ProDev ProMP User’s Guide

007–2603–005

What's New in This Guide

ProDev ProMP User's Guide 007–2603–005

New Features Documented:

Support for Fortran 90 and C programs has been added.

Record of Revision

Version	*Description*
Revision level	1993 Original Printing.
2.9	April 1999 This release adds support for programs written in C and Fortran 90.

Contents

Page

Page

Figures

Page

Page

Preface

This publication documents the ProDev ProMP release running on IRIX systems.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
...	Ellipses indicate that a preceding element can be repeated.

Getting Started With the Parallel Analyzer View [1]

This chapter helps you get the ProDev ProMP parallel analyzer view running on your system. It contains the following sections:

- Setting up your system, see Section 1.1, page 1.
- Running the parallel analyzer view: general features, see Section 1.2, page 2.
- Tutorials, see Section 1.3, page 4.

Note: This product was formerly called WorkShop Pro MPF.

1.1 Setting Up Your System

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

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

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

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

```
% versions
```

If the items mentioned in this section are not installed, consult your sales representative, or in the United States 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.

If you have all the software and memory you need, you can install the ProDev ProMP software.

- For general instructions about software installation, consult the man pages `inst`(1M) and `swmgr`(1M), and the manual, *IRIX Admin: Software Installation and Licensing*.
- See also *Developer Magic: ProDev Pro MP Release Notes* for specific installation instructions.

The executable is `cvpav`(1), which is installed in `/usr/sbin`.

1.2 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 the parallel analyzer view.

1.2.1 Compiling a Program for the Parallel Analyzer View

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

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

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

The command-line options have the following effects:

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

See the *MIPSpro Fortran 77 Programmer’s Guide*, *MIPSpro Compiling and Performance Tuning Guide*, and the `f90`(1) or `f77`(1) man page for more information.

Note: The `cvpav` command assumes that the **`-apo keep`** option was used on each of the Fortran source files named in a single executable or file specifying several executables. If this is not the case, a warning message is posted, and the unprocessed files are marked by an error icon within the parallel analyzer's subroutines and files view, see Section 6.6.4, page 160.

1.2.1.1 Generating Other Reports

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

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

These reports are text files that can be used for analysis. For more detailed information, see *MIPSpro Auto-Parallelizing Option Programmer's Guide*.

1.2.1.2 OpenMP and PCF Directive Support

The MIPSpro auto-parallelizing Fortran compilers support OpenMP directives, unless you are compiling with the `-o32` option. If you put OpenMP directives in your `o32` code, they are treated as comments rather than being interpreted. For more information on OpenMP directives, see the following:

- Section 2.11, page 56
- The OpenMP Architecture Review Board web site at the following URL: `http://www.openmp.org/`

Although using OpenMP directives is recommended, the auto-parallelizing compilers still support PCF directives. For information on analyzing loops containing PCF directives, see Appendix A, page 165.

1.2.2 Reading Files With the Parallel Analyzer View

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

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

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

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

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

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

Note: If you receive a message related to licensing when you start `cvpav`, refer to Chapter 7 in the *ProDev Pro MP Release Notes*. To access the notes, enter the `grelnotes`(1) command and choose `Products > ProMP`.

1.3 Tutorials

For a more detailed introduction to the parallel analyzer view, follow one of tutorials provided with the product in the following chapters:

- Examining Loops for Fortran 77 Code, seeChapter 2, page 5.
- Examining Loops for Fortran 90 Code, see Chapter 3, page 67.
- Examining Loops for C Code, see Chapter 4, page 75.
- Using WorkShop With Parallel Analyzer View, see Chapter 5, page 105.
- Examining Loops Containing PCF Directives, see Appendix A, page 165.

Examining Loops for Fortran 77 Code [2]

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-Parallelizing Fortran 77 compiler. For tutorials using other compilers, see the following sections:

- Fortran 90, see Chapter 3, page 67.
- C, see Chapter 4, page 75.

The sample session analyzes demonstration 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 `omp_demo.f` Sample Session, see Section 2.1, page 6.
  - Starting the Parallel Analyzer View Tutorial, see Section 2.3, page 7.
  - Using the Loop List Display, see Section 2.4, page 9.
  - Sorting and Filtering the Loop List, see Section 2.5, page 12.
  - Viewing Detailed Information About Code and Loops, see Section 2.6, page 16.
- 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, see Section 2.7, page 28.
  - Examining Loops With Obstacles to Parallelization, see Section 2.8, page 32.
  - Examining Nested Loops, see Section 2.9, page 44.
  - Modifying Source Files and Compiling, see Section 2.10, page 45.
  - Examples Using OpenMP Directives, see Section 2.11, page 56.
  - Examples Using Data Distribution Directives, see Section 2.12, page 62.
  - Exiting From the `omp_demo.f` Sample Session, see Section 2.13, page 66.

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

For more details about the Parallel Analyzer View interface, see Chapter 6, page 113.

2.1 Setting Up the `omp_demo.f` Sample Session

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

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

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

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

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

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

2.2 Compiling the Sample Code

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

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

Doing this creates the following files:

- `omp_demo.f`: a copy of the demonstration program created by combining the `*.f_orig` files, which you can view with the Parallel Analyzer View (or any text editor), and print

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

For more information about these files, see the *MIPSpro Auto-Parallelizing Option Programmer's Guide*.

2.3 Starting the Parallel Analyzer View Tutorial

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

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

Note: If you receive a message related to licensing, refer to the *ProDev ProMP Release Notes*.

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

  ```
  % make clean
  ```

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

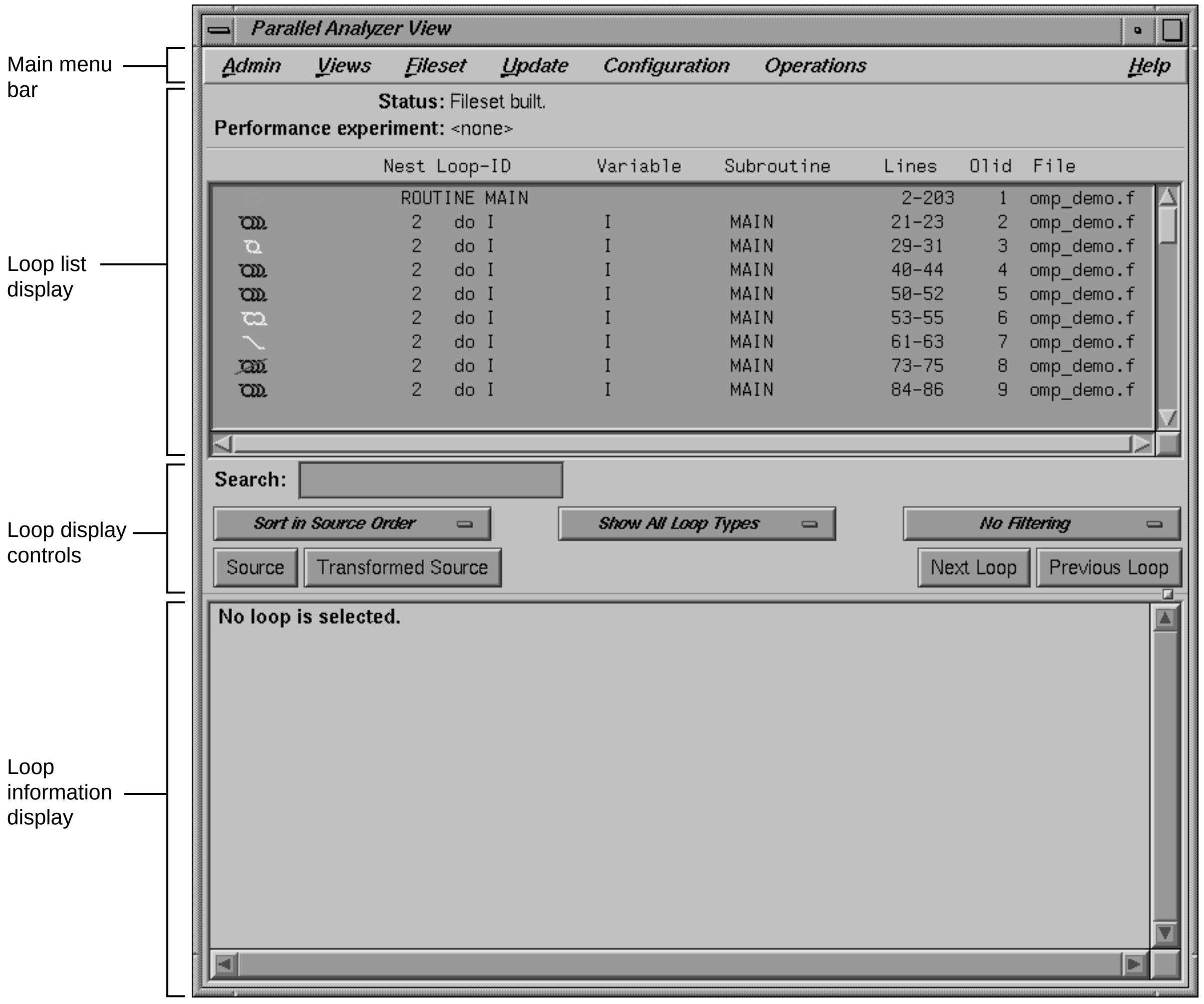
2.3.2 Viewing the Parallel Analyzer View Main Window

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

- Main menu bar, which includes the following menus:
  - `Admin`

- `Views`
- `Fileset`
- `Update`
- `Configuration`
- `Operations`
- `Help`

- List of loops and control structures, which consists of the following:
  - Status information
  - Performance experiment information
  - Loop list
- Loop display controls, which are the following:
  - Search editable text field
  - Sort option button (`Sort in Source Order`)
  - Show loop types option button (`Show All Loop Types`)
  - Filtering option button (`No Filtering`)
  - `Source` and `Transformed Source` control buttons
  - `Next Loop` and `Previous Loop` navigation buttons
- Loop information display



Status: Fileset built.
Performance experiment: <none>

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

No loop is selected.

Figure 1. Parallel Analyzer View Main Window

2.4 Using the Loop List Display

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

2.4.1 Loop List Information Fields

Each loop list entry contains the following fields:

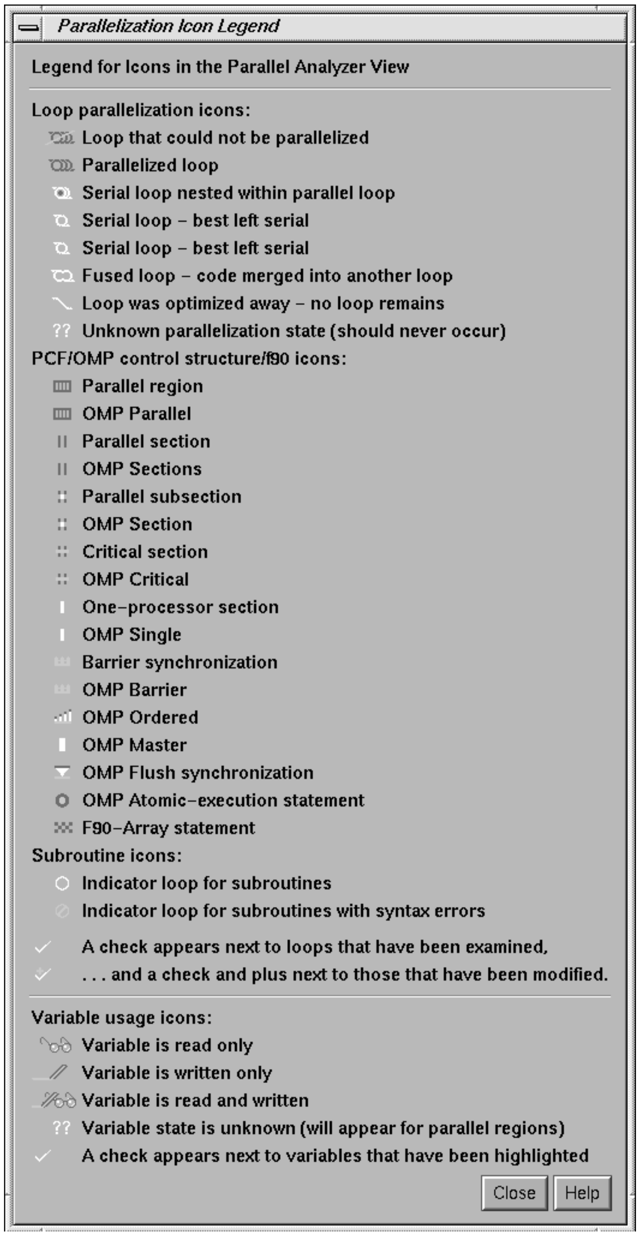
- The icon symbolizes the status of the subroutine or loop.
- The nest field shows the nesting level for the loop.
- The loop-ID gives a description of the loop.
- The variable field indicates the loop index variable.
- The subroutine field contains the name of the subroutine in which the loop is located.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

2.4.2 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 or function.
- The parallelization status of the loop.
- OpenMP control structures.

To understand the meaning of the various icons, choose `Admin` > `Icon Legend...` . (See Figure 2, page 11.)



Parallelization Icon Legend

Legend for Icons in the Parallel Analyzer View

Loop parallelization icons:
- Loop that could not be parallelized
- Parallelized loop
- Serial loop nested within parallel loop
- Serial loop – best left serial
- Serial loop – best left serial
- Fused loop – code merged into another loop
- Loop was optimized away – no loop remains
- ?? Unknown parallelization state (should never occur)

PCF/OMP control structure/f90 icons:
- Parallel region
- OMP Parallel
- Parallel section
- OMP Sections
- Parallel subsection
- OMP Section
- Critical section
- OMP Critical
- One–processor section
- OMP Single
- Barrier synchronization
- OMP Barrier
- OMP Ordered
- OMP Master
- OMP Flush synchronization
- OMP Atomic–execution statement
- F90–Array statement

Subroutine icons:
- Indicator loop for subroutines
- Indicator loop for subroutines with syntax errors

✓ A check appears next to loops that have been examined,
✓ . . . and a check and plus next to those that have been modified.

Variable usage icons:
- Variable is read only
- Variable is written only
- Variable is read and written
- ?? Variable state is unknown (will appear for parallel regions)
- ✓ A check appears next to variables that have been highlighted

Figure 2. The Icon Legend... Window

2.4.3 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. The adjustment button is a small square below the `Previous Loop` button and just above the vertical scroll bar on the right side of the loop information display. (See Figure 61, page 136.) In many of the following figures, the loop list is resized from its original configuration.

2.4.4 Searching the Loop List

The loop list `Search` field allows you to find occurrences of any character in the loop list. You can search for subroutine names, a phrase (such as `parallel` or `region`), or Olid numbers. (See Figure 3, page 12.)

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

2.5 Sorting and Filtering the Loop List

This section describes the loop display controls option buttons. They allow you to sort and filter the loop list, and so focus your attention on particular pieces of your code. As shown in Figure 1, page 9, the buttons are located in the main window, below the loop list display. Figure 3, page 12, shows all of the loop display controls.

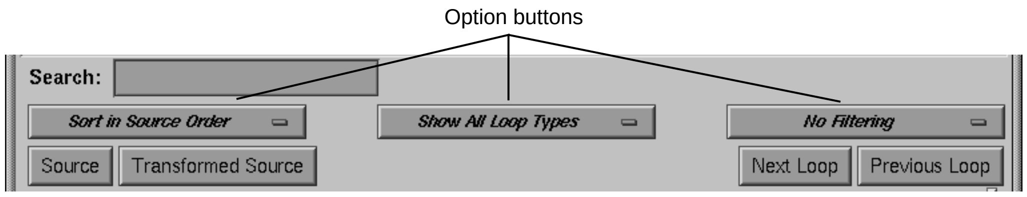


Figure 3. Loop Display Controls

2.5.1 Sorting the Loop List

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

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

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

2.5.2 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 buttons to the right of the sort option button.

2.5.2.1 Filtering the Loop List by Parallelization State

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

Filtering is controlled by the show loop types option button centered below the loop list; the default setting is `Show All Loop Types`, as shown in Figure 4, page 13.

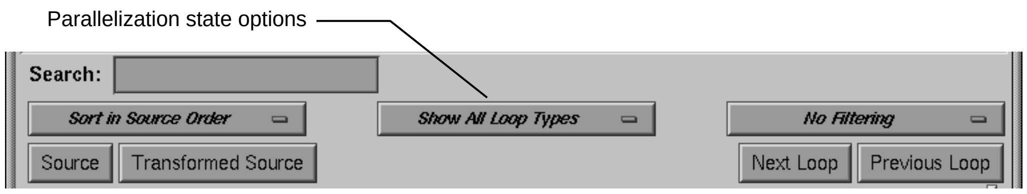


Figure 4. Show Loop Types Option Button

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

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

The second, third, and fourth categories correspond to parallelization icons in the `Icon Legend`... window (see Figure 2, page 11). Making modifications to loops is described in Section 2.10.1, page 46.

To see the effects of these options, choose them in turn by clicking on the option button and selecting each option. If you choose the `Show Modified Loops` option, a message appears that no loops meet the filter criterion, because you have not made any modifications.

2.5.2.2 Filtering the Loop List by Loop Origin

Another way to filter is to choose loops that come from a single file or a single subroutine or function. These are the basic steps:

1. Open a list of subroutines (or functions) and files from which to choose by selecting the `Views` > `Subroutines and Files View` option.
2. Choose the filter criterion from the filtering option button, the right-most option button in the Parallel Analyzer View window. Initially the filter criterion is `No Filtering`. You can filter according to source file or subroutine.

To place filtering information in the editable text field that appears above the option button (Figure 5, page 15), you can do one of the following:

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

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

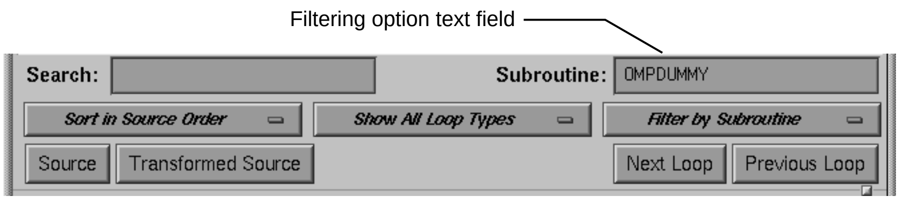
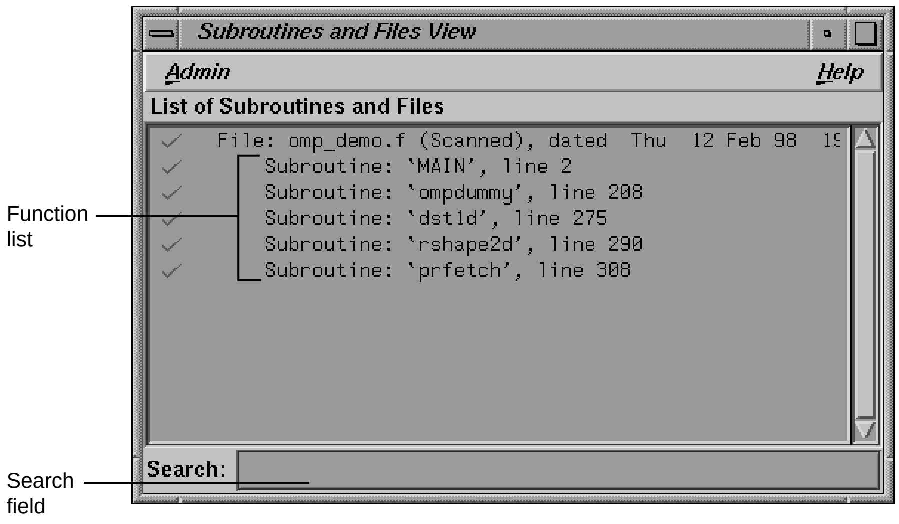


Figure 5. Filtering Option Button

2.5.2.3 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 choosing `Views` > `Subroutines and Files View`. The window opens and lists the subroutines and files in the file set. (See Figure 6, page 15.)



```
File: omp_demo.f (Scanned), dated  Thu  12 Feb 98  1
    Subroutine: 'MAIN', line 2
    Subroutine: 'ompdummy', line 208
    Subroutine: 'dst1d', line 275
    Subroutine: 'rshape2d', line 290
    Subroutine: 'prfetch', line 308
```

Figure 6. Subroutines and Files View

2. Choose `Filter by Subroutine` from the filtering option button (Figure 7, page 16).

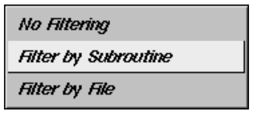


Figure 7. Filtering Option Button

3. Double-click the line for the subroutine `OMPDUMMY()` in the list of the `Subroutines and Files View` window. The name appears in the `Subroutine` filtering option text field (Figure 5, page 15), and the loop list is recreated according to the filter criteria.

   You can also try choosing `Filter by File` with the filtering option button, but this is not very useful for this single-file example.

4. When you are done, display all of the loops in the sample source file again by choosing `No Filtering` with the option button.

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

2.6 Viewing Detailed Information About Code and Loops

This section describes how to examine the following:

- Source code
- Transformed source code
- Details of loop information summarized in the loop list

2.6.1 Viewing Original and Transformed Source

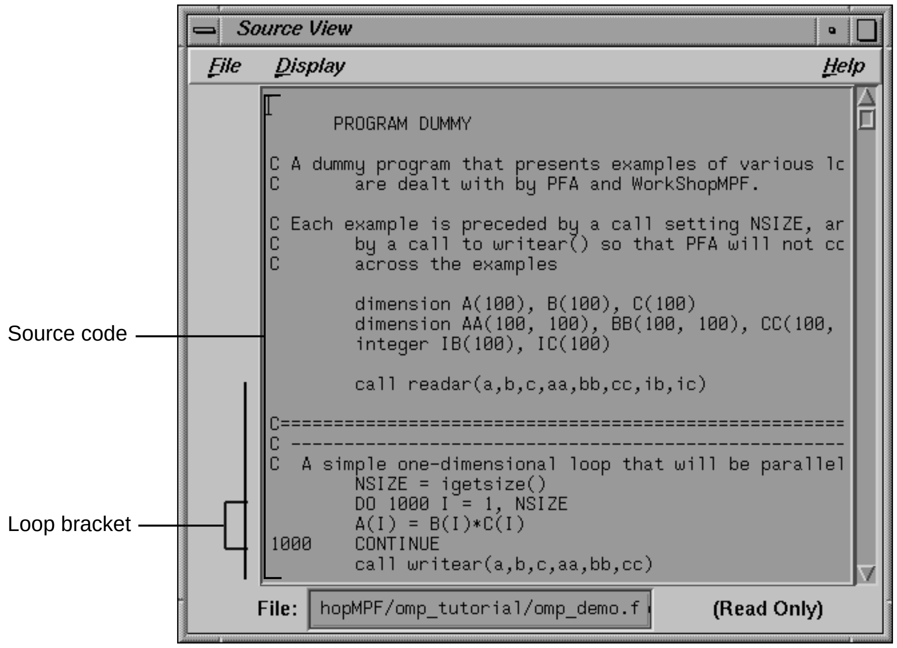
The Parallel Analyzer View gives you views of both your original Fortran source and a listing that mimics the effect on the source as it is transformed by the Auto-Parallelizing compiler.

2.6.1.1 Viewing Original Source

Click the `Source` button on the lower left corner of the loop display controls to bring up the Source View window, shown in Figure 8, page 17.

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 Section 2.6.3, page 20.)

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



```
      PROGRAM DUMMY

C A dummy program that presents examples of various lc
C       are dealt with by PFA and WorkShopMPF.

C Each example is preceded by a call setting NSIZE, ar
C       by a call to writear() so that PFA will not cc
C       across the examples

        dimension A(100), B(100), C(100)
        dimension AA(100, 100), BB(100, 100), CC(100,
        integer IB(100), IC(100)

        call readar(a,b,c,aa,bb,cc,ib,ic)

C=====================================================
C ----------------------------------------------------
C  A simple one-dimensional loop that will be parallel
        NSIZE = igetsize()
        DO 1000 I = 1, NSIZE
        A(I) = B(I)*C(I)
1000    CONTINUE
        call writear(a,b,c,aa,bb,cc)
```

hopMPF/omp_tutorial/omp_demo.f

Figure 8. Source View

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

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

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

For more information about the Source View window, see Section 6.7.1, page 162.

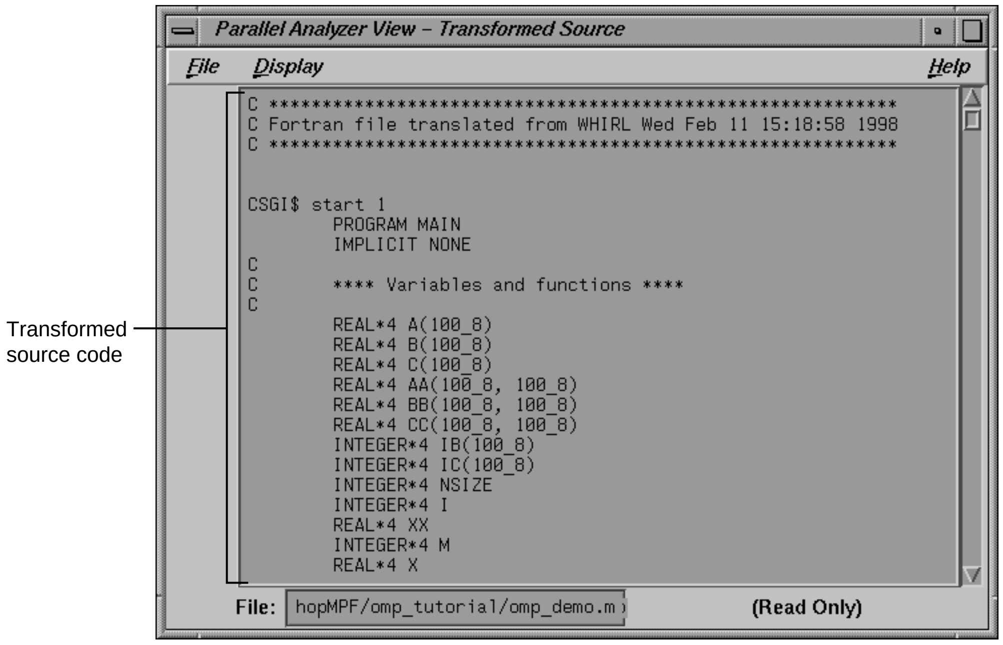
Leave the Source View window open, because subsequent steps in this tutorial refer to the window.

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

2.6.1.2 Viewing Transformed Source

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

Click the `Transformed Source` button in the loop display controls (see Figure 3, page 12). A window labeled Parallel Analyzer View – Transformed Source opens, as shown in Figure 9, page 19.


Parallel Analyzer View – Transformed Source
File
Display
Help
Transformed source code
File:
hopMPF/omp_tutorial/omp_demo.m
(Read Only)

```
C **********************************************************
C Fortran file translated from WHIRL Wed Feb 11 15:18:58 1998
C **********************************************************


CSGI$ start 1
        PROGRAM MAIN
        IMPLICIT NONE
C
C       **** Variables and functions ****
C
        REAL*4 A(100_8)
        REAL*4 B(100_8)
        REAL*4 C(100_8)
        REAL*4 AA(100_8, 100_8)
        REAL*4 BB(100_8, 100_8)
        REAL*4 CC(100_8, 100_8)
        INTEGER*4 IB(100_8)
        INTEGER*4 IC(100_8)
        INTEGER*4 NSIZE
        INTEGER*4 I
        REAL*4 XX
        INTEGER*4 M
        REAL*4 X
```

Figure 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 does not 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 Section 6.7.1, page 162.

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

- Parallel Analyzer View
- Source View
- Transformed Source

2.6.2 Navigating the Loop List

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

- Scrolling through the loop list using one of these:
  - Scroll bar.
  - Page Up and Page Down keys (the cursor must be over the loop list).
  - `Next Loop` and `Previous Loop` buttons.
- Searching for the Olid number using the Search field (see Section 2.4.4, page 12).

2.6.3 Selecting a Loop for Analysis

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

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

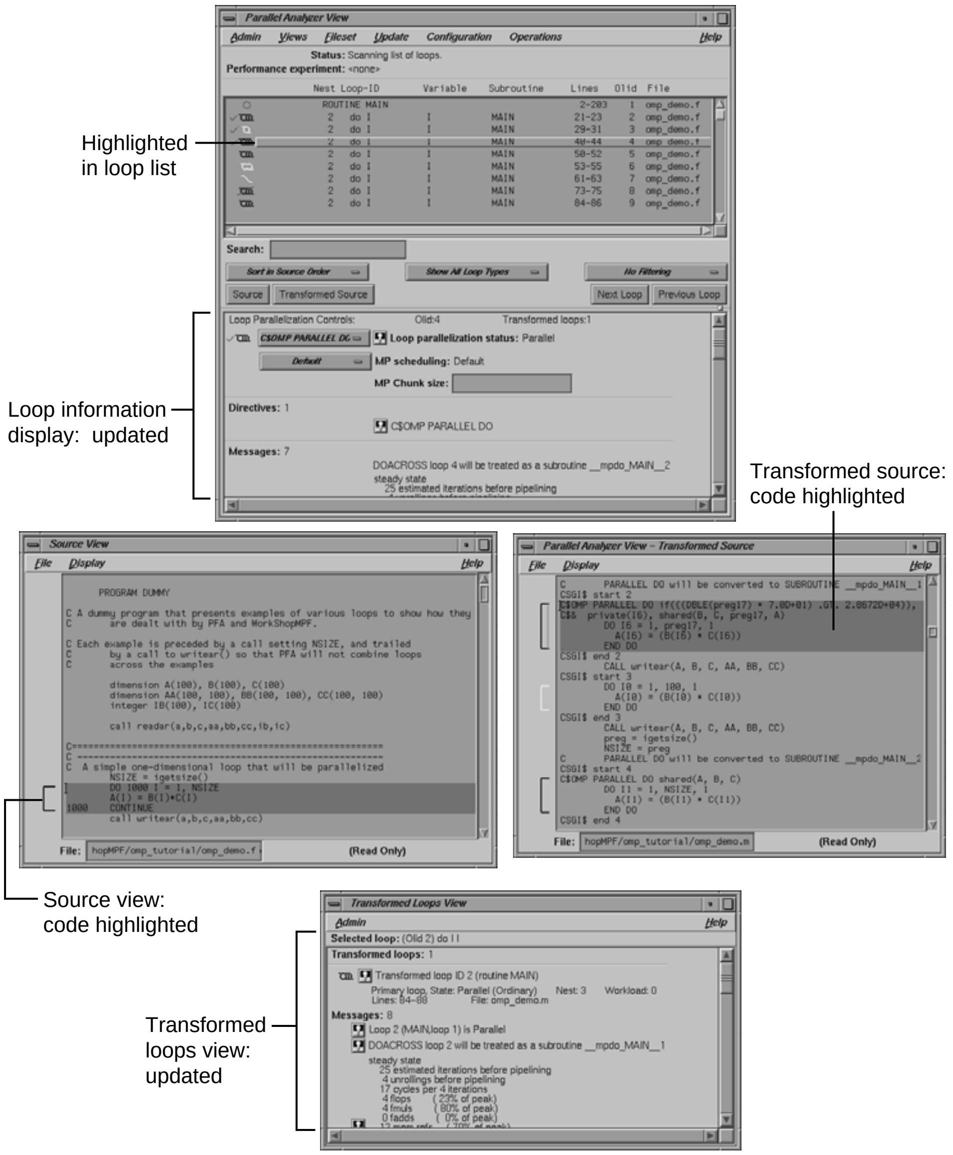
Selecting a loop has a number of effects on the different windows in the Parallel Analyzer View (see Figure 10, page 22). Not all of the windows in the figure are open at this point in the tutorial; you can open them from the Views menu.

- In the Parallel Analyzer View, information about the selected loop appears in the previously empty loop information display (see Section 2.6.4, page 23).
- In the Source View, the original source code of the loop appears and is highlighted (see Section 2.6.1.1, page 16).
- In the Transformed Source, the first of the loops into which the original loop was transformed appears and is highlighted in the window. A bright vertical bar also appears next to each transformed loop that came from the original loop (see Section 2.6.1.2, page 18).
- The Transformed Loops View shows information about the loop after parallelization (see Section 2.6.5, page 25).
- The PFA Analysis Parameters View (`o32` code only) shows parameter values for the selected loop (see Section 6.6.3, page 159).

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

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

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



Status: Scanning list of loops.
Performance experiment: <none>

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

Loop Parallelization Controls: Olid:4 Transformed loops:1

C$OMP PARALLEL DO — **Loop parallelization status:** Parallel

Default — **MP scheduling:** Default

MP Chunk size:

Directives: 1

C$OMP PARALLEL DO

Messages: 7

DOACROSS loop 4 will be treated as a subroutine __mpdo_MAIN__2
steady state
25 estimated iterations before pipelining

```
      PROGRAM DUMMY

C A dummy program that presents examples of various loops to show how they
C       are dealt with by PFA and WorkShopMPF.

C Each example is preceded by a call setting NSIZE, and trailed
C       by a call to writear() so that PFA will not combine loops
C       across the examples

        dimension A(100), B(100), C(100)
        dimension AA(100, 100), BB(100, 100), CC(100, 100)
        integer IB(100), IC(100)

        call readar(a,b,c,aa,bb,cc,ib,ic)

C==========================================================
C ---------------------------------------------------------
C  A simple one-dimensional loop that will be parallelized
        NSIZE = igetsize()
        DO 1000 I = 1, NSIZE
        A(I) = B(I)*C(I)
1000    CONTINUE
        call writear(a,b,c,aa,bb,cc)
```

File: hopMPF/omp_tutorial/omp_demo.f (Read Only)

```
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__1
CSGI$ start 2
C$OMP PARALLEL DO if(((DBLE(preg17) * 7.0D+01) .GT. 2.8672D+04)),
C$&  private(I6), shared(B, C, preg17, A)
        DO I6 = 1, preg17, 1
          A(I6) = (B(I6) * C(I6))
        END DO
CSGI$ end 2
        CALL writear(A, B, C, AA, BB, CC)
CSGI$ start 3
        DO I0 = 1, 100, 1
          A(I0) = (B(I0) * C(I0))
        END DO
CSGI$ end 3
        CALL writear(A, B, C, AA, BB, CC)
        preg = igetsize()
        NSIZE = preg
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__2
CSGI$ start 4
C$OMP PARALLEL DO shared(A, B, C)
        DO I1 = 1, NSIZE, 1
          A(I1) = (B(I1) * C(I1))
        END DO
CSGI$ end 4
```

File: hopMPF/omp_tutorial/omp_demo.m (Read Only)

Selected loop: (Olid 2) do I I

Transformed loops: 1

Transformed loop ID 2 (routine MAIN)
Primary loop, State: Parallel (Ordinary) Nest: 3 Workload: 0
Lines: 84–88 File: omp_demo.m

Messages: 8

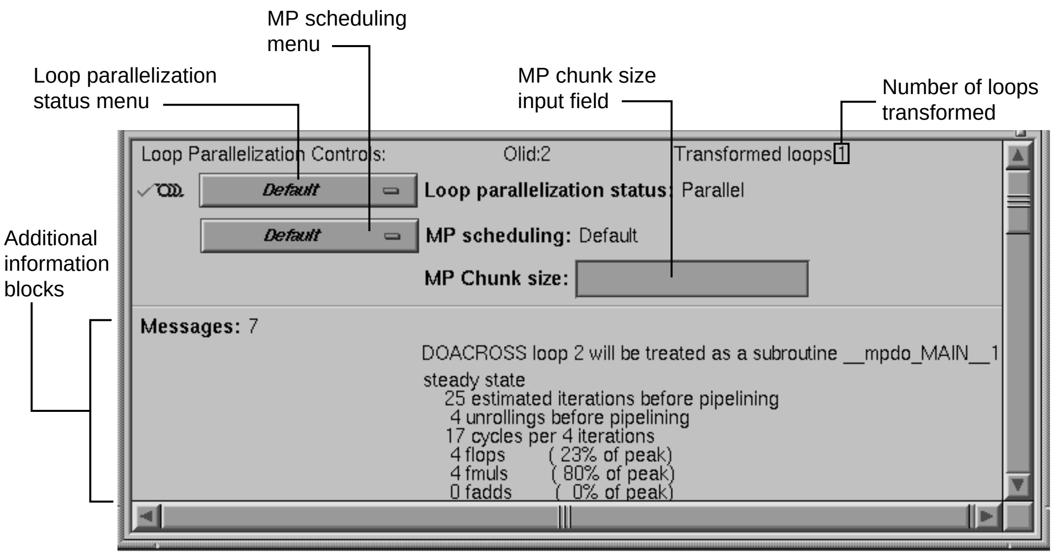
Loop 2 (MAIN,loop 1) is Parallel
DOACROSS loop 2 will be treated as a subroutine __mpdo_MAIN__1

```
steady state
   25 estimated iterations before pipelining
    4 unrollings before pipelining
   17 cycles per 4 iterations
    4 flops      ( 23% of peak)
    4 fmuls      ( 80% of peak)
    0 fadds      (  0% of peak)
```

Figure 10. Global Effects of Selecting a Loop

2.6.4 Using the Loop Information Display

The loop information display occupies the portion of the main view below the loop display controls. Initially, the display shows only `No loop is selected`. After a loop or subroutine is selected, the display contains detailed information and controls for requesting changes to your code (see Figure 11, page 23).


