Developer Magic™: WorkShop Pro MPF User's Guide

Document Number 007-2603-001

CONTRIBUTORS

Written by Robert M. Reimann, Carol Geary and Douglas B. O'Morain
Illustrated by Douglas B. O'Morain and Carol Geary
Edited by Nan Schweiger
Production by Laura Cooper
Engineering contributions by Marty Itzkowitz and Suresh Srinivas

Developer Magic™: WorkShop Pro MPF User's Guide
Document Number 007-2603-001

Contents

Figures

Introduction

Developer Magic: WorkShop Pro MPF is a companion product to the Devloper Magic: WorkShop suite of Computer-Aided Software Engineering (CASE) tools that use a graphical interface to help programmers construct, analyze, and debug software applications.

The WorkShop Pro MPF Parallel Analyzer View *cvpav* helps Fortran 77 programmers better understand the structure and parallelization of multiprocessing applications by providing an interactive, visual comparison of their original source with transformed, parallelized code. The Parallel Analyzer View reads analysis files generated by the POWER Fortran Accelerator™ (PFA) and displays editable parameters for each DO loop found in the Fortran source files. These parameters are easily customized and explored with the help of the Parallel Analyzer View's user-friendly, Motif™-based graphical interface.

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

What This Guide Contains

This guide presents the Parallel Analyzer View from a task-oriented perspective. The first two chapters are designed to get you up and running with the Parallel Analyzer View and to familiarize you with its use; the third chapter is a complete reference of the user interface. Brief descriptions of the chapters in this guide are listed below:

- Chapter 1, "Getting Started with the Parallel Analyzer View," tells you how to install the WorkShopProMPF software and run the Parallel Analyzer View on your Fortran source files.
- Chapter 2, "Analyzing Loops: 32-bit Sample Sessions," provides a tutorial session that steps you through the Parallel Analyzer's features using an illustrative piece of sample Fortran code. This chapter is only applicable for 32-bit code.
- Chapter 3, "Analyzing Loops: 64-bit Sample Sessions," provides a tutorial session that steps you through the Parallel Analyzer's features using an illustrative piece of sample Fortran code. This chapter is only applicable for 64-bit code.
- Chapter 4, "Parallel Analyzer View Reference," describes in detail the graphical user interface of the Parallel Analyzer View.

What You Should Know Before Reading This Guide

This guide assumes that you're somewhat familiar with principles of Fortran programming and multiprocessing.

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

- *WorkShop Environment Guide*
- *Debugger User's Guide*
- *Fortran 77 Programmer's Guide*
- *POWER Fortran Accelerator User's Guide*
- Fortran Reference Pages

The following book is also recommended:

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

Conventions

These are the typographical conventions used in this guide:

- **Bold**— Option flags, data types, and keywords
- *Italics*— File names, button names, Fortran variables, functions, and IRIX commands
- Regular— Menu and window names
- “Quoted”— Menu choices
- `Fixed-width`— Code examples
- **`Bold fixed-width`**— User input

Getting Started with the Parallel Analyzer View

This chapter is designed to help you get the Parallel Analyzer View up and running on your system. It contains the following sections:

- "Setting Up Your System"
- "Starting the Parallel Analyzer View"
- "Tutorials"

Setting Up Your System

The main consideration when installing the WorkShopProMPF software is memory size. At least 16MB is strongly suggested, and 32MB will improve overall performance.

WorkShopProMPF also requires installation of IRIX™ system software version 5.0 or greater, ToolTalk 1.1 or greater, and the WorkShop 2.0 or later Execution Environment. Developer Magic 1.1, WorkShop 2.0 or later, the Fortran 77 compiler, and PFA 4.0 are also required.

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

```
% versions
```

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

If you have all the software and memory you need, you are ready to install the CASEVision / WorkShopProMPF software. Consult the *IRIS Software Installation Guide* for general instructions on software installation, and the

CASEVision/WorkShopProMPF Release Notes for specific installation instructions.

After installation, you may proceed to use your WorkShopProMPF.

Starting the Parallel Analyzer View

Before starting up the Parallel Analyzer View on your Fortran source, you need to run the POWER Fortran Accelerator (PFA 4.0) on it first.

To run PFA 4.0 on a single file, enter:

% **/usr/lib/pfa** *sourcefile***.f**

As an alternative you may also enter:

% **f77 -pfa keep** *sourcefile***.f**

PFA will then generate its usual output files (see the *POWER Fortran Accelerator User's Guide* and man page for more information) and an analysis (*.*anl*) file, which the Parallel Analyzer reads to generate its views. If you use the alternative (**f77 -pfa keep** *sourcefile***.f**), you must specify the **keep** option to save the crucial *.*anl* file.

The Parallel Analyzer View *cvpav* is also installed in */usr/sbin*. To run the Parallel Analyzer View on the source file, enter:

% **cvpav -f** *sourcefile***.f**

You can also run the Parallel Analyzer View on an executable Fortran application or on a specified fileset listed within a text file:

% **cvpav -e** *executable*

% **cvpav -F** *fileset-file*

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

Note: *cvpav* assumes that PFA has been run on each of the Fortran source files named in an executable or fileset. If this is not the case, a warning message is posted, and the unprocessed files are marked within the Parallel Analyzer's Subroutines and Files View (see "C$DOACROSS Parallelization Control View") by an error icon.

Note: If you receive a message related to licensing, refer to the *NetLS System Administration Guide* or *Release Notes* for the product.

The Parallel Analyzer View has several other command line options, as well as several X resources that the user can set. See the *cvpav* man page for more information. Enter:

```
% man cvpav
```

at the shell prompt to view the *cvpav* man page.

Tutorials

For more detailed information on the Parallel Analyzer View, you can follow one of several tutorials provided with the product. This guide contains detailed descriptions of both 32- and 64-bit tutorials. See either Chapter 2, "Analyzing Loops: 32-bit Sample Sessions." or Chapter 3, "Analyzing Loops: 64-bit Sample Sessions." for a discussion of the demos provided in the */usr/demos/WorkShopMPF* directories.

PCF Directive Support

PCF directives are supported by the current 32-bit PFA processor, but only in the 64-bit compiler. If you put them into your code, they will be treated as comments, rather than properly interpreted. Chapter 3, "Analyzing Loops: 64-bit Sample Sessions" contains 64-bit PCF tutorial information.

Analyzing Loops: 32-bit Sample Sessions

This chapter provides three interactive sample sessions that demonstrate most of the Parallel Analyzer View's features for the 32-bit version of MPF. These sessions also demonstrate various aspects of parallelization and the use of the POWER Fortran Accelerator (PFA).

The sample sessions consist of a step-by-step examination of three sample programs. The samples sessions cover the following:

- The dummy sample session is designed to show the various types of FORTRAN loops, how they are transformed by PFA, and how they are displayed by the Parallel Analyzer View. The sample session begin at "Setting Up the Dummy Sample Session" on page 6.
- The linpackd sample session briefly illustrates how the Parallel Analyzer View can be used in conjunction with the WorkShop Performance Analyzer *cvperf*. The sample session begin at "Setting Up the linpackd Sample Session" on page 44.
- The f90 sample session briefly illustrates how to use MPF with Fortran-90 code. The sample session begin at "Setting Up the f90 Sample Session" on page 51.

To use these sample sessions, the subsystem *WorkShopMPF_sw.demos* must be installed.

Note: These sample sessions are applicable for the 32-bit compilers only. For a discussion of the 64-bit version of the compilers, see Chapter 3, "Analyzing Loops: 64-bit Sample Sessions."

Setting Up the Dummy Sample Session

The Parallel Analyzer View comes with a demonstration directory */usr/demos/WorkShopMPF*. It contains a subdirectory *tutorial*, which contains a source file called *dummy.f_orig* and a *Makefile*. The file contains 27 DO loops, each of which exemplifies one aspect of the parallelization process. In that directory, running *make* creates a scratch copy of the demonstration program *dummy.f* and then creates a run of PFA on the copy. PFA produces a transformed source file *dummy.m*, a listing file *dummy.l*, and an “analysis” file *dummy.anl*.

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

```
% cd /usr/demos/WorkShopMPF/tutorial
% make
```

Once the demo directory has been prepared, start the session by entering:

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

The main window of the Parallel Analyzer View opens, displaying the list of loops in the source file, *dummy.f*. Position the view at the upper left of the screen.

Note: If you receive a message related to licensing, refer to the *NetLS License System Administration Guide* or *WorkShopProMPF Release Notes*.

Figure 2-1 shows the Parallel Analyzer View with an alternative color scheme. To start a session in these colors, enter **`cvpav -scheme Potrero -f dummy.f`**. The black and white figures in the hard copy version of this guide were prepared using the Grayscale scheme. Another scheme used in this book is IndigoMagic.

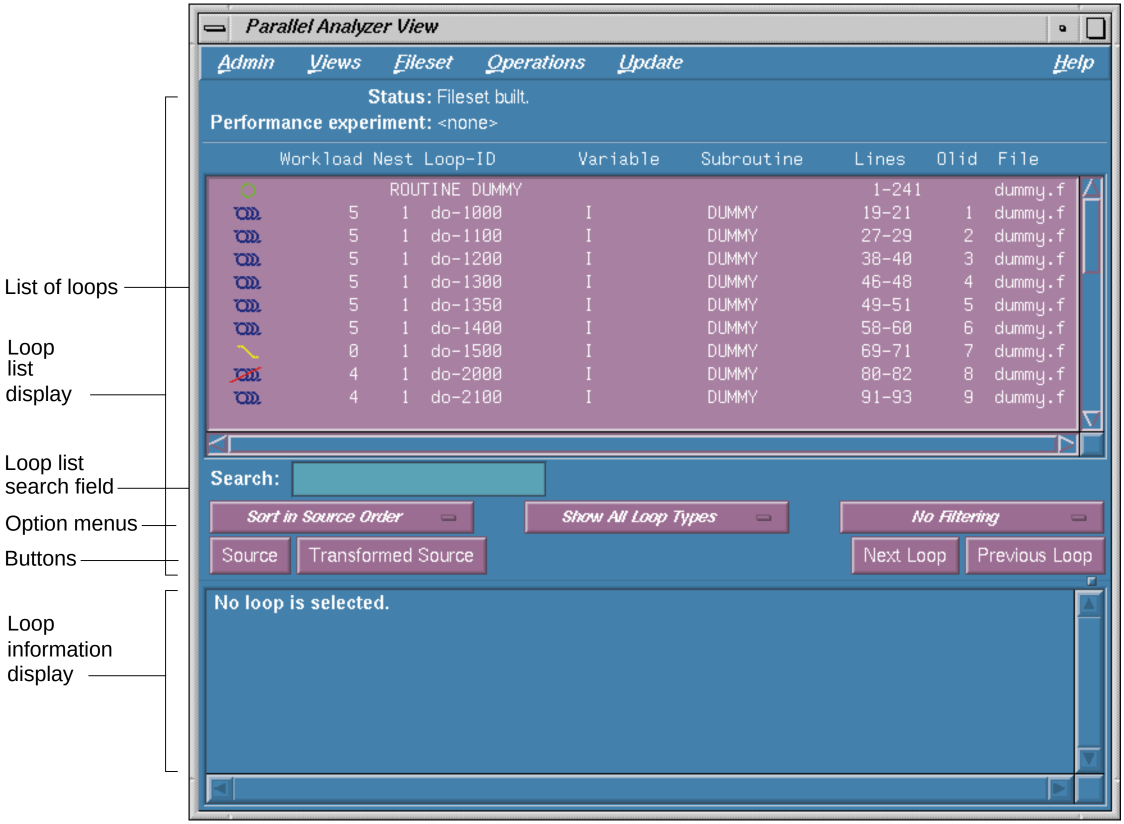


Figure 2-1 Parallel Analyzer View Main Window

Using the Loop List Display

The loop list display shows information about each loop in the program with an icon next to it that reflects the parallelization status of the loop. Pull down the Admin menu and select “Icon Legend...” to bring up a legend dialog box that explains the meaning of the various icons (see Figure 2-2). Move the legend dialog box to the side, and scroll through the list of loops to see the various icons. When you are done, close the legend dialog box by clicking the *Close* button in the lower right of the dialog box.

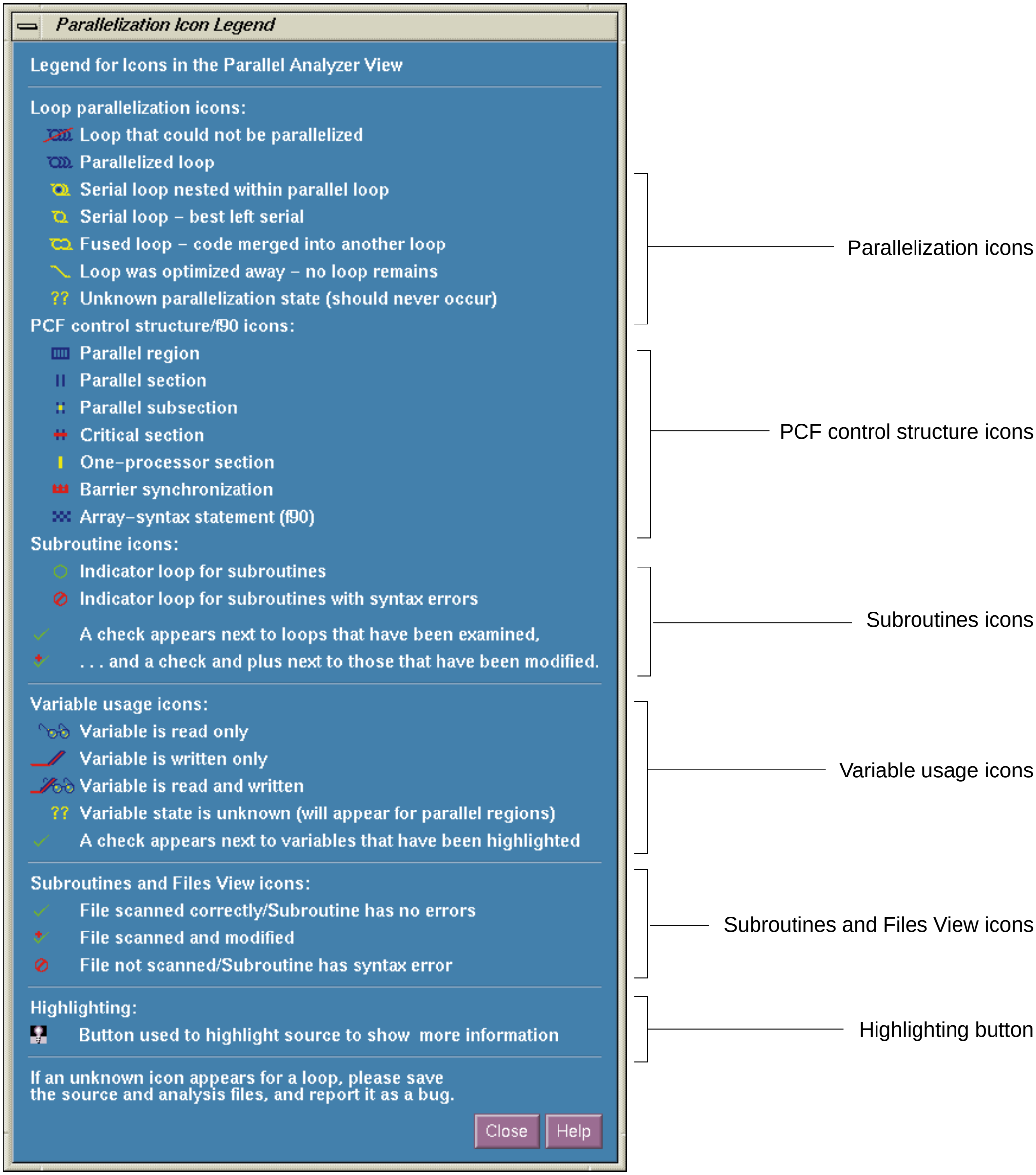


Figure 2-2 Launching the "Icon Legend..." Dialog Box

The loop list display contains the following items:

Workload a number that is supposed to reflect the amount of work done in each iteration of the loop

Nest the nesting level for the loop

Loop-ID the FORTRAN description of the loop

Variable the loop index variable

Subroutine, Lines, File
where the loop is located in the source code

Olid the original loop ID; an internal identifier for the loop (Please refer to this number when reporting bugs.)

Underneath the list display is a search field and a set of option menus and buttons that control the display of information in the loop list.

Sorting the Loop List

You can sort the list either in the order of the source code, or by loop workload, or (if you are running a performance experiment on the program using the WorkShop Performance Analyzer) by performance cost. You control sorting with the option menu to the left below the list.

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 will have to scroll down to see it (see Figure 2-3).

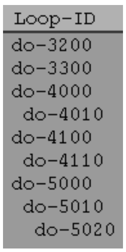


Figure 2-3 Source Order Sort

For other sorting, the list is not indented. Select “Sort by Workload” and notice the Loop-ID is no longer indented (see Figure 2-4). (The same is true of “Sort by Perf. Cost”. It is grayed out because there is no performance tool running at this time.) When you are done, select “Sort in Source Order” once again.

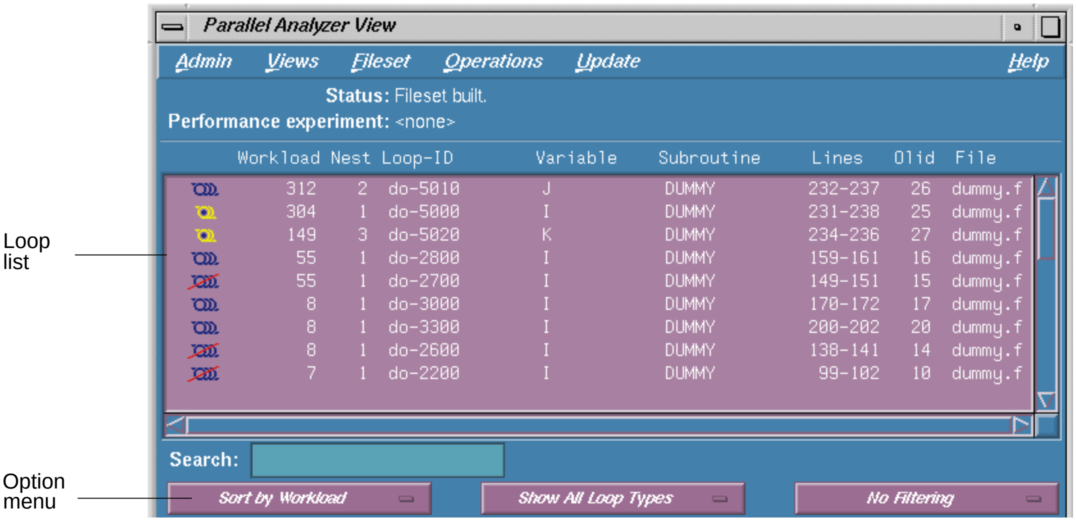


Figure 2-4 Sorting the Loop List by Workload

Filtering the Loop List

You may want to look at only some of the loops in large programs. The list can be filtered in two ways: by parallelization status or by origin of the loop.

Filtering by Parallelization Status

The parallelization status filtering is controlled by an option menu centered below the list. It initially reads "Show All Loop Types".

You can filter the list to show only those loops that cannot be parallelized, those that are parallel, or those that are serial (see Figure 2-5). Try selecting each of these, and then return to "Show All Loop Types". It can also filter to show those loops for which you have requested modifications (requesting modifications to loops is described later in this section). Since you haven't yet requested any modifications, selecting this option will result in a message saying that no loops meet the filter criterion.

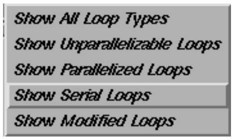


Figure 2-5 Parallelization Status Option Menu

Filtering by Loop Origin

Another way to filter is to show loops that come from a single file or a single subroutine:

1. Open the Subroutines and Files View by pulling down the Views menu and selecting “Subroutines and Files View.” Alternatively, you may use the keyboard accelerator for this operation by typing **`<Ctrl>-F`** with the cursor anywhere in the main view. A subsidiary view that lists the subroutines and files that are in the fileset opens (See Figure 2-6.)

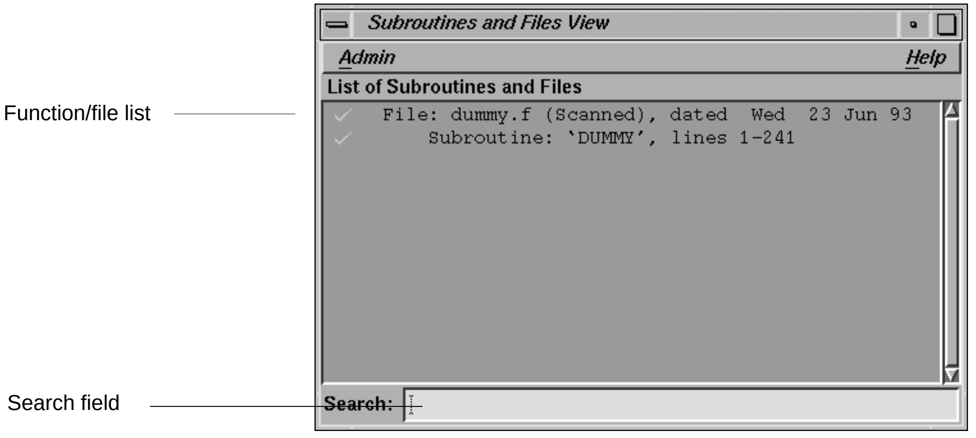


Figure 2-6 Subroutines and Files View

2. From the Filter option menu (figure 2-7), select “Filter by File.”

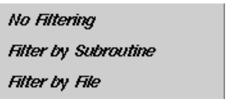


Figure 2-7 Filter Option Menu

3. Double-click the line for the file *dummy.f* in the function / file list of the Subroutines and Files View window. The name will appear in the filtering text field labeled Title: (see Figure 2-8) and the list will be rescanned. Similarly, you may try selecting “Filter by Subroutine” from the main view option menu, and double-click the line for subroutine DUMMY in the Subroutine and Files View.

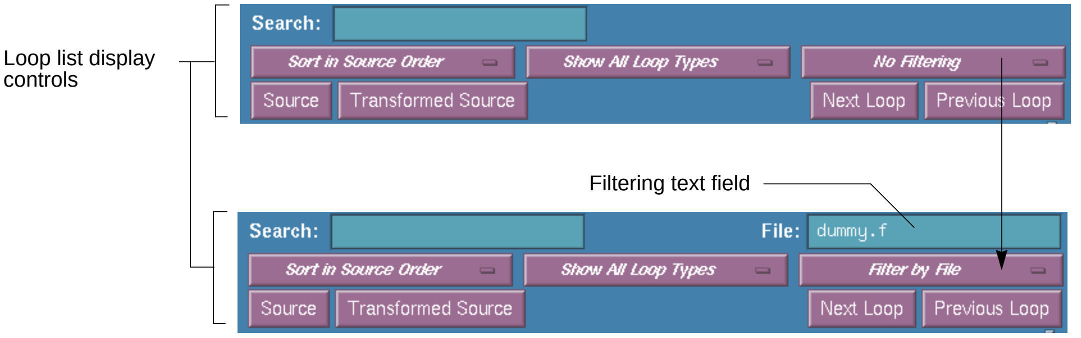


Figure 2-8 Filter by File Option Menu and Text Field

For this example, there is only one file and one subroutine, so the filtering is not very useful, but for large programs with many files and subroutines, it would be. When you are done, display all of the loops in the sample source file once again by selecting "No Filtering" from that option menu.

You won't be needing the Subroutines and Files View further, so close it by pulling down the Admin menu and selecting "Close."

Viewing Source

The Parallel Analyzer View gives you access to views of both your original Fortran source and the source as it is transformed by the POWER Fortran Accelerator.

Viewing Original Source

Click the *Source* button to the left side of the main view to bring up the Source View, as shown in Figure 2-9. This view is the same Source View that is used in the WorkShop Debugger and Performance Analyzer.

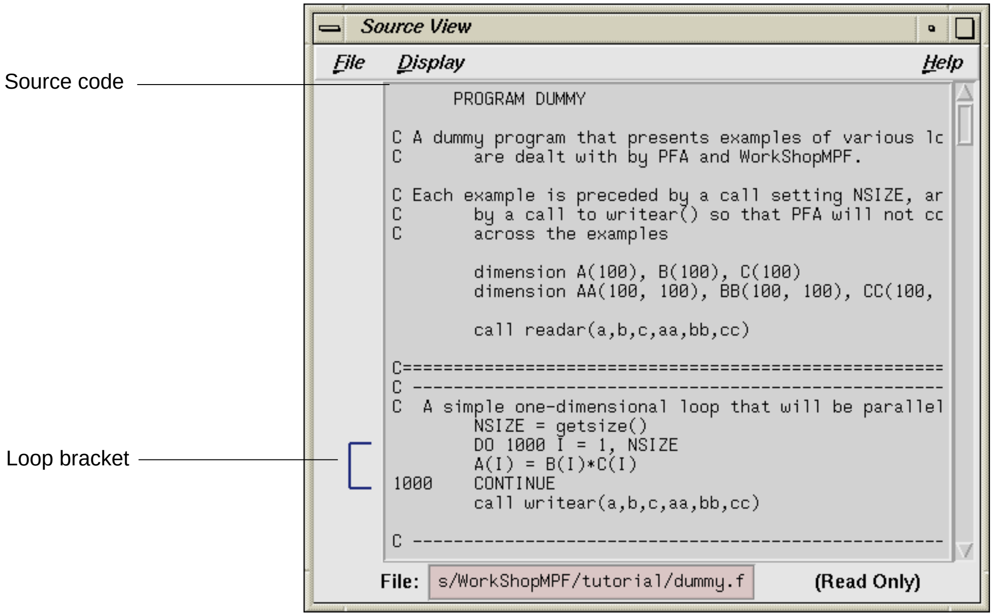


Figure 2-9 Source View

When the source display opens, position it to the right of the main view. (On machines with low-resolution screens, the windows will overlap.) Scroll up and down in the file and observe that the source window displays colored brackets that mark the location of each loop. These colors match the colors of the parallelization icons and serve to indicate the parallelization status of each loop at a glance. The color indicates which loops are parallelized, which are unparallelizable, and which are left serial.

Viewing Transformed Source

PFA is a source-to-source translator that takes the various loops in the program and transforms them both for scalar optimization and for parallelization. Each loop may be rewritten into one, two, or more transformed loops or may be combined with others or optimized away. The result of these transformations is a transformed source file that you may examine.

Click the *Transformed Source* button. Another source window labeled “Parallel Analyzer View — Transformed Source” opens as shown in Figure 2-10.

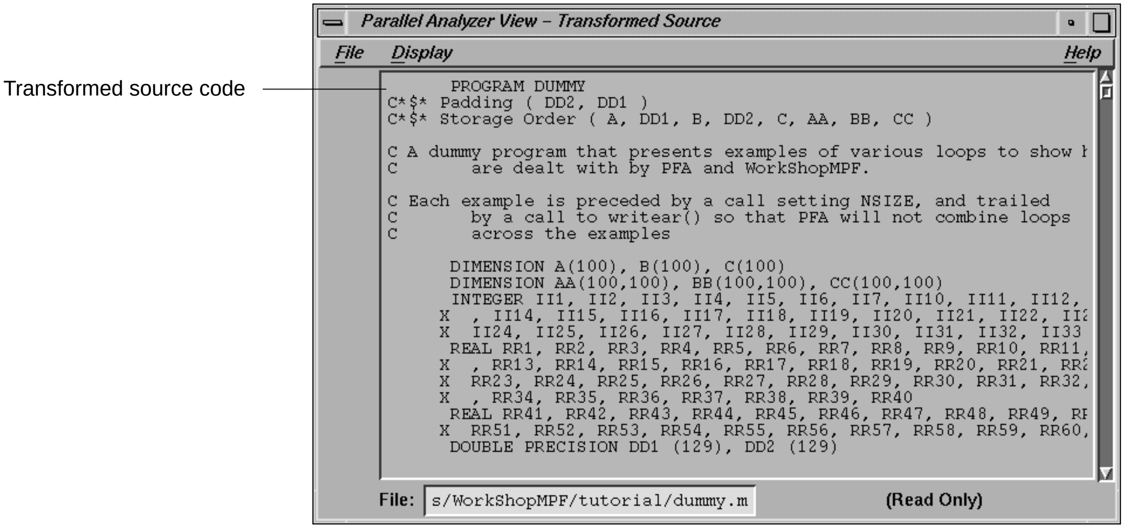


Figure 2-10 Transformed Source Window

Position it below the Source View. Scroll through it, and notice that it, too, has bracketing marking the loops. The bracketing for the transformed source cannot always distinguish between serial loops and unparallelizable loops, so some unparallelizable loops will be displayed as serial (for example, those with data dependencies).

Viewing Detailed Information about a Loop

Each line in the loop list summarizes some information about a loop. Much more information is available, and this section will show you how to examine it.

Selecting a Loop

To get more information about a loop, you must select it by

- double-clicking the loop line text (but not on its icon)
- clicking the brackets in either of the source windows
- stepping through the list with the *Next Loop* and *Previous Loop* buttons

Selecting a loop has a number of effects:

- The previously empty display below the list fills with information on the selected loop.
- The Source View scrolls to the selected loop and highlights the source code of the loop.
- The Transformed Source window highlights the first of the loops into which the original selected loop was transformed and displays a bright vertical bar next to each transformed loop that came from the original loop.

If the Transformed Loops View or the PFA Analysis Parameters View is open, it too will be switched to show the selected loop. We will look at these views later. See Figure 2-11.

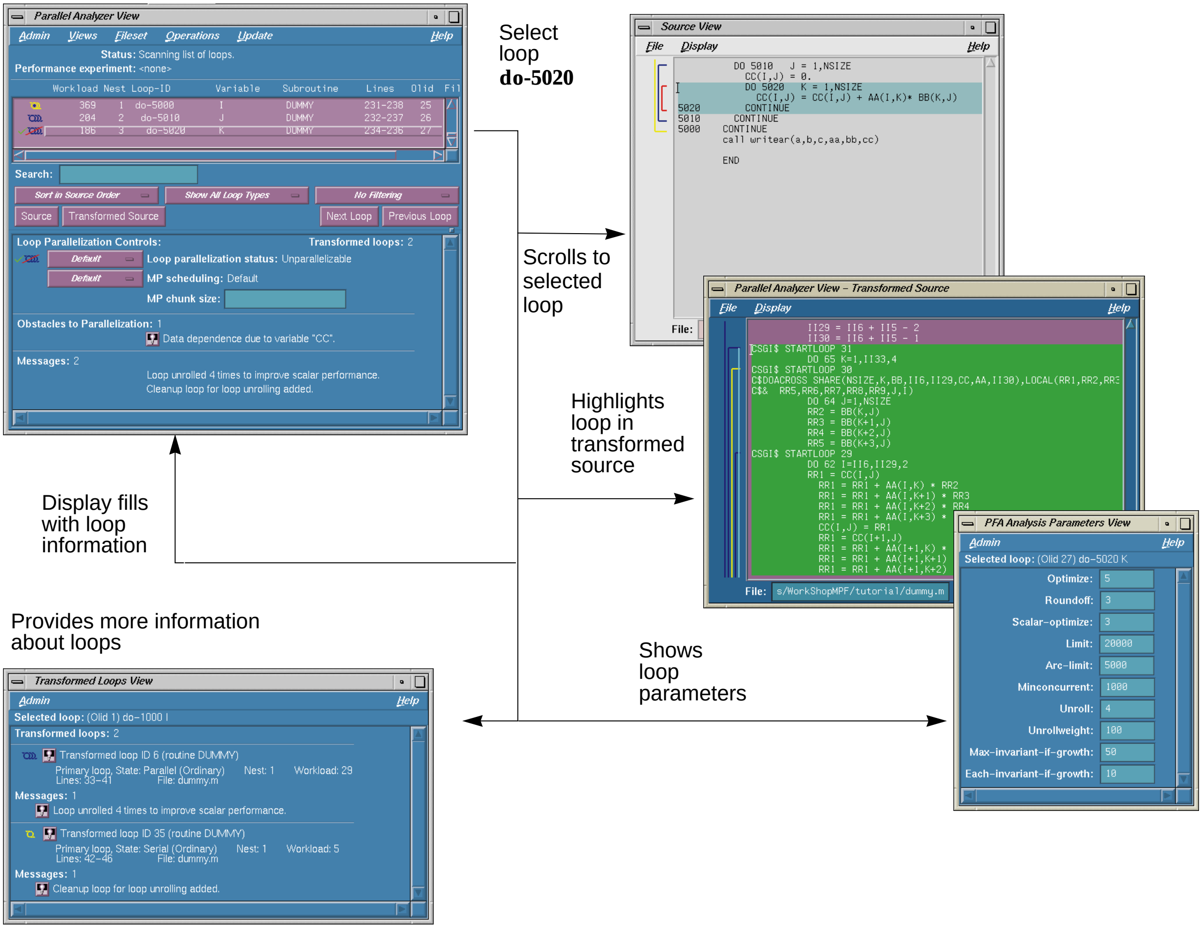


Figure 2-11 Global Effects of Selecting a Loop

In this figure and many of those following, the loop list is resized to reduce the number of loops displayed. The adjustment button is in the lower right hand corner of the loop list display, just above the loop information display. Your screen shows the full list unless you resize it.

Try scrolling through the list and double-clicking various loops, and scrolling through the source displays and clicking the loop brackets to select loops. Note that when you select each loop, its icon acquires a check mark showing that you've looked at it. When you are done, scroll to the top of the loop list in the main view and double-click the first loop's line.

Using the Loop Information Display

The loop information display occupies the lower half of the main view (see Figure 2-12). It contains detailed information about the currently selected loop. It consists of a series of lines in several blocks.

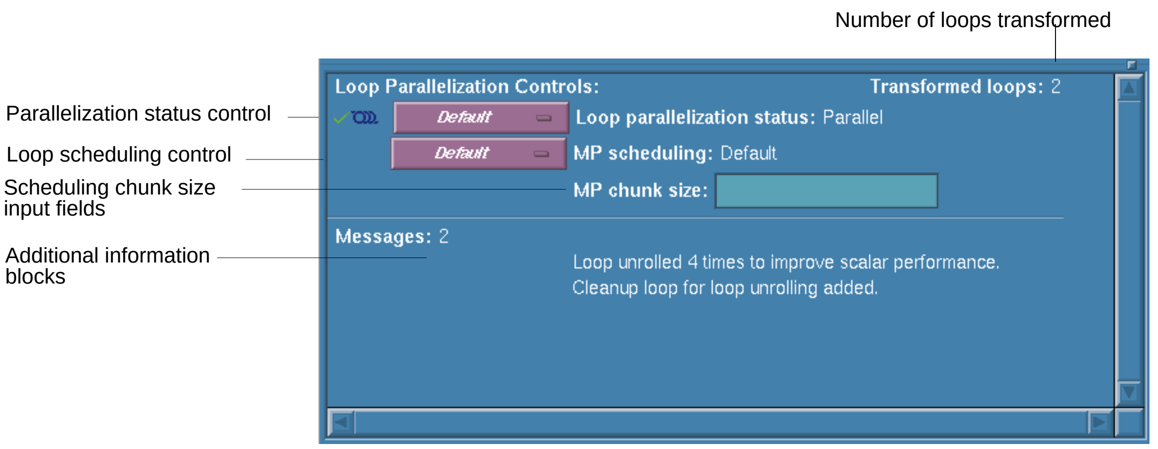


Figure 2-12 Loop Information Display

Parallelization Controls

The first line of the display is labeled Parallelization Controls:. On the far right, the first line shows how many transformed loops were derived from the selected loop. When the session is run with a performance experiment, an additional block appears above the Parallelization Controls. It gives performance information for the loop (shown in Figure 2-39). Since we do not have an experiment on this program (which does not, in fact, execute), the performance information is absent.

Below this are two option menus, the first controlling parallelization status and the second controlling the loop MP scheduling (it is shown for all loops, but is applicable to parallel loops only), and a text input field for adding an expression for the scheduling chunk size. Text labels to the right of the option menus list the current values for parallelization and scheduling.

