-2 Main View Admin Menu 69
Figure 4-3 Directory and File Browser Window 70
Figure 4-4 Launch Tool Submenu 71
Figure 4-5 Project Submenu Commands 74
Figure 4-6 Views Menu 75
Figure 4-7 Fileset Menu 76
Figure 4-8 Operations Menu and Submenus 79
Figure 4-9 Update Menu 80
Figure 4-10 Help Menu 82
Figure 4-11 Loop List Display and Controls 84
Figure 4-12 Column Headings for the Loop List Display 85
Figure 4-13 Sort Option Menu 87
Figure 4-14 Show Loop Types Menu 87
Figure 4-15 Loop Filtering Option Menu 88
Figure 4-16 Loop Information Display 90
Figure 4-17 Parallelization Controls 91
Figure 4-18 MP Chunk Size Input Field Changed 95
Figure 4-19 Obstacles Information Block 96
Figure 4-20 Assertion Information Block and Options for n32 and n64 Compilation 96
Figure 4-21 Parallelization Control View Without Applicable Directive 98
Figure 4-22 C$DOACROSS Parallelization Control View 100
Figure 4-23 C$PAR PDO Parallelization Control View 101
Figure 4-24 Transformed Loops View 105
Figure 4-25 PFA Analysis Parameters View 107
Figure 4-26 Subroutines and Files View 108
Figure 4-27 Original and Transformed Loop Source Windows 110
Figure 4-28 Parallelization Icon Legend 111

List of Tables

Table 2-1 Obstacles to Parallelization Messages 31

Table 4-1 Assertions and Directives in the Operations Menu 78

Table 4-2 Keyboard Shortcuts 83

Table 4-3 Assertions Accessed From the Loop Parallelization Controls 93

Table 4-4 Directives Accessed From the Loop Parallelization Controls 93

About This Guide

Developer Magic™: WorkShop Pro MPF is a companion product to the Developer Magic: WorkShop suite of computer-aided software engineering tools that use a graphical interface to help you construct, analyze, and debug software applications.

The WorkShop Pro MPF product helps you better understand the structure and parallelization of a multiprocessing Fortran 77 application by providing an interactive, visual comparison of the original source with transformed, parallelized code.

The main program of WorkShop Pro MPF is the Parallel Analyzer View, *cvpav*, which reads analysis files generated by the MIPSpro™ Auto-Parallel Fortran77 compiler. It displays editable parameters for each DO loop found in the source files— parameters that are easily customized and explored with the help of the Parallel Analyzer View's graphical interface.

The Parallel Analyzer View is integrated with WorkShop 2.0 (and later versions), allowing you to examine a program's loops in conjunction with a performance experiment on either a uni- or multiprocessor run. When run in this mode, the source displays are annotated with line-level performance data, and the list of loops may be sorted in order of performance cost, allowing you to concentrate your attention on the most compute-intensive loops.

What This Guide Contains

This guide presents the WorkShop Pro MPF Parallel Analyzer View from a task-oriented perspective. This guide includes the following chapters:

- Chapter 1, “Getting Started With the Parallel Analyzer View,” tells you how to install the WorkShopProMPF software and run the Parallel Analyzer View.
- Chapter 2, “Examining Loops, Modifying Source Code,” provides a tutorial session that steps you through the Parallel Analyzer’s basic features using sample Fortran code.
- Chapter 3, “Using WorkShop With Parallel Analyzer View,” provides a tutorial sessions that analyzes the performance of linpackd, a matrix manipulating benchmark program.
- Chapter 4, “Parallel Analyzer View Reference,” describes in detail the graphical user interface of the Parallel Analyzer View.

An index completes this guide.

What You Should Know Before Reading This Guide

This guide assumes that you are familiar with principles of Fortran programming and multiprocessing.

Recommended Reading

These books provide essential background for understanding the MIPSpro parallelization option; they provide details about parallel programming and the directives and assertions you can manipulate with the Parallel Analyzer View:

- *MIPSpro Compiling and Performance Tuning Guide* (part no. 007-2360-007, http://techpubs.engr.sgi.com/lib/makepage.cgi?007-2360-006)
- *MIPSpro Fortran 77 Programmer’s Guide* (part no. 007-2361-005, http://techpubs.engr.sgi.com/lib/makepage.cgi?007-2361-002)
- *MIPSpro Automatic Parallelizer Programmer’s Guide* (part no. 007-3572-001, http://techpubs.engr.sgi.com/lib/makepage.cgi?007-3572-001)

The following manuals, available from Silicon Graphics™, may provide useful supplementary information and are sometimes referenced in this manual:

- *Developer Magic: Debugger User's Guide* (part no. 007-2579-003, http://techpubs.engr.sgi.com/lib/makepage.cgi?007-2579-003)
- *Developer Magic: Performance Analyzer User's Guide* (part no. 007-2581-003, http://techpubs.engr.sgi.com/lib/makepage.cgi?007-2581-003)
- *Developer Magic: ProDev WorkShop Overview* (part no. 007-2582-003, http://techpubs.engr.sgi.com/lib/makepage.cgi?007-2582-003)
- *IRIX Admin: Software Installation and Licensing (*part no. 007-1364-060, http://techpubs.engr.sgi.com/lib/makepage.cgi?007-1364-060)
- *SpeedShop User's Guide* (part no. 007-3311-001, http://techpubs.engr.sgi.com/lib/makepage.cgi?007-3311-001)

The following book is also recommended:

- *Practical Parallel Programming*, by B.E. Bauer, Academic Press, 1992

Conventions

These are the typographical conventions used in this guide:

- **Bold**—Option flags, data types, functions, routines, directives, and keywords
- *Italics*—Filenames, button names, variables, arrays, and IRIX commands
- Regular—Menu and window names
- "Quoted"—Menu choices
- `Fixed-width`—Code examples and screen display
- **`Bold fixed-width`**—User input and nonprinting keys such as **`Ctrl+u`**

Chapter 1

Getting Started With the Parallel Analyzer View

This chapter helps you get the WorkShop Pro MPF Parallel Analyzer View running on your system. It contains the following sections:

- "Setting Up Your System" on page 1
- "Running the Parallel Analyzer View: General Features" on page 2
- "Tutorials" on page 4

Setting Up Your System

To install the WorkShopProMPF software, you should have at least 16 MB of memory; 32 MB improves overall performance.

Required Software

WorkShopProMPF requires that you have the following software installed:

- IRIX™ system software version 6.2 or greater
- MIPSpro Auto-Parallel Fortran77 7.2 compiler
- ToolTalk 1.1 or greater
- WorkShop 2.0 or later Execution Environment
- Developer Magic 1.1

Verifying Currently Installed Software

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

```
% versions
```

If the items mentioned in this section are not installed, consult your sales representative or (in the US) call the Silicon Graphics Technical Assistance Center at 1-(800)-800-4SGI. To order additional memory, consult your sales representative or call 1-(800)-800-SGI1.

Installing WorkShop Pro MPF

If you have all the software and memory you need, you can install the Developer Magic: WorkShop Pro MPF software:

- For general instructions about software installation, consult the man pages for *inst* or *swmgr*, and *IRIX Admin: Software Installation and Licensing*.
- See also *Developer Magic: WorkShop Pro MPF Release Notes* for specific installation instructions.

The executable is *cvpav*, which is installed in */usr/sbin*.

Running the Parallel Analyzer View: General Features

The process of using the Parallel Analyzer View involves two steps:

1. Compiling a program with appropriate options
2. Reading the compiled files with Parallel Analyzer View

Compiling a Program for Parallel Analyzer View

Before starting the Parallel Analyzer View on your Fortran source, you need to run the Auto-Parallel Fortran77 compiler with the appropriate options. For the tutorials presented in subsequent chapters, Makefiles are provided. You can adapt these to your specific source or enter the following command:

```
% f77 -pfa 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 these effects (see the *MIPSpro Fortran 77 Programmer's Guide*, *MIPSpro Automatic Parallelizer Programmer's Guide*, and the f77 man page for more information):

`-pfa keep`	Saves an **.anl* file, which has necessary information for the Parallel Analyzer View.
`-O3`	Sets the compiler for aggressive optimization. The optimization focuses on maximizing code quality even if that requires extensive compile time or relaxing language rules.

Note: *cvpav* assumes that the **`-pfa keep`** option was used on each of the Fortran source files named in an executable or fileset. 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" on page 108).

PCF Directive Support

The MIPSpro Auto-Parallel Fortran77 compiler supports PCF directives, unless you are compiling with the **`-O32`** option. If you put PCF directives in your o32 code, they are treated as comments rather than interpreted. For more information on PCF directives, see the following:

- "Examples With PCF Directives" on page 52
- *MIPSpro Fortran 77 Programmer's Guide*, particularly Chapter 5

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*

cvpav reads information from all Fortran source files compiled into the application.

The Parallel Analyzer View has several other command line options, as well as several X resources that you can set. See the man page cvpav(1) for more information.

Note: If you receive a message related to licensing when you start *cvpav*, refer to Chapter 7 in the *WorkShop Pro MPF Release Notes*. To access the notes, enter *grelnotes* at the command line, and choose WorkShopMPF from the Products pulldown menu.

Tutorials

For a more detailed introduction to the Parallel Analyzer View, you can follow one of tutorials provided with the product, discussed in the following chapters:

- Chapter 2, “Examining Loops, Modifying Source Code”
- Chapter 3, “Using WorkShop With Parallel Analyzer View”

Chapter 2

Examining Loops, Modifying Source Code

This chapter presents an interactive sample session with the Parallel Analyzer View. The session demonstrates basic features of the Parallel Analyzer View and illustrates aspects of parallelization and of the MIPSpro Auto-Parallel Fortran77 compiler. Specifically, the sample session analyzes dummy code to illustrate the following:

- Displaying code and basic loop information; these topics are discussed in the first sections of this chapter:
  - “Setting Up the dummy.f Sample Session” on page 6
  - “Starting the Parallel Analyzer View” on page 7
  - “Using the Loop List Display” on page 10
  - “Sorting and Filtering the Loop List” on page 12
  - “Viewing Detailed Information About Code and Loops” on page 17
- Examining specific loops, applying directives and assertions, and modifying and recompiling; these topics are discussed in the later sections of the chapter:
  - “Examples of Simple Loops” on page 28
  - “Examples of Loops With Obstacles to Parallelization” on page 31
  - “Examples of Nested Loops” on page 41
  - “Modifying Source Files and Compiling” on page 42
  - “Examples With PCF Directives” on page 52
  - “Examples With Data Distribution Directives” on page 56
  - “Exiting From the dummy.f Sample Session” on page 58

The topics are introduced in this chapter as you go 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 4, “Parallel Analyzer View Reference.”

Setting Up the dummy.f Sample Session

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

- */usr/demos/WorkShopMPF* is the demonstration directory
- *WorkShopMPF.sw.demos* must be installed

The sample session discussed in this chapter uses the following source files in the directory */usr/demos/WorkShopMPF* / *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/WorkShopMPF* / *tutorial* also includes a *Makefile* to compile the source files.

Restarting the Tutorial

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

- Quit the Parallel Analyzer View.
- Clean up the tutorial directory; enter the following command:

  ```
  % make clean
  ```

This removes all of the generated files; you can begin again with the *make* command.

Compiling the Sample Code

Prepare for the session by opening a shell window and entering *make* in the */usr/demos/WorkShopMPF/tutorial* directory:

```
% cd /usr/demos/WorkShopMPF/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 any 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

For more information about these files, see the *MIPSpro Automatic Parallelizer Programmer's Guide*.

Starting the Parallel Analyzer View

Once you have the appropriate files from 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 (see Figure 2-1):

```
% cvpav -f dummy.f
```

Note: If you receive a message related to licensing, refer to the *WorkShop Pro MPF Release Notes*.

The main window contains the following (from the top of the window to the bottom, see Figure 2-1):

- X window menu
- status and performance experiment information
- loop list display
- loop list search field, Search:
- option menus; these are the default option values for selecting loops for display:
  - “Sort in Source Order”
  - “Show All Loop Types”
  - “No Filtering”
- control buttons for displaying loop code:
  - Source
  - Transformed Source
- loop list navigation buttons:
  - Next Loop
  - Previous Loop
- loop information display

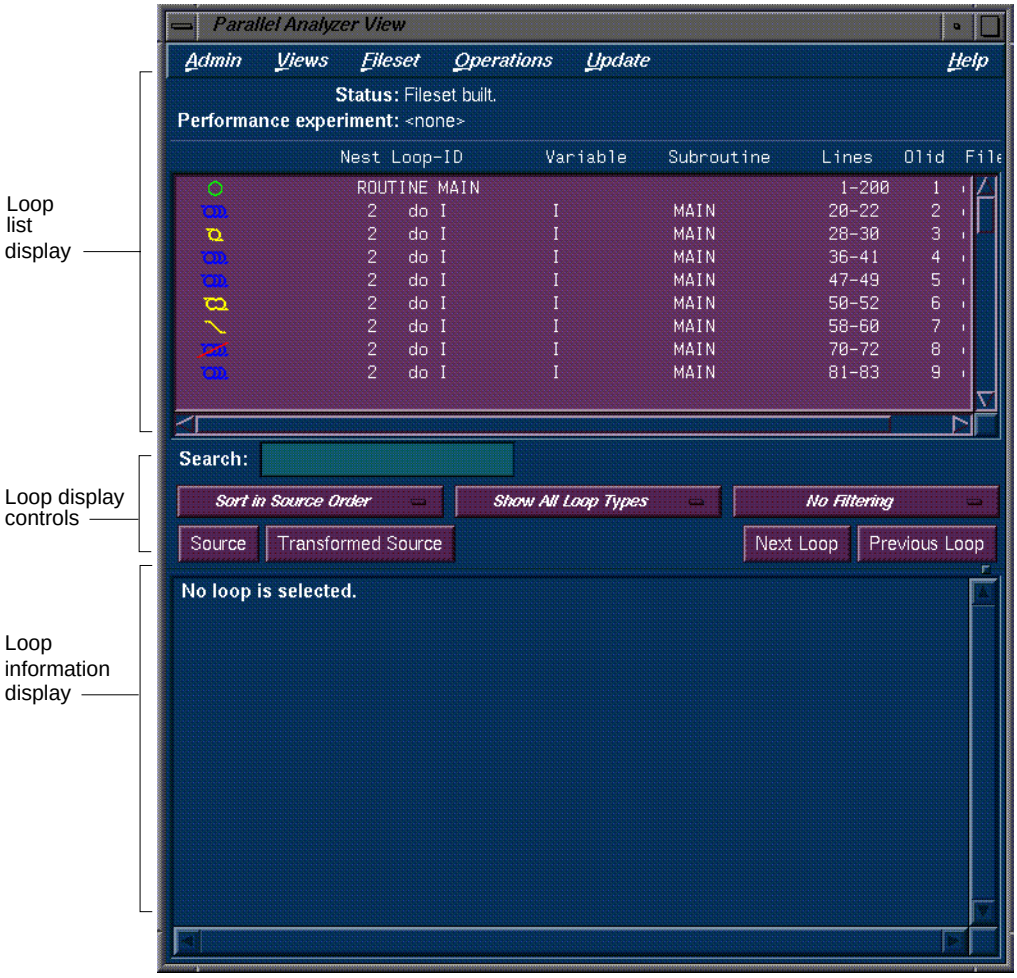

Figure 2-1 Parallel Analyzer View Main Window

Using the Loop List Display

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

Loop List Information Fields

Each loop list display entry contains the following fields:

An Icon	the status of the routine or loop (see "Loop List Icons: The Icon Legend" on page 10)
Nest	the nesting level for the loop
Loop-ID	the Fortran description of the loop
Variable	the loop index variable
Subroutine, Lines, File	where the loop is located in the source code
Olid	the original loop ID, created by the compiler; an internal identifier for the loop (refer to this number when reporting bugs)

Loop List Icons: The Icon Legend

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

- whether the line refers to a subroutine
- parallelization status of the loop
- PCF control structures

To understand the meaning of the various icons, pull down the Admin menu and choose "Icon Legend..." (see Figure 2-2).

To see examples of the various icons, scroll through the list of loops. When you are done, close the icon legend dialog box by clicking the *Close* button in the lower right of the dialog box.

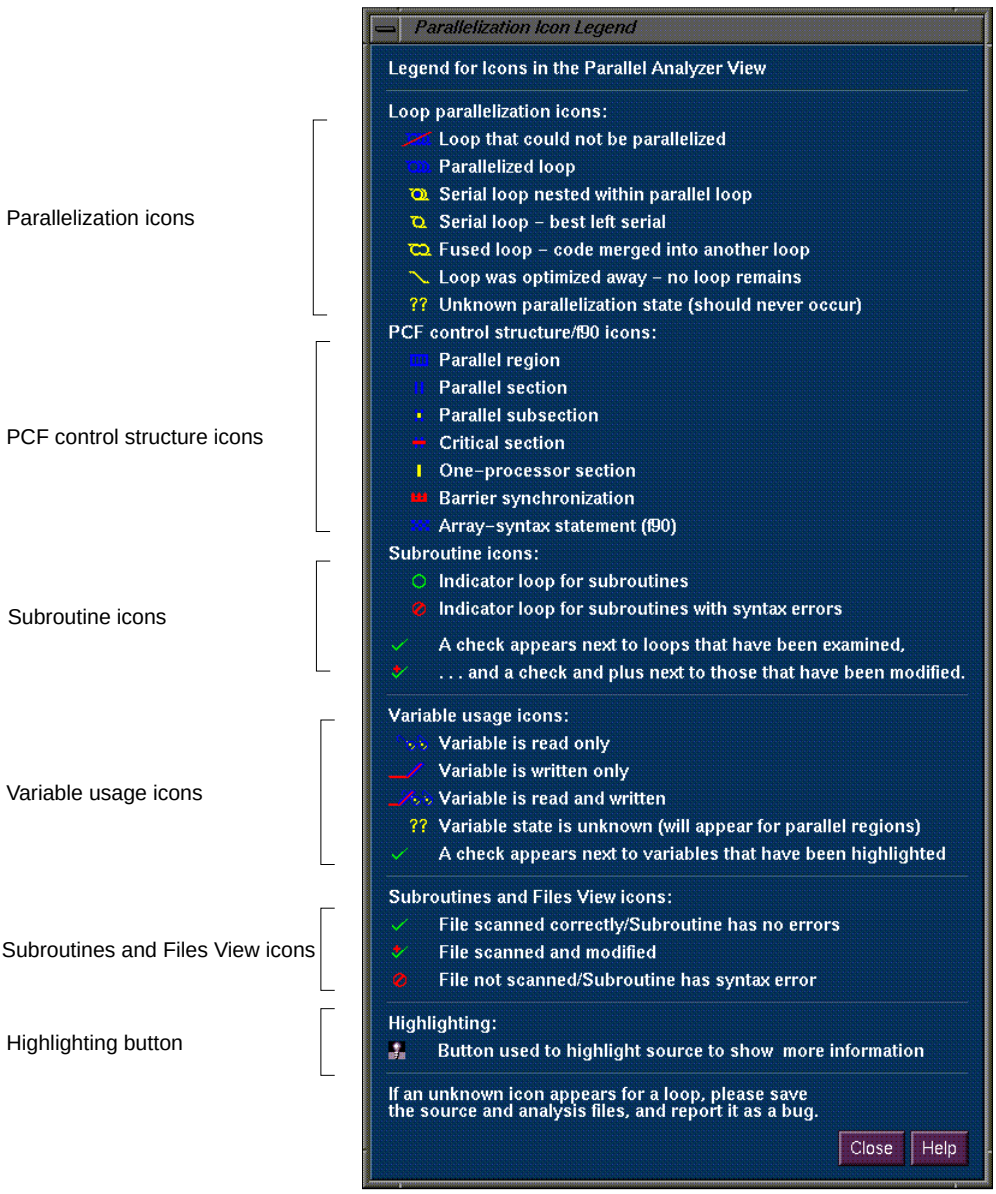

Figure 2-2 The “Icon Legend...” Window

Resizing the Loop List Display

To resize the loop list display and provide more room in the main window for loop information, use the adjustment button: a small square below the button that says “Previous Loop.” The adjustment button is 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.

Searching the Loop List Display

The loop list Search field allows you to find occurrences of any character in the loop list. For example, you can search for subroutine names; a phrase (such as “parallel region”); or Olid numbers (see Figure 2-3).

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 begins the discussion of loop list display controls that allow you to sort and filter the loop list, and so focus your attention on particular pieces of your code. The options menus are located in the main window, below the loop list display (see Figure 2-1). Figure 2-3 shows the display controls.