MP scheduling menu
Loop parallelization status menu
MP chunk size input field
Number of loops transformed
Additional information blocks
Default
Default

Loop Parallelization Controls: Olid:2 Transformed loops 1

Loop parallelization status: Parallel

MP scheduling: Default

MP Chunk size:

Messages: 7

```
DOACROSS loop 2 will be treated as a subroutine __mpdo_MAIN__1
steady state
   25 estimated iterations before pipelining
    4 unrollings before pipelining
   17 cycles per 4 iterations
    4 flops       ( 23% of peak)
    4 fmuls       ( 80% of peak)
    0 fadds       (  0% of peak)
```

Figure 11. Loop Information Display Without Performance Data

2.6.4.1 Loop Parallelization Controls

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

- On the first line is the loop Olid and the number of transformed loops derived from the selected loop.
- The next three lines display two option buttons and an editable text field.
  - The top button controls the loop's parallelization status (see Section 6.5.2.1, page 144).

– The second button controls the loop's multiprocessor scheduling. It is shown for all loops but is applicable to parallel loops only; for more information see Section 6.5.2.2, page 146.

– The MP Chunk size editable text field lets you select the scheduling *chunk size* (see the Glossary). (For more information on the MP Chunk size, see Section 6.5.2.3, page 147).

When the Parallel Analyzer View is run with a performance experiment, by invoking SpeedShop, an additional block (see Figure 47, page 111) appears above the parallelization controls. It gives performance information about the loop.

2.6.4.2 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, when appropriate, the following information:

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

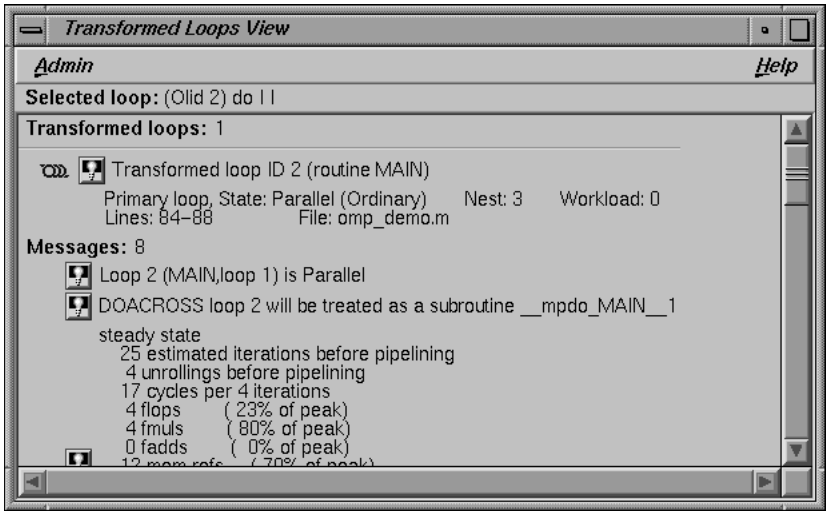
Some of these lines may be accompanied by highlight buttons, represented by small light bulb icons. When you click one of these buttons, it highlights the relevant part of the code in the Source View and the Transformed Source windows.

The loop information display shows directives that apply to an entire subroutine when you select the line with the subroutine's name. If you select Olid 1, you see that there are no global directives in the main program. However, if you find subroutine `dst1d()`, you will see a directive that applies to it (see Section 2.12.1, page 62).

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

2.6.5 Using the Transformed Loops View

To see detailed information about the transformed loops derived from a particular loop, pull down the `Views` > `Transformed Loops View` option (see Figure 12, page 25.).



```
Selected loop: (Olid 2) do I I
Transformed loops: 1

Transformed loop ID 2 (routine MAIN)
    Primary loop, State: Parallel (Ordinary)     Nest: 3     Workload: 0
    Lines: 84–88              File: omp_demo.m
Messages: 8
  Loop 2 (MAIN,loop 1) is Parallel
  DOACROSS loop 2 will be treated as a subroutine __mpdo_MAIN__1
  steady state
     25 estimated iterations before pipelining
      4 unrollings before pipelining
     17 cycles per 4 iterations
      4 flops       ( 23% of peak)
      4 fmuls       ( 80% of peak)
      0 fadds       (  0% of peak)
```

Figure 12. Transformed Loops View for Loop Olid 1

2.6.5.1 Transformed Loops View Description

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

The first line in each block contains:

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

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

- Whether it is a *primary* loop or *secondary* loop. A primary look is transformed from the selected original loop. A secondary loop is transformed from a different original loop, but it incorporates some code from the selected original loop.
- Parallelization state.
- Whether it is an ordinary loop or *interchanged* loop (see the Glossary).
- Nesting level.
- Workload.

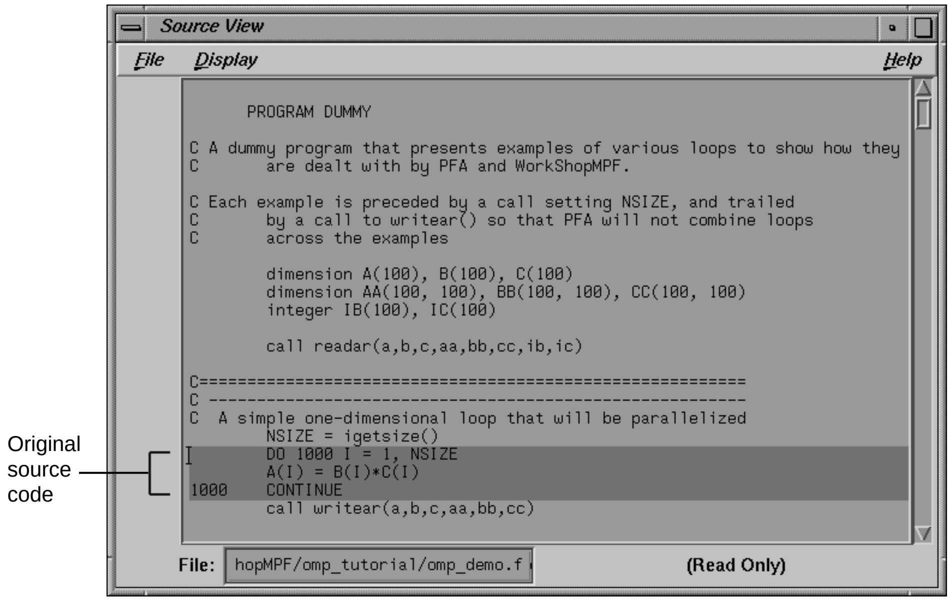
The second line displays the location of the loop in the transformed source.

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

2.6.5.2 Selecting Transformed Loops

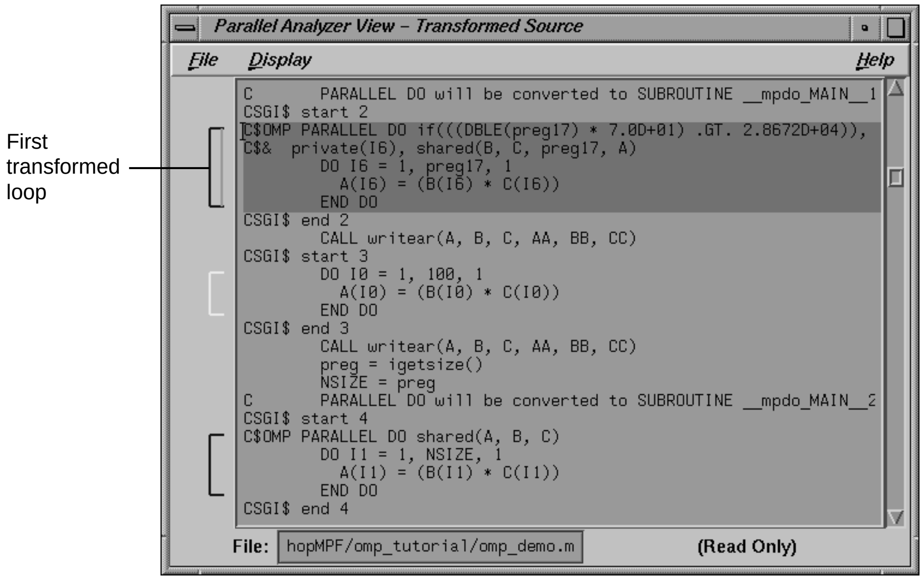
You can also select specific transformed loops. When you click a highlight button in the Transformed Loop View, the highlighting of the original source typically changes color, although for loop Olid 1 the highlighted lines do not (see Figure 13, page 27). For loops with more extensive transformations, the set of highlighted lines is different when you select from the Transformed Loops View (for example, see Section 2.7.4, page 31).

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



```
      PROGRAM DUMMY

C A dummy program that presents examples of various loops to show how they
C       are dealt with by PFA and WorkShopMPF.

C Each example is preceded by a call setting NSIZE, and trailed
C       by a call to writear() so that PFA will not combine loops
C       across the examples

        dimension A(100), B(100), C(100)
        dimension AA(100, 100), BB(100, 100), CC(100, 100)
        integer IB(100), IC(100)

        call readar(a,b,c,aa,bb,cc,ib,ic)

C=========================================================
C ---------------------------------------------------------
C  A simple one-dimensional loop that will be parallelized
        NSIZE = igetsize()
        DO 1000 I = 1, NSIZE
        A(I) = B(I)*C(I)
1000    CONTINUE
        call writear(a,b,c,aa,bb,cc)
```

File: hopMPF/omp_tutorial/omp_demo.f



```
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__1
CSGI$ start 2
C$OMP PARALLEL DO if(((DBLE(preg17) * 7.0D+01) .GT. 2.8672D+04)),
C$&  private(I6), shared(B, C, preg17, A)
        DO I6 = 1, preg17, 1
          A(I6) = (B(I6) * C(I6))
        END DO
CSGI$ end 2
        CALL writear(A, B, C, AA, BB, CC)
CSGI$ start 3
        DO I0 = 1, 100, 1
          A(I0) = (B(I0) * C(I0))
        END DO
CSGI$ end 3
        CALL writear(A, B, C, AA, BB, CC)
        preg = igetsize()
        NSIZE = preg
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__2
CSGI$ start 4
C$OMP PARALLEL DO shared(A, B, C)
        DO I1 = 1, NSIZE, 1
          A(I1) = (B(I1) * C(I1))
        END DO
CSGI$ end 4
```

File: hopMPF/omp_tutorial/omp_demo.m

Figure 13. Transformed Loops in Source Windows

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

2.7 Examples of Simple Loops

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

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

- Simple parallel loop, see Section 2.7.1, page 28.
- Serial loop, see Section 2.7.2, page 29.
- Explicitly parallelized loop, see Section 2.7.3, page 29.
- Fused loops, see Section 2.7.4, page 31.
- Eliminated loop, see Section 2.7.5, page 32.

Two other sections discuss more complicated loops:

- Examining loops with obstacles to parallelization, see Section 2.8, page 32.
- Examining nested loops, see Section 2.9, page 44.

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

2.7.1 Simple Parallel Loop

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

Example 1: Simple Parallel Loop

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

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

2.7.2 Serial Loop

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

Example 2: Serial Loop

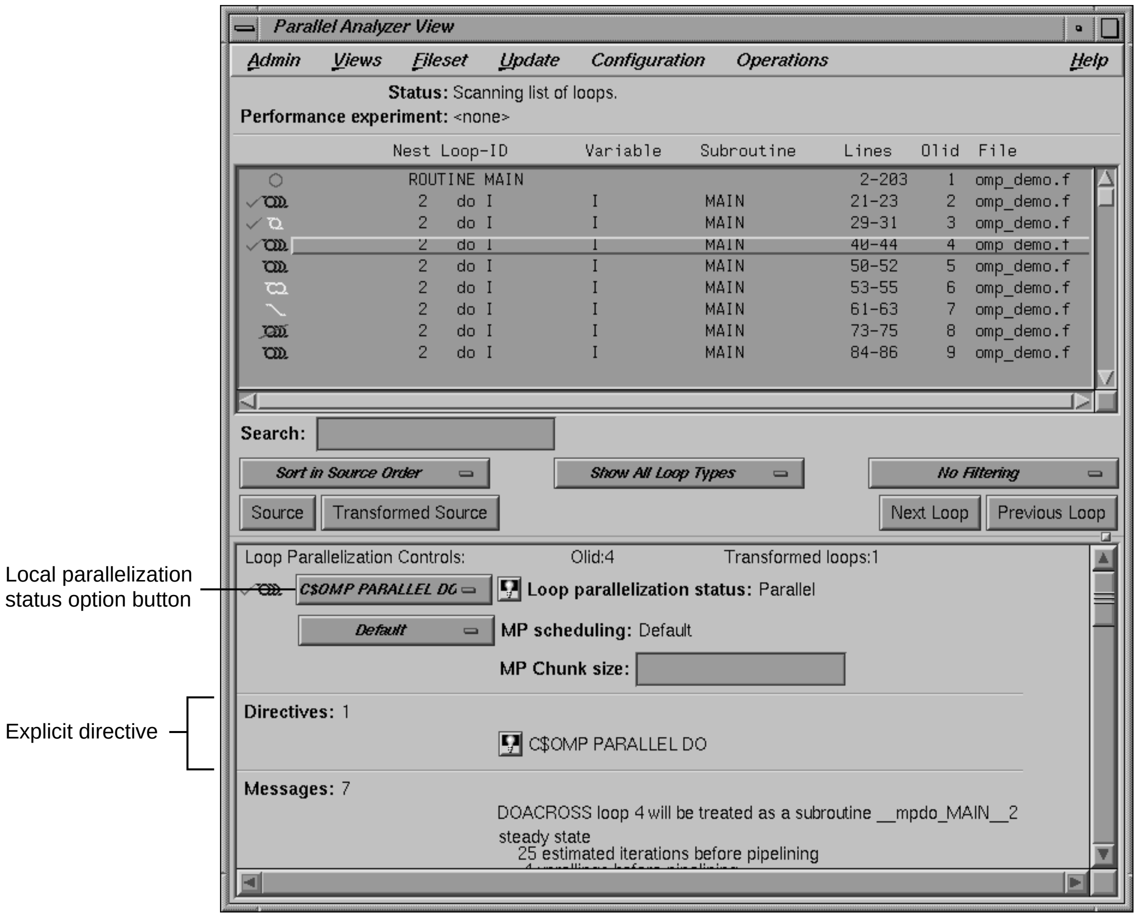
```
      DO 1100 I = 1, NSIZE
        A(I) = B(I)*C(I)
1100  CONTINUE
```

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

2.7.3 Explicitly Parallelized Loop

Loop Olid 3 is parallelized because it contains an explicit `C$OMP PARALLEL DO` directive in the source, as is shown in the loop information display (Figure 14, page 30). The compiler passes the directive through to the transformed source.

The loop parallelization status option button is set to `C$OMP PARALLEL DO`..., and it is shown with a highlight button. Clicking the highlight button brings up both the Source View (Figure 15, page 31), if it is not already opened, and the Parallelization Control View, which shows more information about the parallelization directive.



Status: Scanning list of loops.
Performance experiment: <none>

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

Loop Parallelization Controls: Olid:4 Transformed loops:1

Loop parallelization status: Parallel

MP scheduling: Default

MP Chunk size:

Directives: 1

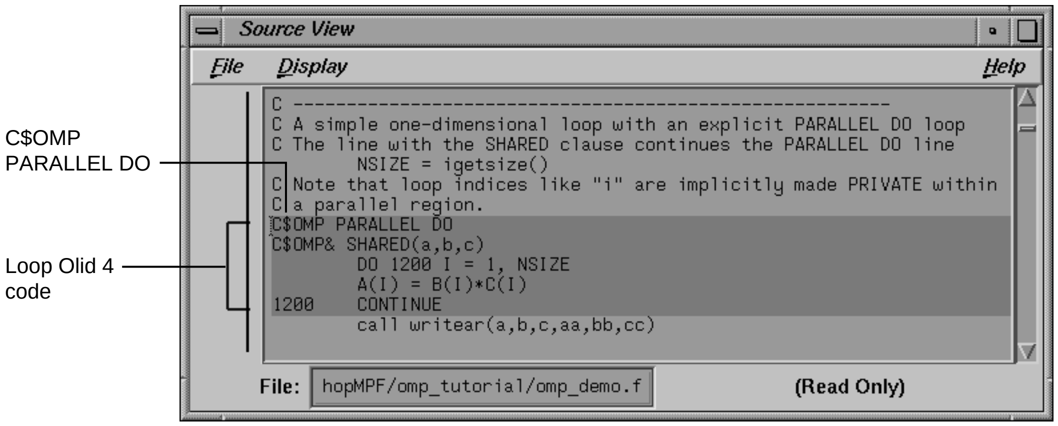
C$OMP PARALLEL DO

Messages: 7

DOACROSS loop 4 will be treated as a subroutine __mpdo_MAIN__2
steady state
25 estimated iterations before pipelining

Figure 14. Explicitly Parallelized Loop

If you clicked on the highlight button, close the Parallelization Control View. (Using the Parallelization Control View is discussed in Section 2.10.1.1, page 46.)



```
C ----------------------------------------------------------
C A simple one-dimensional loop with an explicit PARALLEL DO loop
C The line with the SHARED clause continues the PARALLEL DO line
        NSIZE = igetsize()
C Note that loop indices like "i" are implicitly made PRIVATE within
C a parallel region.
C$OMP PARALLEL DO
C$OMP& SHARED(a,b,c)
        DO 1200 I = 1, NSIZE
        A(I) = B(I)*C(I)
1200    CONTINUE
        call writear(a,b,c,aa,bb,cc)
```

Figure 15. Source View of C$OMP PARALLEL DO Directive

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

2.7.4 Fused Loops

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

Example 3: Fused Loop

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

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

2.7.5 Loop That Is Eliminated

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

Example 4: Eliminated Loop

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

Move to the next loop, Olid 8, by clicking the `Next Loop` button. This loop is discussed in Section 2.8.1.1, page 33.

2.8 Examining Loops With Obstacles to Parallelization

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

- Section 2.8.1, page 32
- Section 2.8.2, page 37
- Section 2.8.3, page 37
- Section 2.8.4, page 38
- Section 2.8.5, page 38

These loops are a few specific examples of the obstacles to parallelization recognized by the compiler. The final part of this section, Section 2.8.6, page 39, contains two tables that list all of the messages generated by the compiler that concern obstacles to parallelization.

2.8.1 Carried Data Dependence

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

- Section 2.8.1.1, page 33

- Section 2.8.1.2, page 34
- Section 2.8.1.3, page 35
- Section 2.8.1.4, page 36

2.8.1.1 Unparallelizable Carried Data Dependence

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

Example 5: Unparallelizable Carried Data Dependence

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

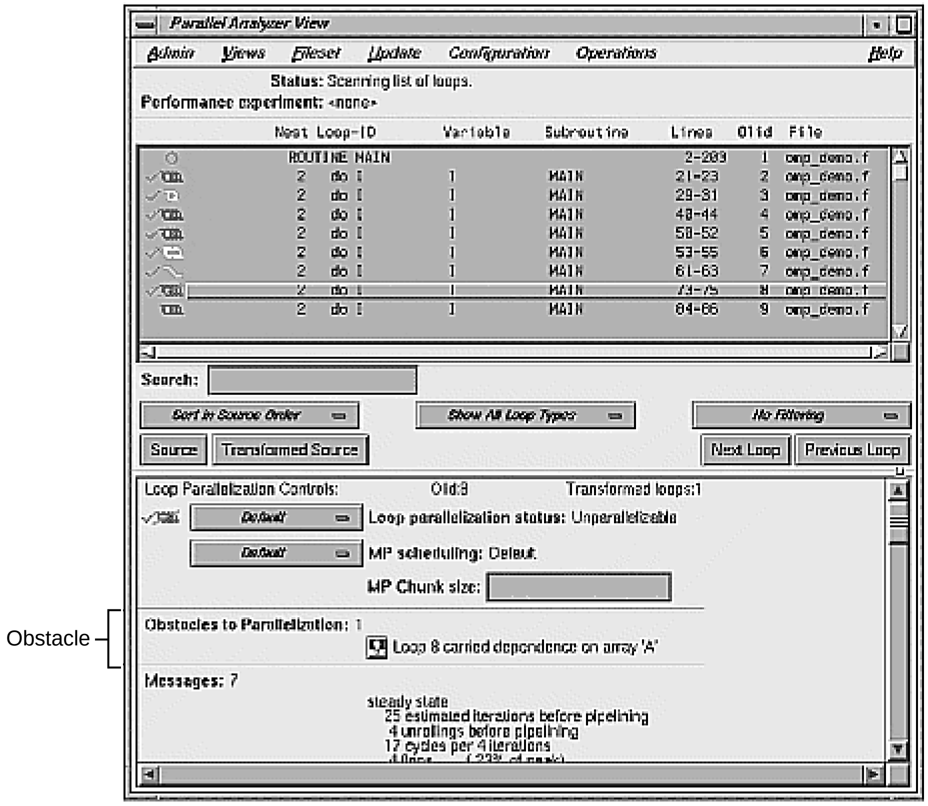
If the loop were nontrivial (if `NSIZE` were greater than two) and if the loop were run in parallel, iterations might execute out of order. For example, iteration 4, which sets `A(4)` to `A(5)`, might occur after iteration 5, which resets the value of `A(5)`; the computation would be unpredictable.

The loop information display in Figure 16, page 34, lists the obstacle to parallelization.

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

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

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



```
Status: Scanning list of loops.
Performance experiment: <none>

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

Loop Parallelization Controls:     Olid:8     Transformed loops:1
Loop parallelization status: Unparallelizable
MP scheduling: Default
MP Chunk size:

Obstacles to Parallelization: 1
     Loop 8 carried dependence on array 'A'

Messages: 7
     steady state
        25 estimated iterations before pipelining
         4 unrollings before pipelining
        17 cycles per 4 iterations
```

Figure 16. Obstacles to Parallelization

2.8.1.2 Parallelizable Carried Data Dependence

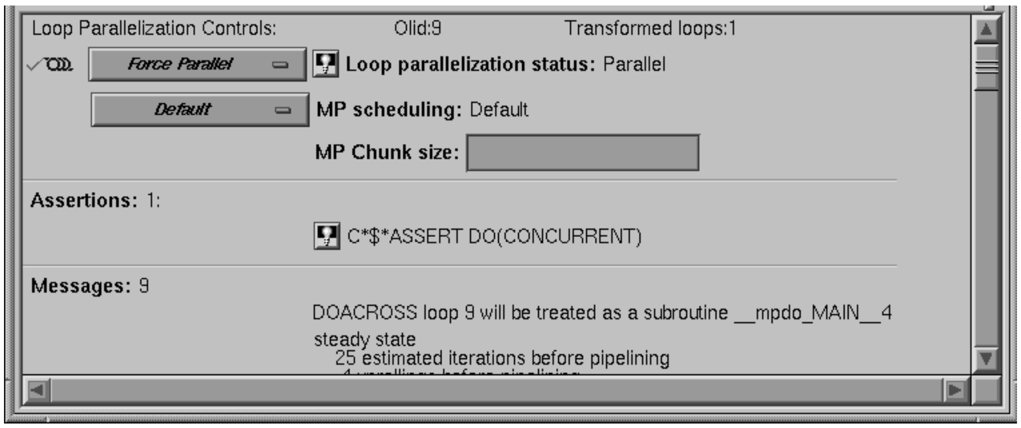
Loop Olid 9 has a structure similar to loop Olid 8. Despite the similarity however, Olid 9 may be parallelized.

Example 6: Parallelizable Carried Data Dependence

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

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

Click the highlight button (Figure 17, page 35) to show the assertion in the Source View.



Loop Parallelization Controls: Olid:9 Transformed loops:1

Loop parallelization status: Parallel

MP scheduling: Default

MP Chunk size:

Assertions: 1:

C*$*ASSERT DO(CONCURRENT)

Messages: 9

```
DOACROSS loop 9 will be treated as a subroutine __mpdo_MAIN__4
steady state
   25 estimated iterations before pipelining
```

Figure 17. Parallelizable Data Dependence

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

2.8.1.3 Multi-line Data Dependence

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

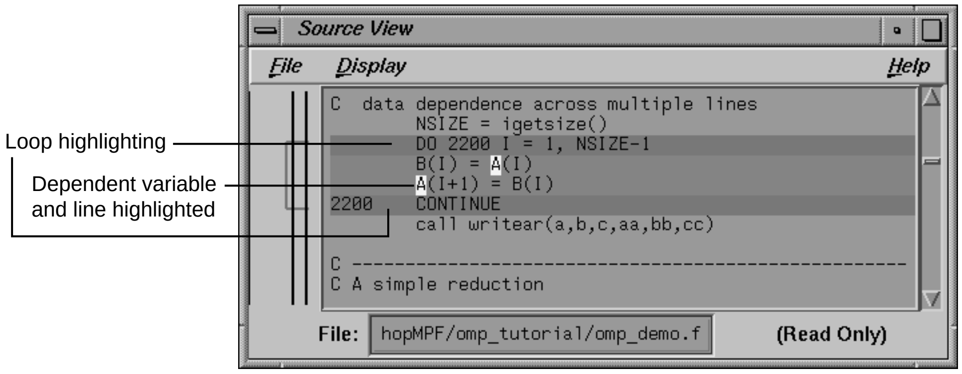
Example 7: Multi-line Data Dependence

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

Click the highlight button on the obstacle line.

In the Source View, highlighting shows the dependency variable on the two lines. (See Figure 18, page 36.) Of course, real programs, typically, have far more complex dependences than this.

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



```
C  data dependence across multiple lines
        NSIZE = igetsize()
        DO 2200 I = 1, NSIZE-1
        B(I) = A(I)
        A(I+1) = B(I)
2200    CONTINUE
        call writear(a,b,c,aa,bb,cc)

C ------------------------------------------------
C A simple reduction
```

hopMPF/omp_tutorial/omp_demo.f

Figure 18. Highlighting on Multiple Lines

2.8.1.4 Reductions

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

Example 8: Reduction

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

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

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

2.8.2 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.

Example 9: Input/Output Operation

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

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

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

2.8.3 Unstructured Control Flow

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

Example 10: Unstructured Control Flow

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

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

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

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

2.8.4 Subroutine Calls

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

2.8.4.1 Unparallelizable Loop With a Subroutine Call

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

Example 11: Unparallelizable Loop With Subroutine Call

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

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

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

2.8.4.2 Parallelizable Loop With a Subroutine Call

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

Example 12: Parallelizable Loop With Subroutine Call

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

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

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

2.8.5 Permutation Vectors

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

2.8.5.1 Unparallelizable Loop With a Permutation Vector

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

Example 13: Unparallelizable Loop With Permutation Vector

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

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

2.8.5.2 Parallelizable Loop With a Permutation Vector

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

Example 14: Parallelizable Loop With Permutation Vector

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

Move to the next loop, Olid 18, by clicking `Next Loop`. This loop is discussed in Section 2.9.1, page 44.

2.8.6 Obstacles to Parallelization Messages

All of the messages that can be found in an Obstacles to Parallelization information block (Figure 16, page 34) are found in Table 1, page 40, and Table 2, page 41. Because they include specific loop and line information, messages that appear in the loop information display differ slightly from those in the tables.

The next table contains messages concerning major issues, such as, whether a loop could have gone parallel, could not have gone parallel, or might be able to go parallel.

Table 1. Major Obstacles to Parallelization Messages

Message	Comments
Loop doesn't have parallelization directive	Auto-parallelization is off. Loop doesn't contain a parallelization directive.
Loop is preferred serial; insufficient work to justify parallelization	Could have been parallelized, but preferred serial. The compiler determined there was not enough work in the loop to make parallelization worthwhile.
Loop is preferred serial; parallelizing inner loop is more efficient	Could have been parallelized, but preferred serial. The compiler determined that making an inner loop parallel would lead to faster execution.
Loop has unstructured control flow	Might be parallelizable. There is a `goto` statement or other unstructured control flow in the loop.
Loop was created by peeling the last iteration of a parallel loop	Might be parallelizable. Loop was created by peeling off the final iteration of another loop to make that loop go parallel. Compiler did not try to parallelize this peeled, last iteration.
User directive specifies serial execution for loop	Might be parallelizable. Loop has a directive that it should not be parallelized.
Loop can not be parallelized; tiled for reshaped array instead	Might be parallelizable. The loop has been tiled because it has reshaped arrays, or is inside a loop with reshaped arrays. The compiler does not parallelize such loops.
Loop is nested inside a parallel loop	Might be parallelizable. Loop is inside a parallel loop. Therefore, the compiler does not consider it to be a candidate for parallelization.

Message	Comments
Loop is the serial version of parallel loop	Might be parallelizable. The loop is part of the serial version of a parallelized loop. This may occur when a loop is in a routine called from a parallelized loop; the called loop is effectively nested in a parallel loop, so the compiler does not parallelize it.
Tough upper bounds	Could not have gone parallel. Loop could not be put in standard form, and therefore could not be analyzed for parallelization. Standard form is `for (i = lb; i <= ub; i++)`
Indirect ref	Could not have gone parallel. Loop contains some complex memory access that is too difficult to analyze.

Table 2, page 41, lists the Obstacles to Parallelization block messages that deal with dependence issues, such as, those involving scalars, arrays, missing information, and finalization.

Table 2. Data Dependence Obstacles to Parallelization

Messages	Comments
Loop has carried dependence on scalar variable	Problem with scalars. The loop has a carried dependence on a scalar variable.
Loop scalar variable is aliased precluding auto parallelization	Problem with scalars. A scalar variable is aliased with another variable, e.g. a statement equivalencing a scalar and an array.
Loop can not determine last value for variable	Problem with scalars. A variable is used out of the loop, and the compiler could not determine a unique last value.
Loop carried dependence on array	Problem with arrays. The loop carries an array dependence from one array member to another array member.

Messages	Comments
Call inhibits auto parallelization	Problem with missing dependence information. A call in the loop has no dependence information, and is assumed to create a data dependence.
Input-output statement	Problem with missing dependence information. The compiler does not parallelize loops with input or output statements.
Insufficient information in array	Problem with missing dependence information. Array has no dependence information.
Insufficient information in reference	Problem with missing dependence information. Unnamed reference has no dependence information.
Loop must finalize value of scalar before it can go parallel	Problem with finalization. Value of scalar must be determined to parallelize loop.
Loop must finalize value of array before it can go parallel	Problem with finalization. Value of array must be determined to parallelize loop.

Messages	Comments

Scalar may not be assigned in final iteration	Problem with finalization.
	The compiler needed to finalize the value of a scalar to parallelize the loop, but it couldn't because the value is not always assigned in the last iteration of the loop.
	The following code is an example. The variable *s* poses a problem; the `if` statement makes it unclear whether the variable is set in the last iteration of the loop.
	`subroutine fun02(a, b, n, s)`
	`    integer a(n), b(n), s, n`
	`    do i = 1, n`
	`      if (a(i) .gt. 0) then`
	`        s = a(i)`
	`      end if`
	`      b(i) = a(i) + s`
	`    end do`
	`    end`

Array may not be assigned in final iteration	Problem with finalization.
	The compiler needed to finalize the value of an array to parallelize the loop, but it couldn't because the values are not always assigned in the last iteration of the loop.
	The following is an example. The variable *b* poses a problem when the compiler tries to parallelize the *i* loop; it is not set in the last iteration.
	`       subroutine fun04(a, b, n)`
	`        integer i, j, k, n`
	`        integer b(n), a(n,n,n)`
	`        do i = 1, n`
	`          do j = i + 3, n`
	`c*$* no fusion`
	`            do k = 1, n`
	`              b(k) = k`
	`            end do`
	`            do k = 1, n`
	`              a(i,j,k) = a(i,j,k) + b(k)`
	`            end do`
	`          end do`
	`        end do`
	`        end`

2.9 Examining Nested Loops

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

2.9.1 Doubly Nested Loop

Loop Olid 18 is the outer loop of a pair of loops and it runs in parallel. The inner loop runs in serial, because the compiler knows that one parallel loop should not be nested inside another. However, you can force parallelization in this context by inserting a `C$OMP PARALLEL DO` directive with the `C$SGI&NEST` clause. For example, see Section 2.12.2, page 64.

Example 15: Doubly Nested Loop

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

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

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.

Click `Next Loo`p again to select the outer loop of the next nested pair.

2.9.2 Interchanged Doubly Nested Loop

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

Example 16: Interchanged Doubly Nested Loop

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

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

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

2.9.3 Triply Nested Loop With an Interchange

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

Example 17: Triply Nested Loop With Interchange

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

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

Because the innermost serial loop, Olid 24, depends without recurrence on the indices of Olid 22 and Olid 23, iterations of loop Olid 22 can run concurrently. The compiler does not recognize this possibility. This brings us to the subject of the next section, the use of the Parallel Analyzer View tools to modify the source.

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

2.10 Modifying Source Files and Compiling

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

You control most of the directives and some of the assertions available from the Parallel Analyzer View with the Operations menu. (See Table 3, page 131.)

You control most of the assertions and the more complex directives, `C$OMP DO` and `C$OMP PARALLEL DO`, with the loop parallelization status option button. (See Figure 19, page 47.)

There are two steps to modifying source files:

1. Make changes using the Parallel Analyzer View controls, discussed in the next subsection, Section 2.10.1, page 46
2. Modify the source and rebuild the program and its analysis files, discussed in Section 2.10.2, page 52.

2.10.1 Making Changes

You make changes by one of the following actions:

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

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

These are the changes discussed in this section:

- Section 2.10.1.1, page 46
- Section 2.10.1.2, page 50
- Section 2.10.1.3, page 51

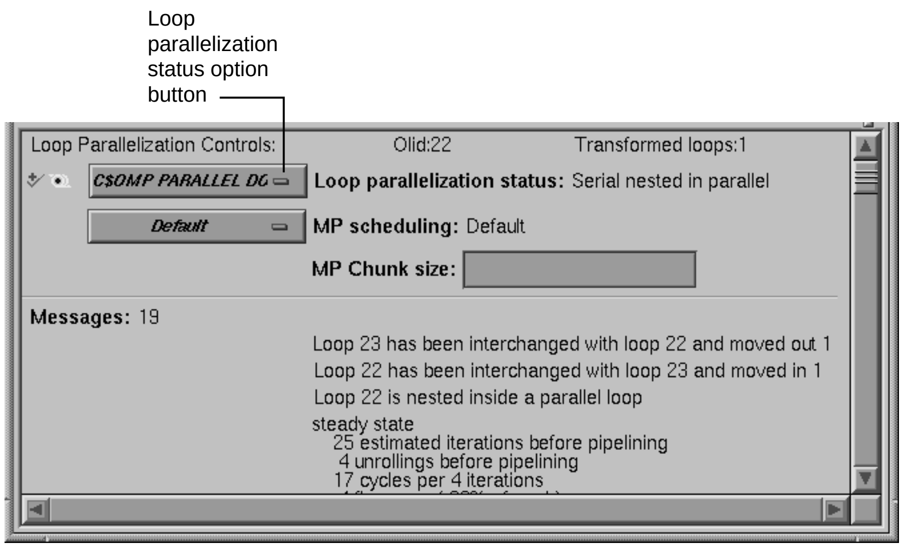
2.10.1.1 Adding `C$OMP PARALLEL DO` Directives and Clauses

Loop Olid 22, shown in Example 17, page 45, is a serial loop nested inside a parallel loop. It is not parallelized, but its iterations could run concurrently.

