

## *Tools for Enhancing MT Programs*

---



Sun provides several tools for enhancing the performance of MT programs. This chapter describes three of them.

### Thread Analyzer

Thread Analyzer displays standard profiling information for each thread in your program. Additionally, Thread Analyzer displays metrics specific to a particular thread (such as Mutex Wait Time and Semaphore Wait Time).

Thread Analyzer can be used with C, C++, and FORTRAN 77 programs.

### LockLint

LockLint verifies the consistent use of mutex and readers/writer locks in multithreaded ANSI C programs.

LockLint performs a static analysis of the use of mutex and readers/writer locks, and looks for inconsistent use of these locking techniques. In looking for inconsistent use of locks, LockLint detects the most common causes of data races and deadlocks.

### LoopTool

LoopTool, along with its sister program LoopReport, profiles loops for FORTRAN programs; it provides information about programs parallelized by SPARCompiler FORTRAN MP. LoopTool displays a graph of loop runtimes, shows which loops were parallelized, and provides compiler hints as to why a loop was not parallelized.

LoopReport creates a summary table of all loop runtimes correlated with compiler hints about why a loop was not parallelized.

This chapter presents scenarios showing how each tool is used:

- Scenario One looks at Mandelbrot, a C program that can be made to run much faster by making it multithreaded. The discussion analyzes the program with Thread Analyzer to see where performance bottlenecks take place, then threads it accordingly.
- Scenario Two (page 171) shows the use of LockLint to check the Mandelbrot program's use of locks.
- Scenario Three (page 176) shows the use of LoopTool to parallelize portions of a library.

### *Scenario: Threading the Mandelbrot Program*

This scenario shows

1. Threading a program to achieve better performance.
2. Examining the program with Thread Analyzer to determine why it hasn't shown optimal speed-up.
3. Re-writing the program to take better advantage of threading.

Mandelbrot is a well-known program that plots vectors on the plane of complex numbers, producing an interesting pattern on the screen.

In the simplest, nonthreaded version of Mandelbrot, the program flow simply repeats this series:

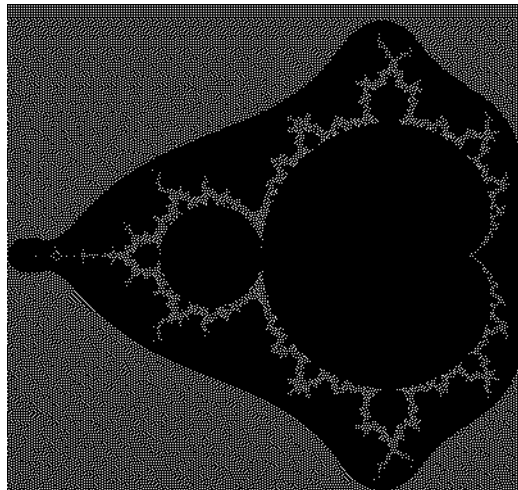
- Calculate each point
- Display each point

Obviously, on a multiprocessor machine this is not the most efficient way to run the program. Since each point can be calculated independently, the program is a good candidate for parallelization.

The program can be threaded to make it more efficient. This time, several threads (one for each processor) are running simultaneously. Each thread calculates and displays a row of points independently.

Thread One	Thread Two
Calculate row	Calculate row
Display row	Display row

However, even though the threaded Mandelbrot is faster than the unthreaded version, it doesn't show the performance speedup that might be expected.



### *Using Thread Analyzer to Evaluate Mandelbrot*

The Thread Analyzer is used to see where the performance bottlenecks are occurring. One thing to check is which procedures were waiting on locks.

After recompiling the program to instrument it for Thread Analyzer, it is loaded.