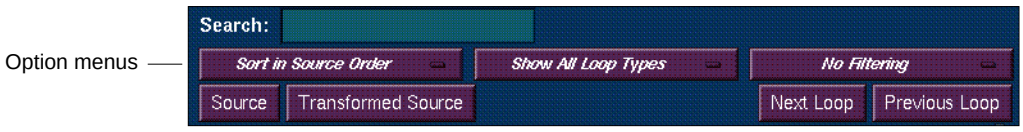

Figure 2-3 Loop Display Controls

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 a performance experiment on the program using the WorkShop Performance Analyzer). You normally control sorting with the left-most option menu, 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 to **Olid 18** and **19** (or use the Search field).

When loops are sorted by performance cost, by using “Sort by Perf. Cost” menu option, the list is not indented. The sorting option is grayed out in the example because no performance tool is 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 two features:

- parallelization status
- loop origin

The filter parameters are controlled by the two option popup menus to the right of the sort option menu button.

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 option popup menu centered below the loop list; the default setting reads “Show All Loop Types,” as in Figure 2-4.

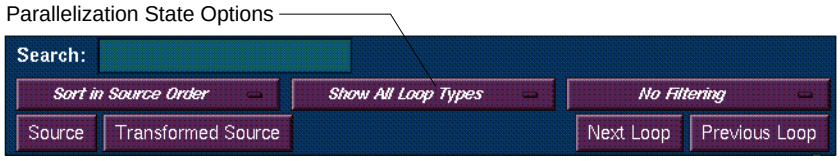

Figure 2-4 Parallelization-State Filter Options

You can select according to the following states of loop parallelization and processing (which are listed when you click on the parallelization state menu option button):

- “Show All Loop Types”: the default
- “Show Unparallelizable Loops”: loops that could not be parallelized, running serially as a result
- “Show Parallelized Loops”
- “Show Serial Loops”: loops that are best run serially
- “Show Modified Loops”: loops for which modifications have been requested

The second, third, and fourth categories correspond to parallelization icons in the “Icon Legend...” window (see Figure 2-2). Requesting modifications to loops is described in “Modifying Source Files and Compiling” on page 42.

To see the effects of the first three options, choose them in turn by clicking on the popup menu button and then clicking on each option. If you choose the “Show Modified Loops” option, a message appears that no loops meet the filter criterion, because you have not requested any modifications yet.

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. These are the basic steps:

1. Open a list of subroutines and files from which to select; to display the list, choose the “Subroutines and Files View” option from the Views pulldown menu.
2. Choose the filter criterion from the right-most option menu in the Parallel Analyzer View window; this filter option menu initially reads “No Filtering.” You can filter according to source file or subroutine.
3. To place filtering information in the text box that appears above the option menu (see Figure 2-5), you can do one of the following:
   - Enter the file or routine name in the box.
   - Choose the file or subroutine of interest in the Subroutines and Files View.

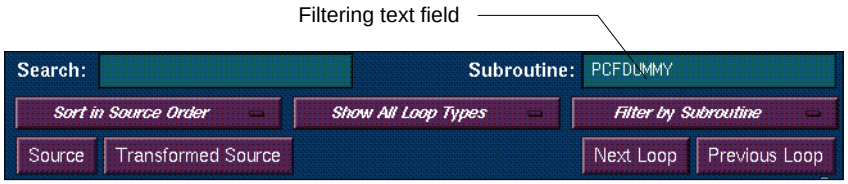

Figure 2-5 Filter by Subroutine Option Menu and Text Field

Filtering by Loop Origin: Details for Sorting by Subroutine

The following procedure describes filtering the loop list by subroutine.

1. Open the Subroutines and Files View by pulling down the Views menu and choosing “Subroutines and Files View”; the window opens and lists the subroutines and files in the fileset (See Figure 2-6.)

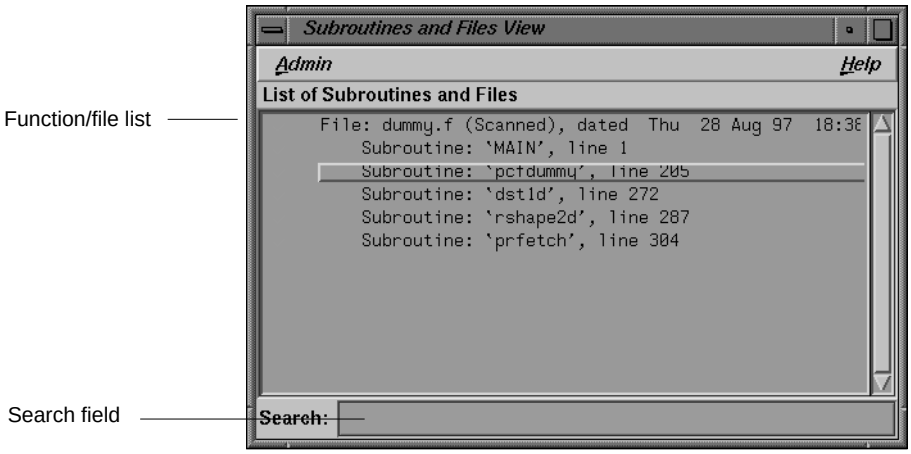

Figure 2-6 Subroutines and Files View

2. From the filter option popup menu, choose “Filter by Subroutine” (Figure 2-7).

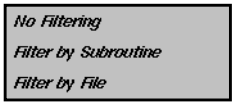

Figure 2-7 Filter Option Menu

3. Double-click the line for the routine **pcfdummy** in the list of the Subroutines and Files View window. The name appears in the filtering text field labeled Subroutine: (see Figure 2-5) and the loop list is re-created according to the filter criteria.

You can also try choosing “Filter by File” from the filter option menu, but for this single-file example, this is not too useful.

When you are done, display all of the loops in the sample source file again by choosing “No Filtering” from the option menu.

There is no further need for the Subroutines and Files View, so close it by pulling down the Admin menu and choosing “Close.”

Viewing Detailed Information About Code and Loops

This section describes how to examine the following:

- source
- transformed source
- details of loop information that is summarized in the loop list

Viewing Original and Transformed Source

The Parallel Analyzer View gives you access to views of both your original Fortran source and a listing that mimics the effect on the source as it is transformed by the Auto-Parallel compiler.

Viewing Original Source

To bring up the Source View window shown in Figure 2-8, click the *Source* button on the lower left corner of the loop list display controls (for example, see Figure 2-5).

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 (see “Selecting a Loop for Analysis” on page 21).

Note that the bracket colors vary as you scroll up and down the listing. These colors correspond to different parallelization icons and indicate at a glance 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 color settings of your monitor.

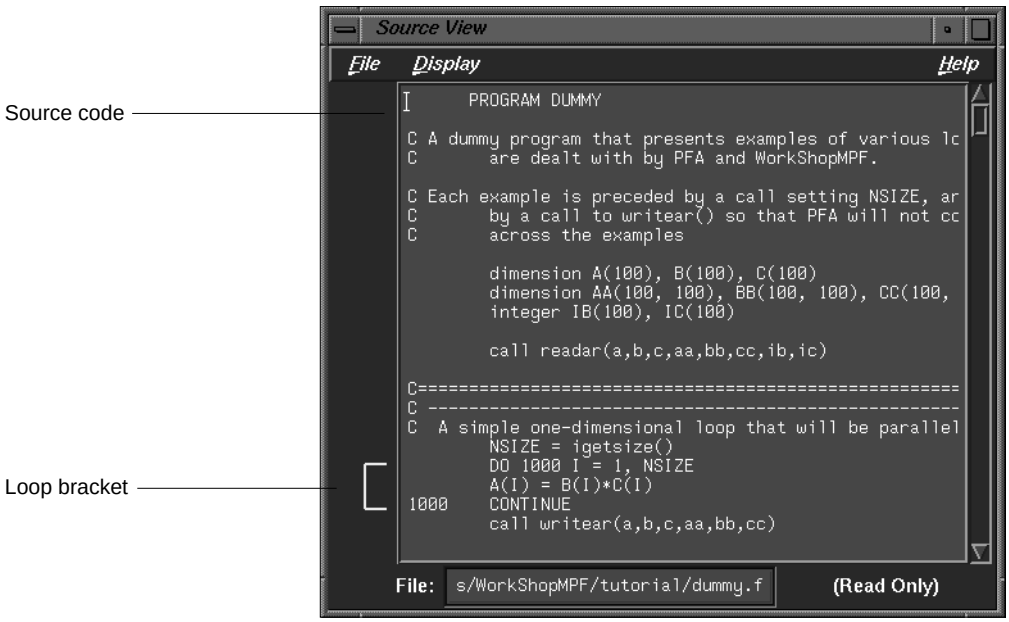

Figure 2-8 Source View

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

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

Thus, you can locate a loop in the source code and click on its colored bracket in the Source View; more information about the loop appears in the loop information display.

Leave the Source View window open and place it conveniently on your screen; 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.

For more information about the Source View window, see “Source View and Transformed Source Windows” on page 109.

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, or it may be combined with others, or optimized away.

Click the *Transformed Source* button in the loop list controls (see Figure 2-3). A window labeled “Parallel Analyzer View — Transformed Source” opens as shown in Figure 2-9.

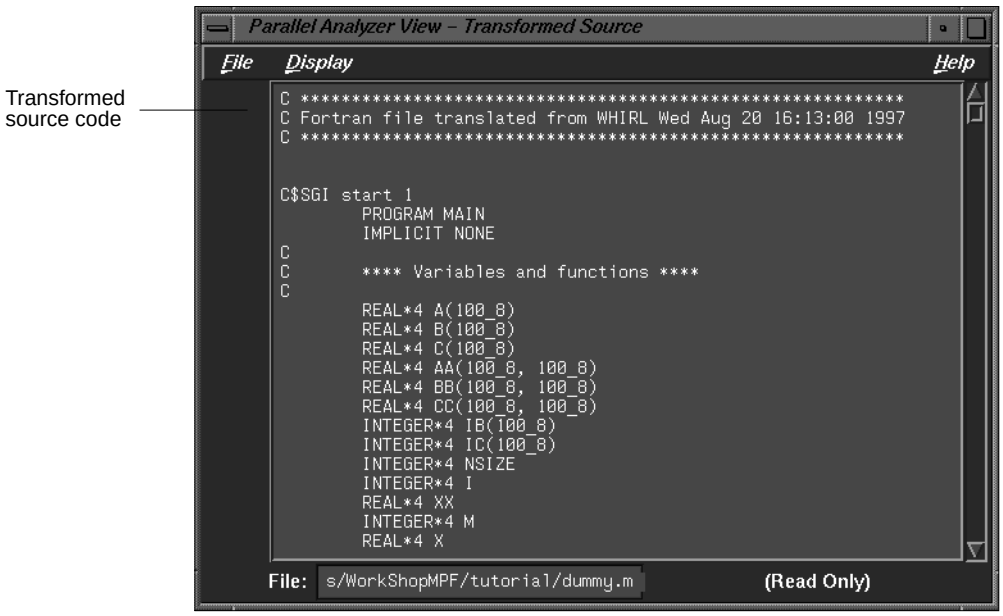

Figure 2-9 Transformed Source Window

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 cannot always distinguish between serial loops and unparallelizable loops; some unparallelizable loops may have the bracket color for a serial loop.

For more information on the Transformed Source window, see “Source View and Transformed Source Windows” on page 109.

Leave the Transformed Source window open and place it conveniently on your screen; subsequent steps in this tutorial refer to the window. You should have three windows open:

- Parallel Analyzer View
- Source View
- Transformed Source

Navigating the Loop List

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

- Scroll, using
  - scroll bar
  - Page Up and Page Down keys (the cursor must be over the loop list)
  - *Next Loop* and *Previous Loop* buttons
- Search for the Olid number using the Search field (see “Searching the Loop List Display” on page 12)

Selecting a Loop for Analysis

To get more information about a loop, first 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 (see Figure 2-10). Not all of the windows in Figure 2-10 are open at this point in the tutorial; you can open the remaining windows from the Views menu:

- Loop information display: information on the selected loop appears in the previously empty display below the loop list (see “Using the Loop Information Display” on page 23).
- Source View: the original source code of the loop appears and is highlighted in the window (see “Viewing Original Source” on page 17).
- 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 (see “Viewing Transformed Source” on page 19).
- Transformed Loops View: shows information about the loop after parallelization (see “Using the Transformed Loops View” on page 25).
- PFA Analysis Parameters View (relevant only for o32 code): shows parameter values for the selected loop (see “PFA Analysis Parameters View” on page 106).

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, indicating that you’ve looked at it.

When you are done, scroll to the top of the loop list in the main view and double-click the line for the first loop, **Olid 2**. Close the Transformed Loops View and the PFA Analysis Parameters View, if you have opened them.

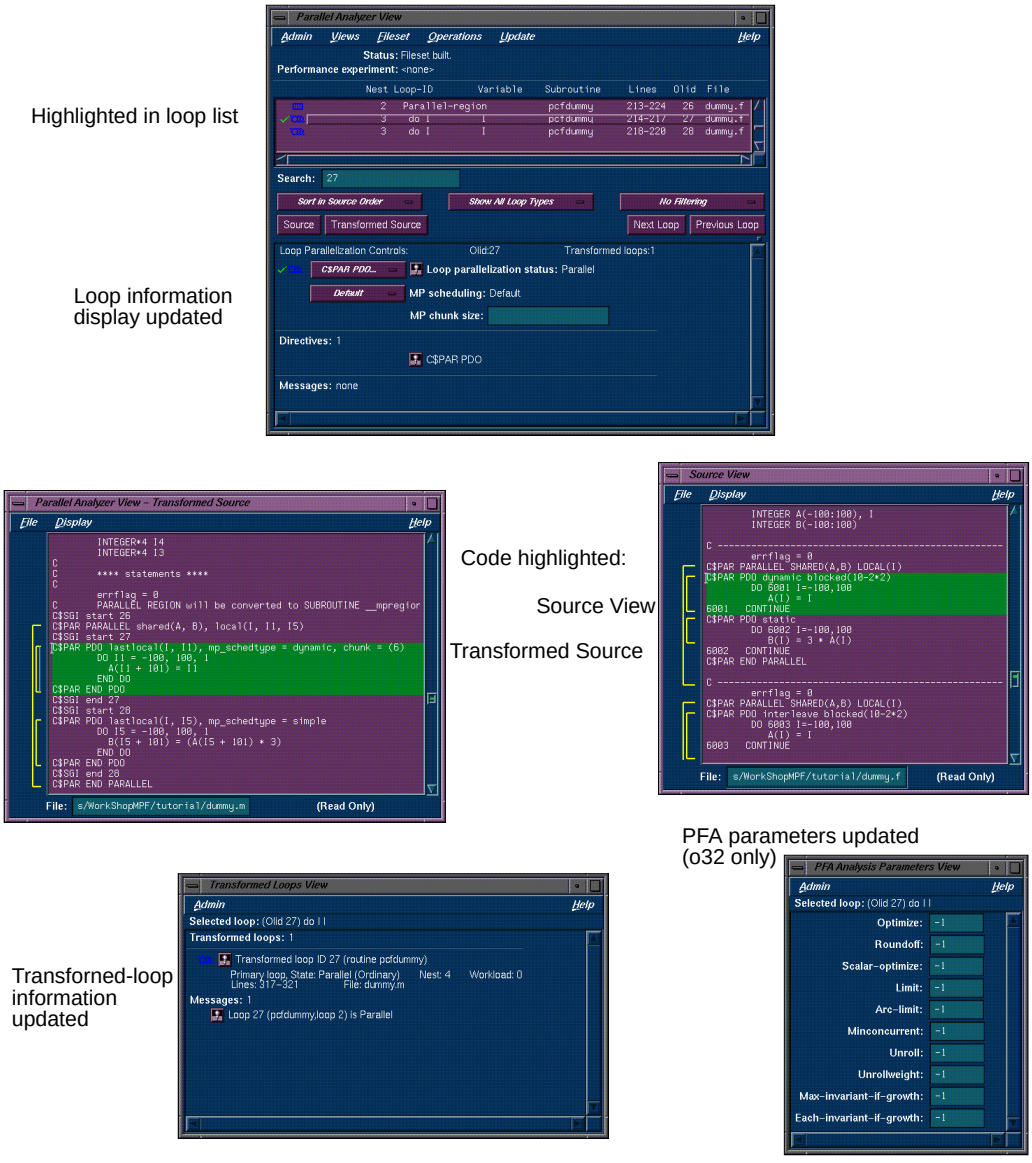

Figure 2-10 Global Effects of Selecting a Loop (Olid 27)

Using the Loop Information Display

The loop information display occupies the portion of the main view below the loop list display controls (see Figure 2-10). Initially, the display shows only “No loop is selected.” After a loop or routine is selected, the display contains detailed information and includes controls for requesting changes to your code (see Figure 2-11).

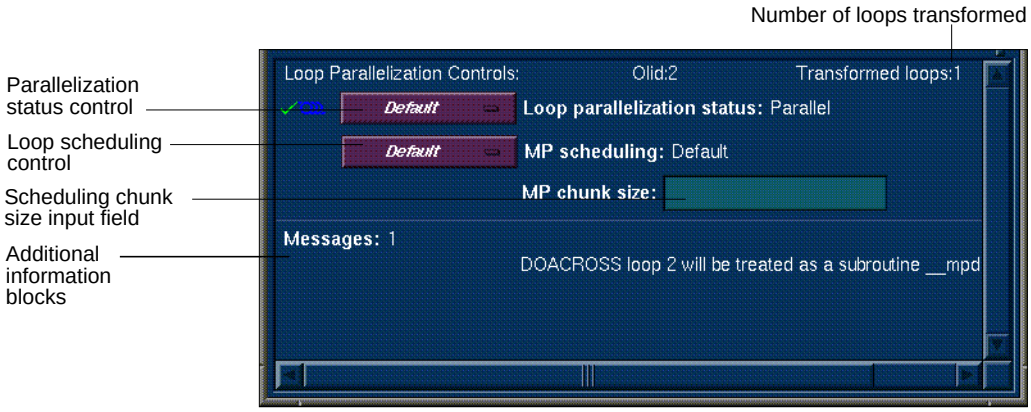

Figure 2-11 Loop Information Display Without Performance Data

Loop Parallelization Controls

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

- On the first line is the loop Olid and to the far right is the number of transformed loops derived from the selected loop.
- The next three lines display two option menu buttons and a text input field:
  - The top menu button controls parallelization status (for more information see “Loop Parallelization Status Option Menu” on page 92)
  - The bottom menu button controls the loop MP scheduling (it is shown for all loops, but is applicable to parallel loops only; for more information see“MP Scheduling Option Menu: Directives for All Loops” on page 94)
  - The text input field is for an expression for the scheduling chunk size (for more information see “MP Scheduling Chunk Size Field” on page 95)

When the Parallel Analyzer View is run with a performance experiment, an additional block appears above the parallelization controls. It gives performance information about the loop (shown in Figure 3-5).

Additional Loop Information and Controls

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

- obstacles to parallelization
- assertions made
- directives applied
- messages
- questions that the compiler asked (o32 only)

Some of these lines may be accompanied by small “light bulb” highlighting buttons (see Figure 2-12). When you click one of these buttons, it highlights the relevant part of the code in the Source View and the Transformed Source window.

Figure 2-12 Highlighting Button

The loop information display shows directives that apply to an entire routine when you select the line with the routine’s name. If you select **Olid 1**, you see that there are no such global directives in **MAIN**. However, if you select **Olid 40**, you see a directive that applies to the subroutine **dist1d** (see “Distributed Arrays: C$DISTRIBUTE” on page 56).

The loop information display shows loop-specific directives when you select a loop. The lines for assertions and directives may have menus accompanying them that provide options; for example, you can delete a directive.

The first loop in the file, **Olid 2**, has no highlighting buttons and one message.

Using the Transformed Loops View

To see detailed information about the transformed loops derived from a particular loop, pull down the Views menu and choose “Transformed Loops View” (see Figure 2-13).