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

1. Make sure loop Olid 22 is selected.
2. Click on the loop parallelization status option button (Figure 19, page 47), and choose `C$OMP PARALLEL DO`... to parallelize Olid 22.

This sequence requests a change in the source code, and opens the Parallelization Control View (Figure 20, page 49). You can now look at variables in the loop and attach clauses to the directive, if needed.



Loop Parallelization Controls: Olid:22 Transformed loops:1

Loop parallelization status: Serial nested in parallel

MP scheduling: Default

MP Chunk size:

Messages: 19

```
Loop 23 has been interchanged with loop 22 and moved out 1
Loop 22 has been interchanged with loop 23 and moved in 1
Loop 22 is nested inside a parallel loop
steady state
   25 estimated iterations before pipelining
    4 unrollings before pipelining
   17 cycles per 4 iterations
```

Figure 19. Requesting a C$OMP PARALLEL DO Directive

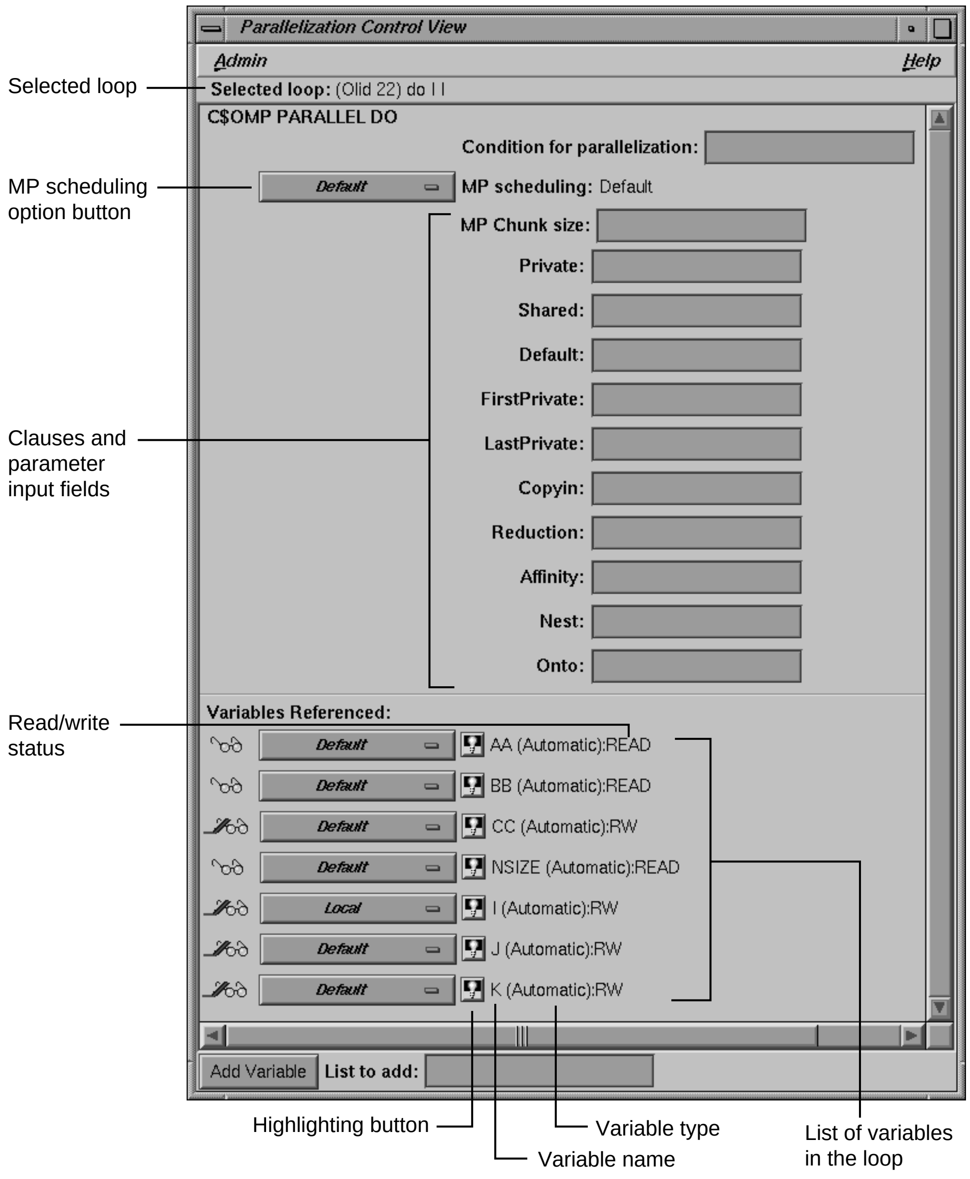
Figure 20, page 49, shows information presented in the Parallelization Control View for a `C$OMP PARALLEL DO` directive. (For the `C$OMP DO` directive, see Section 6.6.1, page 150):

- The selected loop.
- Condition for parallelization editable text field.
- MP scheduling option button.
- MP Chunk size editable text field.
- `PRIVATE`, `SHARED`, `DEFAULT`, `FIRSTPRIVATE`, `LASTPRIVATE`, `COPYIN`, `REDUCTION`, `AFFINITY`, `NEST`, and ONTO clause windows.

- A list of all the variables in the loop, each with an icon indicating whether the variable was read, written, or both; these icons are introduced in Section 2.4.2, page 10.

In the list of variables, each variable has a highlight button to indicate in the Source View its use within the loop; click some of the buttons to see the variables highlighted in the source view. After each variable’s name, there is a descriptor of its storage class: Automatic, Common, or Reference. (See Section 6.6.1.5, page 157.)

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


Parallelization Control View
Admin
Help
Selected loop
MP scheduling option button
Clauses and parameter input fields
Read/write status
Highlighting button
Variable name
Variable type
List of variables in the loop
Default
Local
Add Variable

Selected loop: (Olid 22) do I I

C$OMP PARALLEL DO

Condition for parallelization:

MP scheduling: Default

MP Chunk size:

Private:

Shared:

Default:

FirstPrivate:

LastPrivate:

Copyin:

Reduction:

Affinity:

Nest:

Onto:

Variables Referenced:

AA (Automatic):READ

BB (Automatic):READ

CC (Automatic):RW

NSIZE (Automatic):READ

I (Automatic):RW

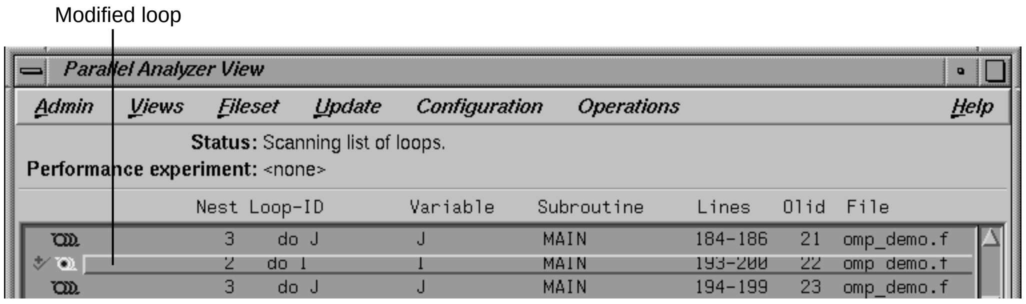
J (Automatic):RW

K (Automatic):RW

List to add:

Figure 20. Parallelization Control View After Choosing `C$OMP PARALLEL DO...`

Notice that in the loop list, there is now a red plus sign next to this loop, indicating that a change has been requested. (See Figure 21, page 50.)



Status: Scanning list of loops.
Performance experiment: <none>

```
Nest Loop-ID      Variable   Subroutine   Lines     Olid  File
  3   do J        J          MAIN         184-186    21   omp_demo.f
  2   do I        I          MAIN         193-200    22   omp_demo.f
  3   do J        J          MAIN         194-199    23   omp_demo.f
```

Figure 21. Effect of Changes on the Loop List

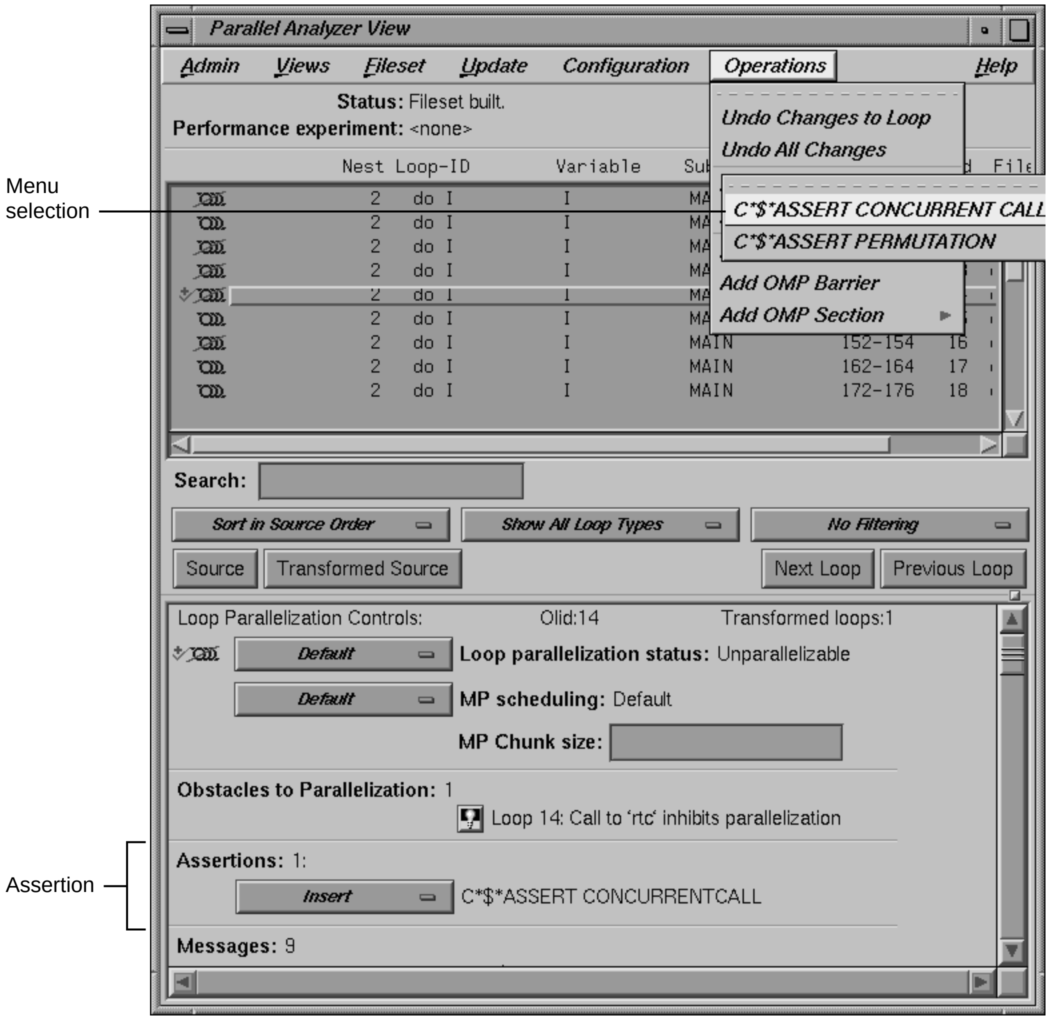
Close the Parallelization Control View by using its Admin > Close option.

2.10.1.2 Adding New Assertions or Directives With the Operations Menu

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

1. Find loop Olid 14 (introduced in Example 11, page 38) either by scrolling the loop list or by using the search feature of the loop list. (Go to the Search field and enter **`14`**.)
2. Double-click the highlighted line in the loop list to select the loop.
3. Pull down Operations > Add Assertion > `C*$*ASSERT CONCURRENT CALL` to request a new assertion. (See Figure 22, page 51.)

This adds an assertion, `C*$* ASSERT CONCURRENT CALL`, that says it is safe to parallelize the loop despite the call to `RTC()`, which the compiler thought might be an obstacle to parallelization. The loop information display shows the new assertion, along with an `Insert option` button to indicate the state of the assertion when you modify the code. (See Figure 22, page 51.)



Status: Fileset built.
Performance experiment: <none>

```
Nest Loop-ID      Variable   Sub
  2   do I          I        MA
  2   do I          I        MA
  2   do I          I        MA
  2   do I          I        MA
  2   do I          I        MA
  2   do I          I        MA
  2   do I          I        MAIN     152-154   16
  2   do I          I        MAIN     162-164   17
  2   do I          I        MAIN     172-176   18
```

Search:

Loop Parallelization Controls: Olid:14 Transformed loops:1

Loop parallelization status: Unparallelizable

MP scheduling: Default

MP Chunk size:

Obstacles to Parallelization: 1

Loop 14: Call to 'rtc' inhibits parallelization

Assertions: 1:

C*$*ASSERT CONCURRENTCALL

Messages: 9

Figure 22. Adding an Assertion

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

2.10.1.3 Deleting Assertions or Directives

Move to the next loop, Olid 15 (shown in Example 12, page 38).

To delete an assertion, follow these steps:

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

Figure 23, page 52, shows the state of the assertion in the information display. A similar procedure is used to delete directives.

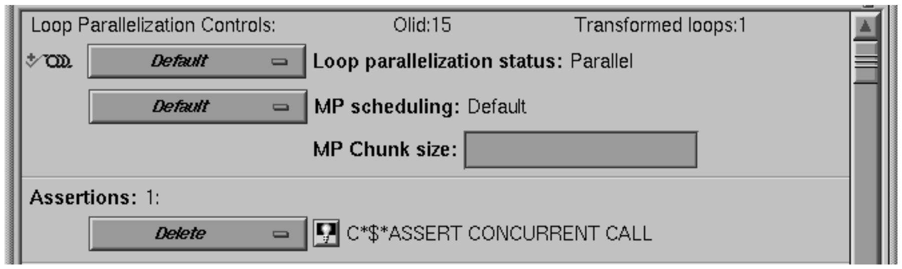


Figure 23. Deleting an Assertion

From this point, the next non-optional step in the tutorial is at the beginning of Section 2.10.2.3, page 54.

2.10.2 Applying Requested Changes

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

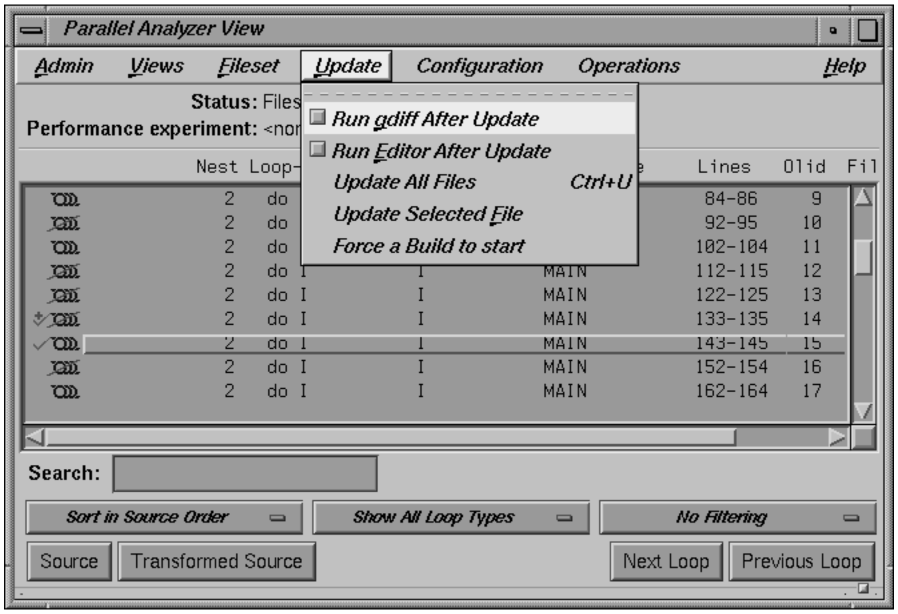
1. Generates a `sed` script to accomplish the following steps.
   - Rename the original file to have the suffix `.old.`
   - Run `sed` on that file to produce a new version of the file, in this case `omp_demo.f.`
2. Depending on how you set the two checkboxes in the Update menu, the Parallel Analyzer View then does one of the following:
   - Spawns the WorkShop Build Manager to rerun the compiler on the new version of the file.

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

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

2.10.2.1 Viewing Changes With gdiff

By default, the Parallel Analyzer View does not open a `gdiff` window. To open a `gdiff` window that shows the requested changes to the source file before compiling the modified code, toggle the checkbox in Update > Run gdiff After Update (Figure 24, page 53).



```
Nest Loop-                                    Lines    Olid Fil
   2   do                                     84-86     9
   2   do                                     92-95    10
   2   do                                    102-104   11
   2   do I        I          MAIN           112-115   12
   2   do I        I          MAIN           122-125   13
   2   do I        I          MAIN           133-135   14
   2   do I        I          MAIN           143-145   15
   2   do I        I          MAIN           152-154   16
   2   do I        I          MAIN           162-164   17
```

Figure 24. Run gdiff After Update

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

```
cvpav*gDiff: True
```

2.10.2.2 Modifying the Source File Further

After running the `sed`script, to make additional changes before compiling the modified code, open an editor by toggling the Update > Run Editor After Update checkbox. (See Figure 25, page 54.) An `xwsh` window with `vi` running in it opens with the source code ready to be edited.

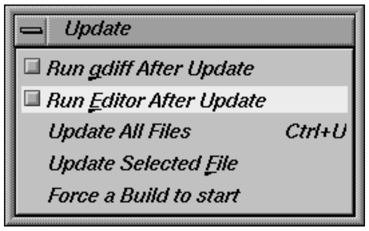


Figure 25. Setting the Checkbox for Run Editor After Update

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

```
cvpav*runUserEdit: True
```

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

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

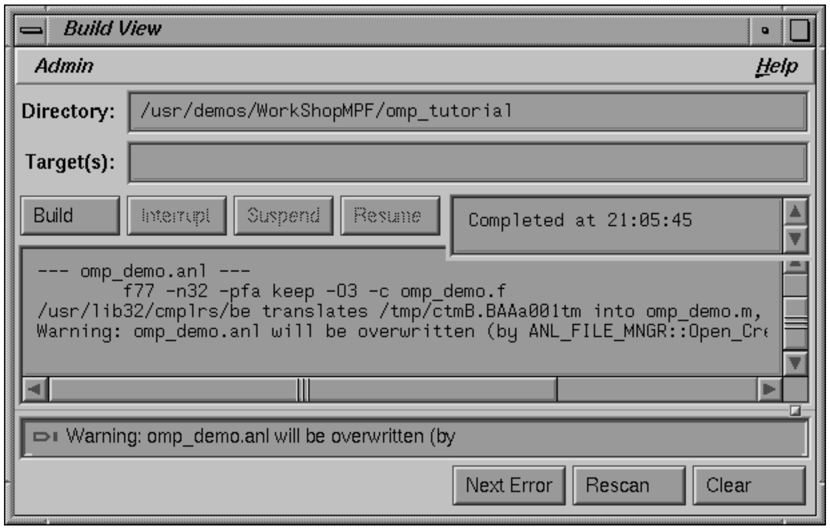
In the above command, the `+%d` tells `vi` at what line to position itself in the file and is replaced with `1` by default. (You can omit the `+%d` parameter if you wish.) The edited file’s name either replaces any explicit `%s`, or if the `%s` is omitted, its filename is appended to the command.

2.10.2.3 Updating the Source File

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

If you have set the checkbox and opened the `gdiff` window or an editor, examine the changes or edit the file as you wish. When you exit these tools, the

Parallel Analyzer View spawns the WorkShop Build Manager (Figure 26, page 55).



```
/usr/demos/WorkShopMPF/omp_tutorial
Completed at 21:05:45
--- omp_demo.anl ---
        f77 -n32 -pfa keep -O3 -c omp_demo.f
/usr/lib32/cmplrs/be translates /tmp/ctmB.BAAa001tm into omp_demo.m,
Warning: omp_demo.anl will be overwritten (by ANL_FILE_MNGR::Open_Cre
Warning: omp_demo.anl will be overwritten (by
```

Figure 26. Build View of Build Manager

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

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

2.10.3 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.

2.10.3.1 Added Assertion

Scroll to Olid 14 to see the effect of the assertion request made in Section 2.10.1.2, page 50. 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.

2.10.3.2 Deleted Assertion

Note that the assertion in loop Olid 15 is gone, as requested in Section 2.10.1.3, page 51, and that the loop no longer runs in parallel. Recall that the loop previously had the assertion that `foo()` was not an obstacle to parallelization.

2.11 Examples Using OpenMP Directives

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

- Section 2.11.1, page 56
- Section 2.11.2, page 59
- Section 2.11.3, page 60
- Section 2.11.4, page 61
- Section 2.11.5, page 61

For more information on OpenMP directives, see the *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual* or the OpenMP Architecture Review Board Web site: `http://www.openmp.org`.

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

To select the first parallel region, double-click the highlighted line in the loop list, Olid 92.

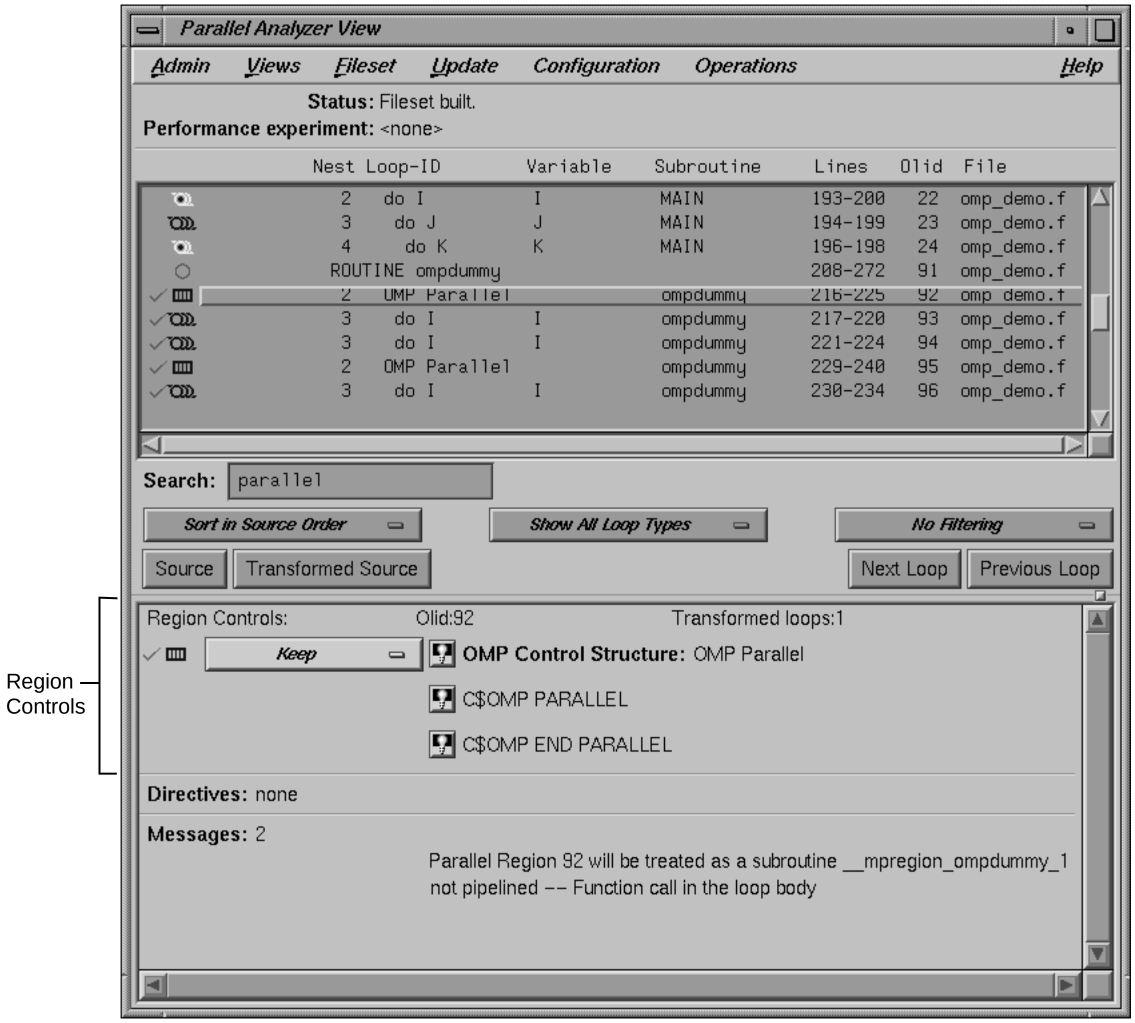
2.11.1 Explicitly Parallelized Loops: `C$OMP DO`

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

Example 18: Explicitly Parallelized Loop Using `C$OMP DO`

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

Notice in Figure 27, page 58, that the controls in the loop information display are now labelled Region Controls. The controls now affect the entire region. The `Keep option` button and the highlight buttons function the same way they do in the Loop Parallelization Controls. (See Section 2.6.4.1, page 23.)



Status: Fileset built.
Performance experiment: <none>

```
Nest Loop-ID         Variable   Subroutine     Lines    Olid  File
2    do I            I          MAIN           193-200  22    omp_demo.f
3     do J           J          MAIN           194-199  23    omp_demo.f
4      do K          K          MAIN           196-198  24    omp_demo.f
ROUTINE ompdummy                               208-272  91    omp_demo.f
2    OMP Parallel               ompdummy       216-225  92    omp_demo.f
3     do I           I          ompdummy       217-220  93    omp_demo.f
3     do I           I          ompdummy       221-224  94    omp_demo.f
2    OMP Parallel               ompdummy       229-240  95    omp_demo.f
3     do I           I          ompdummy       230-234  96    omp_demo.f
```

Search: parallel

Region Controls: Olid:92 Transformed loops:1

OMP Control Structure: OMP Parallel

C$OMP PARALLEL

C$OMP END PARALLEL

Directives: none

Messages: 2

Parallel Region 92 will be treated as a subroutine __mpregion_ompdummy_1
not pipelined -- Function call in the loop body

Figure 27. Loops Explicitly Parallelized Using C$OMP DO

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

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

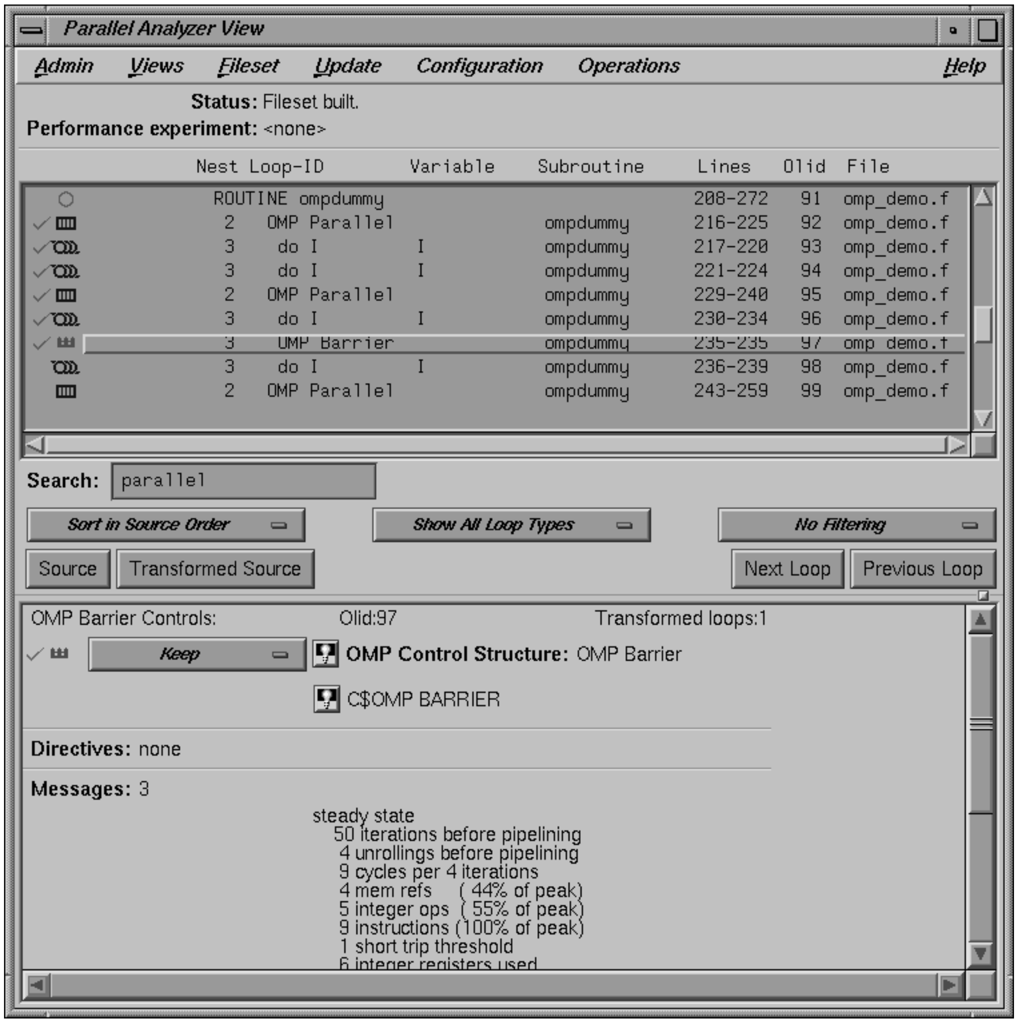
2.11.2 Loops With Barriers: `C$OMP BARRIER`

The second parallel region, Olid 95, contains a pair of loops that are identical to the previous example except for a barrier between them. Because of the barrier, all iterations of the first `C$OMP DO` loop must complete before any iteration of the second loop can begin.

Example 19: Loops Using `C$OMP BARRIER`

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

Click `Next Loop` twice to view the barrier region. (See Figure 28, page 60.)



Status: Fileset built.
Performance experiment: <none>

```
                Nest Loop-ID       Variable    Subroutine     Lines     Olid  File
                    ROUTINE ompdummy                           208-272    91   omp_demo.f
                  2    OMP Parallel             ompdummy       216-225    92   omp_demo.f
                  3     do I          I         ompdummy       217-220    93   omp_demo.f
                  3     do I          I         ompdummy       221-224    94   omp_demo.f
                  2    OMP Parallel             ompdummy       229-240    95   omp_demo.f
                  3     do I          I         ompdummy       230-234    96   omp_demo.f
                  3     OMP Barrier             ompdummy       235-235    97   omp_demo.f
                  3     do I          I         ompdummy       236-239    98   omp_demo.f
                  2    OMP Parallel             ompdummy       243-259    99   omp_demo.f
```

Search: parallel

OMP Barrier Controls: Olid:97 Transformed loops:1

OMP Control Structure: OMP Barrier

C$OMP BARRIER

Directives: none

Messages: 3

```
steady state
   50 iterations before pipelining
    4 unrollings before pipelining
    9 cycles per 4 iterations
    4 mem refs      ( 44% of peak)
    5 integer ops   ( 55% of peak)
    9 instructions (100% of peak)
    1 short trip threshold
    6 integer registers used
```

Figure 28. Loops Using C$OMP BARRIER Synchronization

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

2.11.3 Critical Sections: `C$OMP CRITICAL`

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

Example 20: Critical Section Using `C$OMP CRITICAL`

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

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

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

2.11.4 Single-Process Sections: `C$OMP SINGLE`

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

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

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

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

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

2.11.5 Parallel Sections: `C$OMP SECTIONS`

The fourth and final parallel region of `ompdummy()` provides an example of parallel sections. In this case, there are three parallel subsections, each of which calls a function. Each function is called exactly once, by a single thread. If there are three or more threads in the program, each function may be called from a different thread. The compiler treats this directive as a single-process directive, which guarantees correct semantics.

Example 22: Parallel Sections Using `C$OMP SECTIONS`

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

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

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

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

2.12 Examples Using Data Distribution Directives

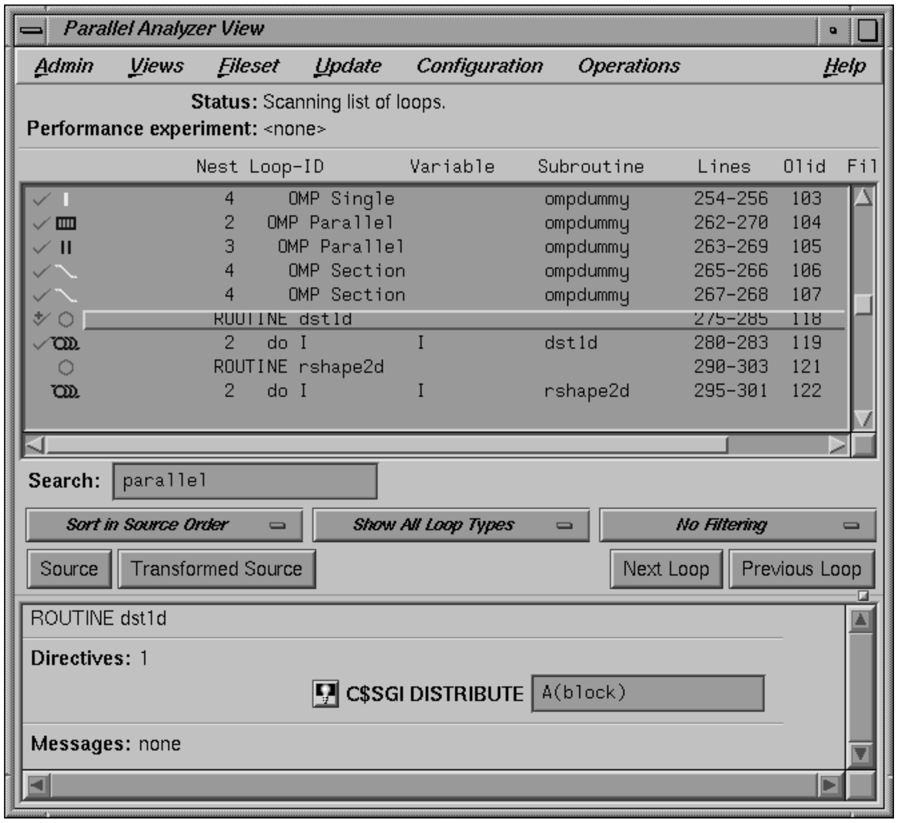
The next series of subroutines illustrate directives that control data distribution and cache storage. The following three directives are discussed:

- Section 2.12.1, page 62
- Section 2.12.2, page 64
- Section 2.12.3, page 65

Descriptions of these directives appear in Table 3, page 131.

2.12.1 Distributed Arrays: `C$SGI DISTRIBUTE`

When you select the subroutine `dst1d()`, a directive is listed in the loop information display that is global to the subroutine. The directive, `C$SGI DISTRIBUTE`, specifies placement of array members in distributed, shared memory. (See Figure 29, page 63.)



Status: Scanning list of loops.
Performance experiment: <none>

```
             Nest Loop-ID        Variable    Subroutine     Lines    Olid  Fil
                4     OMP Single             ompdummy       254-256  103
                2   OMP Parallel             ompdummy       262-270  104
                3    OMP Parallel            ompdummy       263-269  105
                4     OMP Section            ompdummy       265-266  106
                4     OMP Section            ompdummy       267-268  107
                  ROUTINE dst1d                             275-285  118
                2   do I             I       dst1d          280-283  119
                  ROUTINE rshape2d                          290-303  121
                2   do I             I       rshape2d       295-301  122
```

Search: parallel

ROUTINE dst1d

Directives: 1

C$SGI DISTRIBUTE A(block)

Messages: none

Figure 29. `C$SGI DISTRIBUTE` Directive and Text Field

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

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

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

Example 23: Distributed Array Using C$SGI DISTRIBUTE

```
        subroutine dst1d(a)

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

        return
```

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

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

2.12.2 Distributed and Reshaped Arrays: C$SGI DISTRIBUTE_RESHAPE

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

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

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

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

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

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

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

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

If the code had not had the `C$SGI&NEST` clause, you could have inserted it by supplying the arguments in the text field in the Parallelization Control View. You can use the `C$SGI&NEST` clause to parallelize nested loops only when both loops are fully parallel and there is no code between either the `do-i` and `do-j` statements or the `enddo-i` and `enddo-j` statements. (See Chapter 6 of the *MIPSpro Fortran 77 Programmer's Guide*.)