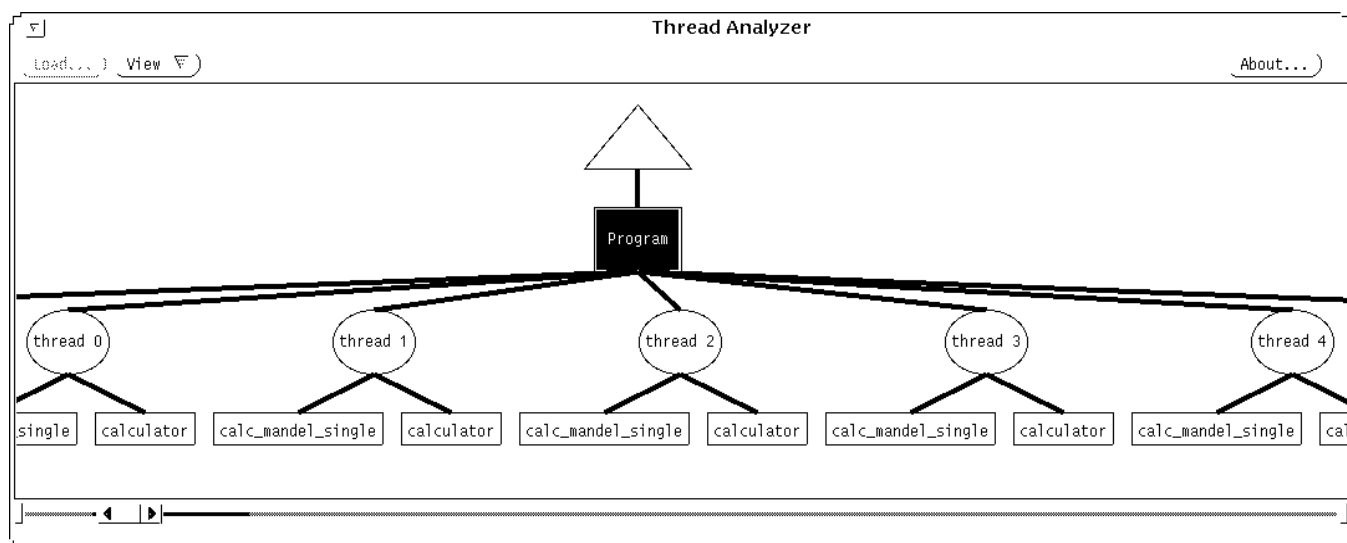


Figure 8-1 Thread Analyzer Main Window (partial)

The main window displays the program's threads and the procedures they call.

Thread Analyzer allows you to view the program in many ways, including the following:

Table 8-1 Thread Analyzer Views

View	Meaning
Graph	Plot the value of selected metrics against wallclock time
gprof(1) Table	Display call-graph profile data for threads and functions
prof(1) Table	Display profile data for program, threads, and functions
Sorted Metric Profile Table	Display a metric for a particular aspect of the program

Table 8-1 Thread Analyzer Views

View	Meaning
Metric Table	Show multiple metrics for a particular thread or function
Filter Threads by CPU	Display the threads whose percent of CPU is equal to or above a designated threshold
Filter Functions by CPU	Display the functions whose percent of CPU is equal to or above a designated threshold

To look at wallclock time and CPU time, choose the Graph view, and select CPU, Wallclock time, and Mutex Wait metrics:

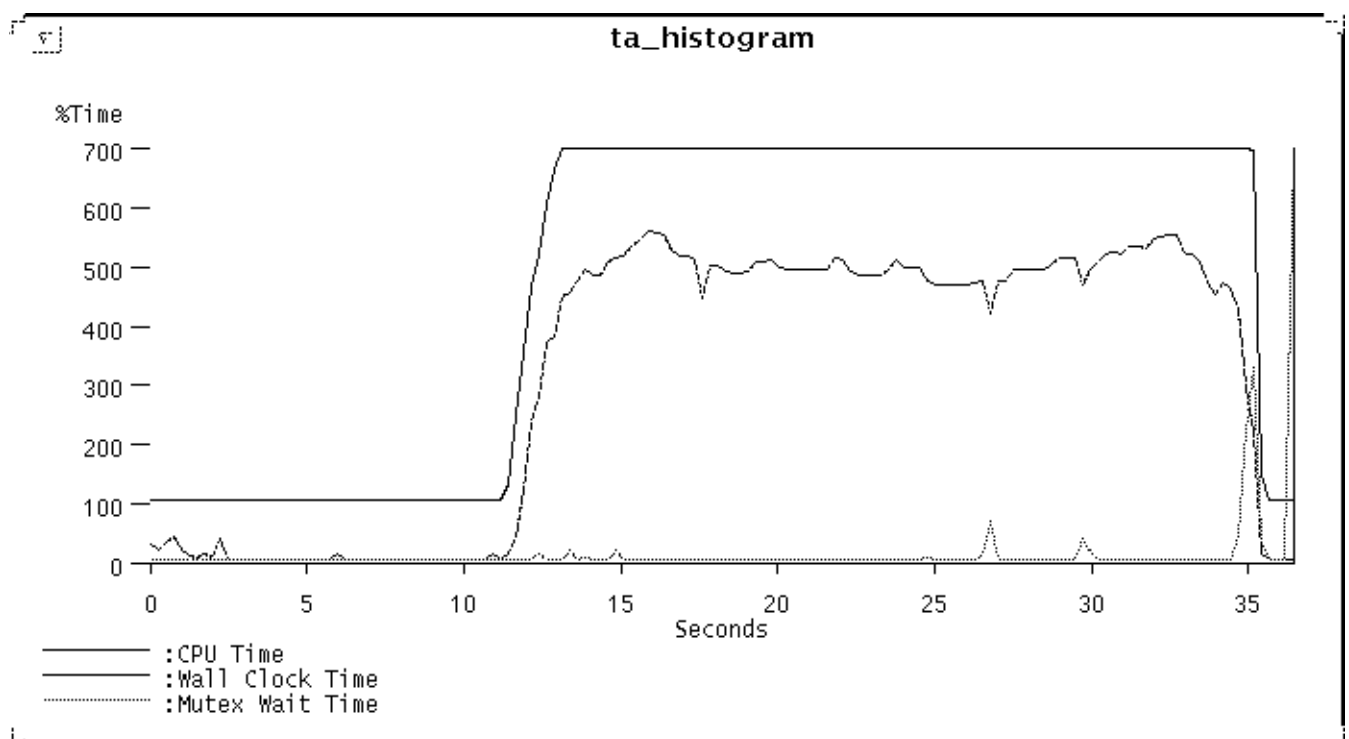
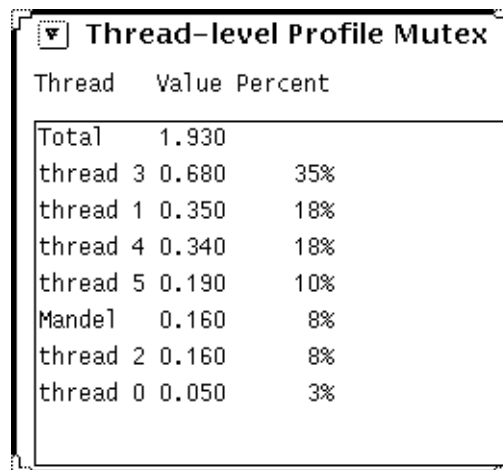


Figure 8-2 Thread Analyzer: Wall-clock and CPU time

According to this graph, CPU time is consistently below wallclock time. This indicates that fewer threads than were allocated are being used, meaning that some threads are blocked (that is, contending for resources).

Look at mutex wait times to see which threads are blocked. To do this, you can select a thread node from the main window, and then **Mutex Wait** from the **Sorted Metrics** menu. The table displays the amount of time each thread spent waiting on mutexes:



Thread	Value	Percent
Total	1.930	
thread 3	0.680	35%
thread 1	0.350	18%
thread 4	0.340	18%
thread 5	0.190	10%
Mandel	0.160	8%
thread 2	0.160	8%
thread 0	0.050	3%

Figure 8-3 Thread Analyzer: Mutex Wait Time

The various threads spend a lot of time waiting for each other to release locks. (The fact that Thread 3 waits so much more than the others is because of randomness.) Because the display is a serial resource — a thread can't display until another thread has finished displaying — the threads are probably waiting for other threads to give up the display lock.

