# Chapter 4.
# Examining Loops for C Code

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

- Fortran 77, see Chapter 2.
- Fortran 90, see Chapter 3.

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

The following sections comprise the C parallel analyzer session:

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

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

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

## 4.1 Setting Up the `c_tutorial.c` Sample Session

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

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

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

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

## 4.2 Compiling the Sample Code

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

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

Doing this creates the following file:

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

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

# 4.3 Starting the Parallel Analyzer View Tutorial

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

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

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

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

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

  ```
  % make clean
  ```

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

# 4.4 Examples of Simple Loops

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

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

Two other sections discuss more complicated loops:

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

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

## 4.4.1 Simple Parallel Loop

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

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

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

Move to the next loop by selecting Olid 6.

### 4.4.2 Serial Loop

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

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

Move to the Olid 2 loop.

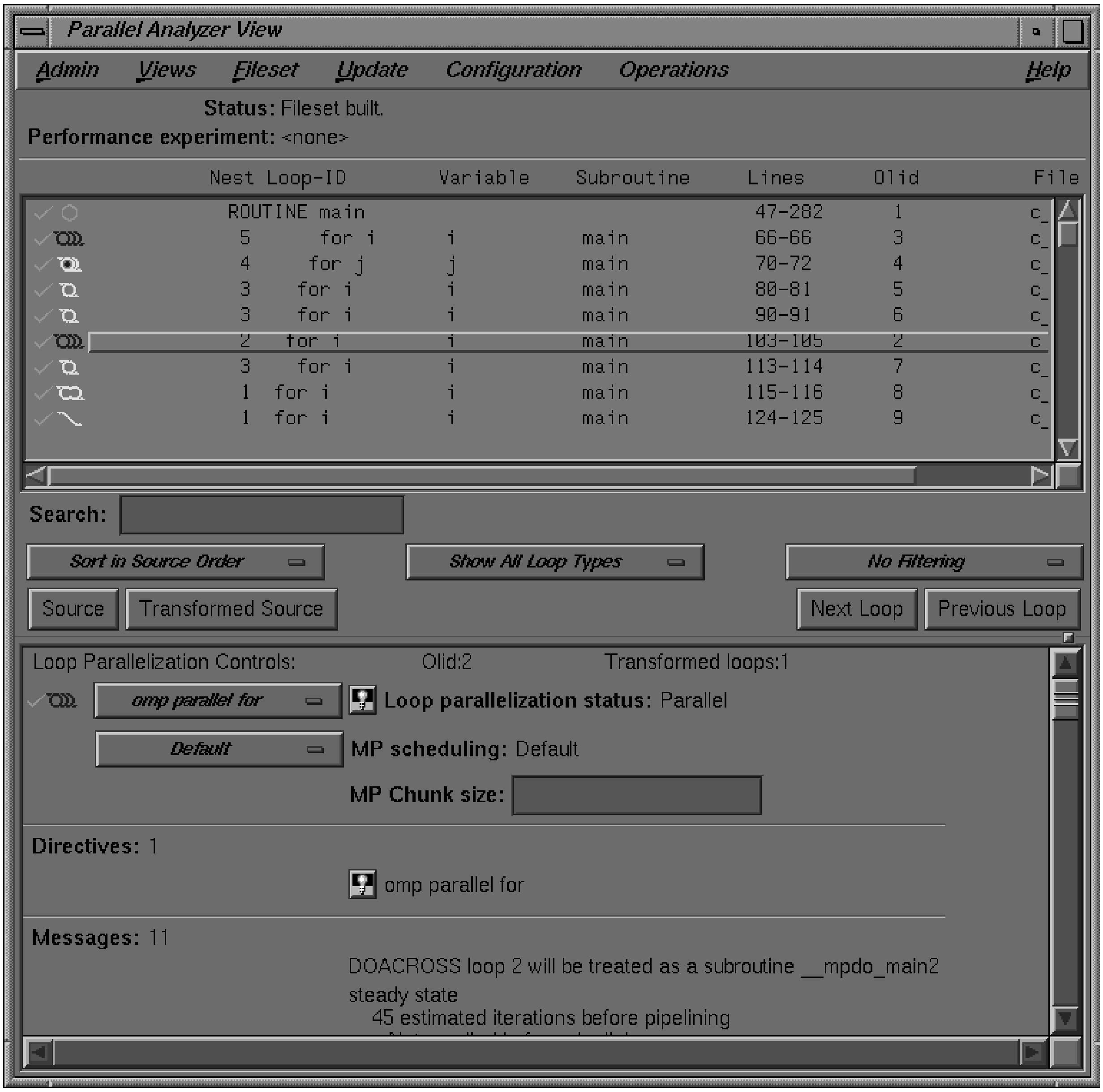
### 4.4.3 Explicitly Parallelized Loop

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

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

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

Figure 34. Explicitly Parallelized Loop




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

## 4.4.4 Fused Loops

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

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

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

### 4.4.5 Loop That Is Eliminated

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

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

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

## 4.5 Examining Loops With Obstacles to Parallelization

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

- Carried Data Dependence, seeSection 4.5.1.
- Input/Output Operations, see Section 4.5.2.
- Function Calls, see Section 4.5.3.
- Permutation Vectors, see Section 4.5.4.