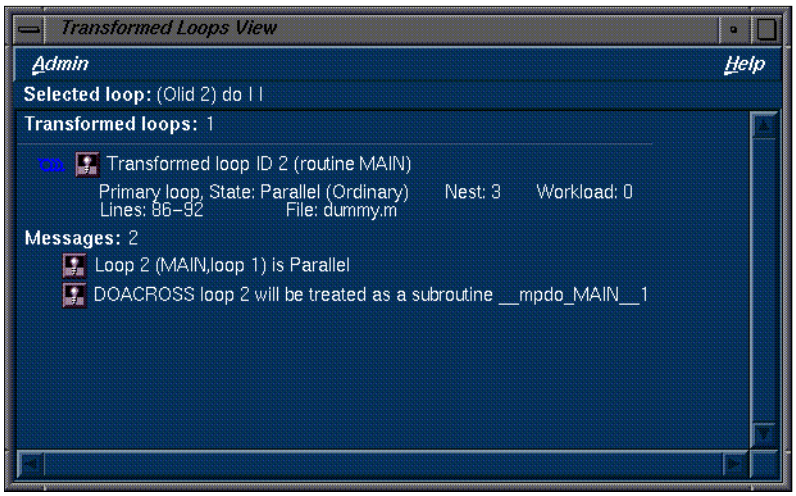

Figure 2-13 Transformed Loops View for Loop Olid 2

Transformed Loops View Description

The Transformed Loops View contains information about the loop(s) 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 highlighting button (if clicked, it highlights the transformed loop in the Transformed Source window and in 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, that is, directly transformed from the selected original loop or transformed from a different original loop, but incorporating some code from the selected original loop
- parallelization state
- whether it is an ordinary loop or interchanged loop
- its nesting level

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 2** the highlighted lines do not (see Figure 2-14). You will see later that for loops with more extensive transformations, the set of highlighted lines is different when you select from the Transformed Loops View (for example, loops **Olid 5** and **Olid 6**; see “Fused Loops” on page 30).

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

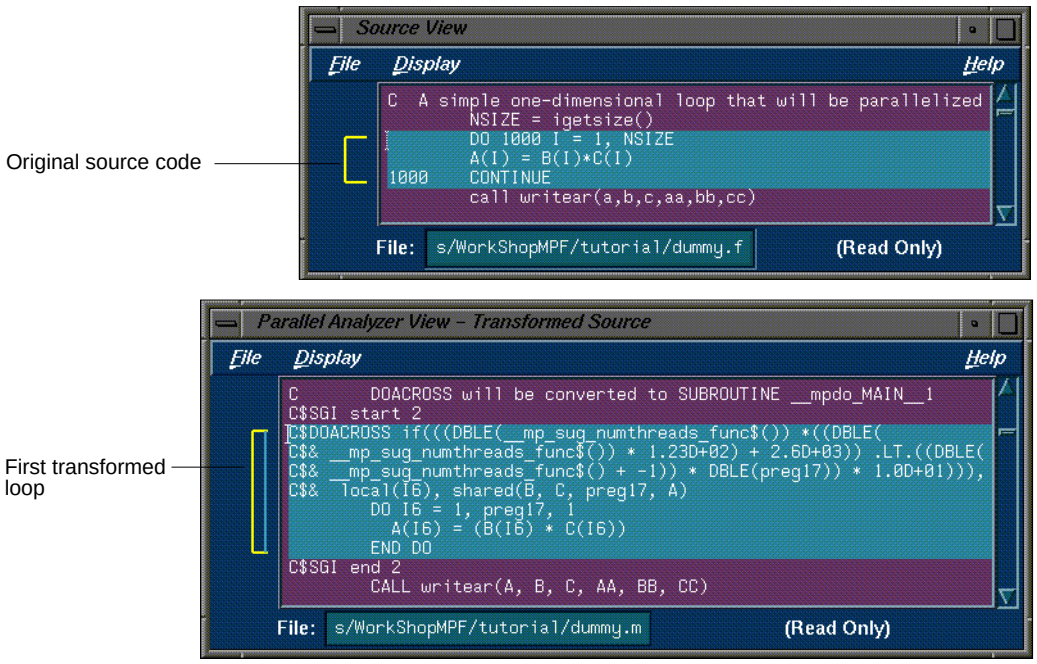

Figure 2-14 Transformed Loops in Source Windows

You may either leave the Transformed Loops View open or close it by pulling down its File menu and choosing “Close.” 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 have familiarized yourself with the basic window features in the Parallel Analyzer View user interface, you can start examining, analyzing, and modifying loops.

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

- "Simple Parallel Loop" on page 28
- "Serial Loop" on page 28
- "Explicitly Parallelized Loop" on page 29
- "Fused Loops" on page 30
- "Loop That Is Optimized Away" on page 30

The next two sections discuss more complicated situations:

- "Examples of Loops With Obstacles to Parallelization" on page 31
- "Examples of Nested Loops" on page 41

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.

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

Serial Loop

Olid 3 is a simple loop with too little work to run in parallel. That is, the compiler determined that the overhead of parallelizing would exceed the benefits; the original loop and the transformed loop are identical.

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

Explicitly Parallelized Loop

Loop **Olid 4** is parallelized because it contains an explicit **C$DOACROSS** directive in the source; the compiler passes the directive through to the transformed source. See Figure 2-15.

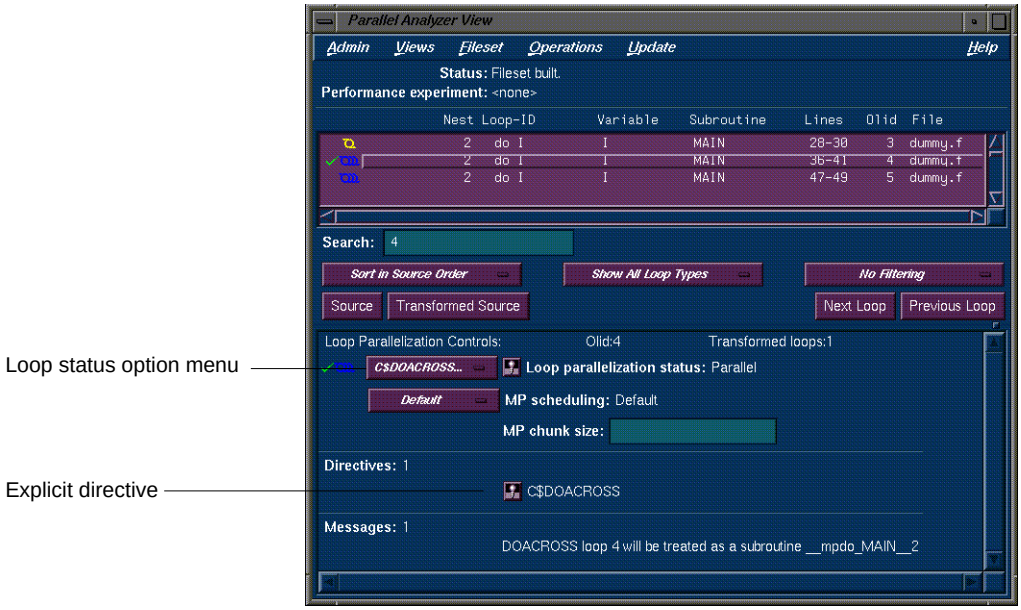

Figure 2-15 Explicitly Parallelized Loop

The loop status option menu is set to “C$DOACROSS...” and it is shown with a highlighting button. Clicking the highlighting button brings up both the Source View, if it is not already opened (Figure 2-16), and the Parallelization Control View, which shows more information about the parallelization directive.

If you clicked on the highlight button, close the Parallelization Control View by pulling down its Admin menu and choosing “Close.” You will come back to the use of this view later (see “Adding a C$DOACROSS Directive and Clauses” on page 43; for more information, see “Parallelization Control View” on page 97). Close the Source View by pulling down its File menu and choosing “Close.”

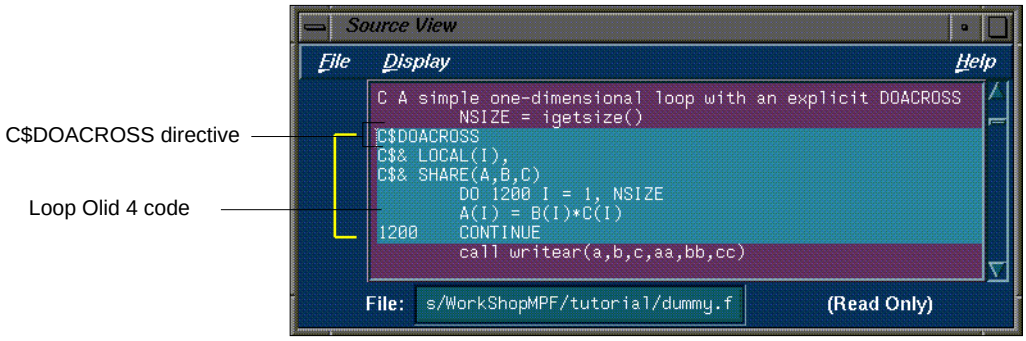

Figure 2-16 Source View of C$DOACROSS Directive

Fused Loops

Loops **Olid 5** and **Olid 6** are simple parallel loops that have similar structure. The compiler combines these 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 while selecting **Olid 5** and **Olid 6**, the identical lines of code are highlighted for each loop.

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

Loop That Is Optimized Away

Loop **Olid 7** is an example of a loop that the compiler could get rid of entirely; the compiler saw that the body of the loop is independent of the loop, so it was moved out, and the loop eliminated. The transformed source is not scrolled and highlighted when you select **Olid 7** because there is no transformed loop derived from the original loop.

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

Examples of Loops With Obstacles to Parallelization

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

- "Carried Data Dependence" on page 35
- "Input-Output Operations" on page 39
- "Unstructured Control Flows" on page 39
- "Subroutine Calls" on page 39
- "Permutation Vectors" on page 40

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

Table 2-1 lists all of the Obstacles to Parallelization messages generated by the compiler. The messages that appear in the loop information display differ slightly from those in the table, because they include specific loop and line information.

Table 2-1 Obstacles to Parallelization Messages

Message	Comments
Loop is preferred serial; insufficient work to justify parallelization	Fundamental obstruction: 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	Fundamental obstruction: 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	Fundamental obstruction: might be parallelizable. There is a `goto` statement or other unstructured control flow in the loop.

Table 2-1 (continued) Obstacles to Parallelization Messages

Message	Comments
Loop was created by peeling the last iteration of a parallel loop	Fundamental obstruction: might be parallelizable. The loop was created by peeling off the final iteration of another loop to make that loop go parallel. The compiler did not try to parallelize this peeled, last iteration.
User directive specifies serial execution for loop	Fundamental obstruction: might be parallelizable. The loop has a directive that indicates it not be parallelized.
Loop can not be parallelized; tiled for reshaped array instead	Fundamental obstruction: 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	Fundamental obstruction: might be parallelizable. The loop is inside a parallel loop, and therefore is not considered to be a candidate for parallelization by the compiler.
Loop is the serial version of parallel loop	Fundamental obstruction: might be parallelizable. The loop is part of the serial version of a parallelized loop. This may occur, for example, 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.
Loop has carried dependence on scalar variable	Data dependence problem: problem with scalars. The loop has a carried dependence on a scalar variable.

Table 2-1 (continued) Obstacles to Parallelization Messages

Message	Comments
Loop scalar variable is aliased precluding auto parallelization	Data dependence problem: problem with scalars. A scalar variable is aliased with another variable, for example, when you have a statement equivalencing a scalar and an array.
Loop can not determine last value for variable	Data dependence problem: 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	Data dependence problem: problem with arrays. The loop carries an array dependence from one array member to another array member.
Call inhibits auto parallelization	Data dependence problem: 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	Data dependence problem: problem with missing dependence information. The compiler does not parallelize loops with input or output statement.

Table 2-1 (continued) Obstacles to Parallelization Messages

Message	Comments

Scalar may not be assigned in final iteration	Data dependence problem: 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`

Table 2-1 (continued) Obstacles to Parallelization Messages

Message	Comments

Array may not be assigned in final iteration	Data dependence problem:
	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`

Carried Data Dependence

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

- "Unparallelizable Carried Data Dependence" on page 36
- "Parallelizable Carried Data Dependence" on page 37
- "Multi-line Data Dependence" on page 38
- "Reductions" on page 38

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. If the loop were non-trivial, that is 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 would have reset the value of A(5); the computation would be unpredictable.

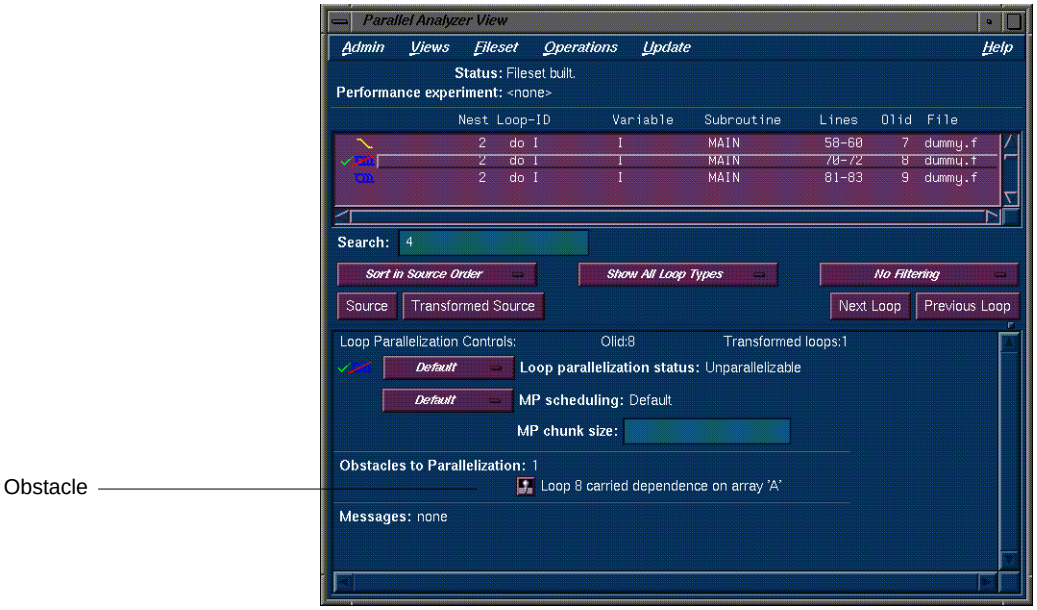

Figure 2-17 Obstacle to Parallelization

In the lower panel of Figure 2-17 (which should match your display) there is a line listing the obstacle to parallelization; click the 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 the *Next Loop* button.

Parallelizable Carried Data Dependence

Loop **Olid 9** has similar structure to loop **Olid 8**. Despite the similarity however, **Olid 9** may be parallelized. Note that the array indices differ by an offset *M* that may be greater than one; if *M* is equal to *NSIZE*, for example, 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. See Figure 2-18. Click the highlighting button to show the assertion in the Source View.

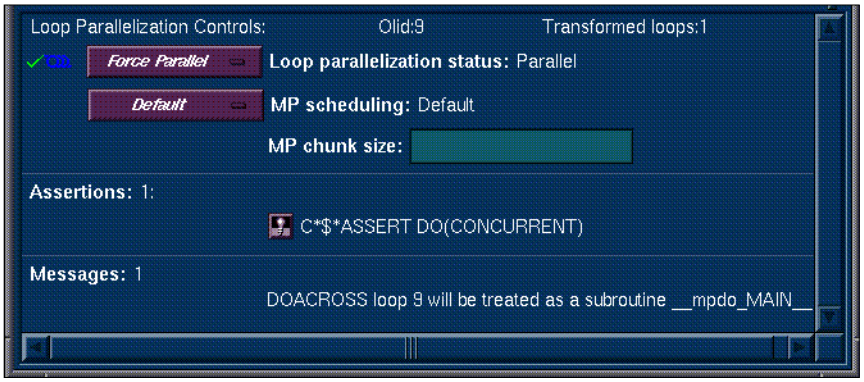

Figure 2-18 Parallelizable Data Dependence

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 use of the variable occurs on a different line than its setting. Click the highlight button on the obstacle line, and note in the Source View that highlighting shows the dependency variable on the two lines (see Figure 2-19). Of course, real programs can, and typically do, have far more complex dependencies than this.

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

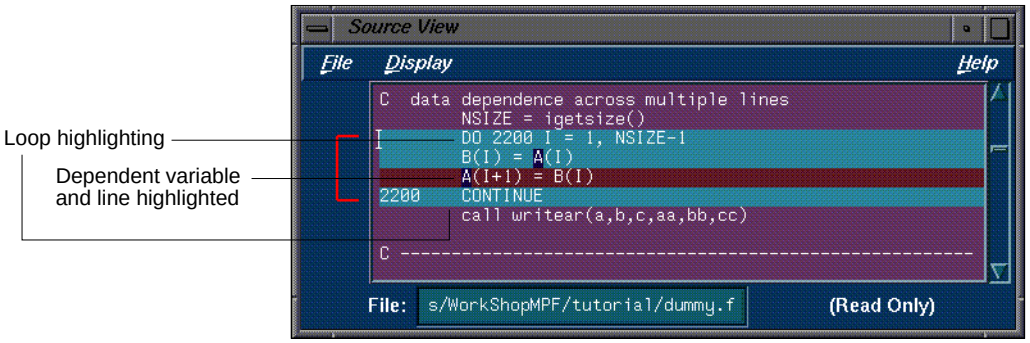

Figure 2-19 Highlighting on Multiple Lines

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. Reductions can be summation, multiplication, or a minimum or maximum determination. For summation, as shown in this loop, the code could accumulate partial sums in each processor, and then add the partial sums at the end.

However, because floating-point arithmetic is inexact, the order of addition might give different answers because of round-off 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 round-off error. With the **-O3** optimization level, the compiler assumes it is OK to introduce round-off error; the loop is parallelized. If you do not want a loop parallelized because of the change in round-off error, compile with the **-OPT:roundoff=0** or **1** option (see *MIPSpro Automatic Parallelizer Programmer's Guide*).

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

Input-Output 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.

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 the *Next Loop* button.

Unstructured Control Flows

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

For **Olid 13**, the compiler cannot determine how many iterations will take place before exiting the loop; the `goto` statement is essential to the program's behavior. 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 section of the loop information display where the message says, "unstructured control flow." Note that the line with the exit from the loop is highlighted in the Source View.

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

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: for example, creating data dependencies.

Unparallelizable Loop With a Subroutine Call

Loop **Olid 14** is unparallelizable because there is a call to a routine, **RTC()**, and there is no explicit assertion to parallelize. 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 an assertion that the call has no side effects that will prevent concurrent processing. 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 the *Next Loop* button.

Permutation Vectors

If you specify array index values by values of another array (which is referred to as a permutation vector), the compiler cannot determine if the values of 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 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.

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

Parallelizable Loop With a Permutation Vector

An assertion has been added before loop **Olid 17** that the index array, *IB(I)*, is indeed a permutation vector, and the loop is parallelized.

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

Examples of Nested Loops

The following loops illustrate somewhat more complicated situations: nested loops.

Doubly Nested Loop

Loop **Olid 18** is the outer loop of a pair of loops; it runs in parallel, and the inner loop runs in serial: 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$DOACROSS** directive with the NEST clause; for example, see “Distributed and Reshaped Arrays: C$DISTRIBUTE_RESHAPE” on page 57.

Move to the inner loop **Olid 19** by clicking the *Next Loop* button, and then click *Next Loop* again to select the outer loop of the next nested pair.

Note: 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.

Interchanged Doubly Nested Loop

Note that the outer loop, loop **Olid 20**, is shown in the information display as serial inside a parallel loop, and the original interior loop is indicated to be parallel: the order of the loops has been interchanged. This happens because the compiler recognized that the two loops can be interchanged, and furthermore, 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.

Move to the inner loop, **Olid 21**, by clicking the *Next Loop* button, and then click the *Next Loop* button 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 for the previous nested loops, the compiler recognizes that cache misses are less likely. As you double-click on **Olid 22**, **Olid 23**, and **Olid 24** in the loop list, note that the loop information display shows that **Olid 22** and **Olid 24** are serial loops inside a parallel loop, **Olid 23**.

But iterations of loop **Olid 22** can run concurrently: the innermost serial loop **Olid 24** depends without recurrence on the indices of **Olid 22** and **Olid 23**. The compiler does not recognize this possibility: this is a good context to illustrate the use of the Parallel Analyzer View tools to modify the source, which is the subject of the next section.

Modifying Source Files and Compiling

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

There are two steps to modifying source files:

1. Request changes using the Parallel Analyzer View controls, discussed in the next subsection, “Requesting Changes.”
2. Modify the source and rebuild the program and its analysis files, discussed in “Applying Requested Changes” on page 48.

Requesting Changes

You request changes by one of the following actions:

- Add or delete assertions or directives using the Operations menu or the Loop Parallelization Controls.
- Add or modify clauses to 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.

These are the topics discussed in this section:

- "Directives and Assertions" on page 43
- "Adding a C$DOACROSS Directive and Clauses" on page 43
- "Adding a New Assertion or Directive With the Operations Menu" on page 46
- "Deleting an Assertion or a Directive" on page 48

Directives and Assertions

You control most of the directives and some assertions available from the Parallel Analyzer View with the Operations menu (see Table 4-1).

You control most of the assertions and the more complex directives (see Table 4-3 and Table 4-4), **C$DOACROSS** and **C$PAR PDO**, with the Loop Status Option menu (see Figure 2-20).