Loop Information Messages

Below the first separator line appear up to five blocks of additional information. These are lists of:

- questions that PFA asked about the loops, if any
- obstacles to parallelization, if any
- assertions made about the loop, if any
- directives applied to the loops, if any
- messages about the loop, if any

Figure 2-13 Highlighting Button

Some of these lines may be accompanied by small “light bulb” highlighting buttons (see Figure 2-13). Each highlights a relevant part of the code in the Source View when clicked. The lines for assertions, directives, and questions also may have menus accompanying them. Lines that refer to parallelization status or PFA parameters will not have menus because they are controlled using the parallelization status menu or from the PFA Analysis Parameters View, respectively. You’ll use these features later in the session. The first loop in the file (which you selected previously) has two messages and no highlighting buttons.

Using the PFA Analysis Parameters View

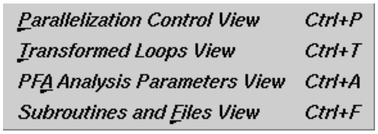


Figure 2-14 Views Menu

The PFA analysis parameters control what kinds of transformations PFA will make on the program. The values for the selected loop may be changed using the PFA Analysis Parameters View. To bring it up, pull down the Views menu and select “PFA Analysis Parameters View” (see Figure 2-14). Alternatively, you may use the keyboard accelerator for this operation by typing **`Ctrl-A`** with the cursor anywhere in the main view.

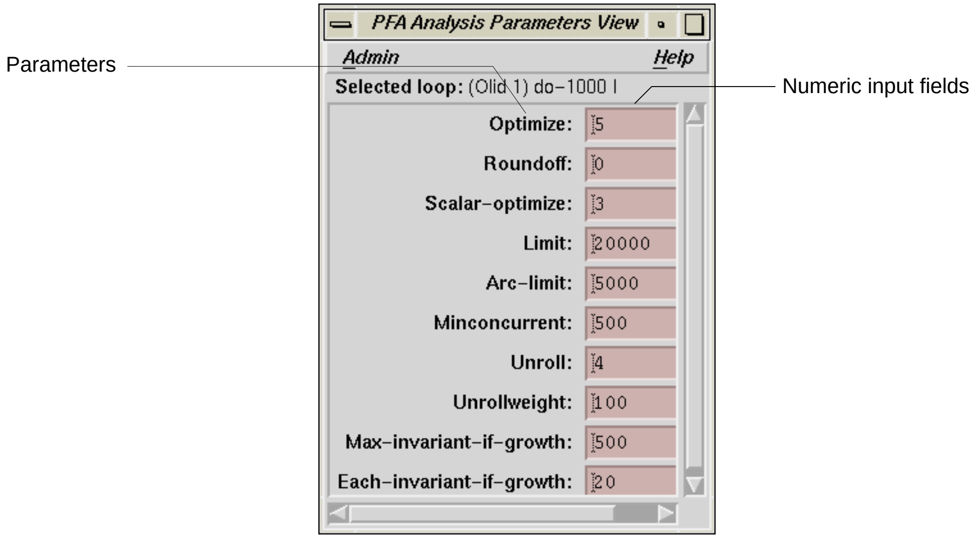


Figure 2-15 PFA Analysis Parameters View

A new view comes up, listing each of the parameters with a numeric input field to the right of each of them. Entering a new numeric value in the input field will request a change to the loop. Don't do this now; close the view by pulling down the View's Admin menu and selecting "Close."

Using the Transformed Loops View

You can also see detailed information about the transformed loops coming from a particular loop (see Figure 2-16). To do so, pull down the Views menu and select "Transformed Loops View." Alternatively, you may use the keyboard accelerator for this operation by typing **`Ctrl-T`** with the cursor anywhere in the main view.

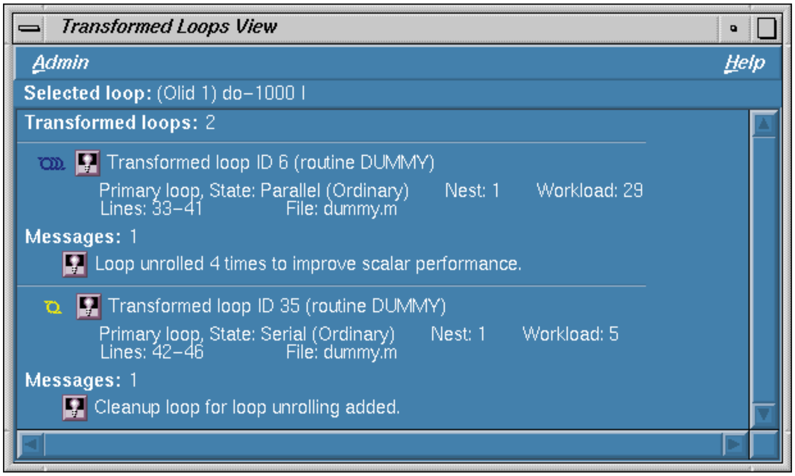


Figure 2-16 Transformed Loops View for Loop **do-1000**

When the view opens, position it at the left of the screen, below the main view. It 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, and the blocks are separated by horizontal lines.

Transformed Loop Description

The first line in each block contains a parallelization status icon, a highlighting button, and the ID of the transformed loop. (The ID is assigned by PFA.) The button, if clicked, highlights the transformed loop in the Transformed Source window and the original loop in the Source View.

The next two lines describe the transformed loop. The first provides information such as whether it is a primary loop (directly transformed from the selected original loop) or secondary loop (transformed from a different original loop but incorporating some code from the selected original loop), its parallelization state, whether it is an ordinary loop or interchanged loop, its nesting level, and workload. The second line displays the location of the loop in the transformed source.

Following the description lines is a list of messages generated by PFA, if any. To the left of the message lines are buttons, and clicking them will highlight the part of the original source that relates to the message. Often it is the first line of the original loop that is shown, since the message refers to the entire loop.

For the currently selected loop (**do-1000**), the original loop was transformed into two loops, one that runs parallelized and one that runs serial. As the messages state, the original loop was unrolled 4 times, and a cleanup loop was added. Unrolling is described in "Loop Unrolling" on page 27.

Selecting Transformed Loops

Transformed loops can also be selected. By default, the first of the transformed loops is selected when the view is brought up, and the transformed source is highlighted to show it. At the same time, the color highlighting of the original source changes, although the lines highlighted have not. See Figure 2-17. You will later see that for loops with more extensive transformations the highlighted lines will be different (for example, loops **do-1300** and **do-1350**, the fused loops).

Now click the button for the second transformed loop. The transformed source will highlight a different region (the cleanup loop), but the original source will highlight the same lines as before, as shown in Figure 2-18. This is because when a transformed loop is selected, those lines in the original source that go into the transformed loop will be highlighted. In this case, the same lines go into both the transformed loops. Transformed loops may also be selected by clicking the corresponding loop brackets in the Transformed Source window.

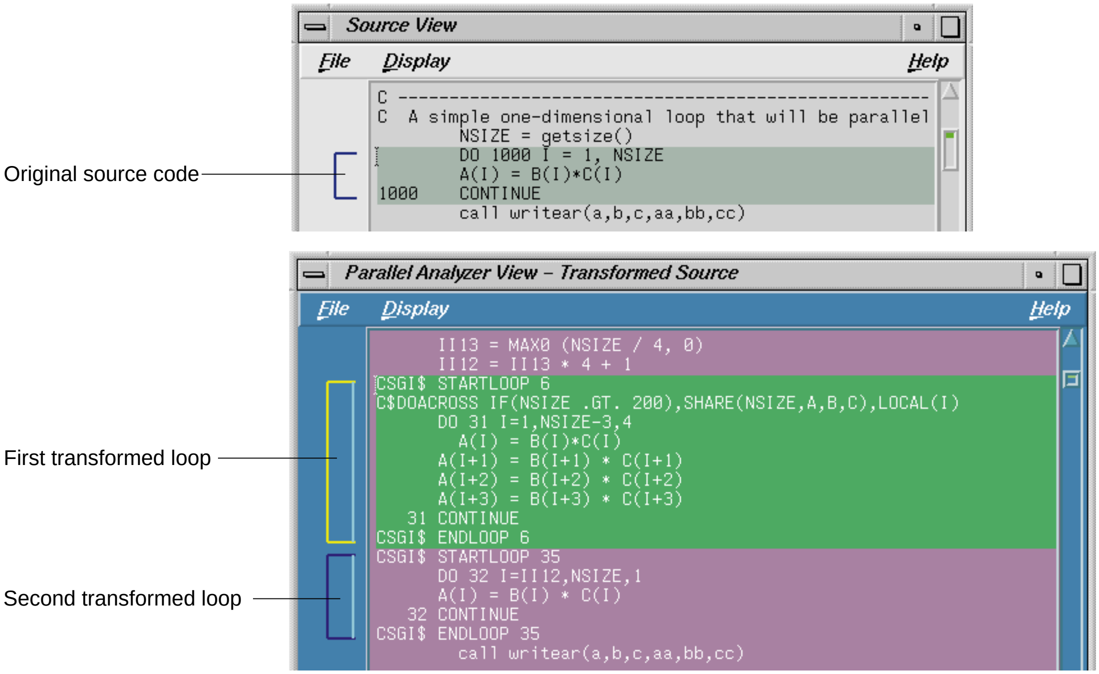


Figure 2-17 Transformed Loops in Source Windows

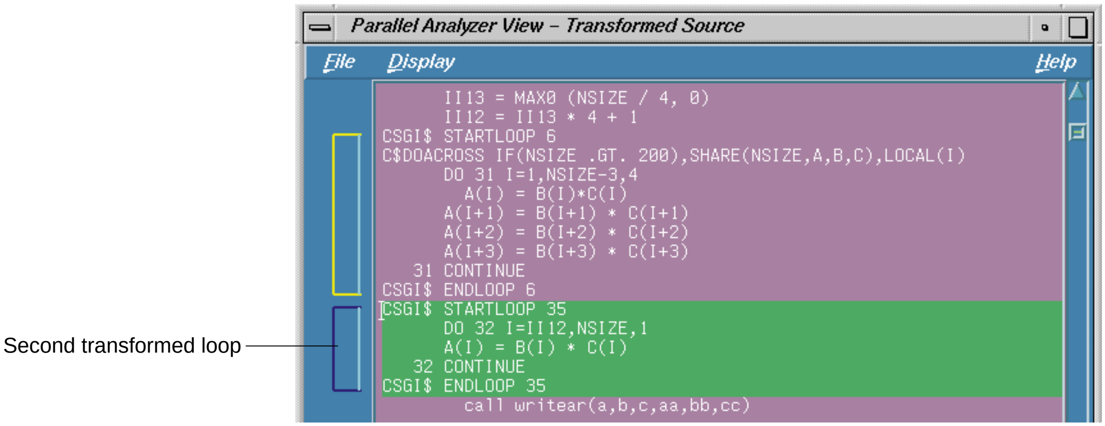


Figure 2-18 Second Transformed Loop Highlighting

You may either leave this window open or close it by selecting the “Close” command from its File menu.

Examining Loops

Now that you have familiarized yourself with the basic windows in the Parallel Analyzer View’s user interface, you can start examining and analyzing loops. First you will look at a few simple loops, next at loops with obstacles to parallelization, then at loops for which PFA asks questions, and finally at more complex, nested loops.

Simple Loops

The six loops you will examine in this section are the simplest kind of Fortran loop.

A Simple Parallelizable Loop

Scroll the list of loops back to the top and select loop **do-1000**. As the two messages state, this loop is transformed into two loops, one an unrolled, parallelized loop, and the second a clean-up loop for unrolling. (Unrolling is discussed in “Loop Unrolling”.)

Move to loop **do-1100** by clicking the *Next Loop* button.

A Preferably Serial Loop

Loop **do-1100** is preferably serial, because the amount of work done is too little to justify the parallelization overhead. Unlike the previous loop, the iteration count is known, so the total work can be computed. See Figure 2-19.

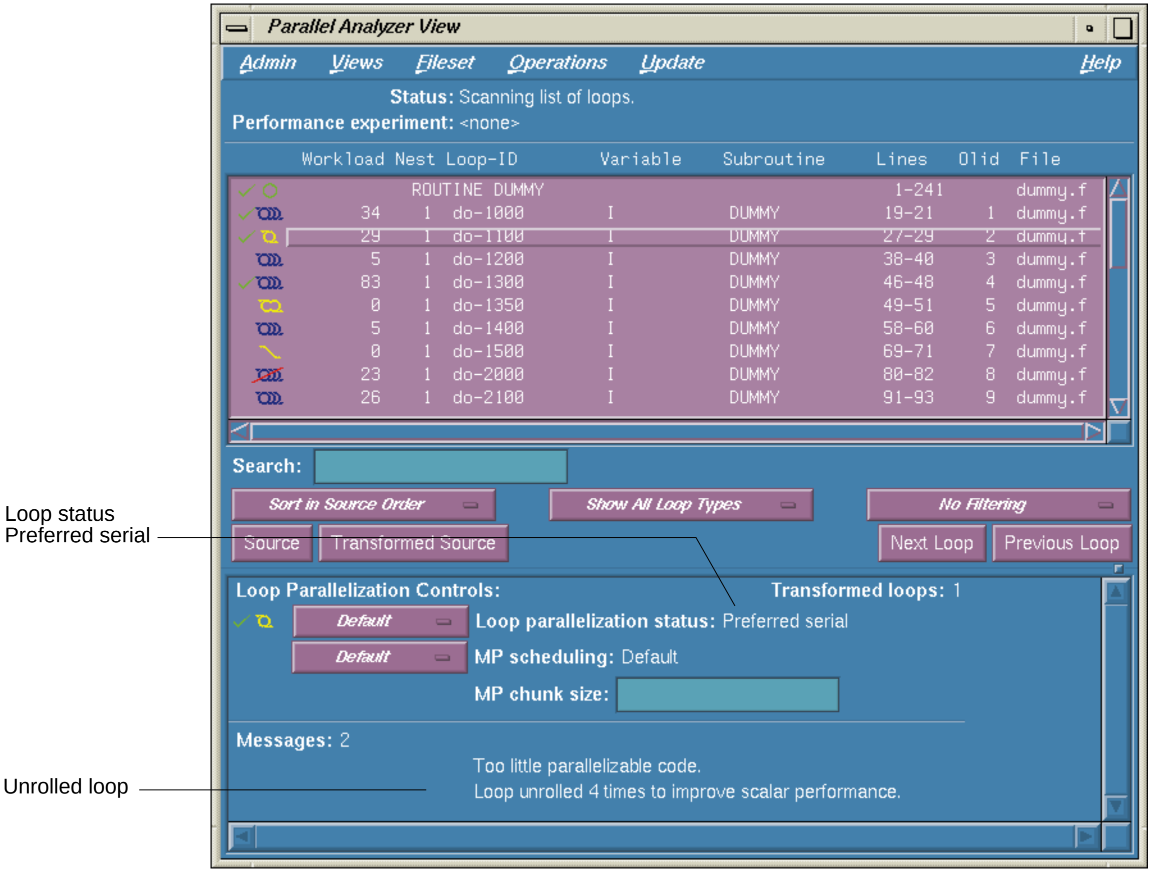


Figure 2-19 Preferably Serial Loop

Also note that this loop is unrolled as the previous one was but that no cleanup loop is needed because the count is known to be a multiple of the unrolling.

Move to loop **do-1200** by clicking the *Next Loop* button.

An Explicitly Parallelized Loop

Loop **do-1200** is parallelized because it contains an explicit C$DOACROSS directive; PFA will pass the directive through in the transformed source but does nothing further with the loop, as the messages indicate. See Figure 2-20.

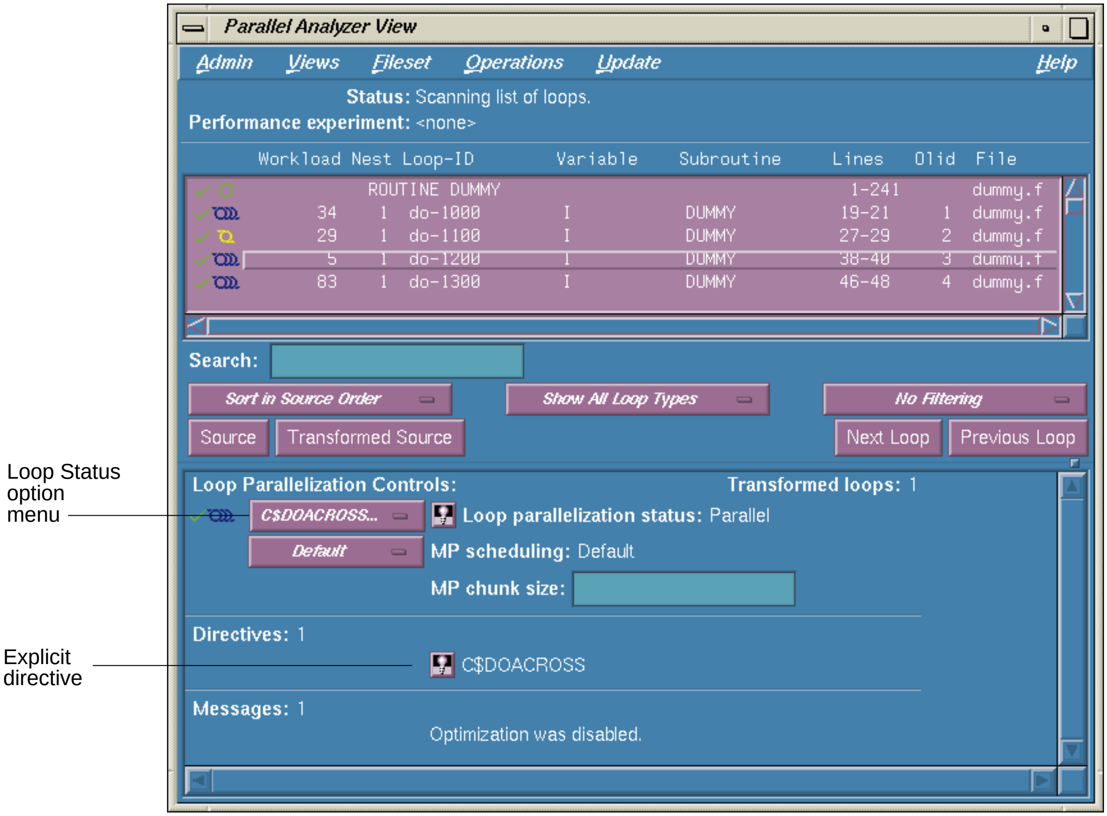


Figure 2-20 Explicitly Parallelized Loop

The loop status option menu is set to “C$DOACROSS...”and it is shown with a highlighting button. Clicking the button will bring up both the Source View and the Parallelization Control View, which shows more information about the parallelization directive. If you have clicked on the button, close the Parallelization Control View by pulling down its Admin menu and selecting “Close.” You will come back to the use of this view later. See

"Building a Custom DOACROSS Directive". Close the Source View by pulling down its File menu and selecting "Close."

The C$DOACROSS directive is displayed with a highlighting button. Click it, and the Source View comes up. Notice the highlighting of the directive in the source. See Figure 2-21.

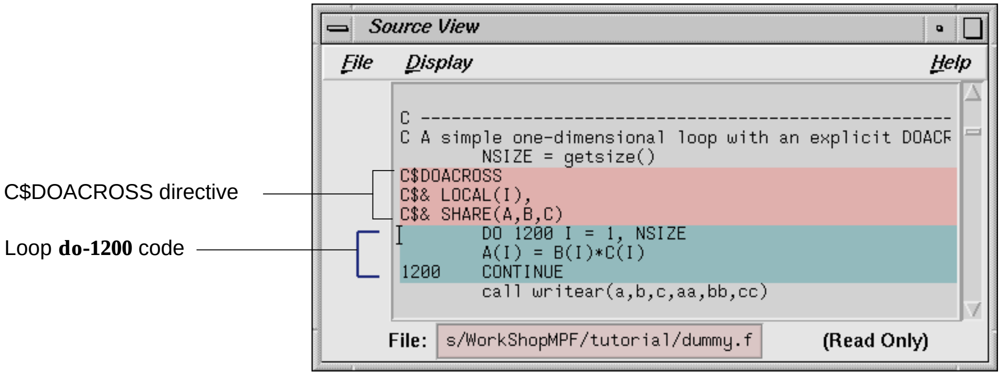


Figure 2-21 Source View of C$DOACROSS Directive

Move to loop **do-1300** by clicking the *Next Loop* button.

A Pair of Fused Loops

Loop **do-1300** is the first of two loops that can be fused. That is, the loops have the same bounds, and the code in the body of the two loops is independent, so they can be combined to save the loop overhead. Even when a loop has been fused, the Source View is highlighted to show only the selected loop, not the other loops that have been fused with it.

Notice that in the Transformed Source window, the highlighted loop has the bodies of the two original loops interleaved, and replicated for unrolling (see Figure 2-22). Click the bracket next to the loop in the transformed source. Now you see that the lines highlighted in the original source come from both loops. Then click the bracket for the loop below it in the transformed source (the cleanup loop for unrolling) and see that it, too, highlights source from both loops.

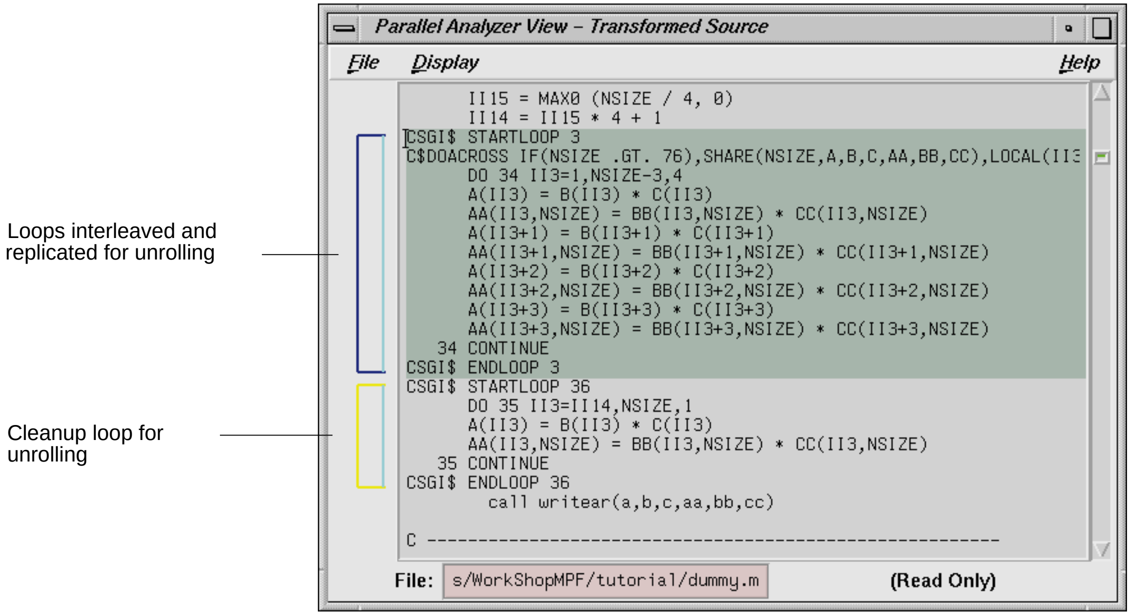


Figure 2-22 Fused Loops in Transformed Source Window

Move to loop **do-1350** by clicking the *Next Loop* button. Loop **do-1350** is the other half of the fused pair. Its icon indicates that it was fused, and the highlighting in the transformed source indicates that it was transformed into the same pair of loops as the previous one.

Move to loop **do-1400** by clicking the *Next Loop* button.

Loop Unrolling

Unrolling is done to reduce the loop overhead relative to the real work of the loop. The simpler the body of the loop, the more profitable unrolling can be. In many cases, the loop iteration count is not known, so an additional loop, called a cleanup loop, is necessary to handle the last few iterations. Sometimes, the iteration count is known but is not a multiple of the unrolling; in such cases, PFA will usually explicitly add code for the last few iterations.

Loop **do-1400** is the same as the first loop in the program, but a directive "SCALAR OPTIMIZE(1)" has been added. The loop is not unrolled. By default, the scalar optimization parameter is set to 3, which allows loop unrolling.

Move to loop **do-1500** by clicking the *Next Loop* button.

A Loop That Is Optimized Away

Loop **do-1500** is an example of a loop so unnecessary that PFA can get rid of it entirely. First, PFA sees that the body of the loop is independent of the loop, so it can be promoted out, and the loop eliminated. Then it sees that the body sets a variable that is not subsequently used, so it can throw that out, too. The transformed source is not scrolled and highlighted because nothing is there. Scroll down a few lines from the previous loop, and note the absence of the code for the loop that was optimized away.

The loop also has a directive controlling scalar optimization, but it is there only to reset the default for subsequent loops.

Move to loop **do-2000** by clicking the *Next Loop* button.

Loops with Obstacles to Parallelization

There are a number of reasons that a loop may not be parallelized. The following loops illustrate various of these reasons, along with variants that allow parallelization. You will step through each of them in turn.

Loops with Data Dependences

Loop **do-2000** is an example of a loop that cannot be parallelized because of a data dependence. In this case, one element of an array is used to set another. (This construct is called a recurrence.) If the loop were to be parallelized, the iterations might execute out of order, and iteration 4, which sets A(4) to A(5), might occur after iteration 5, which would have reset the value of A(5). Consequently, the program would give the wrong answer. See Figure 2-23.

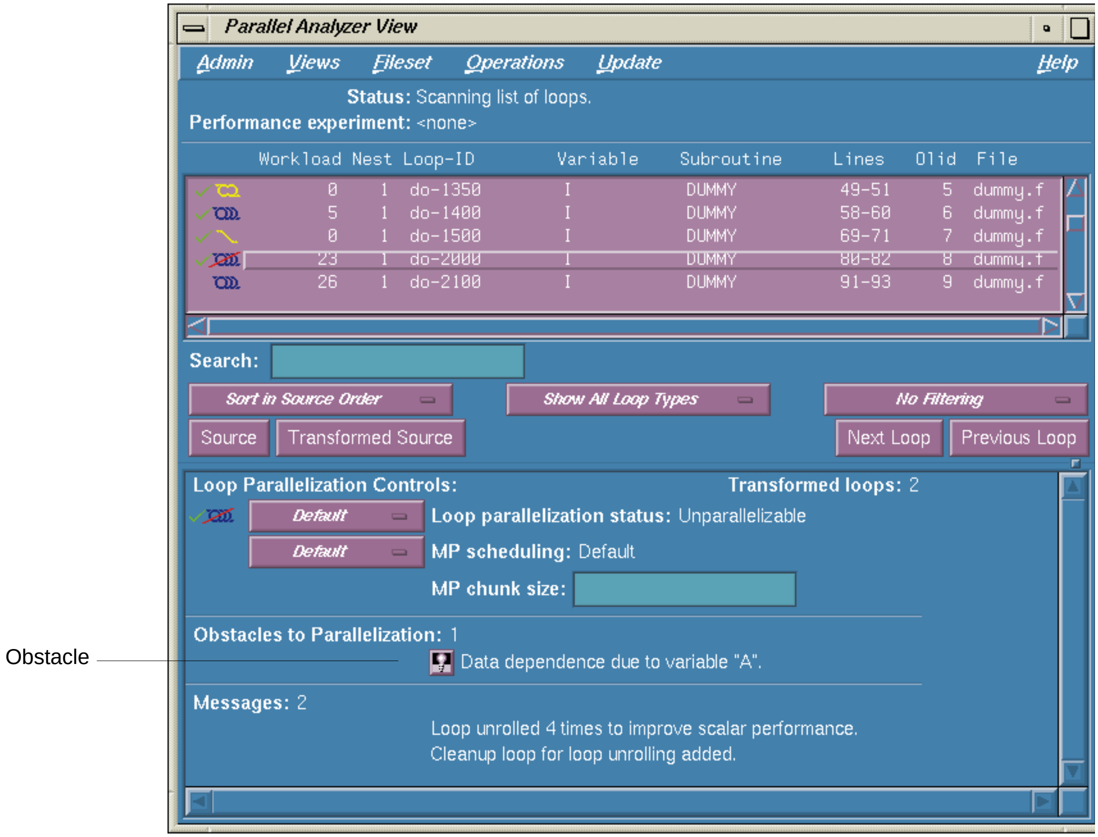


Figure 2-23 Obstacle to Parallelization

There is a line listing the obstacle to parallelization; click the button that accompanies it. Two kinds of highlighting take place. The first is a line highlight showing the relevant line that has the dependence, and the second is a symbol (or token) highlight that shows the uses of the variable that is the obstacle to parallelization. Only the uses of the variable within the loop are highlighted.

Move to loop **do-2100** by clicking the *Next Loop* button.

Not all loops with similar constructs are unparallelizable. Loop **do-2100** is similar to loop **do-2000**, but the array elements used differ by an offset, M. If

M is equal to NSIZE, for example, and the array is twice NSIZE, the code is actually copying the upper half of the array into the lower half, and there is no reason why that cannot be run in parallel. PFA cannot recognize this from the source, but the author has added an assertion that there is no recurrence, so the loop is parallelized. See Figure 2-24. Click the highlighting button to show the assertion.

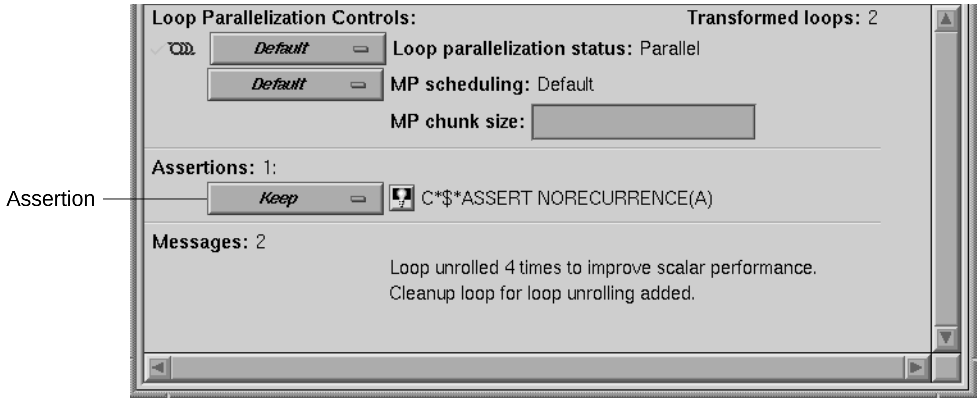


Figure 2-24 Parallelizable Data Dependence

Move to loop **do-2200** by clicking the *Next Loop* button.

Data dependence can involve more than one line of a program. In loop **do-2200**, a similar dependence occurs, but the use of the variable occurs on a different line than its setting. Click the highlight button on the obstacle line, and note that both lines receive the line highlighting, and the token highlighting shows the dependency variable on the two lines (see Figure 2-25). Of course, real programs can, and typically do, have far more complex dependencies than this.

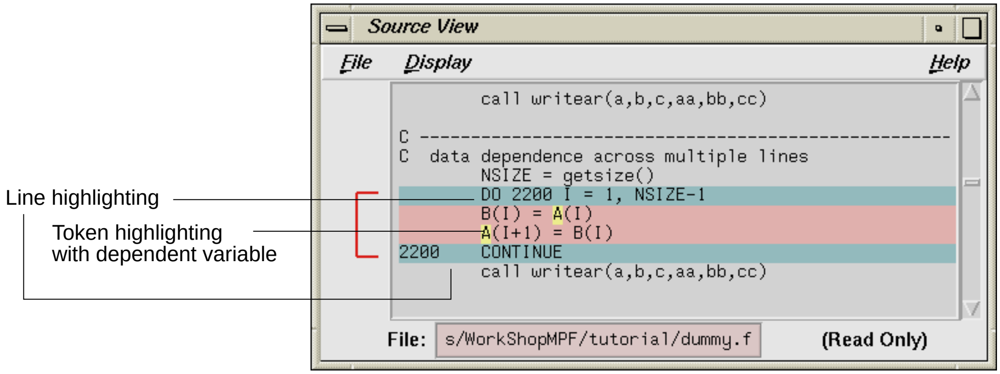


Figure 2-25 Highlighting on Multiple Lines

Move to loop **do-2300** by clicking the *Next Loop* button.

Loops with Reductions

Loop **do-2300** shows a data dependence that is called a *reduction*. In a reduction, the variable responsible for the data dependence is being accumulated or "reduced" in some fashion. Reductions can be summation, multiplication, or a minimum or maximum determination. For summation, as shown in this loop, PFA could accumulate partial sums in each processor, and then add the partial sums at the end. However, because floating-point arithmetic is inexact, the order of addition might give different answers because of round-off error.

This does not imply that the serial execution answer is "correct" and the parallel execution answer is "incorrect"; they are equally valid within the limits of round-off error. Since, by default, PFA assumes it is not OK to introduce round-off error, the loop is left serial. PFA does, however, have a parameter to allow you to say that such round-off error is OK.

Move to loop **do-2400** by clicking the *Next Loop* button.

In loop **do-2400**, the author has added a directive controlling round-off error. The same loop that was left serial above is now parallelized. Click the button for the directive, and you can see how it is highlighted in the source. Refer to the PFA manual for a more detailed explanation of the meaning and use of this directive. The round-off setting will be left at this value for the remainder of the program.

Move to loop **do-2500** by clicking the *Next Loop* button.

Loops with Input-output Operations

Loop **do-2500** has an input/output (I/O) operation in it. It cannot be parallelized, because the output would appear in a different order depending on the scheduling of the individual CPUs. Click the button indicating the obstacle, and note the highlighting of the print statement. Also note that the transformed source shows that this loop is not unrolled, either. Actually, there is no real obstacle to unrolling, but it is not done because the cost of performing the I/O operation is so great compared to the loop iteration overhead that the savings gained are not worth the increase in the size of the program.

Move to loop **do-2600** by clicking the *Next Loop* button.

Loops with Premature Exits

Loop **do-2600** has a premature exit; that is, it cannot be determined at compilation time how many iterations will take place. If PFA did parallelize it, one thread might execute iterations past the point where another has determined to exit the loop.

Click the button indicating the premature exit. Note that the line with the exit from the loop is highlighted in the source.

Move to loop **do-2700** by clicking the *Next Loop* button.

Loops with Subroutine Calls

Loop **do-2700** is also unparallelizable, because there is a call to a routine, *RTC*, and PFA cannot determine whether or not that call will have side effects. Click the obstacle line. Note the highlighting of the line containing the call and the subroutine name. Also note that the loop is not unrolled, as the presence of the call inhibits unrolling.

Move to loop **do-2800** by clicking the *Next Loop* button.

Although loop **do-2800** has a similar subroutine call in it, it can be parallelized because the author has asserted that the call has no side effects that will prevent it from running concurrently. Click the assertion line to highlight the source line containing the assertion.

When you are done, move to loop **do-3000** by clicking the *Next Loop* button.

Loops That Prompt Questions from PFA

Sometimes PFA can parallelize a loop more efficiently if it knows more information than it can infer from the source. In these cases, PFA asks questions that appear in the loop information display for the loop, along with a menu that allows you to answer the question.

Loops with Relationships between Variables

PFA can sometimes parallelize a loop if it can be told the relationship between variables in the program. Although you may know such relationships from the nature of the physical problem the program is dealing with, PFA cannot safely infer the information just from the code.

Loop **do-3000** can be parallelized if it is known that the iterations do not overlap, but not otherwise. PFA will ask three questions, although for this type of construct, it actually generates code to determine the relationship at run time, and the program will execute one of the two sequences depending on that determination. You can see this by observing that the loop was transformed into four loops, one pair of unroll / cleanup loops when it can be parallelized, and a second when it cannot. Look at the transformed source code for each of these pairs.

For any such questions, the line asking them has an associated option menu that will allow you to answer. The generated code will be correct even if you do not answer or do not know. If PFA knows the answer, it can omit the alternate form and produce a tighter program.

Move to loop **do-3100** by clicking the *Next Loop* button.

In loop **do-3100**, the author has added an assertion answering the question, and PFA has generated just one version of the loop, the one that runs in parallel. The menu next to the questions for the previous loop will generate such an assertion.

Move to loop **do-3200** by clicking the *Next Loop* button.