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

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

### 4.5.1 Carried Data Dependence

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

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

#### 4.5.1.1 Unparallelizable Carried Data Dependence

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

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

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

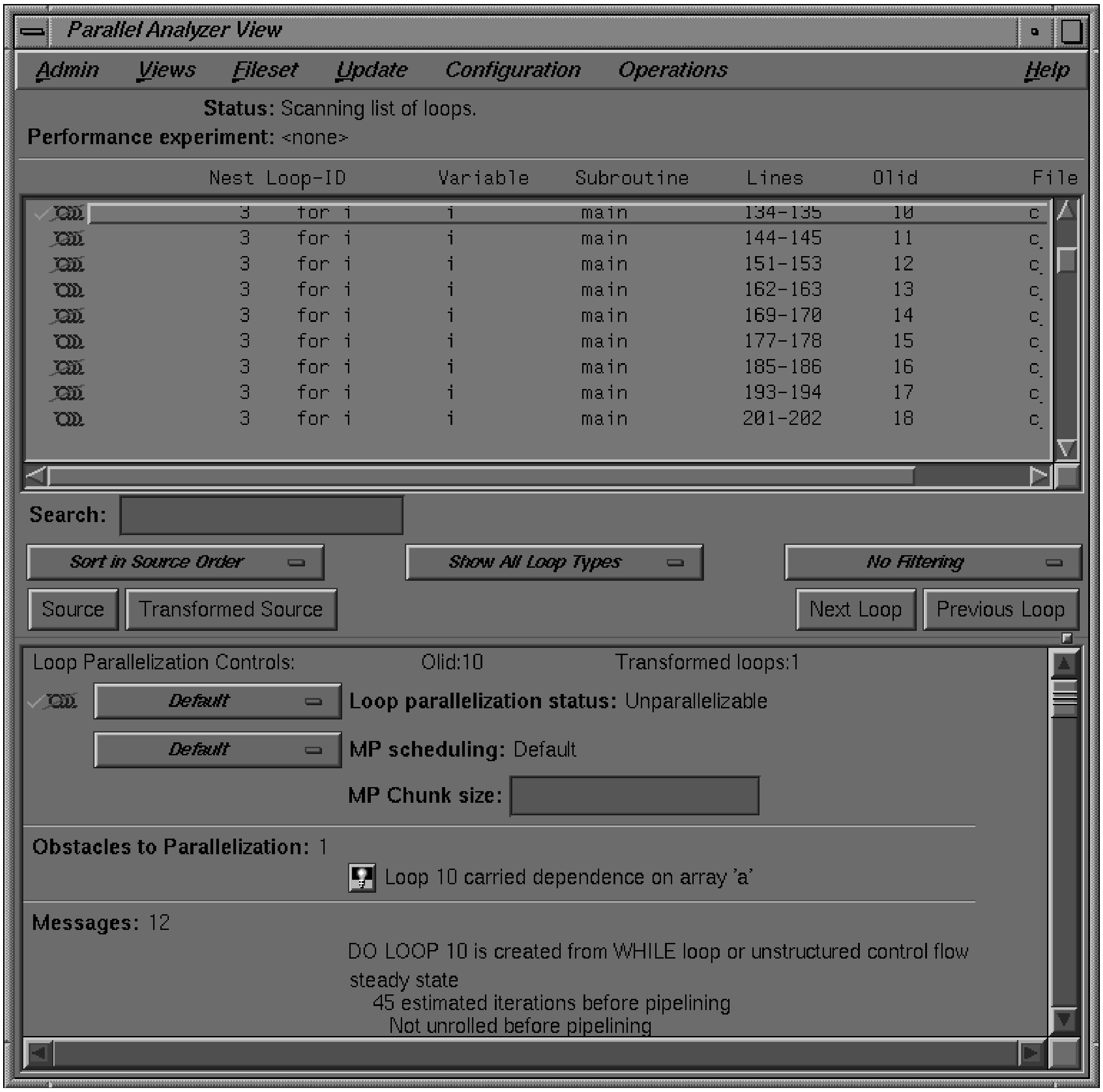
The loop information display in Figure 35, lists the obstacle to parallelization.

