`looptool` *User's Guide*

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-6693-10
Revision A, August 1994

© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, Solaris, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark of Novell, Inc., in the United States and other countries; X/Open Company, Ltd., is the exclusive licensor of such trademark. OPEN LOOK® is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOKand Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME

Please
Recycle

Adobe PostScript

Contents

1. Getting Started 1

Quick Overview 1

Compile 2

What does -Zlp do? 3

-loopinfo Switch 4

-Qoption iropt -Dlooptable Switch 5

Run 7

Analyze 7

2. looptool
Graphical User Interface 9

Starting looptool 9

Overview Window 11

File Button 12

View Button 14

Properties Button 15

About Button 16

Source and Hints Window 17

3. loopreport Command Line Interface 19

Starting `loopreport` 19

4. Information Presented by looptool and loopreport 21

Compiler Hints. 21

0 — No hint available 22

1 — Loop contains procedure call 23

2 — Compiler generated two versions of this loop 23

3 — Loop contains data dependency 24

4 — Loop was significantly transformed during optimization 24

5 — Loop may or may not hold enough work to be profitably parallelized. 24

6 — Loop was marked by user-inserted pragma, DOALL. 25

7 — Loop contains multiple exits 25

8 — Loop contains I/O, or other function calls, that are not MT safe 25

9 — Loop contains backward flow of control. 25

10 — Loop may have been distributed 26

11 — Two or more loops may have been fused 26

12 — Two or more loops may have been interchanged . . . 26

Compiler Optimizations and How They Affect Loops 26

Inlining 27

Loop Transformations — Unrolling, Jamming, Splitting, and Transposing 27

Parallel Loops Nested inside Serial Loops 27

Preface

The *looptool User's Guide* describes the `looptool` and `loopreport` tools that provide information about programs parallelized by the SC 3.0.1 Fortran MP compiler.

`looptool` displays a graph of the loop runtimes, shows which loops were parallelized, and provides compiler hints about 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.

Who Should Use This Book

The *looptool User's Guide* describes how to use `looptool` and `loopreport` to gain an understanding of the performance of loops automatically parallelized by the SC3.0.1 Fortran MP compiler. This manual is intended for Fortran MP programmers who are developing or tuning programs parallelized by the SC3.0.1 Fortran MP compiler.

How This Book Is Organized

The *looptool User's Guide* has the following organization.

Chapter 1, "Getting Started," describes the steps to follow to create an instrumented program that you can analyze with `looptool` and `loopreport`.

Chapter 2, "looptool Graphical User Interface," explains how to start looptool and describes how to use the graphical user interface.

Chapter 3, "loopreport Command Line Interface," explains how to start loopreport.

Chapter 4, "Information Presented by looptool and loopreport," lists the compiler hints and describes what they mean.

Glossary is a list of words and phrases found in this book and their definitions.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

Getting Started 1

`looptool` and `loopreport` are performance analysis tools used with the FORTRAN MP compiler. The FORTRAN MP compiler automatically parallelizes those loops that it determines are safe and profitable to parallelize.

`looptool` provides two additional features. It times all the loops, whether serial or parallel. Additionally, `looptool` collects hints from the compiler, during compilation, that are intended to help you parallelize loops that were not parallelized.

This chapter is organized into the following sections:

Quick Overview
Compile
Run
Analyze

Using `looptool` is like using `gprof`. The three steps are: compile, run, analyze.

Quick Overview

Compile:

```
demo% f77 -O4 -parallel -Zlp cholsky.f -o cholsky
```

1

Run:

```
demo% /bin/time cholsky < cholsky.input >& cholsky.output
```

Analyze:

```
demo% loopreport cholsky > cholsky.loopreport
demo% looptool cholsky &
```

The command line interface is named `loopreport`, and the graphical user interface is named `looptool`.

Compile

Compile your f77 program with the FORTRAN MP compiler. To compile for automatic parallelization, typical compilation switches are `-parallel` and `-O4`. To compile for loopview, add `-Zlp`.

```
demo% f77 -O4 -parallel -Zlp cholsky.f
```

Other combinations of compile switches work as well. Either `-O3` or `-O4` can be used with `-parallel`. If you don't specify `-O3` or `-O4` but you do use `-parallel`, then `-O3` is added. The following table summarizes how switches are added.

Table 1-1 Promotion of Compiler Switches

You type:	Bumped Up To:
`-parallel`	`-parallel -O3`
`-parallel -Zlp`	`-parallel -O3 -Zlp`
`-explicitpar`	`-explicitpar -O3`
`-explicitpar -Zlp`	`-explicitpar -O3 -Zlp`
`-Zlp`	`-depend -O3 -Zlp`

The FORTRAN MP compiler switch `-explicitpar` is used with the pragma DOALL. If you insert DOALL before a loop in your source code, you are explicitly marking that loop for parallelization. The compiler will parallelize this loop when you compile with `-explicitpar`.

The following code fragment shows how to mark a loop explicitly for parallelization.

```
    subroutine adj(a,b,c,x,n)
     real*8 a(n), b(n), c(-n:0), x
      integer n
c$par doall
      do 19 i = 1, n*n
         do 29 k = i, n*n
            a(i) = a(i) + x*b(k)*c(i-k)
 29      continue
 19   continue
      return
      end
```

When you use `-Zlp` by itself, `-depend` and `-O3` are added. The switch `-depend` instructs the compiler to perform the data dependency analysis that it needs to do to identify loops. The switch `-parallel` includes `-depend`, but `-depend` does not imply (or trigger) `-parallel`.

What does `-Zlp` *do?*

Compiling with `-Zlp` has two effects.

1. the compiler inserts timing calls around all the loops that it can identify.
2. the compiler stores hints about the loops in a new ELF section named `.loops`. ELF is the name of the System V Release 4 executable format, and it stands for Executable and Linking Format. A `.o` file and an executable file are examples of ELF files.

The compiler stores information about your program's loops in a `.loops` section inside the `.o` file that it builds. This information includes the loop's source line number, whether or not the loop was parallelized, and reasons why a loop may not have been eligible for parallelization.

A `.o` file that was compiled with `-Zlp` is different from a typical `.o` in two ways.

- It is instrumented: the loops are bracketed with timing calls.
- It contains a `.loops` section, which contains information about each loop.