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

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

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

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

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

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

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

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

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

2.13 Exiting From the `omp_demo.f` Sample Session

This completes the first sample session.

Quit the Parallel Analyzer View by choosing Admin > Exit.

Not all windows opened during the session close when you quit the Parallel Analyzer View. In particular, the Source View remains open because all the Developer Magic tools interoperate, and other tools may share the Source View window. (See Section 2.6.1.1, page 16.) You must close the Source View independently.

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

```
% make clean
```

Examining Loops for Fortran 90 Code [3]

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

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

This chapter describes the following:

- How to set up the sample session. See Section 3.1, page 67.
- Compiling the code. See Section 3.2, page 67.
- Starting the parallel analyzer. See Section 3.3, page 68.
- Demonstrating array statement transformations. See Section 3.4, page 68.
- Exiting from the session. See Section 3.5, page 73.

3.1 Setting Up the Sample Session

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

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

3.2 Compiling the Sample Code

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

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

This creates the following files:

`f90_tutorial.f90`	A copy of the demonstration program created by copying `f90_tutorial.f90_orig`

`f90_tutorial.m`	A transformed source file, which you can view with the `Parallel Analyzer View` and print
`f90_tutorial.l`	A listing file
`f90_tutorial.anl`	An analysis file used by the `Parallel Analyzer View`

For more information about these files, see the *MIPSpro Auto-Parallelizing Option Programmer’s Guide*.

3.3 Starting the Parallel Analyzer View

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

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

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

3.4 Demonstrating Array Statement Transformations

This section demonstrates the following transformations:

- Transforming a simple array statement into a `DO` loop. See Section 3.4.1, page 68.
- Transforming a single array statement into nested `DO` loops. See Section 3.4.2, page 69.
- Transforming a simple array statement into a subroutine. See Section 3.4.3, page 71.

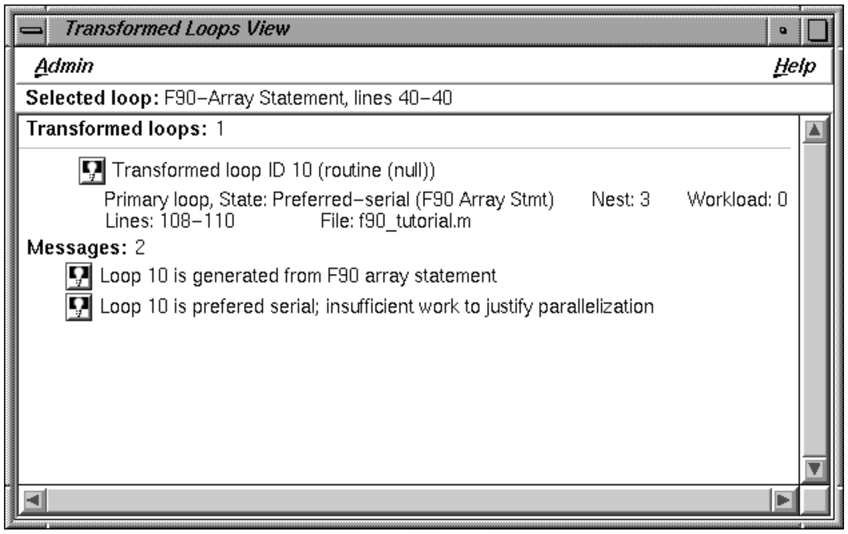
3.4.1 Transforming an Array Statement into a DO Loop

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

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

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

l = .true.
```

The `Transformed Loops View` window (see Figure 30, page 69) identifies line 40 from the source as a Fortran 90 array statement. It notes that a loop was generated but indicates that the loop array statement was not made parallel because it contains too little work.



Selected loop: F90–Array Statement, lines 40–40

Transformed loops: 1

Transformed loop ID 10 (routine (null))

Primary loop, State: Preferred–serial (F90 Array Stmt) Nest: 3 Workload: 0
Lines: 108–110 File: f90_tutorial.m

Messages: 2

Loop 10 is generated from F90 array statement

Loop 10 is prefered serial; insufficient work to justify parallelization

Figure 30. Array Statement into `DO` Loop

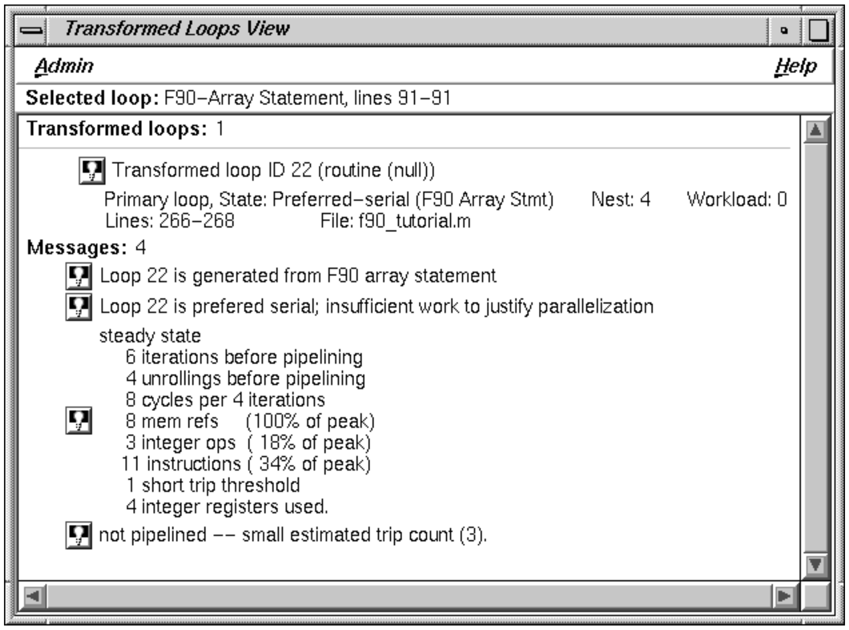
3.4.2 Transforming an Array Statement in Nested DO Loops

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

The following is the array statement in the source:

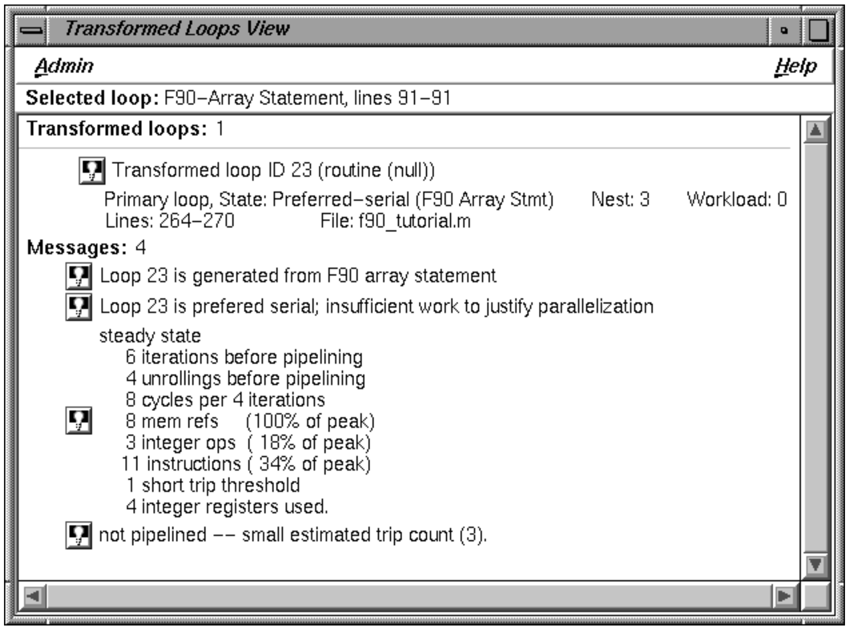
```
logical*8 l(3,12)
.
.
.
l = .true.
```

Because the array has two dimensions, two nested `DO` loops are generated. Double-click first on loop 22, then on loop 23. They are the two new loops generated from the array statement. The `Transformed Loops View` window gives information on each loop. (See Figure 31, page 70 for loop 22 and Figure 32, page 71 for loop 23.)



```
Selected loop: F90–Array Statement, lines 91–91
Transformed loops: 1
    Transformed loop ID 22 (routine (null))
      Primary loop, State: Preferred–serial (F90 Array Stmt)    Nest: 4    Workload: 0
      Lines: 266–268            File: f90_tutorial.m
Messages: 4
    Loop 22 is generated from F90 array statement
    Loop 22 is prefered serial; insufficient work to justify parallelization
    steady state
       6 iterations before pipelining
       4 unrollings before pipelining
       8 cycles per 4 iterations
       8 mem refs     (100% of peak)
       3 integer ops  ( 18% of peak)
      11 instructions ( 34% of peak)
       1 short trip threshold
       4 integer registers used.
    not pipelined –– small estimated trip count (3).
```

Figure 31. Loop 22



Selected loop: F90–Array Statement, lines 91–91

Transformed loops: 1

Transformed loop ID 23 (routine (null))
Primary loop, State: Preferred–serial (F90 Array Stmt) Nest: 3 Workload: 0
Lines: 264–270 File: f90_tutorial.m

Messages: 4

Loop 23 is generated from F90 array statement

Loop 23 is prefered serial; insufficient work to justify parallelization

```
steady state
    6 iterations before pipelining
    4 unrollings before pipelining
    8 cycles per 4 iterations
    8 mem refs     (100% of peak)
    3 integer ops  ( 18% of peak)
   11 instructions ( 34% of peak)
    1 short trip threshold
    4 integer registers used.
```

not pipelined –– small estimated trip count (3).

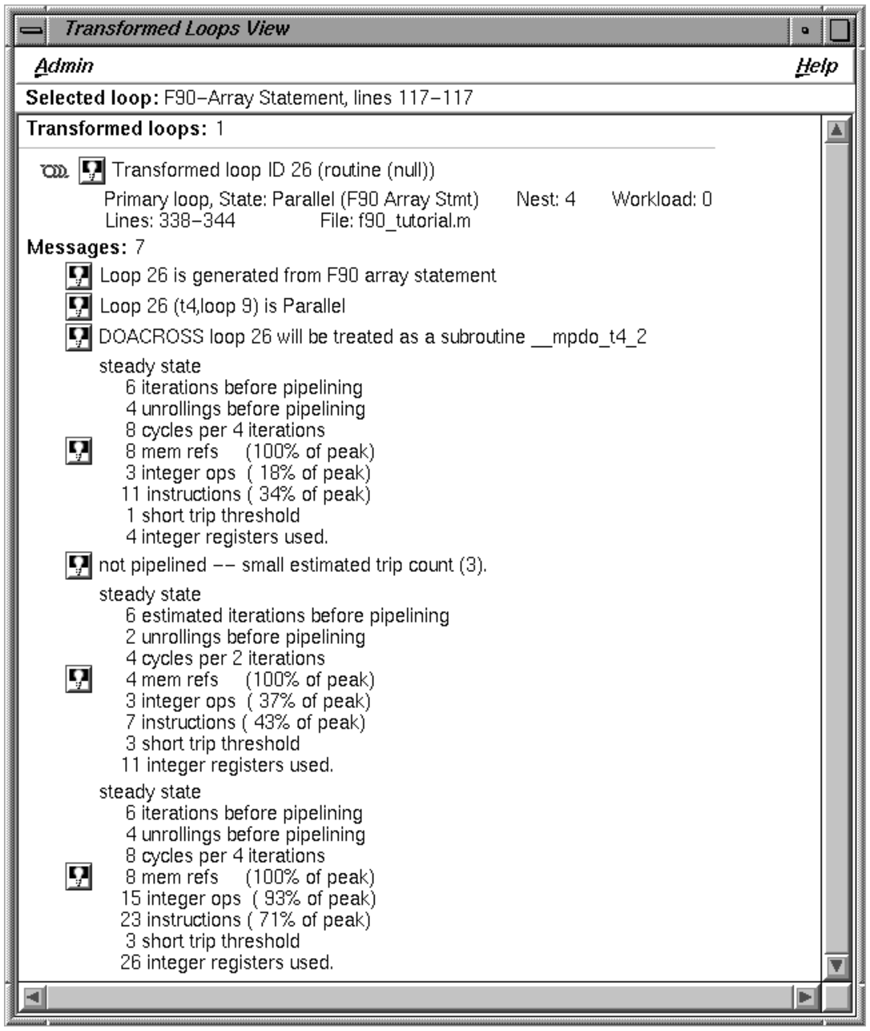
Figure 32. Loop 23

3.4.3 Transforming an Array Statement into a Subroutine

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

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

The `Transformed Loops View` (see Figure 33, page 72) shows the process of converting first to a parallel loop and then to a subroutine:



```
Selected loop: F90–Array Statement, lines 117–117
Transformed loops: 1
    Transformed loop ID 26 (routine (null))
        Primary loop, State: Parallel (F90 Array Stmt)    Nest: 4    Workload: 0
        Lines: 338–344              File: f90_tutorial.m
Messages: 7
    Loop 26 is generated from F90 array statement
    Loop 26 (t4,loop 9) is Parallel
    DOACROSS loop 26 will be treated as a subroutine __mpdo_t4_2
    steady state
       6 iterations before pipelining
       4 unrollings before pipelining
       8 cycles per 4 iterations
       8 mem refs     (100% of peak)
       3 integer ops  ( 18% of peak)
      11 instructions ( 34% of peak)
       1 short trip threshold
       4 integer registers used.
    not pipelined -- small estimated trip count (3).
    steady state
       6 estimated iterations before pipelining
       2 unrollings before pipelining
       4 cycles per 2 iterations
       4 mem refs     (100% of peak)
       3 integer ops  ( 37% of peak)
       7 instructions ( 43% of peak)
       3 short trip threshold
      11 integer registers used.
    steady state
       6 iterations before pipelining
       4 unrollings before pipelining
       8 cycles per 4 iterations
       8 mem refs     (100% of peak)
      15 integer ops  ( 93% of peak)
      23 instructions ( 71% of peak)
       3 short trip threshold
      26 integer registers used.
```

Figure 33. Array Statement into a Subroutine

3.5 Exiting From the Session

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

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

```
% make clean
```

Examining Loops for C Code [4]

This chapter presents another interactive sample session with the `Parallel Analyzer View`. The session illustrates aspects of the MIPSpro Auto-Parallelizing C compiler. For tutorials using other compilers, see the following sections:

- Fortran 77, see Chapter 2, page 5.
- Fortran 90, see Chapter 3, page 67.

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

The following sections comprise the C parallel analyzer session:

- Setting Up the `c_tutorial.c` Sample Session, see Section 4.1, page 76.
- Compiling the Sample Code, see Section 4.2, page 76.
- Starting the `Parallel Analyzer View` Tutorial, see Section 4.3, page 76.
- Examples of Simple Loops, see Section 4.4, page 77.
- Examining Loops With Obstacles to Parallelization, see Section 4.5, page 80.
- Examining Nested Loops, see Section 4.6, page 86.
- Modifying Source Files and Compiling, see Section 4.7, page 87.
- Examples Using OpenMP Directives, see Section 4.8, page 96.
- Examples Using Data Distribution Directives, see Section 4.9, page 99.
- Exiting From the `c_tutorial.c` Sample Session, see Section 4.10, page 103.

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

For more details about the `Parallel Analyzer View` interface, see Chapter 6, page 113.

4.1 Setting Up the `c_tutorial.c` Sample Session

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

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

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

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

4.2 Compiling the Sample Code

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

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

Doing this creates the following file:

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

For more information about these files, see the *MIPSpro Auto-Parallelizing Option Programmer's Guide*.

4.3 Starting the Parallel Analyzer View Tutorial

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

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

Note: If you receive a message related to licensing, refer to the *ProDev ProMP Release Notes*.

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

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

```
% make clean
```

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

4.4 Examples of Simple Loops

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

- Simple parallel loop, see Section 4.4.1, page 77.
- Serial loop, see Section 4.4.2, page 78.
- Explicitly parallelized loop, see Section 4.4.3, page 78.
- Fused loops, see Section 4.4.4, page 80.
- Eliminated loop, see Section 4.4.5, page 80.

Two other sections discuss more complicated loops:

- Examining loops with obstacles to parallelization, see Section 4.5, page 80.
- Examining nested loops, see Section 4.6, page 86.

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

4.4.1 Simple Parallel Loop

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

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

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

Move to the next loop by selecting Olid 6.

4.4.2 Serial Loop

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

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

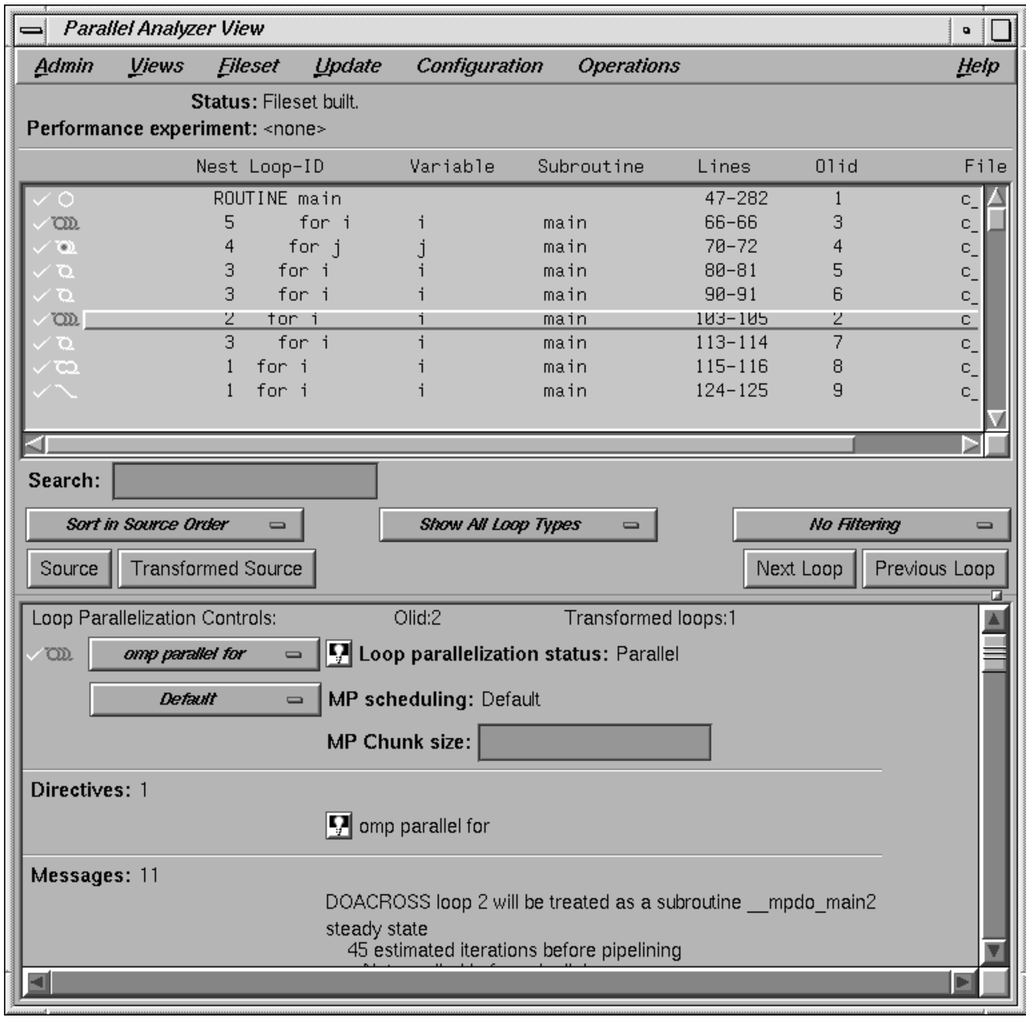
Move to the Olid 2 loop.

4.4.3 Explicitly Parallelized Loop

Loop Olid 2 is parallelized because it contains an explicit `#pragma omp parallel for` directive in the source, as shown in the Loop Parallelization Controls area of the window (see Figure 34, page 79). The compiler passes the directive through to the transformed source.

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

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



Status: Fileset built.
Performance experiment: <none>

```
                  Nest Loop-ID        Variable    Subroutine     Lines      Olid         File
                    ROUTINE main                                 47-282       1          c_
                     5      for i       i           main         66-66        3          c_
                     4     for j        j           main         70-72        4          c_
                     3    for i         i           main         80-81        5          c_
                     3    for i         i           main         90-91        6          c_
                     2   for i          i           main        103-105       2          c
                     3    for i         i           main        113-114       7          c_
                     1   for i          i           main        115-116       8          c_
                     1   for i          i           main        124-125       9          c_
```

Loop Parallelization Controls: Olid:2 Transformed loops:1

Loop parallelization status: Parallel

MP scheduling: Default

MP Chunk size:

Directives: 1

omp parallel for

Messages: 11

DOACROSS loop 2 will be treated as a subroutine __mpdo_main2
steady state
   45 estimated iterations before pipelining

Figure 34. Explicitly Parallelized Loop

If you clicked on the highlight button, close the `Parallelization Control View`. (Using the `Parallelization Control View` is discussed in Section 4.7.1.1, page 88.) Close the `Source View` and move to the next loop by clicking the `Next Loop` button.

4.4.4 Fused Loops

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

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

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

4.4.5 Loop That Is Eliminated

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

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

Move to the next loop, Olid 10, by clicking the `Next Loop` button. This loop is discussed in Section 4.5.1.1, page 81.

4.5 Examining Loops With Obstacles to Parallelization

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

- Carried Data Dependence, seeSection 4.5.1, page 81.
- Input/Output Operations, see Section 4.5.2, page 84.
- Function Calls, see Section 4.5.3, page 84.

- Permutation Vectors, see Section 4.5.4, page 85.

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

Messages that appear in the graphical user interface offer further tips on obstacles to parallelization. See Section 2.8.6, page 39 for two tables that list messages generated by the compiler that concern obstacles to parallelization.

4.5.1 Carried Data Dependence

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

- Unparallelizable Carried Data Dependence, see Section 4.5.1.1, page 81.
- Parallelizable Carried Data Dependence, see Section 4.5.1.2, page 83.
- Multi-line Data Dependence, see Section 4.5.1.3, page 83.
- Reductions, see Section 4.5.1.4, page 83.

4.5.1.1 Unparallelizable Carried Data Dependence

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

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

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

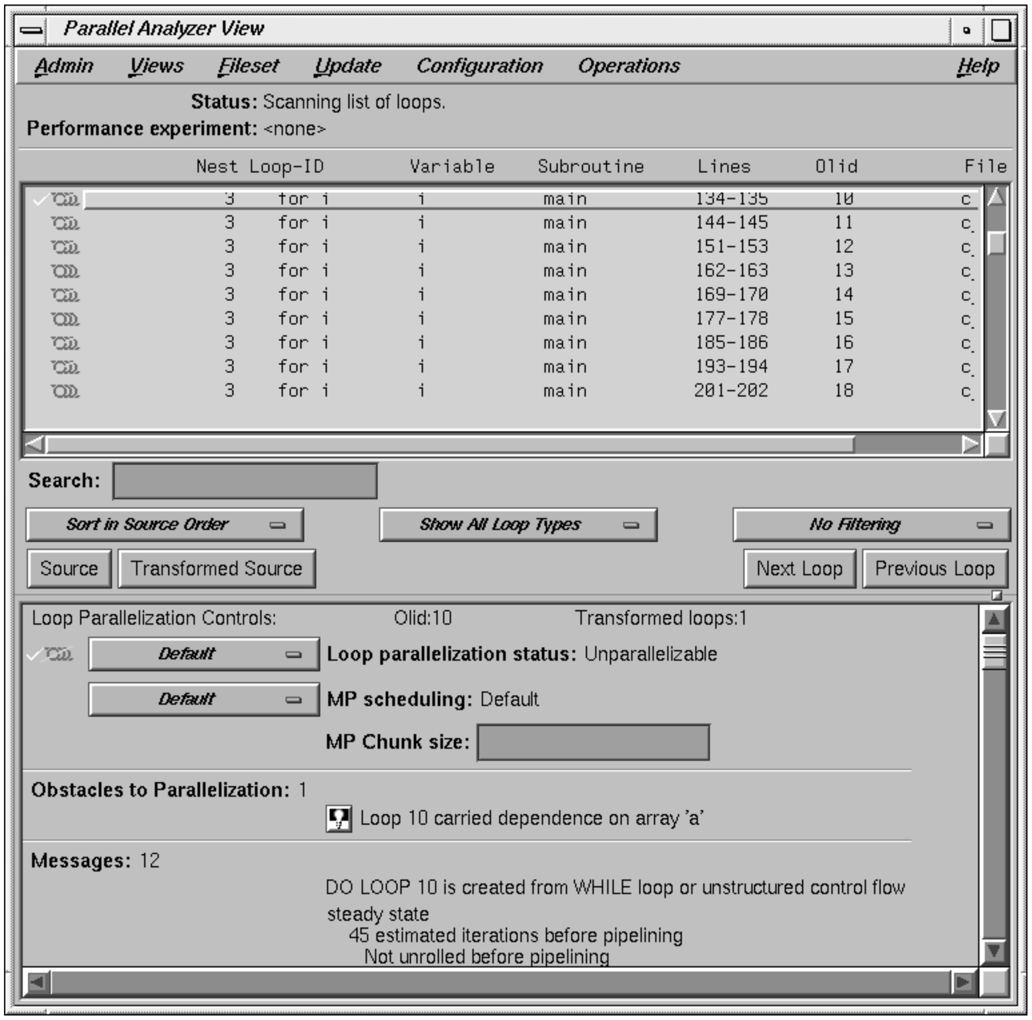
The loop information display in Figure 35, page 82, lists the obstacle to parallelization.

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

- The relevant line that has the dependence.

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

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



Status: Scanning list of loops.
Performance experiment: <none>

```
Nest Loop-ID   Variable  Subroutine  Lines    Olid  File
   3  for i    i         main        134-135  10    c
   3  for i    i         main        144-145  11    c
   3  for i    i         main        151-153  12    c
   3  for i    i         main        162-163  13    c
   3  for i    i         main        169-170  14    c
   3  for i    i         main        177-178  15    c
   3  for i    i         main        185-186  16    c
   3  for i    i         main        193-194  17    c
   3  for i    i         main        201-202  18    c
```

Loop Parallelization Controls: Olid:10 Transformed loops:1

Loop parallelization status: Unparallelizable

MP scheduling: Default

MP Chunk size:

Obstacles to Parallelization: 1

Loop 10 carried dependence on array 'a'

Messages: 12

```
DO LOOP 10 is created from WHILE loop or unstructured control flow
steady state
   45 estimated iterations before pipelining
    Not unrolled before pipelining
```

Figure 35. Obstacles to Parallelization

4.5.1.2 Parallelizable Carried Data Dependence

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

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

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

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

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

4.5.1.3 Multi-line Data Dependence

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

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

Click the highlight button on the obstacle line.

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

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

4.5.1.4 Reductions

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

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

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

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

4.5.2 Input/Output Operations

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

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

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

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

4.5.3 Function Calls

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

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

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

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

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

4.5.4 Permutation Vectors

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

4.5.4.1 Unparallelizable Loop With a Permutation Vector

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

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

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

4.5.4.2 Parallelizable Loop With a Permutation Vector

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

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

Move to the next loop, Olid 18, by clicking `Next Loop`. This loop is discussed in Section 4.6.1, page 86.

4.6 Examining Nested Loops

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

4.6.1 Doubly Nested Loop

Loop Olid 18 is the outer loop of a pair of loops, and it runs in parallel. The inner loop runs in serial because the compiler knows that one parallel loop should not be nested inside another. However, you can force parallelization in for the inner loop by inserting a `#pragma omp parallel for` directive in front of the outer loop. For example, see Section 4.9.2, page 101.

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

Click `Next Loop` to move to Olid 19.

4.6.2 Doubly Nested Loop

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

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

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

4.6.3 Triple Nested Loop

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

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

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

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

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

4.7 Modifying Source Files and Compiling

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

You control most of the directives and some of the assertions available from the `Parallel Analyzer View` with the `Operations` menu . You control most of the assertions and the more complex directives, `#pragma omp for` and `#pragma omp parallel for`, with the loop parallelization status option button (see Figure 36, page 89).

There are two steps to modifying source files:

1. Making changes using the `Parallel Analyzer View` controls, discussed in the next subsection, Section 4.7.1, page 88.
2. Modifying the source and rebuilding the program and its analysis files, discussed in Section 4.7.2, page 95.

4.7.1 Making Changes

You make changes by one of the following actions:

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

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

These are the changes discussed:

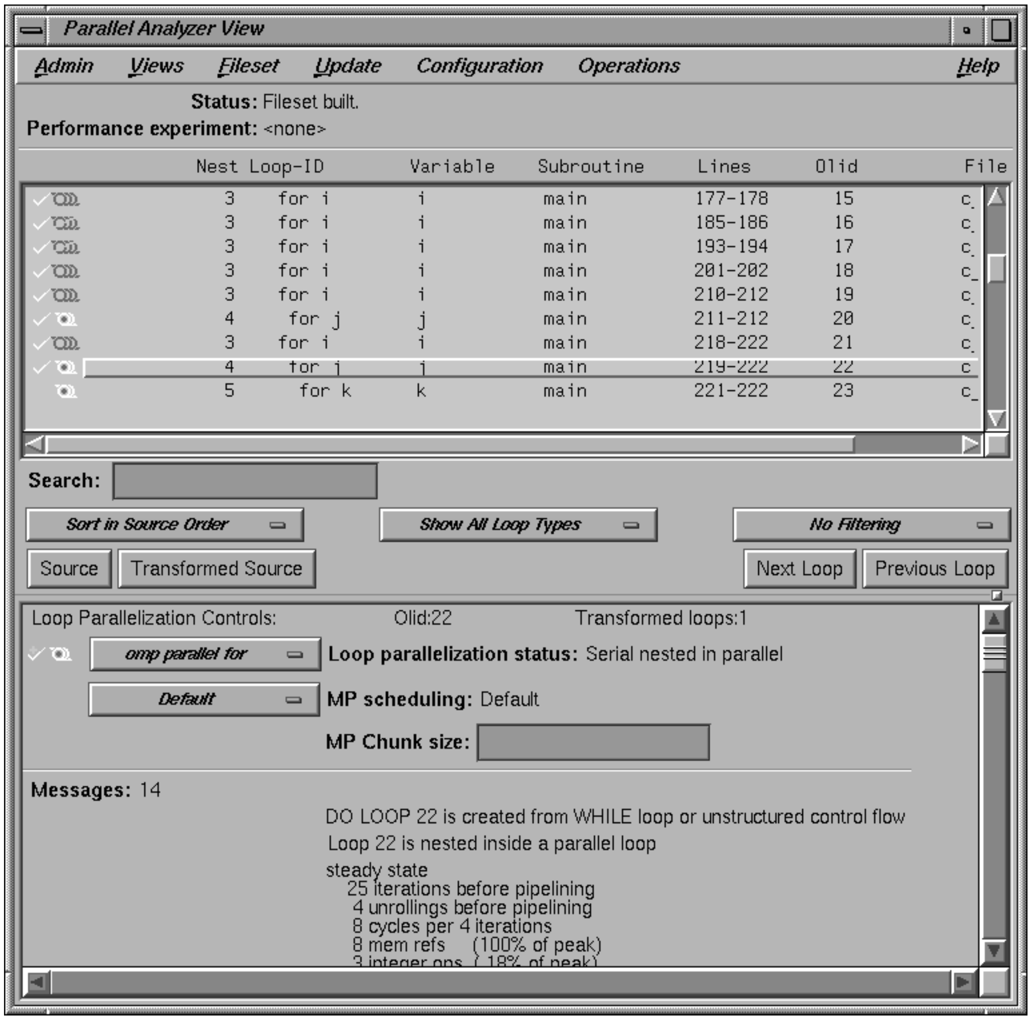
- Adding `#pragma omp parallel for` Directives and Clauses, see Section 4.7.1.1, page 88.
- Adding New Assertions or Directives With the Operations Menu, see Section 4.7.1.2, page 91.
- Deleting Assertions or Directives, see Section 4.7.1.3, page 92.

4.7.1.1 Adding `#pragma omp parallel for` Directives and Clauses

Loop Olid 22, shown in Section 4.6.3, page 86, is a serial loop nested inside a parallel loop. It is not parallelized, but its iterations could run concurrently.

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

1. Make sure loop Olid 22 is selected.
2. Click on the loop parallelization status option button (see Figure 36, page 89) and choose `omp parallel for` to parallelize Olid 22.



Status: Fileset built.
Performance experiment: <none>

```
     Nest Loop-ID   Variable   Subroutine   Lines     Olid   File
✓    3    for i     i          main         177-178   15     c
✓    3    for i     i          main         185-186   16     c
✓    3    for i     i          main         193-194   17     c
✓    3    for i     i          main         201-202   18     c
✓    3    for i     i          main         210-212   19     c
✓    4     for j    j          main         211-212   20     c
✓    3    for i     i          main         218-222   21     c
✓    4     for j    j          main         219-222   22     c
     5      for k   k          main         221-222   23     c
```

Loop Parallelization Controls: Olid:22 Transformed loops:1

Loop parallelization status: Serial nested in parallel

MP scheduling: Default

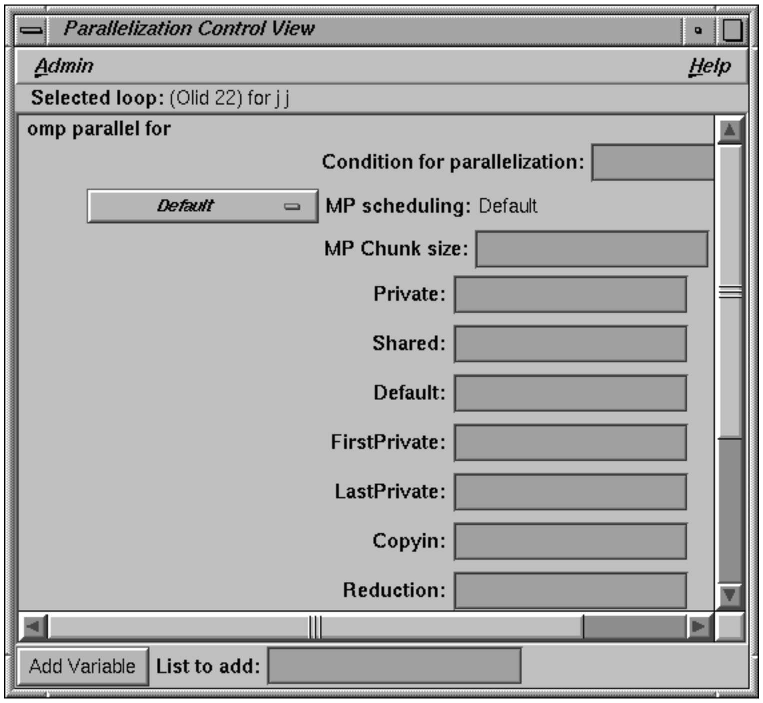
MP Chunk size:

Messages: 14

```
DO LOOP 22 is created from WHILE loop or unstructured control flow
Loop 22 is nested inside a parallel loop
steady state
   25 iterations before pipelining
    4 unrollings before pipelining
    8 cycles per 4 iterations
    8 mem refs     (100% of peak)
    3 integer ops  ( 18% of peak)
```

Figure 36. Creating a Parallel Directive

This sequence requests a change in the source code and opens the `Parallelization Control View` (see Figure 37, page 90). You can now look at variables in the loop and attach clauses to the directive, if needed.



Selected loop: (Olid 22) for j j

omp parallel for

Condition for parallelization:

MP scheduling: Default (Default)

MP Chunk size:

Private:

Shared:

Default:

FirstPrivate:

LastPrivate:

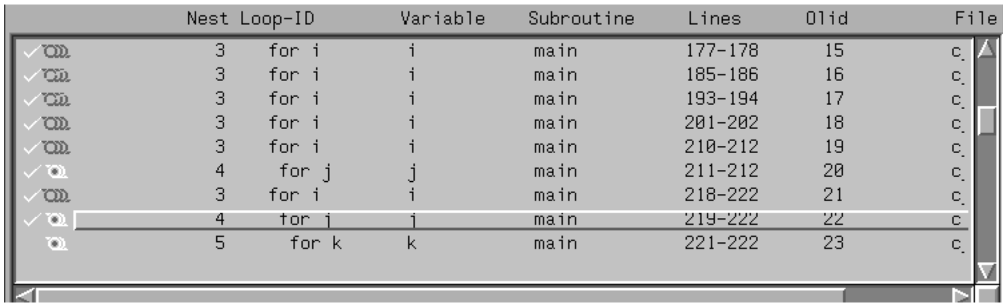
Copyin:

Reduction:

List to add:

Figure 37. `Parallelization Control View`

Notice that in the loop list there is now a red plus sign next to this loop, indicating that a change has been requested. (See Figure 38, page 90.)