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

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

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

Figure 35. Obstacles to Parallelization




## 4.5.1.2 Parallelizable Carried Data Dependence

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

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

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

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

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

#### 4.5.1.3 Multi−line Data Dependence

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

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

Click the highlight button on the obstacle line.

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

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

#### 4.5.1.4 Reductions

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

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

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

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

### 4.5.2 Input/Output Operations

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

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

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

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

### 4.5.3 Function Calls

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

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

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

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

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

### 4.5.4 Permutation Vectors

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

#### 4.5.4.1 Unparallelizable Loop With a Permutation Vector

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

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

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

#### 4.5.4.2 Parallelizable Loop With a Permutation Vector

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

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

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

## 4.6 Examining Nested Loops

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

### 4.6.1 Doubly Nested Loop

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

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

Click `Next Loop` to move to Olid 19.

### 4.6.2 Doubly Nested Loop

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

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

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

### 4.6.3 Triple Nested Loop

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

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

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

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

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

## 4.7 Modifying Source Files and Compiling

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

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

There are two steps to modifying source files:

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

### 4.7.1 Making Changes

You make changes by one of the following actions:

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

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

These are the changes discussed:

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

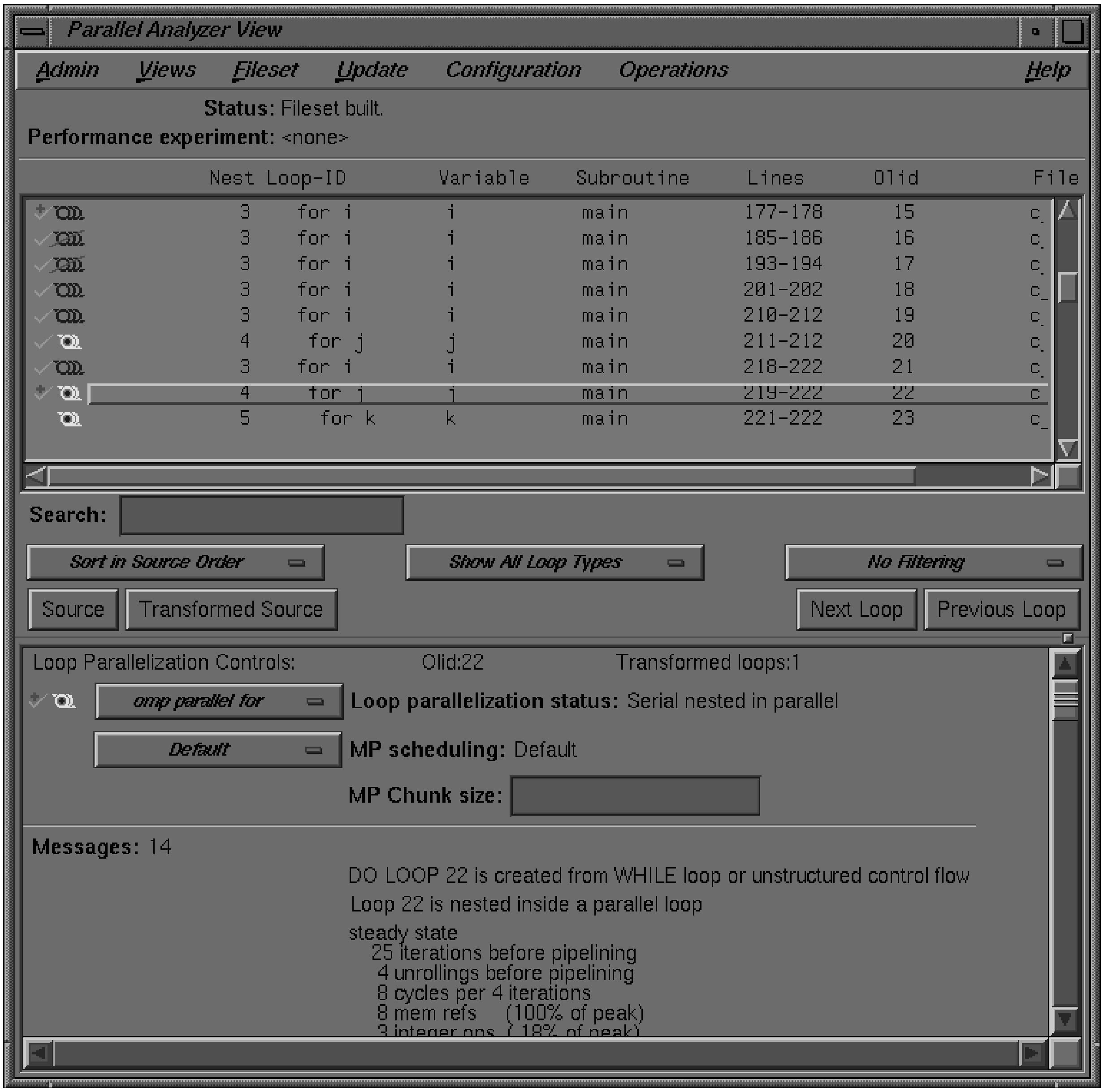
#### 4.7.1.1 Adding `#pragma omp parallel for` Directives and Clauses