Adding a C$DOACROSS Directive and Clauses

Loop **Olid 22** is a serial loop nested inside a parallel loop, but its iterations could run concurrently. To parallelize **Olid 22**, do the following:

1. Click on the Loop Status Option menu, which is the first Loop Parallelization Control in the loop information display. It is between the loop icon and the text "Loop parallelization status" and initially reads "Default" (see, for example, Figure 2-18).
2. Choose "C$DOACROSS..." This requests a change in the source code, and opens the Parallelization Control View (see Figure 2-21), which allows you to look at variables in the loop and to attach clauses to the directive, if needed. The loop information display should appear as in Figure 2-20.

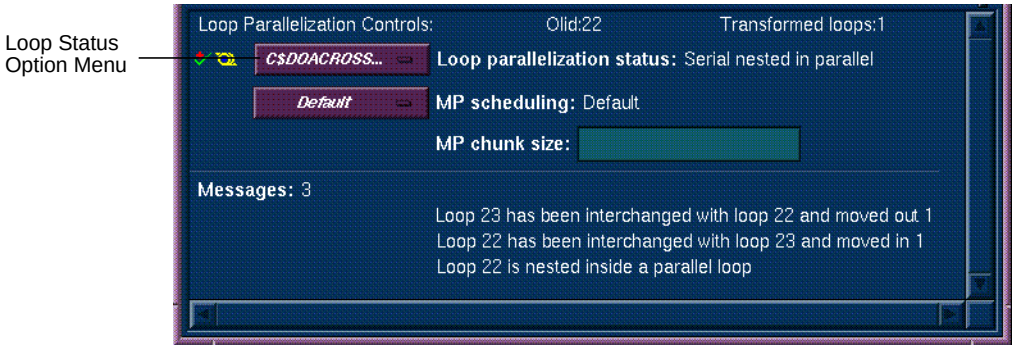

Figure 2-20 Requesting a C$DOACROSS for Olid 22

Figure 2-21 shows information presented in the Parallelization Control View (for details, see “Parallelization Control View” on page 97):

- the selected loop
- “Condition for parallelization” text field (“C$DOACROSS...” only)
- MP Scheduling menu
- MP chunk size text field
- Synchronization Construct menu (“C$PAR PDO...” only)
- 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 described in the Icon Legend

  In the list of variables, each variable has a highlighting button to indicate in the Source View its use within the loop; click on 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 “Parallelization Control View Variable List: Storage Labeling” on page 104).

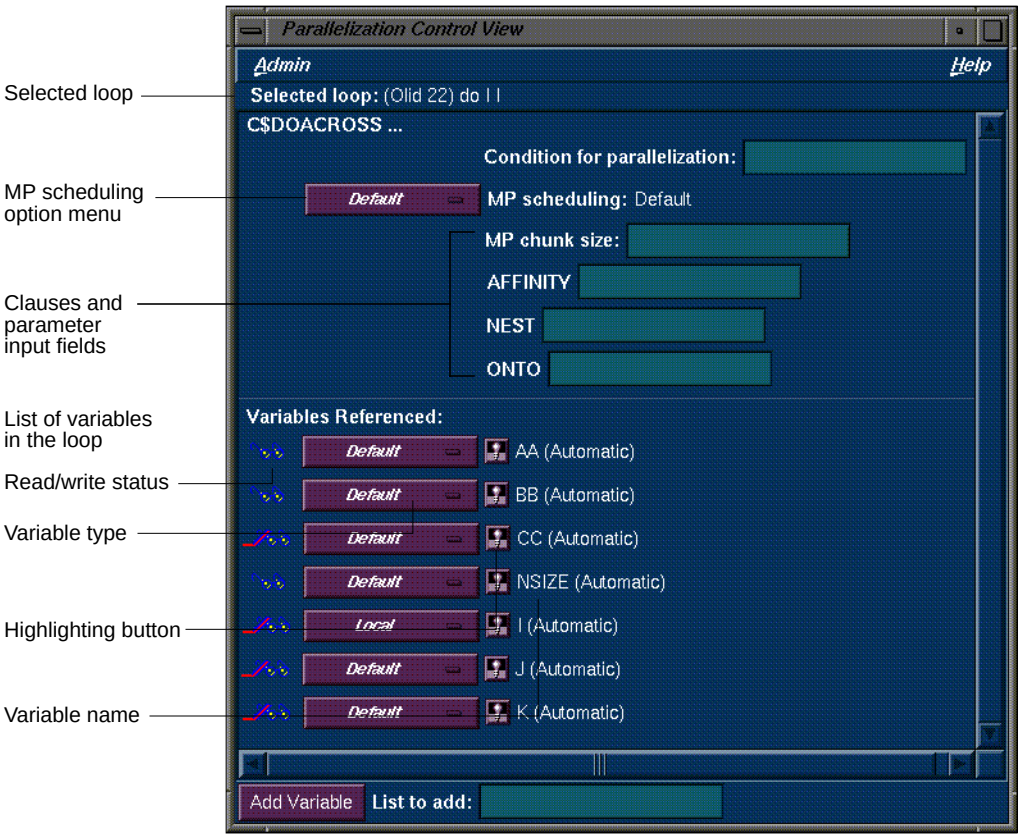

Figure 2-21 Parallelization Control View for Loop Olid 22 After Choosing “C$DOACROSS...”

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

Notice that in the loop list display, there is now a red plus sign next to this loop, indicating that a change has been requested (see Figure 2-22).

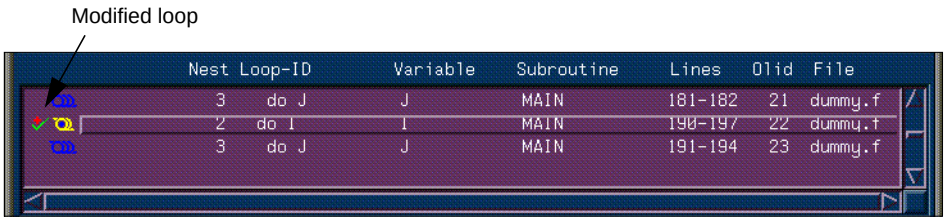

Figure 2-22 Effect of Changes on the Loop List Display

Close the Parallelization Control View; it is no longer needed.

Adding a New Assertion or Directive With the Operations Menu

Now you can add an assertion to a loop.

Find the loop **Olid 14**. You can do this either by scrolling the loop list display or by using the search feature of the loop list (go to the Search field, and enter **14)**. Double-click the highlighted line in the loop list to select the loop.

To request a new assertion

1. Pull down the Operations menu
2. Choose the Add Assertion submenu

For this example, choose “C*$*ASSERT CONCURRENT CALL” (see Figure 2-23). This adds an assertion that it is safe to parallelize the loop despite the call to **RTC()**, which the compiler thought might be an obstacle to parallelization.

After you choose “C*$*ASSERT CONCURRENT CALL,” the loop information display shows the new assertion, along with a menu labeled “Insert” to indicate the state of the assertion when you modify the code (see Figure 2-23).

The procedure for adding directives is similar; start by choosing the Add Directive submenu of the Operations menu.

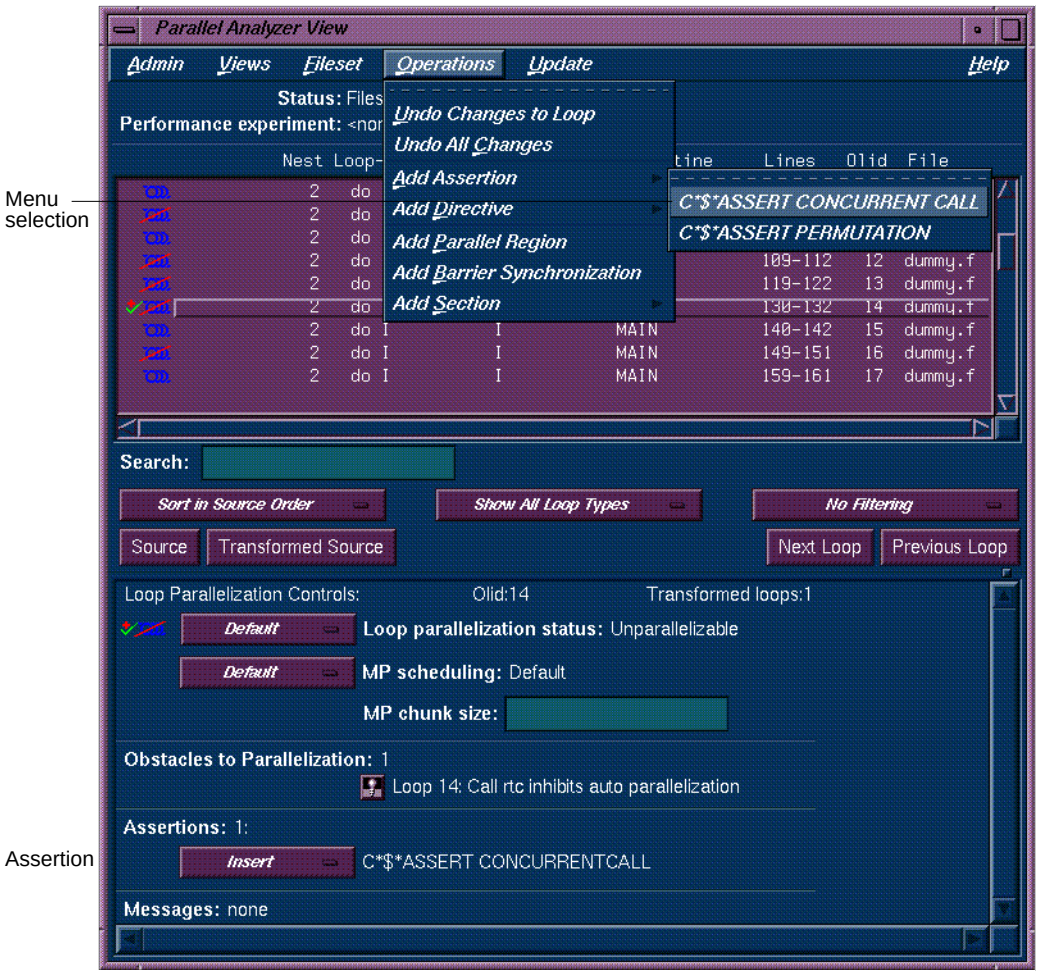

Figure 2-23 Adding an Assertion

Deleting an Assertion or a Directive

Move to the next loop, **Olid 15**, and note “ASSERT CONCURRENT CALL” in the loop information display. Pull down its option menu and choose “Delete.” Figure 2-24 shows the state of the assertion in the information display. The same procedure can be used to delete directives.

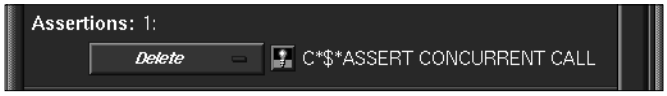

Figure 2-24 Deleting an Assertion

Applying Requested Changes

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

1. Generates a *sed* script for the following steps.
2. Renames the original file to have the suffix *.old*.
3. Runs *sed* on that file to produce a new version of the file, in this case *dummy.f*.
4. Depending on how you set the two toggles 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 toggles for both windows. When you quit these tools, the Parallel Analyzer View spawns the WorkShop Build Manager.
5. After the build, the Parallel Analyzer View rescans the files and loads the modified code for further interaction.

Viewing Changes With gdiff

To open a *gdiff* window that shows the requested changes to the source file before compiling the modified code, choose the toggle labeled “Run gdiff After Update” from the Update menu.

By default, the Parallel Analyzer View does not open a *gdiff* window. 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

To open an editor and make additional changes after the *sed* script runs and before compiling the modified code, choose the toggle labeled “Run Editor After Update” in the Update menu (see Figure 2-25). An *xwsh* window with *vi* running in it opens with the source code ready to be edited.

Figure 2-25 Setting the Run Editor Toggle

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 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.

Updating the Source File

To update the source file to include requested changes, choose “Update All Files” from the Update menu (see Figure 2-26); alternatively, you can use the keyboard shortcut for this operation, **`Ctrl+U`**, with the cursor anywhere in the main view.

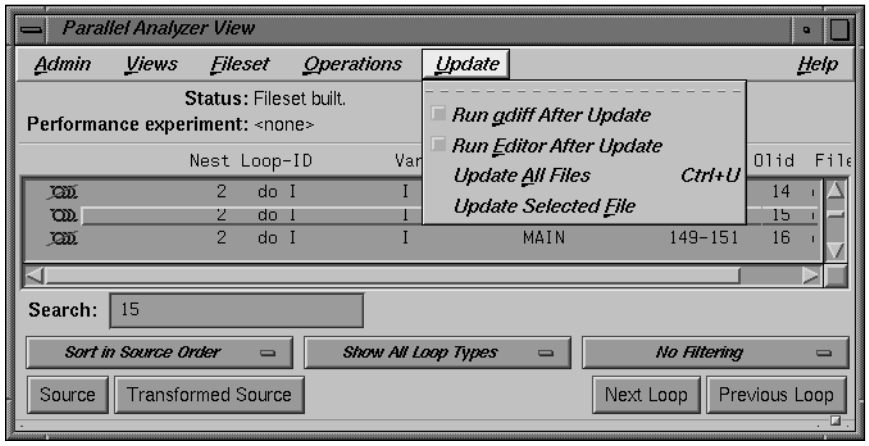

Figure 2-26 Update All Files

If you have set the toggle 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 directory in which you are running the sample session; if not, change it.

Then, 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.

New Assertion

To see the effect of the assertion request in “Adding a New Assertion or Directive With the Operations Menu” on page 46, scroll to **Olid 14** or use the Search field. 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 an Assertion or a Directive” on page 48, and the loop no longer runs in parallel. Recall that the loop previously had the assertion that **foo()** was not an obstacle to parallelization.

Examples With PCF Directives

This section discusses examining the subroutine **pcfdummy**, which contains four parallel regions that illustrate the use of PCF directives:

- "Explicitly Parallelized Loops: C$PAR PDO" on page 52
- "Loops With Barriers: C$PAR BARRIER" on page 54
- "Critical Sections: C$PAR CRITICAL SECTION" on page 55
- "Single-Process Sections: C$PAR SINGLE PROCESS" on page 55
- "Parallel Sections: C$PAR PSECTIONS" on page 55

For more information on PCF directives, see Chapter 5 of the *MIPSpro Fortran 77 Programmer's Guide*.

To go to the first parallel region of **pcfdummy**, scroll down the loop list, or use the Search field (enter **`parallel`**).

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

Explicitly Parallelized Loops: C$PAR PDO

The first construct in routine **pcfdummy** is a parallel region, **Olid 26**, containing two loops that are explicitly parallelized with **C$PAR PDO** statements (see Figure 2-27). With this construct, the second loop can start before all iterations of the first complete.

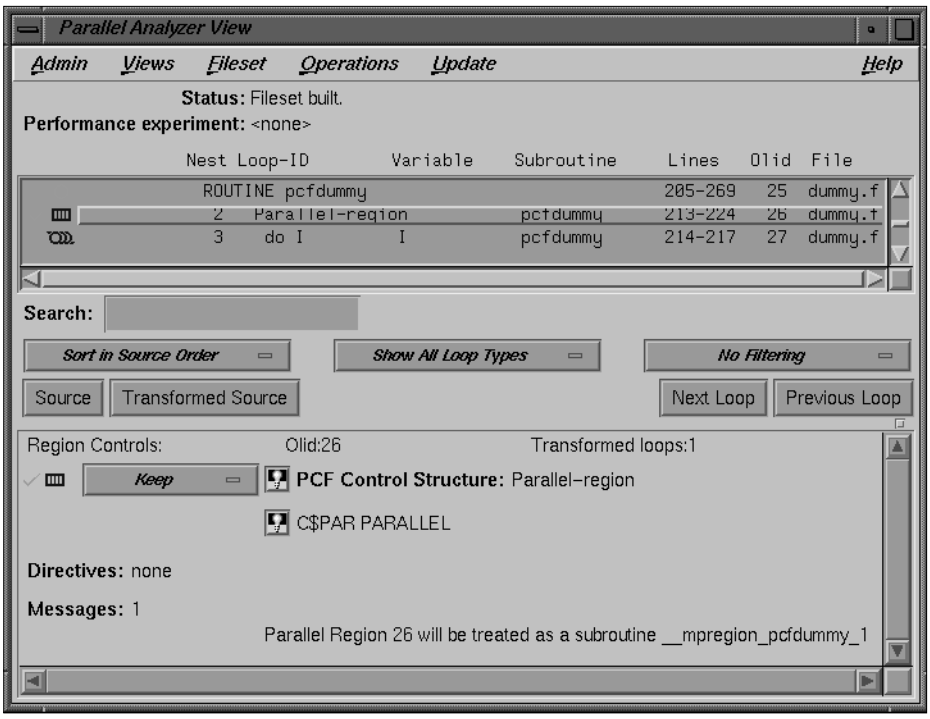

Figure 2-27 Explicitly Parallelized Loops With C$PAR PDO

Notice in Figure 2-27 that the parallel region has controls specific to the region as a whole. The “Keep / Delete” option menu and the highlight buttons function the same way they do in the Loop Parallelization Controls (see “Loop Parallelization Controls” on page 23).

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

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

Loops With Barriers: C$PAR BARRIER

The second parallel region, **Olid 29**, contains an identical pair of loops, but with a barrier between them. Click *Next Loop* twice to view the barrier region (see Figure 2-28).

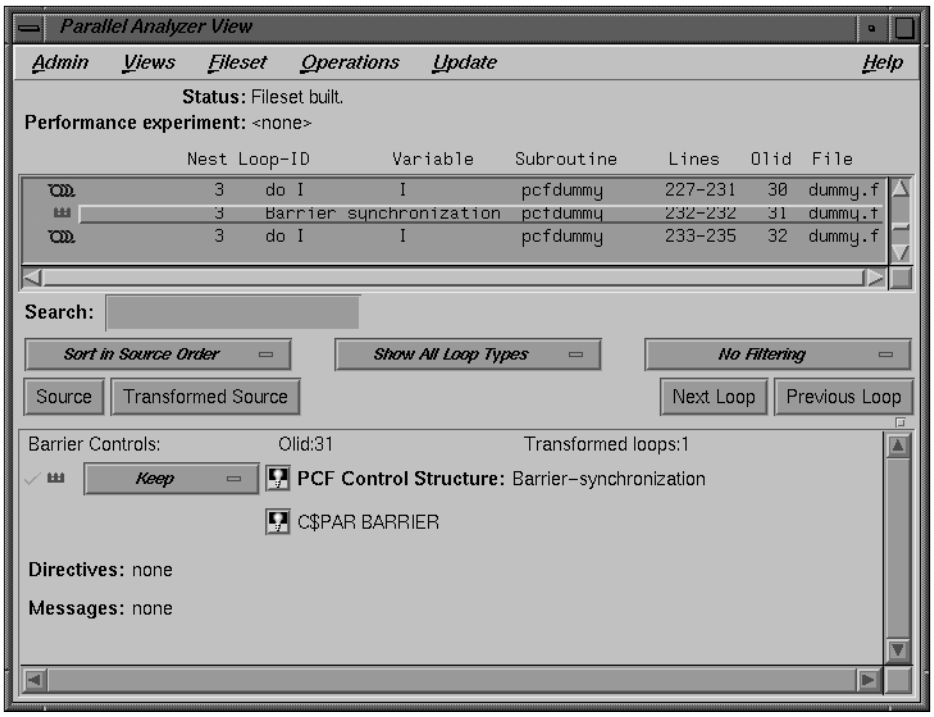

Figure 2-28 Loops With Barrier Synchronization

All iterations of the first **C$PAR PDO** must complete before any iteration of the second loop can begin.

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

Critical Sections: C$PAR CRITICAL SECTION

To view the first of the two loops in the third parallel region, **Olid 33**, click *Next Loop*. This loop contains a critical section. Click *Next Loop* to view the critical section.

The critical section uses a named locking variable (*S3* in this case) and uses the lock to prevent simultaneous update of *S1* from multiple threads. This is a standard construct for performing a reduction.

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

Single-Process Sections: C$PAR SINGLE PROCESS

Loop **Olid 36** has a single-process section, which ensures that only one thread can execute the statement in the section. Highlighting shows the begin and end directives. Click *Next Loop* to view information about the single-process section.

Move to **pcfdummy**'s final parallel region by clicking the *Next Loop* button.

Parallel Sections: C$PAR PSECTIONS

The fourth and final parallel region of **pcfdummy**, **Olid 38**, 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.

Click *Next Loop* to view the parallel section.

