THE UNIVERSITY of York

High Performance Computing - Profiling

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

Overview

- Hardware / Algorithms / Implementation
- Establishing a Baseline
  - Timing vs Sampling
- Profiling with gprof

Choices

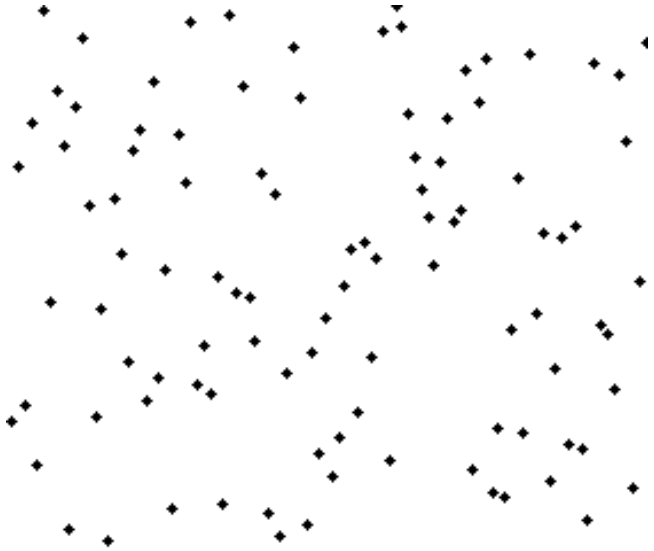
- What factors affect the speed of execution of your code?
  - Hardware – see previous lectures
  - Algorithm – significant!
  - Implementation – rest of this & another lecture

Algorithm

- Once you have understood the problem you wish to solve, you need to design your algorithm(s)
- Many standard algorithms for common tasks already exist
  - Eg sorting, linear algebra, random numbers, special functions, curve fitting, etc.
- Many already implemented in libraries
- DO NOT REINVENT THE WHEEL!

Choice of Algorithm

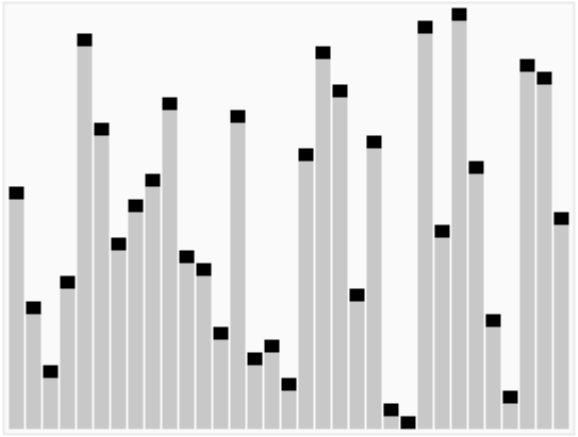
- You might think sorting a list of numbers into ascending order is trivial:
  - Step through list and compare X_i and X_{i+1}
  - Swap if $X_i > X_{i+1}$
  - Repeat until complete pass through list has no swaps
- This is algorithm is known as "Bubble Sort" - simple to code, minimal storage and stable.
- BUT requires $\mathcal{O}(N^2)$ operations



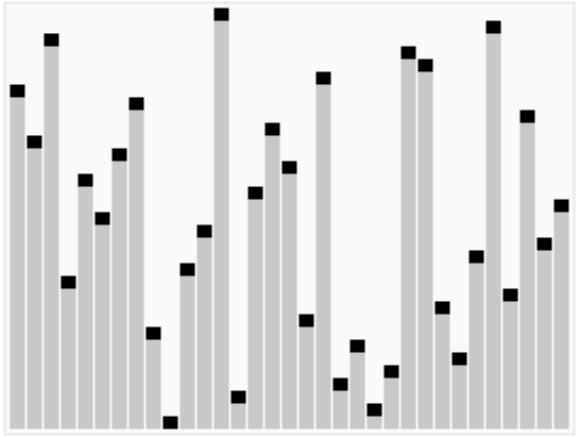
Gif from Wikipedia

Choice of Sorting Algorithm

- MANY alternatives, e.g.
  - Quicksort usually $\mathcal{O}(N \log N)$ but can be $\mathcal{O}(N^2)$
    - Stable if $\mathcal{O}(N^2)$ memory version but unstable if $\mathcal{O}(N \log N)$ memory version ...
  - Heapsort always $\mathcal{O}(N \log N)$ but unstable
  - New algorithms still being invented ...



Quicksort in action



Heapsort in action

Algorithm Impact

- One way to make a big impact is to invent & publish a new algorithm
  - Ideally one that is more efficient than previous, less additional storage, and stable
- E.g. Car-Parrinello paper
  - showed how to solve numerical QM without doing matrix diagonalisation – cost now $\mathcal{O}(N^2M)$ not $\mathcal{O}(M^3)$ for system with N electrons and M basis functions with M » N
  - One of the most highly cited condensed matter physics papers of all time!