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

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

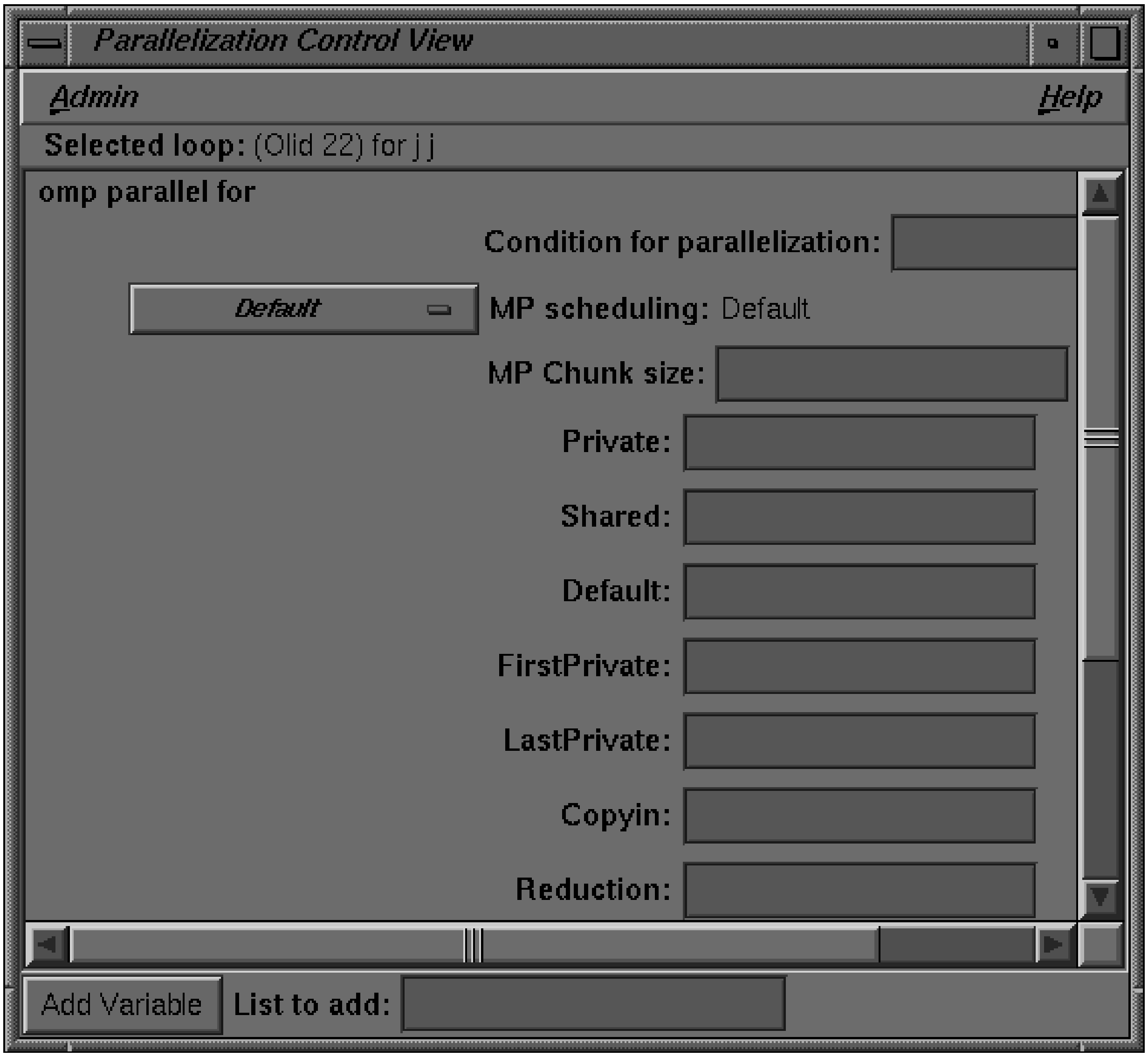
1. Make sure loop Olid 22 is selected.
2. Click on the loop parallelization status option button (see Figure 36) and choose `omp parallel for` to parallelize Olid 22.