```
            Nest Loop-ID       Variable    Subroutine    Lines      Olid        File
✓              3    for i        i           main        177-178     15           c
✓              3    for i        i           main        185-186     16           c
✓              3    for i        i           main        193-194     17           c
✓              3    for i        i           main        201-202     18           c
✓              3    for i        i           main        210-212     19           c
✓              4     for j       j           main        211-212     20           c
✓              3    for i        i           main        218-222     21           c
✓              4     for j       j           main        219-222     22           c
               5      for k      k           main        221-222     23           c
```

Figure 38. Changed Loop List

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

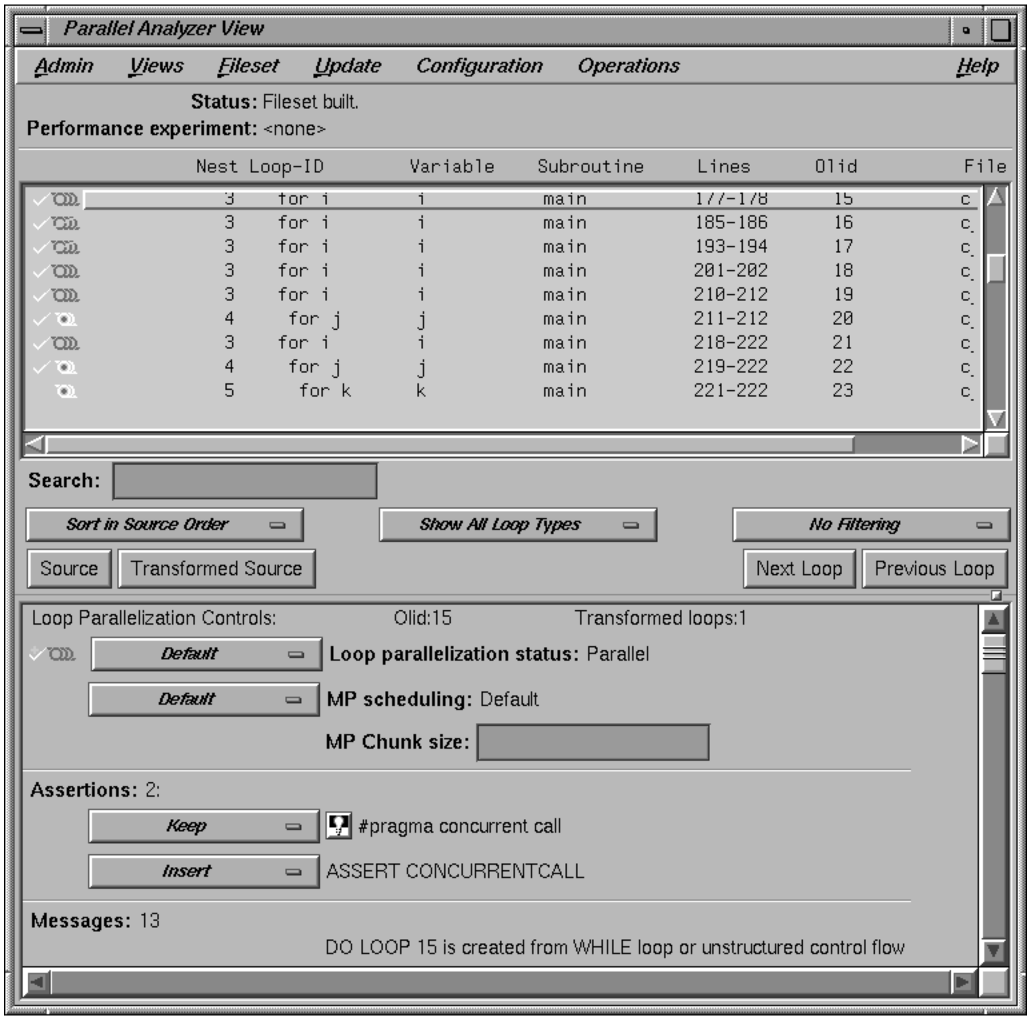
4.7.1.2 Adding New Assertions or Directives With the Operations Menu

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

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

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

The loop information display shows the new assertion, along with an `Insert` button to indicate the state of the assertion when you modify the code. (See Figure 39, page 92.)



Status: Fileset built.
Performance experiment: <none>

```
Nest Loop-ID    Variable  Subroutine  Lines    Olid   File
   3  for i       i       main       177-178   15     c
   3  for i       i       main       185-186   16     c
   3  for i       i       main       193-194   17     c
   3  for i       i       main       201-202   18     c
   3  for i       i       main       210-212   19     c
   4   for j      j       main       211-212   20     c
   3  for i       i       main       218-222   21     c
   4   for j      j       main       219-222   22     c
   5    for k     k       main       221-222   23     c
```

Loop Parallelization Controls: Olid:15 Transformed loops:1

Loop parallelization status: Parallel

MP scheduling: Default

MP Chunk size:

Assertions: 2:

#pragma concurrent call

ASSERT CONCURRENTCALL

Messages: 13

DO LOOP 15 is created from WHILE loop or unstructured control flow

Figure 39. Adding an Assertion

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

4.7.1.3 Deleting Assertions or Directives

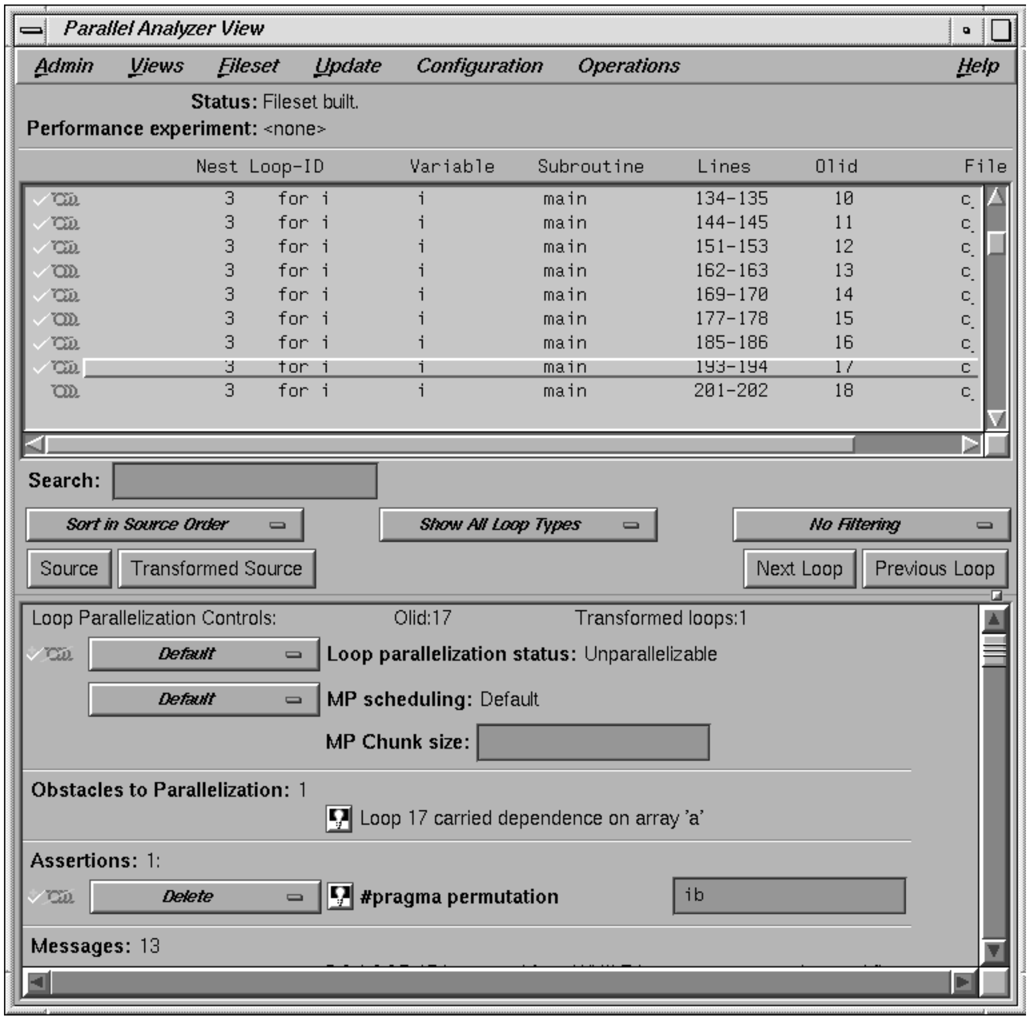
Move to loop Olid 17 (shown in Section 4.5.4.2, page 85).

To delete an assertion, follow these steps:

1. Find the assertion `#pragma permutation(ib)` in the loop information display.

2. Select its `Delete` option button.

Figure 40, page 94, shows the state of the assertion in the information display. A similar procedure is used to delete directives.



Status: Fileset built.
Performance experiment: <none>

```
Nest Loop-ID      Variable    Subroutine    Lines      Olid       File
 3    for i        i           main        134-135     10         c
 3    for i        i           main        144-145     11         c
 3    for i        i           main        151-153     12         c
 3    for i        i           main        162-163     13         c
 3    for i        i           main        169-170     14         c
 3    for i        i           main        177-178     15         c
 3    for i        i           main        185-186     16         c
 3    for i        i           main        193-194     17         c
 3    for i        i           main        201-202     18         c
```

Search:

Loop Parallelization Controls: Olid:17 Transformed loops:1

Loop parallelization status: Unparallelizable

MP scheduling: Default

MP Chunk size:

Obstacles to Parallelization: 1

Loop 17 carried dependence on array 'a'

Assertions: 1:

#pragma permutation ib

Messages: 13

Figure 40. Deleting an Assertion

For information on applying changes and viewing the changes in a `gdiff` window, see Section 4.7.2, page 95.

4.7.2 Updating the Source File

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

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

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

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

4.7.3 Examining the Modified Source File

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

4.7.3.1 Added Assertion

Scroll to Olid 15 to see the effect of the assertion request made in Section 4.7.1.2, page 91. Notice the icon indicating that loop Olid 15, which previously was unparallelizable because of the call to the function `foo`, is now parallel.

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

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

4.7.3.2 Deleted Assertion

Note that the assertion in loop Olid 16 is gone, as requested in Section 4.7.1.3, page 92, and that the loop no longer runs in parallel. Recall that the loop previously had the assertion that `ib` was not an obstacle to parallelization.

4.8 Examples Using OpenMP Directives

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

- Explicitly Parallelized Loops: `#pragma omp for`, see Section 4.8.1, page 96.
- Loops With Barriers: `#pragma omp barrier`, see Section 4.8.2, page 97.
- Critical Sections: `#pragma omp critical`, see Section 4.8.3, page 98.
- Single-Process Sections: `#pragma omp single`, see Section 4.8.4, page 98.
- Parallel Sections: `#pragma omp sections`, see Section 4.8.5, page 98.

For more information on OpenMP directives, see the *MIPSpro C and C++ Pragmas* or the OpenMP Architecture Review Board Web site: `http://www.openmp.org`.

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

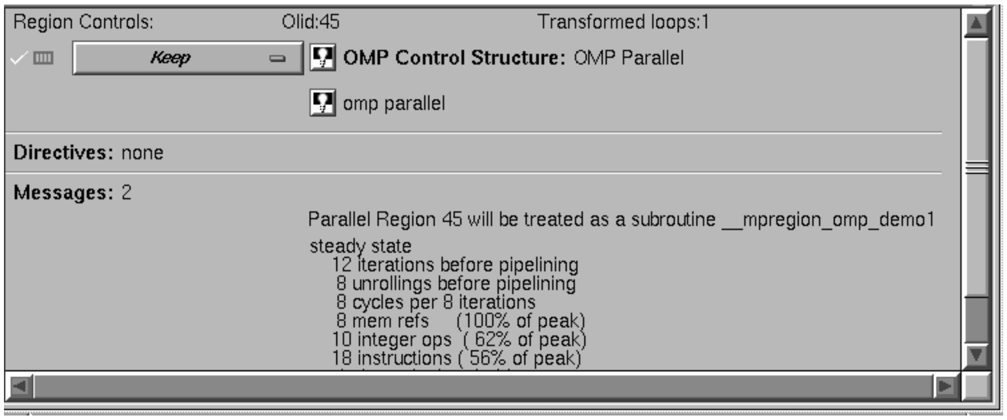
To select the first parallel region, double-click the highlighted line in the loop list, Olid 53.

4.8.1 Explicitly Parallelized Loops: `#pragma omp for`

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

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

Notice in Figure 41, page 97, that the controls in the loop information display are now labelled `Region Controls`. The controls now affect the entire region. The `Keep` option button and the highlight buttons function the same way as in the Loop Parallelization Controls.



```
Parallel Region 45 will be treated as a subroutine __mpregion_omp_demo1
steady state
   12 iterations before pipelining
    8 unrollings before pipelining
    8 cycles per 8 iterations
    8 mem refs     (100% of peak)
   10 integer ops  ( 62% of peak)
   18 instructions ( 56% of peak)
```

Figure 41. Loops Explicitly Parallelized Using `#pragma omp for`

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

4.8.2 Loops With Barriers: `#pragma omp barrier`

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

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

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

4.8.3 Critical Sections: #pragma omp critical

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

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

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

Move to the next loop by using Next Loop.

4.8.4 Single-Process Sections: #pragma omp single

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

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

} /* omp end parallel */
```

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

4.8.5 Parallel Sections: #pragma omp sections

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

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

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

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

4.9 Examples Using Data Distribution Directives

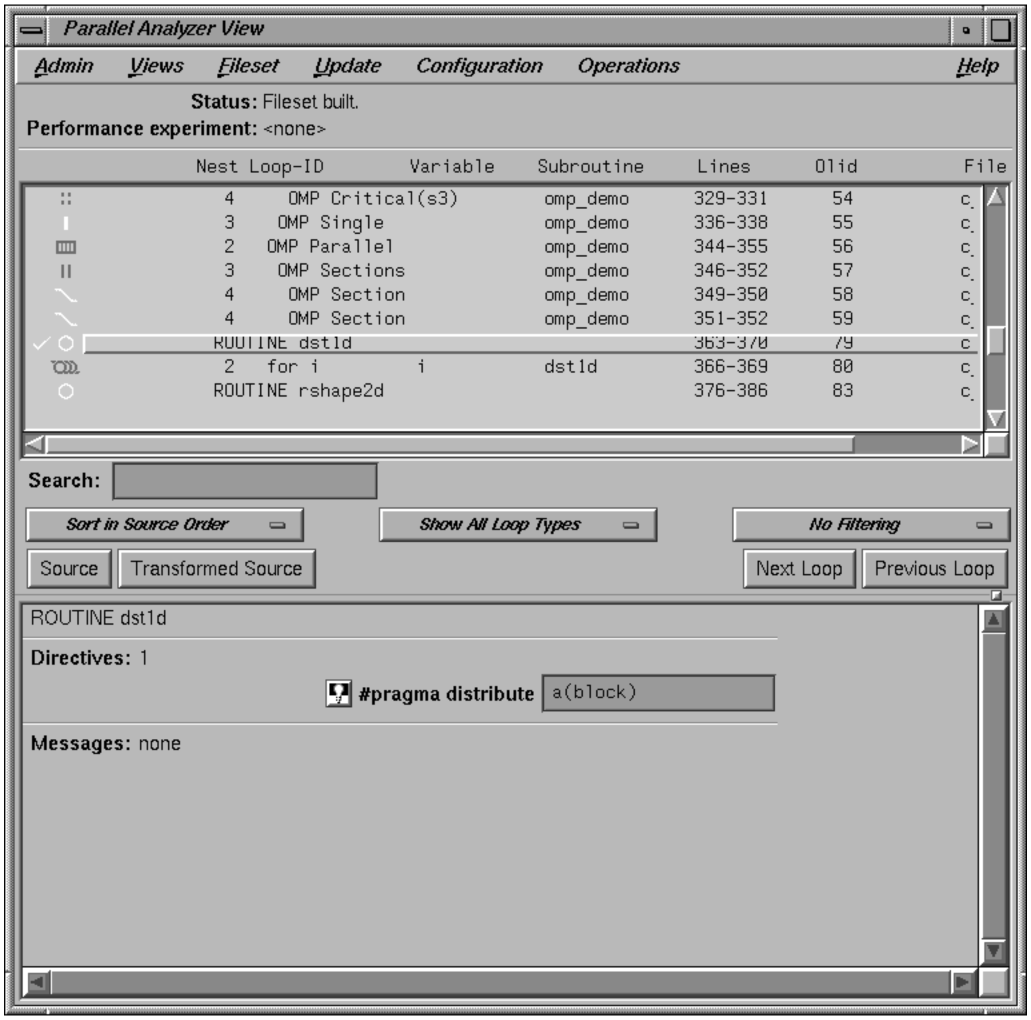
The next series of functions illustrates directives that control data distribution and cache storage. The following topics are described in this section:

- Distributed Arrays: `#pragma distribute`, see Section 4.9.1, page 99.
- Distributed and Reshaped Arrays: `#pragma distribute_reshape`, see Section 4.9.2, page 101.
- Prefetching Data From Cache: `#pragma prefetch_ref`, see Section 4.9.3, page 102.

Brief descriptions of these directives appear in Table 3, page 131.

4.9.1 Distributed Arrays: `#pragma distribute`

When you select the function `dst1d()`, a parallelized loop icon is listed in the loop information display. The `#pragma distribute` directive specifies placement of array members in distributed, shared memory. (See Figure 42, page 100.)



Status: Fileset built.
Performance experiment: <none>

```
Nest Loop-ID            Variable   Subroutine   Lines     Olid    File
  4      OMP Critical(s3)          omp_demo     329-331    54     c
  3     OMP Single                 omp_demo     336-338    55     c
  2    OMP Parallel                omp_demo     344-355    56     c
  3     OMP Sections               omp_demo     346-352    57     c
  4      OMP Section               omp_demo     349-350    58     c
  4      OMP Section               omp_demo     351-352    59     c
    ROUTINE dst1d                               363-370    79     c
  2    for i            i          dst1d        366-369    80     c
    ROUTINE rshape2d                            376-386    83     c
```

ROUTINE dst1d

Directives: 1

#pragma distribute a(block)

Messages: none

Figure 42. `#pragma distribute` Directive and Text Field

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

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

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

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

}
```

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

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

4.9.2 Distributed and Reshaped Arrays: `#pragma distribute_reshape`

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

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

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

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

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

}
```

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

For more information on the #pragma distribute_reshape directive, see Chapter 13 of the *C Language Reference Manual*.

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

Click Next Loop to view the prfetch function.

4.9.3 Prefetching Data From Cache: #pragma prefetch_ref

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

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

      int i, j;

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

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

4.10 Exiting From the Sample Session

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

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

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

```
% make clean
```

Using WorkShop With Parallel Analyzer View [5]

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

This sample session examines LINPACK, a standard benchmark designed to measure CPU performance in solving dense linear equations. 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 previous chapters. You can also consult Chapter 6, page 113, for more information.

5.1 Setting Up the linpackd Sample Session

Start by entering the following commands:

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

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

5.1.1 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 loop list and the Source View. (Click the `Source` button to open it.) 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.

5.1.2 Starting the Performance Analyzer

Pull down Admin > Launch Tool > Performance Analyzer to start the Performance Analyzer, as shown in Figure 43, page 107.

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.

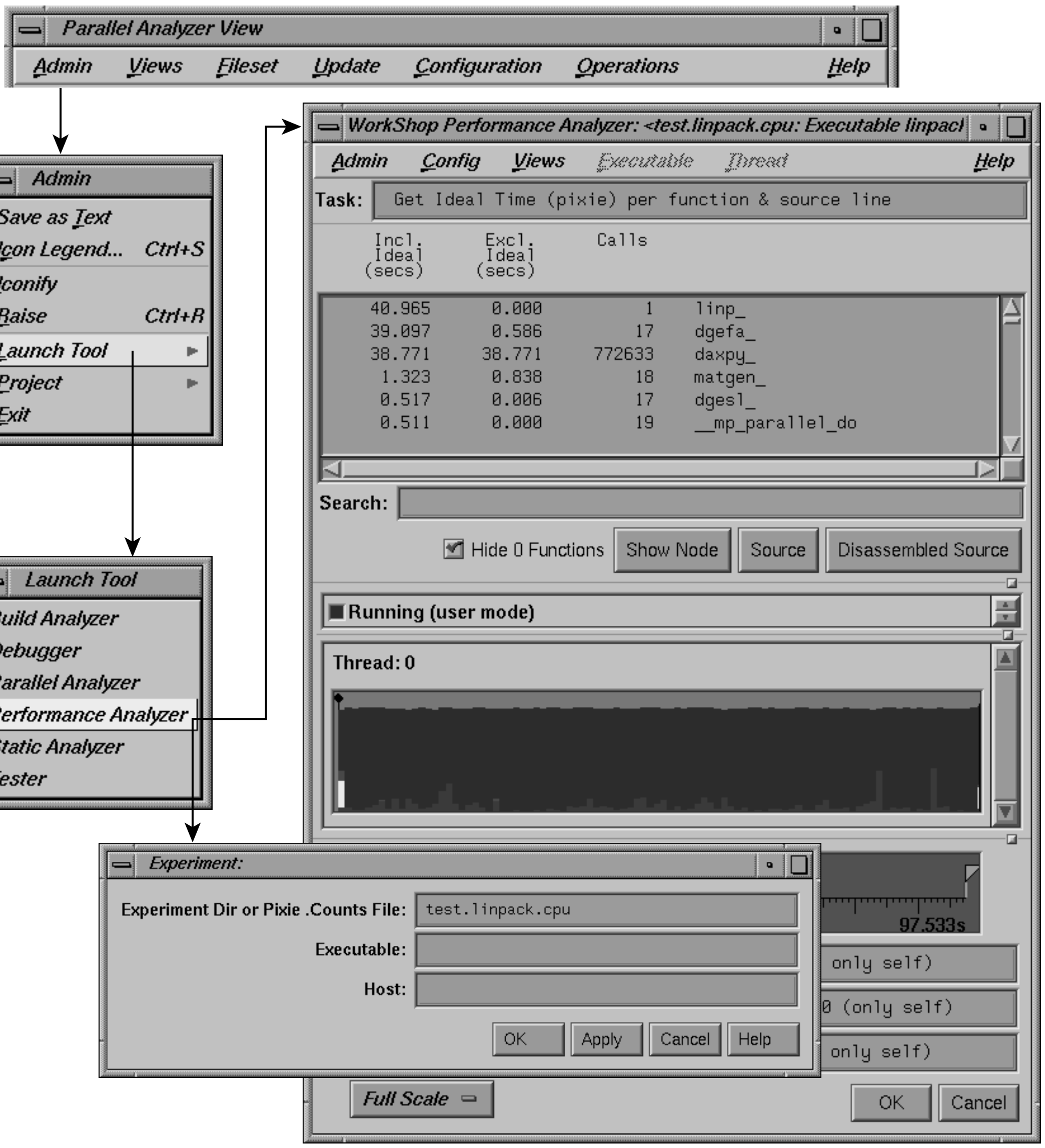
In the Experiment Dir ...: text field in the Experiment: window, enter

`test.linpack.cpu`

Click the `OK` button. (See Figure 43, page 107.)

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

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



Task: Get Ideal Time (pixie) per function & source line

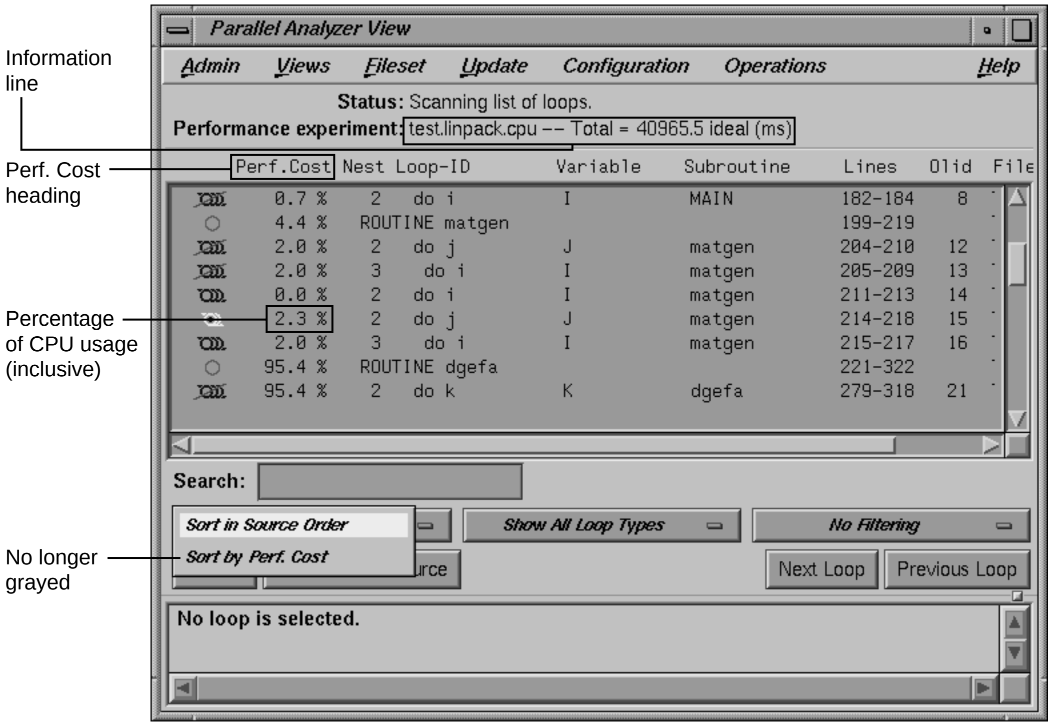
```
  Incl.      Excl.       Calls
  Ideal      Ideal
  (secs)     (secs)
  40.965      0.000          1    linp_
  39.097      0.586         17    dgefa_
  38.771     38.771     772633    daxpy_
   1.323      0.838         18    matgen_
   0.517      0.006         17    dgesl_
   0.511      0.000         19    __mp_parallel_do
```

Experiment Dir or Pixie .Counts File: test.linpack.cpu
Executable:
Host:

Figure 43. Starting the Performance Analyzer

5.2 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 44, page 108.



```
Status: Scanning list of loops.
Performance experiment: test.linpack.cpu -- Total = 40965.5 ideal (ms)

Perf.Cost Nest Loop-ID         Variable   Subroutine   Lines     Olid  File
  0.7 %    2   do i            I          MAIN         182-184    8
  4.4 %   ROUTINE matgen                               199-219
  2.0 %    2   do j            J          matgen       204-210   12
  2.0 %    3    do i           I          matgen       205-209   13
  0.0 %    2   do i            I          matgen       211-213   14
  2.3 %    2   do j            J          matgen       214-218   15
  2.0 %    3    do i           I          matgen       215-217   16
 95.4 %   ROUTINE dgefa                                221-322
 95.4 %    2   do k            K          dgefa        279-318   21

No loop is selected.
```

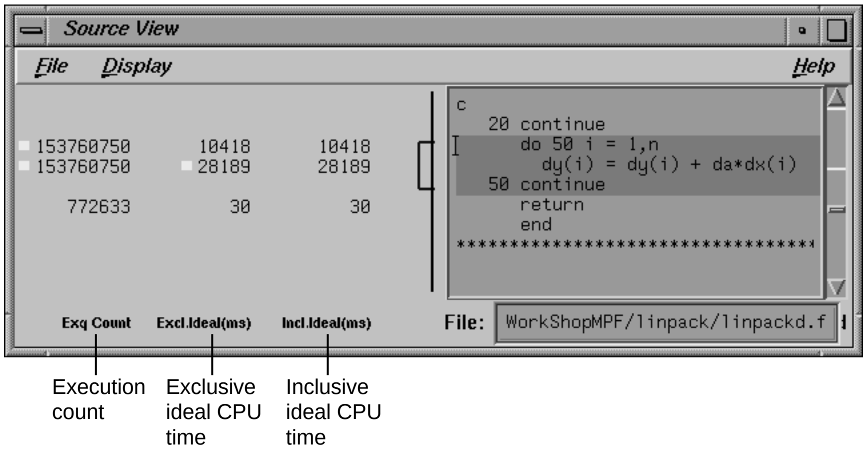
Figure 44. Parallel Analyzer View — Performance Data Loaded

- A new column, Perf. Cost, appears in the loop list next to the icon column. The values in this column are inclusive: each reflects the time spent in the loop and in any nested loops or functions called from within the loop.
- The Performance experiment line, in the main view below the menu bar, now shows the name of the performance experiment and the total cost of the run in milliseconds.
- The `Sort by Perf.Cost` option of the sort option button is now available.

- In the Source View, three columns appear to the left of the loop brackets. (These columns may take a few moments to load.) They reflect the measured performance data:
  - Exq Count: the number of times the line has been executed
  - Excl Ideal(ms): exclusive, ideal CPU time in milliseconds
  - Incl Ideal(ms): inclusive, ideal CPU time in milliseconds

5.2.1 Effect of Performance Data on the Source View

To see the effect of the performance data on the Source View, select Olid 30, which is in subroutine `daxpy()`. The Source View appears as shown in Figure 45, page 109.

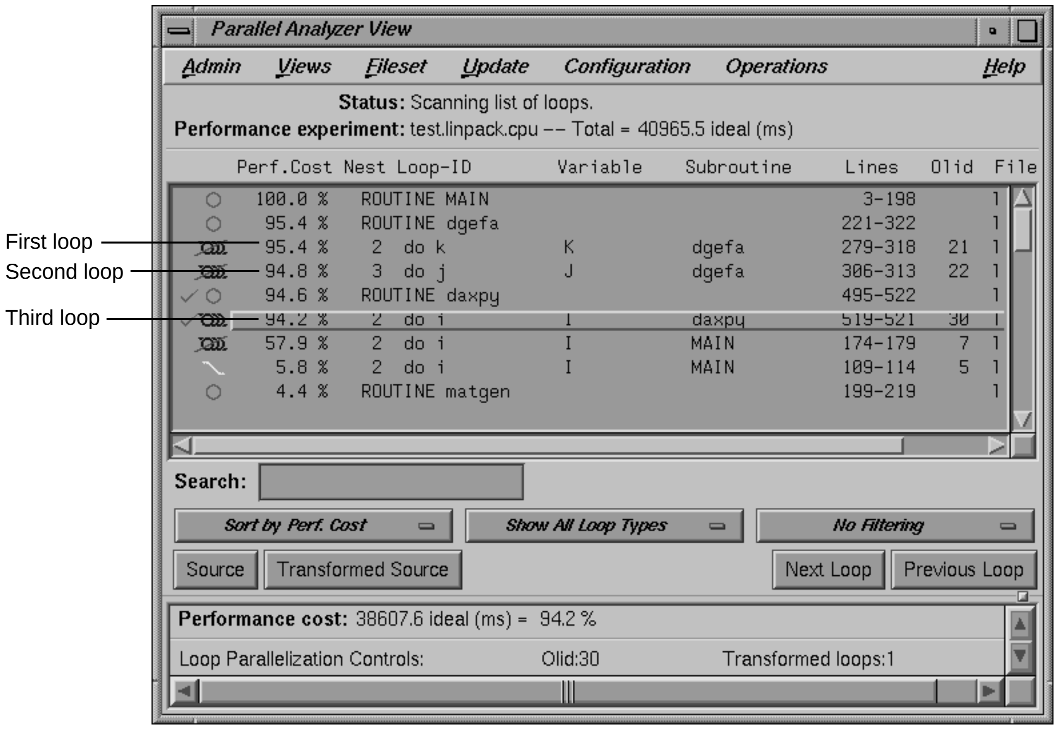


```
                                          c
                                             20 continue
153760750       10418       10418               do 50 i = 1,n
153760750       28189       28189                 dy(i) = dy(i) + da*dx(i)
                                             50 continue
   772633          30          30               return
                                                end
                                          ***********************************
```

Figure 45. Source View for Performance Experiment

5.2.2 Sorting the Loop List by Performance Cost

Choose the `Sort by Perf.Cost` sort option. Note that the third most expensive loop listed, Olid 30 of subroutine `daxpy()`, represents approximately 94% of the total time. (See Figure 46, page 110.)



Status: Scanning list of loops.
Performance experiment: test.linpack.cpu -- Total = 40965.5 ideal (ms)

```
   Perf.Cost Nest Loop-ID        Variable    Subroutine     Lines   Olid  File
     100.0 %    ROUTINE MAIN                                3-198          1
      95.4 %    ROUTINE dgefa                             221-322          1
      95.4 %     2  do k         K           dgefa        279-318     21   1
      94.8 %     3  do j         J           dgefa        306-313     22   1
      94.6 %    ROUTINE daxpy                             495-522          1
      94.2 %     2  do i         I           daxpy        519-521     30   1
      57.9 %     2  do i         I           MAIN         174-179      7   1
       5.8 %     2  do i         I           MAIN         109-114      5   1
       4.4 %    ROUTINE matgen                            199-219          1
```

Performance cost: 38607.6 ideal (ms) = 94.2 %

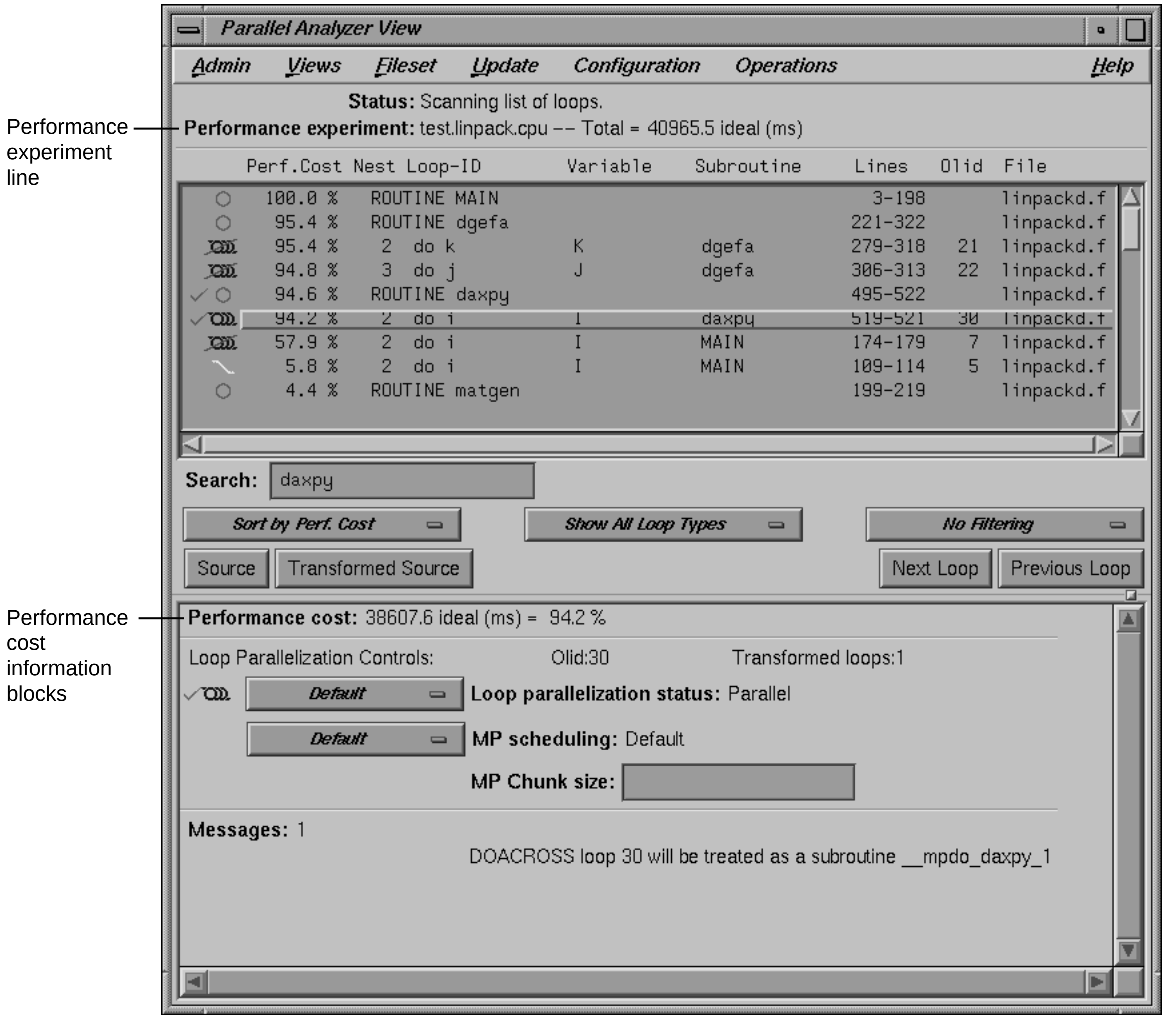
Loop Parallelization Controls: Olid:30 Transformed loops:1

Figure 46. Sort by Performance Cost

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

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

Double-click Olid 30. Note that the loop information display contains a line of text listing the performance cost of the loop, both in time and as a percentage of the total time. (See Figure 47, page 111.)