In other words, this is what's happening:

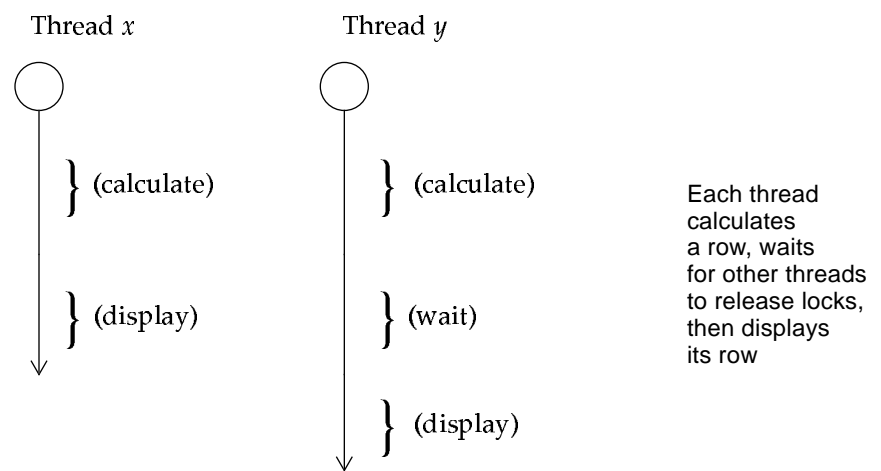


Figure 8-4 Mandelbrot Multithreaded: Each Thread Calculates and Displays

To speed things up further, code can be rewritten so that the calculations and the display are entirely separate. In this version, several threads are simultaneously calculating rows of points and writing into a buffer, while another thread reads from the buffer and displays rows:

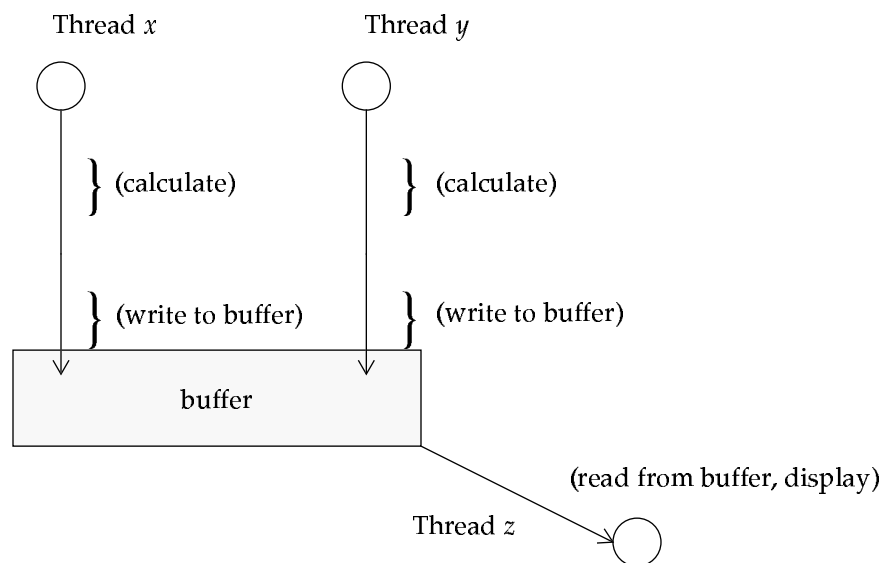


Figure 8-5 Mandelbrot Threaded: Separate Display Thread

Now, instead of the display procedure of each thread waiting on another thread to calculate and display, only the display thread waits (for the current line of the buffer to be filled). While it waits, other threads are calculating and writing, so that there is little time spent waiting for the display lock.