Permutation Vectors

Loop **do-3200** has a construct known as a permutation vector. In it, an array is referenced by an index value contained in another array. If the B(I) values are all distinct, the iterations do not depend on each other, and the loop can be parallelized; if the same value occurs in more than one B(I), it cannot. PFA asks the question but leaves the loop serial. Note that both the question and the data dependence message have associated highlighting buttons.

Move to loop **do-3300** by clicking the *Next Loop* button.

Here an assertion has been added that the index array, B(I), is indeed a permutation vector, and the loop is parallelized.

Move to loop **do-4000** by clicking the *Next Loop* button.

Complex Loops and Loop Nests

Finally, let's look at somewhat more complicated, nested loops.

Doubly-nested Loops and Interchanges

Loop **do-4000** is the outer loop of a pair of loops; it runs in parallel, and the inner loop runs in serial: one parallel loop cannot be nested inside another. Also note that the outer loop is not unrolled, but the inner loop is.

Move to loop **do-4010** by clicking the *Next Loop* button to show the inner loop, and then click *Next Loop* again to select the outer loop of the next pair.

Note that this outer loop, loop **do-4100**, is shown as serial inside a parallel loop, and the following loop is parallel. How can this be? It happens because PFA has recognized that the two loops can be interchanged, and furthermore, that the CPU cache is likely to be more efficiently used if the loops are run in the interchanged order.

Move to loop **do-4110** to show the inner loop, and then click the *Next Loop* button once again to move to the following triply-nested loop.

Triply-nested Loops and Strip-mining

The next set of loops is a triply-nested matrix multiply. Just as PFA optimized a doubly-nested pair of loops by interchanging the loops, it will do even more to get optimal cache performance by “strip-mining” a triply-nested loop. In this case, different sections of the matrix will be executed by different threads, so that the threads will not cause cache conflicts among themselves.

The outer original loop, **do-5000**, is interchanged, unrolled, and split into block and strip loops, in a fairly complicated way; it is transformed into ten loops. The middle loop has part of its work in a second-level unrolled loop, and part of it in parallelized third-level loops. The inner loop is shown as unparallelizable, although it is actually preferably serial. (This is a bug in the current version of WorkShopProMPF.) Do not be surprised if the code seems difficult to understand; the strip-mining transformation is very complex and confusing.

Use the *Next Loop* button to first step to the middle of the three, loop **do-5010**, and then the inner one, loop **do-5020**. Notice how each of the loops is transformed into various combinations of loops at different nesting levels.

This brings you to the end of your examination of the loops under analysis. In the next section, you will find out how to modify your source code using the Parallel Analyzer.

Modifying Source Files

So far, you've ignored the controls that can be used to change the source file and allow a subsequent pass of PFA to do a better job. Now you will go back and make changes. There are two steps in modifying source files:

1. Asking for the changes using the Parallel Analyzer View controls.
2. Actually modifying the files and rebuilding the program and its analysis files.

Asking for Changes

You may ask for changes by answering any of the questions that PFA poses, by building a DOACROSS for a specific loop, by modifying the analysis parameters that PFA uses for its processing, or by adding or deleting assertions or directives. In this sample session, you will request changes to loops in the order they appear in the file, but they may be requested in any order.

Changing the PFA Analysis Parameters

Scroll to the top of the loop list and select the first loop, which was unrolled four times. Pull down the Views menu and select "PFA Analysis Parameters View" to open the PFA Analysis Parameters View. Locate the line that reads:

```
Unroll:
```

Figure 2-26 Changing a PFA Analysis Parameter

Enter **6** into the numeric field next to it (it contains 4 by default). First click the field and then type **<Backspace>** followed by **6**. This changes the loop unrolling from 4 to 6. Note the turned-down corner in the text field as shown in Figure 2-26. Clicking this corner toggles between the old and new values in the field.

Close the View by pulling down the Admin menu and selecting "Close." Notice that a red plus sign now appears in the icon next to the loop, indicating that a change has been requested for it as shown in Figure 2-27. Move to loop **do-1100** by clicking the *Next Loop* button.

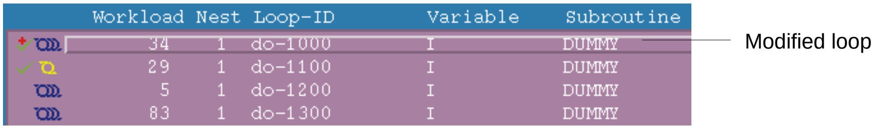


Figure 2-27 Effect of Changes on the Loop List Display

Building a Custom DOACROSS Directive

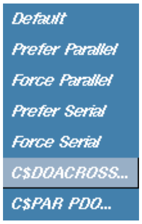


Figure 2-28
DOACROSS Menu

Loop **do-1100** was left serial because it was too small; sometimes you might want such a loop parallelized anyway. Go to the Loop Status option menu (to the left of the loop status icon in the loop information display that reads “Default”), and select “C$DOACROSS...” as shown in Figure 2-28. This brings up the Parallelization Control View (see Figure 2-29), showing the loop that was selected, a parallelized condition input field into which you can type a condition for parallelization, an MP scheduling option menu, an MP chunk size input field, and a list of all the variables in the loop, with an icon indicating whether the variable was read, written, or both. (These icons are described in the Icon Legend.) Notice that each variable has a highlighting button that shows its use within the loop.Notice also the red plus sign next to this loop in the main view.

Dismiss the View by pulling down the Admin menu and selecting “Close.”

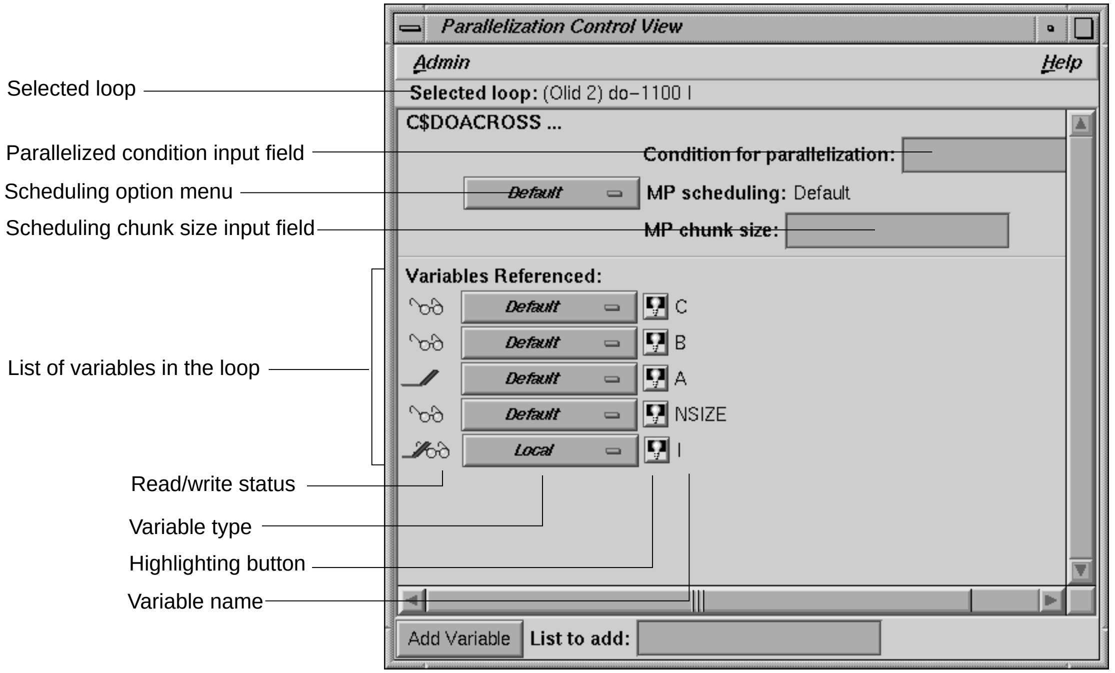


Figure 2-29 Parallelization Control View for Loop **do-1100**

Adding a New Assertion

Now you will add an assertion to a loop. Find the loop with ID **do-2700** by using the search feature of the loop list. Go to the search field, and enter **`2700`**. You can double-click the highlighted line in the loop list to select the loop.

You're going to add a concurrent call assertion. To add the assertion,pull down the Operations menu, pull down the Add Assertion submenu, and select "C*$*ASSERT CONCURRENT CALL."

This adds an assertion that the call to *RTC*(), which PFA thought to be an obstacle to parallelization, is actually safe to parallelize. When you add the assertion, the loop information display updates to show the new assertion, along with its menu labeled "Insert" as shown in Figure 2-30.

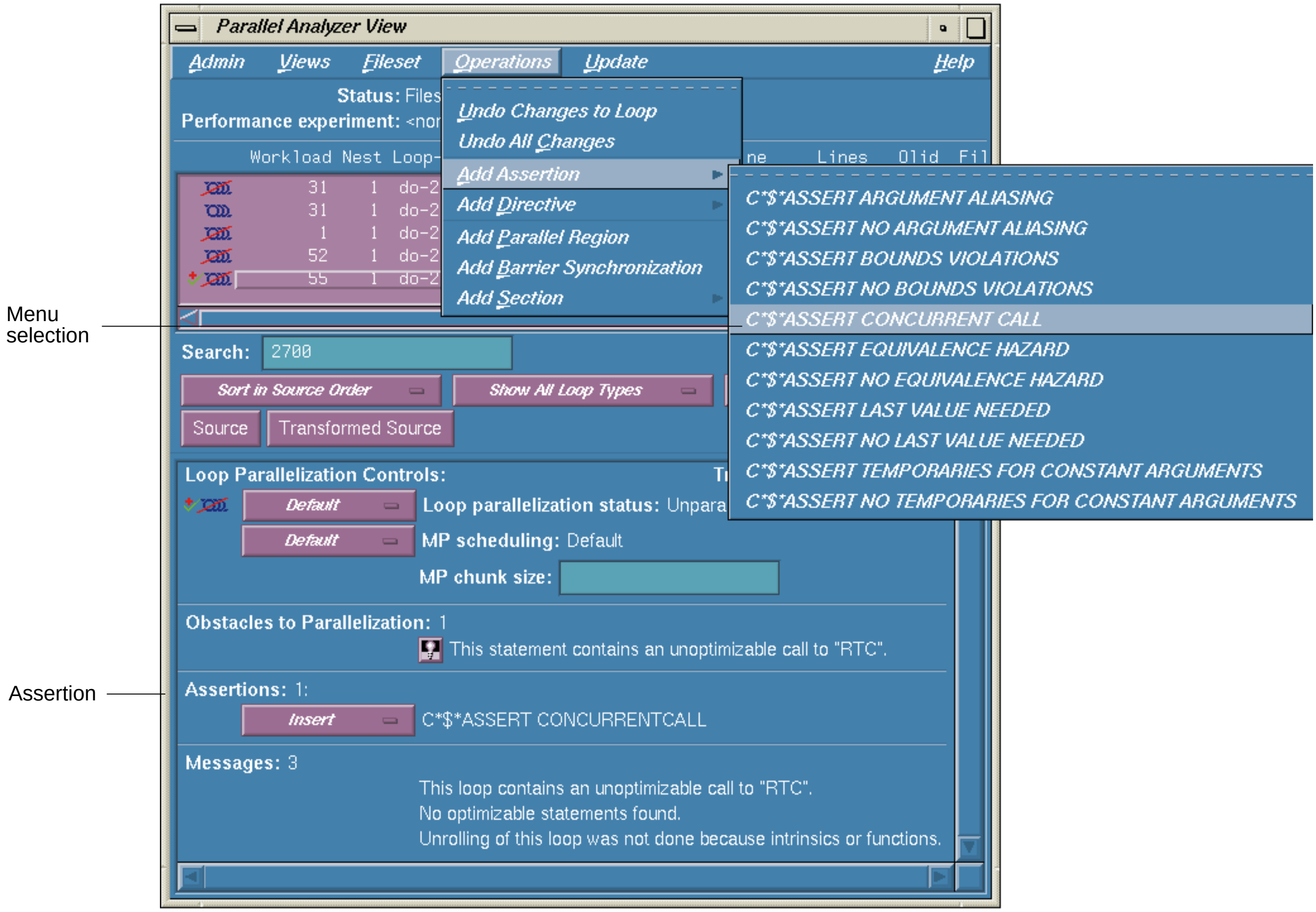


Figure 2-30 Adding an Assertion

Answering a Question

Now try answering a question. Put the cursor into the search field, backspace to remove the previous contents, and enter **`3200`** into the field. Select that loop by double-clicking. Loop **do-3200** has a question about a permutation vector. Pull down the option menu next to the question in the loop information display, and select "Assert True" as shown in Figure 2-31.

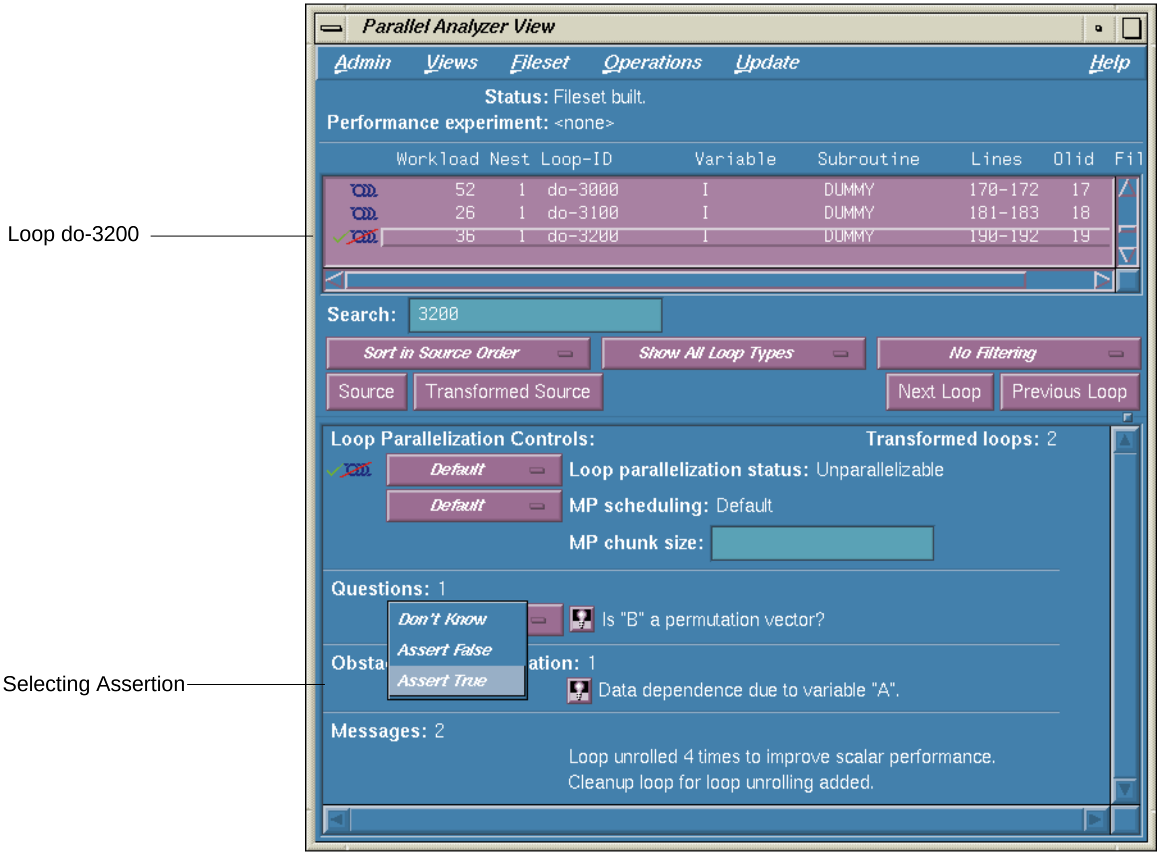


Figure 2-31 Answering a Question

Deleting an Existing Assertion

Now let's delete an existing assertion. Move to loop **do-3300** using the *Next Loop* button, and go to the "ASSERT PERMUTATION(B)" assertion. Pull down its option menu and select "Delete". Figure 2-32 shows the result. The same procedure can be used for directives.

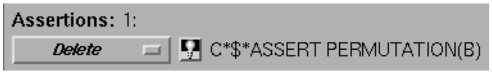


Figure 2-32 Deleting an Assertion

Updating the File

Now you have made a set of changes and can update the file. Select “Update All Files” from the Update menu (see Figure 2-33); alternatively, you may use the keyboard accelerator for this operation by typing **`Ctrl-U`** with the cursor anywhere in the main view. The Parallel Analyzer View will generate a *sed* script to modify the source, rename the original file to one with the suffix *.old*, run *sed* on that file to produce a new version of the file *dummy.f*, and then spawn the WorkShop Build Manager to rerun PFA on the new version of the file.

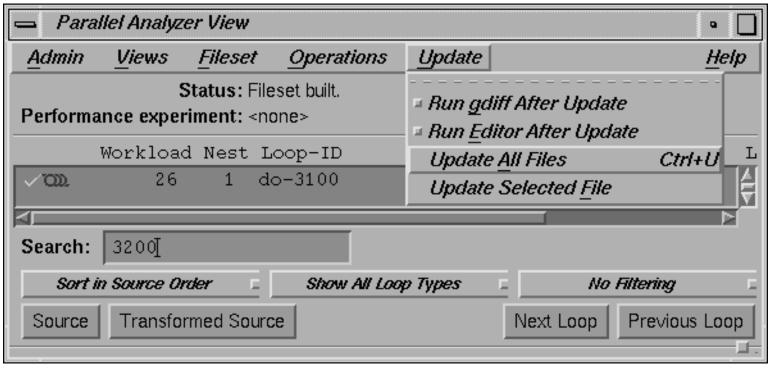


Figure 2-33 Update All Files

The Parallel Analyzer View can also open a *gdiff* window showing the changes, but by default it does not. If you select the toggle labeled “Run gdiff After Update” from the Update menu, it will do so. If you have selected it, use the right mouse button to step through the changes, and then quit *gdiff*. If you always wish to see the *gdiff* window, you can set the resource in your *.Xdefaults* file:

```
cvpav*gDiff: True
```

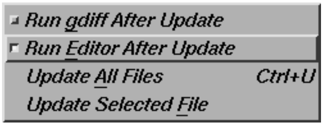


Figure 2-34 Setting the Run Editor Toggle

The Parallel Analyzer View can also open an editor for you to make additional changes after running *sed*. To do so, select the toggle labeled “Run Editor After Update” in the Update menu (see Figure 2-34). If you do so, an *xwsh* window with *vi* running in it opens after you update the 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 change the resource in your *.Xdefaults* file, changing the *xwsh* and / or *vi* as you prefer:

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

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 will either replace any explicit **%s**, or if the **%s** is omitted, the file name will be appended to the command.

After you quit from the *gdiff* window and / or editor (if you have selected them), the program will spawn the WorkShop Build Manager. When it comes up, verify that the directory shown is the directory in which you are running the sample session; if not, change it. Then, click the *Build* button, and it will start to reprocess the changed file.

Examining the Modified File

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

Unroll Change

Click the *Next Loop* button twice to select the first loop. Notice that loop **do-1000** is now shown as being unrolled six times, not four as it was before. Also the loop has a directive, implementing the change in unrolling that was requested.

Move to loop **do-1100** by clicking the *Next Loop* button.

New Custom DOACROSS

Loop **do-1200** previously was serial because it had too little work in it, but is now parallel because it was explicitly parallelized.

New Assertion

Go to the search field and enter **`2700`**. Double-click the line and notice that loop **do-2700**, which previously was unparallelizable because of the call to *RTC*(), is now parallel. It also has the assertion that was added.

Answered Question

Clear the search field, enter **`3200`** in it, and double-click the selected line. Notice that loop **do-3200** now has an assertion in it, added as a result of your reply to the question. The loop is also now parallelized.

Move to loop **do-3300** by clicking the *Next Loop* button.

Deleted Assertion

Loop **do-3300** previously had the assertion that B was a permutation vector; note that the assertion is gone, and PFA now asks the question.

Examing Subroutines That Use PCF Directives

PCF directives are not supported by the current 32-bit PFA processor. If you put them into your code, they will be treated as comments, rather than properly interpreted. The six loops, do-6001 through do-6006 are processed ignoring the directives. To see the effect of the directives, see “Examining Subroutines That Use PCF Directives” in Chapter 3.

Examining a Subroutine That Contains Syntax Errors

The PFA preprocessor does not provide error messages in the analysis file to show what the syntax errors were, so WorkShopProMP cannot show them. The routine itself is shown with the error indicator for it, but no highlighting button and messages will appear. To understand the errors, look at the listing file, dummy.l, in the directory. More information is provided in the 64-bit tutorial, q.v.

Exiting from the Dummy Sample Session

This completes the first sample session. Quit the Parallel Analyzer View by pulling down the Admin menu and selecting “Exit.”

To clean up the directory, so that the session can be rerun, enter:

```
% make clean
```

in your shell window. All of the generated files will be removed.

Setting Up the linpackd Sample Session

The second sample session is a brief demonstration of the integration of WorkShopProMPF and the WorkShop performance tools. It requires that WorkShop also be installed.

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

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

This will update the directory by compiling the source program *linpackd.f* and creating the necessary files. The performance experiment you will use is already there. This operation will take a few minutes.

Starting the Parallel Analysis View

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

```
% cvpav -e linpackd
```

from within the directory (note the flag is **`-e`**, not **`-f`** as in the previous sample session). The main window of the Parallel Analysis View will open, showing the list of loops in the program.

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

Starting the Performance Analyzer

Start the Performance Analyzer by pulling down the Admin menu, selecting the Launch Tool submenu, and selecting “Performance Analyzer,” as shown in Figure 2-35.

The main window of the Performance Analyzer will open, although it will be empty. A small window labeled “Experiment:” will also open at the same time. This window is used to enter the name of an experiment. For this session, we will use the prerecorded experiment that is installed. Type:

```
test.linpack.cpu
```

in the “Experiment Dir:” field in the Experiment: window, and click the *OK* button. See Figure 2-35. The Performance Analyzer will show a busy cursor, fill in its main window with the list of functions, and highlight the function **main**().

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

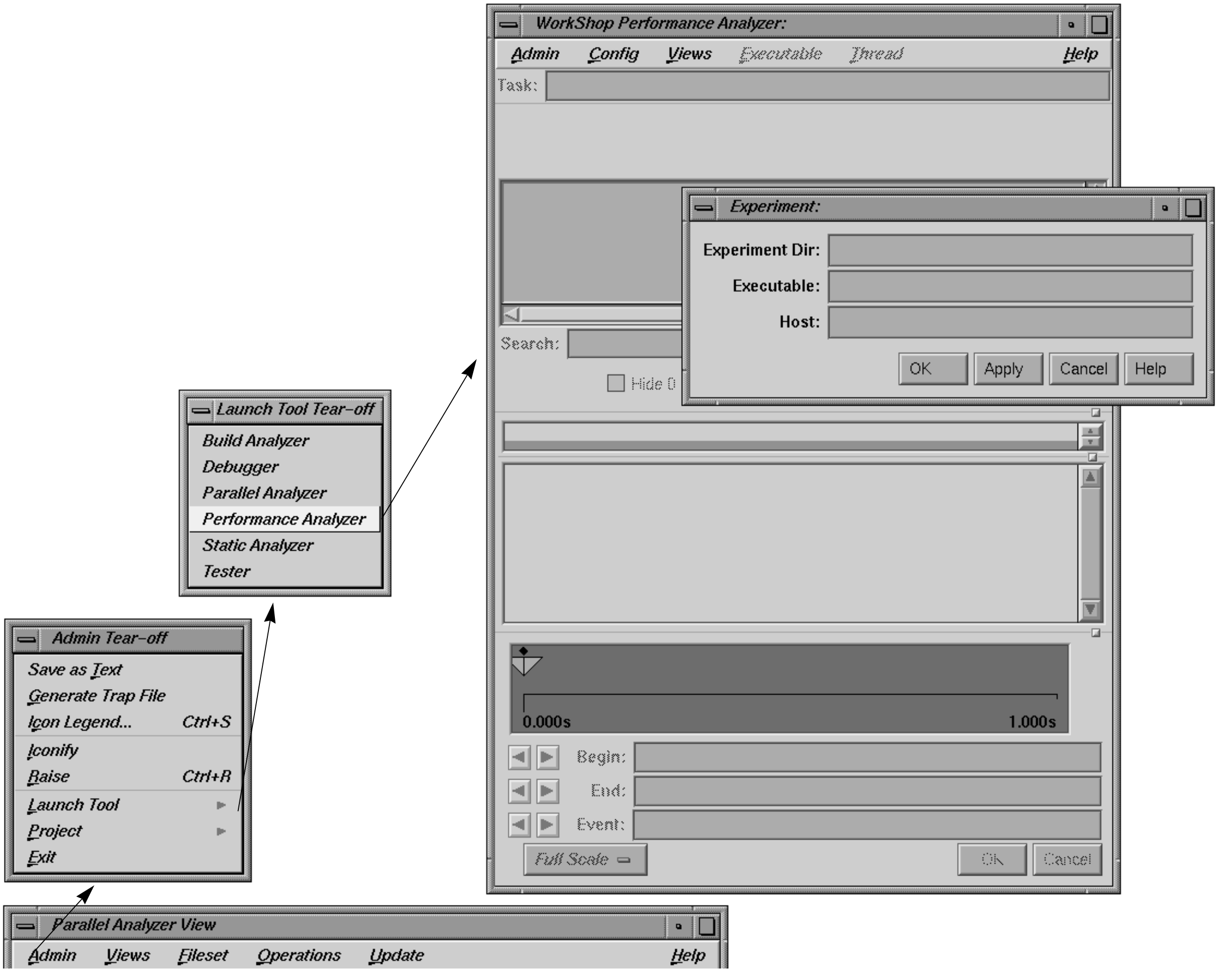


Figure 2-35 Starting the Performance Analyzer

Using the Parallel Analyzer with Performance Data

At the same time the Performance Analyzer window fills in, the Parallel Analyzer recognizes that there is now a performance analyzer, and posts a busy cursor with a message “Loading Performance Data.” When the message goes away, performance data will have been imported by the

Parallel Analyzer, and a number of changes will have taken place as shown in Figure 2-36:

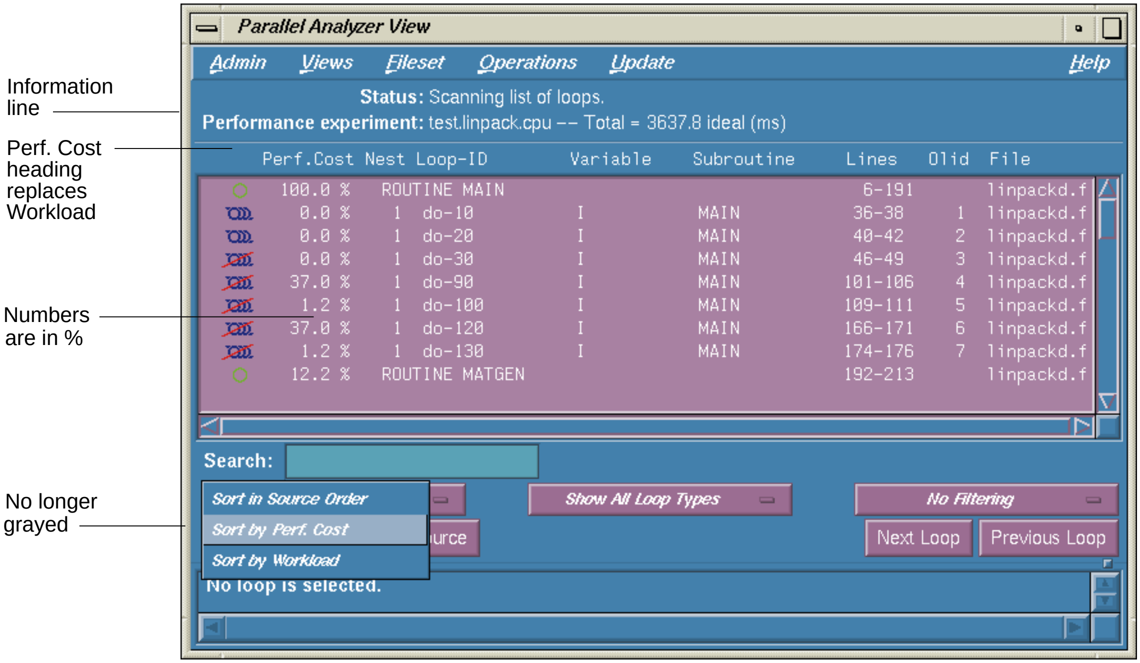


Figure 2-36 Performance Data — Parallel Analyzer View

- The second column of the list of loops has changed from reading “Workload” to reading “Perf. Cost”, and the numbers below it are now percentages.
- The second line in the view now shows the name of the performance experiment and shows the total cost of the run. In addition, the sort menu’s second entry “Sort by Perf. Cost” is no longer grayed-out.
- The Source View now has three additional columns to the left of the loop brackets that show the performance metrics, including the number of times the line has been executed and ideal CPU times as shown in Figure 2-37. The times are exclusive, inclusive, ideal, or CPU time in milliseconds.

These columns reflect the measured performance data. If you select loop **do-30** of subroutine DAXPY from the main view, the Source View displays as shown in Figure 2-37.

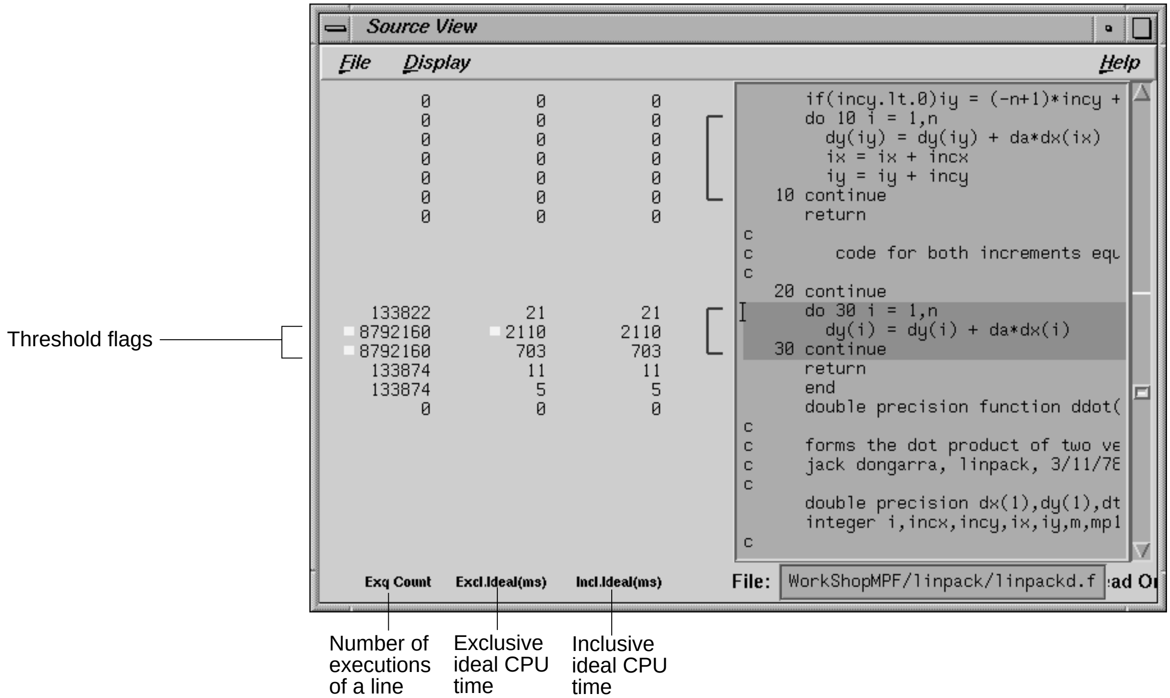


Figure 2-37 Source View for Performance Experiment

Select the “Sort by Perf. Cost” entry. Note that the top three lines now show three loops that represent approximately 85%, 82%, and 81% of the total time. These numbers are inclusive numbers, with each reflecting the time in the loop and in any nested loops or functions called from within the loop. See Figure 2-38.

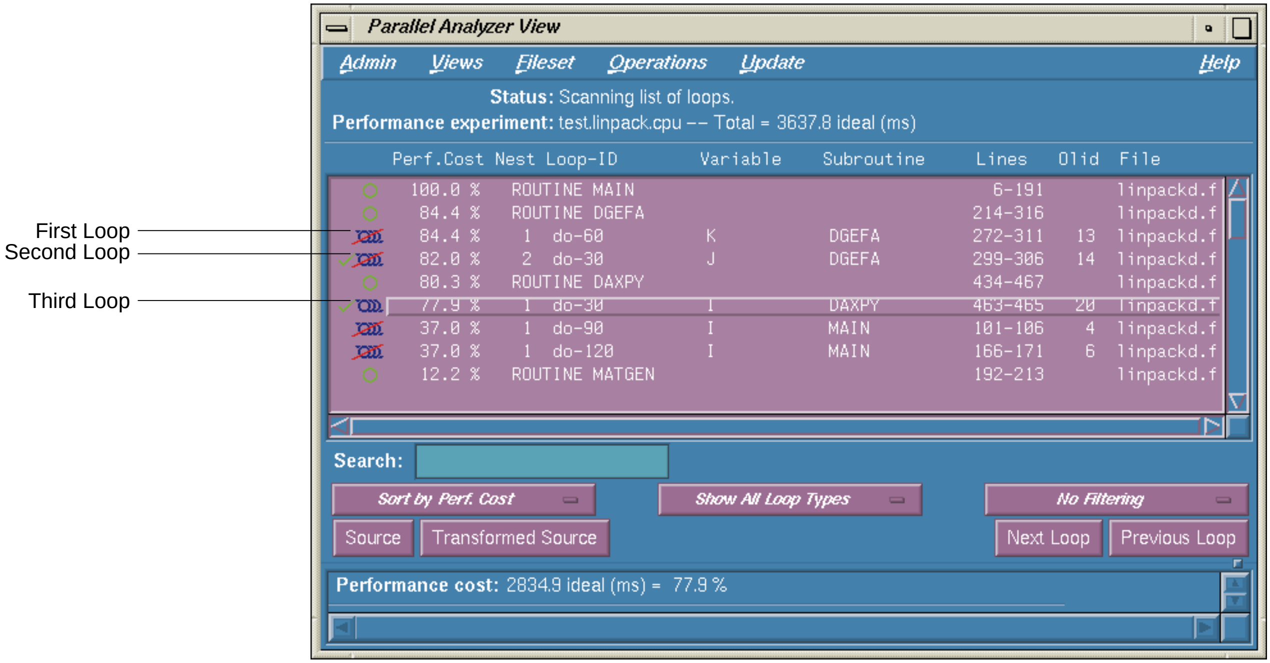


Figure 2-38 Sort by Performance Cost

The first of these loops contains the second loop nested inside it. The second loop calls the subroutine DAXPY, which contains the third loop. The third loop is the heart of the *linpack* benchmark and is already parallel.

Double-click the third loop. Note that the loop information display now contains an additional line of text listing the performance cost of the loop, both in time and as a percentage of the total time. See Figure 2-39.