Status: Scanning list of loops.
Performance experiment: test.linpack.cpu -- Total = 40965.5 ideal (ms)

```
Perf.Cost Nest Loop-ID        Variable    Subroutine     Lines    Olid  File
  100.0 %   ROUTINE MAIN                                  3-198          linpackd.f
   95.4 %   ROUTINE dgefa                               221-322          linpackd.f
   95.4 %    2  do k           K          dgefa         279-318    21   linpackd.f
   94.8 %    3  do j           J          dgefa         306-313    22   linpackd.f
   94.6 %   ROUTINE daxpy                               495-522          linpackd.f
   94.2 %    2  do i           I          daxpy         519-521    30   linpackd.f
   57.9 %    2  do i           I          MAIN          174-179     7   linpackd.f
    5.8 %    2  do i           I          MAIN          109-114     5   linpackd.f
    4.4 %   ROUTINE matgen                              199-219          linpackd.f
```

Search: daxpy

Performance cost: 38607.6 ideal (ms) = 94.2 %

Loop Parallelization Controls: Olid:30 Transformed loops:1

Loop parallelization status: Parallel

MP scheduling: Default

MP Chunk size:

Messages: 1

DOACROSS loop 30 will be treated as a subroutine __mpdo_daxpy_1

Figure 47. Loop Information Display With Performance Data

5.3 Exiting From the linpackd Sample Session

This completes the second sample session.

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

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

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

```
% make clean
```

Parallel Analyzer View Reference [6]

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

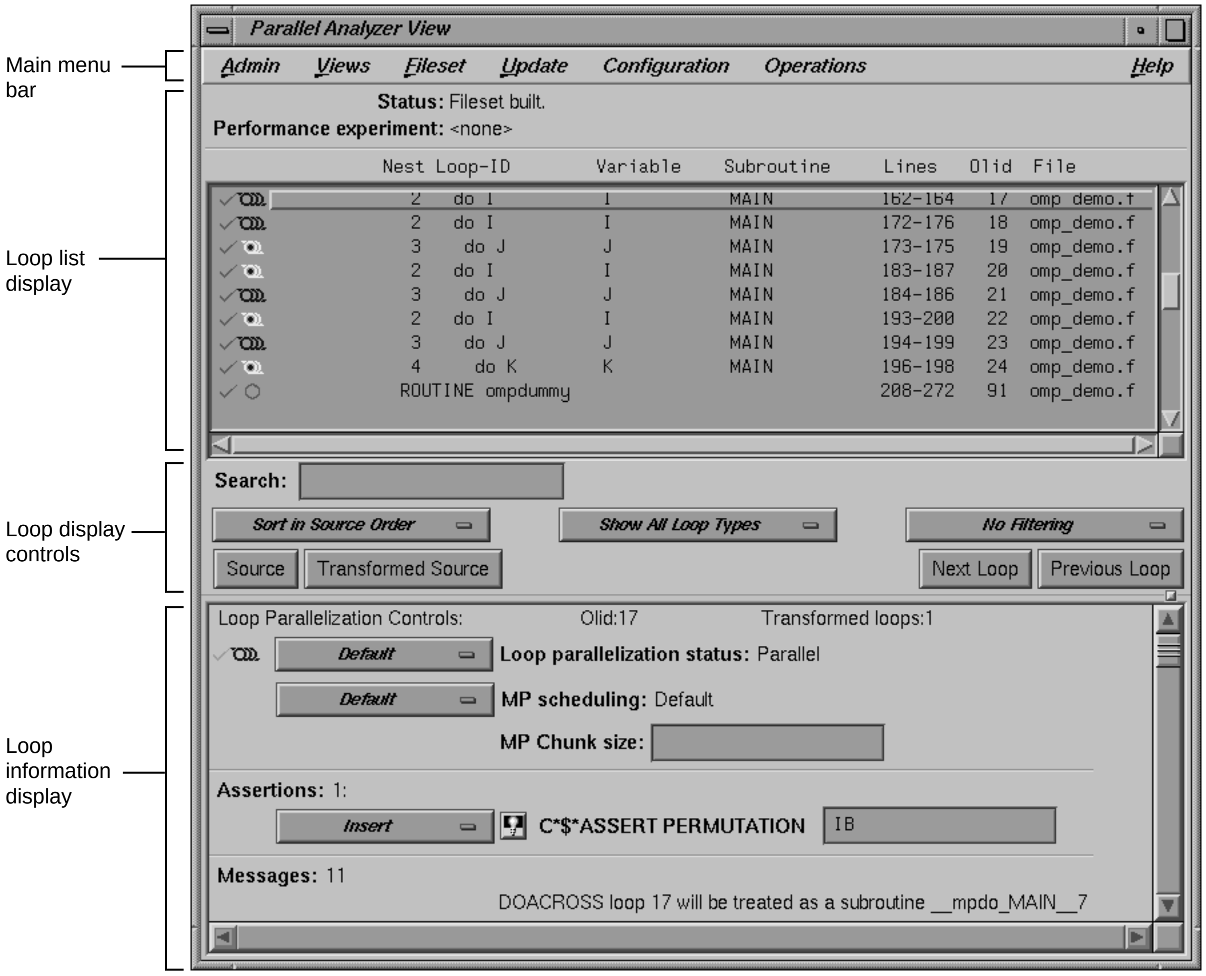
- Section 6.1, page 113
- Section 6.2, page 115
- Section 6.3, page 136
- Section 6.4, page 138
- Section 6.5, page 142
- Section 6.6, page 150
  - Section 6.6.1, page 150
  - Section 6.6.2, page 158
  - Section 6.6.3, page 159
  - Section 6.6.4, page 160
- Section 6.7, page 162
  - Section 6.7.1, page 162

6.1 Parallel Analyzer View Main Window

The main window is displayed when the Parallel Analyzer View begins. It consists of the following elements, shown in Figure 48, page 115:

- Main menu bar, containing these menus:
  - Admin: Discussed in Section 6.2.1, page 117.
  - Views: See Section 6.2.2, page 125.
  - Fileset: See discussion in Section 6.2.3, page 126.
  - Update: See Section 6.2.4, page 127.
  - Configuration: Find in Section 6.2.5, page 129.

  - Operations: See Section 6.2.6, page 130.
  - Help: See discussion in Section 6.2.7, page 134.
- Loop list display, which has the following members:
  - Status information: See Section 6.3.2, page 136.
  - Performance experiment information: Find in Section 6.3.2, page 136.
  - Loop list: See Section 6.3.3, page 137.
- Loop display controls, consisting of the following:
  - Search editable text field: See Section 6.4.1, page 139.
  - Three option buttons displaying default values: `Sort in Source Order`, `Show All Loop Types` and `No Filtering`. These buttons are described in Section 6.4.2, page 139, Section 6.4.3, page 140, and Section 6.4.4, page 141.
  - `Source` and `Transformed Source control` buttons: See Section 6.4.5, page 142.
  - `Next Loop` and `Previous Loop` loop list navigation buttons: See description in Section 6.4.5, page 142.
- Loop information display: See Section 6.5, page 142.



Status: Fileset built.
Performance experiment: <none>

```
Nest Loop-ID         Variable    Subroutine     Lines    Olid  File
 2   do I            I           MAIN           162-164   17   omp_demo.f
 2   do I            I           MAIN           172-176   18   omp_demo.f
 3    do J           J           MAIN           173-175   19   omp_demo.f
 2   do I            I           MAIN           183-187   20   omp_demo.f
 3    do J           J           MAIN           184-186   21   omp_demo.f
 2   do I            I           MAIN           193-200   22   omp_demo.f
 3    do J           J           MAIN           194-199   23   omp_demo.f
 4     do K          K           MAIN           196-198   24   omp_demo.f
ROUTINE ompdummy                                208-272   91   omp_demo.f
```

Search:

Loop Parallelization Controls: Olid:17 Transformed loops:1

Loop parallelization status: Parallel

MP scheduling: Default

MP Chunk size:

Assertions: 1:

C*$*ASSERT PERMUTATION IB

Messages: 11

DOACROSS loop 17 will be treated as a subroutine __mpdo_MAIN__7

Figure 48. Parallel Analyzer View Main Window

6.2 Parallel Analyzer View Menu Bar

This section describes the menus found in the menu bar located at the top of the Parallel Analyzer View main window as shown in Figure 49, page 116. The menus are discussed in these sections:

- Section 6.2.1, page 117
- Section 6.2.2, page 125

- Section 6.2.3, page 126
- Section 6.2.4, page 127
- Section 6.2.5, page 129
- Section 6.2.6, page 130
- Section 6.2.7, page 134

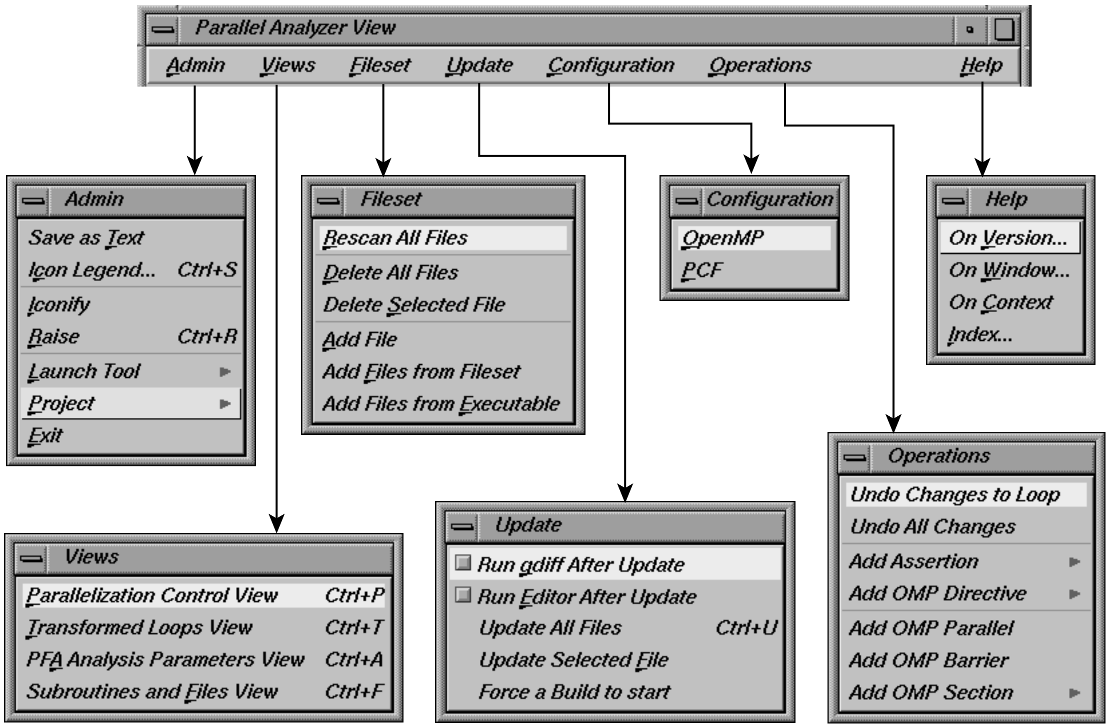


Figure 49. Parallel Analyzer View Menu Bar and Menus

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 Section 6.2.8, page 135.

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.

6.2.1 Admin Menu

Figure 50, page 117, shows the Parallel Analyzer View Admin menu, which contains file-writing commands, other administrative commands, and commands for launching and manipulating other WorkShop application views.

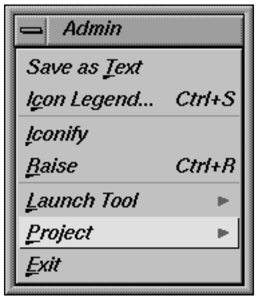


Figure 50. Admin Menu

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 Admin > Save as Text brings up a File Selection dialog, which lets you choose where to save the file and what name to call it. (See Figure 51, page 118.) 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.

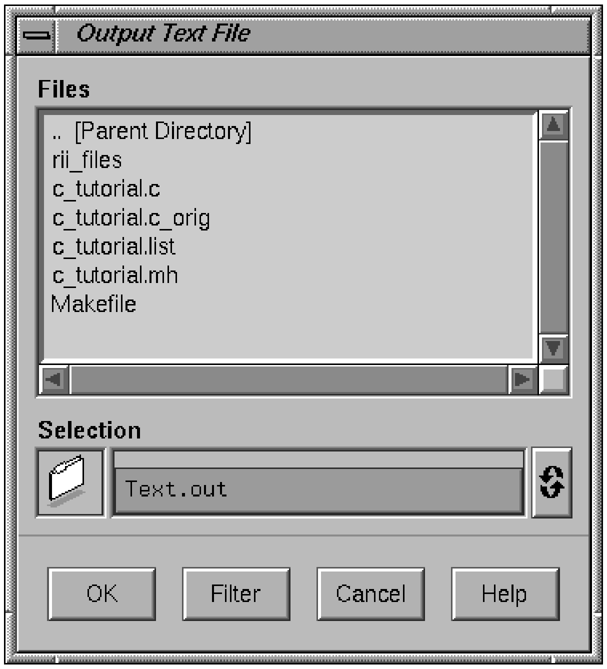
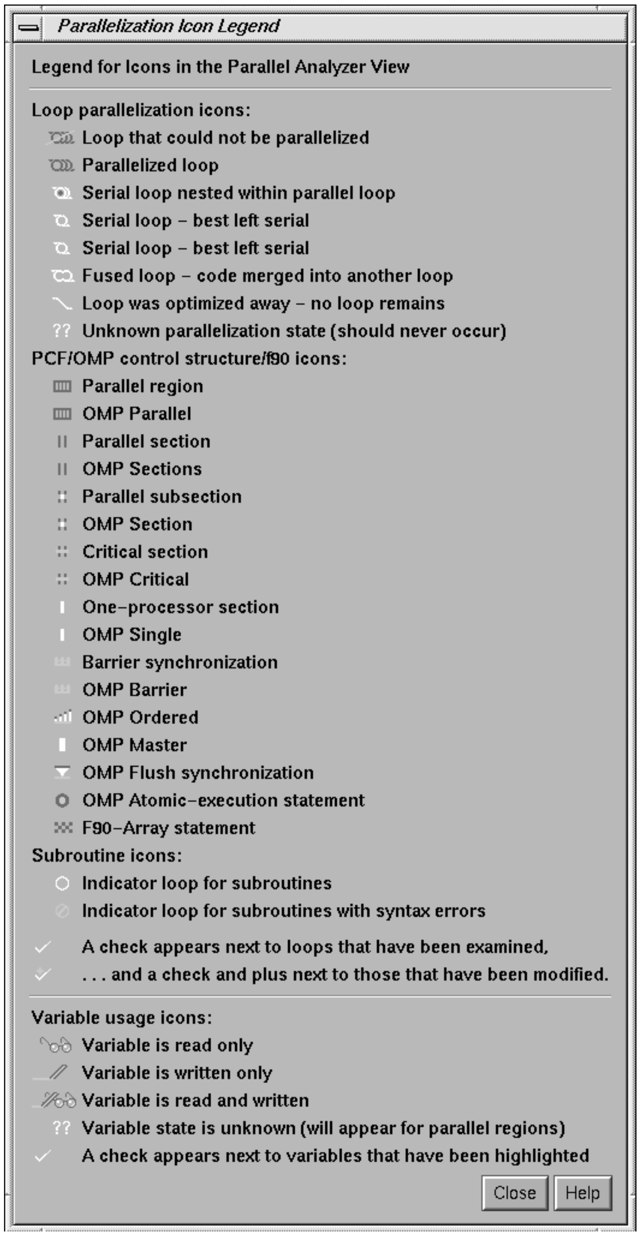


Figure 51. Output Text File Selection Dialog

Icon Legend...	Provides an explanation of the graphical icons used in several of the views. Shortcut: **`Ctrl+S`**. See Section 6.2.1.1, page 121.
Iconify	Stows all the open windows belonging to a given invocation of the Parallel Analyzer View as icons in the style of the window manager you are using.
Raise	Brings all open windows in the current session to the foreground of the screen, in front of other windows. The command also opens any previously iconified windows belonging to the invocation of the Parallel Analyzer View and brings them to the foreground. Shortcut: **`Ctrl+R`**.
Launch Tool	Opens various WorkShop tools. See Section 6.2.1.2, page 121.

Project	Controls project windows. See Section 6.2.1.3, page 122.
Exit	Quits the current session of the Parallel Analyzer View, closing all windows. If you have not updated source files and have pending requests for changes, a dialog box asks if it is OK to discard the changes. Click `OK` only if you want to **discard** any changes; otherwise, click `Cancel` to update the files.



Parallelization Icon Legend

Legend for Icons in the Parallel Analyzer View

Loop parallelization icons:

- Loop that could not be parallelized
- Parallelized loop
- Serial loop nested within parallel loop
- Serial loop – best left serial
- Serial loop – best left serial
- Fused loop – code merged into another loop
- Loop was optimized away – no loop remains
- ?? Unknown parallelization state (should never occur)

PCF/OMP control structure/f90 icons:

- Parallel region
- OMP Parallel
- Parallel section
- OMP Sections
- Parallel subsection
- OMP Section
- Critical section
- OMP Critical
- One–processor section
- OMP Single
- Barrier synchronization
- OMP Barrier
- OMP Ordered
- OMP Master
- OMP Flush synchronization
- OMP Atomic–execution statement
- F90–Array statement

Subroutine icons:

- Indicator loop for subroutines
- Indicator loop for subroutines with syntax errors

✓ A check appears next to loops that have been examined,

✓ . . . and a check and plus next to those that have been modified.

Variable usage icons:

- Variable is read only
- Variable is written only
- Variable is read and written
- ?? Variable state is unknown (will appear for parallel regions)
- ✓ A check appears next to variables that have been highlighted

Close Help

Figure 52. Parallelization Icon Legend

6.2.1.1 Icon Legend... Option

This Admin menu option opens the Parallelization Icon Legend (Figure 52, page 120) which provides the meanings of the icons that appear in various views, such as the following:

- Parallel Analyzer View, shown in Figure 1, page 9
- Transformed Loops View, shown in Figure 75, page 158
- Subroutines and Files View, shown in Figure 77, page 161
- Parallelization Control View, shown in Figure 72, page 151

6.2.1.2 Launch Tool Submenu

The Admin menu's Launch Tool submenu contains commands for launching other WorkShop tools, as well as new sessions of the Parallel Analyzer. (See Figure 53, page 121.)

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 Section 1.2, page 2.)
- Choose the Fileset > Add File menu option. (See Section 6.2.3, page 126.)

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

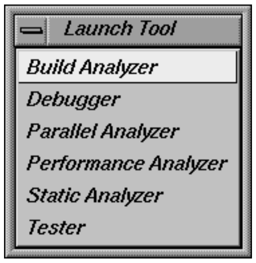


Figure 53. Launch Tool Submenu

The following six options launch applications from the Launch Tool submenu:

Build Analyzer	Launches the Build Manager, a utility that lets you compile software without leaving the WorkShop environment. For more information, see Appendix B, "Using the Build Manager," in the *Developer Magic: Debugger User's Guide*.
Debugger	Launches the WorkShop 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 the *Developer Magic: Performance Analyzer User's Guide*.
Static Analyzer	Launches the Static Analyzer, a utility that allows you to analyze and display source code written in C, C++, Fortran, or Ada. For more information, see 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 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.

6.2.1.3 Project Submenu

The Project submenu of the Admin menu contains commands that affect all the windows containing WorkShop or ProDev ProMP applications that have been launched to manipulate a single executable. The set of windows is a WorkShop

project. The Project submenu and windows that you can open from it are shown in Figure 54, page 124.

The Project submenu commands are as follows:

Iconify	Stows all the windows in the current project as icons, in the style of the window manager you are using.
Raise	Brings all open windows in the current project to the foreground of the screen, in front of other windows. The command also opens any previously iconified windows in the current project and brings them to the foreground.
Remap Paths…	Lets you modify the set of mappings used to redirect references to filenames located in your code to their actual locations in your file system. However, if you compile your code on one tree and mount it on another, you may need to remap the root prefix to access the named files.
Project View…	Launches the WorkShop Project View, a tool that helps you manage project windows.
Exit	Quits the current project, closing all windows, including those of related open applications. Thus the Source View closes, as well as, for example, the Parallel Analyzer. If you have not updated source files and have pending requests for changes, a dialog box asks if it is OK to discard the changes. Click `OK` only if

you want to discard any changes; otherwise, click `Cancel` and update the files.

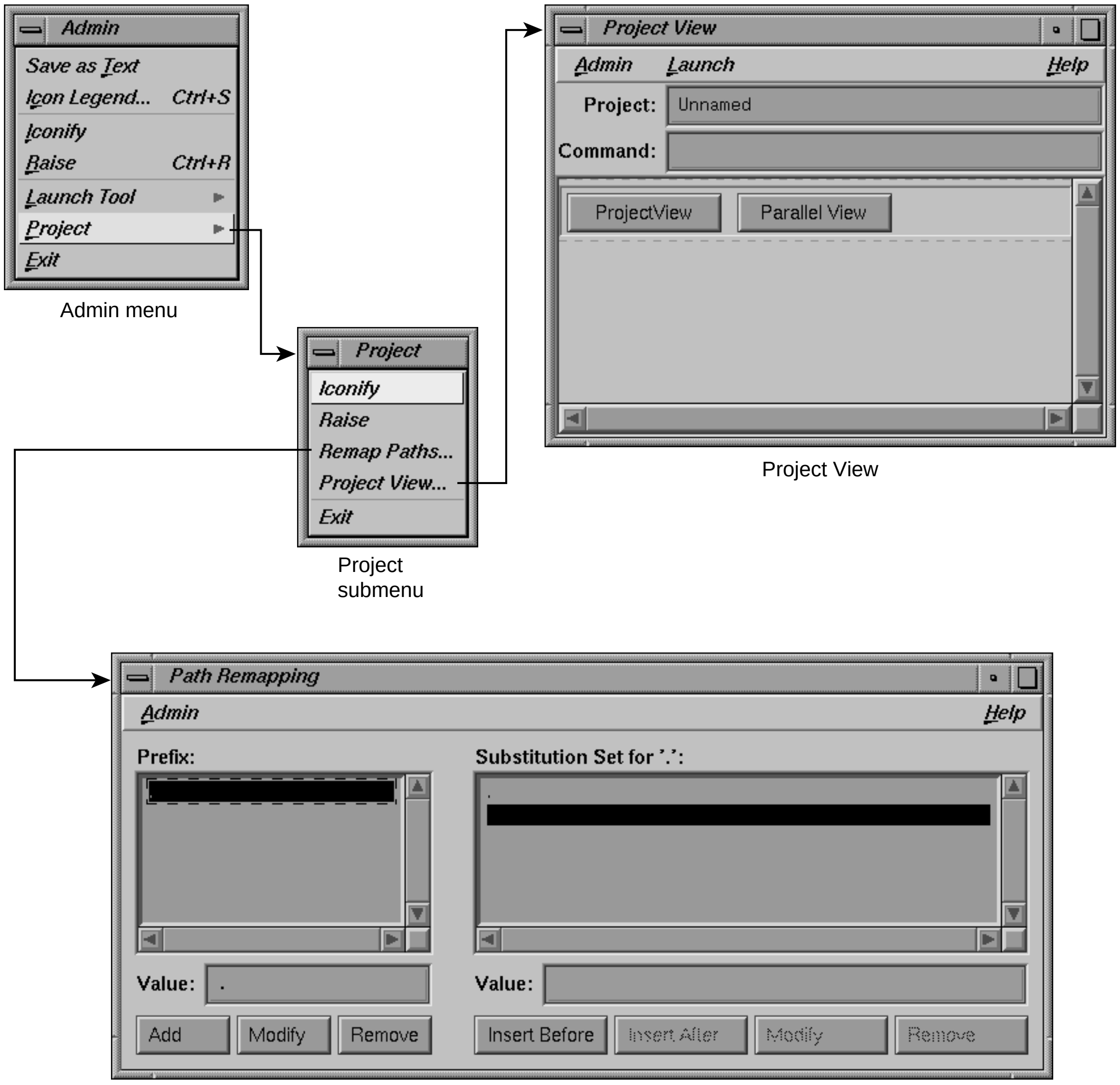


Figure 54. Project Submenu and Windows

6.2.2 Views Menu

The Views menu of the Parallel Analyzer View (Figure 55, page 125) 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.

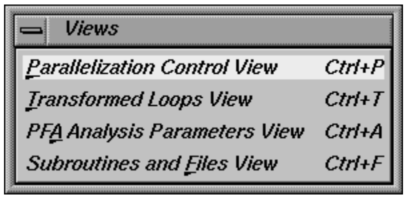

Views
Parallelization Control View Ctrl+P
Transformed Loops View Ctrl+T
PFA Analysis Parameters View Ctrl+A
Subroutines and Files View Ctrl+F

Figure 55. 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. Shortcut: **`Ctrl+P`**. For more information on this view, see Section 6.6.1, page 150.

Transformed Loops View

Opens a Transformed Loops View for the loop currently selected from the loop list display. Shortcut: **`Ctrl+T`**. For more information on this view, see Section 6.6.2, page 158.

PFA Analysis Parameters View

Opens the PFA Analysis Parameters View, which provides a means of modifying a variety of PFA parameters. Shortcut: **`Ctrl+A`**. This view is further described in Section 6.6.3, page 159.

Subroutines and Files View

Opens the Subroutines and Files View, which provides a complete list of subroutine and file names being examined within the current session of the Parallel Analyzer View.

Shortcut: **`Ctrl+F`**. This view is further described in Section 6.6.4, page 160.

6.2.3 Fileset Menu

The Fileset menu (Figure 56, page 126) 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.

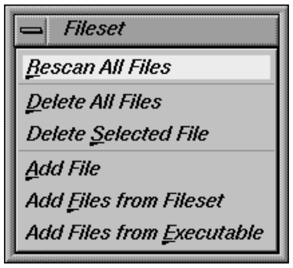


Figure 56. Fileset Menu

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 file system. The Parallel Analyzer View rereads only the files it needs to.
Delete All Files	Removes all files from the current session of the Parallel Analyzer View. You can then add new files using the Add File, Add Files from Fileset, or Add Files from Executable options, described below.
Delete Selected File	Deletes a selected file from the current session of the Parallel Analyzer View. To select a file for deletion, open the Subroutines and Files View and double-click the desired filename.
Add File	Adds a new source file to the current session of the Parallel Analyzer View. Selecting this command brings up a File Selection dialog that lets you select a Fortran source file.

	Before you can select a given source file, you must compile it to create the `.anl` file needed by the Parallel Analyzer View. (See Section 1.2.1, page 2.) If the current session is based on an executable, you cannot add files to it until you have deleted the executable's fileset. (See the Add Files from Executable option, described below.)
Add Files from Fileset	Lets you add a list of new source files to the current session of the Parallel Analyzer View. Choosing this command brings up a File Selection dialog as it does for the Add File option. If you select a file containing a fileset list, all Fortran source files in the list are loaded into the current session (other files in the list are ignored). If the current session is based on an executable, you cannot add files to it until you have deleted the executable's fileset.
Add Files from Executable	Imports all the Fortran source files listed in the symbol table of a compiled Fortran application. This command works only if there are no files in the current session of the Parallel Analyzer View when the command is selected from the menu. Selecting this command brings up a File Selection dialog as it does for the Add File option. Other WorkShop applications can also operate on files imported from an executable.

6.2.4 Update Menu

The Parallel Analyzer View Update menu (Figure 57, page 128) contains commands for placing requested changes to directives and assertions in your Fortran source code.

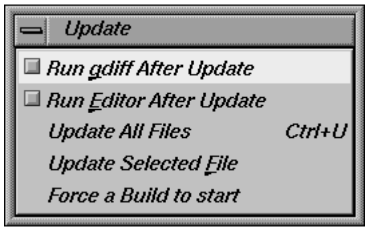


Figure 57. Update Menu

The options in the Update menu have the following effects:

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

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

```
cvpav*gDiff: True
```

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

Run Editor After Update	Sets a checkbox that opens an `xwsh` shell window with the `vi` editor on the updated source file.

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

```
cvpav*runUserEdit: True
```

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

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

In the above command, the **+%d** tells `vi` at what line to position itself in the file and is replaced

	with **1** by default (you can also omit the **+%d** parameter if you wish). The edited file's name either replaces any explicit **%s**, or if the **%s** is omitted, the filename is appended to the command.
Update All Files	Writes to the appropriate source files all changes to loops requested during the current session of the Parallel Analyzer View. Shortcut: **`Ctrl+U`**.
Update Selected File	Writes to a selected file changes to loops requested during the current session of the Parallel Analyzer View. You choose a file for updating by double-clicking in the Subroutines and Files View the line corresponding to the desired filename. (See also Section 6.6.4, page 160.)
Force a Build to start	Performs the Update All Files option and starts a build.

6.2.5 Configuration Menu

The Configuration menu (Figure 58, page 129) allows you to choose between having the Parallel Analyzer View use OpenMP or PCF directives.

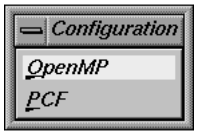


Figure 58. Configuration Menu

The options are the following:

OpenMP	Causes the Parallel Analyzer View to use OpenMP directives.

PCF	Causes the Parallel Analyzer View to use PCF directives.

6.2.6 Operations Menu

The Parallel Analyzer View Operations menu contains commands for adding assertions and directives to loops, and removing pending changes to source files (Figure 59, page 130). 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 Section 6.2.4, page 127.

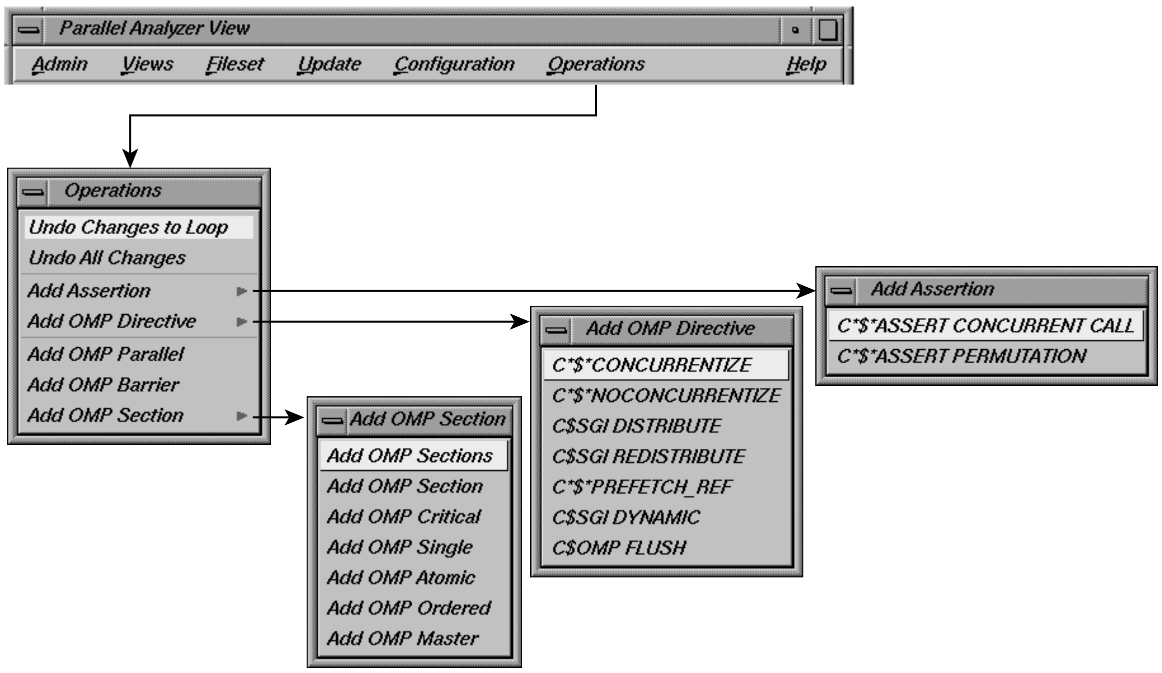


Figure 59. Operations Menu and Submenus

The Operations menu is one of two points in the Parallel Analyzer View where you can add assertions and directives. The other point is discussed in Section 6.5.2, page 144. These two menus focus on different aspects of the parallelization task:

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

The assertions and directives you can add from the Operations menu are listed in two tables. Table 3, page 131, contains a list of directives and assertions for parallelizing code that can be added with the Add Assertion and Add OMP Directive menus. Table 4, page 134, lists directives that are available from the Add OMP Section menu and are used to synchronize access to sections of code by threads.

Table 3. Add Assertion and Add OMP Directive Menu Options

Option	Effect on Compilation	For More Information
`C*$* ASSERT CONCURRENT CALL`	Ignore dependences in subroutine calls that would inhibit parallelizing.	*MIPSpro Auto-Parallelizing Option Programmer's Guide* , Chapter 3
`C*$* ASSERT PERMUTATION (`*array_name*`)`	Array *array_name* is a permutation array.	*MIPSpro Auto-Parallelizing Option Programmer's Guide* , Chapter 3
`C*$* CONCURRENTIZE`	Selectively override `C*$* NOCONCURRENTIZE`. Typically inserted during automatic parallelization.	*MIPSpro Auto-Parallelizing Option Programmer's Guide* , Chapter 3
`C*$* NOCONCURRENTIZE`	Do not parallelize file subroutine (depending on placement). Typically inserted during automatic parallelization.	*MIPSpro Auto-Parallelizing Option Programmer's Guide* , Chapter 3
`C$SGI DISTRIBUTE` `C$SGI REDISTRIBUTE`	Distribute array storage among processors. For Origin2000 systems.	*MIPSpro Fortran 77 Programmer's Guide*, Chapter 6
`C*$* PREFETCH_REF`	Load data into cache. May be used with nonconcurrent code.	*MIPSpro Compiling and Performance Tuning Guide*, Chapter 4

Option	Effect on Compilation	For More Information
`C$SGI DYNAMIC`	Allow run-time array redistribution. For Origin2000 systems.	*MIPSpro Fortran 77 Programmer's Guide*, Chapter 6
`C$OMP FLUSH`	Identifies synchronization points at which the implementation is required to provide a consistent view of memory.	*OpenMP Fortran Application Program Interface*, see http://www.openmp.org

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

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

Add Assertion — Opens the Add Assertion menu which allows you to add the following assertions, which are described in Table 3, page 131:

- `C*$*ASSERT CONCURRENT CALL`
- `C*$*ASSERT PERMUTATION`

Add OMP Directive — Opens the Add OMP Directive menu which allows you to add these directives, described in Table 3, page 131:

- `C*$* CONCURRENTIZE`
- `C*$* NOCONCURRENTIZE`
- `C$SGI DISTRIBUTE` (formerly `C*$* DISTRIBUTE`)
- `C$SGI REDISTRIBUTE` (formerly `C*$* REDISTRIBUTE`)
- `C*$* PREFETCH_REF`
- `C$SGI DYNAMIC` (formerly `C*$* DYNAMIC`)

	• `C$OMP FLUSH`
Add OMP Parallel	Allows you to add the `C$OMP PARALLEL` directive. The directive defines a parallel region, that is a block of code that is to be executed by multiple threads in parallel.
Add OMP Barrier	Allows you to add the `C$OMP BARRIER` synchronization directive. This directive causes each thread to wait at the designated point until all have reached it.
Add OMP Section	Opens the Add OMP Section submenu whose seven options allow you to add the OpenMP synchronization directives shown below. The directives are explained in Table 4, page 134. • Add OMP Sections: `C$OMP SECTIONS` • Add OMP Section: `C$OMP SECTION` • Add OMP Critical: `C$OMP CRITICAL` • Add OMP Single: `C$OMP SINGLE` • Add OMP Atomic: `C$OMP ATOMIC` • Add OMP Ordered: `C$OMP ORDERED` • Add OMP Master: `C$OMP MASTER`

To use the Add OMP Section option do the following:

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

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

Table 4, page 134, lists the directives that can be added with the Add OMP Section menu. A more detailed explanation of them can be found in the document *OpenMP Fortran Application Program Interface* located at Web site of the OpenMP Architecture Review Board, http://www.openmp.org.