`-loopinfo` *Switch*

The `-loopinfo` option prints hints about loops to `stderr` (the Unix standard error file, on file descriptor 2) when you compile your program.

The hints include the routine names, line number for the start of the loop, whether the loop was parallelized, and the reason it was not parallelized.

The following example redirects hints about loops to the file named `gamtab.loopinfo`:

```
sh$ f77 -O3 -parallel -loopinfo -Zlp gamteb.F 2> gamteb.loopinfo
sh$ more gamteb.loopinfo
/tmp/cpp.17018.0.f:
 MAIN gamteb:
        track:
        klein:
        isos:
        rotas:
        rannum:
        tme:
        himom:
"gamteb.F", line 162: PARALLELIZED, fused
"gamteb.F", line 214: not parallelized, call may be unsafe
"gamteb.F", line 230: not parallelized, call may be unsafe
"gamteb.F", line 239: not parallelized, loop has multiple exits
"gamteb.F", line 294: not parallelized, not profitable
"gamteb.F", line 301: not parallelized, not profitable
"gamteb.F", line 369: PARALLELIZED
"gamteb.F", line 403: PARALLELIZED
"gamteb.F", line 452: not parallelized, not profitable
"gamteb.F", line 456: not parallelized, not profitable
"gamteb.F", line 461: not parallelized, call may be unsafe
"gamteb.F", line 490: not parallelized, unsafe dependence (dlss jas)
"gamteb.F", line 523: not parallelized, unsafe dependence (dlss jas)
"gamteb.F", not parallelized, call may be unsafe
"gamteb.F", not parallelized, call may be unsafe
```

-Qoption iropt -Dlooptable *Switch*

The -Qoption iropt -Dloop_table switch, prints a table of loop information to stdout (Unix standard output, on file descriptor 1.)

The table entries show the routine name, line number for the start of the loop, loop index, and loop nesting level.

The following example shows the loop summary tables for the MAIN, isos, track, and rotas routines in the gamtab.F program:

```
csh% f77 -O3 -parallel -Qoption iropt -Dloop_table gamteb.F > gamteb.loop_table
csh% more gamteb.loop_table
       Loop Summary Table -- gamteb.F : MAIN

=============================================================================
 Line  Index    Nest    | Comments
-----------------------+-----------------------------------------------
 162*  i        1       | PARALLELIZED
-----------------------+-----------------------------------------------
 214            1       | not-parallelized, call may be unsafe
-----------------------+-----------------------------------------------
 230            2       | not-parallelized, call may be unsafe
-----------------------+-----------------------------------------------
 239            3       | not-parallelized, loop has multiple exits
-----------------------+-----------------------------------------------
 294   ix       3       | not-parallelized, non-profitable
-----------------------+-----------------------------------------------
 301   ix       3       | not-parallelized, non-profitable
-----------------------+-----------------------------------------------
 369   i        2       | PARALLELIZED
-----------------------+-----------------------------------------------
 403   i        1       | PARALLELIZED
-----------------------+-----------------------------------------------
 452   i        1       | not-parallelized, non-profitable
-----------------------+-----------------------------------------------
 456   i        1       | not-parallelized, non-profitable
-----------------------+-----------------------------------------------
 461   isum     1       | not-parallelized, call may be unsafe
=============================================================================
*  the loop line number may be inaccurate due to loop transformation.
```

```
       Loop Summary Table -- gamteb.F : track

=============================================================================
 Line  Index    Nest    | Comments
-----------------------+-----------------------------------------------
 490   j        1       | not-parallelized, carried dependence
-----------------------+-----------------------------------------------
 523   j        1       | not-parallelized, carried dependence
=============================================================================


       Loop Summary Table -- gamteb.F : isos

=============================================================================
 Line  Index    Nest    | Comments
-----------------------+-----------------------------------------------
 0              1       | not-parallelized, call may be unsafe
=============================================================================


       Loop Summary Table -- gamteb.F : rotas

=============================================================================
 Line  Index    Nest    | Comments
-----------------------+-----------------------------------------------
 0              1       | not-parallelized, call may be unsafe
=============================================================================
```

It's OK to use `-Zlp`, `-loopinfo` and `-Qoption iropt -Dloop_table` in any combination. The static loop information that you get from any of the three switches is the same, although wording may vary slightly. `-Zlp` is intended to be the easiest for application programmers to use, although depending on your application, any one of the switches might be simplest for you to use.

The main difference between `-Zlp` and `-loopinfo`, and between `-Zlp` and `-Qoption iropt -Dloop_table`, is that in addition to providing you with compiler hints about loops, `-Zlp` also instruments your program so that timing statistics are recorded at runtime. For this reason, also, `looptool` and `loopreport` only analyze programs that have been compiled with `-Zlp`.

For more information on the meaning of the FORTRAN MP switches, refer to the `f77`(1) man page and to the FORTRAN User's Guide.

Run

Now that you have an instrumented executable ready, simply run it as you would ordinarily.

```
demo% /bin/time cholsky < cholsky.input >& cholsky.output
```

When the program finishes, it writes the loop times to a file named `a.out.looptimes`, where `a.out` is the name of your executable. In our example, the loop times are written to a file named `cholsky.looptimes`.

```
demo% ls
cholsky     cholsky.input    cholsky.looptimes  cholsky.output
```

The file `cholsky.looptimes` is a simple ascii file. It contains the raw loop timings. You should have no need to look at this file.

By default, the `.looptimes` file is created in the directory where you run your program. You can override this. If the environment variable `LVPATH` is set to a valid directory, then the timing files are written there. In this case, the timing files are named <*pid*>.<*aout*>, where *pid* is the process ID of your running program, and *aout* is the name of your running program.

```
demo% setenv LVPATH /home/sally/looptimes
demo% /bin/time cholsky < cholsky.input >& cholsky.output
demo% ls /home/sally/looptimes
16245.cholsky
```

Analyze

You can use `loopreport`, the command line interface, or `looptool`, the graphical user interface, to interpret the loop timing files.

```
demo% loopreport cholsky > cholsky.loopreport

demo% looptool cholsky &
```

1

`looptool` *Graphical User Interface* 2

The `looptool` graphical user interface displays a graph of loop runtimes and shows which loops were parallelized. It also displays hints about why a particular loop was not parallelized. See Chapter 4, “Information Presented by looptool and loopreport,” for a description of hints.