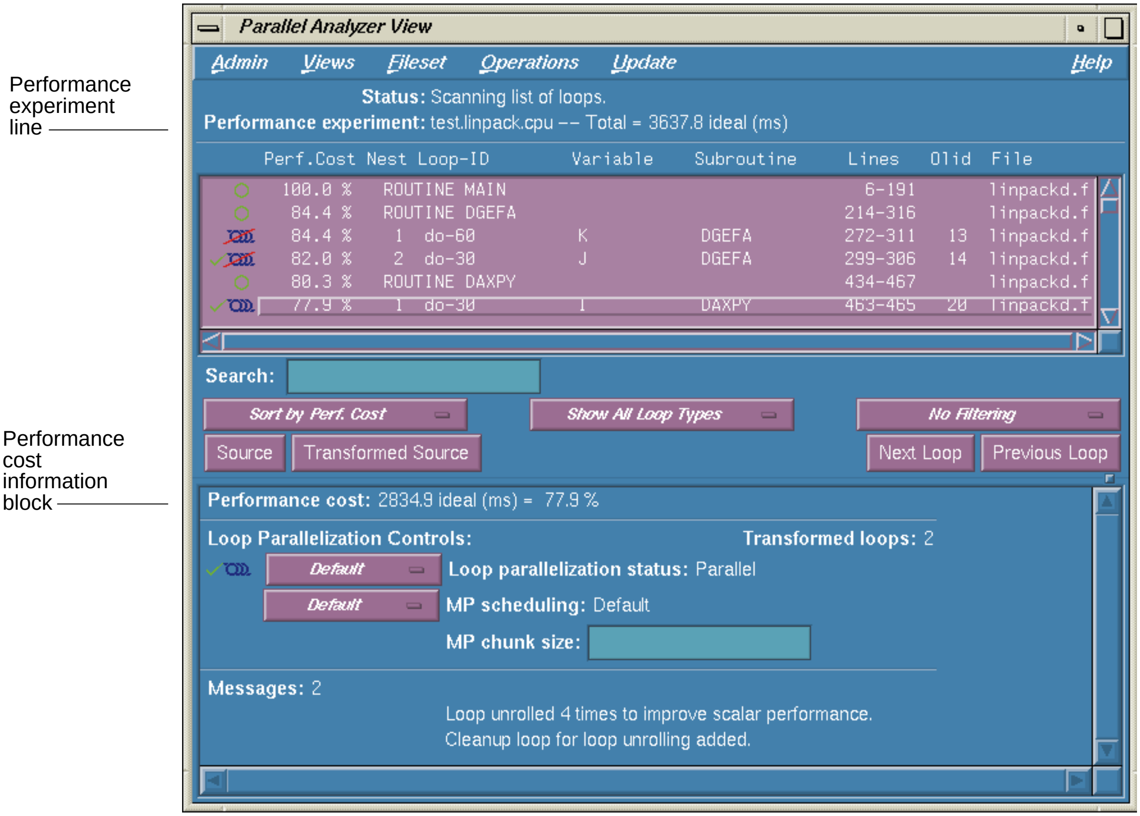


Figure 2-39 Loop Information Display with Performance Data

Exiting from the linpackd Sample Session

This completes the second sample session. Quit by selecting the "Exit" command from the Project submenu of the Admin menu in the Parallel Analyzer View. All the windows will close.

You don't need to clean the directory, because you haven't made any changes in this session. If you do make changes, when you are finished you can clean up the directory by entering:

```
% make clean
```

in your shell window. All generated files will be removed.

Setting Up the f90 Sample Session

The f90 sample session is located in the directory */usr/demos/WorkShopMPF/cgdriver*. Prepare for the session by changing directories to the demo directory and creating the needed files:

```
% cd /usr/demos/WorkShopMPF/cgdriver
% make
```

Once the demo directory has been prepared, start the session by entering:

```
% cvpav -f cgdriver.f
```

Notice that the loop list contains Fortran 90 array syntax statements. Double click on the first statement in **CGTEST** (b = 0). You can see in the loop information display that the array-syntax is an implied loop and the statement was converted from array notation into a serial loop.

Click on the *Source* button. Notice that in source view, Fortran 90 array syntax statements (in the subroutine **CGTEST**) are bracketed in blue (they are shown as loops). Click on the *Transformed Source* button to see the transformation that PFA has performed. You can see that since *b* is a 3-dimensional array which is initialized to 0, the transformed source contains 3 nested do loops (each one spanning one dimension).

Exiting from the f90 Sample Session

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

`To clean up the directory, so that the session can be rerun, enter:`

```
% make clean
```

in your shell window. All of the generated files will be removed.

Analyzing Loops: 64-bit Sample Sessions

This chapter provides three interactive sample sessions that demonstrate most of the Parallel Analyzer View's features for the 64-bit version of MPF. These sessions also demonstrate various aspects of parallelization and the use of the POWER Fortran Accelerator (PFA).

The sample sessions consist of a step-by-step examination of three sample programs. The samples sessions cover the following:

- The dummy sample session is designed to show the various types of FORTRAN loops, how they are transformed by PFA, and how they are displayed by the Parallel Analyzer View. (The major difference between this and the 32-bit dummy sample session is the use of PCF directives.) The sample session begin at "Setting Up the Dummy Sample Session" on page 54.
- The linpackd sample session briefly illustrates how the Parallel Analyzer View can be used in conjunction with the WorkShop Performance Analyzer *cvperf*. The sample session begin at "Setting Up the linpackd Sample Session" on page 93.
- The f90 sample session briefly illustrates how to use MPF with Fortran-90 code. The sample session begin at "Setting Up the f90 Sample Session" on page 100.

To use these sample sessions, the subsystem *WorkShopMPF_sw.demos* must be installed.

Note: These sample sessions are applicable for the 64-bit compilers only. For a discussion of the 32-bit version of the compilers, see Chapter 2, "Analyzing Loops: 32-bit Sample Sessions."

Setting Up the Dummy Sample Session

The Parallel Analyzer View comes with a demonstration directory */usr/demos/WorkShopMPF*. It contains a subdirectory *tutorial*, which contains a source file called *dummy.f_orig* and a *Makefile*. The file contains 27 DO loops, each of which exemplifies one aspect of the parallelization process. In that directory, running *make* creates a scratch copy of the demonstration program *dummy.f* and then creates a run of PFA on the copy. PFA produces a transformed source file *dummy.m*, a listing file *dummy.l*, and an “analysis” file *dummy.anl*.

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

```
% cd /usr/demos/WorkShopMPF/tutorial64
% make
```

You will get a series of *make* errors concluding with the following:

```
4 errors 1 warning in file dummy.f
*** Error code 1
smake: 1 error
```

These errors are in the code intentionally. You will study them later in “Examining a Subroutine That Contains Syntax Errors” on page 91.

Once the demo directory has been prepared, start the session by entering:

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

The main window of the Parallel Analyzer View opens, displaying the list of loops in the source file, *dummy.f*. Position the view at the upper left of the screen.

Note: If you receive a message related to licensing, refer to the *NetLS License System Administration Guide* or *WorkShopProMPF Release Notes*.

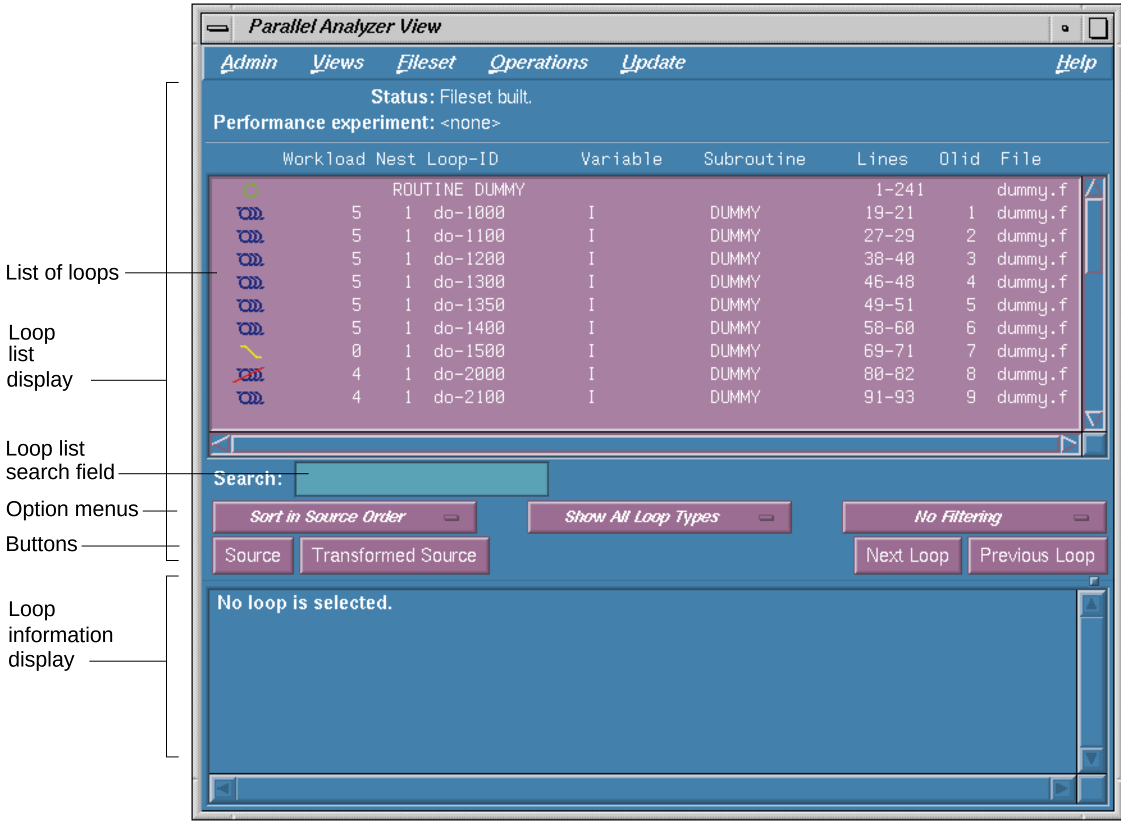


Figure 3-1 Parallel Analyzer View Main Window

Using the Loop List Display

The loop list display shows information about each loop in the program with an icon next to it that reflects the parallelization status of the loop. Pull down the Admin menu and select “Icon Legend...” to bring up a legend dialog box that explains the meaning of the various icons (see Figure 3-2). Move the legend dialog box to the side, and scroll through the list of loops to see the various icons. When you are done, close the legend dialog box by clicking the *Close* button in the lower right of the dialog box.

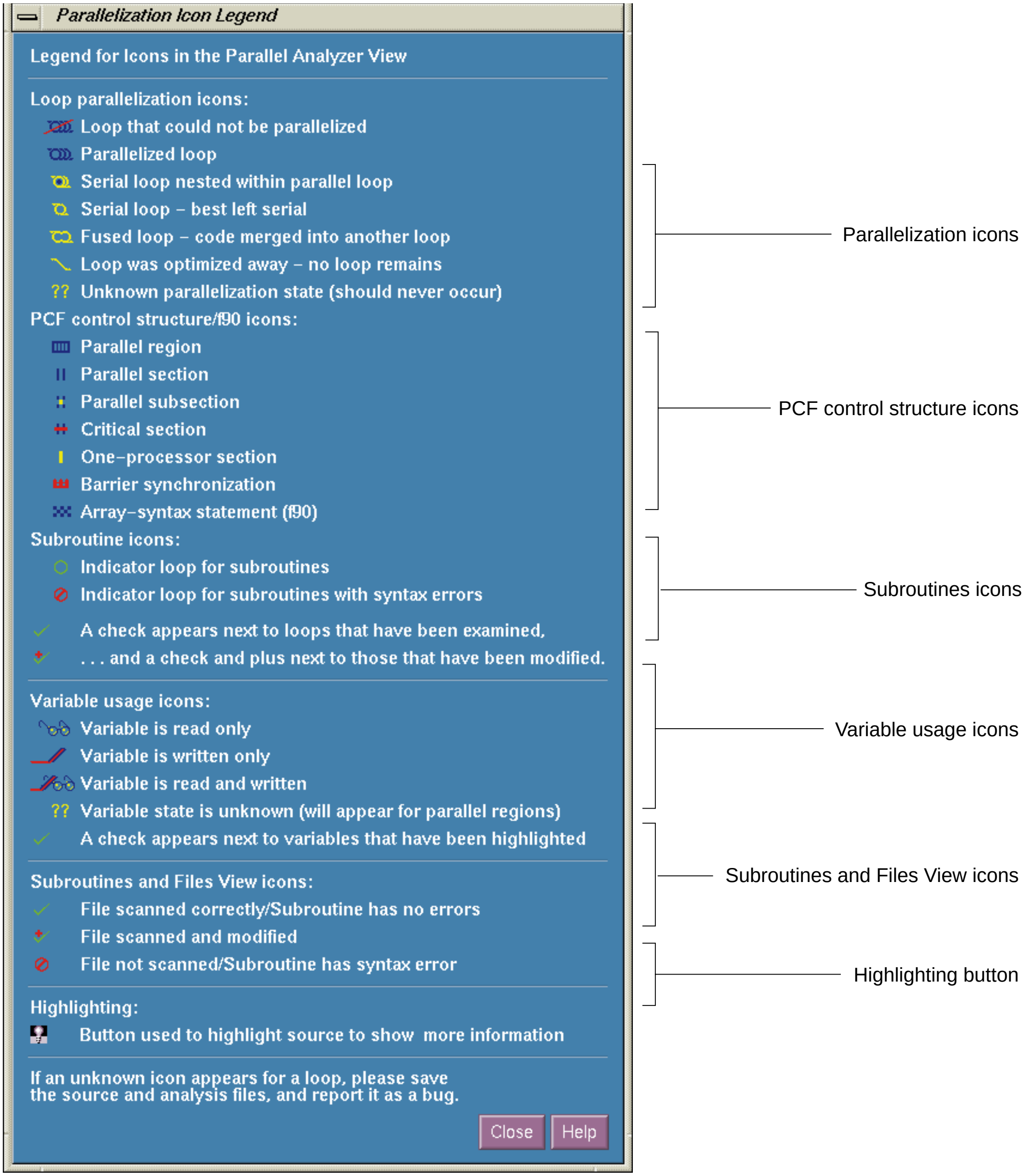


Figure 3-2 Launching the "Icon Legend..." Dialog Box

The loop list display contains the following items:

Workload a number that is supposed to reflect the amount of work done in each iteration of the loop

Nest the nesting level for the loop

Loop-ID the FORTRAN description of the loop

Variable the loop index variable

Subroutine, Lines, File
where the loop is located in the source code

Olid the original loop ID; an internal identifier for the loop (Please refer to this number when reporting bugs.)

Underneath the list display is a search field and a set of option menus and buttons that control the display of information in the loop list.

Sorting the Loop List

You can sort the list either in the order of the source code, or by loop workload, or (if you are running a performance experiment on the program using the WorkShop Performance Analyzer) by performance cost. You control sorting with the option menu to the left below the list.

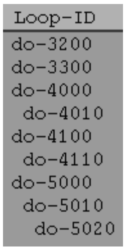


Figure 3-3 Source Order Sort

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 will have to scroll down to see it (see Figure 3-3).

For other sorting, the list is not indented. Select “Sort by Workload” and notice the Loop-ID is no longer indented (see Figure 3-4). (The same is true of “Sort by Perf. Cost”. It is grayed out because there is no performance tool running at this time.) When you are done, select “Sort in Source Order” once again.

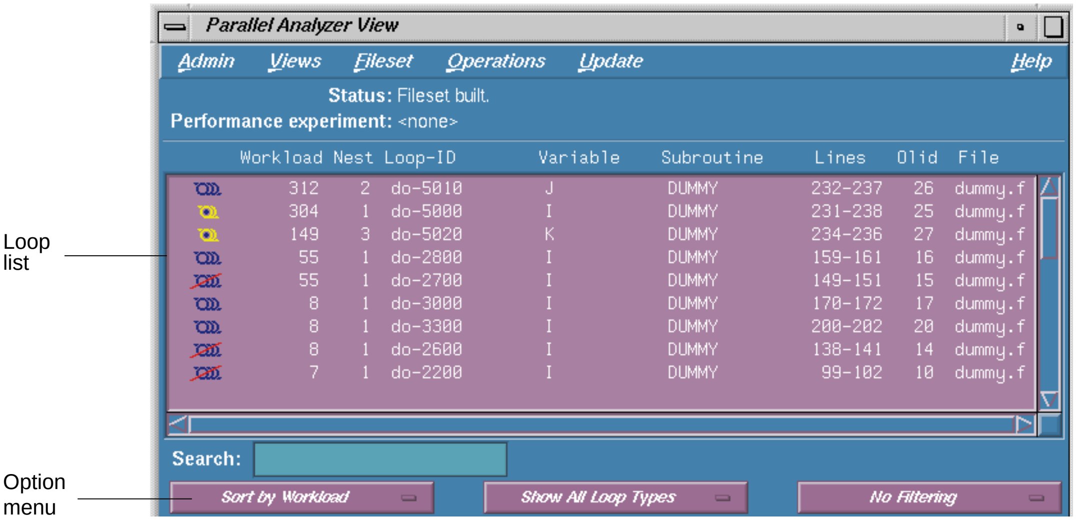


Figure 3-4 Sorting the Loop List by Workload

Filtering the Loop List

You may want to look at only some of the loops in large programs. The list can be filtered in two ways: by parallelization status or by origin of the loop.

Filtering by Parallelization Status

The parallelization status filtering is controlled by an option menu centered below the list. It initially reads “Show All Loop Types”.

You can filter the list to show only those loops that cannot be parallelized, those that are parallel, or those that are serial (see Figure 3-5). Try selecting each of these, and then return to “Show All Loop Types”. It can also filter to show those loops for which you have requested modifications (requesting modifications to loops is described later in this section). Since you haven’t yet requested any modifications, selecting this option will result in a message saying that no loops meet the filter criterion.

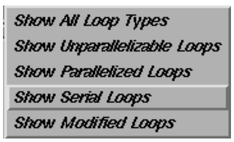


Figure 3-5 Parallelization Status Option Menu

Filtering by Loop Origin

Another way to filter is to show loops that come from a single file or a single subroutine:

1. Open the Subroutines and Files View by pulling down the Views menu and selecting “Subroutines and Files View.” Alternatively, you may use the keyboard accelerator for this operation by typing **`<Ctrl>-F`** with the cursor anywhere in the main view. A subsidiary view that lists the subroutines and files that are in the fileset opens (See Figure 3-6.)

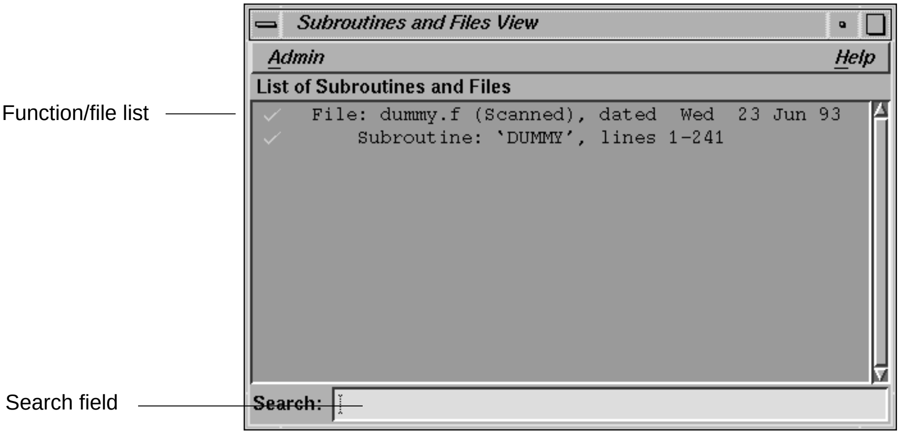


Figure 3-6 Subroutines and Files View

2. From the Filter option menu (figure 3-7), select “Filter by File.”

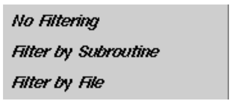


Figure 3-7 Filter Option Menu

3. Double-click the line for the file *dummy.f* in the function / file list of the Subroutines and Files View window. The name will appear in the filtering text field labeled Title: (see Figure 3-8) and the list will be rescanned. Similarly, you may try selecting “Filter by Subroutine” from the main view option menu, and double-click the line for subroutine DUMMY in the Subroutine and Files View.

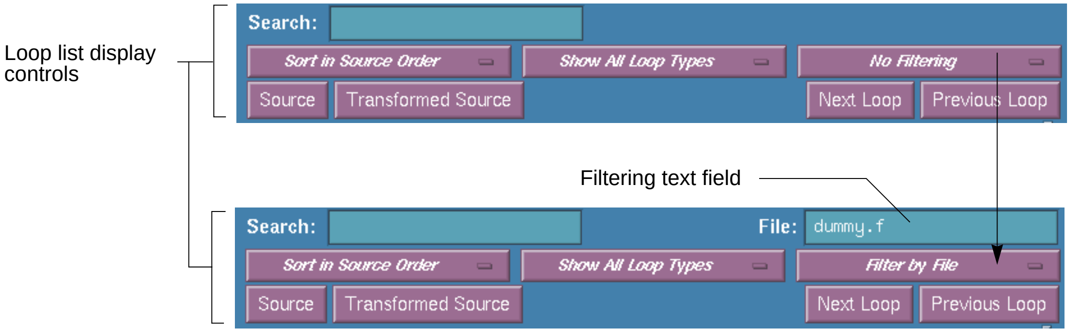


Figure 3-8 Filter by File Option Menu and Text Field

For this example, there is only one file and one subroutine, so the filtering is not very useful, but for large programs with many files and subroutines, it would be. When you are done, display all of the loops in the sample source file once again by selecting "No Filtering" from that option menu.

You won't be needing the Subroutines and Files View further, so close it by pulling down the Admin menu and selecting "Close."

Viewing Source

The Parallel Analyzer View gives you access to views of both your original Fortran source and the source as it is transformed by the POWER Fortran Accelerator.

Viewing Original Source

Click the *Source* button to the left side of the main view to bring up the Source View, as shown in Figure 3-9. This view is the same Source View that is used in the WorkShop Debugger and Performance Analyzer.

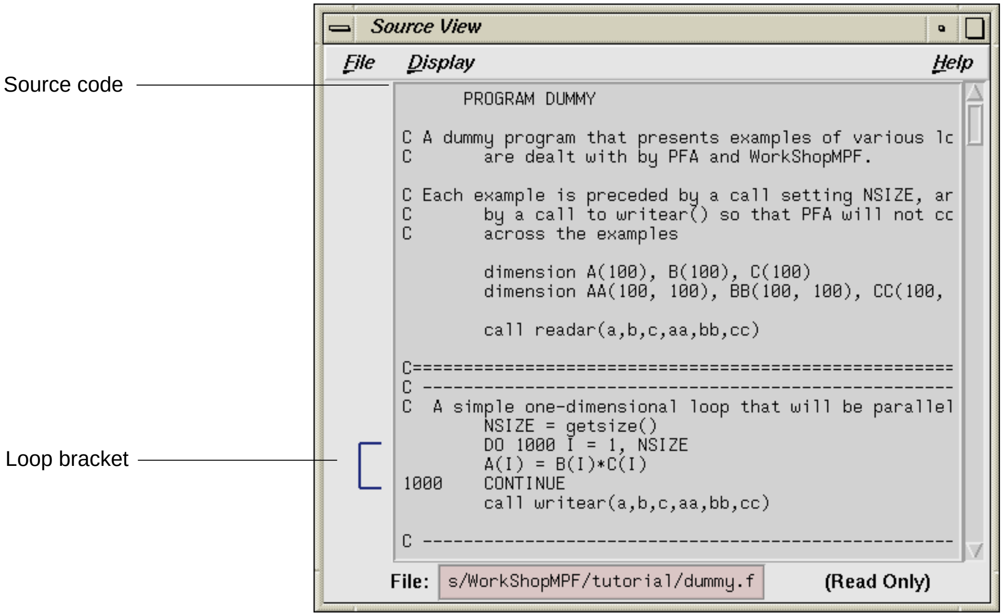


Figure 3-9 Source View

When the source display opens, position it to the right of the main view. (On machines with low-resolution screens, the windows will overlap.) Scroll up and down in the file and observe that the source window displays colored brackets that mark the location of each loop. These colors match the colors of the parallelization icons and serve to indicate the parallelization status of each loop at a glance. The color indicates which loops are parallelized, which are unparallelizable, and which are left serial.

Viewing Transformed Source

PFA is a source-to-source translator that takes the various loops in the program and transforms them both for scalar optimization and for parallelization. Each loop may be rewritten into one, two, or more transformed loops or may be combined with others or optimized away. The result of these transformations is a transformed source file that you may examine.

Click the *Transformed Source* button. Another source window labeled "Parallel Analyzer View — Transformed Source" opens as shown in Figure 3-10.

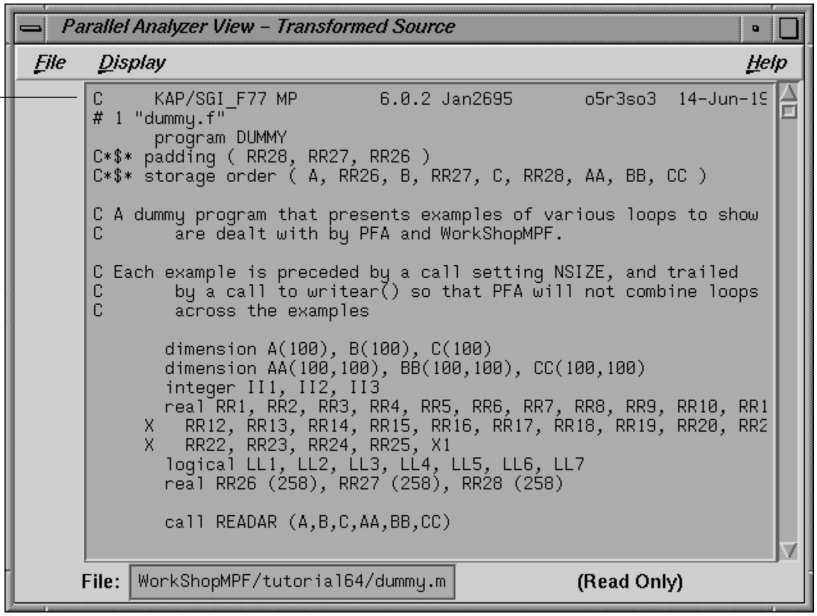


Figure 3-10 Transformed Source Window

Position it below the Source View. Scroll through it, and notice that it, too, has bracketing marking the loops. The bracketing for the transformed source cannot always distinguish between serial loops and unparallelizable loops, so some unparallelizable loops will be displayed as serial (for example, those with data dependencies).

Viewing Detailed Information about a Loop

Each line in the loop list summarizes some information about a loop. Much more information is available, and this section will show you how to examine it.

Selecting a Loop

To get more information about a loop, you must select it by

- double-clicking the loop line text (but not on its icon)
- clicking the brackets in either of the source windows
- stepping through the list with the *Next Loop* and *Previous Loop* buttons

Selecting a loop has a number of effects:

- The previously empty display below the list fills with information on the selected loop.
- The Source View scrolls to the selected loop and highlights the source code of the loop.
- The Transformed Source window highlights the first of the loops into which the original selected loop was transformed and displays a bright vertical bar next to each transformed loop that came from the original loop.

If the Transformed Loops View or the PFA Analysis Parameters View is open, it too will be switched to show the selected loop. We will look at these views later. See Figure 3-11.

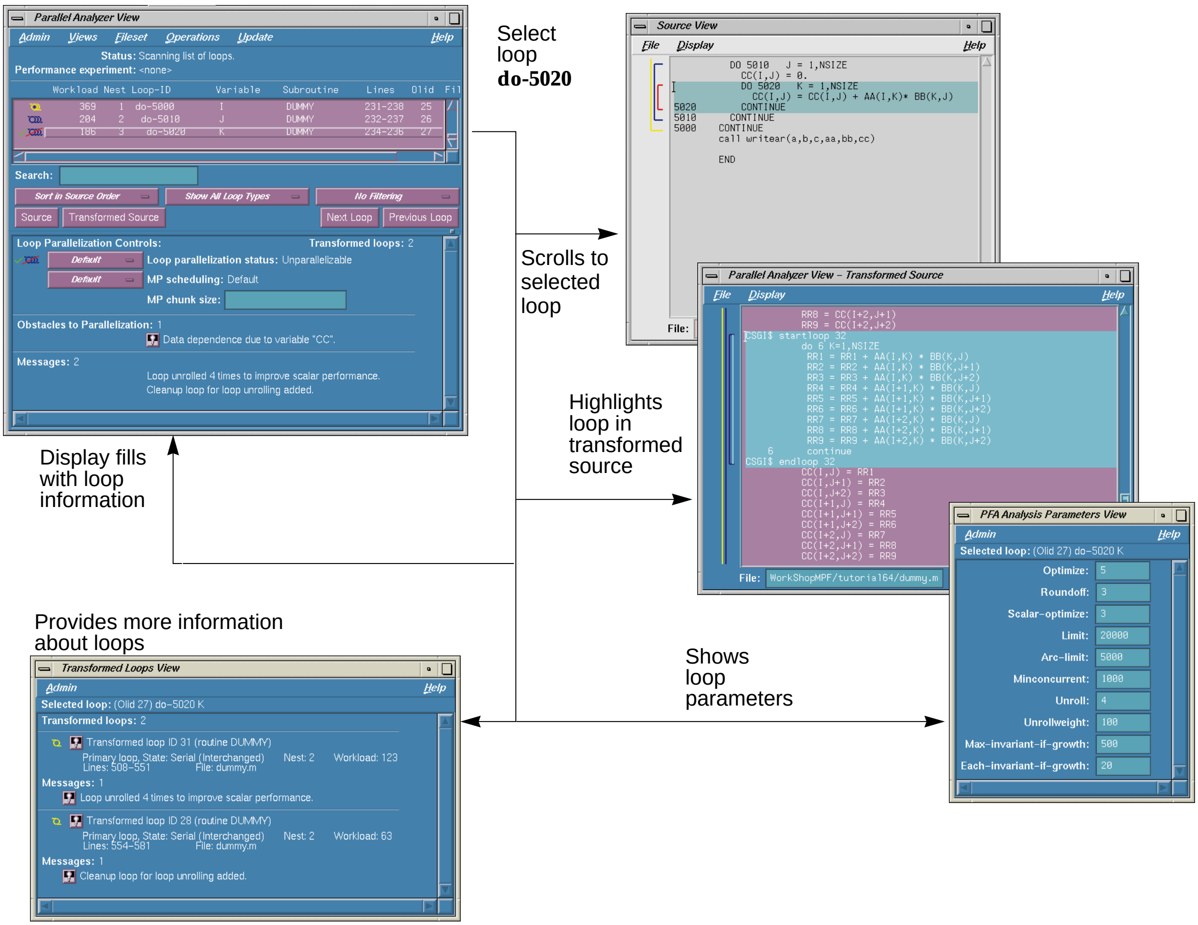


Figure 3-11 Global Effects of Selecting a Loop

In this figure and many of those following, the loop list is resized to reduce the number of loops displayed. The adjustment button is in the lower right hand corner of the loop list display, just above the loop information display. Your screen shows the full list unless you resize it.

Try scrolling through the list and double-clicking various loops, and scrolling through the source displays and clicking the loop brackets to select loops. Note that when you select each loop, its icon acquires a check mark showing that you've looked at it. When you are done, scroll to the top of the loop list in the main view and double-click the first loop's line.

Using the Loop Information Display

The loop information display occupies the lower half of the main view (see Figure 3-12). It contains detailed information about the currently selected loop. It consists of a series of lines in several blocks.

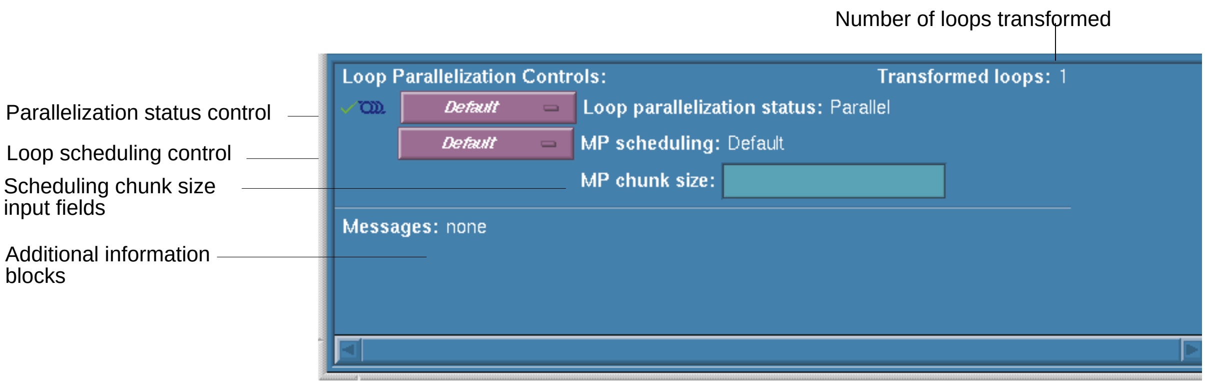


Figure 3-12 Loop Information Display

Parallelization Controls

The first line of the display is labeled Parallelization Controls:. On the far right, the first line shows how many transformed loops were derived from the selected loop. When the session is run with a performance experiment, an additional block appears above the Parallelization Controls. It gives performance information for the loop (shown in Figure 3-39). Since we do not have an experiment on this program (which does not, in fact, execute), the performance information is absent.

Below this are two option menus, the first controlling parallelization status and the second controlling the loop MP scheduling (it is shown for all loops, but is applicable to parallel loops only), and a text input field for adding an expression for the scheduling chunk size. Text labels to the right of the option menus list the current values for parallelization and scheduling.

Loop Information Messages

Below the first separator line appear up to five blocks of additional information. These are lists of:

- questions that PFA asked about the loops, if any
- obstacles to parallelization, if any
- assertions made about the loop, if any
- directives applied to the loops, if any
- messages about the loop, if any

Figure 3-13 Highlighting Button

Some of these lines may be accompanied by small “light bulb” highlighting buttons (see Figure 3-13). Each highlights a relevant part of the code in the Source View when clicked. The lines for assertions, directives, and questions also may have menus accompanying them. Lines that refer to parallelization status or PFA parameters will not have menus because they are controlled using the parallelization status menu or from the PFA Analysis Parameters View, respectively. You’ll use these features later in the session. The first loop in the file (which you selected previously) has no messages and no highlighting buttons.

Using the PFA Analysis Parameters View

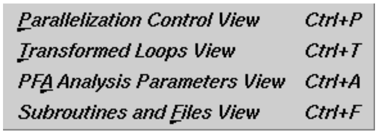


Figure 3-14 Views Menu

The PFA analysis parameters control what kinds of transformations PFA will make on the program. The values for the selected loop may be changed using the PFA Analysis Parameters View. To bring it up, pull down the Views menu and select “PFA Analysis Parameters View” (see Figure 3-14). Alternatively, you may use the keyboard accelerator for this operation by typing **`Ctrl-A`** with the cursor anywhere in the main view.

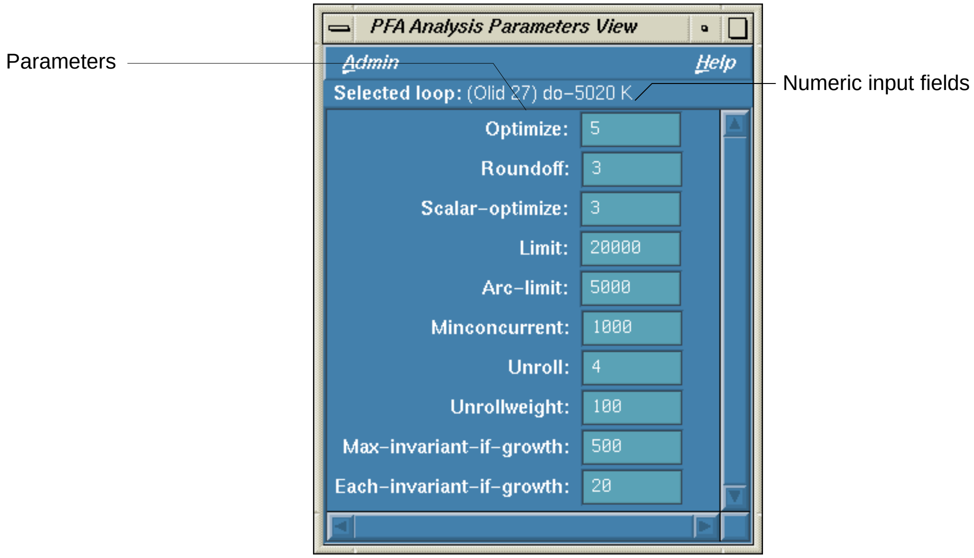


Figure 3-15 PFA Analysis Parameters View

A new view comes up, listing each of the parameters with a numeric input field to the right of each of them. Entering a new numeric value in the input field will request a change to the loop. Don't do this now; close the view by pulling down the View's Admin menu and selecting "Close."