The Thread Analyzer confirms this:

Thread	Value	Percent
Total	0.270	
Mandel	0.230	85%
thread 1	0.020	7%
thread 0	0.010	4%
thread 3	0.010	4%

Figure 8-6 Thread Analyzer: Mutex Wait Time (Separate Display Thread)

Now the program spends almost all its time in the main loop (Mandel), and the time spent waiting for locks is reduced significantly. And Mandelbrot runs noticeably faster.

### Scenario: Checking a Program With LockLint

A program can run efficiently but still contain potential problems. One such problem is data that two threads might try to access at the same time. This can lead to

- Deadlocks — when two threads are mutually waiting for the other to release a lock.
- Data races — when two or more threads have overlapping read/write access to data, causing unexpected data values. For example, suppose Thread A writes the variable *calc*, goes off and does something else, and then comes back to read *calc*; in the meantime Thread B writes to *calc* and changes its value to something Thread A does not “expect.”

Here's how you can use LockLint to see if data is adequately protected.

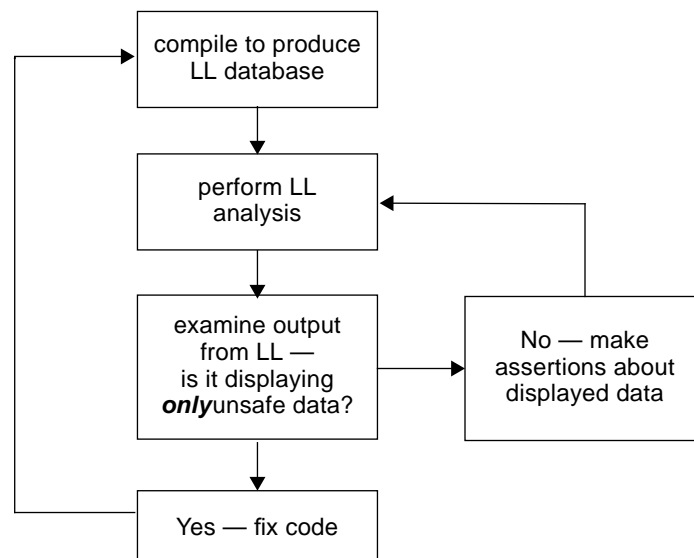


Figure 8-7 The LockLint Usage Pathway

1. **Compile the program with LockLint instrumentation.**  
The compiler has an option to produce a version of the program that LockLint can use for analysis.
2. **Create a LockLint shell and load the instrumented program.**  
You can use this shell as you would any other, including running scripts.
3. **Save the executable's state.**  
LockLint is designed to run iteratively. You run it over and over, making progressively stronger assertions about the data it is analyzing, until you find a problem or are satisfied that the data is safe.

Analyzing the program with LockLint changes its state; that is, once you've done an analysis, you can't add further assertions. By saving and restoring the state, you can run the analysis over and over, with different assertions about the program's data.

4. **Analyze the program.**  
The analyze performs consistency checks on the program's data.

### 5. Search for unsafe data.

Having run the analysis, you can look for unprotected elements.

Here variables are displayed that did not have locks consistently held on them while they were accessed (indicated by the empty brackets); further, an asterisk indicates that these variables were written to. An asterisk, therefore, means that LockLint “believes” the data is not safe.

```
$ lock_lint analyze
$ lock_lint vars -h | grep held
:arrow_cursor          *held={ }
:bottom_row            *held={ }
:box_height            *held={ }
:box_width             *held={ }
:box_x                 *held={ }
:busy_cursor   *held={ }
:c_text   *held={ }
:calc_mandel   *held={ }
:calc_type   *held={ }
:canvas   *held={ }
:canvas_proc/drag   *held={ }
:canvas_proc/x   *held={ }
[ . . . ]
:gap              *held={ }
:gc              *held={ }
:next_row        *held={ }
:now.tv_sec      held={ }
:now.tv_usec     held={ }
:p_text         *held={ }
:panel          *held={ }
:picture_cols   *held={ }
:picture_id     *held={ }
:picture_rows   *held={ }
:picture_state  *held={ }
:pw            *held={ }
:ramp.blue      *held={ }
:ramp.green     *held={ }
:ramp.red       *held={ }
:rectangle_selected*held={ }
:row_inc        *held={ }
:run_button     *held={ }
[ . . . ]
```

Figure 8-8 Fragment of Initial LockLint Output

However, this analysis is only of limited usefulness, because many of the variables displayed do not *need* to be protected, such as variables that aren't written to, except when they're initialized. By excluding some data from consideration, and having LockLint repeat its analyses, you can find only the unprotected variables that you are interested in.