Table 4. Add OMP Section Menu Options

Option	Meaning
`C$OMP SECTIONS`	Specifies that the enclosed sections of code are to be divided among threads in a team.
`C$OMP SECTION`	Delineates a section within `C$OMP SECTIONS`.
`C$OMP CRITICAL`	Restrict access to enclosed code to one thread at a time.
`C$OMP SINGLE`	Only one thread executes the enclosed code
`C$OMP ATOMIC`	Update memory location atomically, not simultaneously.
`C$OMP ORDERED`	Execute enclosed code in same order as sequential execution.
`C$OMP MASTER`	Specify code to be executed by master thread.

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.

6.2.7 Help Menu

The Help menu contains commands that allow you to access online information and documentation for the Parallel Analyzer View. (See Figure 60, page 134.)

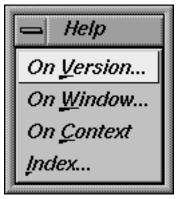


Figure 60. Help Menu

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 this option, the normal mouse cursor (an arrow) is replaced with a question mark. When you click on graphical features of the application with the left mouse, or position the cursor over the feature and press the **F1** key, the Help Viewer displays information on that context.
Index…	Invokes the Help Viewer and displays the list of available help topics, which you can browse alphabetically, hierarchically, or graphically.

6.2.8 Keyboard Shortcuts

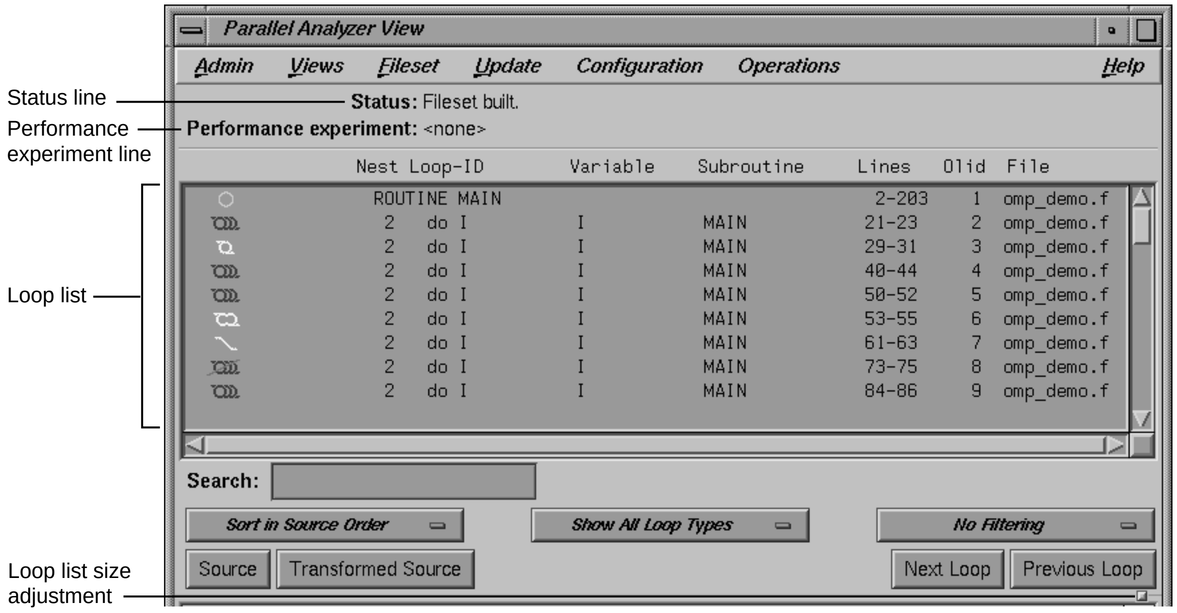
Table 5, page 135, lists the keyboard shortcuts available in the Parallel Analyzer View:

Table 5. Parallel Analyzer View Keyboard Shortcuts

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

6.3 Loop List Display

This section describes the loop list display and the various option buttons and fields that manipulate the information shown in the loop list display, shown in Figure 61, page 136.



Status: Fileset built.
Performance experiment: <none>

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

Figure 61. Loop List Display

6.3.1 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.

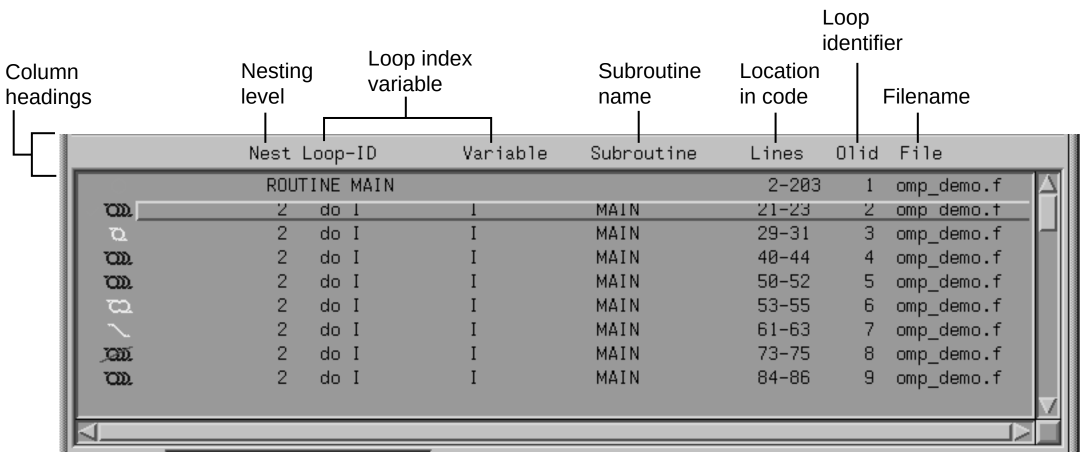
6.3.2 Status and Performance Experiment Lines

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 Section 6.2.1.2, page 121, 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>.

6.3.3 Loop List

The loop list lets you select and manipulate any Fortran DO loop contained in the source files loaded into the Parallel Analyzer View. Information about the loops is displayed in columns in the list; the headings of the columns are shown at the top of Figure 62, page 137, and described below.



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

Figure 62. Loop List with Column Headings

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

- Parallelization icon: Indicates the parallelization status of each loop. The meaning of each icon is described in the Icon Legend dialog box. (See Section 6.2.1.1, page 121.) When a loop is displayed in the loop information display (by double-clicking the loop's row), a green check mark is placed to the left of the icon to indicate that it has been examined. If any changes are made from within the loop information display, a red plus sign is placed above the check mark.
- Perf. Cost (not shown in Figure 62, page 137): The performance cost is displayed when the WorkShop Performance Analyzer is launched on the

current fileset. (See Section 6.2.1.2, page 121.) The loops can be sorted by Perf. Cost via the sort option button. (See Section 6.4.2, page 139.)

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

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

To *highlight* a loop in the list, click the left mouse anywhere in a loop's row; typing unique text from the row into the Search field does the same thing. (See Section 6.4.1, page 139.)

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 Section 6.5, page 142.) Selecting a loop affects other displays. (See Section 2.6.3, page 20.)

6.4 Loop Display Controls

The loop display controls are shown in Figure 63, page 139, and are discussed in the next sections.

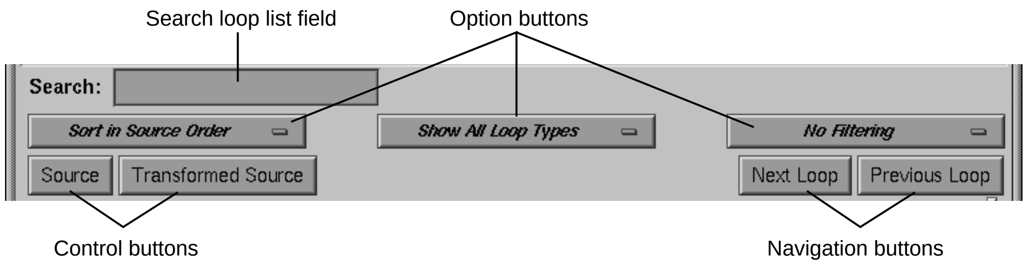


Figure 63. Loop Display Controls

6.4.1 Search Loop List Field

You can use the search loop list editable text field, shown at the top left of Figure 63, page 139, to find a specific loop in the loop list display. The Parallel Analyzer View matches any text typed into the field to the first instance of that text in the loop list, and highlights the row of the list in which the text occurs. The search field matches its text against the contents of each column in the loop list.

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

6.4.2 Sort Option Button

The sort option button is the left-most option button under the loop list search field shown in Figure 63, page 139. It controls the order in which the loops are displayed in the loop list display.

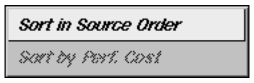


Figure 64. Sort Option Button

The choices in the sort option button (Figure 64, page 139) have the following effects:

`Sort in Source Order`	Orders the loops as they appear in the source file.
`Sort by Perf.Cost`	Orders the loops by their performance cost (from greatest to least) as calculated by the Workshop Performance Analyzer. This is the default setting. You must invoke the Performance Analyzer from the current session of the Parallel Analyzer View to make use of this option. See Section 6.2.1.2, page 121, for information on how to open the Performance Analyzer from the current session of the Parallel Analyzer View.

6.4.3 Show Loop Types Option Button

The show loop types option button is the center option button under the loop list search field shown in Figure 63, page 139. It controls what kind of loops are displayed for each file and subroutine in the loop list.

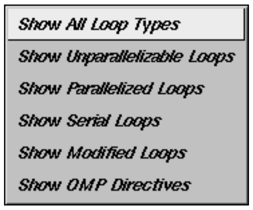


Figure 65. Show Loop Types Option Button

The options in the show loop types button (Figure 65, page 140) have the following effects:

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

6.4.4 Filtering Option Button

The filtering option button is the right-most option button under the loop list search field shown in Figure 63, page 139. It lets you display only those loops contained within a given subroutine or source file.

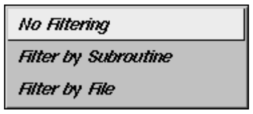


Figure 66. Filtering Option Button

The button choices have the following effects:

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

option button. Only loops contained in that file are displayed in the loop list.

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

6.4.5 Loop Display Buttons

The loop display controls (Figure 63, page 139) include two control buttons:

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

  For more information on the Source View window, see Section 6.7.1, page 162.

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

  For more information on the Transformed Source window, see Section 6.7.1, page 162.

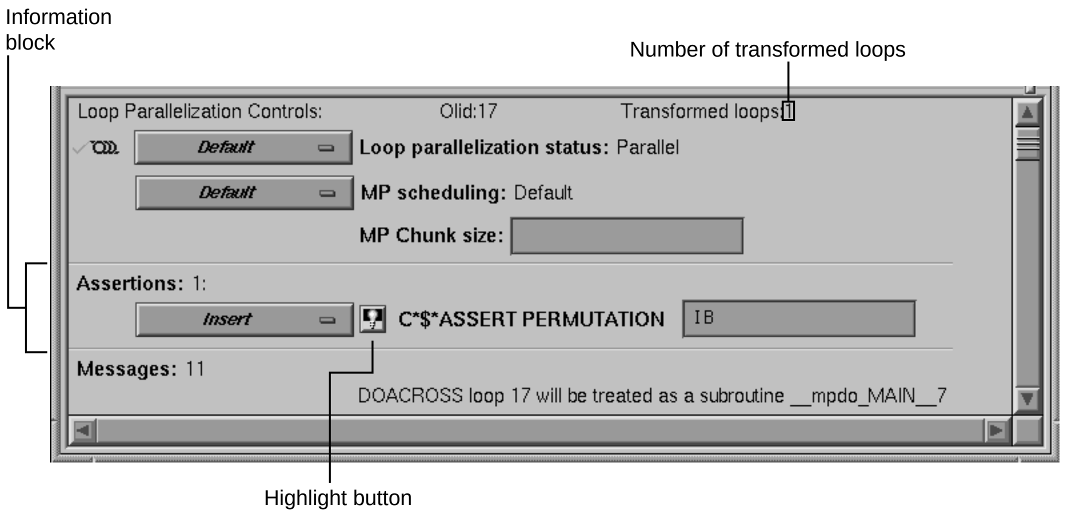
The loop display controls also include two navigation buttons:

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

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

6.5 Loop Information Display

The loop information display provides detailed information on various loop parameters, and allows you to alter those parameters to incorporate the changes into the Fortran source. The display is divided into several information blocks displayed in a scrolling list as shown in Figure 67, page 143.



Loop Parallelization Controls: Olid:17 Transformed loops:1

Loop parallelization status: Parallel

MP scheduling: Default

MP Chunk size:

Assertions: 1:

C*$*ASSERT PERMUTATION IB

Messages: 11

DOACROSS loop 17 will be treated as a subroutine __mpdo_MAIN__7

Figure 67. 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.

6.5.1 Highlight Buttons

A highlight button (light bulb, see Figure 67, page 143) 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 Section 6.7.1, page 162.)
- If a directive appears in the options menu next to it, the highlight button presents details about directive clauses in a Parallelization Control View. (See Section 6.6.1, page 150.)

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.

6.5.2 Loop Parallelization Controls in the Loop Information Display

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

Controls for altering the parallelization of the selected loop are shown in Figure 68, page 144. The controls in this section allow you to place parallelization assertions and directives in your code. Recall that you have similar controls available through the Operations menu. (See Section 6.2.6, page 130.)

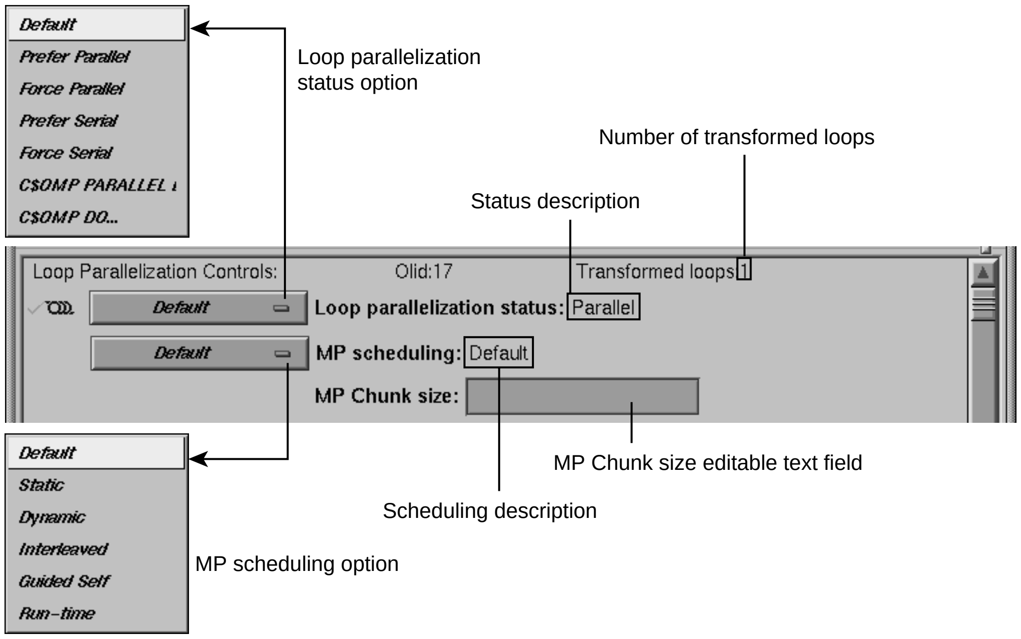


Figure 68. Loop Parallelization Controls

6.5.2.1 Loop Parallelization Status Option Button

The loop parallelization status option button (shown in Figure 68, page 144) lets you alter a loop's parallelization scheme. To the right of the option button is the Loop parallelization status field, a description of the current loop status as implemented in the transformed source. A small highlight button appears to the left of this description if the status was set by a directive.

The loop parallelization status option button choices follow below. The directives and assertions mentioned in the choices are described in Table 6, page 146.

`Default`	Always selects the parallelization scheme that the compiler picked for the selected loop.
`Prefer Parallel`	Adds the assertion `C*$*ASSERT DO PREFER` (CONCURRENT).
`Force Parallel`	Adds the assertion `C*$*ASSERT DO` (CONCURRENT).
`Prefer Serial`	Adds the assertion `C*$ASSERT DO PREFER` (SERIAL).
`Force Serial`	Adds the assertion `C*$*ASSERT DO` (SERIAL).
`C$OMP PARALLEL DO...`	Adds the OpenMP directive `C$OMP PARALLEL DO`. Selecting this item opens the Parallelization Control View. See Section 6.6.1, page 150, for more information.
`C$OMP DO...`	Launches the Parallelization Control View, which allows you to manipulate the scheduling clauses for the OpenMP `C$OMP DO` directive and to set each of the referenced variables as either region-default or last-local. A `C$OMP DO` must be within a parallel region, although the tool does not enforce this restriction. If one is added outside of a region, the compiler reports an error.

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

Table 6, page 146, lists the assertions and directives that you control from the loop parallelization status option button.

Table 6. Assertions and Directives Accessed From the Loop Parallelization Controls

Assertion or Directive	Effect on Compilation	For More Information
`C*$* ASSERT DO (CONCURRENT)`	Parallelize the loop; ignore possible data dependences.	*MIPSpro Auto-Parallelizing Option Programmer's Guide*, Chapter 3
`C*$* ASSERT DO PREFER (CONCURRENT)`	Attempt to parallelize the selected loop. If not possible, try each nested loop.	*MIPSpro Auto-Parallelizing Option Programmer's Guide*, Chapter 3
`C*$* ASSERT DO (SERIAL)`	Do not parallelize the loop.	*MIPSpro Auto-Parallelizing Option Programmer's Guide*, Chapter 3
`C*$* ASSERT DO PREFER (SERIAL)`	Do not parallelize the loop.	*MIPSpro Auto-Parallelizing Option Programmer's Guide*, Chapter 3
`C$OMP PARALLEL DO`	Parallelize the loop, ignore automatic parallelizer.	*OpenMP Fortran Application Program Interface*, see http://www.openmp.org
`C$OMP DO`	Assign each loop iteration to a different thread, ignore automatic parallelizer.	*OpenMP Fortran Application Program Interface*, see http://www.openmp.org

6.5.2.2 MP Scheduling Option Button: Directives for All Loops

The MP scheduling option button (Figure 68, page 144) lets you alter a loop's scheduling scheme by changing `C$MP_SCHEDTYPE` modes and values for `C$CHUNK`. For those modes that require a chunk size, there is a editable text field to enter the value. (See Section 6.5.2.3, page 147.)

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$OMP PARALLEL DO` directive clause. (See Section 6.6.1.3, page 156.)

The button choices are as follows:

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

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

6.5.2.3 MP Chunk Size Field

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

When you change an entry in the field, the upper right corner of the field turns down, indicating the change (Figure 69, page 148). 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.

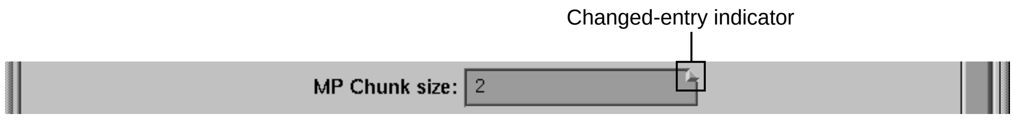


Figure 69. MP Chunk Size Field Changed

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

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

6.5.3 Obstacles to Parallelization Information Block

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

Figure 70, page 149, illustrates a message describing an obstacle. The message has a highlight button directly to its left to indicate the troublesome line(s) in the Source View window, and opens the window if necessary. If appropriate, the referenced variable or function call is highlighted in a contrasting color.

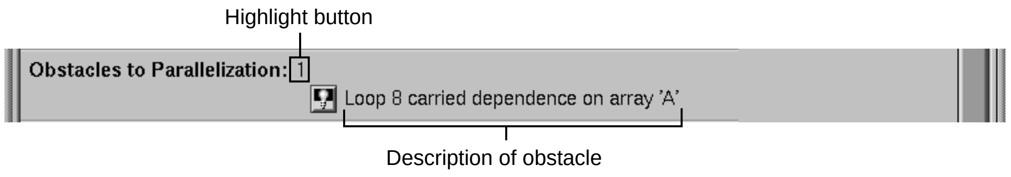

Highlight button
Obstacles to Parallelization: 1
Loop 8 carried dependence on array 'A'
Description of obstacle

Figure 70. Obstacles to Parallelization Block

6.5.4 Assertions and Directives Information Blocks

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

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

Some assertions or directives appear with an information block option button that allows you to `Keep` or `Delete it`. (If you compile `o32`, you can also `->Reverse it`.) Figure 71, page 149, shows an assertion block and its option button.

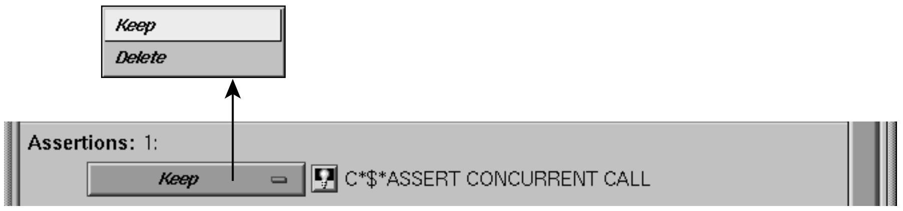


Figure 71. Assertion Information Block and Options (n32 and n64 Compilation)

Assertions and directives that govern loop parallelization or scheduling do not have associated option buttons; those functions are controlled by the loop parallelization status option button and the MP scheduling option button. (See Section 6.5.2, page 144.)

6.5.5 Compiler Messages

The Loop information display also shows any messages generated by the compiler to describe aspects of the loops created by transforming original source loops. As an example, the loop information display in Figure 67, page 143, shows there are 11 messages present although only one is shown. Some messages have associated buttons that highlight sections of the selected loop in the Source View window.

6.6 Views Menu Options

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

- Admin menu: This menu contains a single Close command that closes the corresponding view.
- Help menu: This menu provides access to the online help system. (See Section 6.2.7, page 134, for an explanation of the commands in this menu.)

6.6.1 Parallelization Control View

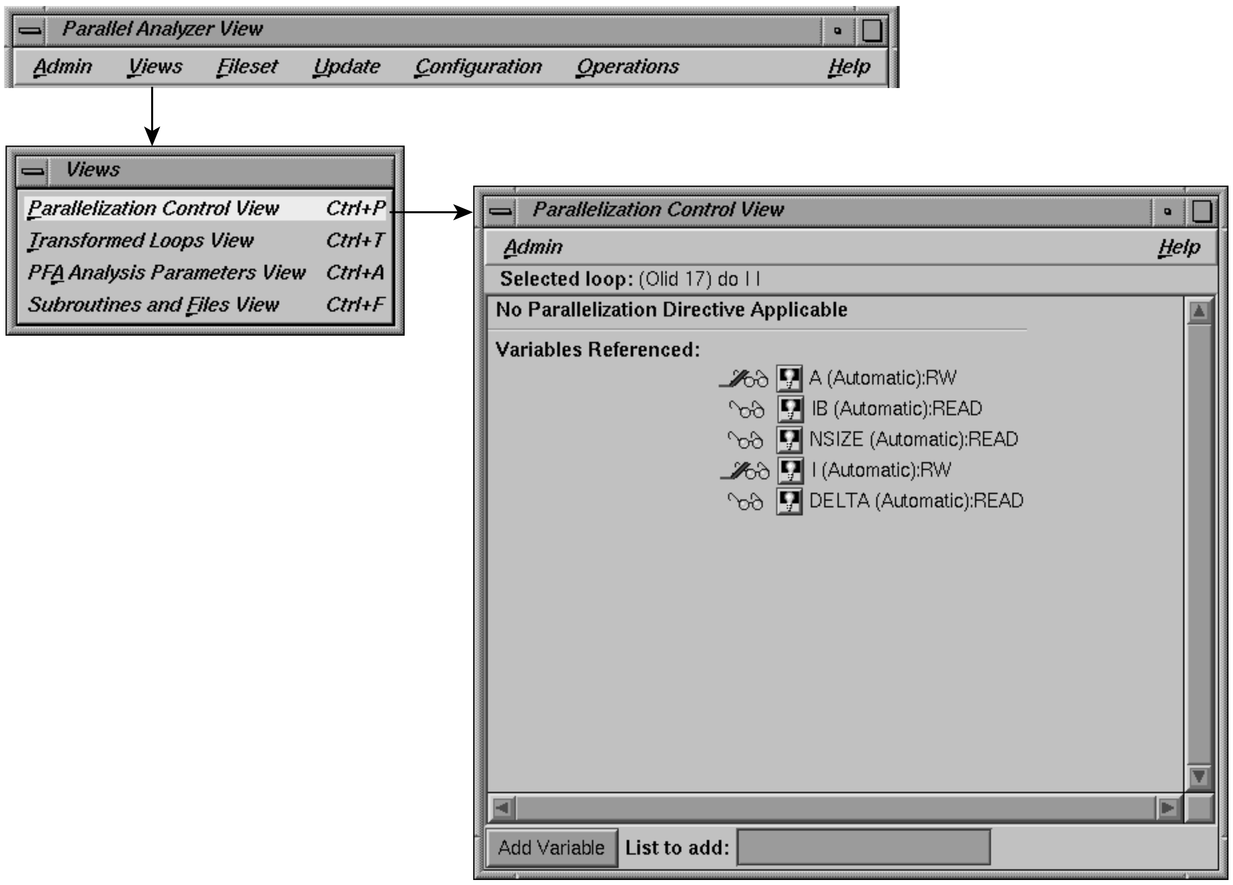
The Parallelization Control View shows parallelization controls (directives and their clauses), where applicable, and all the variables referenced in the selected loop, OpenMP construct, or subroutine. It can be opened by either of two ways.

- Selecting the Views > Parallelization Control View option. Figure 72, page 151, shows the Parallelization Control View when it is launched from the Views menu with the Default loop parallelization status option button; this is the display for loops without directives.
- Selecting `C$OMP PARALLEL DO`... or `C$OMP DO`... in the loop parallelization status option button (Figure 73, page 153, and Figure 74, page 154). This approach provides controls for clauses you can append to these directives.

Features that appear no matter which method is used to open the Parallelization Control View are discussed under Section 6.6.1.1, page 151. Features that appear only when the view is opened from the loop parallelization status option button with `C$OMP PARALLEL DO`... or `C$OMP DO`... selected are discussed in the following:

- Section 6.6.1.2, page 152

- Section 6.6.1.3, page 156
- Section 6.6.1.4, page 156


Parallel Analyzer View
Admin Views Fileset Update Configuration Operations Help
Views
Parallelization Control View Ctrl+P
Transformed Loops View Ctrl+T
PFA Analysis Parameters View Ctrl+A
Subroutines and Files View Ctrl+F
Parallelization Control View
Admin Help
Add Variable
List to add:

Selected loop: (Olid 17) do I I

No Parallelization Directive Applicable

Variables Referenced:

A (Automatic):RW
IB (Automatic):READ
NSIZE (Automatic):READ
I (Automatic):RW
DELTA (Automatic):READ

Figure 72. Parallelization Control View

6.6.1.1 Common Features of the Parallelization Control View

Independently of how you open the Parallelization Control View, these elements appear in the window (Figure 72, page 151):

- Selected loop: Contains the Olid of the loop, and the information about the loop from the Loop-ID and Variable columns of the loop list.

- Directive information section: If a directive is applicable to the loop, this section lists directive, clauses, and parameter values. (See Section 6.6.1.2, page 152.)
- Variables Referenced: The listing has two icons for each variable. They allow you to highlight the variable in the Source View and to determine the variable's read/write status; see Section 6.2.1.1, page 121, for an explanation of these icons.

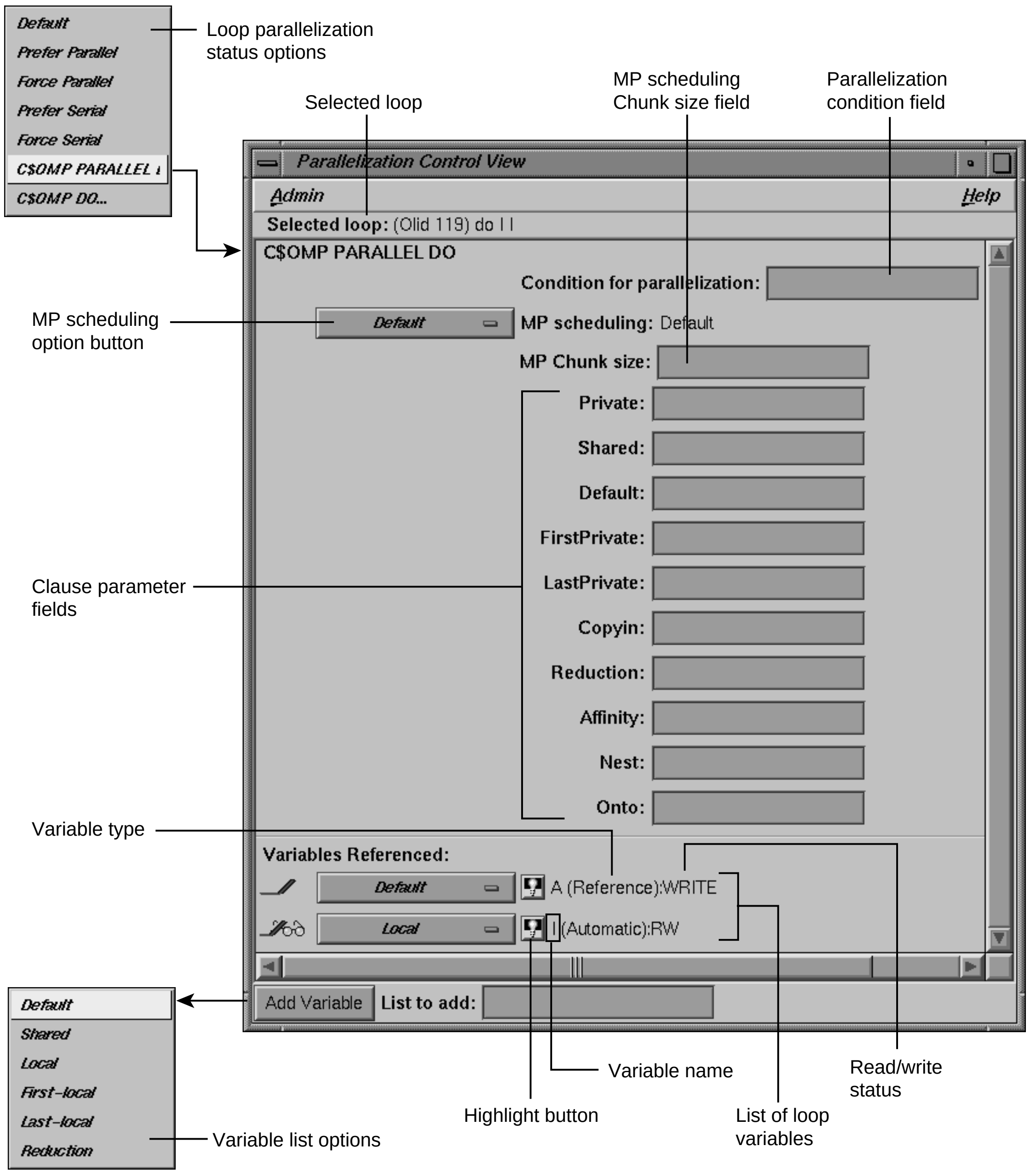
  For discussion of added option buttons that appear if the view is opened from the loop parallelization status option button when `C$OMP PARALLEL DO`... or `C$OMP DO`... is selected, see Section 6.6.1.4, page 156.
- `Add Variable`: Located at the bottom of the window frame, this button allows you to add new variables to a loop.
- List to add: Located at the bottom of the window frame, this editable text field allows you to indicate the variables you wish to add to the loop. You may enter multiple variables, with each variable name separated by a space or comma.

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

Option buttons and editable text fields in addition to those described in Section 6.6.1.1, page 151, are available if you open the Parallelization Control View from the loop parallelization status option button with either `C$OMP PARALLEL DO`... or `C$OMP DO`... selected. (See Figure 73, page 153, and Figure 74, page 154.)

There are two additional option buttons available:

- MP scheduling option button: This button allows you to alter a loop's scheduling scheme by changing the `C$MP_SCHEDTYPE` clause. See Section 6.6.1.3, page 156, for further information. This is the same button shown in Figure 68, page 144.
- Synchronization construct option button (`C$OMP DO`... only): This button allows you to set the `NOWAIT` clause at the end of the `C$OMP END DO` directive to avoid the implied `BARRIER`.



Selected loop: (Olid 119) do I I

C$OMP PARALLEL DO

Condition for parallelization:

MP scheduling: Default

MP Chunk size:

Private:

Shared:

Default:

FirstPrivate:

LastPrivate:

Copyin:

Reduction:

Affinity:

Nest:

Onto:

A (Reference):WRITE

I (Automatic):RW

Figure 73. Parallelization Control View With `C$OMP PARALLEL DO` Directive

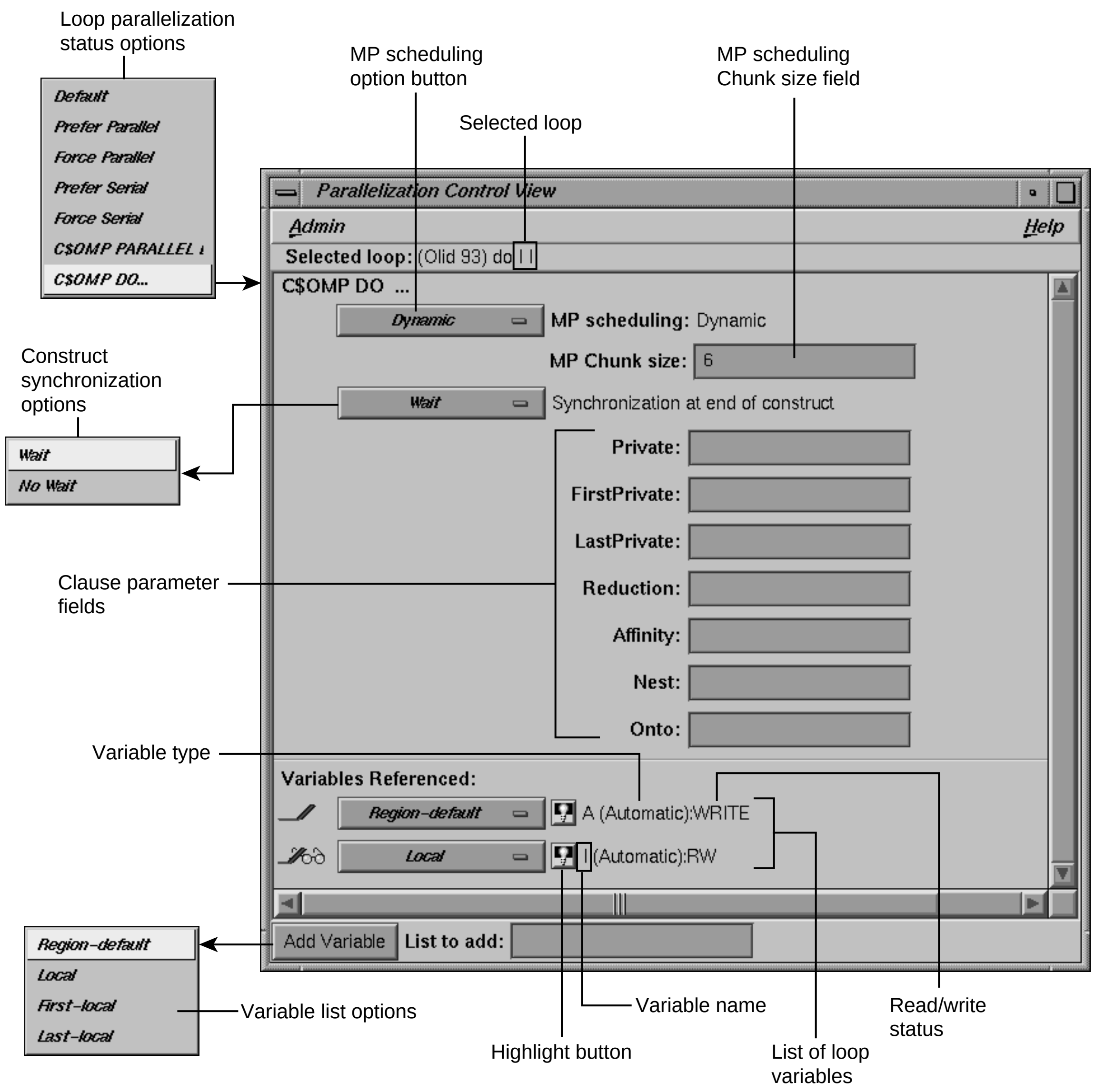


Figure 74. Parallelization Control View With `C$OMP DO` Directive

The following is a list of additional editable text fields that allow you to specify clauses for the `C$OMP PARALLEL DO` or `C$OMP DO` directives. Unless otherwise specified, the clause descriptions come from the *OpenMP Fortran Application Program Interface*.