Using the Transformed Loops View

You can also see detailed information about the transformed loops coming from a particular loop (see Figure 3-16). To do so, pull down the Views menu and select "Transformed Loops View." Alternatively, you may use the keyboard accelerator for this operation by typing **`Ctrl-T`** with the cursor anywhere in the main view.

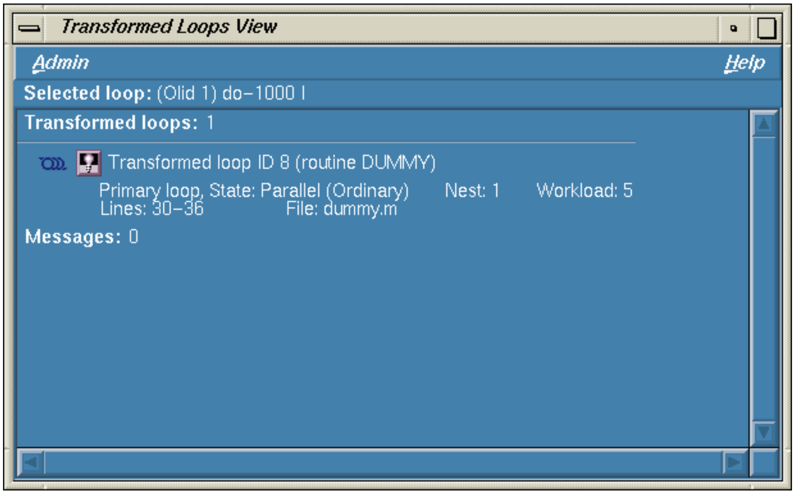


Figure 3-16 Transformed Loops View for Loop **do-1000**

When the view opens, position it at the left of the screen, below the main view. It 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, and the blocks are separated by horizontal lines.

Transformed Loop Description

The first line in each block contains a parallelization status icon, a highlighting button, and the ID of the transformed loop. (The ID is assigned by PFA.) The button, if clicked, highlights the transformed loop in the Transformed Source window and the original loop in the Source View.

The next two lines describe the transformed loop. The first provides information such as whether it is a primary loop (directly transformed from the selected original loop) or secondary loop (transformed from a different original loop but incorporating some code from the selected original loop), its parallelization state, whether it is an ordinary loop or interchanged loop, its nesting level, and workload. The second line displays the location of the loop in the transformed source.

Following the description lines is a list of messages generated by PFA, if any. To the left of the message lines are buttons, and clicking them will highlight the part of the original source that relates to the message. Often it is the first line of the original loop that is shown, since the message refers to the entire loop.

Selecting Transformed Loops

Transformed loops can also be selected. When you click the highlight button in the Transformed Loop View, the color highlighting of the original source changes, although the lines highlighted have not. See Figure 3-17. You will later see that for loops with more extensive transformations the highlighted lines will be different (for example, loops **do-1300** and **do-1350**, the fused loops).

Now click the button for the second transformed loop. The transformed source will highlight a different region (the cleanup loop), but the original source will highlight the same lines as before, as shown in Figure 3-18. This is because when a transformed loop is selected, those lines in the original source that go into the transformed loop will be highlighted. In this case, the same lines go into both the transformed loops. Transformed loops may also be selected by clicking the corresponding loop brackets in the Transformed Source window.

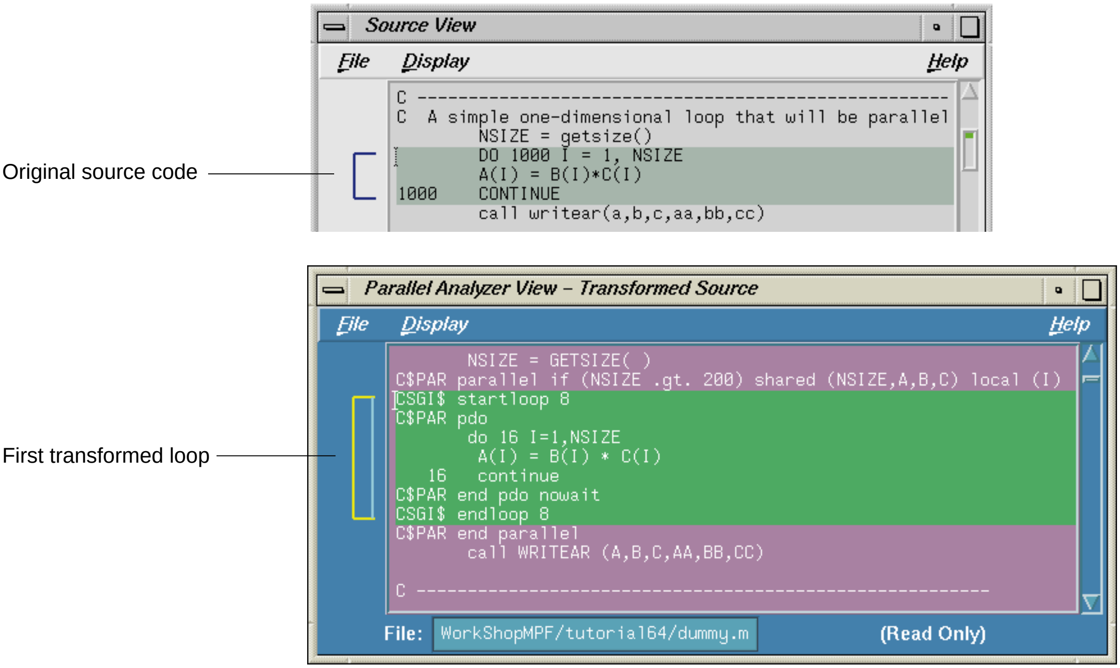


Figure 3-17 Transformed Loops in Source Windows

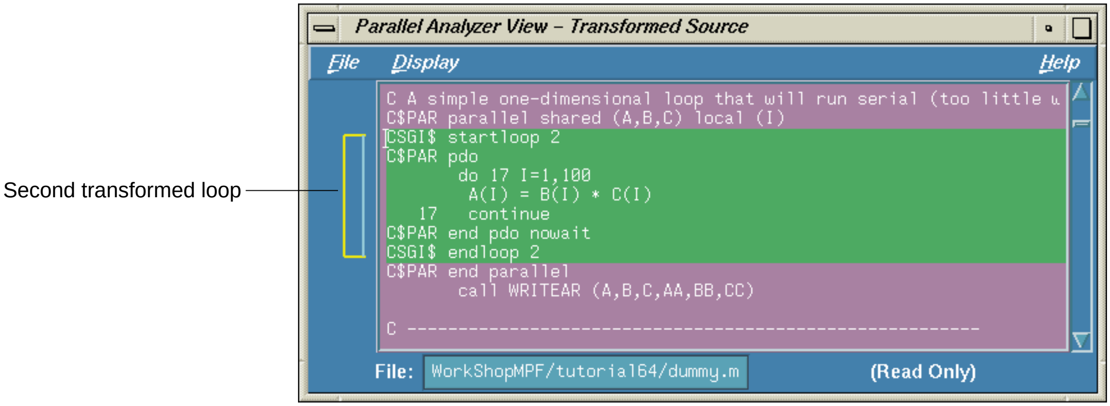


Figure 3-18 Second Transformed Loop Highlighting

You may either leave this window open or close it by pulling down its File menu and selecting the “Close.”

Examining Loops

Now that you have familiarized yourself with the basic windows in the Parallel Analyzer View’s user interface, you can start examining and analyzing loops. First you will look at a few simple loops, next at loops with obstacles to parallelization, then at loops for which PFA asks questions, and finally at more complex, nested loops.

Simple Loops

The six loops you will examine in this section are the simplest kind of Fortran loop.

Simple Parallel Loops

Scroll the list of loops back to the top and select loop **do-1000**. This loop is a simple parallel loop. Loop **do-1100** is also a simple parallel loop.

Move to loop **do-1200** by clicking the *Next Loop* button twice.

An Explicitly Parallelized Loop

Loop **do-1200** is parallelized because it contains an explicit **C$DOACROSS** directive; PFA will pass the directive through in the transformed source but does nothing further with the loop, as the messages indicate. See Figure 3-19.

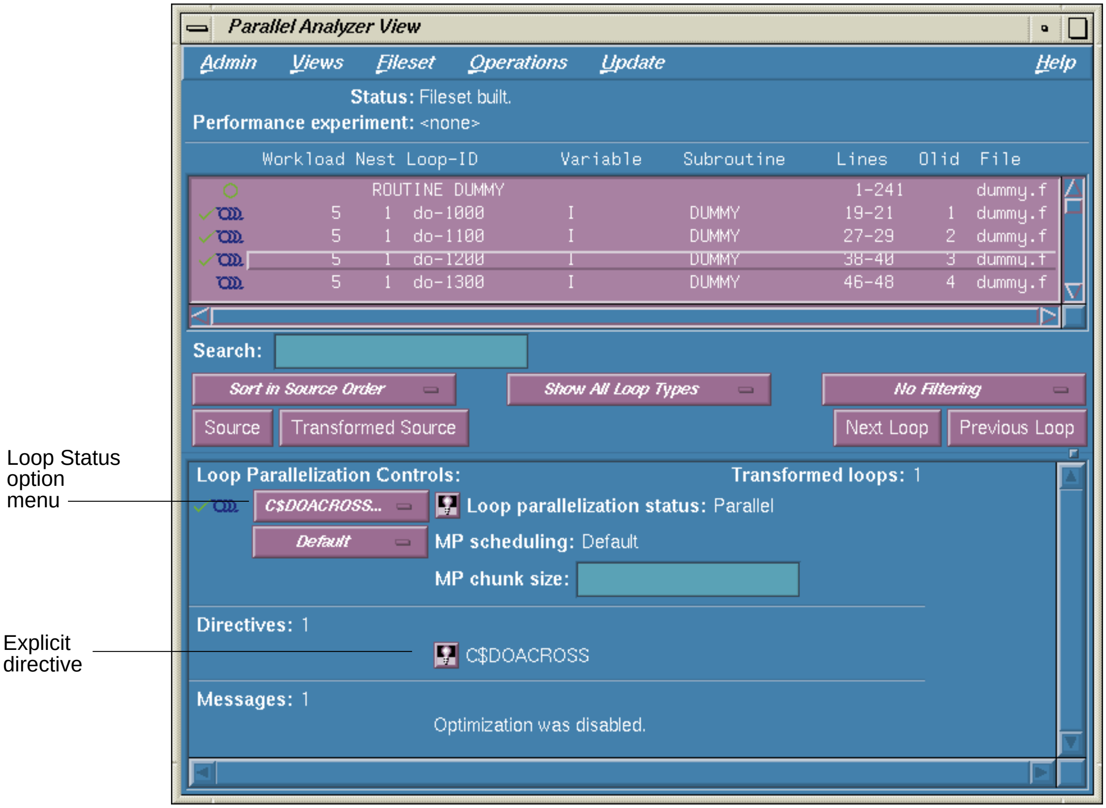


Figure 3-19 Explicitly Parallelized Loop

The loop status option menu is set to “C$DOACROSS...”and it is shown with a highlighting button. Clicking the button will bring up both the Source View (Figure 3-20) and the Parallelization Control View, which shows more information about the parallelization directive. If you have clicked on the button, close the Parallelization Control View by pulling down its Admin menu and selecting “Close.” You will come back to the use of this view later. See “Building a Custom DOACROSS Directive”. Close the Source View by pulling down its File menu and selecting “Close.”

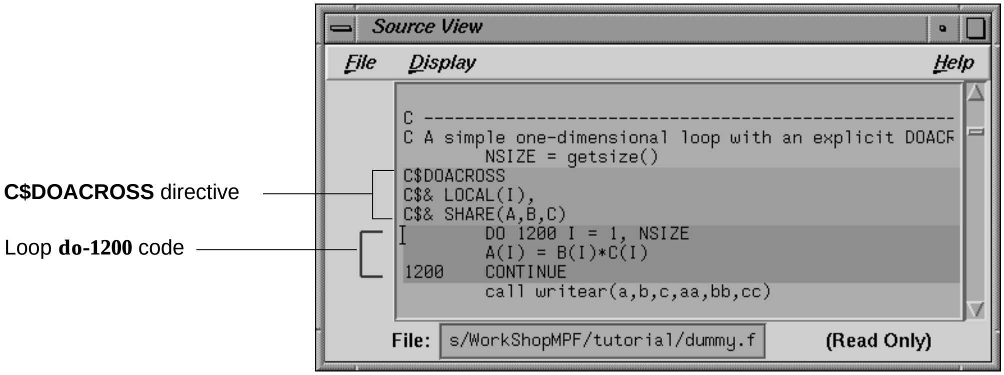


Figure 3-20 Source View of C$DOACROSS Directive

Loops **do-1300** and **do-1350** are simple parallel loops. Move to loop **do-1400** by clicking the *Next Loop* button three times.

Loop Unrolling

Unrolling is done to reduce the loop overhead relative to the real work of the loop. The simpler the body of the loop, the more profitable unrolling can be. In many cases, the loop iteration count is not known, so an additional loop, called a cleanup loop, is necessary to handle the last few iterations. Sometimes, the iteration count is known but is not a multiple of the unrolling; in such cases, PFA will usually explicitly add code for the last few iterations.

Loop **do-1400** is the same as the first loop in the program, but a directive "SCALAR OPTIMIZE(1)" has been added. The loop is not unrolled. By default, the scalar optimization parameter is set to 3, which allows loop unrolling.

Move to loop **do-1500** by clicking the *Next Loop* button.

A Loop That Is Optimized Away

Loop **do-1500** is an example of a loop so unnecessary that PFA can get rid of it entirely. First, PFA sees that the body of the loop is independent of the loop, so it can be promoted out, and the loop eliminated. Then it sees that the body sets a variable that is not subsequently used, so it can throw that out, too. The transformed source is not scrolled and highlighted because nothing is there. Scroll down a few lines from the previous loop, and note the absence of the code for the loop that was optimized away.

The loop also has a directive controlling scalar optimization, but it is there only to reset the default for subsequent loops.

Move to loop **do-2000** by clicking the *Next Loop* button.

Loops with Obstacles to Parallelization

There are a number of reasons that a loop may not be parallelized. The following loops illustrate various of these reasons, along with variants that allow parallelization. You will step through each of them in turn.

Loops with Data Dependences

Loop **do-2000** is an example of a loop that cannot be parallelized because of a data dependence. In this case, one element of an array is used to set another. (This construct is called a recurrence.) If the loop were to be parallelized, the iterations might execute out of order, and iteration 4, which sets A(4) to A(5), might occur after iteration 5, which would have reset the value of A(5). Consequently, the program would give the wrong answer. See Figure 3-21.

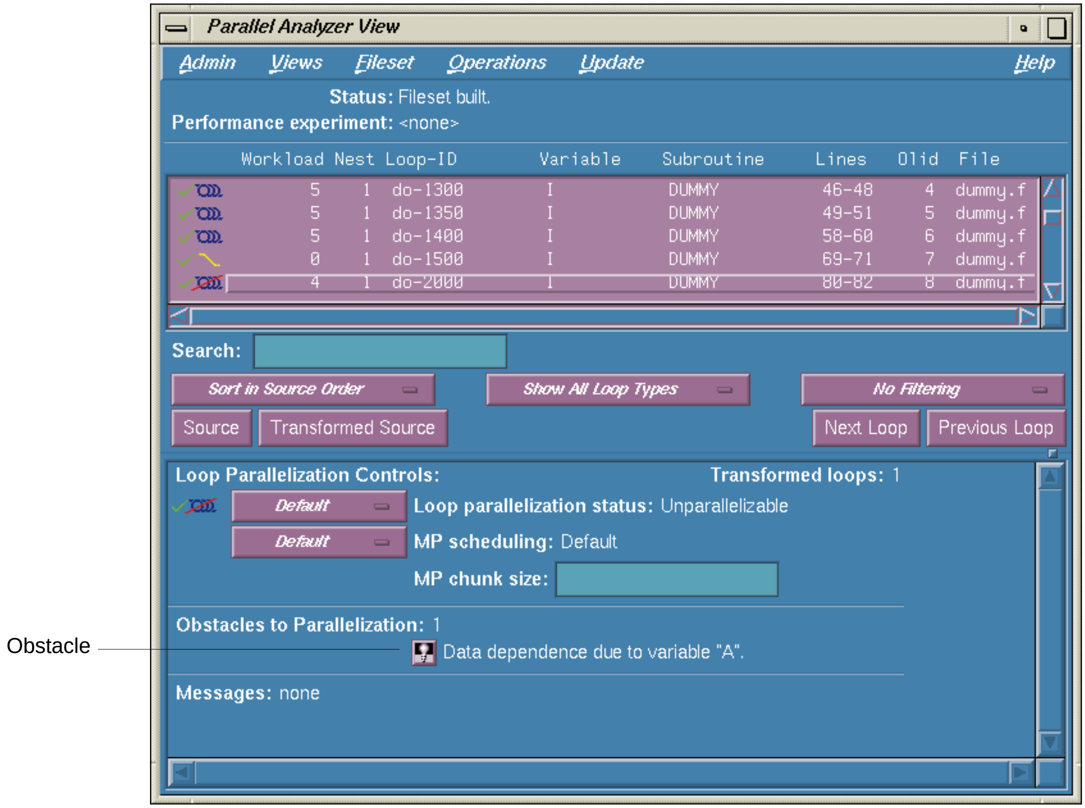


Figure 3-21 Obstacle to Parallelization

There is a line listing the obstacle to parallelization; click the button that accompanies it. Two kinds of highlighting take place. The first is a line highlight showing the relevant line that has the dependence, and the second is a symbol (or token) highlight that shows the uses of the variable that is the obstacle to parallelization. Only the uses of the variable within the loop are highlighted.

Move to loop **do-2100** by clicking the *Next Loop* button.

Not all loops with similar constructs are unparallelizable. Loop **do-2100** is similar to loop **do-2000**, but the array elements used differ by an offset, M. If M is equal to NSIZE, for example, and the array is twice NSIZE, the code is actually copying the upper half of the array into the lower half, and there is no reason why that cannot be run in parallel. PFA cannot recognize this from the source, but the author has added an assertion that there is no recurrence, so the loop is parallelized. See Figure 3-22. Click the highlighting button to show the assertion.

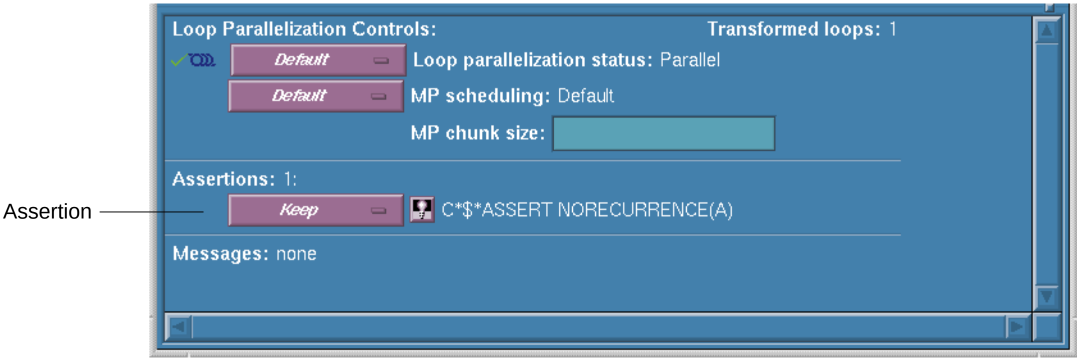


Figure 3-22 Parallelizable Data Dependence

Move to loop **do-2200** by clicking the *Next Loop* button.

Data dependence can involve more than one line of a program. In loop **do-2200**, a similar dependence occurs, but the use of the variable occurs on a different line than its setting. Click the highlight button on the obstacle line, and note that both lines receive the line highlighting, and the token highlighting shows the dependency variable on the two lines (see Figure 3-23). Of course, real programs can, and typically do, have far more complex dependencies than this.

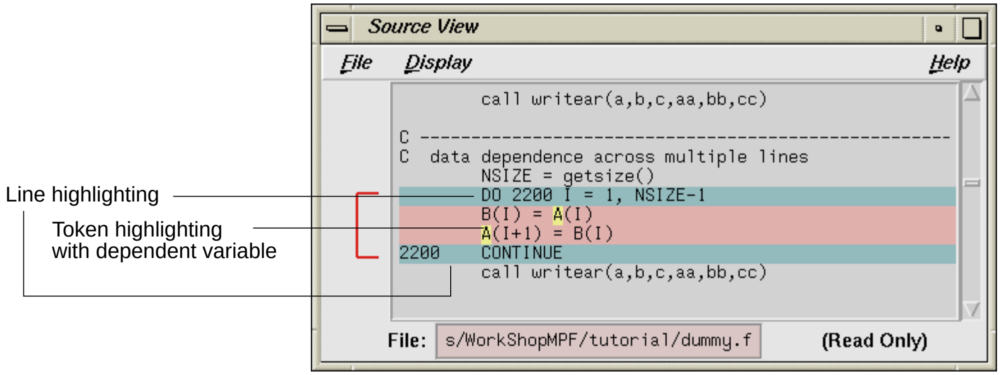


Figure 3-23 Highlighting on Multiple Lines

Move to loop **do-2300** by clicking the *Next Loop* button.

Loops with Reductions

Loop **do-2300** shows a data dependence that is called a *reduction*. In a reduction, the variable responsible for the data dependence is being accumulated or "reduced" in some fashion. Reductions can be summation, multiplication, or a minimum or maximum determination. For summation, as shown in this loop, PFA could accumulate partial sums in each processor, and then add the partial sums at the end. However, because floating-point arithmetic is inexact, the order of addition might give different answers because of round-off error.

This does not imply that the serial execution answer is "correct" and the parallel execution answer is "incorrect"; they are equally valid within the limits of round-off error. Since, by default, PFA assumes it is not OK to introduce round-off error, the loop is left serial. PFA does, however, have a parameter to allow you to say that such round-off error is OK.

Move to loop **do-2400** by clicking the *Next Loop* button.

In loop **do-2400**, the author has added a directive controlling round-off error. The same loop that was left serial above is now parallelized. Click the button for the directive, and you can see how it is highlighted in the source. Refer to the PFA manual for a more detailed explanation of the meaning and use of this directive. The round-off setting will be left at this value for the remainder of the program.

Move to loop **do-2500** by clicking the *Next Loop* button.

Loops with Input-output Operations

Loop **do-2500** has an input/output (I/O) operation in it. It cannot be parallelized, because the output would appear in a different order depending on the scheduling of the individual CPUs. Click the button indicating the obstacle, and note the highlighting of the print statement. Also note that the transformed source shows that this loop is not unrolled, either. Actually, there is no real obstacle to unrolling, but it is not done because the cost of performing the I/O operation is so great compared to the loop iteration overhead that the savings gained are not worth the increase in the size of the program.

Move to loop **do-2600** by clicking the *Next Loop* button.

Loops with Premature Exits

Loop **do-2600** has a premature exit; that is, it cannot be determined at compilation time how many iterations will take place. If PFA did parallelize it, one thread might execute iterations past the point where another has determined to exit the loop.

Click the button indicating the premature exit. Note that the line with the exit from the loop is highlighted in the source.

Move to loop **do-2700** by clicking the *Next Loop* button.

Loops with Subroutine Calls

Loop **do-2700** is also unparallelizable, because there is a call to a routine, *RTC*, and PFA cannot determine whether or not that call will have side effects. Click the obstacle line. Note the highlighting of the line containing the call and the subroutine name. Also note that the loop is not unrolled, as the presence of the call inhibits unrolling.

Move to loop **do-2800** by clicking the *Next Loop* button.

Although loop **do-2800** has a similar subroutine call in it, it can be parallelized because the author has asserted that the call has no side effects that will prevent it from running concurrently. Click the assertion line to highlight the source line containing the assertion.

When you are done, move to loop **do-3000** by clicking the *Next Loop* button.

Loops That Prompt Questions from PFA

Sometimes PFA can parallelize a loop more efficiently if it knows more information than it can infer from the source. In these cases, PFA asks questions that appear in the loop information display for the loop, along with a menu that allows you to answer the question.

Loops with Relationships between Variables

PFA can sometimes parallelize a loop if it can be told the relationship between variables in the program. Although you may know such relationships from the nature of the physical problem the program is dealing with, PFA cannot safely infer the information just from the code.

Loop **do-3000** can be parallelized if it is known that the iterations do not overlap, but not otherwise. PFA will ask three questions, although for this type of construct, it actually generates code to determine the relationship at run time, and the program will execute one of the two sequences depending on that determination. You can see this by observing that the loop was transformed into four loops, one pair of unroll / cleanup loops when it can be parallelized, and a second when it cannot. Look at the transformed source code for each of these pairs.

For any such questions, the line asking them has an associated option menu that will allow you to answer. The generated code will be correct even if you do not answer or do not know. If PFA knows the answer, it can omit the alternate form and produce a tighter program.

Move to loop **do-3100** by clicking the *Next Loop* button.

In loop **do-3100**, the author has added an assertion answering the question, and PFA has generated just one version of the loop, the one that runs in parallel. The menu next to the questions for the previous loop will generate such an assertion.

Move to loop **do-3200** by clicking the *Next Loop* button.

Permutation Vectors

Loop **do-3200** has a construct known as a permutation vector. In it, an array is referenced by an index value contained in another array. If the B(I) values are all distinct, the iterations do not depend on each other, and the loop can be parallelized; if the same value occurs in more than one B(I), it cannot. PFA asks the question but leaves the loop serial. Note that both the question and the data dependence message have associated highlighting buttons.

Move to loop **do-3300** by clicking the *Next Loop* button.

Here an assertion has been added that the index array, B(I), is indeed a permutation vector, and the loop is parallelized.

Move to loop **do-4000** by clicking the *Next Loop* button.

Complex Loops and Loop Nests

Finally, let's look at somewhat more complicated, nested loops.

Doubly-nested Loops and Interchanges

Loop **do-4000** is the outer loop of a pair of loops; it runs in parallel, and the inner loop runs in serial: one parallel loop cannot be nested inside another. Also note that the outer loop is not unrolled, but the inner loop is.

Move to loop **do-4010** by clicking the *Next Loop* button to show the inner loop, and then click *Next Loop* again to select the outer loop of the next pair.

Note that this outer loop, loop **do-4100**, is shown as serial inside a parallel loop, and the following loop is parallel. How can this be? It happens because PFA has recognized that the two loops can be interchanged, and furthermore, that the CPU cache is likely to be more efficiently used if the loops are run in the interchanged order.

Move to loop **do-4110** to show the inner loop, and then click the *Next Loop* button once again to move to the following triply-nested loop.

Modifying Source Files

So far, you've ignored the controls that can be used to change the source file and allow a subsequent pass of PFA to do a better job. Now you will go back and make changes. There are two steps in modifying source files:

1. Asking for the changes using the Parallel Analyzer View controls.
2. Actually modifying the files and rebuilding the program and its analysis files.

Asking for Changes

You may ask for changes by answering any of the questions that PFA poses, by building a DOACROSS for a specific loop, by modifying the analysis parameters that PFA uses for its processing, or by adding or deleting assertions or directives. In this sample session, you will request changes to loops in the order they appear in the file, but they may be requested in any order.

Building a Custom DOACROSS Directive

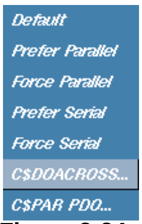


Figure 3-24 DOACROSS Menu

Loop **do-5000** is serial nested inside a parallel loop. If you wanted to change it to parallel, you would go to the Loop Status option menu (to the left of the loop status icon in the loop information display that reads “Default”), and select “C$DOACROSS...” as shown in Figure 3-24. This brings up the Parallelization Control View (see Figure 3-25), showing the loop that was selected, a parallelized condition input field into which you can type a condition for parallelization, an MP scheduling option menu, an MP chunk size input field, and a list of all the variables in the loop, with an icon indicating whether the variable was read, written, or both. (These icons are described in the Icon Legend.)

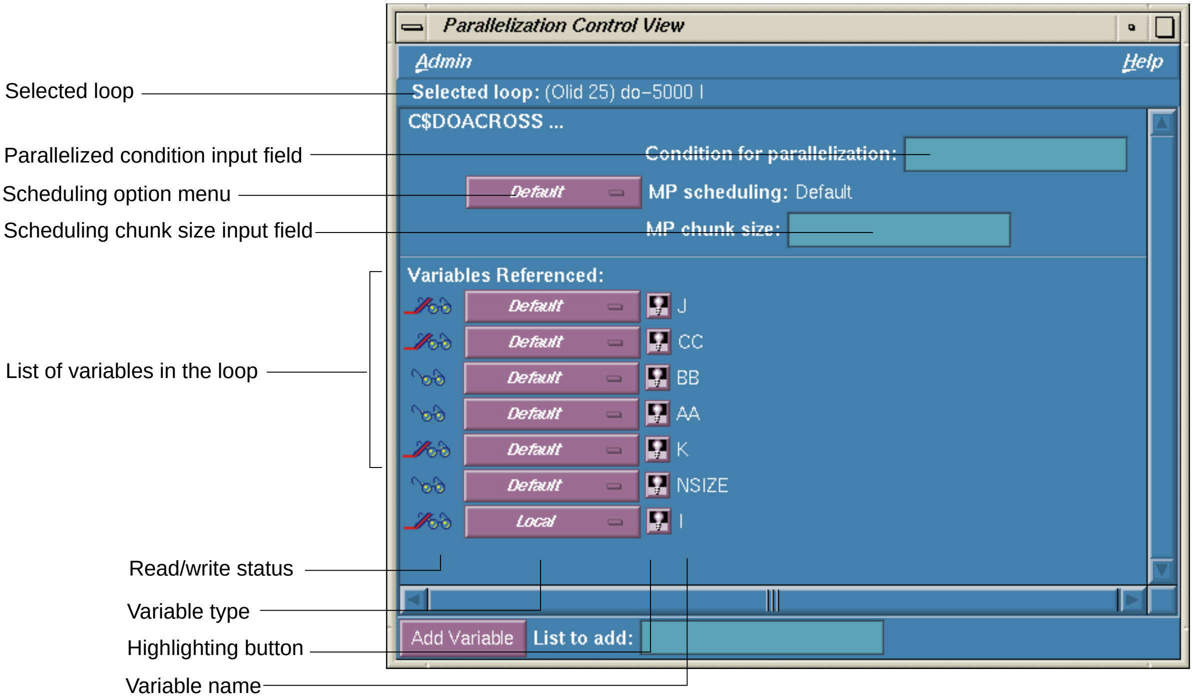


Figure 3-25 Parallelization Control View for Loop **do-5000**

Notice that each variable has a highlighting button that shows its use within the loop.Notice also the red plus sign next to this loop in the main view, indicating that a change has been requested for it as shown in Figure 3-26.

Close the View by pulling down the Admin menu and selecting “Close.” Move to loop **do-1100** by clicking the *Next Loop* button.

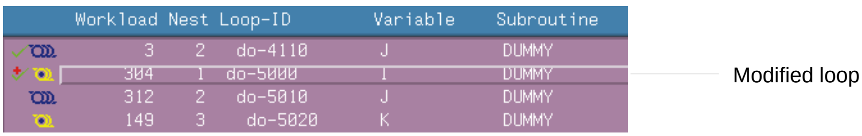


Figure 3-26 Effect of Changes on the Loop List Display

Adding a New Assertion

Now you will add an assertion to a loop. Find the loop with ID **do-2700** by using the search feature of the loop list. Go to the search field, and enter `2700`. Double-click the highlighted line in the loop list to select the loop.

You're going to add a concurrent call assertion. To add the assertion,pull down the Operations menu, pull down the Add Assertion submenu, and select "C*$*ASSERT CONCURRENT CALL."

This adds an assertion that the call to *RTC*(), which PFA thought to be an obstacle to parallelization, is actually safe to parallelize. When you add the assertion, the loop information display updates to show the new assertion, along with its menu labeled "Insert" as shown in Figure 3-27.

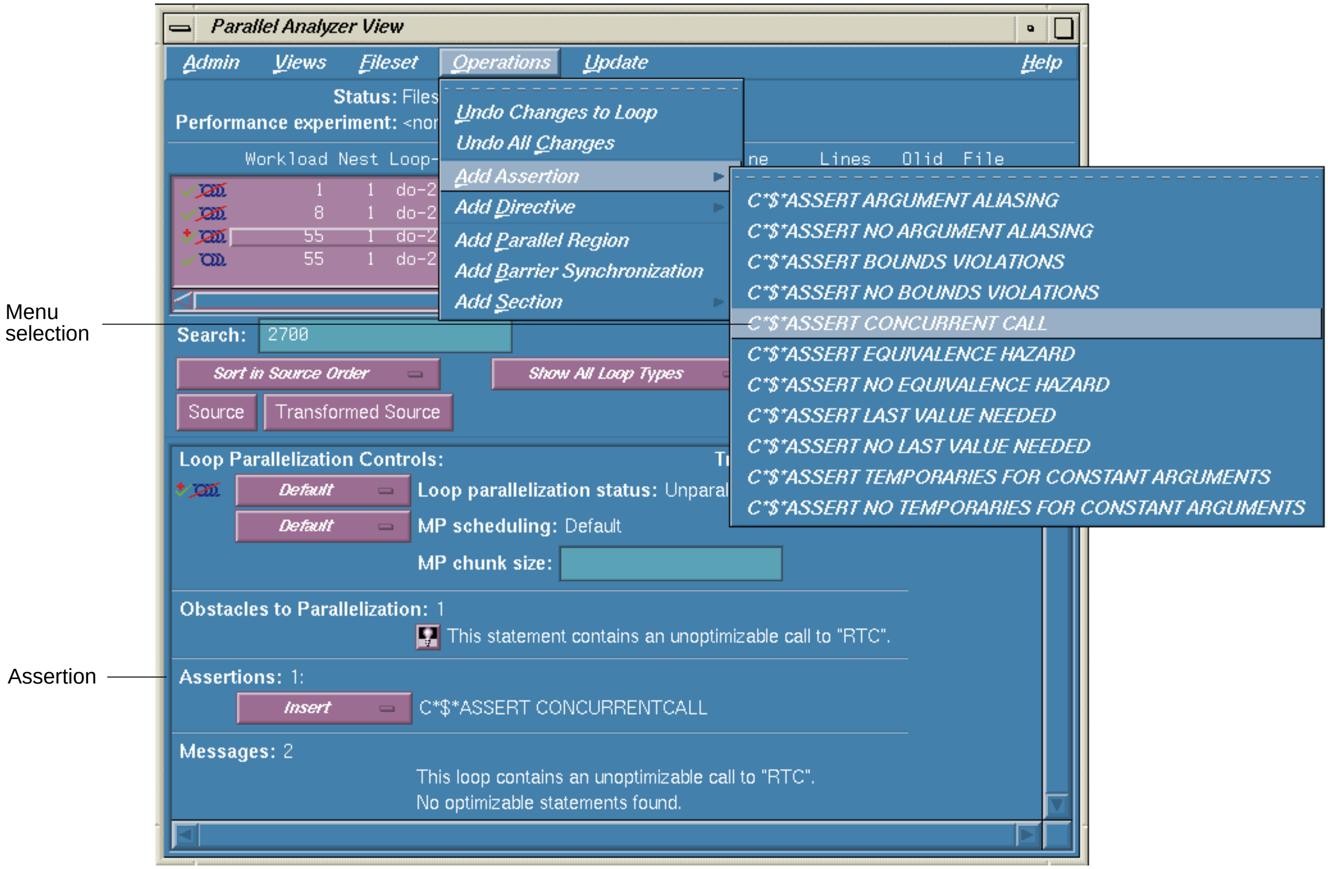


Figure 3-27 Adding an Assertion

Answering a Question

Now try answering a question. Put the cursor into the search field, backspace to remove the previous contents, and enter **`3200`** into the field. Select that loop by double-clicking. Loop **do-3200** has a question about a permutation vector. Pull down the option menu next to the question in the loop information display, and select “Assert True” as shown in Figure 3-28.