**6. Restore the program to its saved state.**

To be able to run the analysis again, pop the state back to what it was before the program was last analyzed.

**7. Refine the analysis by excluding some data.**

For example, you can ignore variables that aren't written to — since they don't change, they won't cause data races. And you can ignore places where the variables are initialized (if they're not visible to other threads).

You can ignore the variables that you know are safe by making *assertions* about them. In the example below, the following are done:

- Initialization functions are ignored (because no data is overwritten at initialization)
- Some variables are asserted to be read-only

(For clarity's sake this is done on the command line, the long way; you can use aliases and shell scripts to make the task easier.)

```
$ lock_lint ignore CreateXStuff run_proc canvas_proc main
$ lock_lint assert read only bottom_row
$ lock_lint assert read only calc_mandel
etc.
```

**8. Analyze the program again, and search for unsafe data.**  
Now the list of unsafe data is considerably reduced.

```
$ lock_lint vars -h | grep held
:bottom_row          held={ }
:calc_mandel          held={ }
:colors              held={ }
:corner_i            held={ }
:corner_r            held={ }
:display             held={ }
:drawable            held={ }
:frame              held={ }
:gap                 held={ }
:gc                  held={ }
:next_row            held={ }
mandel_display.c:next_row_lock }
:picture_cols        held={ }
:picture_id          held={ }
:picture_rows        *held={ }
:picture_state       *held={ }
:row_inc             held={ }
```

*Figure 8-9* Unsafe Data Reported by LockLint

This time only two variables were written to (`picture_rows` and `picture_state`) and are flagged by LockLint as inconsistently protected.

(The analysis also flags the variable `next_row`, which the calculator threads use to find the next chunk of work to be done. However, as the analysis states, this variable is consistently protected.)

Now you can alter your source code to properly protect these two unsafe variables.

## Scenario: Parallelizing Loops with LoopTool

IMSL™ is a popular math library used by many FORTRAN and C programmers.<sup>1</sup> One of its routines is a good candidate for parallelizing with LoopTool.

This example is a FORTRAN program called `l2trg.f`. (It computes LU factorization of a single-precision general matrix.) The program is compiled without any parallelization; then checked to see how long it takes to run with the `time(1)` command:

```
$ f77 l2trg.f -cg92 -03 -lmsl
$ /bin/time a.out
real      44.8
user      43.5
sys       1.0
```

Figure 8-10 Original Times for `l2trg.f` (Not Parallelized)

To look at the program with LoopTool, recompile with the LoopTool instrumentation, using the `-Zlp` option:

```
$ f77 l2trg.f -cg92 -03 -Zlp -lmsl
```

---

1. IMSL is a registered trademark of IMSL, Inc. This example is used with permission.

This is what LoopTool shows:

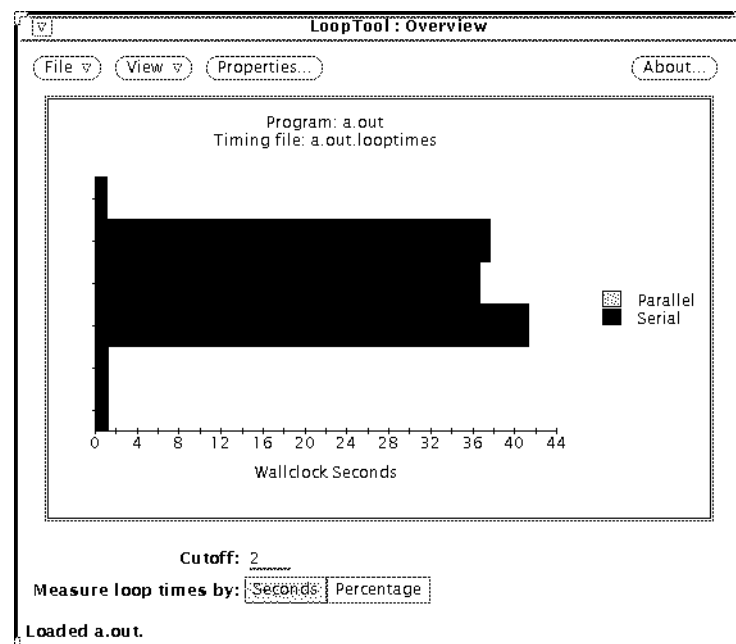


Figure 8-11 LoopTool View Before Parallelization

Putting the cursor over a loop gives its line number; clicking on it brings up a window that displays the loop in the source code. (Contrast the display in this example with the display on page 181.)

Most of the program's time is spent in three loops; looking at the source shows that they are nested. The middle loop gives a hint about parallelization:

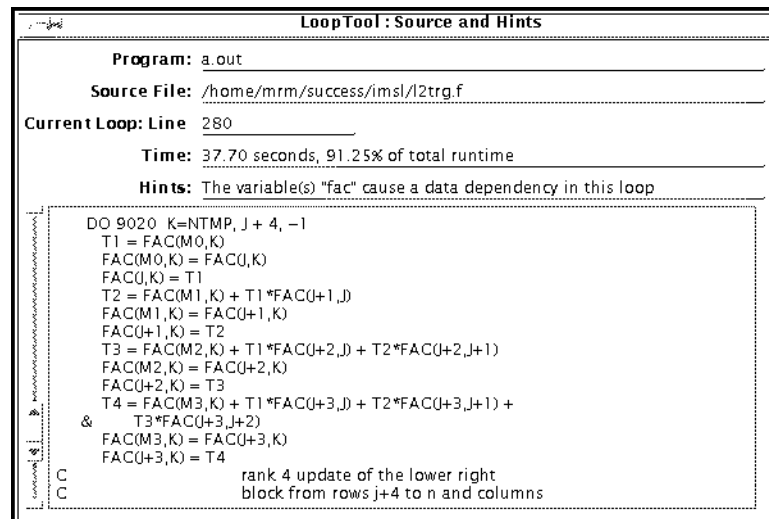


Figure 8-12 LoopTool Hint About Parallelization

In this case, LoopTool gives the message

The variable "fac" causes a data dependency in this loop



And, indeed, looking at the source, you can see that `fac` is calculated in the nested, innermost loop (9030):

```

C                                update the remaining rectangular
C                                block of U, rows j to j+3 and
C                                columns j+4 to n

      DO 9020 K=NTMP, J + 4, -1
        T1 = FAC(M0,K)
        FAC(M0,K) = FAC(J,K)
        FAC(J,K) = T1
        T2 = FAC(M1,K) + T1*FAC(J+1,J)
        FAC(M1,K) = FAC(J+1,K)
        FAC(J+1,K) = T2
        T3 = FAC(M2,K) + T1*FAC(J+2,J) + T2*FAC(J+2,J+1)
        FAC(M2,K) = FAC(J+2,K)
        FAC(J+2,K) = T3
        T4 = FAC(M3,K) + T1*FAC(J+3,J) + T2*FAC(J+3,J+1) +
&          T3*FAC(J+3,J+2)
        FAC(M3,K) = FAC(J+3,K)
        FAC(J+3,K) = T4

C                                rank 4 update of the lower right
C                                block from rows j+4 to n and columns
C                                j+4 to n
      DO 9030 I=KBEG, NTMP
        FAC(I,K) = FAC(I,K) + T1*FAC(I,J) + T2*FAC(I,J+1) +
&          T3*FAC(I,J+2) + T4*FAC(I,J+3)
9030    CONTINUE
9020  CONTINUE

```

The loop index, `I`, of the innermost loop is used to access rows of the array `fac`. So the innermost loop updates the  $I^{\text{th}}$  row of `fac`. Since updating these rows doesn't depend on updates of any other rows of `fac`, it's safe to parallelize this loop.