Figure 36. Creating a Parallel Directive



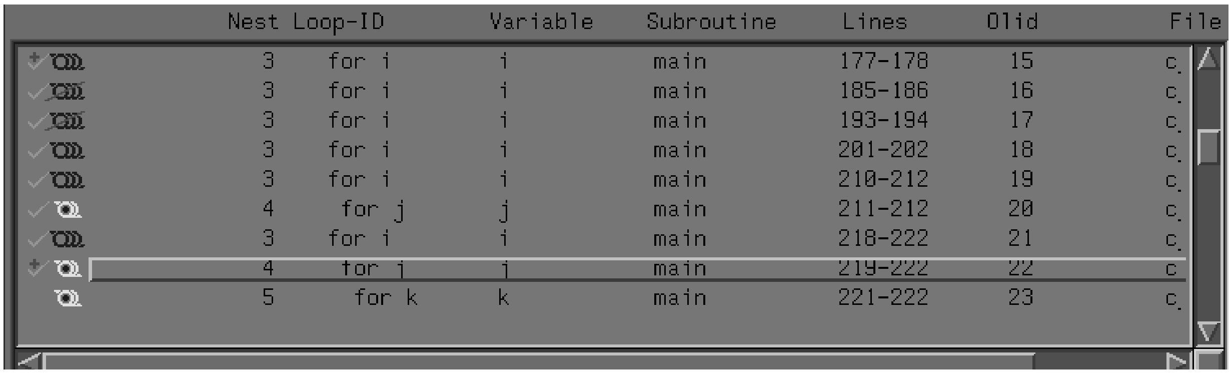

This sequence requests a change in the source code and opens the `Parallelization Control View` (see Figure 37). You can now look at variables in the loop and attach clauses to the directive, if needed.

Figure 37. `Parallelization Control View`




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

Figure 38. Changed Loop List




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

#### 4.7.1.2 Adding New Assertions or Directives With the Operations Menu

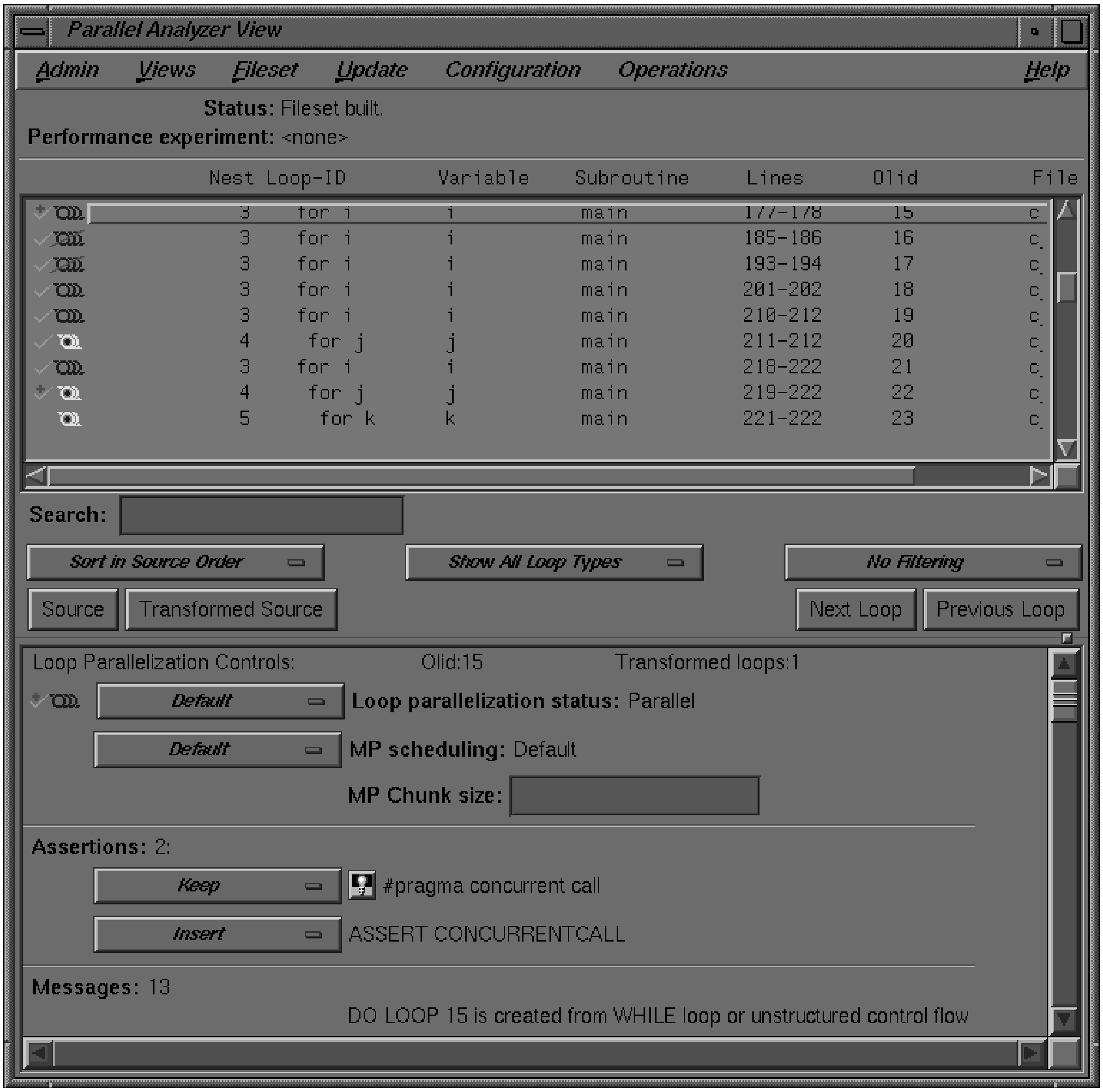
To add a new assertion to a loop, do the following:

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

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

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

Figure 39. Adding an Assertion




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

### 4.7.1.3 Deleting Assertions or Directives

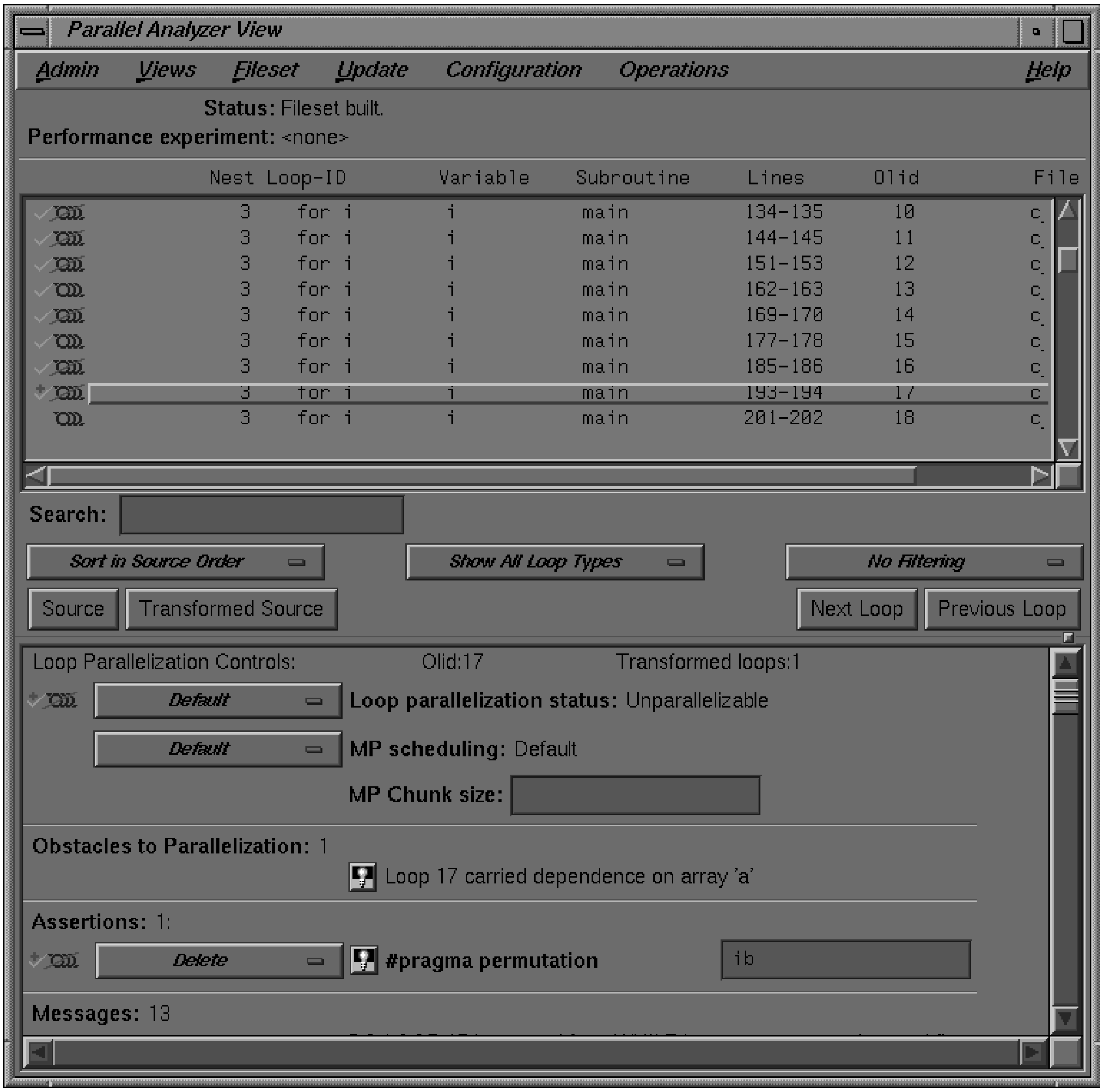
Move to loop Olid 17 (shown in Section 4.5.4.2).