This chapter has the following organization:

Starting looptool
Overview Window
Source and Hints Window

Starting `looptool`

`looptool` can be given the name of a file to load,

```
demo% looptool cholsky &
```

or you can bring up the graphical user interface by itself, and use its file chooser to select the name of your program.

```
demo% looptool &
```

The file chooser automatically pops up and prompts you for a file name, if you do not name one.

looptool reads the timing file associated with your program. Typically, this file is named *programname.looptimes*, and is found in the same directory as your program. If this is the case, you do not need to do anything special to tell looptool where to find the timing file.

There are four places where the timing files may be located. The following discussion describes the rules and syntax for specifying the location of timing files.

- First, if a timing file is named on the command line, then looptool uses it.

```
demo% looptool cholsky newtimes &
```

- Second, if the command line option -p is used, then looptool looks in the directory named by -p for a timing file.

```
demo% looptool -p /home/sally/timingfiles cholsky &
```

- Third, if the environment variable LVPATH is set, looptool looks in that directory for a timing file.

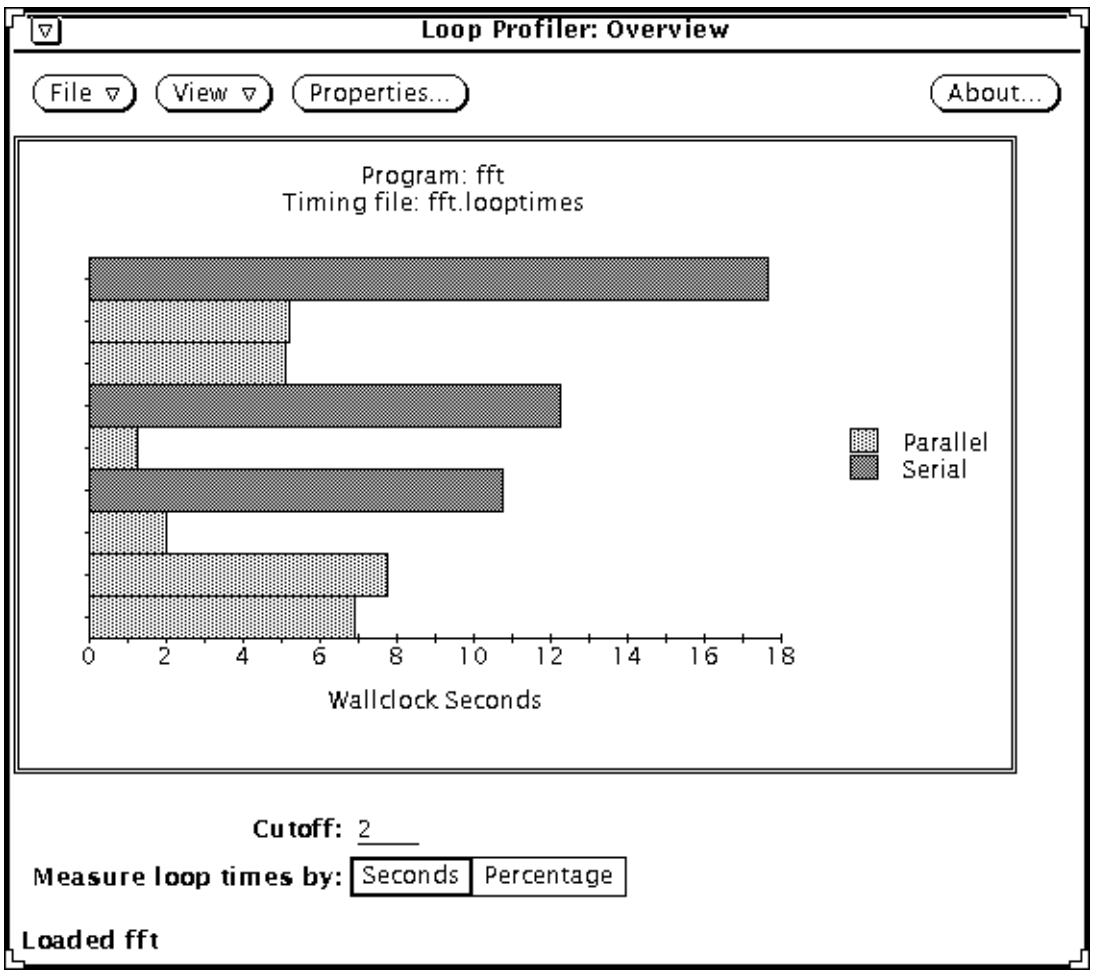
```
demo% setenv LVPATH /home/sally/timingfiles
demo% looptool cholsky &
```

- Fourth, looptool looks in the executable's directory for a timing file. This is the typical situation.

```
demo% looptool cholsky &
```

Overview Window

`looptool` has two displays. The first display is an overview of your program's loops' performance. It is a bar chart of the loops' runtimes, arranged in the order that the source files were presented to the compiler. On a color monitor, parallel loops are shown in red, and serial loops are shown in yellow.



Cutoff: sets the threshold value for loops accounting for percent of longest running loop. To specify the threshold, type a number and press Return.

Measure loop times by: specifies whether to draw the graph against Seconds (Wallclock seconds) or Percentage (Percentage of Total Runtime).

The window footer provides status and error information.

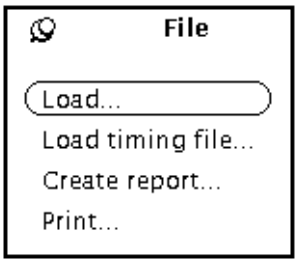
The overview window has four buttons:

- File
- View
- Properties ...
- About ...