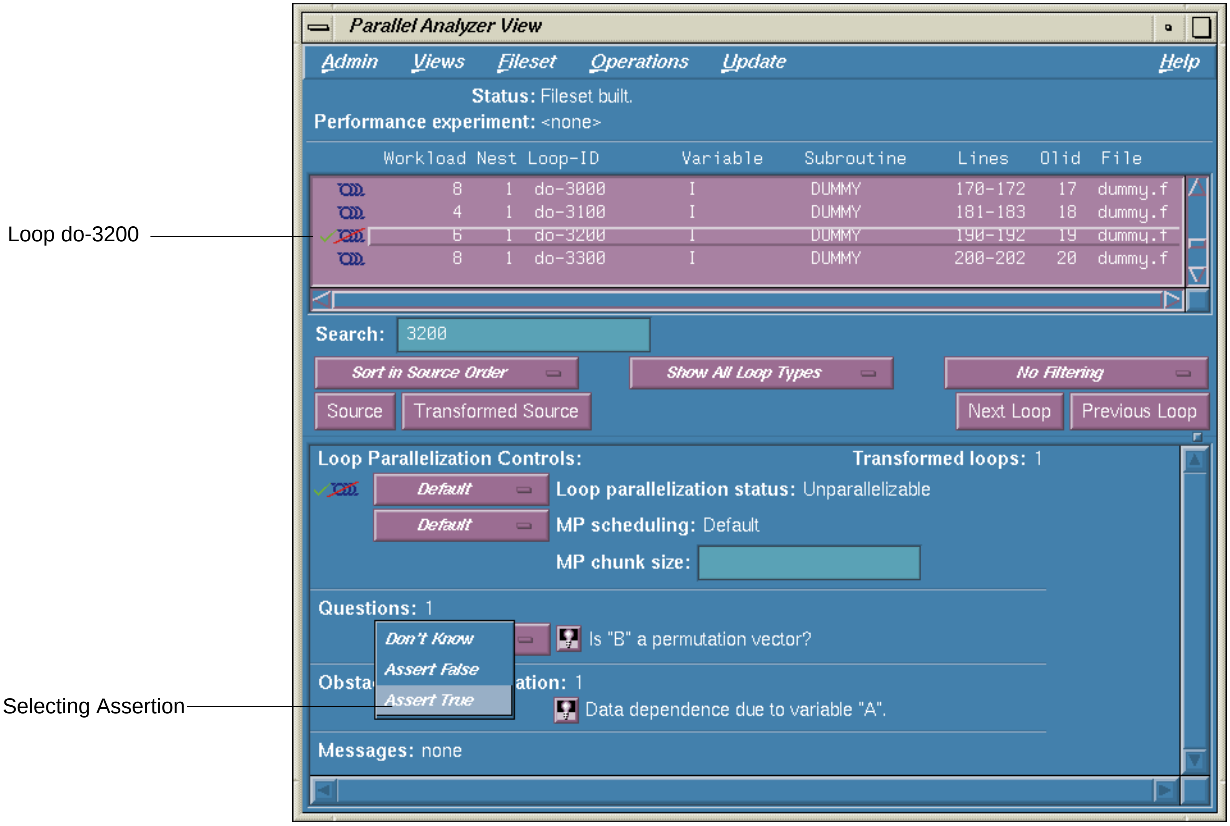


Figure 3-28 Answering a Question

Deleting an Existing Assertion

Now let's delete an existing assertion. Move to loop **do-3300** using the *Next Loop* button, and go to the "ASSERT PERMUTATION(B)" assertion. Pull down its option menu and select "Delete". Figure 3-29 shows the result. The same procedure can be used for directives.

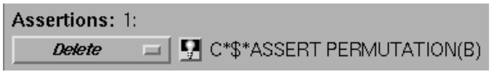


Figure 3-29 Deleting an Assertion

Updating the File

Now you have made a set of changes and can update the file. Select “Update All Files” from the Update menu (see Figure 3-30); alternatively, you may use the keyboard accelerator for this operation by typing **`Ctrl-U`** with the cursor anywhere in the main view. The Parallel Analyzer View will generate a *sed* script to modify the source, rename the original file to one with the suffix *.old*, run *sed* on that file to produce a new version of the file *dummy.f*, and then spawn the WorkShop Build Manager to rerun PFA on the new version of the file.

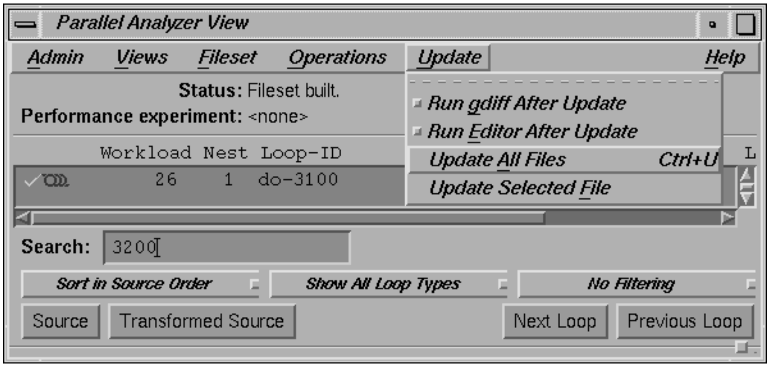


Figure 3-30 Update All Files

The Parallel Analyzer View can also open a *gdiff* window showing the changes, but by default it does not. If you select the toggle labeled “Run gdiff After Update” from the Update menu, it will do so. If you have selected it, use the right mouse button to step through the changes, and then quit *gdiff*. If you always wish to see the *gdiff* window, you can set the resource in your *.Xdefaults* file:

```
cvpav*gDiff: True
```

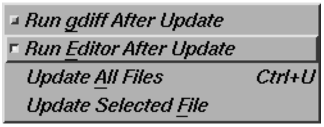


Figure 3-31 Setting the Run Editor Toggle

The Parallel Analyzer View can also open an editor for you to make additional changes after running *sed*. To do so, select the toggle labeled "Run Editor After Update" in the Update menu (see Figure 3-31). If you do so, an *xwsh* window with *vi* running in it opens after you update the 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 change the resource in your *.Xdefaults* file, changing the *xwsh* and / or *vi* as you prefer:

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

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 will either replace any explicit **%s**, or if the **%s** is omitted, the file name will be appended to the command.

After you quit from the *gdiff* window and / or editor (if you have selected them), the program will spawn the WorkShop Build Manager. When it comes up, verify that the directory shown is the directory in which you are running the sample session; if not, change it. Then, click the *Build* button, and it will start to reprocess the changed file.

Examining the Modified File

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

New Assertion

Go to the search field and enter **2700**. Double-click the line and notice that loop **do-2700**, which previously was unparallelizable because of the call to *RTC*(), is now parallel. It also has the assertion that was added.

Answered Question

Clear the search field, enter **`3200`** in it, and double-click the selected line. Notice that loop **do-3200** now has an assertion in it, added as a result of your reply to the question. The loop is also now parallelized.

Move to loop **do-3300** by clicking the *Next Loop* button.

Deleted Assertion

Loop **do-3300** previously had the assertion that B was a permutation vector; note that the assertion is gone, and PFA now asks the question.

Now enter **`parallel`** in the the Search field of the main view. This takes you to the first parallel region of **PCFDUMMY**, the second function in *dummy.f*. Double-click that line to begin your examination of the constructs in **PCFDUMMY**.

Examining Subroutines That Use PCF Directives

PCFDUMMY contains four parallel regions, each of which illustrates some of the PCF directives. Click *Next Loop* to go to **do-6001**, the first loop of the first parallel region.

Explicitly Parallelized Loops With C$PAR DO

The first construct in routine **PCFDUMMY** is a parallel region that contains two loops that are explicitly parallelized with **C$PAR PDO** statements. See Figure 3-32.

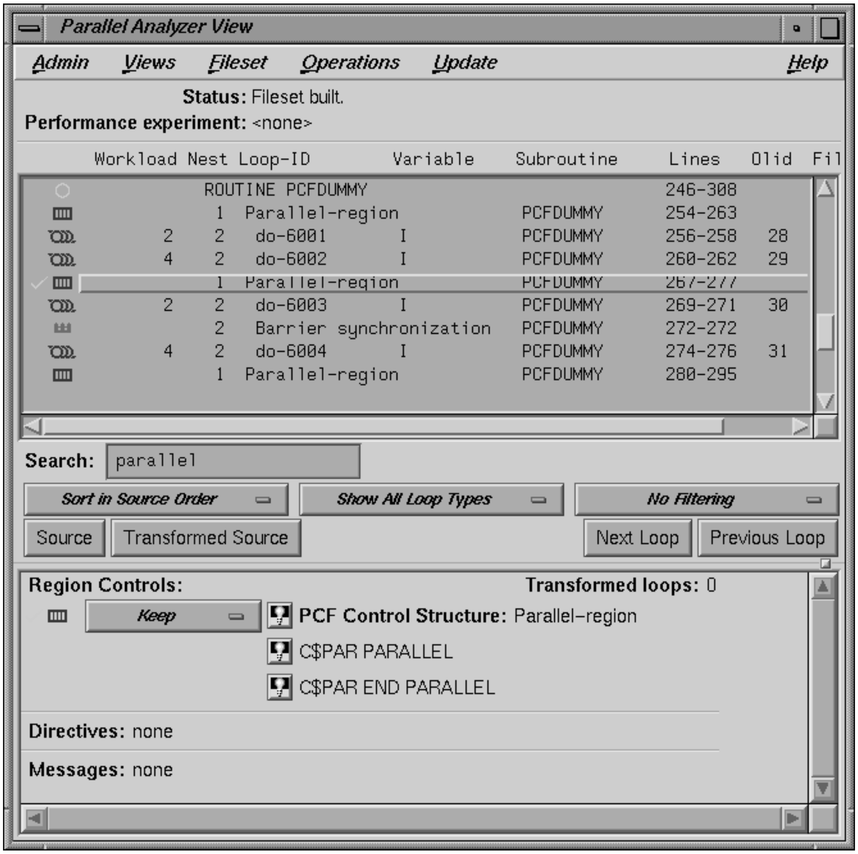


Figure 3-32 Explicitly Parallelized Loops With C$PAR DO

Notice that the parallel region has controls specific to the region as a whole. The “Keep / Delete” option menu and the highlight buttons function the same way they do in the Loop Parallelization Controls.

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

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

Loops With Barriers

The second parallel region contains the same two loops, but in this example there is a barrier between them. Click *Next Loop* twice to view the barrier region. See Figure 3-33.

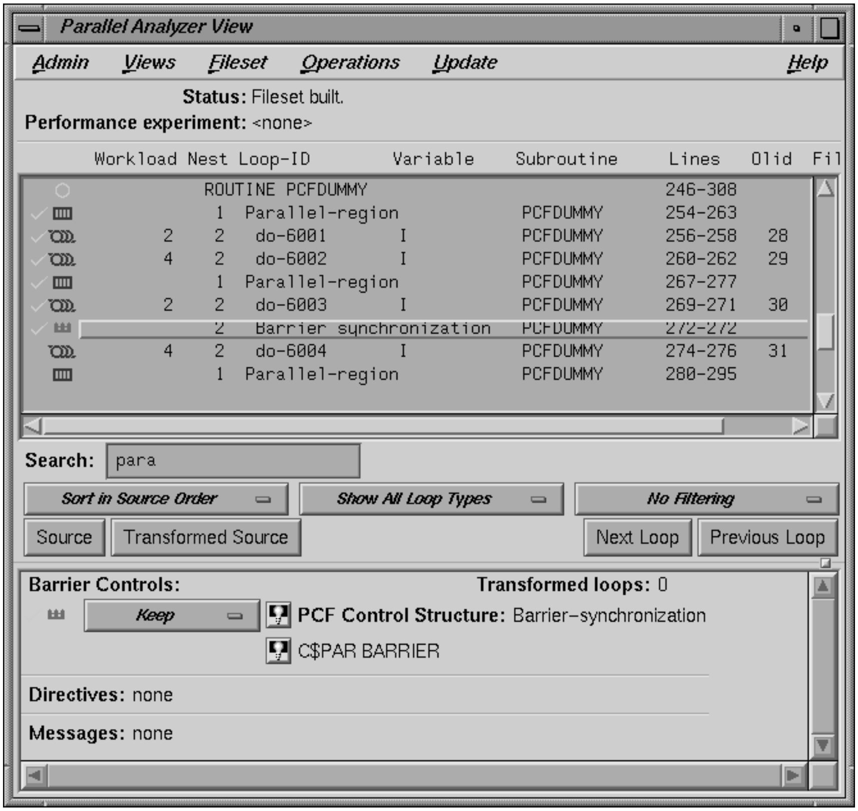


Figure 3-33 Loops With Barrier Synchronization

All iterations of the first **C$PAR DO** must complete before any iteration of the second loop can begin. (In the first set of loops, the second could start before all iterations of the first is completed.)

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

Critical Section in a Loop

The third parallel region contains two loops. Click *Next Loop* to view loop **do-6005**. This loop contains a critical section. Click *Next Loop* to view the critical section. The critical section uses a named locking variable (*S3* in this case) and uses the lock to prevent simultaneous update of *S1* from multiple threads. This is a standard construct for performing a reduction.

Move to loop **do-6006** by clicking the *Next Loop* button.

Loop **do-6006** has a single-process section. It ensures that only one thread will ever execute the statement in the section. Click *Next Loop* to view the single-process section information.

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

Parallel Sections

The fourth and final parallel region of **PCFDUMMY** provides an example of parallel sections. In this case, there are three parallel subsections, each of which calls a function. Each function will be called exactly once, by a single thread. If there are three or more threads in the program, each function will be called from a different thread. Click *Next Loop* to view each parallel section and subsection.

When you are finished, scroll to the end of the program in the main view and double-click the `ROUTINE SYNTAXERR` line.

Examining a Subroutine That Contains Syntax Errors

The **SYNTAXERR** routine contains a number of errors in the source code. During the compilation, the compiler generates error messages for them and flags the routine as having syntax errors.

The compiler provides error messages for four errors that it has detected in compiling this routine. Each of the errors has a message, and a highlighting button to show the error in the source. See Figure 3-34.

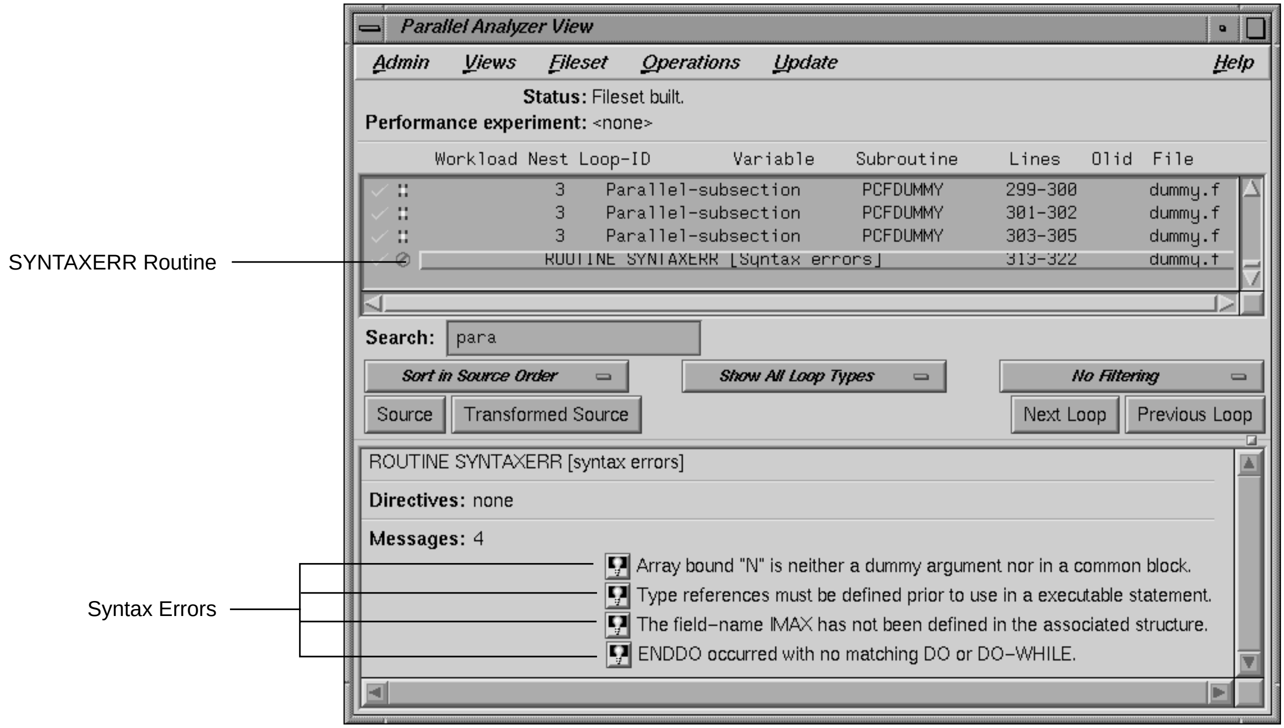


Figure 3-34 Examing Syntax Errors

The four syntax errors are caused by two errors in the source code:

- The second DIMENSION statement uses a variable (*n*) as a dimension. This can only be correct if the variable is in a COMMON block, or an input parameter to the subroutine.
- There is a typo in the **do** statement—a period rather than a comma is used. This causes the compiler to expect an arithmetic statement to assign the value of a structure element to a variable, *doi*. This causes the next three errors: two for the statement itself, and the third for the presence of an **enddo** statement with no **do** statement.

To view or modify the source to correct these errors, click on the appropriate highligh button to bring up the Source View.

Exiting From the Dummy Sample Session

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

To clean up the directory, so that the session can be rerun, enter:

```
% make clean
```

in your shell window. All of the generated files will be removed.

Setting Up the linpackd Sample Session

The second sample session is a brief demonstration of the integration of WorkShopProMPF and the WorkShop performance tools. It requires that WorkShop also be installed.

Go to the subdirectory *linpack.mips4* in the */usr/demos/WorkShopMPF* directory and run *make*:

```
% cd /usr/demos/WorkShopMPF/linpack.mips4
% make
```

This will update the directory by compiling the source program *linpackd.f* and creating the necessary files. The performance experiment you will use is already there. This operation will take a few minutes.

Starting the Parallel Analysis View

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

```
% cvpav -e linpackd
```

from within the directory (note the flag is **-e**, not **-f** as in the previous sample session). The main window of the Parallel Analysis View will open, showing the list of loops in the program.

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

Starting the Performance Analyzer

Start the Performance Analyzer by pulling down the Admin menu, selecting the Launch Tool submenu, and selecting “Performance Analyzer,” as shown in Figure 3-35.

The main window of the Performance Analyzer will open, although it will be empty. A small window labeled “Experiment:” will also open at the same time. This window is used to enter the name of an experiment. For this session, we will use the prerecorded experiment that is installed. Type:

```
test0001
```

in the “Experiment Dir:” field in the Experiment: window, and click the *OK* button. See Figure 3-35. The Performance Analyzer will show a busy cursor, fill in its main window with the list of functions, and highlight the function **main**().

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

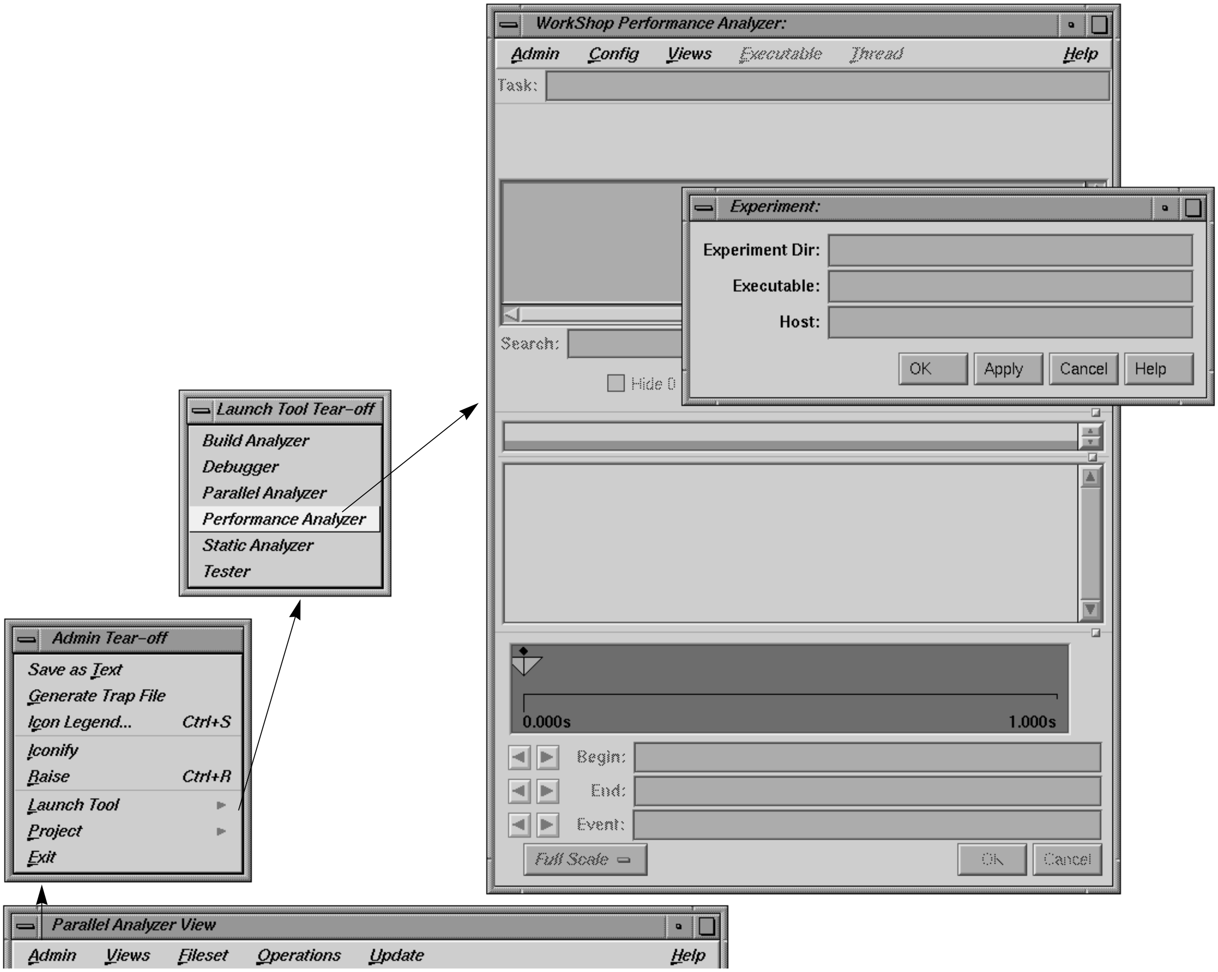


Figure 3-35 Starting the Performance Analyzer

Using the Parallel Analyzer with Performance Data

At the same time the Performance Analyzer window fills in, the Parallel Analyzer recognizes that there is now a performance analyzer, and posts a busy cursor with a message "Loading Performance Data." When the message goes away, performance data will have been imported by the Parallel Analyzer, and a number of changes will have taken place as shown in Figure 3-36:

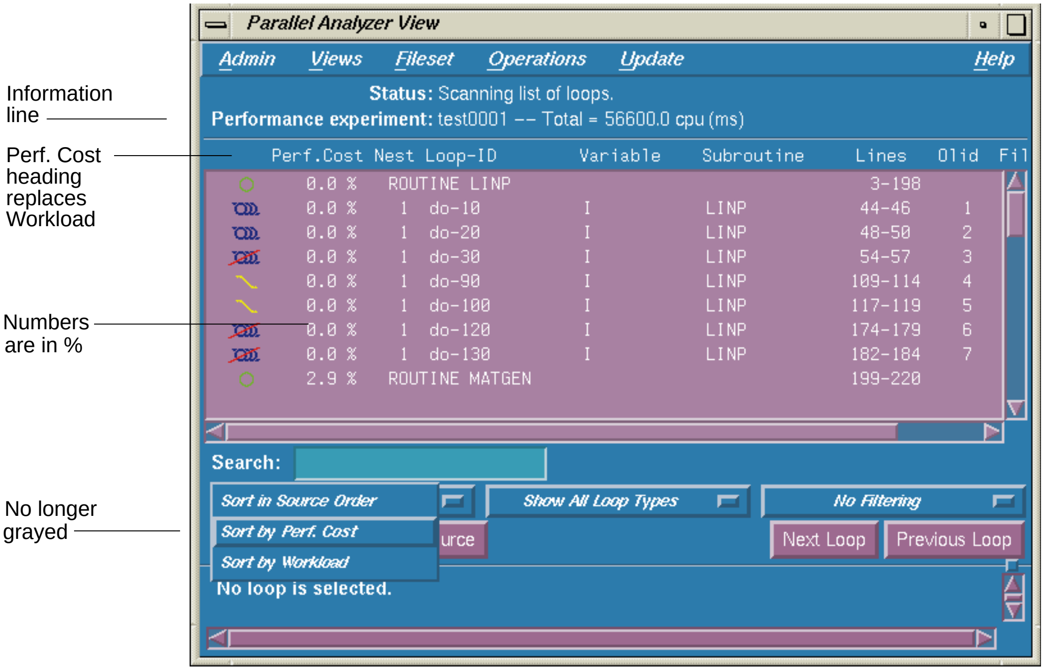


Figure 3-36 Performance Data — Parallel Analyzer View

- The second column of the list of loops has changed from reading “Workload” to reading “Perf. Cost”, and the numbers below it are now percentages.
- The second line in the view now shows the name of the performance experiment and shows the total cost of the run. In addition, the sort menu’s second entry “Sort by Perf. Cost” is no longer grayed-out.
- The Source View now has three additional columns to the left of the loop brackets that show the performance metrics, including the number of times the line has been executed and ideal CPU times as shown in Figure 3-37. The times are exclusive, inclusive, ideal, or CPU time in milliseconds.

These columns reflect the measured performance data. If you select loop **do-50** of subroutine **DAXPY** from the main view, the Source View displays as shown in Figure 3-37.

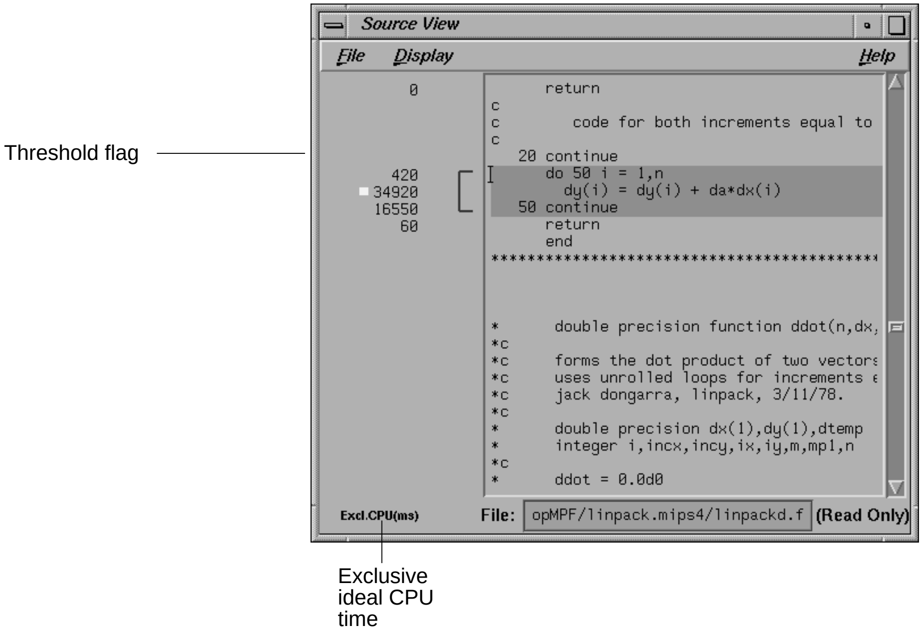


Figure 3-37 Source View for Performance Experiment

Select the “Sort by Perf. Cost” entry. Note that loop do-50 of subroutine DAXPY represents approximately 92% of the total time. These numbers are inclusive numbers, with each reflecting the time in the loop and in any nested loops or functions called from within the loop. See Figure 3-38.

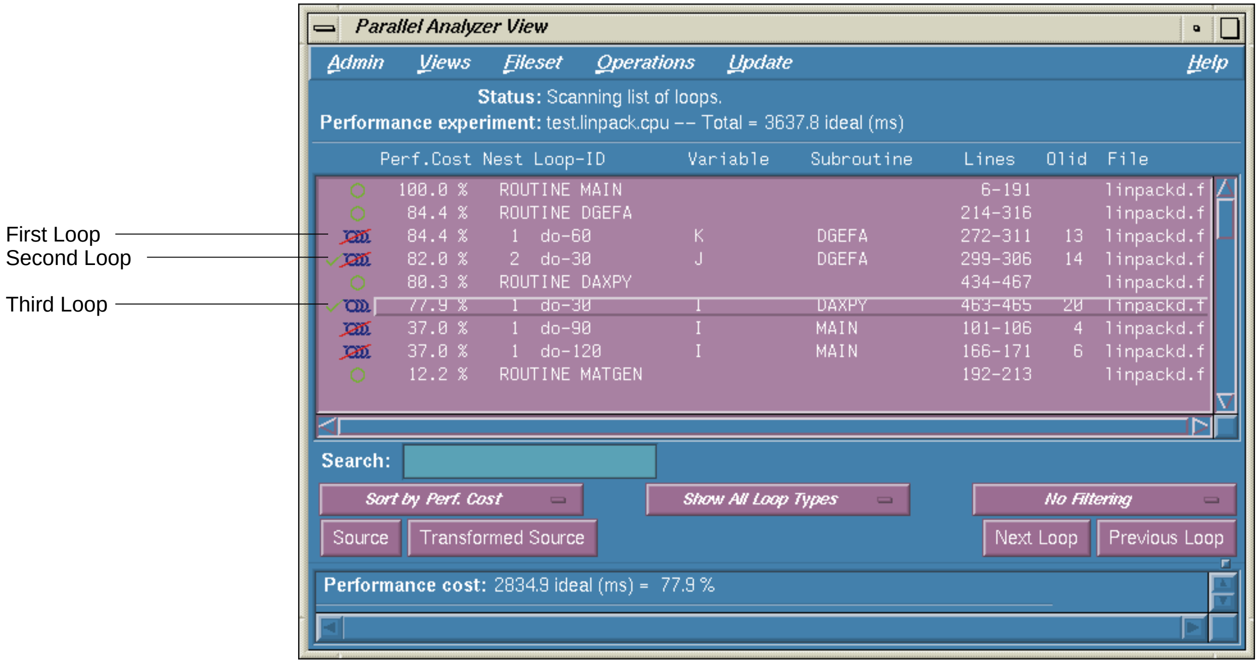


Figure 3-38 Sort by Performance Cost

The first of these loops contains the second loop nested inside it. The second loop calls the subroutine DAXPY, which contains the third loop. The third loop is the heart of the *linpack* benchmark and is already parallel.

Double-click the third loop. Note that the loop information display now contains an additional line of text listing the performance cost of the loop, both in time and as a percentage of the total time. See Figure 3-39.

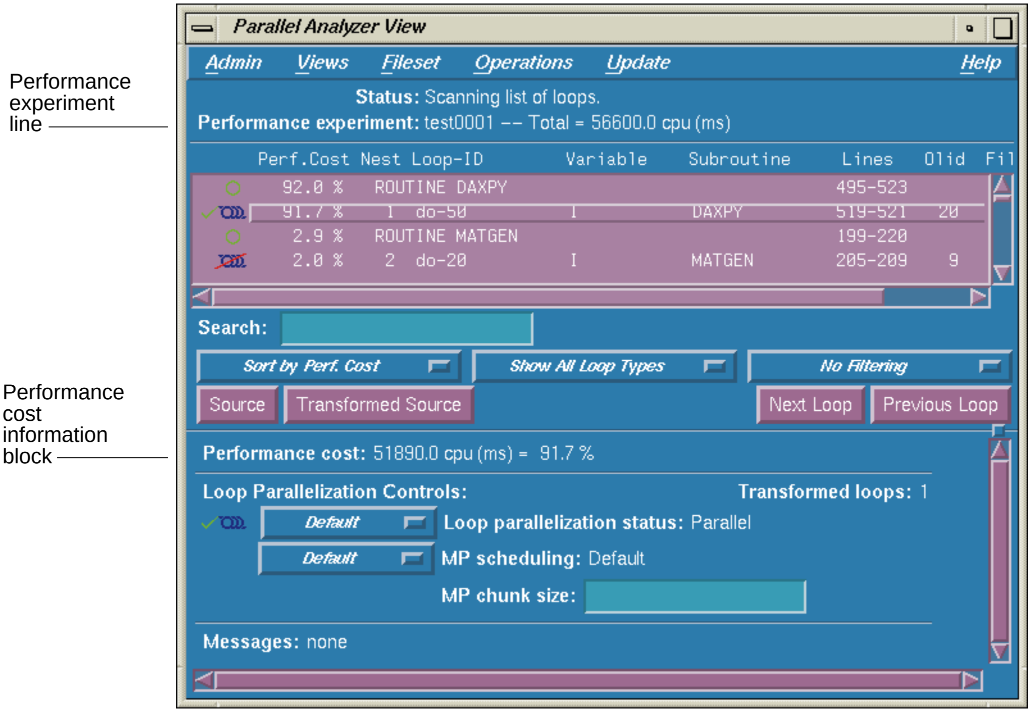


Figure 3-39 Loop Information Display with Performance Data

Exiting from the linpackd Sample Session

This completes the second sample session. Quit by selecting the “Exit” command from the Project submenu of the Admin menu in the Parallel Analyzer View. All the windows will close.

You don’t need to clean the directory, because you haven’t made any changes in this session. If you do make changes, when you are finished you can clean up the directory by entering:

```
% make clean
```

in your shell window. All generated files will be removed.

Setting Up the f90 Sample Session

The f90 sample session is located in the directory */usr/demos/WorkShopMPF/cgdriver*. Prepare for the session by changing directories to the demo directory and creating the needed files:

```
% cd /usr/demos/WorkShopMPF/cgdriver
% make
```

Once the demo directory has been prepared, start the session by entering:

```
% cvpav -f cgdriver.f
```

Notice that the loop list contains Fortran 90 array syntax statements. Double click on the first statement in **CGTEST** (b = 0). You can see in the loop information display that the array-syntax is an implied loop and the statement was converted from array notation into a serial loop.

Click on the *Source* button. Notice that in source view, Fortran 90 array syntax statements (in the subroutine **CGTEST**) are bracketed in blue (they are shown as loops). Click on the *Transformed Source* button to see the transformation that PFA has performed. You can see that since *b* is a 3-dimensional array which is initialized to 0, the transformed source contains 3 nested do loops (each one spanning one dimension).

Exiting from the f90 Sample Session

This completes the third sample session. Quit the Parallel Analyzer View by selecting “Exit” from the Admin menu.

To clean up the directory, so that the session can be rerun, enter:

```
% make clean
```

in your shell window. All of the generated files will be removed.

Parallel Analyzer View Reference

Figure 4-1 Icon for *cvpav*

This chapter describes in detail the function of each window, menu, and display in the WorkShopProMPF Parallel Analyzer View's user interface. Figure 4-1 shows the application's icon.

This chapter contains the following sections:

- "Main View Menu Bar"
- "Loop List"
- "Loop Information Display"
- "Other Views"
- "Original and Transformed Source Windows"
- "C$DOACROSS Parallelization Control View"
- "C$PAR PDO Parallelization Control View"
- "Icon Legend"

Main View Menu Bar

This section describes the menus found in the menu bar of the Parallel Analyzer View main window as shown in Figure 4-2. By selecting the dashed line (the first item in each of the menus), you can "tear off" the menu from the menu bar, so that it is displayed in its own window, with each menu command visible at all times. Some menus contain submenus, which can also be torn off and displayed in their own window.