Therefore, if the calculation of `fac` can be speeded by parallelizing loop 9030, there should be a significant performance improvement. Force explicit parallelization by inserting a `DOALL` directive in front of loop 9030:

```
C$PAR DOALL                                (Add DOALL directive here)
      DO 9030  I=KBEG, NTMP
          FAC(I,K) = FAC(I,K) + T1*FAC(I,J) + T2*FAC(I,J+1) +
&          T3*FAC(I,J+2) + T4*FAC(I,J+3)
9030    CONTINUE
```

Now recompile, forcing explicit parallelization of that loop with the `-explicitpar` option. First, though, make sure to use all the processors on the machine; do that by setting the `PARALLEL` environment variable. Finally, run the program and compare its time with that of the original times (shown in Figure 8-10 on page 176).

```
$ setenv PARALLEL 2                        (2 is the # of processors on
the machine)
$ f77 l2trg.f -cg92 -03 -explicitpar -ims1
$ /bin/time a.out
real        28.4
user        53.8
sys         1.1
```

Figure 8-13 Post-Parallelization Times for `l2trg.f`

The program now runs over a third faster. (The higher number for user reflects the fact that there are now two processes running.) Looking at the program again in LoopTool, you see that, in fact, the innermost loop is now parallel.

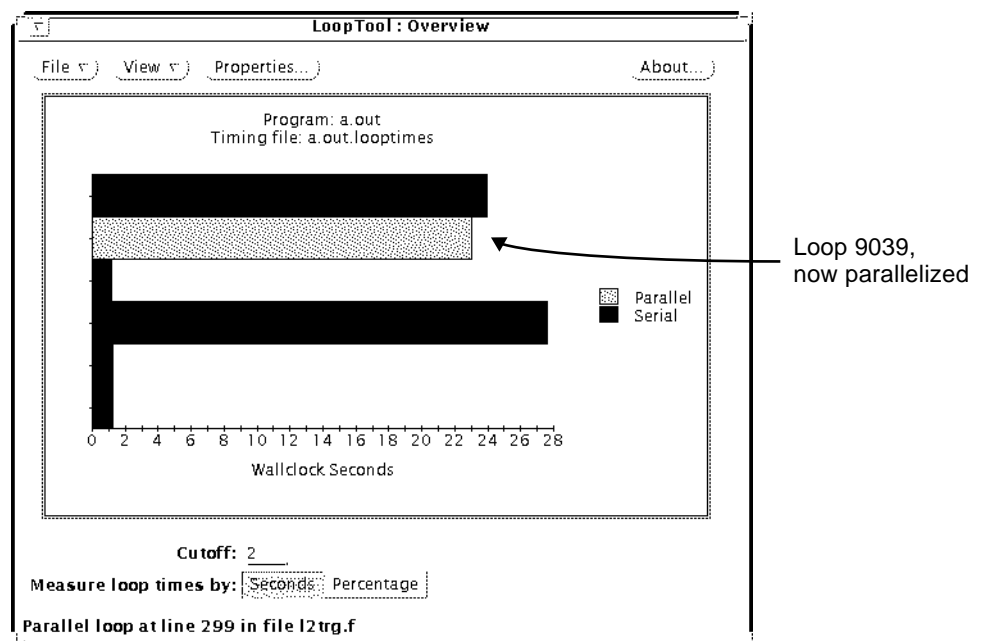


Figure 8-14 LoopTool View After Parallelization

## For More Information

You might be able to find out more about Solaris threads and related issues on the World Wide Web (WWW) at the following URL:

<http://www.sun.com/sunsoft/Products/Developer-products/sig/threads>

Also, the following manuals might be of interest:

*Thread Analyzer User's Guide* p/n 801-6691-10

*LockLint User's Guide* p/n 801-6692-10

*LoopTool User's Guide* p/n 801-6693-10