File Button

The File button opens the following menu entries:

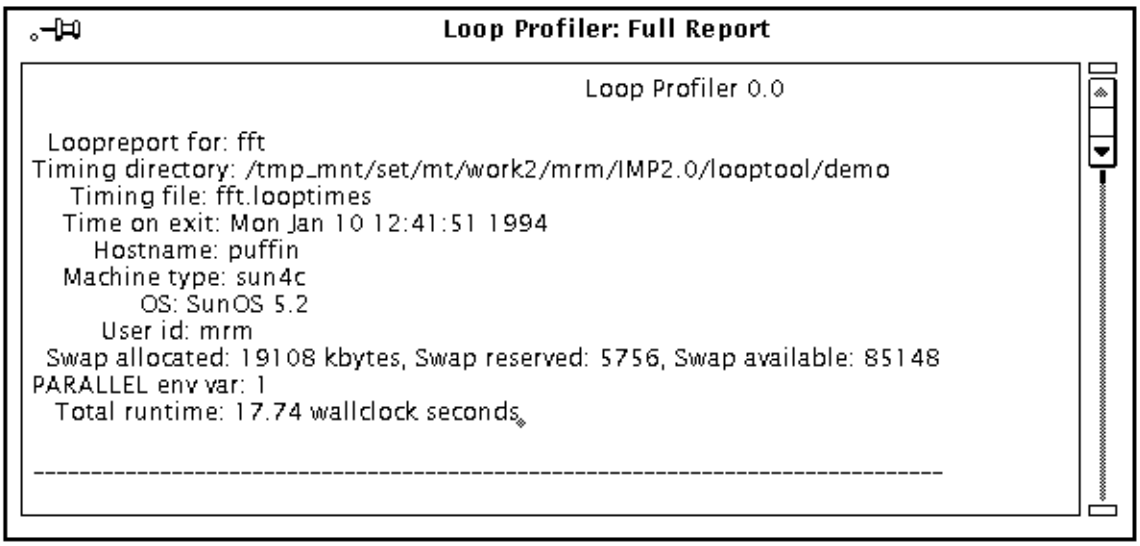
- load an instrumented object file
- load the timing file associated with the instrumented object file
- create a report
- print



To select a menu entry, press the right mouse button, drag to the desired entry and release the mouse button

Create Report

A loop profiler report looks like:



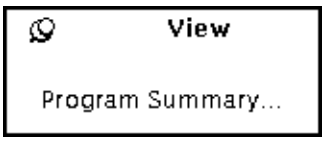
The report lists:

- name of the instrumented object file
- timing file directory
- name of the timing file
- date and time of program exit
- hostname, machine type, operating system version, and user_id
- swap space
- name of environment variable used
- total runtime in wallclock seconds
- summary information about all loops in the program

Use the right-margin scrollbar to display the entire report.

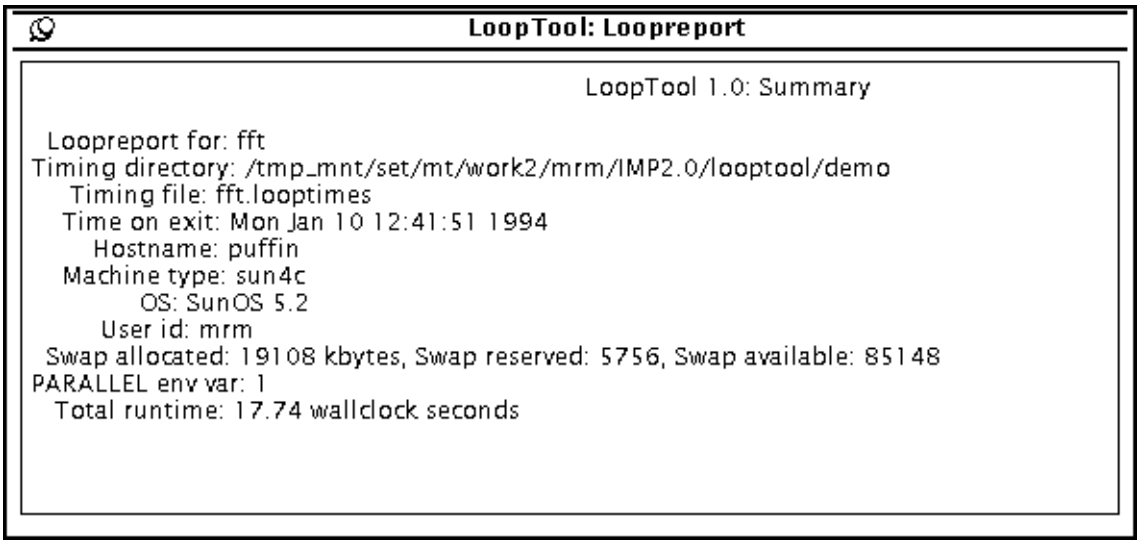
View Button

The View button allows you to display a summary of your program.



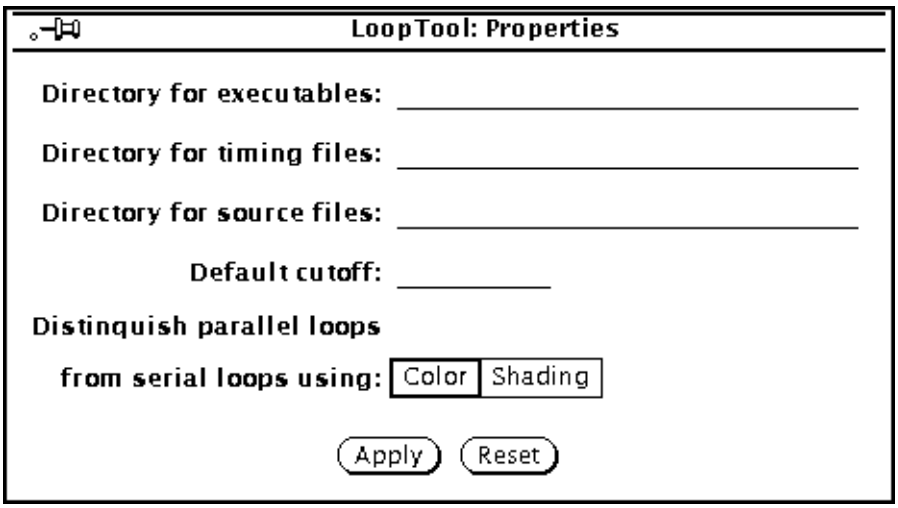
The summary information contains:

- name of your program
- name of the timing directory
- name of timing file
- time the program was exited
- hostname, machine type, and SunOS version
- user_id
- swap space allocated
- environment variable used
- total wallclock time elapsed for the program run.