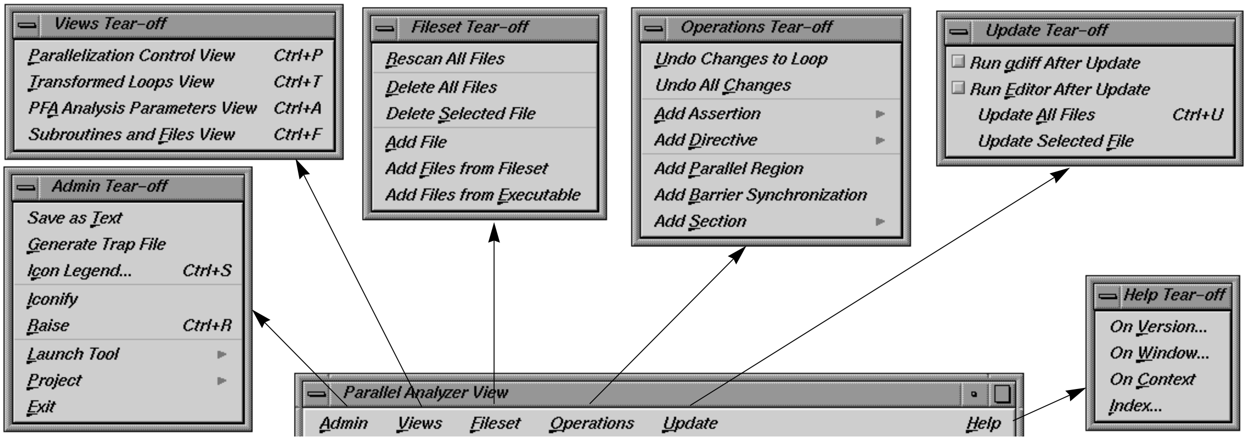


Figure 4-2 Parallel Analyzer View Menu Bar

Admin Menu

The Admin menu contains general administrative commands and commands for launching and manipulating other WorkShop application views as shown in Figure 4-3. The commands are described as follows:

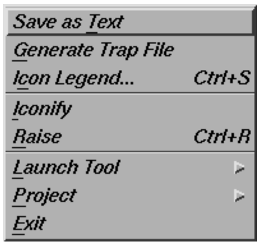


Figure 4-3 Main View Admin Menu

"Save as Text"

saves the complete loop information for all files and subroutines in the current session into a plain ASCII file. Selecting "Save as Text" brings up the directory and file browser, which lets you select where to save the file and what name to call it (see Figure 4-4). The default directory is the same one the Parallel Analyzer View was invoked from at the shell prompt; the default file name is *Text.out*. The Parallel Analyzer View will ask for confirmation before overwriting an existing file.

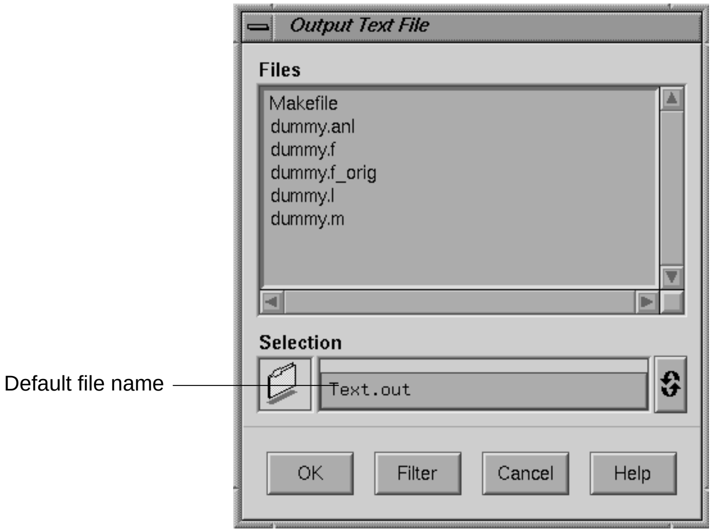


Figure 4-4 Directory and File Browser Window

"Generate Trap File"
: generates a file for use in conjunction with the WorkShop Trap Manager. The trap file specifies sample traps at the entry and exit to each outer loop. See Chapter 3, "A Short Debugger Tutorial," in the *Debugger User's Guide* for more information on trap files and the Trap Manager. The default directory is the same one the Parallel Analyzer View was invoked from at the shell prompt; the default file name is *cvmpTrapFile*. The Parallel Analyzer View will ask for confirmation before overwriting an existing file.

"Icon Legend..."
: opens the Icon Legend dialog box, which provides an explanation of the graphical icons used in the Parallel Analyzer View. See "Icon Legend" on page 139. Shortcut: **`<Ctrl>-S`**

“Iconify” stows all the open windows belonging to a given invocation of the Parallel Analyzer View as icons, as per 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 a given invocation of the Parallel Analyzer View and brings them to the foreground. Shortcut: **`<Ctrl>-R`**

“Launch Tool” See “Launch Tool Submenu” on page 104.

“Project” See “Project Submenu” on page 106.

“Exit” quits the current session of the Parallel Analyzer View, closing all windows. If you have made changes to source files without updating them, a dialog box asks if it is okay to discard the changes. Click on *OK* only if you want to *discard* any changes you’ve made; otherwise, click on *Cancel*.

Launch Tool Submenu

The Launch Tool submenu contains commands for launching other WorkShop applications, as well as new sessions of the Parallel Analyzer (see Figure 4-5). In order to work properly with the other WorkShop applications, the files in the current fileset must have been loaded into the Parallel Analyzer from an executable using either the **-e** option on the command line (see “Starting the Parallel Analyzer View” on page 2) or the “Add Files from Executable” command found in the Fileset menu (see “Fileset Menu” on page 108). If launched from a session not based on an executable, the tools will be launched without arguments.

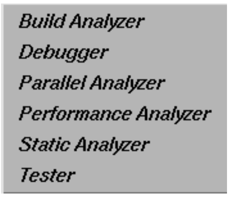


Figure 4-5 Launch Tool Submenu

The applications launchable from the menu are the following:

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

WorkShop Debugger
: Launches the Debugger, a UNIX source-level debugging tool that provides special windows (views) for displaying program data and execution status. See Chapter 1, “Getting Started with the WorkShop Debugger,” in the *Developer Magic: Debugger User’s Guide* for further information.

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. See Chapter 1, “Introduction to the Performance Analyzer,” in the *Developer Magic: Performance Analyzer User’s Guide* for further information.

Static Analyzer
: Launces the Static Analyzer, a utility which allows you to analyze and display source code written in C, C++, Fortran, or Ada. See Chapter , “Introduction to the WorkShop Static Analyzer,” in the *Developer Magic: Static Analyzer User’s Guide* for further information.

Tester
: Launches the Tester, a UNIX-based software quality assurance toolset for dynamic test coverage over any set of tests. See Chapter 5, “Using Tester,” in the *Developer Magic: Performance Analyzer User’s Guide* for further information.

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

Project Submenu

The Project submenu contains commands that affect all the windows in a WorkShop *project*, that is, all the windows containing WorkShop or WorkShopProMPF applications that have been launched to manipulate a single executable as shown in Figure 4-6.

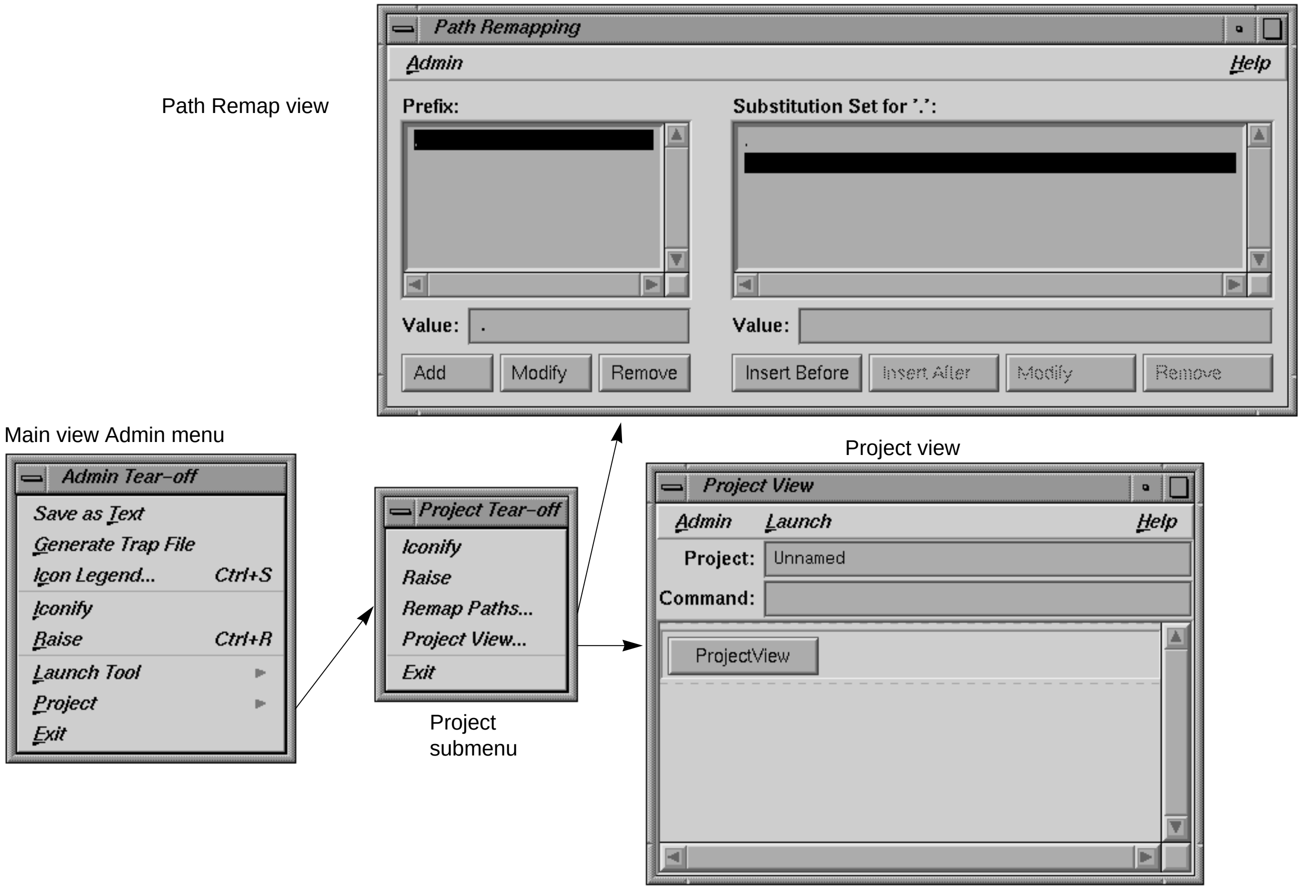


Figure 4-6 Project Submenu Commands

The Project submenu commands are as follows:

"Iconify" stows all the windows in the current project as icons, as per 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 file names 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. If you have made changes to source files without updating them, a dialog box asks if it is okay to discard the changes. Click on *OK* only if you want to *disregard* any changes you've made; otherwise, click on *Cancel*.

Views Menu

The Views menu (see Figure 4-7) contains commands for launching a variety of secondary windows, or *views*, the function each of which is described as follows:

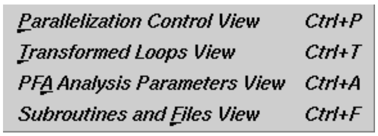


Figure 4-7 Views Menu

"Parallelization Control View"
opens a Parallelization Control View for the looper currently selected (double-clicked) from the loop list display. For more information on this view, see "Parallelization Control View" on page 127. Shortcut: **`<Ctrl>-T`**

"Transformed Loops View"
: opens a Transformed Loops View for the loop currently selected (double-clicked) from the loop list display. For more information on this view, see "Transformed Loops View" on page 134 Shortcut: **<Ctrl>-T**

"PFA Analysis Parameters View"
: opens the PFA Analysis Parameters View, which provides a means of modifying a variety of PFA parameters. This view is further described in "PFA Analysis Parameters View" on page 135. Shortcut: **<Ctrl>-P**

"Subroutines and Files View"
: opens the Subroutines and Files View, which provides a complete list of subroutine and file names currently being examined within the current session of the Parallel Analyzer View. This view is further described in "C$DOACROSS Parallelization Control View" on page 130. Shortcut: **<Ctrl>-F**

Fileset Menu

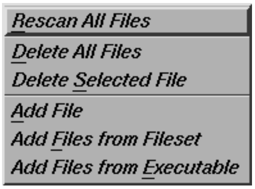


Figure 4-8 Fileset Menu

The Fileset menu (see Figure 4-8) contains commands for manipulating the files displayed by the Parallel Analyzer View. The selections are as follows:

"Rescan All Files"
: causes the Parallel Analyzer View to check and update all the source files loaded into its current session to match the versions of those files in the file system. It will only reread the files it needs to.

"Delete All Files"
: removes all files from the current session of the Parallel Analyzer View. You can then add new files using the "Add File", "Add Files from Fileset", or "Add Files from Executable" commands, described below.

"Delete Selected File"
: deletes a selected file from the current session of the Parallel Analyzer View. You can select a file for deletion by

double-clicking with the left mouse button within the Subroutines and Files View on the line corresponding to the desired file name.

“Add File” adds a new source file to the current session of the Parallel Analyzer View. Selecting this command brings up a file and directory browser that lets you select a Fortran source file. Before you can select a given source file, you will need to run PFA on it. If the current session is based on an executable (see the “Add Files from Executable” command, described below), you cannot add files to it until you have deleted the executable’s fileset.

“Add Files from Fileset”
lets you add a list of new source files to the current session of the Parallel Analyzer View. A *fileset* is a list of source file names contained in an ASCII file, each on a separate line. Selecting the “Add Files from Fileset” command will bring up the file and directory browser as it does for the “Add File” command. If you select a file containing a fileset list, all Fortran source files in the list are loaded into the current session (other files in the list are ignored). If the current session is based on an executable (see “Add Files From Executable”), you cannot add files to it until you have deleted the executable’s fileset.

“Add Files from Executable”
imports all the Fortran source files listed in the symbol table of a compiled Fortran application. This command will only work if there are no files in the current session of the Parallel Analyzer View when the command is selected from the menu. Other WorkShop applications (see “Launch Tool Submenu” on page 104) will also be able to operate on files imported from an executable.

Operations Menu

The Operations menu contains commands for undoing changes to source files and for adding assertions and directives to loops as shown in Figure 4-9.

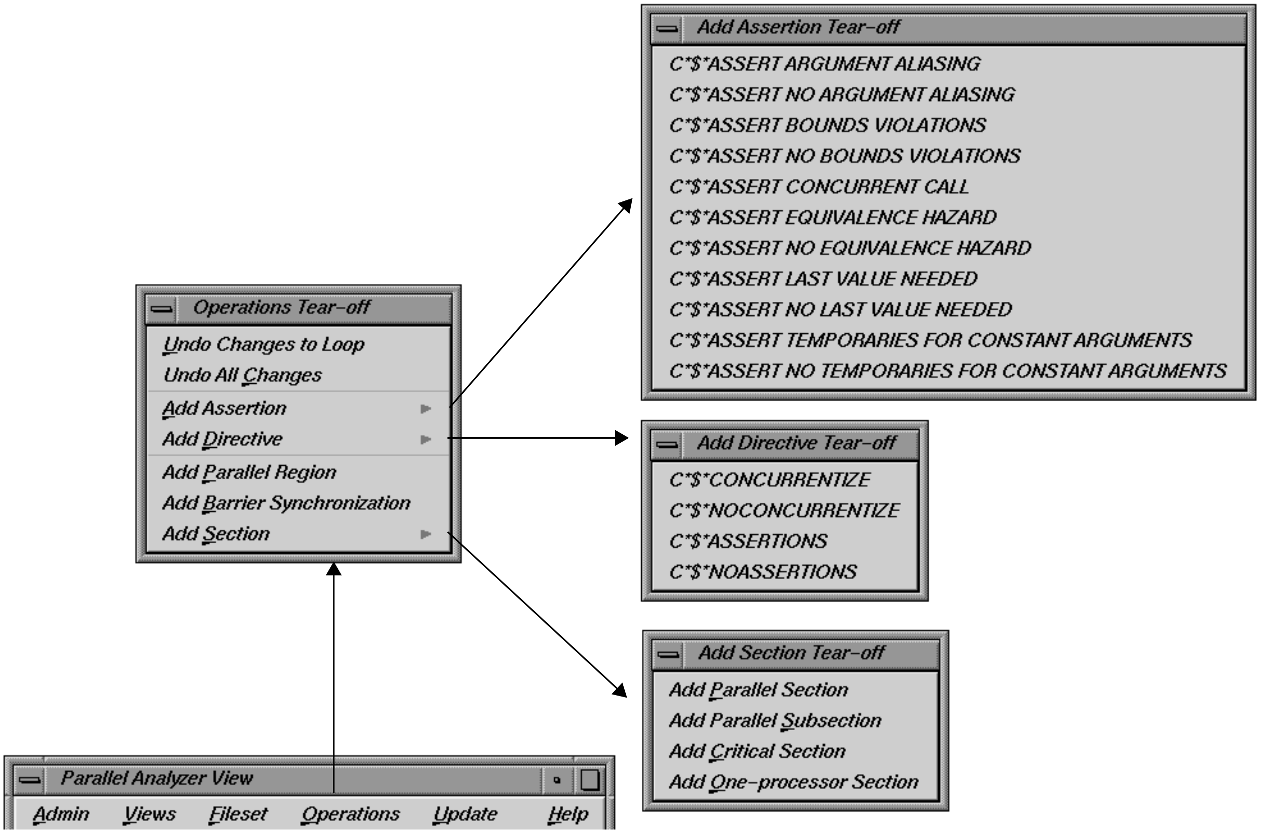


Figure 4-9 Operations Menu and Submenus

"Undo Changes to Loop"
removes any non-updated changes to the currently selected loop that were made using the option menus in the loop information display. Changes that have already been written to the source file using the Update menu commands cannot be undone.

"Undo All Changes"
removes any non-updated 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.

The Add Assertion Submenu
: contains a set of PFA assertions that you can select in order to add them to the currently selected loop. These assertions are explained in detail in “Appendix C, PFA Assertions” in the *POWER Fortran Accelerator User’s Guide*.

The Add Directive Submenu
: contains a set of PFA directives that you can select in order to add them to the currently selected loop. These directives are explained in detail in “Appendix B, PFA Directives” in the *POWER Fortran Accelerator User’s Guide*.

“Add Parallel Region”
: allows you to add a parallel region PCF construct.

“Add Barrier Synchronization”
: allows you to add a barrier synchronization PCF construct.

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

When you add a new construct, the list is redrawn with the new construct in place, and the new construct is selected. Brackets defining the new constructs are NOT added to the file loop annotations. 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 will be not be in the correct order.

Update Menu

Figure 4-10 Update Menu

The Update menu (see Figure 4-10) contains commands for managing changes to PFA directives and assertions made in the Parallel Analyzer View to your Fortran source code.

"Run gdiff After Update"

sets a toggle switch that will cause a *gdiff* window to open after you have updated changes to your source file. This window graphically illustrates the differences between the unchanged source and the newly updated source. If you always wish to see the *gdiff* window, you may set the resource in your *.Xdefaults* file:

```
cvpav*gDiff: True
```

See the man page for *gdiff*(1) for more information on using *gdiff*.

"Run Editor After Update"

sets a toggle switch that will cause an *xwsh* shell window with the *vi* editor running it to open the updated source file. See Figure 4-11.

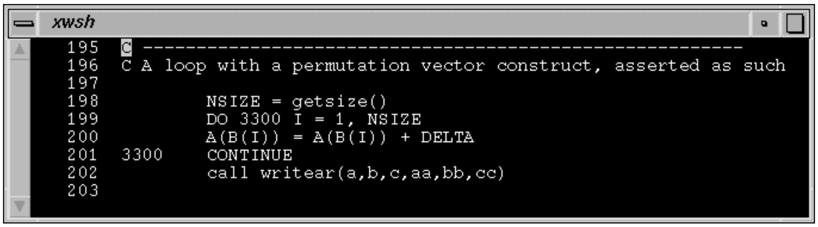


Figure 4-11 Viewing the Updated Source in an Editor

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 change the following resource in your *.Xdefaults* file, changing the *xwsh* and / or *vi* as you prefer:

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

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 will either replace any explicit **%s**, or if the **%s** is omitted, the file name will be appended to the command.

"Update All Files"
: writes all changes made to loops in the current session of the Parallel Analyzer View to the appropriate source files. Shortcut: **<Ctrl>-U**

"Update Selected File"
: writes changes made to loops found within a selected file from the current session of the Parallel Analyzer View. You can select a file for saving by double-clicking with the left mouse button within the Subroutines and Files View on the line corresponding to the desired file name. See "C$DOACROSS Parallelization Control View" on page 130.

Help Menu

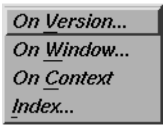


Figure 4-12 Help Menu

The Help menu contains commands that allow you to access on-line information and documentation for the Parallel Analyzer View as shown in Figure 4-12.

"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 select the "On Context" command, the normal mouse cursor (an arrow) is replaced with a question mark. When you click on graphical features of the application with the left mouse or position the cursor over the feature and press the **<F1>** key, the Help Viewer displays information on that context.

"Index..."	invokes the Help Viewer, that displays the list of available help topics, which you can browse alphabetically, hierarchically, or graphically.

Keyboard Shortcuts

The following accelarator keys are available through MPF:

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 Analyzis Parameters View
Ctrl-F	Views -> Subroutines and Files View
Ctrl-U	Update -> Update All Files

Loop List

This section describes the loop list and the various option menus and fields that manipulate the information shown in the loop list display as shown in Figure 4-13.

You may resize the loop list to reduce the number of loops displayed. You might have noticed that many figures in this manual show the loop list focused on the selected loop. The adjustment button is in the lower right hand corner of the loop list display, just above the loop information display. Your screen shows the full list unless you resize it.

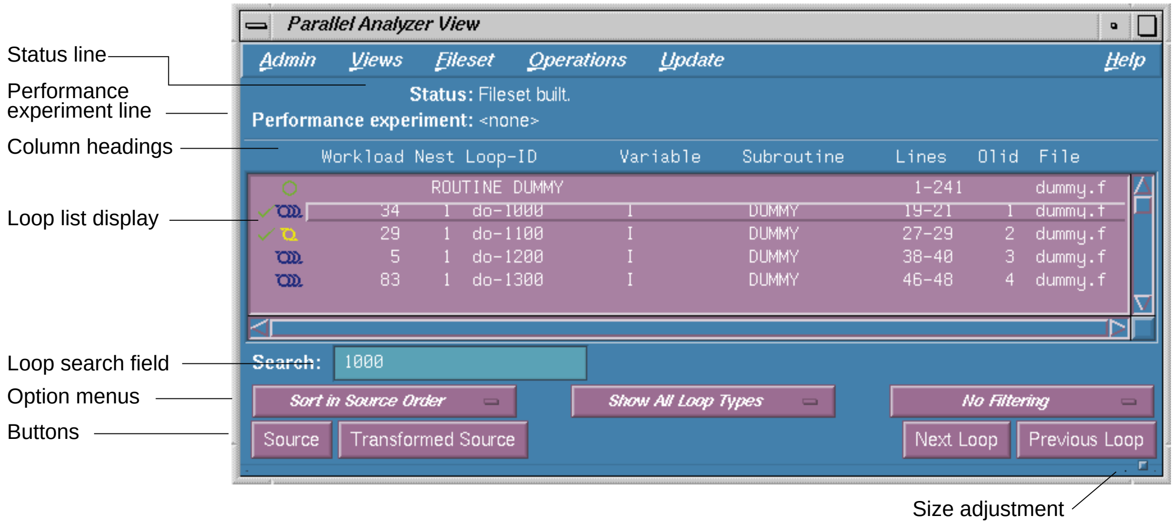


Figure 4-13 Loop List Display and Controls

Status and Performance Experiment Lines

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

The performance experiment line displays the name of the current experiment directory and the type of experiment data derived from the WorkShop Performance Analyzer (see “Launch Tool Submenu” on page 104 for information on invoking the Performance Analyzer from the Parallel Analyzer View), as well as total data for the current caliper setting in the Performance Analyzer. If the Performance Analyzer is not being used, the performance experiment line displays <none>.

Loop List Display

The loop list display lets you select and manipulate any Fortran DO loop contained in the source files loaded into the Parallel Analyzer View as part of the current session. The loops themselves are stacked as rows in the list display; information about the loops is displayed in columns, the contents of which are shown in Figure 4-14 and described below.

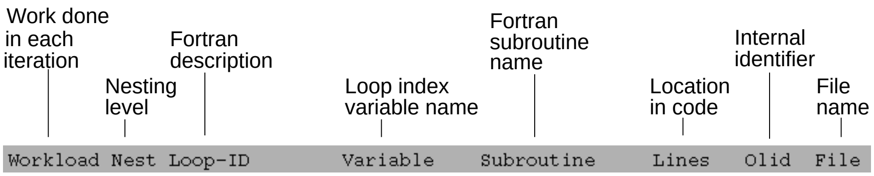


Figure 4-14 Column Headings for the Loop List Display

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

parallelization icon

describes the parallelization status of each loop. The meaning of each of these icons is described in the Icon Legend dialog box (see “Icon Legend” on page 139). When a loop is displayed in the loop information display (by double-clicking with the left mouse button elsewhere in 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.

Workload and **Perf. Cost** (performance cost)

allow you to gauge loop performance. Workload provides a means of roughly determining the relative amount of work done in each iteration of the loop. The loops can be sorted in the loop list display by the workload value, instead of by physical ordering in the file. See “Sort Option Menu” on page 118. Workload is displayed when no performance data (from the WorkShop Performance Analyzer) is available.

Perf. Cost replaces Workload when the WorkShop Performance Analyzer is launched on the current fileset (see “Launch Tool Submenu”). Performance experiment data from the Performance Analyzer is then listed in place of workload data. As with Workload, the loops can be sorted by Perf. Cost via the sort option menu.

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 shows the nesting level of the given loop.

Loop-ID provides 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 provides the name of the loop index variable.

Subroutine provides the name of the Fortran subroutine in which the loop occurs.

Lines provides the lines in the source file that comprise the body of the loop.

Olid provides a unique internal identifier for the loops generated by PFA. Please use this value when reporting bugs.

File provides the name of the Fortran source file that contains the loop.

Clicking the left mouse anywhere in a given row highlights that loop in the list display, and typing text into the Search field (see "Loop List Search Field" on page 117) will do the same. Double-clicking on a row will bring up detailed information in the loop information display below the loop list display (see "Loop Information Display" on page 120).

Loop List Search Field

You can use the loop list search field to find a specific loop in the loop list display. The field will match any text typed into it to the first instance of that text in the loop list display, and will highlight the row of the display in which that text occurs. The search field will match its text against the contents of each column in the loop list display.

As you type into the field, the list will highlight the first entry that matches what you have already typed, scrolling the list if necessary. If you type `<Enter>`, the highlight will move to the next match. If no match is found, the system will beep, and typing **<Enter>** will position the highlight at the top of the list again.

Sort Option Menu

The sort option menu (see Figure 4-15) controls the order in which the loops are displayed in the loop list display. The choices are as follows:

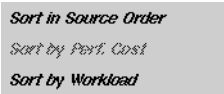


Figure 4-15 Sort Option Menu

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

Sort By Performance Cost
orders the loops by their performance cost (from greatest to least) as calculated by the Workshop Performance Analyzer. You need to have invoked the Performance Analyzer from the current session of the Parallel Analyzer View to make use of this option. See "Launch Tool Submenu" on page 104 for information on how to open the Performance Analyzer from the current session of the Parallel Analyzer View.

Sort By WorkLoad
orders the loops from largest to smallest workload.

Show Loop Types Option Menu

The show loop types option menu (see Figure 4-16) controls what kind of loops are displayed for each file and subroutine in the loop list display. The choices are as follows:

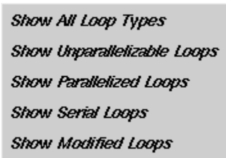


Figure 4-16 Show Loop Types Menu

- **Show All Loop Types** is the default setting.
- **Show Unparallelizable Loops** shows only loops that could not be parallelized.
- **Show Parallelized Loops** shows only loops that are parallelized.
- **Show Serial Loops** shows only loops that are preferably serial.
- **Show Modified Loops** shows only loops with pending changes.

Filtering Option Menu

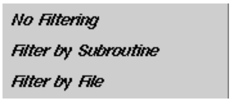


Figure 4-17 Filtering Option Menu

The filtering option menu (see Figure 4-17) lets you display only those loops contained within a given subroutine or source file. The choices are as follows:

No Filtering — is the default setting.

Filter By Subroutine — lets you enter a subroutine name into a filtering text field that appears above the option menu. Only loops contained in that subroutine will be displayed in the loop list display.

Filter By File — lets you enter a Fortran source file name into a filtering text field that appears above the option menu. Only loops contained in that file will be displayed in the loop list display.

Double-clicking on a line in the Subroutines and Files View will cause the name of that subroutine or file to be inserted into the appropriate filter text field. If the appropriate type of filtering is currently selected, the loop list is rescanned.

Loop List Buttons

The Loop List contains the buttons described below.

Source — opens the Original Source window, with the source file containing the loop currently selected (double-clicked) in the loop list display. The body of the loop is highlighted within the window. For more information on the Original Source window, see “Original and Transformed Source Windows” on page 138. If no loop is selected, the last selected file is loaded; if no file is selected, the first file in the fileset will be loaded.

Transformed Source — opens a Transformed Source window, with the PFA-processed source file containing the loop currently selected (double-clicked) in the loop list display. The body of the loop is highlighted within the window. For more information on the Transformed Source window, see

"Original and Transformed Source Windows" on page 138. If no loop is selected, the last selected file is loaded; if no file is selected, the first file in the fileset will be loaded.

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

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

Loop Information Display

The loop information display provides detailed information on various loop parameters and allows you to alter those parameters so that the changes can be incorporated into the Fortran source. The display is divided into several information blocks displayed in a scrolling list as shown in Figure 4-18.

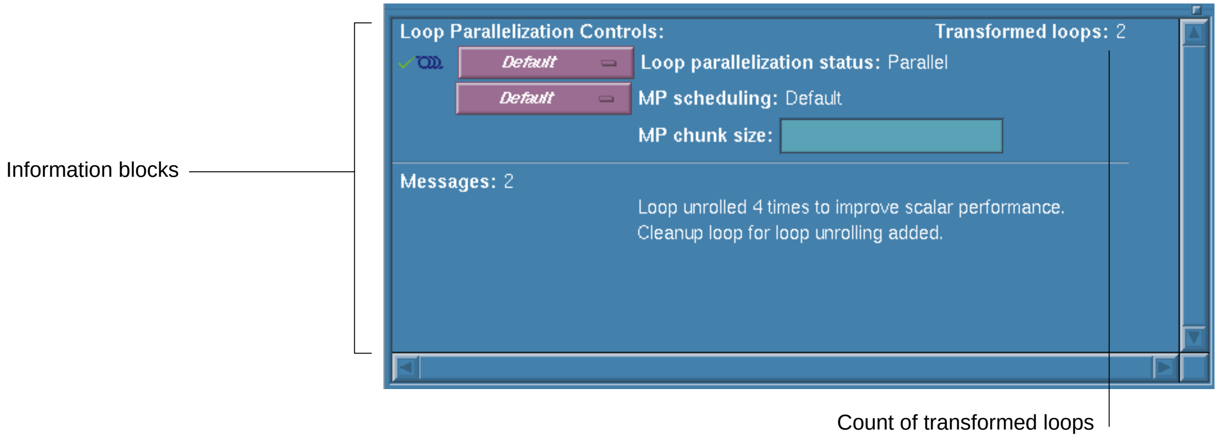


Figure 4-18 Loop Information Display

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

Figure 4-19 Highlighting Button

A highlighting button (light bulb, see Figure 4-19 and Figure 4-20) appears as a shortcut to related information. Clicking the button opens an Original Source window (if necessary), highlighting the loop and the line that generated the question.

Parallelization Controls

The first section contains controls for altering the parallelization of the selected loop that are described below. See Figure 4-20. On the far right, the first line of the Parallelization Controls section shows how many transformed loops were derived from the selected loop.

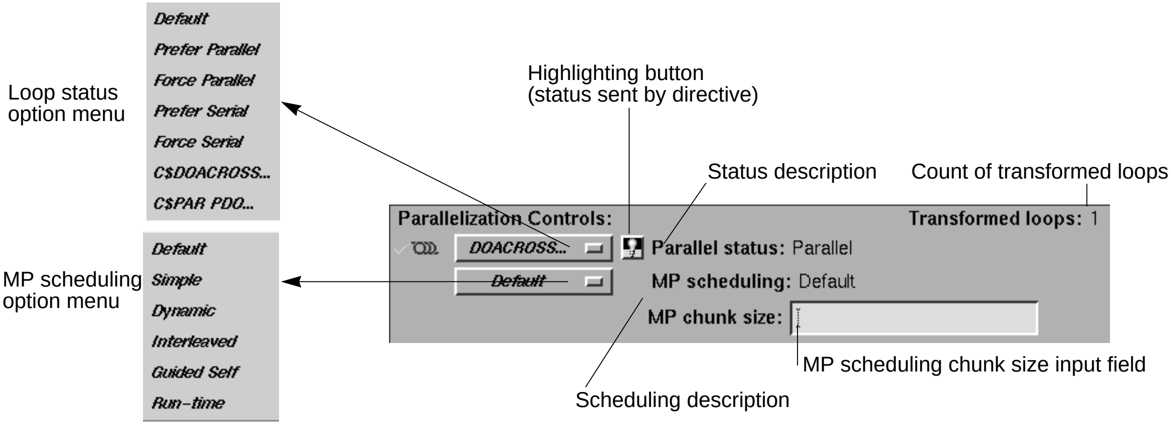


Figure 4-20 Parallelization Controls

Loop Status Option Menu

The loop status option menu lets you alter a loop's parallelization scheme. To the right of the option menu is a description of the current loop status as implemented in the transformed source. A small highlighting button appears to the left of this description if the status was set by a directive as shown in Figure 4-20. See Chapter 5, "Fine Tuning for PFA," in the *POWER Fortran Accelerator User's Guide* for more information on the menu choices.

The menu choices are as follows:

Default	always selects the parallelization scheme that PFA has picked for the selected loop.
Prefer Parallel	adds the assertion C*$*ASSERT DO PREFER (CONCURRENT), which causes PFA to try to transform the selected loop into a parallel loop. If this is not possible, PFA will try to run each nested loop in parallel.
Force Parallel	addsan assertion C*$*ASSERT DO (CONCURRENT), which causes PFA to ignore assumed data dependencies that would normally be considered obstacles to parallelization on the selected loop and any nested loops.
Prefer Serial	addsthe assertion C*$ASSERT DO PREFER (SERIAL), which prevents PFA from trying to parallelize the selected loop.
Force Serial	adds the assertion C*$*ASSERT DO (SERIAL), which prevents PFA from trying to parallelize the selected loop or any loop that surrounds it.
C$DOACROSS...	adds the directive C$DOACROSS, which tells the Fortran compiler to generate parallel code for the selected loop without any interference by PFA. Selecting this item opens the Custom DOACROSS Dialog box. See "C$DOACROSS Parallelization Control View" for more information.
C$PAR PDO...	launches a Parallel DO Dialog, which allows you to manipulate the scheduling clauses for the Parallel-DO and to set each of the referenced variables as either region-default or last-local. A Parallel-DO must be within a Parallel Region, although the tool does not enforce this restriction. If one is added outside of a region, the compiler will report an error.

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

MP Scheduling Option Menu

The MP scheduling option menu lets you alter a loop's scheduling scheme by changing the C$MP_SCHEDTYPE and C$CHUNK directives. These directives affect the current loop *and all subsequent loops* in a source file. For control over a single loop, see “Parallelization Control View MP Scheduling Option Menu” on page 129.

The menu choices are as follows:

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

See Section 5.3, “Writing Parallel Fortran,” in the *Fortran 77 Programmer’s Guide* for more information on the functions listed above.

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

MP Scheduling Chunk Size Field

Below the scheduling description is an input field that allows you to set the CHUNK size for the scheduling scheme you select. When you change an entry in the field, the upper right corner of the field will turn down, indicating the change. To toggle back to the original value, left-click the turned-down corner (changed-entry indicator). The corner will unfold, 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 will always remember the original value, which will always be displayed after you click on the changed-entry indicator. See Figure 4-21.