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

Examples With Data Distribution Directives

The last series of subroutines illustrate directives that control data distribution and cache storage:

- “Distributed Arrays: C$DISTRIBUTE” on page 56
- “Distributed and Reshaped Arrays: C$DISTRIBUTE_RESHAPE” on page 57
- “Prefetching Data From Cache: C*$*PREFETCH_REF” on page 58

Brief descriptions of these directives appear in Table 4-1. For more details about these directives, see the *MIPSpro Fortran 77 Programmer's Guide*, Chapter 6; and *MIPSpro Compiling and Performance Tuning Guide*, Chapter 4 (for **C*$*PREFETCH_REF**).

Distributed Arrays: C$DISTRIBUTE

Note that when you select the subroutine **dst1d**, which is **Olid 40**, a directive is listed in the loop information display that is global to the routine; the directive **C$DISTRIBUTE** specifies placement of array members in distributed, shared memory.

In the text box 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 on the highlight button.

Click on the *Next Loop* button to move to the parallel loop, **Olid 41**.

The loop has a **C$DOACROSS** directive, which works with the **C$DISTRIBUTE** directive so that each processor manipulates locally stored data.

You can highlight the **C$DOACROSS** directive in the Source View with either of the highlight buttons in the loop information display. If you use the highlight button in the Loop Parallelization Controls, the Parallelization Control View presents more information about the directive and allows you to make changes to **C$DOACROSS** 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$DISTRIBUTE_RESHAPE

When you select the subroutine **rshape2d**, which is **Olid 42**, the routine's global directive is listed in the loop information display. The directive **C$DISTRIBUTE_RESHAPE** also specifies placement of array members in distributed, shared memory; it differs from **C$DISTRIBUTE** mainly in that the unit of memory allocation is not necessarily a page.

In the text box 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, **Olid 43**.

The loop has a **C$DOACROSS** directive, which works with the **C$DISTRIBUTE_RESHAPE** directive so that each processor manipulates locally stored data.

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 NEST clause, you could have inserted it by supplying the arguments in the text field in the Parallelization Control View. Recall that you can use the NEST clause to parallelize nested loops only when there is no code between either the `do-i` and `do-j` statements or the `enddo-i` and `enddo-j` statements (see Chapter 6 of the *MIPSpro Fortran 77 Programmer's Guide*).

Click the *Next Loop* button to move to the nested loop, **Olid 44**.

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* again to view the next subroutine; click again to go to the first loop in the subroutine **prfetch**, **Olid 46**.

Prefetching Data From Cache: C*$*PREFETCH_REF

As for the nested loops **Olid 20** and **21**, the compiler switched the order of execution of the nested loops **Olid 46** and **47**; to see this, look at the Transformed Source view.

Click on the *Next Loop* button to move to the nested loop, **Olid 47**.

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 dummy.f Sample Session

This completes the first sample session. Quit the Parallel Analyzer View by choosing "Exit" from the Admin menu.

Not all windows opened during the session close when you quit. In particular, the Source View remains open. This is because all the Developer Magic tools interoperate, and other tools may share the Source View window (see "Viewing Original Source" on page 17). You close the Source View independently.

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

```
% make clean
```

Chapter 3

Using WorkShop With Parallel Analyzer View

The second sample session is a brief demonstration of the integration of WorkShop Pro MPF 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. Chapter 3 of the *SpeedShop User's Guide* presents a tutorial analysis of linpack.

This tutorial assumes you are already familiar with the basic features of the Parallel Analyzer View discussed in the previous chapter. You can also consult Chapter 4, "Parallel Analyzer View Reference," for more information.

Setting Up the linpackd Sample Session

Go to */usr/demos/WorkShopMPF/linpack* directory and run *make*:

```
% cd /usr/demos/WorkShopMPF/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 directory, in the file *test.linpack.cp*.

Starting the Parallel Analyzer View

Once 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. Scroll briefly through the list and the Source View (by clicking the *Source* button in the main view). 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 Performance Analyzer

Start the Performance Analyzer by pulling down the Admin menu, choosing the Launch Tool submenu, and choosing "Performance Analyzer," as shown in Figure 3-1.

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. Enter:

```
test.linpack.cpu
```

in the "Experiment Dir:" field in the Experiment: window, and click the *OK* button. See Figure 3-1. 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 message "Loading Performance Data." When the message goes away, performance data will have been imported by the Parallel Analyzer, and a number of changes will have taken place as shown in Figure 3-2.

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

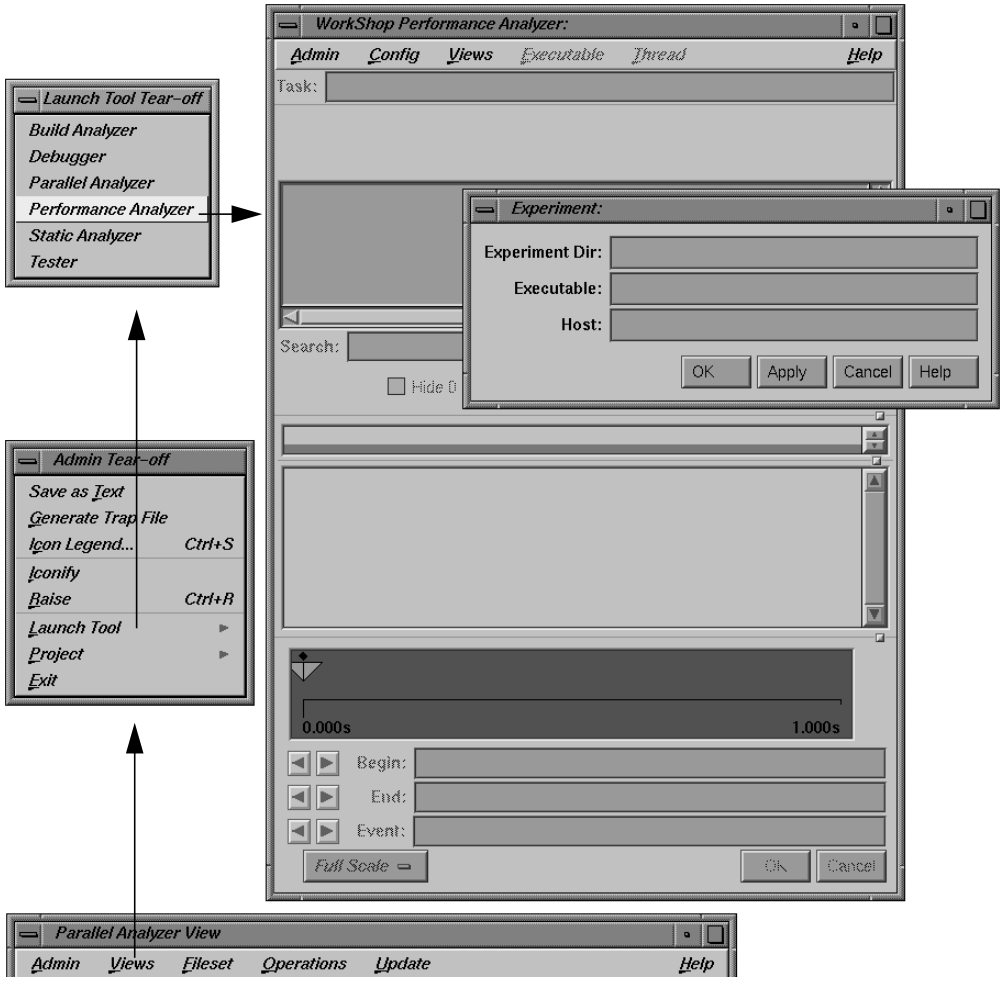

Figure 3-1 Starting the Performance Analyzer

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 3-2:

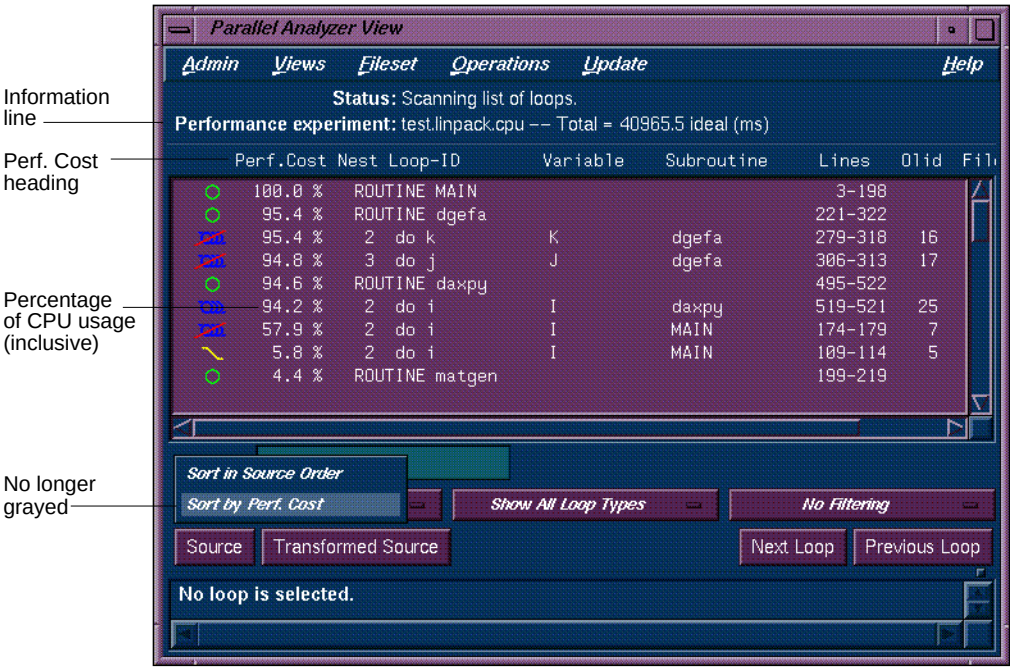

Figure 3-2 Performance Data — Parallel Analyzer View

- A new column titled“Perf. Cost” appears in the list of loop next to the icon column. The numbers in this column are inclusive: each reflects the time in the loop and in any nested loops or functions called from within the loop.
- The second line in the view, below the menu bar, now shows the name of the performance experiment and shows the total cost of the run in milliseconds.
- The sort menu’s second option, “Sort by Perf. Cost,” is active.
- In the Source View, three additional columns appear to the left of the loop brackets; they reflect the measured performance data (these columns may take a few moments to load):
  - the number of times the line has been executed
  - exclusive, ideal CPU time in milliseconds
  - 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 25**, which is in subroutine **daxpy**; the Source View appears as shown in Figure 3-3.

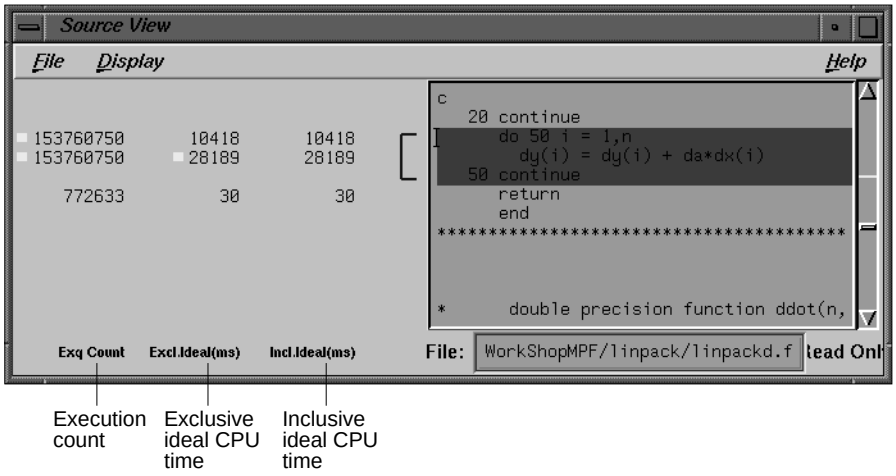

Figure 3-3 Source View for Performance Experiment

Sorting the Loop List by Performance Cost

Choose the “Sort by Perf. Cost” entry. Note that the third most expensive loop listed, **Olid 25** of subroutine **daxpy**, represents approximately 92% of the total time (see Figure 3-4).

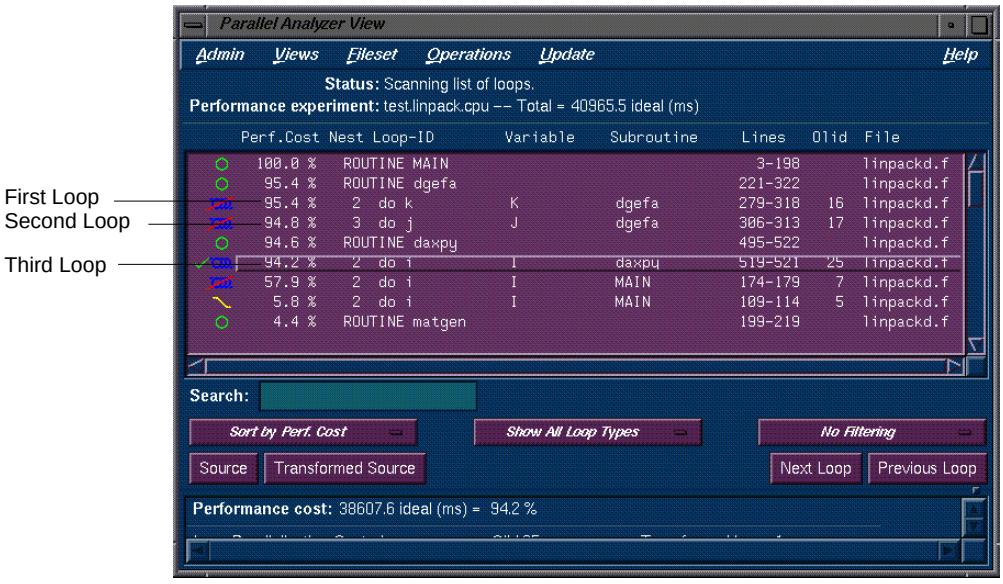

Figure 3-4 Sort by Performance Cost

The first of the high-cost loops (**Olid 16** in subroutine **dgefa**) contains the second loop nested inside it (**Olid 17)**. The second loop calls **daxpy**, which contains **Olid 25**—the heart of the *linpack* benchmark; **Olid 25** performs the central operation of scaling a vector and adding it to another vector. **Olid 25** was parallelized by the compiler; note the C$DOACROSS directive that appears for this loop in the Transformed Source View.

The loop following **Olid 25**, loop **dgefa**, uses approximately 58% of the CPU time. This loop is the most frequent caller of **dgefa**, and so of **Olid 25**.

Double-click **Olid 25**. 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 3-5).

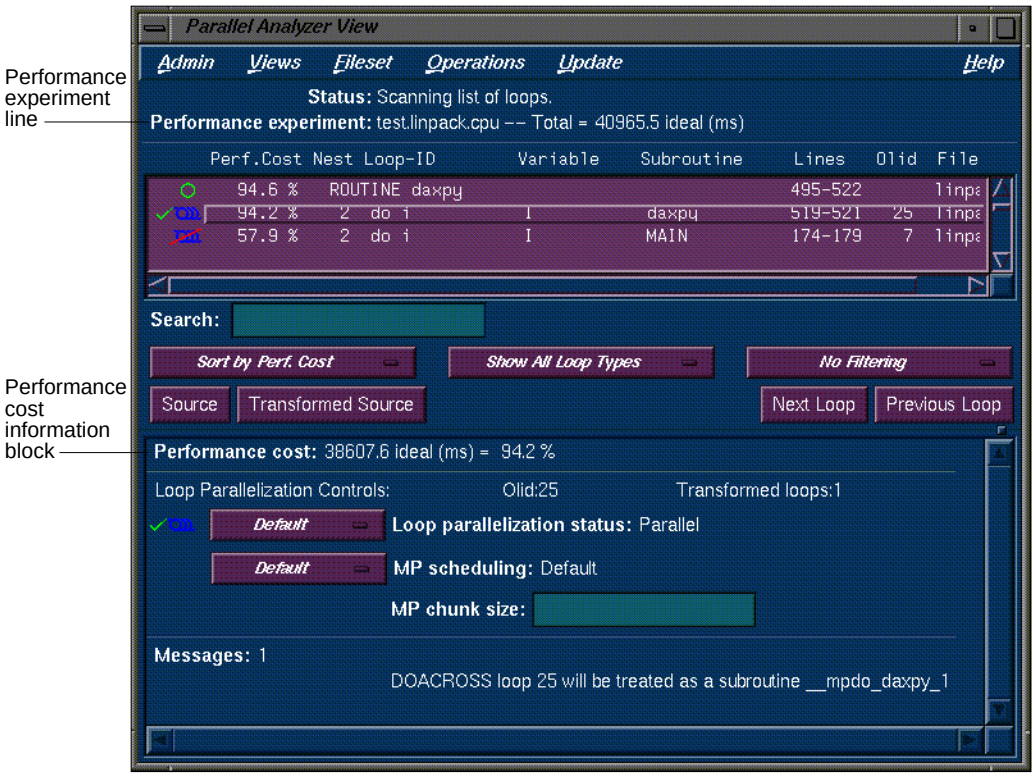

Figure 3-5 Loop Information Display With Performance Data

Exiting From the linpackd Sample Session

This completes the second sample session. To 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, quit by selecting the “Exit” command from the Project submenu of the Admin menu in the Parallel Analyzer View.

You don’t need to clean the directory, because you haven’t made any changes in this session. If you experiment and 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
```

Chapter 4

Parallel Analyzer View Reference

This chapter describes in detail the function of each window, menu, and display in the WorkShop Pro MPF Parallel Analyzer View's user interface.

This chapter contains the following main sections:

- "Main View Menu Bar" on page 68
- "Loop List" on page 84
- "Loop Information Display" on page 90
- "Other Views" on page 97
  - "Parallelization Control View" on page 97
  - "Transformed Loops View" on page 105
  - "PFA Analysis Parameters View" on page 106
  - "Subroutines and Files View"
  - "Source View and Transformed Source Windows" on page 109
- "Icon Legend" on page 110

Main View Menu Bar

This section describes the menus found in the menu bar located at the top of the Parallel Analyzer View main window as shown in Figure 4-1.

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” on page 83.

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.

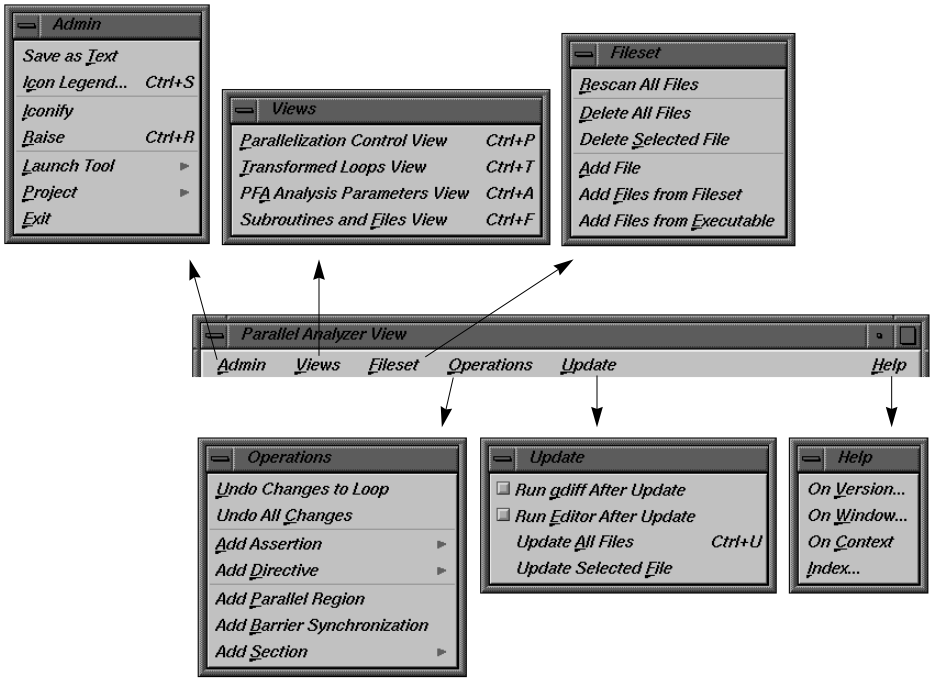

Figure 4-1 Parallel Analyzer View Menu Bar and Pulldown Menus

Admin Menu

Figure 4-2 shows the Admin menu, which contains file-writing commands, other administrative commands, and commands for launching and manipulating other WorkShop application views.

Figure 4-2 Main View Admin Menu

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 "Save as Text" brings up the directory and file browser, which lets you choose where to save the file and what name to call it (see Figure 4-3).

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.