Properties Button

Click the Properties button to open the Properties window.:

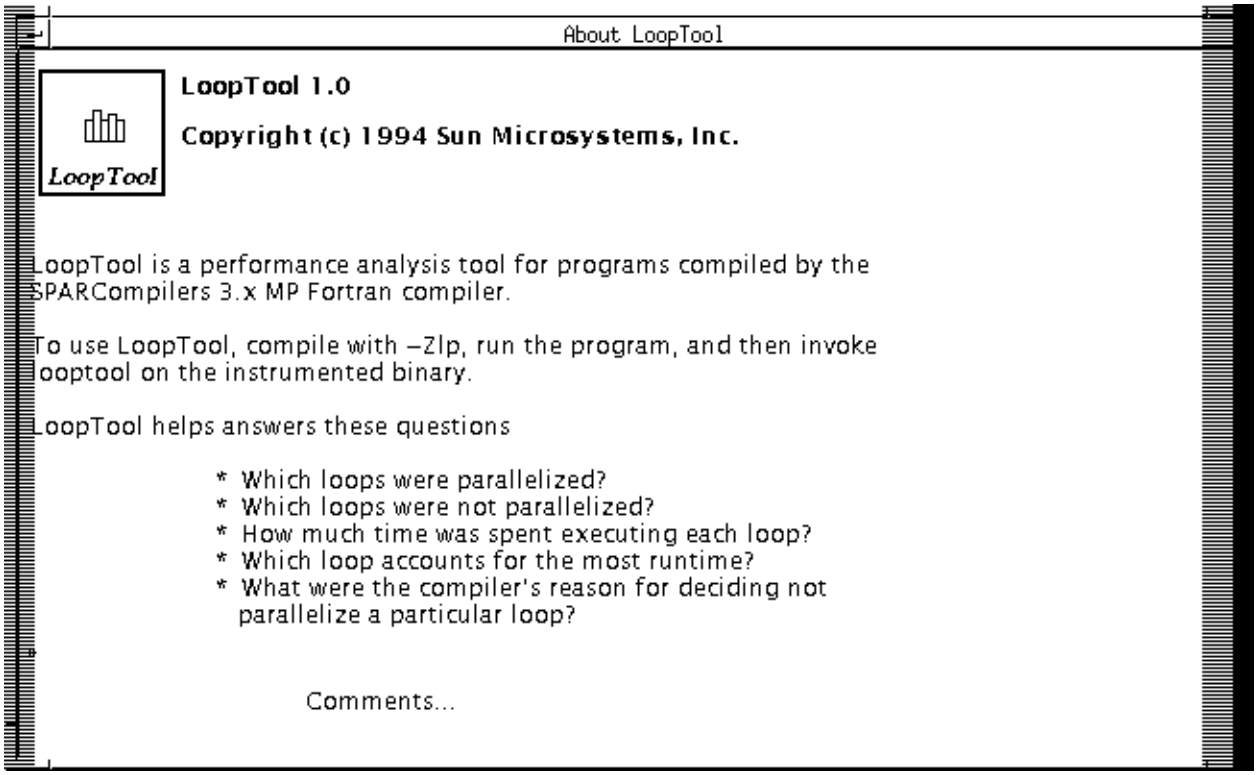


This window allows you to specify the:

- directory pathname for the desired instrumented object files
- directory pathname for the associated timing files
- directory pathname for the source files
- threshold value for loops accounting for percent of longest running loop.
- way to distinguish parallel loops from serial loops through color or shading (monochrome display).

About Button

About box looks like:



Use the right-margin scrollbar to display the entire about box.

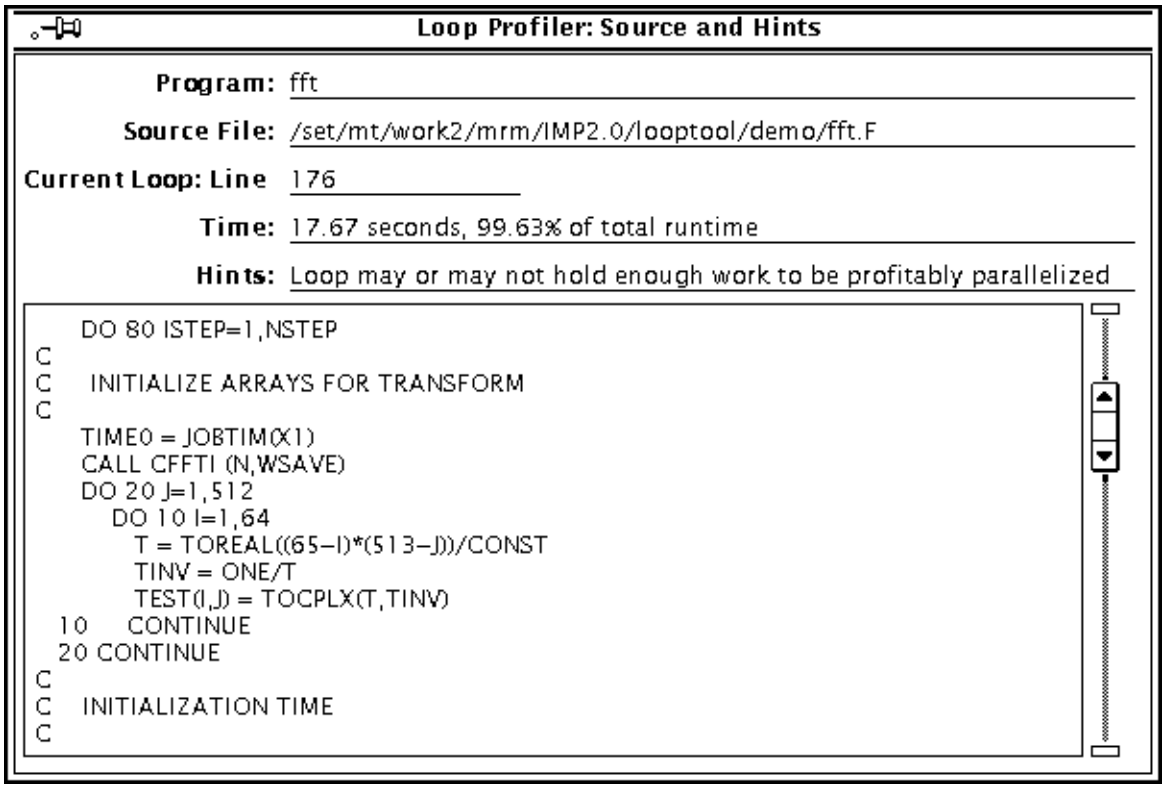
The about box contains:

- Purpose — reason for using `looptool`
- Background — performance-tuning f77 program compiled on SC3.0.1 FORTAN MP compiler
- How to Use — summary of compile, run, and analyze
- Sample Results

To send email comments about `looptool`, click the comments button to open the comments window.