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

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

- MP Chunk size: Allows you to set the `C$CHUNK` size for the scheduling scheme you select. For further information, see Section 6.5.2.3, page 147.
- Private: Declares the variables in a list to be `PRIVATE` to each thread in a team.
- Shared: Makes variables that appear in a list shared among all the threads in a team. All threads within a team access the same storage area for `SHARED` data. (`C$OMP PARALLEL DO`... only.)
- Default: Allows you to specify a `PRIVATE`, `SHARED`, or `NONE` scope attribute for all variables in the lexical extent of any parallel region. Variables in `THREADPRIVATE` common blocks are not affected by this clause. (`C$OMP PARALLEL DO`... only.)
- Firstprivate: Provides a superset of the functionality provided by the `PRIVATE` clause.
- Lastprivate: Provides a superset of the functionality provided by the `PRIVATE` clause.
- Copyin: Applies only to common blocks that are declared as `THREADPRIVATE`. (`C$OMP PARALLEL DO`... only.)

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

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

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

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

6.6.1.3 MP Scheduling Option Button: Clauses for One Loop

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

The options that appear have the same names as those for the MP scheduling option button in the loop information display, shown in Figure 68, page 144. However, the option button in the Parallelization Control View affects the `C$MP_SCHEDTYPE` and `C$CHUNK` clauses in the `C$OMP PARALLEL DO` directive, and so affects only the currently selected loop. Recall that the MP scheduling option button in the loop information display affects the placement of the `C$MP_SCHEDTYPE` and `C$CHUNK` directives and thus all subsequent loops.

Except for this difference in scope, the effects of both option buttons are the same; for a description, see Section 6.5.2.2, page 146. For more information, see the *MIPSpro Fortran 77 Programmer's Guide*.

6.6.1.4 Variable List Option Buttons

If the Parallelization Control View is opened from the loop parallelization status option button when either `C$OMP PARALLEL DO`... or `C$OMP DO`... is selected, each variable listed in the lower portion of the view appears with an option button. The menu allows you to append a clause to the directive, enabling you to control how the processors manage the variable. It is an addition to the highlight and read/write icons discussed in Section 6.6.1.1, page 151.

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

If the view is opened from the loop parallelization status option button when `C$OMP PARALLEL DO`... is selected, these are the variable list option button choices (Figure 73, page 153):

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

If the view is opened from the loop parallelization status option button when `C$OMP DO`... is selected, these are the variable list option button choices (Figure 74, page 154):

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

6.6.1.5 Variable List Storage Labeling

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

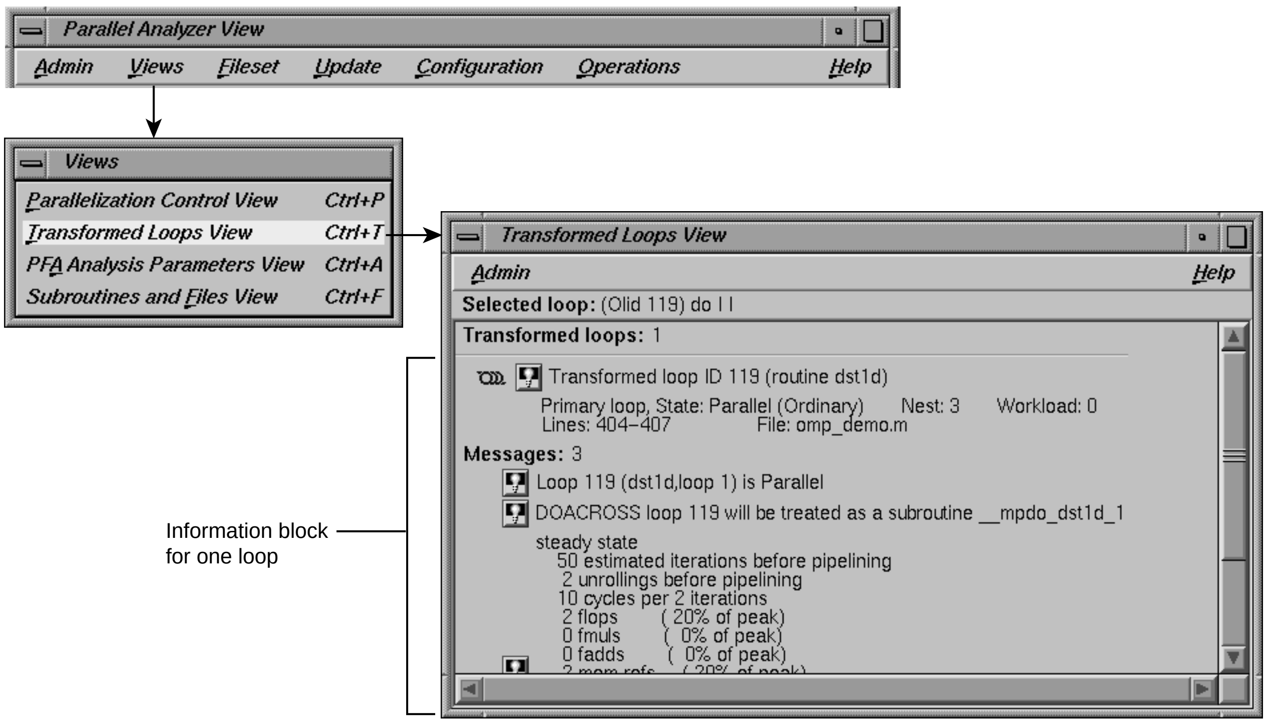
- Automatic: The variable is local to the subroutine, and is allocated on the stack.
- Common: The variable is in a common block.

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

6.6.2 Transformed Loops View

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

To open this view, choose Views > Transformed Loops View. (See Figure 75, page 158)



```
Selected loop: (Olid 119) do I I
Transformed loops: 1
Transformed loop ID 119 (routine dst1d)
    Primary loop, State: Parallel (Ordinary)    Nest: 3    Workload: 0
    Lines: 404-407              File: omp_demo.m
Messages: 3
  Loop 119 (dst1d,loop 1) is Parallel
  DOACROSS loop 119 will be treated as a subroutine __mpdo_dst1d_1
  steady state
     50 estimated iterations before pipelining
      2 unrollings before pipelining
     10 cycles per 2 iterations
      2 flops      ( 20% of peak)
      0 fmuls      (  0% of peak)
      0 fadds      (  0% of peak)
```

Figure 75. Transformed Loops View

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

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

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

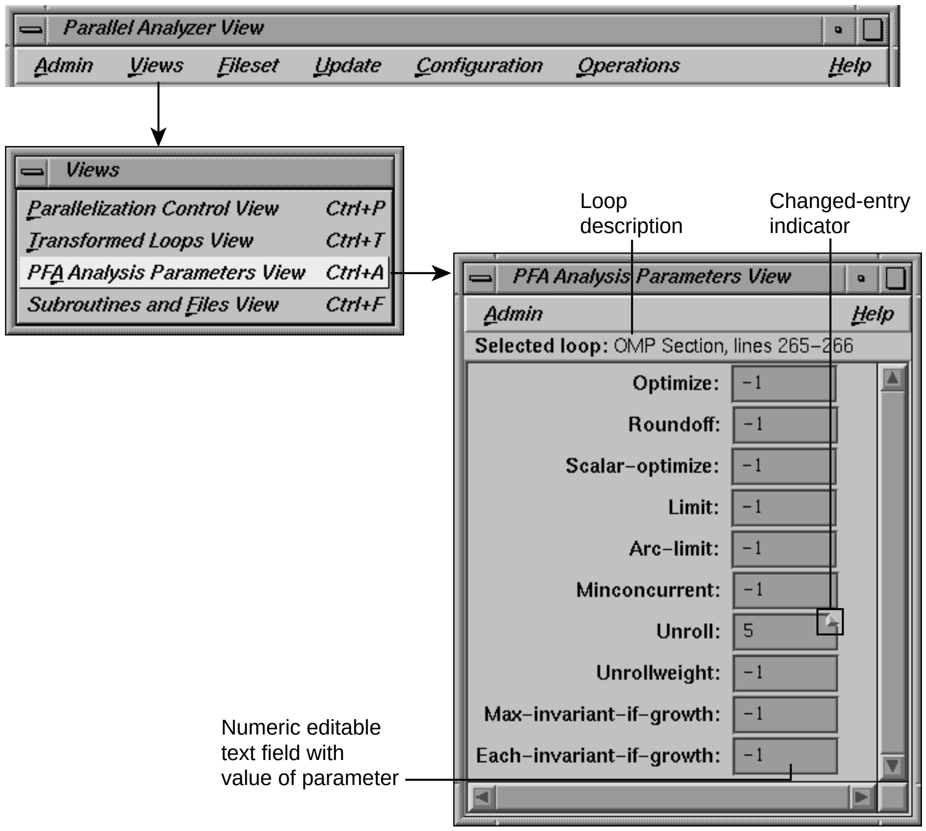
Any messages generated by the compiler are below the loop information blocks. To the left of the message lines are highlight buttons. Left-clicking them highlights in the Transformed View the part of the original source that relates to the message. Often it is the first line of the original loop that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

6.6.3 PFA Analysis Parameters View

If you compile with `o32`, you can use the PFA Analysis Parameters View, which contains a list of PFA execution parameters accompanied by fields into which you can enter new values. If you compile with `n32` or `n64`, these parameters have no effect and this view is not useful.

To open this view, choose Views > PFA Analysis Parameters View in the main window. (See Figure 75, page 158.)

When you update a source file, any PFA parameters you alter are changed for that file (Figure 76, page 160). When you change a parameter, the upper right corner of the field window turns down, as discussed in Section 6.5.2.3, page 147.


Parallel Analyzer View
Admin
Views
Fileset
Update
Configuration
Operations
Help
Views
Parallelization Control View Ctrl+P
Transformed Loops View Ctrl+T
PFA Analysis Parameters View Ctrl+A
Subroutines and Files View Ctrl+F
Loop description
Changed-entry indicator
PFA Analysis Parameters View
Admin
Help
Numeric editable text field with value of parameter

Selected loop: OMP Section, lines 265–266

Optimize:	-1
Roundoff:	-1
Scalar-optimize:	-1
Limit:	-1
Arc-limit:	-1
Minconcurrent:	-1
Unroll:	5
Unrollweight:	-1
Max-invariant-if-growth:	-1
Each-invariant-if-growth:	-1

Figure 76. PFA Analysis Parameters View

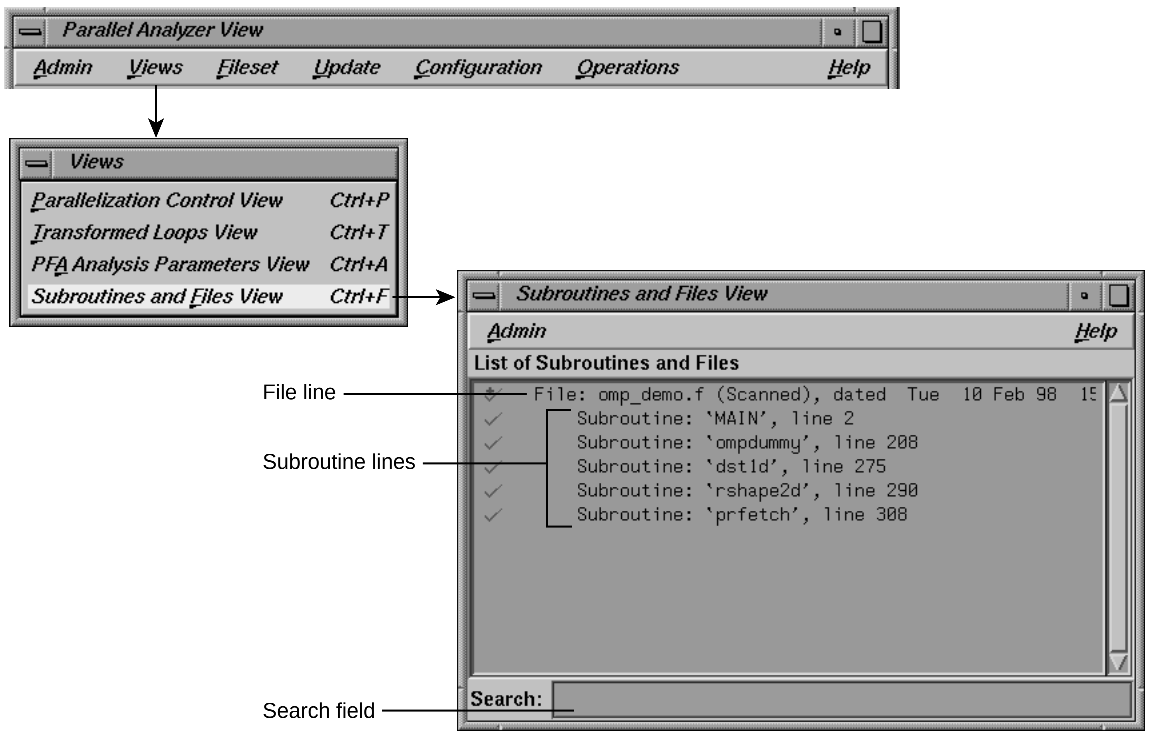
A full explanation of the PFA parameters, shown in Figure 76, page 160, can be found in Chapter 4, "Customizing PFA Execution," in the *POWER Fortran Accelerator User's Guide*.

6.6.4 Subroutines and Files View

The Subroutines and Files View contains a list from the file(s) in the current session of the Parallel Analyzer View (Figure 77, page 161). 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.


Parallel Analyzer View
Admin
Views
Fileset
Update
Configuration
Operations
Help
Views
Parallelization Control View Ctrl+P
Transformed Loops View Ctrl+T
PFA Analysis Parameters View Ctrl+A
Subroutines and Files View Ctrl+F
Subroutines and Files View
Admin
Help
List of Subroutines and Files
File line
Subroutine lines
Search field
Search:

```
File: omp_demo.f (Scanned), dated  Tue  10 Feb 98  1!
    Subroutine: 'MAIN', line 2
    Subroutine: 'ompdummy', line 208
    Subroutine: 'dst1d', line 275
    Subroutine: 'rshape2d', line 290
    Subroutine: 'prfetch', line 308
```

Figure 77. 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 Section 6.4.1, page 139.

You can select items in the list for two purposes:

- To save changes to a selected file: click the filename and use the Update > Update Selected File option at the top of the Parallel Analyzer View main window. (See Section 6.2.4, page 127.)

- To select a file or subroutine for loop list filtering, discussed in Section 6.4.4, page 141, double-click on it. The selected name appears in the filtering text field; if the item is appropriate for the selected filtering option, the loop list is rescanned.

At the bottom of the window is a Search editable text field, which you can use to search the list of files and subroutines.

6.7 Loop Display Control Button Views

These views are summoned by clicking on the `Source` and `Transformed Source loop` display control buttons.

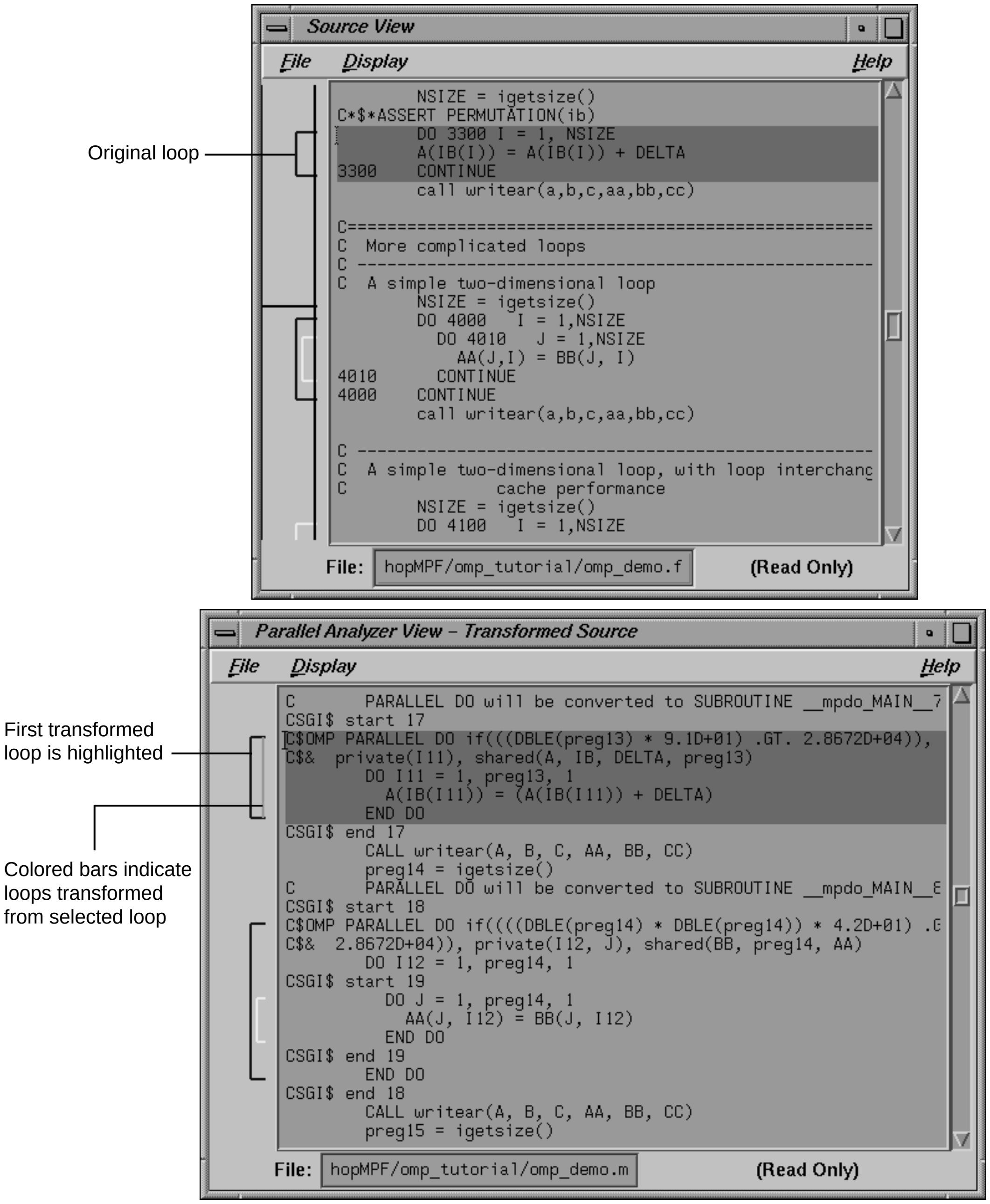
6.7.1 Source View and Parallel Analyzer View - Transformed Source

The Source View window and the Transformed Source window together present views of the source code before and after compiler optimization (Figure 78, page 163). The two windows use the WorkShop Source View interface.

Both the Source View and Transformed Source windows contain bracket annotations in the left margin that mark the location and nesting level of each loop in the source file. Clicking on a loop bracket to the left of the code chooses and highlights the corresponding loop.

In the Transformed Source window, an indicator bar (a vertical line in a different color) indicates each loop that was transformed from the selected original loop.

If the source windows are invoked from a session linked to the WorkShop Performance Analyzer (see Section 6.2.1.2, page 121), any displayed sources files known to the Performance Analyzer are annotated with performance data.


Original loop
Source View
File Display Help
First transformed loop is highlighted
Colored bars indicate loops transformed from selected loop
Parallel Analyzer View – Transformed Source
File Display Help

```
      NSIZE = igetsize()
C*$*ASSERT PERMUTATION(ib)
      DO 3300 I = 1, NSIZE
      A(IB(I)) = A(IB(I)) + DELTA
3300  CONTINUE
      call writear(a,b,c,aa,bb,cc)

C==================================================
C  More complicated loops
C --------------------------------------------------
C  A simple two-dimensional loop
      NSIZE = igetsize()
      DO 4000   I = 1,NSIZE
        DO 4010   J = 1,NSIZE
          AA(J,I) = BB(J, I)
4010    CONTINUE
4000  CONTINUE
      call writear(a,b,c,aa,bb,cc)

C --------------------------------------------------
C  A simple two-dimensional loop, with loop interchanc
C              cache performance
      NSIZE = igetsize()
      DO 4100   I = 1,NSIZE
```

File: hopMPF/omp_tutorial/omp_demo.f (Read Only)

```
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__7
CSGI$ start 17
C$OMP PARALLEL DO if(((DBLE(preg13) * 9.1D+01) .GT. 2.8672D+04)),
C$&  private(I11), shared(A, IB, DELTA, preg13)
        DO I11 = 1, preg13, 1
          A(IB(I11)) = (A(IB(I11)) + DELTA)
        END DO
CSGI$ end 17
        CALL writear(A, B, C, AA, BB, CC)
        preg14 = igetsize()
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__8
CSGI$ start 18
C$OMP PARALLEL DO if((((DBLE(preg14) * DBLE(preg14)) * 4.2D+01) .G
C$&  2.8672D+04)), private(I12, J), shared(BB, preg14, AA)
        DO I12 = 1, preg14, 1
CSGI$ start 19
          DO J = 1, preg14, 1
            AA(J, I12) = BB(J, I12)
          END DO
CSGI$ end 19
        END DO
CSGI$ end 18
        CALL writear(A, B, C, AA, BB, CC)
        preg15 = igetsize()
```

File: hopMPF/omp_tutorial/omp_demo.m (Read Only)

Figure 78. Original and Transformed Source Windows

Note: The File and Display menus shown in the Source View and Transformed Source windows are standard Source View menus, and are described in the *Developer Magic: Debugger User’s Guide*.

Examining Loops Containing PCF Directives [A]

The content of this appendix is similar to that of Section 2.11, page 56, except it uses the older PCF (Parallel Computing Forum) directives instead of OpenMP directives. For more information on PCF directives, see the *MIPSpro Fortran 77 Programmer's Guide*.

A.1 Setting Up the dummy.f Sample Session

To use this sample session, note the following:

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

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

- `dummy.f_orig`
- `pcf.f_orig`
- `reshape.f_orig`
- `dist.f_orig`

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

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

A.2 Compiling the Sample Code

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

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

This creates the following files:

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

For more information about these files, see the *MIPSpro Auto-Parallelizing Option Programmer's Guide*.

A.3 Starting the Parallel Analyzer View

Once you have created the appropriate files with the compiler, start the session by entering the following command, which opens the main window of the Parallel Analyzer View loaded with the sample file data:

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

Open the Source View window by clicking the `Source` button after the Parallel Analyzer View main window opens.

A.4 Examples Using PCF Directives

This section discusses the subroutine `pcfdummy()`, which contains four parallel regions and a single-process section that illustrate the use of PCF directives:

- Section A.4.1, page 167
- Section A.4.2, page 168
- Section A.4.3, page 170
- Section A.4.4, page 171
- Section A.4.5, page 171

To go to the first explicitly parallelized loop in `pcfdummy()`, scroll down the loop list to Olid 92.

Select this loop by double-clicking the highlighted line in the loop list.

A.4.1 Explicitly Parallelized Loops: `C$PAR PDO`

The first construct in subroutine `pcfdummy()` is a parallel region, Olid 92, containing two loops that are explicitly parallelized with `C$PAR PDO` statements. (See Figure 79, page 168.) With this construct, the second loop can start before all iterations of the first complete.

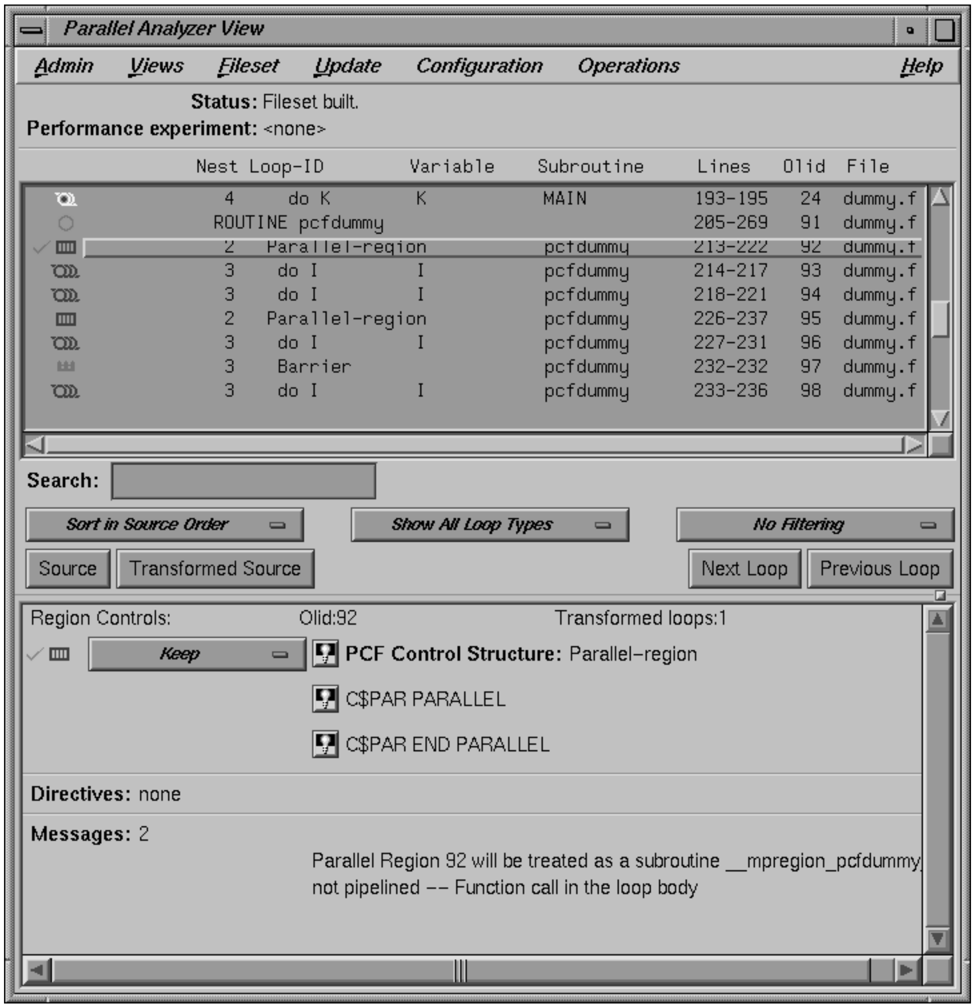
Example 26: Explicitly Parallelized Loop Using `C$PAR PDO`

```
C$PAR PARALLEL SHARED(A,B) LOCAL(I)
C$PAR PDO dynamic blocked(10-2*2)
        DO 6001 I=-100,100
           A(I) = I
6001   CONTINUE
C$PAR PDO static
        DO 6002 I=-100,100
           B(I) = 3 * A(I)
6002   CONTINUE
C$PAR END PARALLEL
```

Notice in the loop information display that the parallel region has controls for the region as a whole. The `Keep option` button and the highlight buttons function the same way they do in the Loop Parallelization Controls. (See Section 2.6.4.1, page 23.)

Click `Next Loop` twice to step through the two loops. You can see in the Source View that both loops contain a `C$PAR PDO` directive.

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



Status: Fileset built.
Performance experiment: <none>

```
Nest Loop-ID         Variable   Subroutine    Lines    Olid  File
  4    do K          K          MAIN          193-195   24   dummy.f
ROUTINE pcfdummy                              205-269   91   dummy.f
  2   Parallel-region           pcfdummy      213-222   92   dummy.f
  3    do I          I          pcfdummy      214-217   93   dummy.f
  3    do I          I          pcfdummy      218-221   94   dummy.f
  2   Parallel-region           pcfdummy      226-237   95   dummy.f
  3    do I          I          pcfdummy      227-231   96   dummy.f
  3    Barrier                  pcfdummy      232-232   97   dummy.f
  3    do I          I          pcfdummy      233-236   98   dummy.f
```

Region Controls: Olid:92 Transformed loops:1

PCF Control Structure: Parallel–region
C$PAR PARALLEL
C$PAR END PARALLEL

Directives: none

Messages: 2

Parallel Region 92 will be treated as a subroutine __mpregion_pcfdummy
not pipelined –– Function call in the loop body

Figure 79. Explicitly Parallelized Loops Using `C$PAR PDO`

A.4.2 Loops With Barriers: `C$PAR BARRIER`

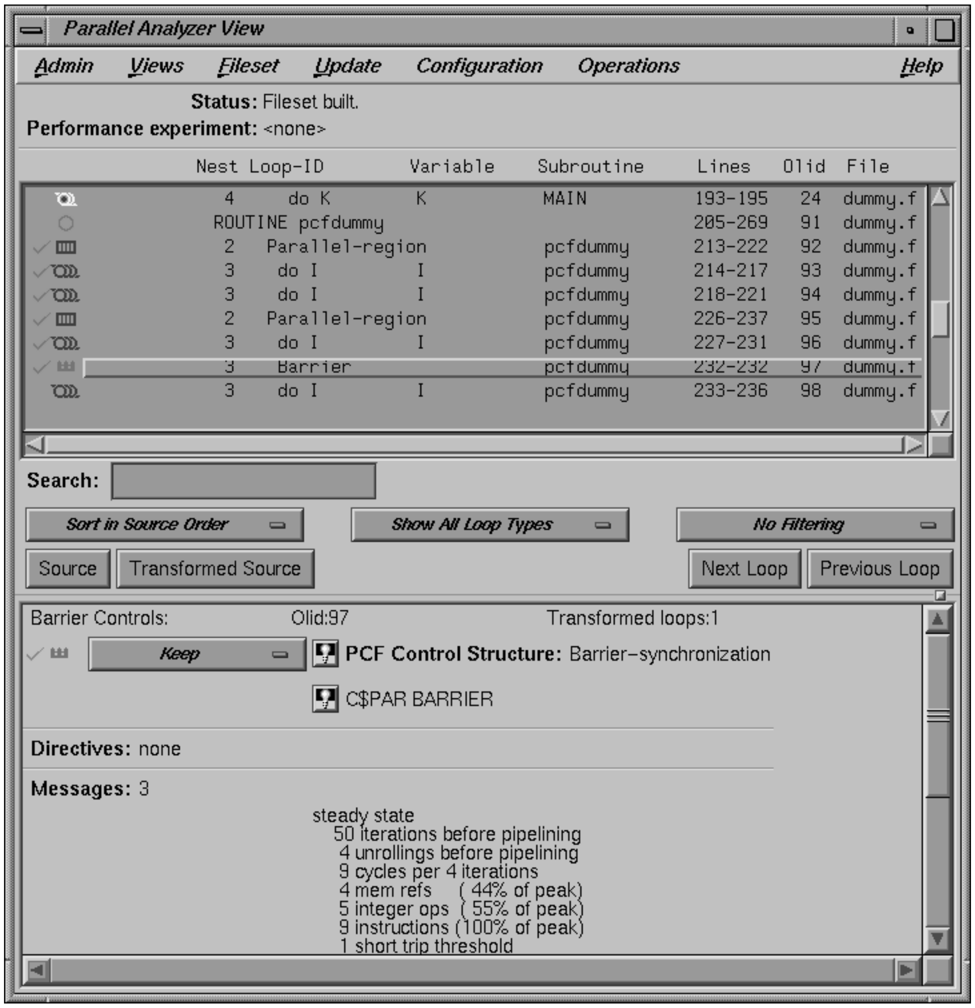
The second parallel region, Olid 95, contains a pair of loops identical to the previous example, but with a barrier between them. Because of the barrier, all iterations of the first `C$PAR PDO` must complete before any iteration of the second loop can begin.

Example 27: Loops Using `C$PAR BARRIER`

```
C$PAR PARALLEL SHARED(A,B) LOCAL(I)
C$PAR PDO interleave blocked(10-2*2)
        DO 6003 I=-100,100
           A(I) = I
6003   CONTINUE
C$PAR END PDO NOWAIT
C$PAR barrier
C$PAR PDO static
        DO 6004 I=-100,100
           B(I) = 3 * A(I)
6004   CONTINUE
C$PAR END PARALLEL
```

Click `Next Loop` twice to view the barrier region. (See Figure 80, page 170.)

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



Status: Fileset built.
Performance experiment: <none>

```
Nest Loop-ID          Variable   Subroutine     Lines    Olid  File
  4    do K           K          MAIN          193-195    24   dummy.f
ROUTINE pcfdummy                               205-269    91   dummy.f
  2   Parallel-region            pcfdummy      213-222    92   dummy.f
  3    do I           I          pcfdummy      214-217    93   dummy.f
  3    do I           I          pcfdummy      218-221    94   dummy.f
  2   Parallel-region            pcfdummy      226-237    95   dummy.f
  3    do I           I          pcfdummy      227-231    96   dummy.f
  3    Barrier                   pcfdummy      232-232    97   dummy.f
  3    do I           I          pcfdummy      233-236    98   dummy.f
```

Barrier Controls: Olid:97 Transformed loops:1

PCF Control Structure: Barrier−synchronization

C$PAR BARRIER

Directives: none

Messages: 3

```
steady state
   50 iterations before pipelining
    4 unrollings before pipelining
    9 cycles per 4 iterations
    4 mem refs      ( 44% of peak)
    5 integer ops   ( 55% of peak)
    9 instructions (100% of peak)
    1 short trip threshold
```

Figure 80. Loops Using C$PAR BARRIER Synchronization

A.4.3 Critical Sections: `C$PAR CRITICAL SECTION`

Click `Next Loop` to view the first of the two loops in the third parallel region, Olid 100. This loop contains a critical section.

Example 28: Critical Section Using `C$PAR CRITICAL SECTION`

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

Click `Next Loop` to view the critical section.

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

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

A.4.4 Single-Process Sections: `C$PAR SINGLE PROCESS`

Loop Olid 102 has a single-process section, which ensures that only one thread can execute the statement in the section. Highlighting in the Source View shows the begin and end directives.

Example 29: Single-Process Section Using `C$PAR SINGLE PROCESS`

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

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

Move to the final parallel region in `pcfdummy()` by clicking `Next Loop`.

A.4.5 Parallel Sections: `C$PAR PSECTIONS`

The fourth and final parallel region of `pcfdummy()`, Olid 104, provides an example of parallel sections. In this case, there are three parallel subsections, each of which calls a function. Each function is called exactly once, by a single thread. If there are three or more threads in the program, each function may be called from a different thread. The compiler treats this directive as a single-process directive, which guarantees correct semantics.

Example 30: Parallel Section Using `C$PAR PSECTIONS`

```
C$PAR PARALLEL shared(a,c) local(i,j)
C$PAR PSECTIONS
        call boo
C$PAR SECTION
        call bar
C$PAR SECTION
        call baz
C$PAR END PSECTIONS
C$PAR END PARALLEL
```

Click `Next Loop` to view the parallel section.

A.5 Exiting From the dummy.f Sample Session

This completes the PCF sample session.

Close the Source View window by choosing its File > Close option.

Quit the Parallel Analyzer View by choosing Admin > Exit.

To clean up the directory, enter the following in your shell window to remove all of the generated files:

```
% make clean
```

Index