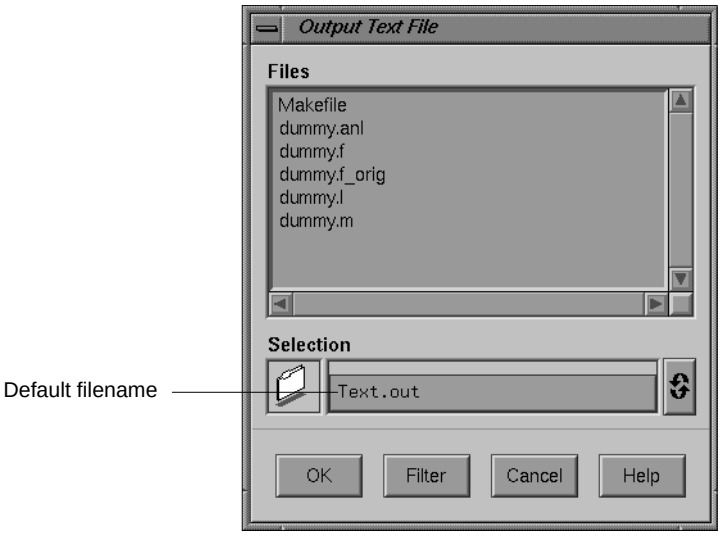

Figure 4-3 Directory and File Browser Window

Icon Legend... Opens the Icon Legend window, which provides an explanation of the graphical icons used in the Parallel Analyzer View. See "Icon Legend" on page 110. Shortcut: **`Ctrl+S`**

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 See "Launch Tool Submenu" on page 71.

Project See "Project Submenu" on page 73.

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* and update the files.

Launch Tool Submenu

The Launch Tool submenu contains commands for launching other WorkShop tools, as well as new sessions of the Parallel Analyzer (see Figure 4-4).

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 (see “Running the Parallel Analyzer View: General Features” on page 2)
- Use the “Add Files from Executable” command found in the Fileset menu (see “Fileset Menu” on page 76).

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

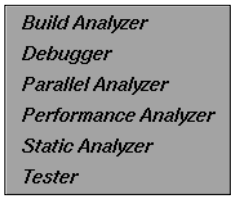

Figure 4-4 Launch Tool Submenu

The following applications can be launched from the Launch Tool menu:

Build Manager
: Launches the Build Manager, a utility that lets you compile software without leaving the WorkShop environment. For more information, see Appendix B, “Using the Build Manager,” in the *Developer Magic: Debugger User’s Guide*.

WorkShop Debugger
: Launches the Debugger, a UNIX® source-level debugging tool that provides special windows for displaying program data and execution status. For more information, see Chapter 1, “Getting Started with the WorkShop Debugger,” in the *Developer Magic: 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 Chapter 1, “Introduction to the Performance Analyzer,” in the *Developer Magic: Performance Analyzer User’s Guide*.

Static Analyzer
: Launces the Static Analyzer, a utility that allows you to analyze and display source code written in C, C++, Fortran, or Ada. For more information, see Chapter 1, “Introduction to the WorkShop Static Analyzer,” in the *Developer Magic: 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 Chapter 5, “Using Tester,” in the *Developer Magic: Performance Analyzer 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 contains commands that affect all the windows containing WorkShop or WorkShop Pro MPF 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 4-5.

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 filesystem. 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.

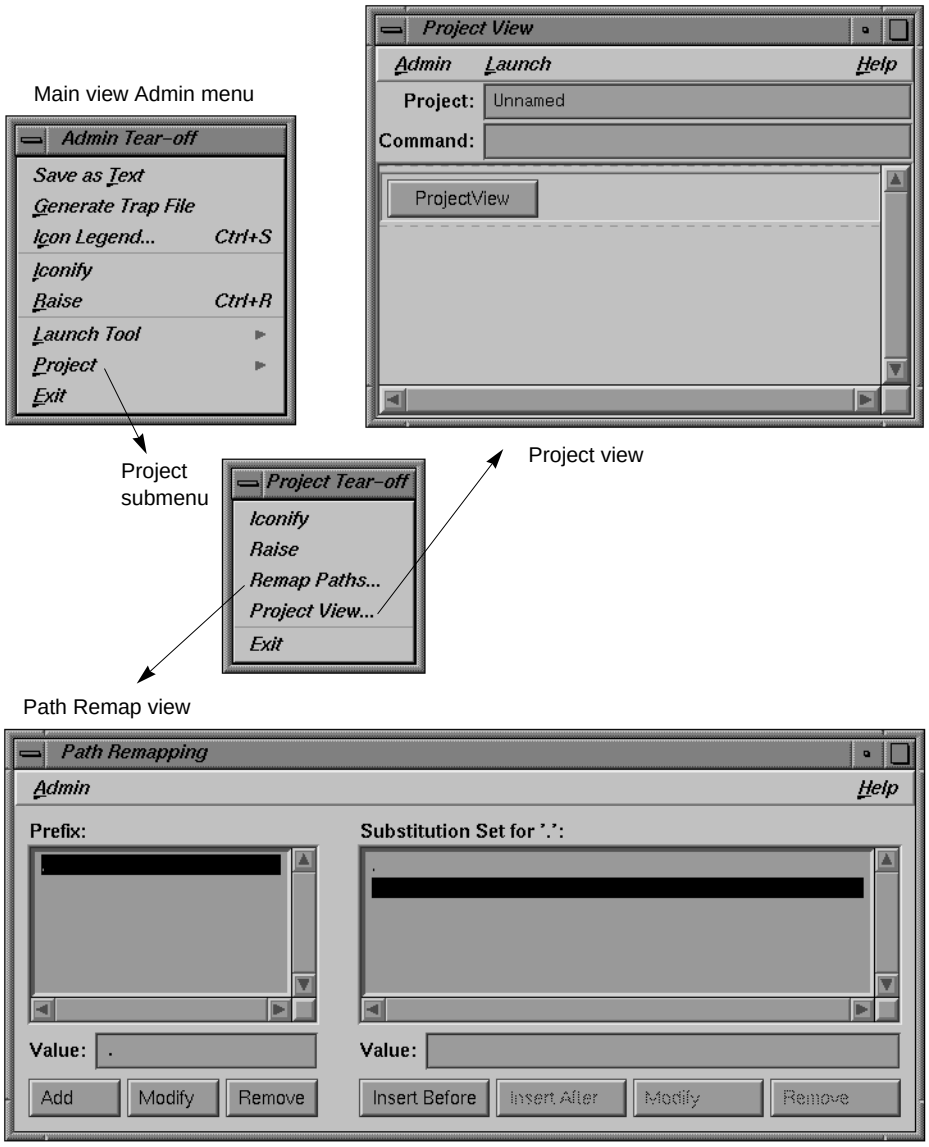

Figure 4-5 Project Submenu Commands

Views Menu

The Views menu (see Figure 4-6) 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.

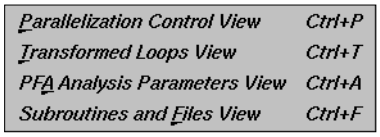

Figure 4-6 Views Menu

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. For more information on this view, see "Parallelization Control View" on page 97. Shortcut: **`Ctrl+P`**

Transformed Loops View
: Opens a Transformed Loops View for the loop currently selected from the loop list display. For more information on this view, see "Transformed Loops View" on page 105. Shortcut: **`Ctrl+T`**

PFA Analysis Parameters View
: Opens the PFA Analysis Parameters View, which provides a means of modifying a variety of PFA parameters. This view is further described in "PFA Analysis Parameters View" on page 106. Shortcut: **`Ctrl+A`**

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. This view is further described in "Subroutines and Files View" on page 108. Shortcut: **`Ctrl+F`**

Fileset Menu

The Fileset menu (see Figure 4-7) 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.

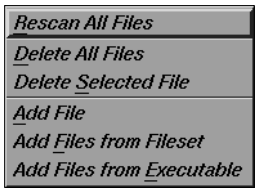

Figure 4-7 Fileset Menu

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 filesystem. The Parallel Analyzer View only rereads 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" commands, 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 and directory browser 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 Parallel Analyzer View" on page 3).

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

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 the file and directory browser as it does for the “Add File” command. 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 (see “Add Files From 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 the file and directory browser as it does for the “Add File” command. Other WorkShop applications can also operate on files imported from an executable.

Operations Menu

The Operations menu contains commands for adding assertions and directives to loops and removing pending changes to source files (see Figure 4-8). 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 the “Update Menu” on page 80.

The assertions and directives you can add from the Operations menu are listed in Table 4-1; the manuals where you can find more information are also listed.

The Operations menu is one of two points in the Parallel Analyzer View where you can add assertions and directives. The other is discussed in “Loop Parallelization Controls in the Loop Information Display” on page 91. 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 automatic parallelizer, and memory distribution.
- The parallelization controls in the Loop information display focus on “manual” parallelization controls, which you can insert to further parallelize your code.

Table 4-1 Assertions and Directives in the Operations Menu

Assertion or Directive	Effect on Compilation	For More Information
C*$*ASSERT_CONCURRENT CALL	Ignore possible dependencies due to subroutine calls. Typically inserted during automatic parallelization.	*MIPSpro Automatic Parallelizer Programmer's Guide*, Chapter 3
C*$*ASSERT PERMUTATION	The indexing array is a permutation. Typically inserted during automatic parallelization.	*MIPSpro Automatic Parallelizer Programmer's Guide*, Chapter 3
C*$*CONCURRENTIZE	Overrides C*$*NOCONCURRENTIZE.Typically inserted during automatic parallelization.	*MIPSpro Automatic Parallelizer Programmer's Guide*, Chapter 3
C*$*NOCONCURRENTIZE	Do not parallelize file subroutine (depending on placement). Typically inserted during automatic parallelization.	*MIPSpro Automatic Parallelizer Programmer's Guide*, Chapter 3
C$DISTRIBUTE and C$REDISTRIBUTE	Distribute array storage among processors. For Origin2000 systems.	*MIPSpro Fortran 77 Programmer's Guide*, Chapter 6
C*$*PREFETCH_REF	Load data into cache. May be used with nonconcurrent code.	*MIPSpro Compiling and Performance Tuning Guide*, Chapter 4
C$DYNAMIC	Allow run-time array redistribution. For Origin2000 systems.	*MIPSpro Fortran 77 Programmer's Guide*, Chapter 6
C$COPYIN	Copy COMMON block into local thread.	*MIPSpro Fortran 77 Programmer's Guide*, Chapter 5

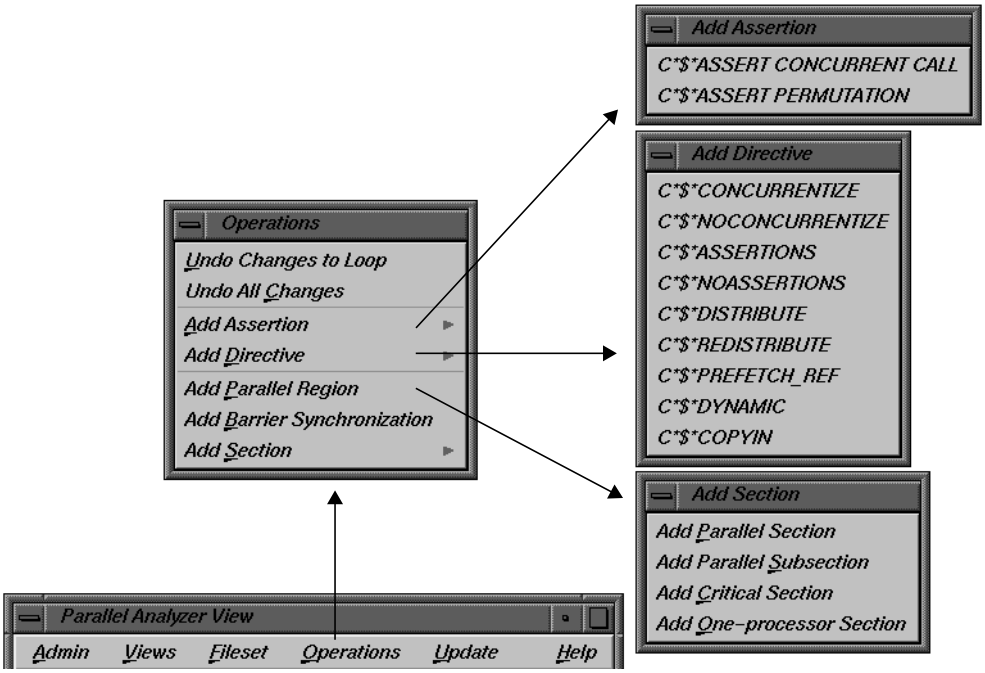

Figure 4-8 Operations Menu and Submenus

The options in the Operations 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 submenu
: Contains a set of assertions that you can add to the currently selected loop.

Add Directive submenu
: Contains a set of directives to add to the currently selected loop.

Add Parallel Region
: Allows you to add a parallel region PCF construct.

Add Barrier Synchronization
: Allows you to add a barrier synchronization PCF construct.

Add Section submenu
: Allows you to add a parallel-, critical- or one-processor section. To use them, bring up in the Source View, and using the mouse, sweep out a range of lines for the new construct. Then invoke the appropriate menu item to add the new construct.

  When you add a new 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.

  Note: 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.

Update Menu

The Update menu (see Figure 4-9) contains commands for placing in your Fortran source code requested changes to directives and assertions that you made with the Parallel Analyzer View.

Figure 4-9 Update Menu

The options in the Update menu have the following effects:

Run gdiff After Update

Sets a toggle switch that causes a *gdiff* window to open after you have updated changes to your source file. This window graphically illustrates 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 toggle switch that opens *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 *.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 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" on page 108).

Help Menu

The Help menu contains commands that allow you to access online information and documentation for the Parallel Analyzer View (see Figure 4-10).

Figure 4-10 Help Menu

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 the "On Context" command, 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 4-2 lists the keyboard shortcuts that work in the Parallel Analyzer View:

Table 4-2 Keyboard Shortcuts

Keyboard Shortcut	Menu	Submenu
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	PFA Analysis Parameters View
Ctrl+U	Update	Update All Files

Loop List

This section describes the loop list and the various option menus and fields that manipulate the information shown in the loop list display, shown in Figure 4-11.

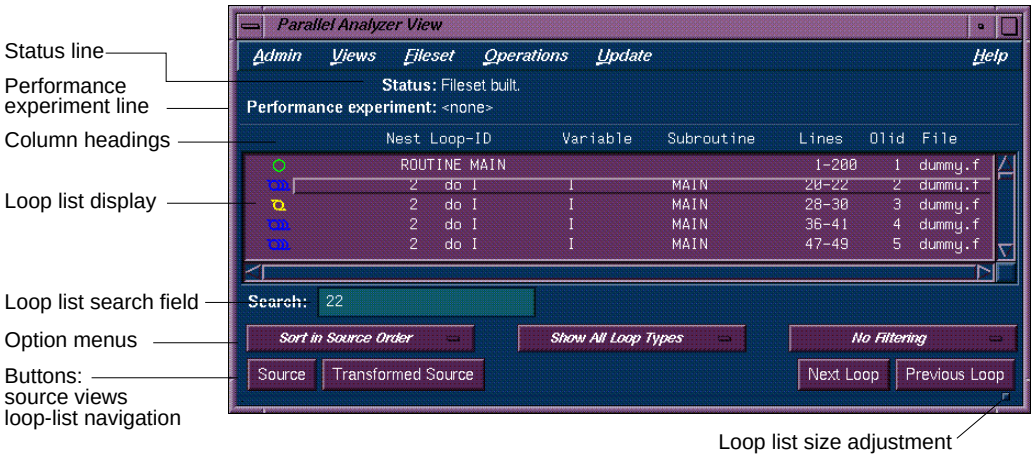

Figure 4-11 Loop List Display and Controls

Resizing the Loop List

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” on page 71 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 Display

The loop list display 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 display; the headings of the columns are shown in Figure 4-12 and described below.

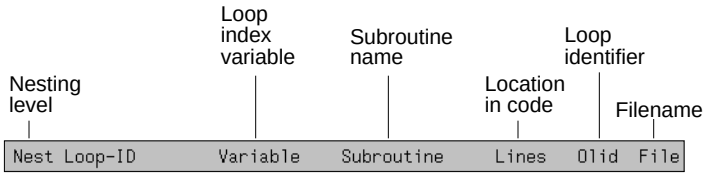

Figure 4-12 Column Headings for the Loop List Display

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

Parallelization icon
: Indicates the parallelization status of each loop. The meaning of each of these icons is described in the Icon Legend dialog box (see “Icon Legend” on page 110). 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 (performance cost)
: Displayed when the WorkShop Performance Analyzer is launched on the current fileset (see “Launch Tool Submenu” on page 71). The loops can be sorted by Perf. Cost via the sort option menu (see “Sort Option Menu” on page 87).

  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	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 display, click the left mouse anywhere in a loop's row; typing unique text from the row into the Search field (see "Loop List Search Field" on page 86) does the same thing.

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" on page 90). Selecting a loop affects other displays (see "Selecting a Loop for Analysis" on page 21).

Loop List Search Field

You can use the loop list search field to find a specific loop in the loop list display. For the location of the search field, see Figure 4-11. The field matches any text typed into it to the first instance of that text in the loop list display, and highlights the row of the display in which the text occurs. The search field matches its text against the contents of each column in the loop list display.

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 type **`Enter`**, the highlight moves to the next match. If no match is found, the system beeps, and typing **`Enter`** positions the highlight at the top of the list again.

Sort Option Menu

The sort option menu is the left-most option menu under the loop list search field shown in Figure 4-11. It controls the order in which the loops are displayed in the loop list display.

Figure 4-13 Sort Option Menu

The choices in the sort option menu (see Figure 4-13) have the following effects:

Sort In Source Order
: Orders the loops as they appear in the source file. This is the default setting.

Sort By Performance Cost
: Orders the loops by their performance cost (from greatest to least) as calculated by the Workshop Performance Analyzer. You need to have invoked the Performance Analyzer from the current session of the Parallel Analyzer View to make use of this option. See “Launch Tool Submenu” on page 71 for information on how to open the Performance Analyzer from the current session of the Parallel Analyzer View.

Show Loop Types Option Menu

The show loop types option menu is the center option menu under the loop list search field shown in Figure 4-11. It controls what kind of loops are displayed for each file and subroutine in the loop list display.

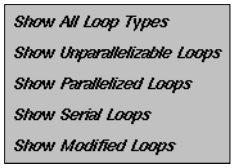

Figure 4-14 Show Loop Types Menu

The options in the show loop types menu (see Figure 4-14) have the following effects:

Show All Loop Types
: Default setting

Show Unparallelizable Loops
: Show only loops that could not be parallelized (runs serially as a result)

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

Filtering Option Menu

The filtering option menu is the right-most option menu under the loop list search field shown in Figure 4-11. It lets you display only those loops contained within a given subroutine or source file.

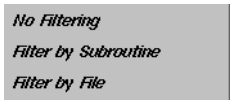

Figure 4-15 Loop Filtering Option Menu

The menu 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 text field that appears above the option menu. Only loops contained in that subroutine are displayed in the loop list display.

Filter By File
: Lets you enter a Fortran source filename into a filtering text field that appears above the option menu. Only loops contained in that file are displayed in the loop list display.

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 List Buttons

The loop list window contains these display control buttons:

Source — Opens the Source View window, with the source file containing the loop currently selected (double-clicked) in the loop list display. 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 Transformed Source Windows” on page 109.

Transformed Source — Opens a Transformed Source window, with the compiled source file containing the loop currently selected (double-clicked) in the loop list display. 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 Transformed Source Windows” on page 109.

Next Loop — Selects the next loop in the loop list display. 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 display. 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 so that the changes can be incorporated into the Fortran source. The display is divided into several information blocks displayed in a scrolling list as shown in Figure 4-16.

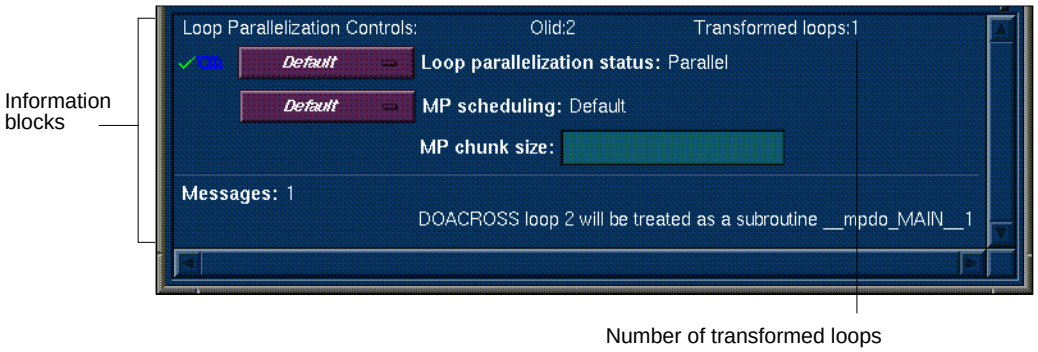

Figure 4-16 Loop Information Display

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

Highlighting Buttons

A highlighting button (light bulb, see Figure 4-17) 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 Transformed Source Windows” on page 109)
- if a directive appears in the options menu next to the highlight button, presents details about directive clauses in a Parallelization Control View (see “Parallelization Control View” on page 97).