Source and Hints Window

To display information about a particular loop, click the left mouse button on a loop. `looptool` displays the *Source and Hints* window. It shows the source code for the selected loop, and it lists the loops' statistics and any compiler hints regarding this loop. (The footer in the *Overview* window identifies the loop as serial or parallel and shows the source file name and loop starting line number.)



2

loopreport
Command Line Interface 3

This chapter describes the loopreport command line interface.

Tbis chapter has the following organization:

Starting loopreport

Starting loopreport

loopreport expects to be given the name of your program.

```
demo% loopreport cholsky > cholsky.loopreport
```

If you invoke loopreport without giving it the name of a program, it looks for a file named a.out in the current working directory.

```
demo% loopreport > a.out.loopreport
```

loopreport also reads the .looptimes file associated with your program. Typically, the timing file is named programname.looptimes and is in the same directory as programname. If this is the case, then you do not need to tell loopreport where to find the file. By default, loopreport searches the current working directory for a timing file named programname.looptimes.

However, there are four ways to specify a timing file's location. `loopreport` chooses a timing file according to these rules:

- First, if a timing file is named on the command line, then `loopreport` uses it.

```
demo% loopreport cholsky newtimes > cholsky.loopreport
```

- Second, if the command line option `-p` is used, then `loopreport` looks in the directory named by `-p` for a timing file.

```
demo% loopreport cholsky -p /home/sally/timingfiles >
      cholsky.loopreport
```

- Third, if the environment variable `LVPATH` is set, `loopreport` looks in that directory for a timing file.

```
demo% setenv LVPATH /home/sally/timingfiles
demo% loopreport cholsky > cholsky.loopreport
```

- Fourth, `loopreport` looks in the executable's directory for a timing file. This is the typical situation.

```
demo% loopreport cholsky > cholsky.loopreport
```

`loopreport` writes the table of loop statistics to `stdout`, the standard output. You can also redirect the output to a file,

```
demo% loopreport cholsky > cholsky.loopreport
```

or pipe it into another command.

```
demo% loopreport cholsky | more
```

Information Presented by `looptool` *and* `loopreport` 4

This chapter lists the compiler hints and describes what they mean. It also describes the affect particular compiler options have on loops.

This chapter has the following organization:

Compiler Hints
Compiler Optimizations and How They Affect Loops

Compiler Hints

`looptool` and `loopreport` present you with somewhat cryptic hints about the optimizations applied to a particular loop, and in particular, about why a particular loop may not have been parallelized.

Note – The hints are heuristics gathered by the compiler during the optimization pass. They should be understood in that context; they are *not* absolute facts about the code generated for a given loop. However, the hints are often very useful indications of how you can transform your code, so that the compiler can perform more aggressive optimizations, including parallelizing loops.

Some of the hints seem redundant; that is, two hints may mean essentially the same thing. Do not worry about this, but do let us know which of the hints help you, or, what other sorts of hints you need from the compiler. You can send feedback to us by using the Comment button (inside the About box) available in the `looptool` GUI.

Finally, read the sections in the SC3.0.1 Fortran User's Guide that address parallelization. There are useful explanations and tips inside that manual.

The following summary lists the hints about the optimizations applied to loops:

0 No hint available

1 Loop contains procedure call

2 Compiler generated two versions of this loop

3 Loop contains data dependency

4 Loop was significantly transformed during optimization

5 Loop may or may not hold enough work to be profitably parallelized

6 Loop was marked by user-inserted pragma, DOALL

7 Loop contains multiple exits

8 Loop contains I/O, or other function calls, that are not MT safe

9 Loop contains backward flow of control

10 Loop may have been distributed

11 Two or more loops may have been fused

12 Two or more loops may have been interchanged

0 — No hint available

None of the other hints (1 through 12) applied to this loop. It does not mean that none of the other hints might apply — it simply means that the compiler did not infer any of those hints.

1 — Loop contains procedure call

The loop could not be parallelized since it contains a procedure call that is not MT safe. If such a loop were parallelized, there is a chance that multiple copies of the loop could instantiate the function call simultaneously, trample on each others' use of any variables local to that function, or trample on return values, and generally invalidate the function's purpose. If you are certain that the procedure calls in this loop are, in fact, MT safe, you can direct the compiler to parallelize this loop no matter what, by inserting the DOALL pragma before the body of the loop. For example, if foo is an MT safe function call, then you can force this inner to be parallelized by inserting c$par doall:

```
c$par doall
      do 19 i = 1, n*n
           do 29 k = i, n*n
                a(i) = a(i) + x*b(k)*c(i-k)
                call foo()
 29        continue
 19   continue
```

The DOALL pragmas are interpreted by the compiler only when you compile with -parallel or with -explicitpar; if you compile with -autopar, then the compiler ignores the DOALL pragmas. This can be handy for debugging or fine-tuning.

2 — Compiler generated two versions of this loop

The compiler couldn't tell, at compile time, if the loop contained enough work to be profitable to parallelize. The compiler generated two versions of the loop, a serial version and a parallel version, and a runtime check that will choose, at runtime, which version should execute. The runtime check determines the amount of work that the loop has to do by checking the loop iteration values.

3 — Loop contains data dependency

A variable inside the loop is affected by the value of a variable in a previous iteration of the loop. For example,

```
do 99 i=1,n
      do 99 j = 1,m
            a[i, j+1] = a[i,j] + a[i,j-1]
99 continue
```

This is a contrived example, since for such a simple loop, the optimizer would simply swap the inner and outer loops, so that the inner loop could be parallelized. But this example demonstrates the concept of data dependency. This is often referred to as "data carried dependency."