Figure 4-21 MP Chunk Size Input Field Changed

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

Questions

In some cases, PFA asks one or more questions when it encounters a data dependence. The Parallel Analyzer View creates option menus allowing you to answer “Don’t Know”, “Assert False”, or “Assert True” to each question as shown in Figure 4-22. When you click on the small highlighting button to the left of a question, an Original Source window opens (if necessary), highlighting the loop and the line that generated the question. For the questions, it also highlights a variable name.

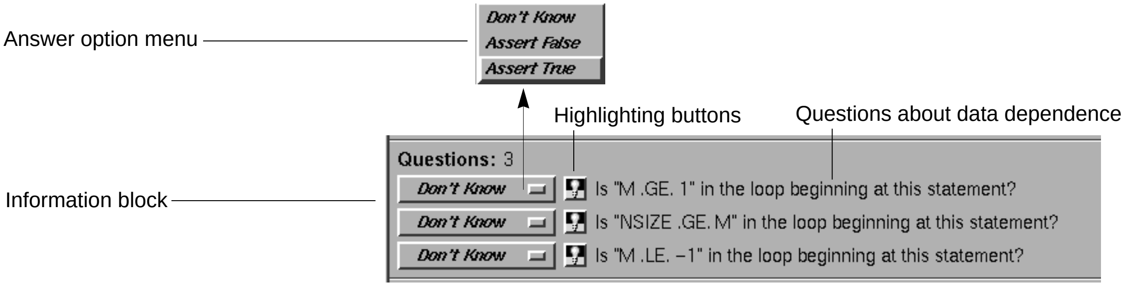


Figure 4-22 Questions Information Block

Obstacles to Parallelization

Obstacles to parallelization are listed when PFA discovers aspects of a loop's structure that make it impossible to parallelize. See Figure 4-23. These are listed messages describing an obstacle, and each has a corresponding button directly to its left. When you left-click on one of these buttons, the Parallel Analyzer View highlights the troublesome line in the Original Source window, opening the window if necessary. If appropriate, the referenced variable or function call is highlighted in a contrasting color.

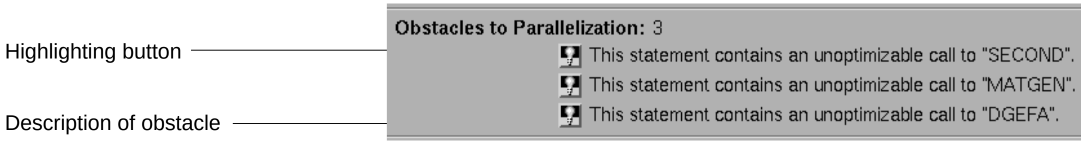


Figure 4-23 Obstacles Information Block

Assertions and Directives

Assertions and directives are special POWER Fortran source comments used to tell PFA how to transform Fortran code. Directives enable, disable, or modify features of PFA when it runs on the source. Assertions provide PFA with additional information about the source code that can sometimes improve optimization. Figure 4-24 shows an assertion block and its option menu.

Figure 4-24 Assertion Information Block

The Parallel Analyzer View lists assertions and directives along with buttons in the loop information display. Some are also listed with an option menu that allows you to “Keep”, “Delete”, or “Reverse” (if appropriate) the corresponding assertion or directive. When you left-click the small highlighting button to the left of an assertion or directive, an Original Source window shows the selected loop with the assertion or directive highlighted in the code. Assertions and directives that govern loop parallelization or scheduling do not have associated option menus; those functions are controlled by the loop status option menu and the MP scheduling option menu (see “Parallelization Controls” on page 121).

PFA Messages

PFA sometimes generates messages describing aspects of the loops it creates by transforming original source loops. The Parallel Analyzer View displays these messages; some also have associated buttons that highlight sections of the selected loop in the Original Source window.

Other Views

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

Admin menu — contains a single “Close” command that closes the corresponding view

Help menu — provides access to the on-line help system (see “Help Menu” on page 113 for an explanation of the commands in this menu)

Parallelization Control View

The Parallelization Control View shows parallelization controls, where applicable, and all the variables referenced in the selected loop / PCF-construct / routine. In addition to being raised when ‘‘C$DOACROSS’’ or ‘‘C$PAR PDO’’ is selected for a loop, it can be raised from the Views menus, and it need not be closed to move from loop to loop. For loops, the variable list is obtained as in the previous releases, that is, from the analysis file; for other constructs, the variable list is obtained from the WorkShop Static Analyzer. If no response is received from the Static Analyzer, a dialog suggesting that you invoke it is raised. In addition, there is a text field for you to enter a comma-separated list of variables and an ‘‘Add Variable’’ button.

You can open this view by one of the following:

- pulling down the Views menu of the Parallel Analyzer View and selecting “Parallelization Control View” (see “Views Menu” on page 107)
- selecting either “C$DOACROSS...” from the loop status option menu in the loop information display
- selecting either “C$PAR PDO...” from the loop status option menu in the loop information display

Figure 4-25 displays the view when it is launched from the Views menu, with the loop status option menu set to `Default`.

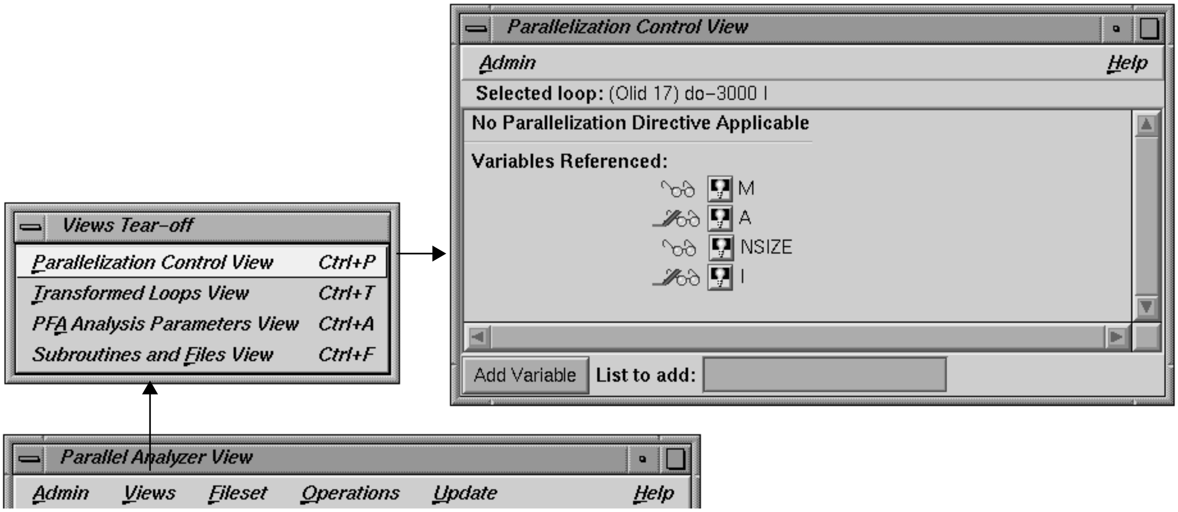


Figure 4-25 Parallelization Control View

Both the C$DOACROSS and C$PAR PDO modes of the Parallelization Control View contain the following items:

Admin menu
: Contains only one selection, “Close,” which closes the View.

MP Scheduling menu
: Allows you to alter a loop’s scheduling scheme by changing the C$MP_SCHEDTYPE and C$CHUNK directives. See “Parallelization Control View MP Scheduling Option Menu” on page 129 for futher information.

“MP chunk size” text field
: Allows you to set the CHUNK size for the scheduling scheme you select.

Variable Option menus
: Allows you to select the variable type. See “Parallelization Control View Variable Option Menus” on page 130 for further information.

Add Variable button
: Allows you to add new variables to a loop.

"List to add" 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.

For further details on the C$DOACROSS and C$PAR PDO modes of the Parallelization Control View, see "C$DOACROSS Parallelization Control View" on page 130 and "C$PAR PDO Parallelization Control View" on page 132.

Parallelization Control View MP Scheduling Option Menu

The Parallelization Control View contains an MP scheduling option menu (see Figure 4-26) identical to the one that appears for a selected loop in the loop information display. This option menu affects the MP_SCHEDTYPE and CHUNK clauses in the C$DOACROSS directive, which affect only the currently selected loop.

Figure 4-26 MP Scheduling Option Menu

The menu choices are as follows:

Default
: always selects the scheduling scheme that PFA has picked for the selected loop.

Simple
: divides iterations of the selected loop among the processors by dividing them into contiguous pieces, and assigns one to each processor.

Dynamic
: divides iterations of the selected loop among the processors by dividing them into pieces of size CHUNK. As each processor finishes a piece, it enters a critical section to grab the next piece. This scheme provides good load balancing at the price of higher overhead.

Interleaved
: divides the iterations into pieces of size CHUNK and the execution of those pieces is interleaved among the processors. For example, if there are four processors and CHUNK=2, then the first processor executes iterations 1-2, 9-10, 17-18,...; the second processor executes iterations 3-4, 11-12, 19-20,...; and so on.

Guided-Self
: divides the iterations into pieces. The size of each piece is determined by the number of total iterations remaining. By parceling out relatively large pieces to start with and

relatively small pieces toward the end, the idea is to achieve good load balancing while reducing the number of entries into the critical section.

Run-time — lets the user specify the scheduling type at run-time.

Parallelization Control View Variable Option Menus

Below the MP scheduling option menu is a display area containing each of the variables found in the selected loop. Each variable name is displayed to the right of a highlighting button. To the left of each button is a variable option menu. An icon to the left of the menu displays the read / write status of the variable; see “Icon Legend” on page 139 for an explanation of these icons. Clicking on the small highlighting buttons opens an Original Source window that displays each instance of the variable within the loop in highlighted form.

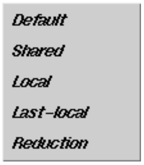


Figure 4-27 Variable Type Option Menu

An option menu (see Figure 4-27) allows you to select the variable type. The choices are as follows:

Shared	One copy of the variable is used by all threads of the MP process.
Local	Each thread has its own copy of the variable.
Last-local	Similar to Local, except the value of the variable after the loop will be as the logically-last iteration would leave it.
Reduction	A sum / product / min / max computation of the variable can be done partially in each thread and then combined afterwards.

C$DOACROSS Parallelization Control View

The $DOACROSS Parallelization Control View opens when you select “C$DOACROSS...” from the loop status option menu in the loop information display as shown in Figure 4-28.

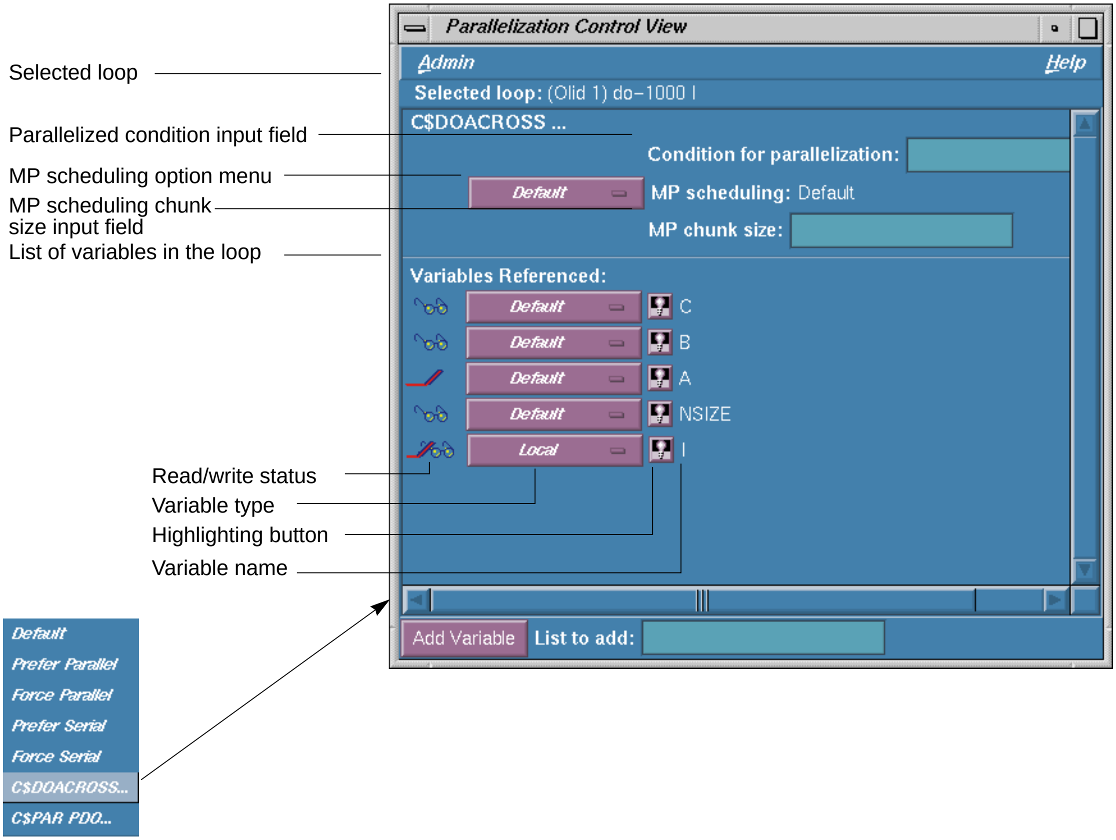


Figure 4-28 C$DOACROSS Parallelization Control View

The C$DOACROSS Parallelization Control View contains the following items:

"Condition for parallelization" text field

Allows you to enter a Fortran conditional statement (for example, `NSIZE .GT. 83`). This statement determines the circumstances under which the loop will be parallelized. The upper right corner of the field changes when you type in the field. Your entry must be syntactically correct; it is not checked.

MP Scheduling menu
: Allows you to alter a loop's scheduling scheme by changing the C$MP_SCHEDTYPE and C$CHUNK directives. See "Parallelization Control View MP Scheduling Option Menu" on page 129 for futher information.

"MP chunk size" text field
: Allows you to set the CHUNK size for the scheduling scheme you select. See "MP Scheduling Chunk Size Field" on page 124 for further information.

Variable Option menus
: Allows you to select the variable type. See "Parallelization Control View Variable Option Menus" on page 130 for further information.

C$PAR PDO Parallelization Control View

The C$PAR PDO Parallelization Control View opens when "C$PAR PDO..." is selected from the loop status option menu in the loop information display as shown in Figure 4-29.

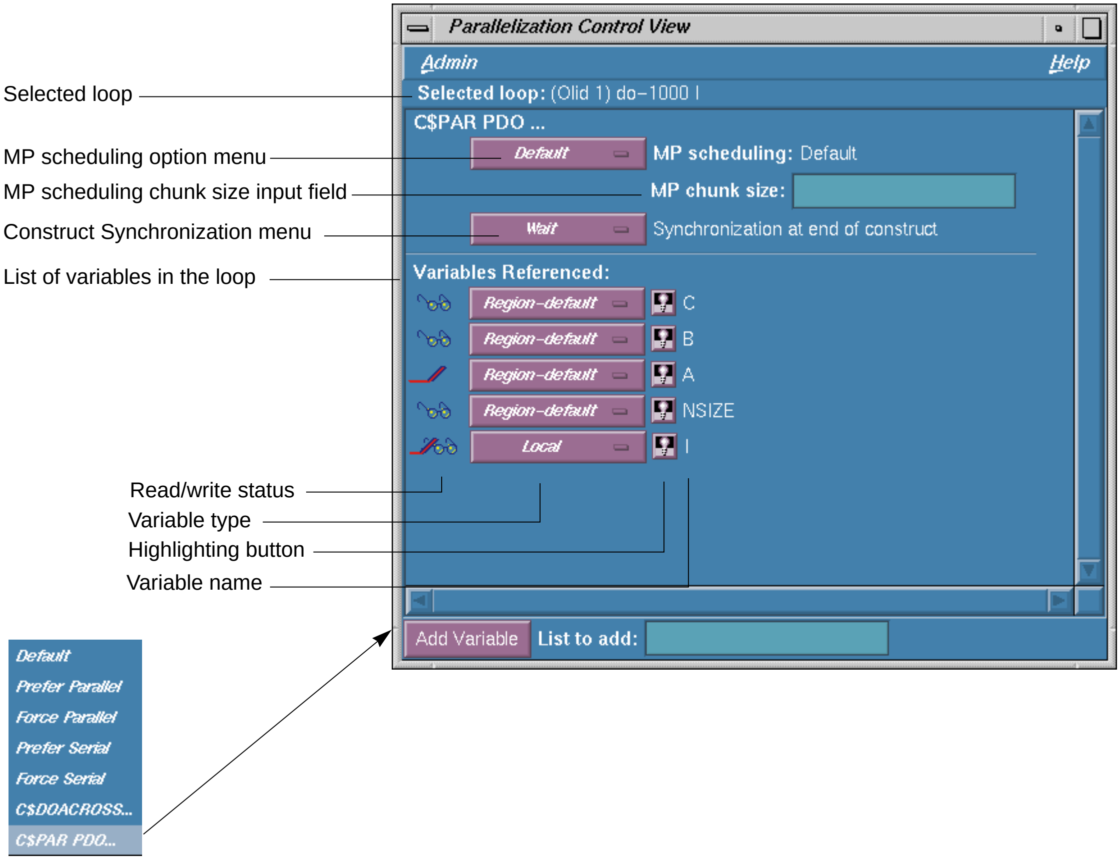


Figure 4-29 C$PAR PDO Parallelization Control View

The C$PAR PDO Parallelization Control View contains the following items:

MP Scheduling menu
: Allows you to alter a loop's scheduling scheme by changing the C$MP_SCHEDTYPE and C$CHUNK directives. See "Parallelization Control View MP Scheduling Option Menu" on page 129 for futher information.

"MP chunk size" text field
: Allows you to set the CHUNK size for the scheduling scheme you select. See "MP Scheduling Chunk Size Field" on page 124 for further information.

Figure 4-30 Synchronization Construct Menu

Synchronization Construct menu

Allows you to set the synchronization at the end of the construct to either "Wait" or "No Wait."

Variable Option menus

Allows you to select the variable type. See "Parallelization Control View Variable Option Menus" on page 130 for further information.

Transformed Loops View

The Transformed Loops View (see Figure 4-31) contains information about how each loop selected from the loop list display is rewritten by PFA into one or more *transformed loops*. You can open this view by pulling down the Views menu of the Parallel Analyzer View and selecting the "Transformed Loops View" command (see "Views Menu" on page 107).

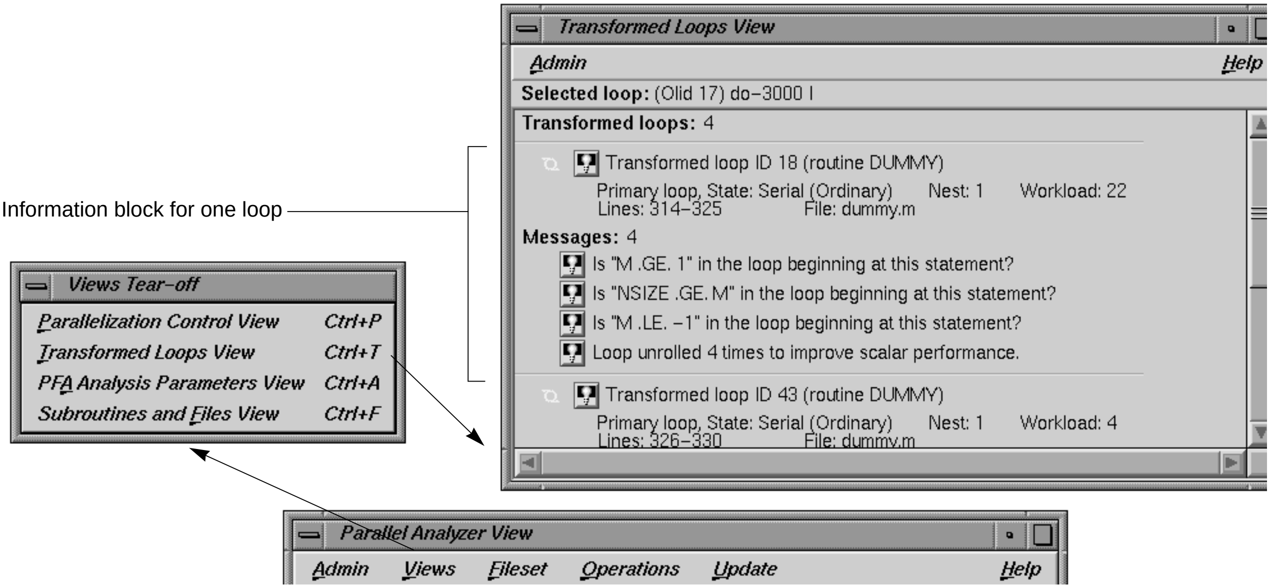


Figure 4-31 Transformed Loops View

Each transformed loop is displayed in its own section of the scrolling display within the Transformed Loops View. Each transformed loop has a highlighting button associated with it. This button is directly to the right of the parallelization icon describing the loop's parallelization status. Left-clicking on this button opens the Transformed Source window (if necessary), showing the original loop and the selected transformed loop.

The next two lines describe the transformed loop, listing the following information about it:

- whether it is a *primary loop* (directly transformed from the selected original loop) or a *secondary loop* (transformed from a different original loop but incorporating some code from the selected original loop)
- its parallelization status
- whether it is an ordinary loop or an interchanged loopits nesting levelits workloadthe corresponding lines in the transformed source
- the name of the file in which it is located

In addition to this information, each transformed loop also may list one or more messages, which and are presented with small highlighting buttons to the left of each message. These are messages from PFA describing some aspect of the loop transformation. Left-clicking on a message button opens an Original Source window showing the original, untransformed loop and highlighting the line of the loop to which the message corresponds.

PFA Analysis Parameters View

The PFA Analysis Parameters View contains a list of PFA execution parameters accompanied by fields into which you can enter new values for the parameters. When you update a source file, any PFA parameters you alter will be changed for that file. See Figure 4-32. For a description of the changed-entry indicators, see “MP Scheduling Chunk Size Field” on page 4-124.

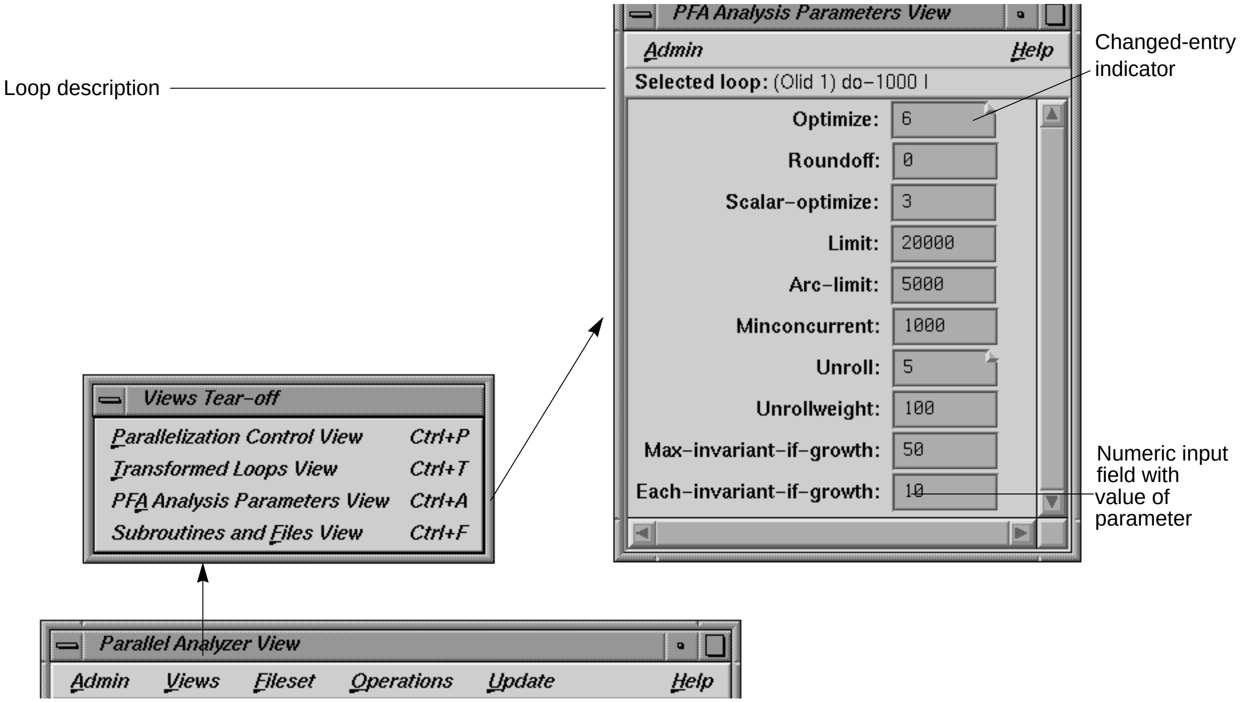


Figure 4-32 PFA Analysis Parameters View

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

Subroutines and Files View

The Subroutines and Files View contains a list from the file in the current session of the Parallel Analyzer View as shown in Figure 4-33. Below each file listing is an indented list of the Fortran subroutines in each file. You can select any file or subroutine by left-clicking on it. You can delete or save changes to a file selected in this view by subsequently selecting the appropriate item from the Parallel Analyzer View menu bar. If a file has been scanned correctly or a subroutine has no errors, a green check mark appears to the left of the file or subroutine listing. If any changes have been made to loops in the file using the Parallel Analyzer View, a red plus sign is above the green check mark to the left of the file listing. If a file could not be scanned or a subroutine had errors, a red international "not" symbol replaces the check mark, denoting an error.

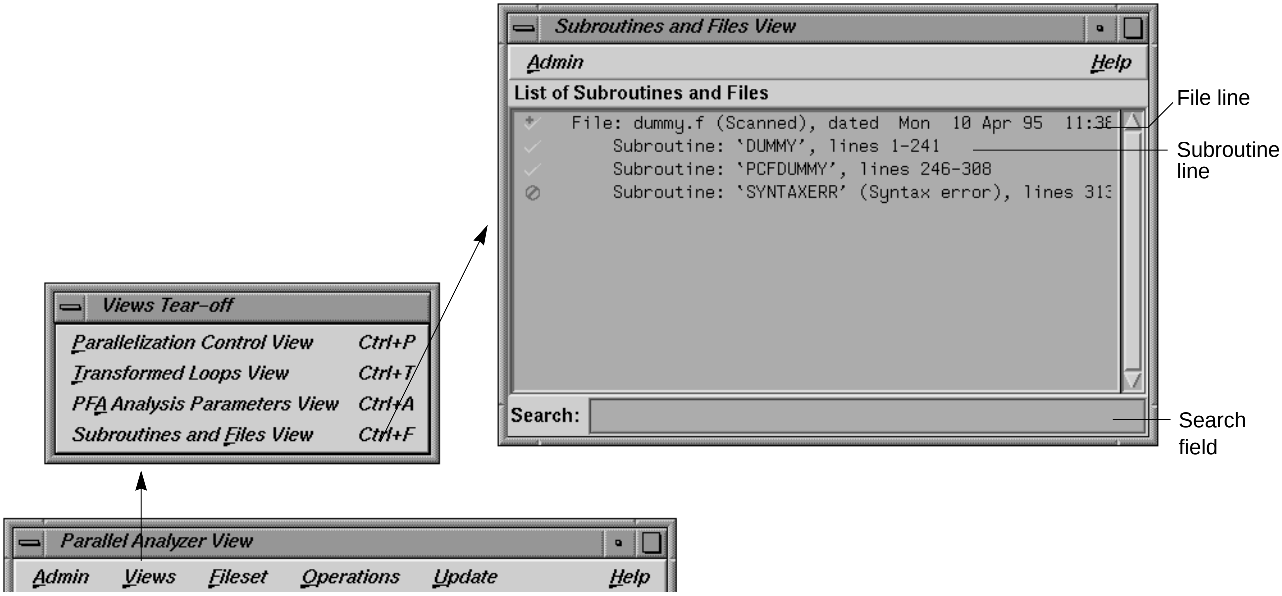


Figure 4-33 Subroutines and Files View

If filtering by file or by subroutine is selected from the filtering option menu in the Parallel Analyzer View (see "Filtering Option Menu" on page 119), double-clicking on a file or subroutine from the Subroutines and Files view will automatically insert the name into the appropriate filtering text field; if that choice is currently selected from the filtering option menu, the loop list is rescanned.

The Search field matches against subroutine and file names listed in the Subroutines and Files View. The matching occurs as you type; the first name in the list that matches what has already been typed is selected. If there is no match, the system will beep in response.

Original and Transformed Source Windows

The Original Source window and the Transformed Source window together present a before and after view of the source code. The former is a view of the source before PFA has run on it, the latter is a view of the source after PFA has parallelized it as shown in Figure 4-34. The two windows use the WorkShop Source View interface.

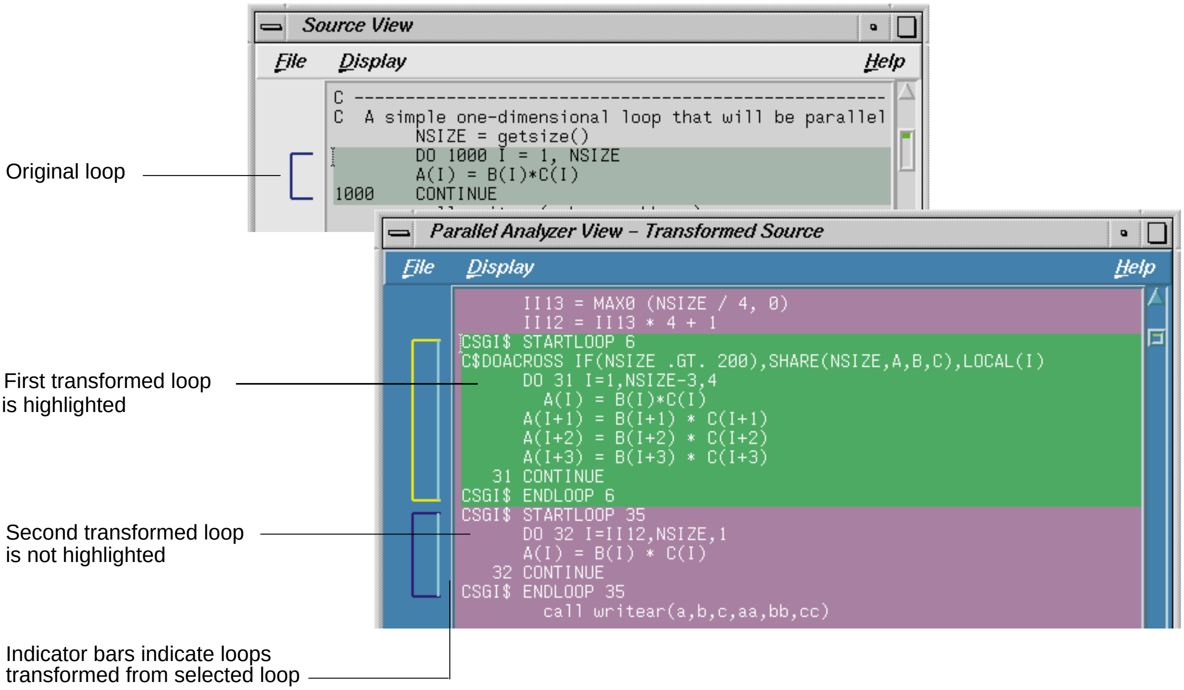


Figure 4-34 Original and Transformed Loop Source Windows

Both the Original Source 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 selects and highlights the corresponding loop.

In a Transformed Source window, an indicator bar (vertical line in a different color) indicates each loop that was transformed from the selected original loop.

If the source windows are invoked from a session linked to the WorkShop Performance Analyzer (see “Launch Tool Submenu” on page 104), any displayed sources files known to the Performance Analyzer will be annotated with performance data.

Icon Legend

The Icon Legend dialog box provides a key explaining the meaning of the icons that appear in the Parallel Analyzer View, the Transformed Loops View, the Subroutines and Files View, and Custom DOACROSS Dialog box. See Figure 4-35.

Icon Legend Buttons

The Icon Legend also contains two buttons, described below.

Close closes the Icon Legend.

Help opens the WorkShop Help Viewer for on-line help in using the Icon Legend.

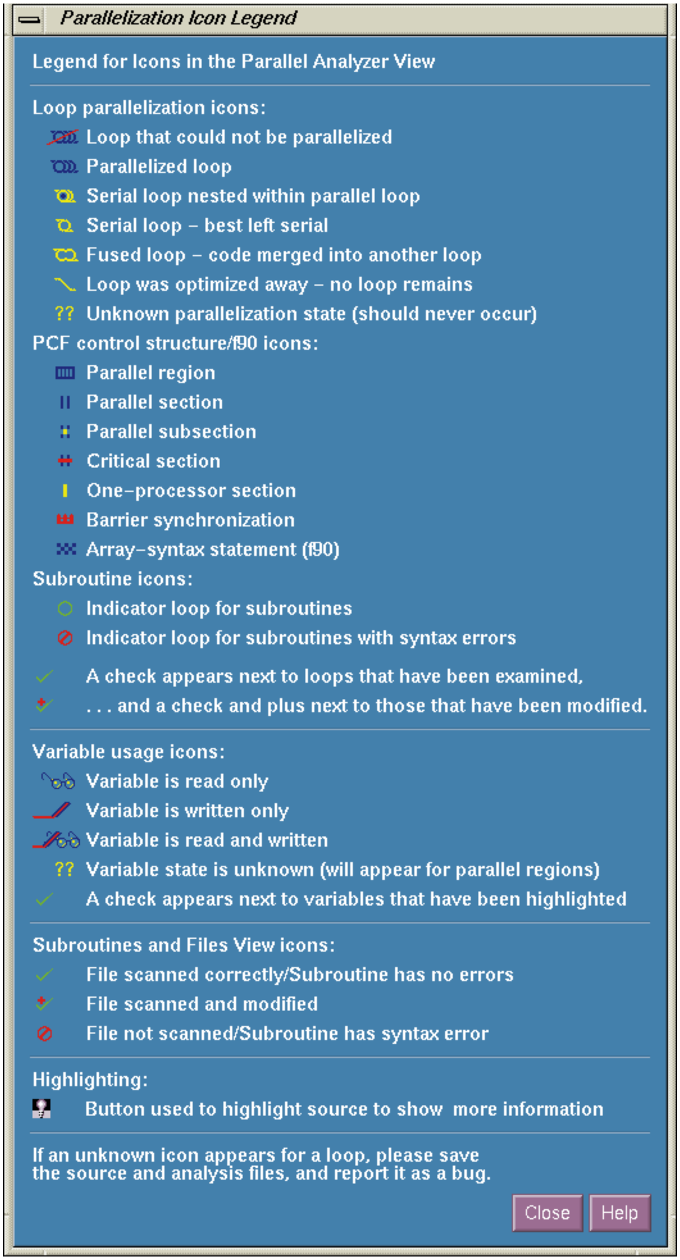


Figure 4-35 Parallelization Icon Legend

Index

N

O

P

U

V

W

X

We'd Like to Hear From You

As a user of Silicon Graphics documentation, your comments are important to us. They help us to better understand your needs and to improve the quality of our documentation.

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

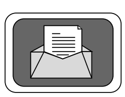
- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please include the title and part number of the document you are commenting on. The part number for this document is 007-2603-001.

Thank you!

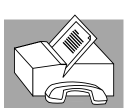
Three Ways to Reach Us

The **postcard** opposite this page has space for your comments. Write your comments on the postage-paid card for your country, then detach and mail it. If your country is not listed, either use the international card and apply the necessary postage or use electronic mail or FAX for your reply.



If **electronic mail** is available to you, write your comments in an e-mail message and mail it to either of these addresses:

- If you are on the Internet, use this address: techpubs@sgi.com
- For UUCP mail, use this address through any backbone site: *[your_site]*!sgi!techpubs



You can forward your comments (or annotated copies of manual pages) to Technical Publications at this **FAX** number:

415 965-0964