To delete an assertion, follow these steps:

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

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

Figure 40. Deleting an Assertion




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

## *4.7.2 Updating the Source File*

Choose `Update > Update All Files` to update the source file to include the changes made in this tutorial.

Alternatively, you can use the keyboard shortcut for this operation, **Ctrl+U**, with the cursor anywhere in the main view.

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

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

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

## *4.7.3 Examining the Modified Source File*

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

### 4.7.3.1 Added Assertion

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

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

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

### 4.7.3.2 Deleted Assertion

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

# 4.8 Examples Using OpenMP Directives

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

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

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

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

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

## *4.8.1 Explicitly Parallelized Loops: #pragma omp for*

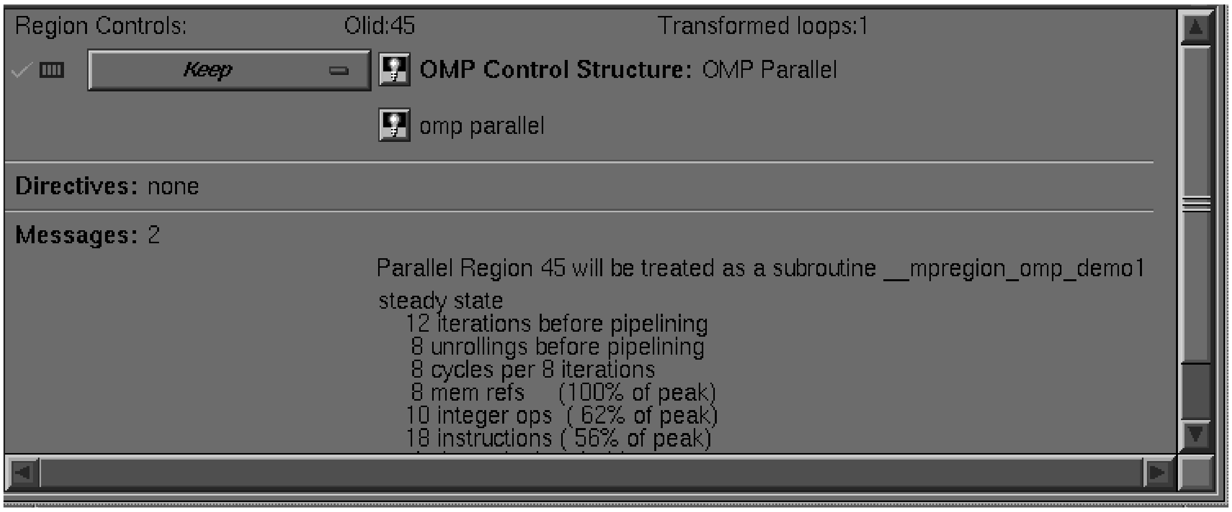
The omp_demo function declares a parallel region containing three loops, the third of which is nested in the second. The first two loops are explicitly parallelized with #pragma omp for directives.

```
#pragma omp parallel shared(a,b)  
{  
#pragma omp for schedule(dynamic,10−2*2)  
```