The compiler will often be able to tell you the names of the variables that cause the data carried dependency. If you rearrange your program to remove (or minimize) such dependencies, then the compiler will be able to perform more aggressive optimizations.

4 — Loop was significantly transformed during optimization

The compiler performed some optimizations on this loop that might make it almost impossible to associate the generated code with the source code. For this reason, line numbers may be incorrect. Examples of optimizations that can radically alter a loop are loop distribution, loop fusion and loop interchange (see below.)

5 — Loop may or may not hold enough work to be profitably parallelized

The compiler was not able to determine at compile time that this loop definitely held enough work to warrant the overhead of parallelizing. Often, loops that are labeled with Hint 5 may also be labeled as "parallellized", meaning that the compiler generated two versions of the loop (see hint 2), and that it will be decided at runtime whether the parallel version or the serial version should be used.

Since all the compiler hints, including the flag that indicates whether or not a loop is parallelized, are generated at compile time, there's no way to be 100% certain that a loop labeled as "parallelized" actually executes in parallel. To do

this, we need to perform additional runtime tracing, such as can be accomplished with the Thread Analyzer. You can compile your programs with both -Zlp (for LoopTool) and with -Ztha (for Thread analyzer) and compare the analysis of both tools to get as much information as possible about the runtime behavior of your program.

6 — Loop was marked by user-inserted pragma, DOALL

This loop was parallelized because the compiler was instructed to do so via the DOALL pragma. This hint is a useful reminder, more than anything else, to help you easily identify those loops that you explicitly wanted to parallelize.

The DOALL pragmas are interpreted by the compiler only when you compile with -parallel or -explicitpar; if you compile with -autopar, then the compiler will ignore the DOALL pragmas, which can be handy for debugging or fine-tuning.

7 — Loop contains multiple exits

The loop contains a GOTO or some other branch out of the loop other than the natural loop end point. For this reason, it is not safe to parallelize the loop, since the compiler has no way of predicting the runtime behavior of the loop.

8 — Loop contains I/O, or other function calls, that are not MT safe

This hint is similar to Hint 1. The difference is that Hint 8 often focuses on I/O that is not MT safe, and Hint 1 could refer to any sort of MT unsafe function call.

9 — Loop contains backward flow of control

The loop contains a GOTO or other control flow up and out of the body of the loop. That is, some statement inside the loop jumps back to some previously executed portion of code, as far as the compiler control flow can determine! As with the case of a loop that contains multiple exits, this means that this loop is not safe to parallelize.

If you can reduce or minimize backward flows of control, the compiler will be able to perform more aggressive optimizations.

10 — Loop may have been distributed

The contents of the loop may have been distributed over several iterations of the loop. That is, the compiler may have been able to rewrite the body of the loop so that the loop could be parallelized. However, since this rewriting takes place in the language of the internal representation of the optimizer, it's very difficult to associate the original source code with the rewritten version. For this reason, hints about a distributed loop may refer to line numbers that don't correspond to line numbers in your source code.

11 — Two or more loops may have been fused

Two consecutive loops were combined into one, so that the resulting larger loop contains enough work to be profitably parallelized. Again, in this case, source line numbers for the loop may be misleading.

12 — Two or more loops may have been interchanged

The loop indices of an inner and an outer loop have been swapped, to move data dependencies as far away from the inner loop as possible, and to enable this nested loop to be parallelized. In the case of deeply nested loops, the interchange may have occurred with more than two loops.

Compiler Optimizations and How They Affect Loops

As you might infer from the descriptions of the compiler hints, it can be tricky to associate optimized code with source code. Clearly, you would prefer to see information from the compiler presented to you in a way that relates as directly as possible to your source code. You do not care about the compiler's internal representation of your program! Unfortunately, the compiler optimizer is "reading" your program in terms of this internal language, and although it tries its best to relate that to your source code, it is not always successful.

Some optimizations, in particular, can cause confusion.

Inlining

"Inlining" is an optimization applied only at optimization level -O4, and only for functions contained with one file. That is, if one file contains 17 Fortran functions, and 16 of those can be inlined into the first function, and you compile at -O4, then potentially, the source code for those 16 functions are copied into the body of the first function. Then, further optimizations are applied, and it's difficult to say at this point which loop on which source line number was subjected to which optimization.

If the compiler hints seem particularly opaque, consider compiling with `-O3 -parallel -Zlp`, so that you can see what the compiler has to say about your loops before it tries to inline any of your functions.

In particular, if you notice "phantom" loops - that is, loops that the compiler claims to exist, but which you know do not exist in your source code — this could well be a symptom of inlining.

Loop Transformations — Unrolling, Jamming, Splitting, and Transposing

There are many optimizations the compiler performs on loops that radically change the body of the loop. These include loop unrolling, loop jamming, loop splitting, and loop transposition. Our goal is to try to provide you with what hints we can, that make as much sense as possible, but given the nature of the problem of associating optimized code with source code, the hints may be misleading. If you are interested in this topic, refer to compiler books such as *Compilers: Principles, Techniques and Tools* by Aho, Sethi and Ullman, for more information on what these optimizations do for your code.

Parallel Loops Nested inside Serial Loops

If a parallel loop is nested inside a serial loop, the runtime information reported by `looptool` and `loopreport` may be misleading.

The reason is that each loop is stipulated to use the wallclock time of each of its loop iterations. If an inner loop is parallelized, it will be assigned the wallclock time of each iteration, although some of those iterations are running in parallel.

However, the outer loop will only be assigned the runtime of its child, the parallel loop, which will be the runtime of the longest parallel instantiation of the inner loop.

This leads to the anomaly of the outer loop consuming "less" time than the inner loop! Keep in mind the funny nature of "time" when measuring events that occur in parallel, and this will help keep all these wallclock times in perspective.

Glossary

-autopar A compiler optimization flag recognized by the SC3.0.1 f77 compiler. -autopar instructs the compiler to parallelize those loops that the compiler automatically determines to be safe for parallelism. That is, if your source code contains the DOALL pragma for explicit parallelization, if you compile with -autopar, the compile ignores the DOALL pragmas. This is useful for debugging or fine-tuning.

carry dependency A carry dependency exists if the value of a variable in one iteration of a loop depends on its value in a previous iteration of the loop. This is also known as data carried dependency, or data dependency.