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 under the text “Loop Parallelization Controls.” These sections in the loop information display contain controls that allow you to place parallelization assertions and directives in your code (see Figure 4-17). Recall that you have other controls available through the Operations menu (see “Operations Menu” on page 77).

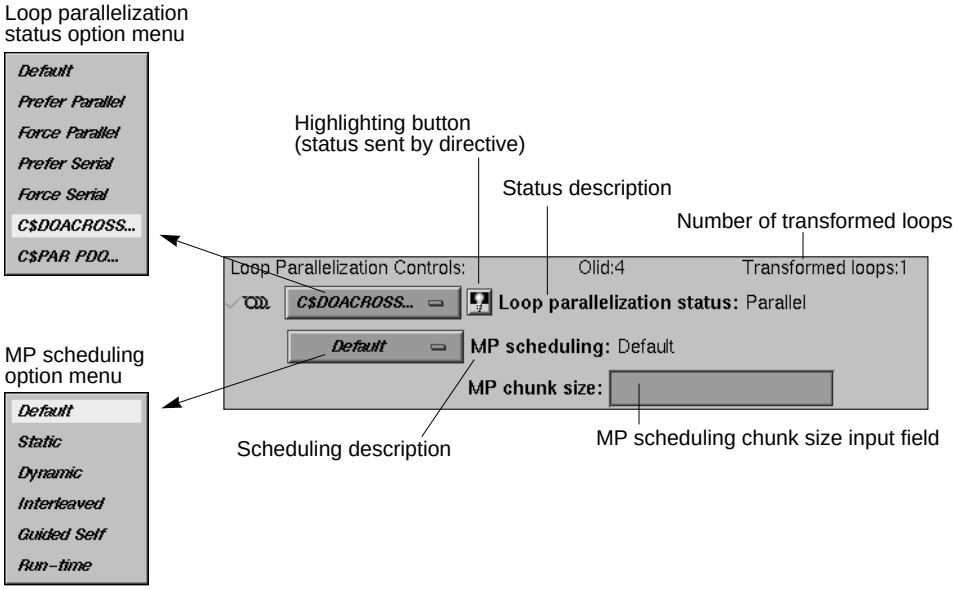

Figure 4-17 Parallelization Controls

Loop Parallelization Status Option Menu

The loop status option menu lets you alter a loop's parallelization scheme. To the right of the option menu is a description of the current loop status as implemented in the transformed source. A small highlighting button appears to the left of this description if the status was set by a directive (shown in Figure 4-17).

The menu choices are as follows:

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$DOACROSS...	Adds the directive C$DOACROSS. Selecting this item opens the Parallelization Control View. See "Parallelization Control View" on page 97 for more information.
C$PAR PDO...	Launches the Parallelization Control View, which allows you to manipulate the PCF scheduling clauses for the Parallel-DO and to set each of the referenced variables as either region-default or last-local. A Parallel-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, you invoked *cvpav* with the **-ro True** option, or the loop comes from an included file. So in some cases you are not allowed to change the menu setting.

Table 4-3 lists the assertions that you control from the Loop Parallelization Controls. For more information about these directives, see Chapter 3 in the *MIPSpro Automatic Parallelizer Programmer's Guide.*

Table 4-3 Assertions Accessed From the Loop Parallelization Controls

Assertion or Directive	Effect on Compilation	Menu Option
C*$*ASSERT DO PREFER (CONCURRENT)	Attempt to parallelize the selected loop. If not possible, try each nested loop.	Prefer Parallel
C*$*ASSERT DO (CONCURRENT)	Parallelize the loop; ignore possible data dependencies.	Force Parallel
C*$*ASSERT DO PREFER (SERIAL)	Do not parallelize the loop.	Prefer Serial
C*$*ASSERT DO (SERIAL)	Do not parallelize the loop.	Force Serial

Table 4-4 lists the directives that you control from the Loop Parallelization Controls. For more information, see Chapter 5 in the *MIPSpro Fortran 77 Programmer's Guide*

Table 4-4 Directives Accessed From the Loop Parallelization Controls

Assertion or Directive	Effect on Compilation	Menu Option
C$DOACROSS	Parallelize the loop, ignore automatic parallelizer.	C$DOACROSS... (opens dialog box to control effects of the directive)
C$PAR PDO	Assign each loop iteration to a different thread, ignore automatic parallelizer.	C$PAR PDO... (opens dialog box to control effects of the directive)

MP Scheduling Option Menu: Directives for All Loops

The MP scheduling option menu 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 field to enter the value (see "MP Scheduling Chunk Size Field" on page 95.

These directives affect the current loop *and all subsequent loops* in a source file. For more information, see Chapter 5 in the *MIPSpro Fortran 77 Programmer's Guide*. For control over a single loop, use the C$DOACROSS directive clauses MP_SCHEDTYPE and CHUNK (see "Parallelization Control View MP Scheduling Option Menu: Clauses for One Loop" on page 103).

The menu 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 assigns one to each processor.
Dynamic	Divides iterations of the selected loop among the processors by dividing them into pieces of size 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 CHUNK and the execution of those pieces is interleaved among the processors. For example, if there are four processors and 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. By parceling out relatively large pieces at the start and relatively small pieces toward the end, the idea is to achieve good load balancing while reducing the number of entries into the critical section.
Run-time	Lets you specify the scheduling type at run time.

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

MP Scheduling Chunk Size Field

Below the scheduling description is an input 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 (see Figure 4-18).

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 4-18 MP Chunk Size Input Field Changed

Be aware of the following when you use the chunk size field:

- Your entry should be syntactically correct, although it is not checked.
- The background color indicates that you cannot make changes if you are looking at a read-only file, if you invoked *cvpav* with the **-ro True** option, or the loop comes from an included file; in some other cases you are not allowed to change the value.

Obstacles to Parallelization Display

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 4-19 illustrates a set of messages describing an obstacle. Each message has a highlight button directly to its left to indicate the troublesome line(s) in the Source View window, opening the window if necessary. If appropriate, the referenced variable or function call is highlighted in a contrasting color.

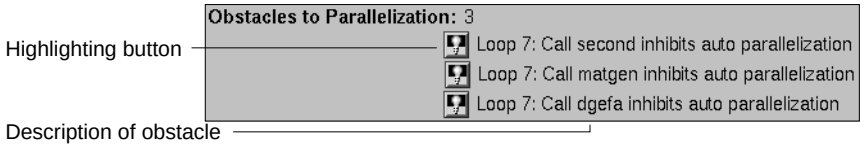

Figure 4-19 Obstacles Information Block

Assertions and Directives Display

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 runs on the source. Assertions provide the compiler with additional information about the source code that can sometimes improve optimization.

When appropriate, an assertion or directive appears with an option menu that allows you to “Keep”, “Delete”, or (if you compile o32) “Reverse” it. Figure 4-20 shows an assertion block and its option menu.

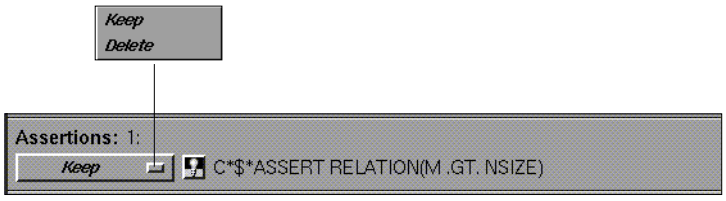

Figure 4-20 Assertion Information Block and Options for n32 and n64 Compilation

Assertions and directives that govern loop parallelization or scheduling do not have associated option menus; those functions are controlled by the loop status option menu and the MP scheduling option menu (see “Loop Parallelization Controls in the Loop Information Display” on page 91).

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. Some messages have associated buttons that highlight sections of the selected loop in the Source View window.

Other Views

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	Contains a single “Close” command that closes the corresponding view
Help menu	Provides access to the online help system (see “Help Menu” on page 82 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, PCF construct, or routine. It can be opened from one of the following:

- The Views menus, which gives basic information about the loop (see “Views Menu” on page 75).
- The Loop parallelization controls option menu when “C$DOACROSS...” or “C$PAR PDO...” is selected. This provides controls for clauses you can append to these directives.

Figure 4-21 shows the view when it is launched from the Views menu with the loop status option menu set to “Default”; this is the display for loops without directives.

Features that appear when the view is opened from the loop parallelization controls option menu when “C$DOACROSS...” or “C$PAR PDO...” is selected are discussed in the following:

- “Adding C$DOACROSS... or C$PAR PDO... Clauses” on page 99.
- “Parallelization Control View MP Scheduling Option Menu: Clauses for One Loop” on page 103
- “Parallelization Control View Variable List: Option Menus” on page 103

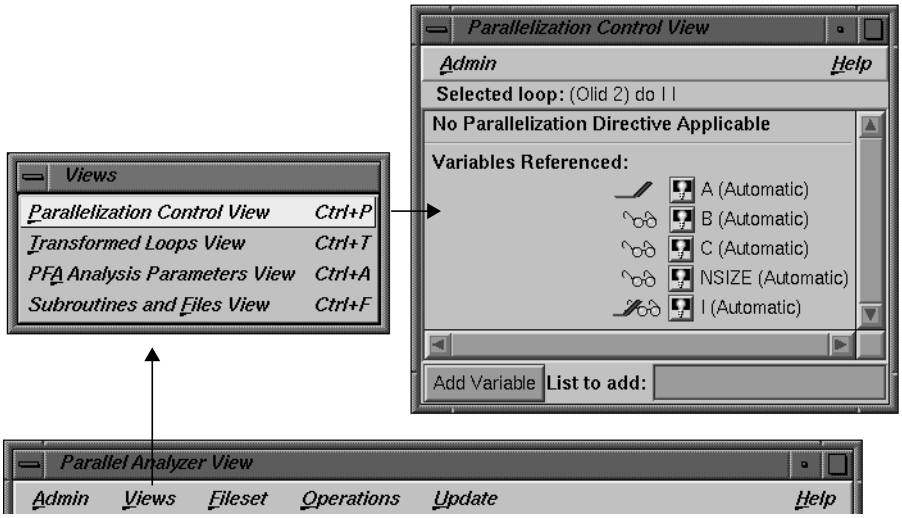

Figure 4-21 Parallelization Control View Without Applicable Directive

Common Features of the Parallelization Control View

Independent of where you are when open the Parallelization Control View, these elements appear in the window (see Figure 4-21):

Admin menu — Contains only one selection, “Close.”

Selected loop — Contains the Olid of the loop, and the inforation about the loop from the Loop-ID and Variable columns of the loop information display.

Directive Information — If a directive is applicable to the loop, lists directive, clauses, and parameter values (see Figure 4-22 and Figure 4-23).

Variables Referenced list

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" on page 110 (or pull down the Icon Legend from the Admin menu in the Parallel Analyzer View) for an explanation of these icons.

For discussion of added option menus that appear if the view is opened from the loop parallelization controls option menu when "C$DOACROSS..." or "C$PAR PDO..." is selected, see "Parallelization Control View Variable List: Option Menus" on page 103.

Add Variable button

Located at the bottom of the window frame, this button allows you to add new variables to a loop.

"List to add" text field

Located at the bottom of the window frame, this 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.

Adding C$DOACROSS... or C$PAR PDO... Clauses

Fields that allow you to specify clauses for C$DOACROSS or C$PAR PDO directives appear in addition to the fields described in "Common Features of the Parallelization Control View" on page 98, if you open the Parallelization Control View from the loop parallelization controls option menu when either "C$DOACROSS..." or "C$PAR PDO..." is selected (see Figure 4-22 and Figure 4-23).

Most of the clauses are the same for these two modes of access. However, notice that in the upper portions of the windows, Figure 4-22 and Figure 4-23 have one unique clause that does not appear in the other figure: "Condition for parallelization" appears when you open the view from "C$DOACROSS..." (Figure 4-22) and "Synchronization at end of construct" when you open from "C$PAR PDO..." (Figure 4-23).

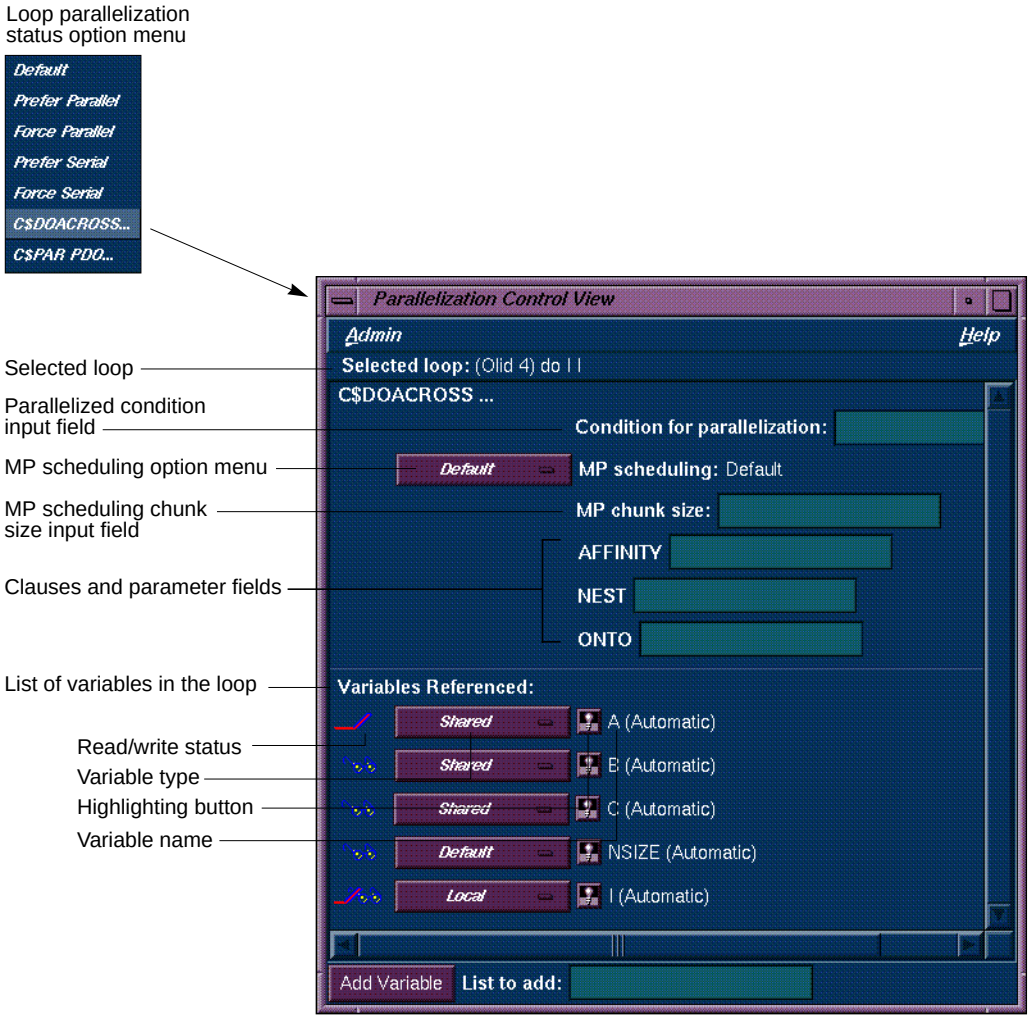

Figure 4-22 C$DOACROSS Parallelization Control View

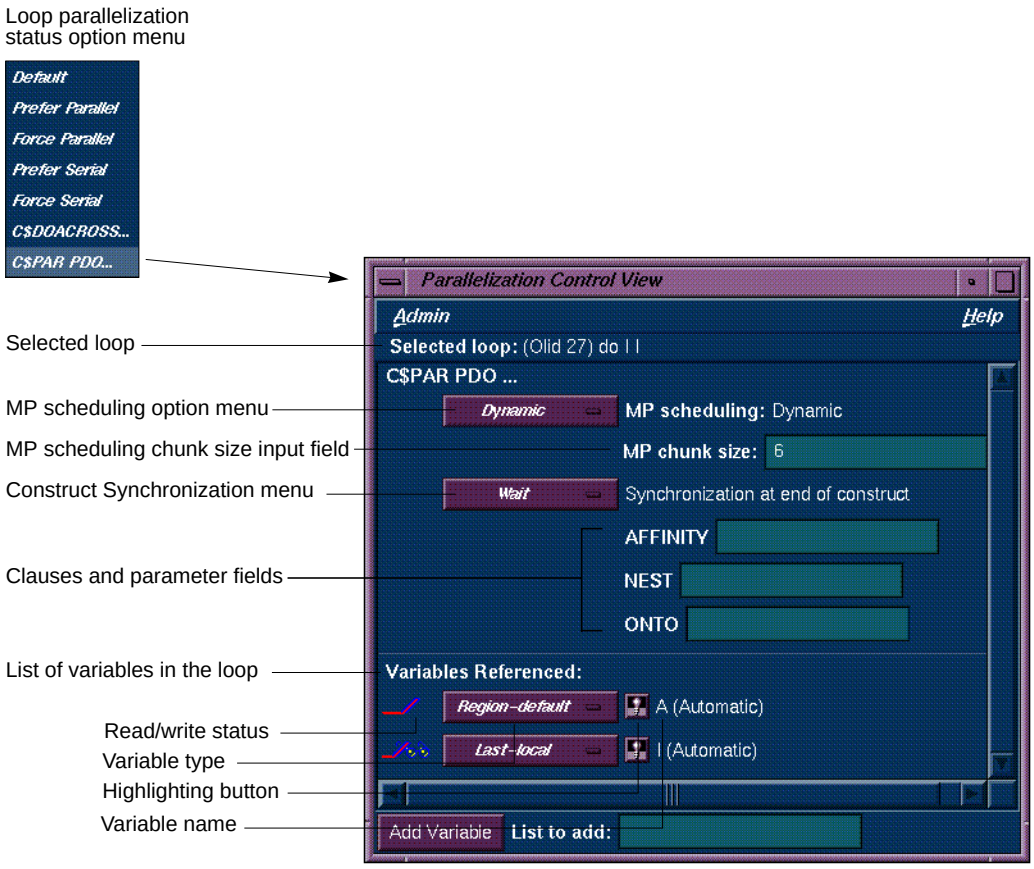

Figure 4-23 C$PAR PDO Parallelization Control View

These are the fields that appear in addition to those discussed in “Common Features of the Parallelization Control View” on page 98:

“Condition for parallelization” text field (“C$DOACROSS...” only)
: Allows you to enter a Fortran conditional statement (for example, `NSIZE .GT. 83`). 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 Scheduling menu
: Allows you to alter a loop’s scheduling scheme by changing the MP_SCHEDTYPE clause. See “Parallelization Control View MP Scheduling Option Menu: Clauses for One Loop” on page 103 for further information.

“MP chunk size” text field
: Allows you to set the CHUNK size for the scheduling scheme you select. For further information, see “MP Scheduling Chunk Size Field” on page 95.

Synchronization Construct menu (“C$PAR PDO...” only)
: Allows you to set the C$PAR END PDO directive at the end of the construct to either “Wait” or “No Wait.”

AFFINITY
: Allows you to specify the parameters for the affinity scheduling clause.

  There are two types of affinity scheduling (for more details and syntax, see Chapter 6 of 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. Recall that 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 Chapter 6 of 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 Chapter 6 of the *MIPSpro Fortran 77 Programmer’s Guide*.

Parallelization Control View MP Scheduling Option Menu: Clauses for One Loop

The Parallelization Control View contains an MP scheduling option menu if it is opened from the loop parallelization controls option menu with either ‘‘C$DOACROSS...’’ or ‘‘C$PAR PDO...’’ selected.

The options that appear have the same names as those for the MP scheduling options in the loop information display, discussed in “MP Scheduling Option Menu: Directives for All Loops” on page 94. However, the option menu in the parallelization control view affects the MP_SCHEDTYPE and CHUNK clauses in the C$DOACROSS directive, and so affects only the currently selected loop. Recall that the MP scheduling option menu in the loop information display affects the placement of the C$MP_SCHETYPE and C$CHUNK directives and thus all subsequent loops.

Except for this difference in scope, the effects of both option menus are the same; for a description, see “MP Scheduling Option Menu: Directives for All Loops” on page 94. For more information, see Chapter 5 in the *MIPSpro Fortran 77 Programmer’s Guide*.

Parallelization Control View Variable List: Option Menus

If the Parallelization Control View is opened from the loop parallelization controls option menu when either ‘‘C$DOACROSS...’’ or ‘‘C$PAR PDO...’’ is selected, each variable listed in the lower portion of the view appears with an option menu. The menu allows you to append a clause to the directive, allowing 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” on page 98. The variable option menu is between the highlight button and the read / write icon.