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

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

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




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

## 4.8.2 Loops With Barriers: `#pragma omp barrier`

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

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

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

## 4.8.3 Critical Sections: `#pragma omp critical`

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

```
#pragma omp for  
```

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

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

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

### 4.8.4 Single–Process Sections: `#pragma omp single`

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

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

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

### 4.8.5 Parallel Sections: `#pragma omp sections`

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

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

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

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

## 4.9 Examples Using Data Distribution Directives

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

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

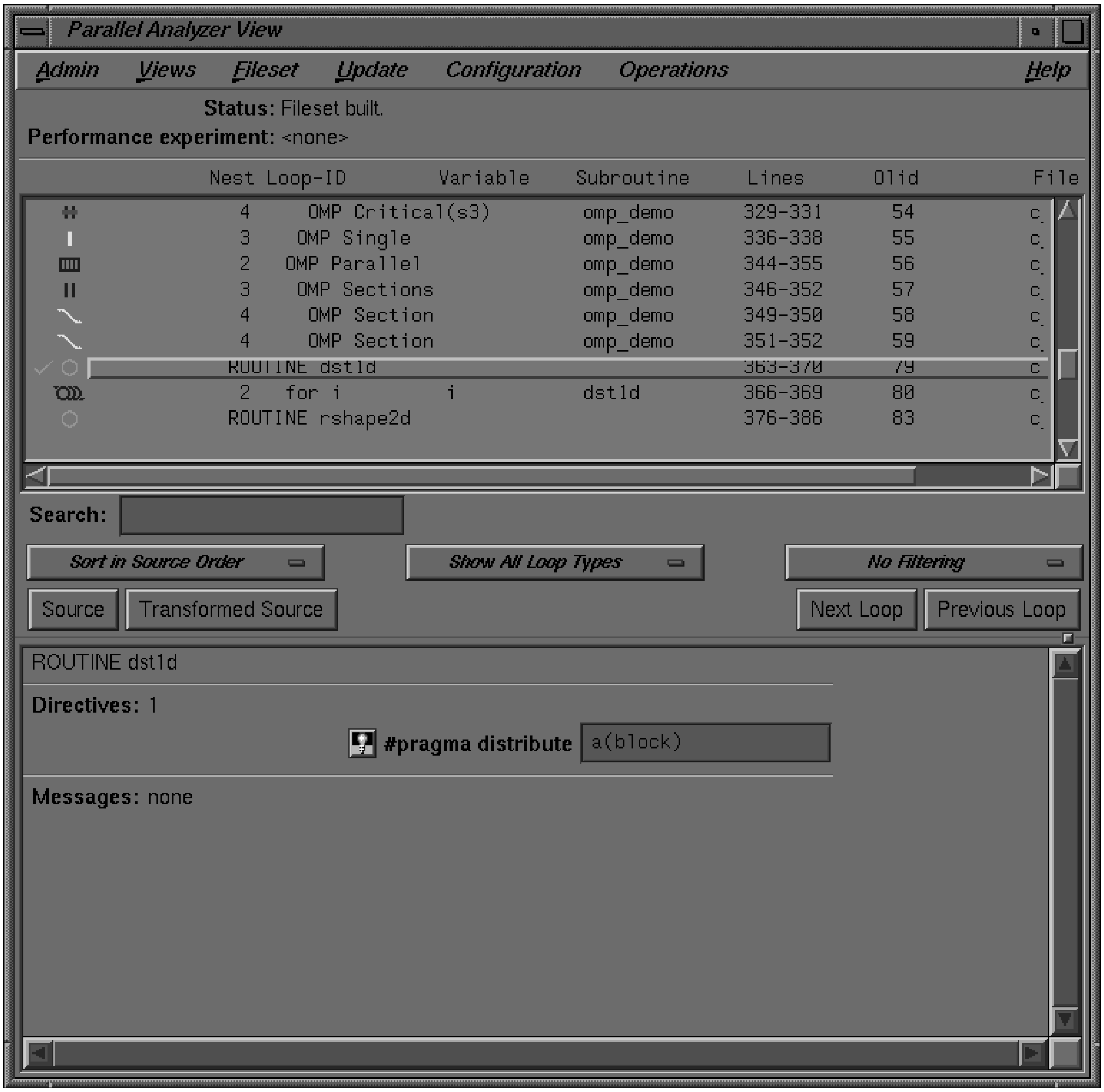
Brief descriptions of these directives appear in Table 3.

### 4.9.1 Distributed Arrays: `#pragma distribute`

When you select the function `dst1d()`, a parallelized loop icon is listed in the loop information display. The

`#pragma distribute` directive specifies placement of array members in distributed, shared memory. (See Figure 42.)

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




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

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

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

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

```
  
}
```

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

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

### *4.9.2 Distributed and Reshaped Arrays:* ***#pragma distribute_reshape***

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

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

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

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

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

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

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

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

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

### *4.9.3 Prefetching Data From Cache:* ***#pragma prefetch_ref***

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

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

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

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

# 4.10 Exiting From the Sample Session

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

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

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

```
% make clean
```