DOALL The DevPro pragma for instructing the compiler to parallelize a loop, regardless of its judgement as to whether or not it is safe to do so. The syntax of the DOALL pragma is c$par doall

The pragma must begin in the first column of the line. The DOALL pragma is interpreted by the compiler only when you compile with -explicitpar or with -parallel.

-explicitpar A compiler optimization flag recognized by the SC3.0.1 f77 compiler. -explicitpar instructs the compiler to parallelize those loops that are marked with the user-inserted DOALL pragma.

liblv.a A Fortran program compiled with -Zlp is linked with the library liblv.a, which contains the runtime timing calls. liblv.a in turn calls the libc function, gethrtime, to time the instrumented loops. The high resolution timer in libc is an accurate and efficient timer that delivers its results in nanoseconds. (Granted, this granularity of results may not be meaningful, given the overhead of what is measured! But the timer's results are much better than other, standard unix timers.)

The library liblv.a is delivered as part of the MP Fortran package.

libmtsk.a The microtasking library that is part of the MP Fortran compiler. Parallelized loops make calls to the microtasking library, which in turn makes calls to the Solaris 2 libthread library.

The library libmtsk.a is delivered as part of the MP Fortran package.

-loopinfo A compiler instrumentation flag that is recognized by the SC3.0.1 f77 compiler. -loopinfo prints compiler hints about the loops in your code to stderr.

loopreport The command line interface for the loop profiling tool.

looptool The graphical user interface for the loop profiling tool.

MP FORTRAN A common name for the automatically parallelizing Fortran 77 compiler, SC3.0.1 f77. This compiler requires the MP/MT license. It provides the parallelization switches -parallel, -autopar, and -explicitpar, and the instrumentation/debugging switches -Zlp, -loopinfo and -Qoption iropt -Dloop_table.

Although the compiler is often referred to as an MP compiler, and indeed, the benefits of parallelization are seen best on multiprocessor systems, the compiler's microtasking library uses the Solaris 2 threads technology as the basis for its implementation. Currently, the parallelizing compiler's microtasking library does instantiate bound LWPs, although this is subject to change.

Refer to the Solaris 2 documentation on threads, libthread.so and LWPs for more information.

-parallel A compiler optimization flag recognized by the SC3.0.1 f77 compiler. -parallel instructs the compiler to perform both automatic and explicit parallelization. Automatic parallelization refers to loops that the compiler determines are safe to parallelize; explicit parallelization refers to loops that you instruct the compiler to parallelize, by placing the DOALL pragma before any such loops.

-Qoption iropt -Dloop_table A compiler instrumentation flag that is recognized by the SC3.0.1 f77 compiler. -Qoption iropt -Dloop_table prints compiler hints about the loops in your code to stdout.

-Zlp A compiler instrumentation flag recognized by the SC3.0.1 f77 compiler. -Zlp instruments your code so that it can be analyzed by the loop profiling tools, looptool and loopreport. -Zlp does essentially two things: it collects compile-time hints about the optimizations performed on your loops, and it places

timing calls around all the loops in your program. The loop profiling tools correlate these two sets of information so that you can easily identify long-running loops and see whether or not they were parallelized, and if not, if there's anything you can do to the code to make it more amenable to parallelization.

-Ztha A compiler instrumentation flag that is recognized by the SC3.0.1 f77, cc and C++ compilers. Code compiled with -Ztha can be analyzed by the Thread Analyzer.

Index

Symbols

`.loops` section
  information about loops, 3
  loop parallelized or not, 3
  reason why loop not eligible for parallelization, 3
  source line number of loop, 3
`.looptimes` file
  raw loop timings, 7

A

affects of optimizations applied to loops
  inlining, 27
  loop transformation, 27
  phantom loops, 27
automatic parallelization
  `-parallel` compiler switch, 2

C

choose a timing file
  `loopreport`, 20
command line interface
  `loopreport`, 19
compiler hints about loop
  `looptool`, 17
compiler switch
  `-depend`, 3
  `-loopinfo`, 4
  `-O4`, 2
  `-parallel`, 2
  promotion of, 2
  `-Qoption iropt -Dlooptable`, 5
  `-Zlp`, 2, 3

D

`-depend`
  perform data dependency analysis, 3
DOALL pragma
  mark loop for parallelization, 3

E

ELF file, 3
environment variable
  LVPATH, 7

H

hints about optimizations applied to loops, 22
  compiler generated two versions of this loop, 23
  loop contains backward flow of control, 25

loop contains data dependency, 24
loop contains I/O, or other function calls, that are not MT safe, 25
loop contains multiple exits, 25
loop contains procedure call, 23
loop marked by user-inserted pragma, DOALL, 25
loop may have been distributed, 26
loop may or may not hold enough work to be profitably parallelized, 24
loop significantly transformed during optimization, 24
no hint available, 22
two or more loops may have been fused, 26
two or more loops may have been interchanged, 26

L

`-loopinfo`
print hints about loops, 4
`loopreport`
command line interface, 19
`looptool`
About button, 16
Commants, 16
bar chart of loop runtimes, 11
compiler hints about loop, 17
File button, 12
loop profiler report, 13
graphical user interface, 9
loop runtime bar chart, 11
Properties button, 15
LVPATH
set directory for timing files, 7

O

optimizations applied to loops
inlining, 27
loop transformations
jamming, 27
splitting, 27
transposition, 27
unrolling, 27
phantom loops, 27

P

`-parallel`
automatic parellelization, 2
parallel loop nested inside serial loop
wallclock anomaly, 27
parallelize loop marked by DOALL pragma
`-explicitpar`, 3
pragma
DOALL, 3
promotion of compiler switches, 2

Q

`-Qoption iropt -Dloop_table`
print loop summary table, 5

R

run instrumented oject
write loop times, 7

T

table of loop statistics
pipe to another command, 20
redirect to file, 20
write to standard output, 20
timing file location
`loopreport`
current directory, 20
specified via command line, 20
specified via environment variable, 20
specified via `-p` option, 20
`looptool`
current directory, 10
specified via command line, 10
specified via environment variable, 10
specified via `-p` option, 10

W

wallclock anomaly
    parallel loop nested inside serial
        loop, 27

Z

`-Zlp`
    `.o` object file, 4
    insert timing calls around loops, 3
    store hints about loops, 3