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

Depending on which directive is relevant, the menus are as follows:

- If the view is opened from the loop parallelization controls option menu when "C$DOACROSS..." is selected, these are the options:

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, min, or max computation of the variable can be done partially in each thread and then combined afterwards.

- If the view is opened from the loop parallelization controls option menu when "C$PAR PDO..." is selected, these are the options:

Region-default	Uses the control established by the compiler for the parallel region.
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.

Parallelization Control View Variable List: Storage Labeling

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

Automatic	The variable is local to the routine, 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 display is rewritten by the compiler into one or more *transformed loops* (see Figure 4-24).

To open this view, pull down the Views menu of the Parallel Analyzer View and choose “Transformed Loops View” (see “Views Menu” on page 75).

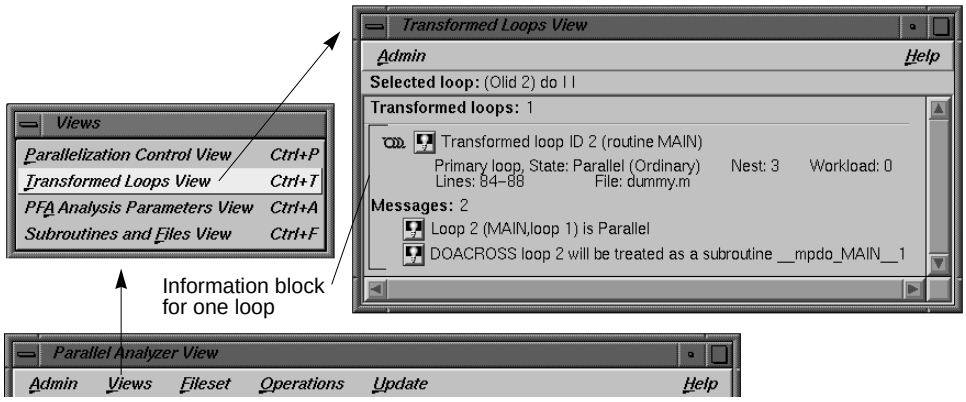

Figure 4-24 Transformed Loops View

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 (if clicked, highlights the transformed 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, that is, directly transformed from the selected original loop or transformed from a different original loop, but incorporating 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, 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.

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 the PFA Analysis Parameters View, choose it from the View menu in the main window (see Figure 4-24).

When you update a source file, any PFA parameters you alter are changed for that file (see Figure 4-25). When you change a parameter, the upper right corner of the field window "turns down," as discussed in "MP Scheduling Chunk Size Field" on page 95.

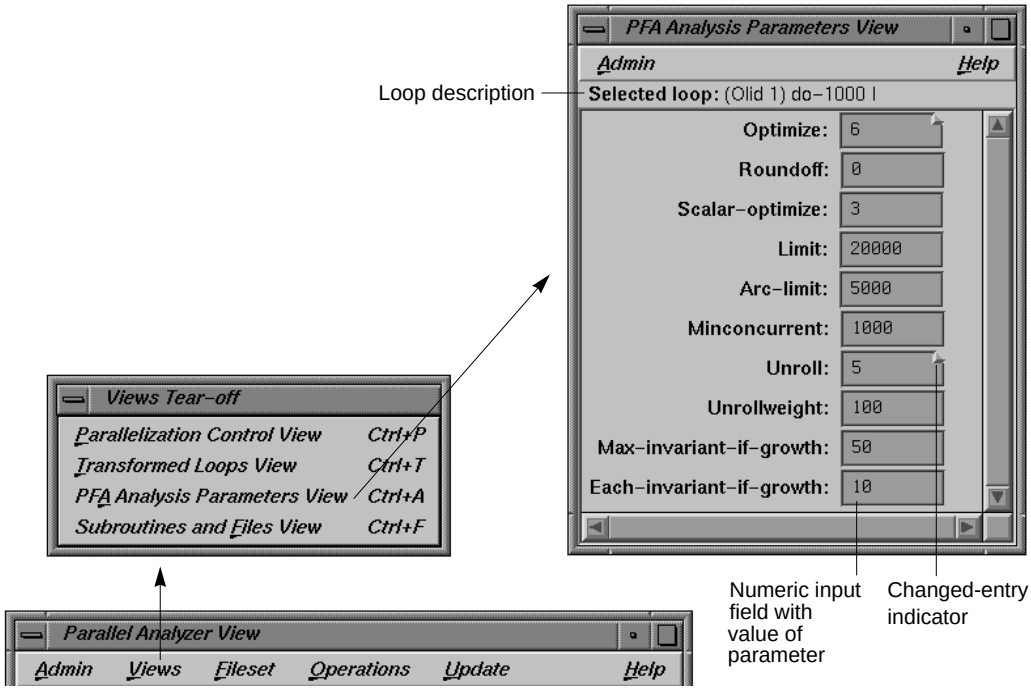

Figure 4-25 PFA Analysis Parameters View

A full explanation of the PFA parameters listed in this view can be found in Chapter 4, "Customizing PFA Execution," in the *POWER Fortran Accelerator User's Guide.*

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 (see Figure 4-26). 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.

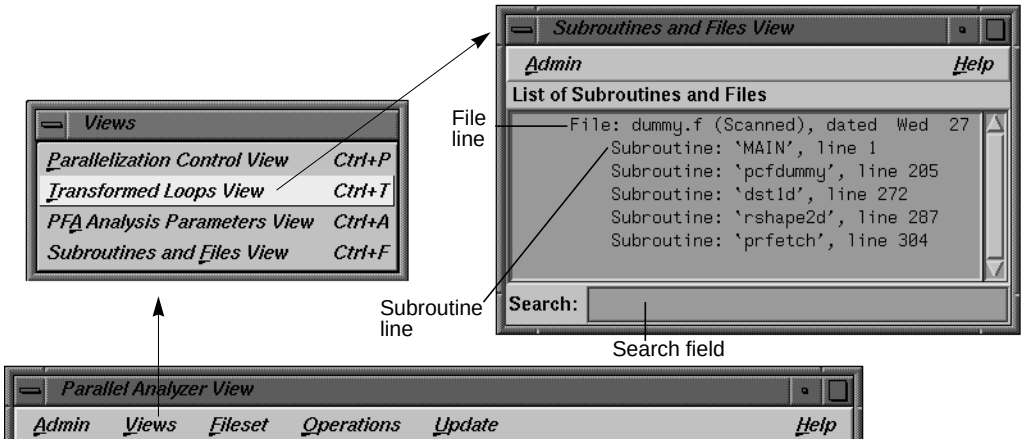

Figure 4-26 Subroutines and Files View

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 “Loop List Search Field” on page 86.

You can select items in the list for two purposes:

- To save changes to a selected file: click the filename and use the Update menu at the top of the Parallel Analyzer View main window to choose the option “Update Selected File” (see “Update Menu” on page 80).
- To select a file or subroutine for loop list filtering (see “Filtering Option Menu” on page 88): 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 field, which you can use to search the list of files and subroutines.

Source View and Transformed Source Windows

The Source View window and the Transformed Source window together present views of the source code before and after compiler optimization (see Figure 4-27). 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 (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” on page 71), any displayed sources files known to the Performance Analyzer are annotated with performance data.

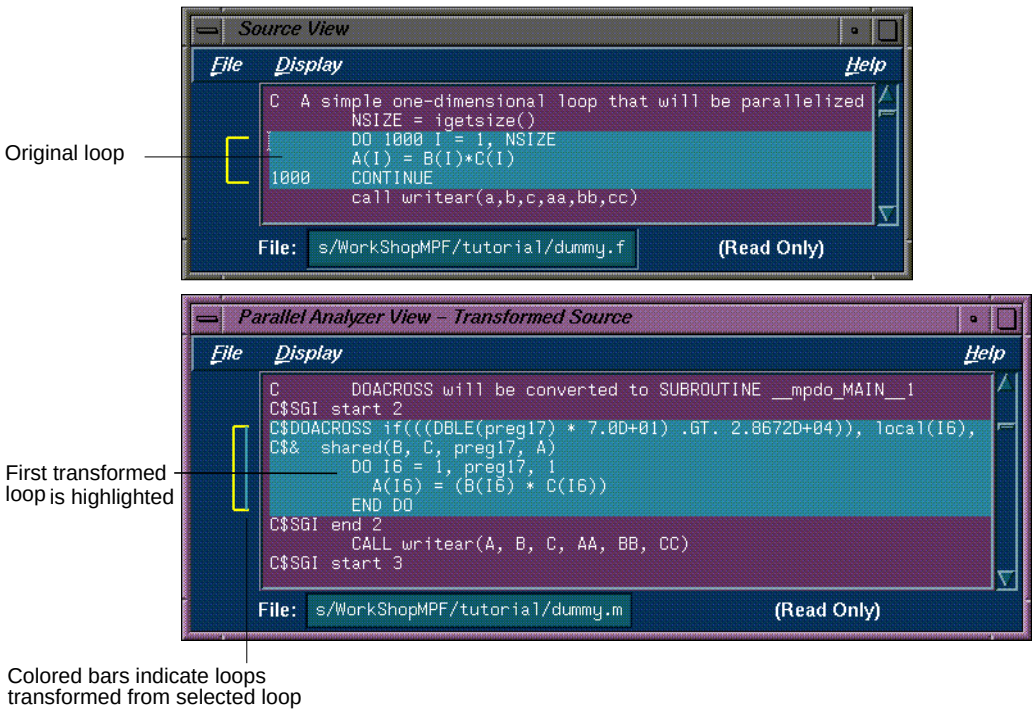

Figure 4-27 Original and Transformed Loop Source Windows

Icon Legend

The Icon Legend window is a key to the meanings of the icons that appear in the Parallel Analyzer View, the Transformed Loops View, the Subroutines and Files View, and Custom DOACROSS Dialog box. See Figure 4-28.

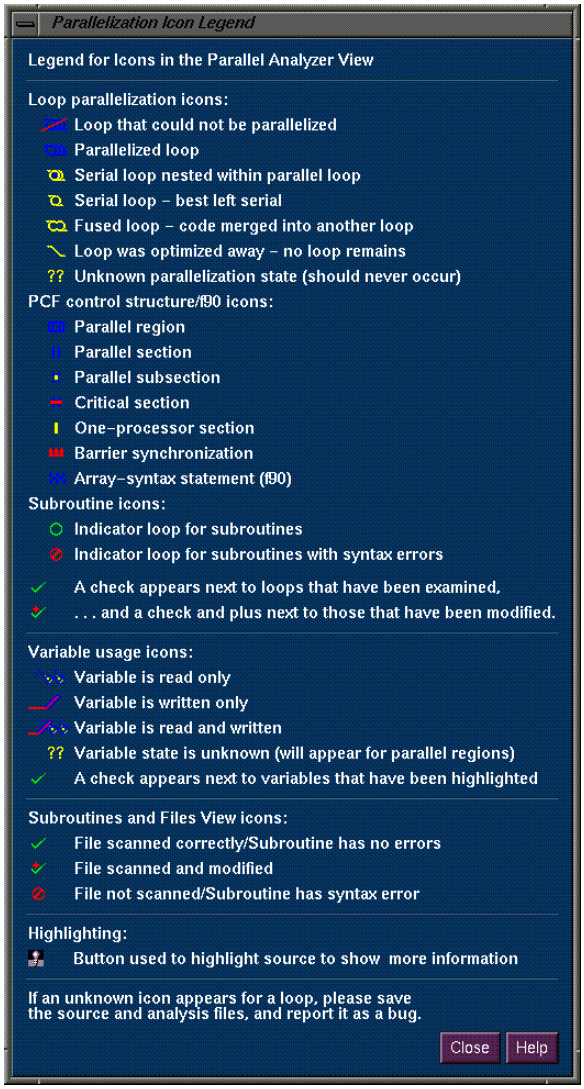

Figure 4-28 Parallelization Icon Legend

Index

A

Add Assertion submenu, 79
Add Directive submenu, 79
Add File command, 76
Add Files from Executable command, 77
Add Files from Fileset command, 77
adding an assertion, 46
adjustment button, resize loop list display, 12, 84
Admin menu, Parallel Analyzer View, 69
AFFINITY, 102
analysis files, xiii
assertions, 37, 43, 96
  adding from Operations menu, 77
  adding from Parallelization Controls menu, 91
  deleting, 48

B

brackets
  colors, 17
  loop, 26
  source windows and, 109
bugs, reporting, 86
Build Manager, 48
  launching, 72
button
  adjust loop list display, 84
  highlighting, 90
  Next Loop, 89
  Previous Loop, 89
  Source, 89
  Transformed Source, 89

C

C*$*ASSERT_CONCURRENT CALL, 78
C*$*ASSERT DO (CONCURRENT), 93
C*$*ASSERT DO (SERIAL), 93
C*$*ASSERT DO PREFER (CONCURRENT), 93
C*$*ASSERT DO PREFER (SERIAL), 93
C*$*ASSERT PERMUTATION, 78
C*$*CONCURRENTIZE, 78
C*$*NOCONCURRENTIZE, 78
C*$*PREFETCH_REF, 78
caliper setting in Performance Analyzer, 84
C$CHUNK, 94, 95
C$COPYIN, 78
C$DISTRIBUTE, 56, 78
C$DISTRIBUTE_RESHAPE, 57
C$DOACROSS, 56, 93
  clauses, 99
  NEST, 57
C$DOACROSS..., 43, 92
C$DYNAMIC, 78
changed-entry indicator, 95
check mark, 85
CHUNK, 102
CHUNK size, 95, 103
closing all windows, Project menu, “Exit”, 73
C$MP_SCHEDTYPE, 94
colors, brackets and icons, 17
command line options, 4
compiling for Parallel Analyzer View, 3

concurrent call assertion, 46
conditional statement input field, DOACROSS, 102
conventions, font, for manual, xv
C$PAR END PDO, 102
C$PAR PDO, 52, 93
  clauses, 99
C$PAR PDO..., 92
C$REDISTRIBUTE, 78
Custom DOACROSS Dialog, 29
cvpav
  compiling for, 3
  opening editor, 49, 81
  starting, 4

D

data dependence, 36
Default, 92
  C$MP SCHEDTYPE mode, 94
  DOACROSS MP_SCHEDTYPE, 103
default directory for file writing, 69
Delete, 96
Delete All Files command, 76
Delete Selected File command, 76
deleting an assertion, 48
demonstration directory, 6
directives, 43, 96
  adding from MP scheduling option menu, 94
  adding from Operations menu, 77
  adding from Parallelization Controls menu, 91
  deleting, 48
DOACROSS, custom, 43
documentation, recommended reading, xiv
doubly nested loops, 41
Dynamic
  C$MP SCHEDTYPE mode, 94
  DOACROSS MP_SCHEDTYPE, 103

E

Exit command
  Admin menu, 70
  Project menu, 73
exiting, 65
explicitly parallelized loop, 29

F

File, 10
  filtering text field, 16
file
  update, 48
fileset, Add Files from Fileset command, 77
Fileset menu, 76
Filter By File, 88
Filter By Subroutine, 88
filtering
  by file, 16
  by parallelization state, 14
  option menu, 88
    Subroutines and Files View and, 109
  option menus, 13
font conventions, for manual, xv
Force Parallel, 92
Force Serial, 92

G

gdiff, 49
Guided-Self, 103
  Scheduling, C$MP SCHEDTYPE mode, 94

H

Help menu, Parallel Analyzer View, 82
highlighting a loop, 86
highlighting button, 24
  Directives, 90
  Loop parallelization status, 90

I

Iconify command
  Admin menu, 70
  Project submenu, 73
Icon Legend
  command, 70
  dialog box, 110
icons, 10
  check mark, 21
  description, 110
  parallelization, 85
Index... command, 82
indicator bar, 109
input-output operation, 39
installation, 1
interchanged loops, 41
Interleaved
  C$MP SCHEDTYPE mode, 94
  DOACROSS MP_SCHEDTYPE, 103

K

Keep, 96

L

Last-local, Parallelization Control View
  C$DOACROSS LASTLOCAL, 104
  C$PDO LASTLOCAL, 104
Launch Tool submenu, 71
light bulb button, 24
line highlighting, 36, 38
Lines, 10
Lines, loop list heading, 86
linpack, 59
Local, Parallelization Control View
  C$DOACROSS LOCAL, 104
  C$PDO LOCAL, 104
loop
  complex, 41
  detailed information, 17
  examining simple, 28
  information blocks, 24
  information display, 23, 90
  ordinary or interchanged, 106
  primary or secondary, 106
  status, 85
  with obstacles to parallelization, 31
Loop-ID, 10, 86
Loop List
  filtering, 13
  sorting, 13
loop list display, 10, 84
  column headings, 85
Loop Status Menu, 43
loop status option menu, 92

M

main window, 7
  menu bar, 68
make clean, 6, 58, 65
memory, 1
Messages, 97
  transformed loop, 106
messages
  obstacles to parallelization, 31
modifying source files, 42
MP_SCHEDTYPE, 102
MP scheduling chunk size field, 95
MP scheduling option menu, 94
  Custom DOACROSS, 103

N

NEST, 102
Nest, 10, 85
nested loops, 41
  transformed, 106
No Filtering, 88

O

obstacles to parallelization, 31
Obstacle to Parallelization, 95
Olid, 10
  loop list heading, 86
On Context command, 82
ONTO, 102
Operations menu, 77
option menu
  assertions and directives, 96
  filtering, 88
  loop status, 92
  MP scheduling, 94
  show loop types, 87
  variable type, Parallelization Control View
    DOACROSS, 103
original loop ID. *See*Olid
Original Source window, 26

P

Paralellization Controls, 91
Parallel Analyzer, launching, 72
Parallel Analyzer View
  menu bar, 68
  Source View, 17
parallelization
  controls, 23
  status option menu, 14
Parallelization Control View
  brought up by a highlight button, 56
  command, 75
  DOACROSS variable option menu, 103
  loop status option menu and, 92
Parallelization Icon Legend, 110
Perf. Cost. *See* performance
performance, 1
  cost per loop, 85
  data, 109
  information line, 23
Performance Analyzer, 59
  launching, 72
  performance experiment line, 84
  source windows and, 109

performance experiment demo, 60
performance experiment line, 84
permutation vector, 40
PFA Analysis Parameters View, 106
  command, 75
plus sign, 85
Prefer Parallel, 92
Prefer Serial, 92
premature exit, 39
primary loop, 106
Project submenu, 73
Project View command, 73

R

Raise command, 70, 73
recurrence, 36
red plus sign, 85
reduction, 38
Reduction, Parallelization Control View,
    C$DOACROSS REDUCTION, 104
Remap Paths... command, 73
Rescan All Files command, 76
resize loop list display, 12, 84
Reverse, 96
round-off, 38
Run-time
  C$MP SCHEDTYPE mode, 94
  DOACROSS MP_SCHEDTYPE, 103

S

sample session
  analyzing loops, 5
  Performance Analyzer, 59
Save As Text command, 69
Search field
  Loop List, 46, 86
  Subroutines and Files View, 109
searching source code, 18
secondary loop, 106
selecting a loop, 21, 86
Shared, Parallelization Control View C$DOACROSS
    SHARE, 104
show loop types option menu, 87
sorting
  by performance cost, 64, 85
  option menu, 87
Source button, 89
source files
  manipulating fileset, 76
  modifying, 42
  undoing changes, 77
  updating, 48, 80
  viewing, 17
Source View window, 109
  opening, 89
Static (or Simple)
  C$MP SCHEDTYPE mode, 94
  DOACROSS MP_SCHEDTYPE, 103
Static Analyzer, launching, 72
status line, 84
storage classes for variables, 104
Subroutine, 10
Subroutine, loop list heading, 86
Subroutine and Files View, 16
  keyboard shortcut, 16
subroutine calls, 40
Subroutines and Files View, 108
  command, 75
  Delete Selected File command and, 76
  filtering text field and, 88

T

Technical Assistance Center, 2
Text.out, default file name, 69
token highlighting, 38
transformed
  loop, 26, 106
    selecting, 26
  source files, viewing, 19
Transformed Loops View, 105
  command, 75
  using, 25
Transformed Source, 26
  window, opening, 89
Transformed Source button, 89
Transformed Source window, 109
turned-down corner of field, 95

U

Undo All Changes command, 79
updating files, 48, 49

V

Variable, 10
  storage class, 104
Variable, loop index, 86
variable highlighting, 36
vi, 49
viewing source, 17
Views menu, 75
  other views, 97

W

windows, closing all, Project menu, “Exit”, 73
WorkShop, 59
  Debugger, launching, 72

X

.Xdefaults, 49, 81
X resources, 4
xwsh, 49, 81

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2603-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389