Implementation

- Having chosen your algorithm, there can be surprising performance differences depending on how it is implemented!
- Hence need to be able to time code and then measure speedup
- Investigate compiler flags for simple gains
- And profile code to identify ‘hot spots’ for more work
  - ‘cost-benefit’ balance – time to optimise vs time to run code vs number of runs ...

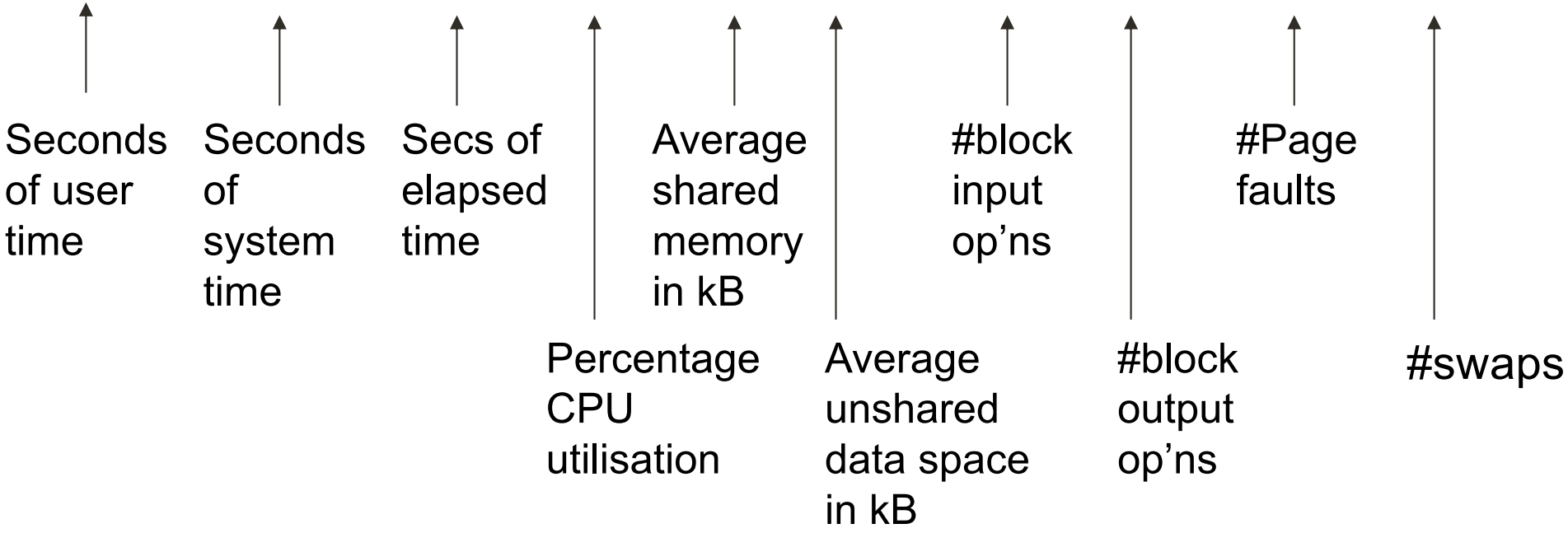
Timing

- NOT your watch!
  - ‘elapsed time’ or ‘wall time’ depends upon machine load and has poor repeatability
- Your program will switch between *user mode* (your code) and *kernel mode* (O/S services) :
  - **user time:** is the total CPU time your program used
  - **system time:** O/S tasks (e.g. writing to a file or screen)
  - **CPU time** is user time + system time, and ideally would want user time » system time, otherwise code is I/O bound or in some sort of memory trouble
  - **wall clock or real time:** is the total elapsed time to run your code and is the time you want to reduce.

UNIX `time` command

- As an example of a global timing command, consider the c-shell built-in `time` command (the bash time command is less useful):

```
% time ./myprogram.exe
14.9u 1.42s 0:19 83% 4+1060k 27+86io 47pf+0w
```



`time` limitations

- Can use `time` as an overall assessment of performance and to give clues as to possible bottlenecks in code.
- Useful as a base line to measure performance improvements BUT
  - Has low timing resolution (typically 0.1 seconds) hence only useful for long calculations
  - And only gives a global summary – no breakdown into different sections of the code
- Hence need a more fine-grained approach.

Explicit Coding

- One solution to this problem is to add timing routines into the code yourself.
  - Can be labour intensive.
  - Can have problems with *portability* depending on language used – easier with MPI parallel codes as MPI contains a truly portable timer.
  - Need to be careful to distinguish wall clock and CPU timing measurements – CPU time is best as not affected by load but elapsed time is useful too, particularly for parallel codes.
  - Can be useful – can build in timers to denote progress of code, and high-level profiling.
    - E.g. CASTEP has a compile-time option to use either elapsed-time or CPU time clock, and a run-time switch to turn clock output on/off (for testing – makes it easier to `diff` output on different machines).

Using `dtime`

- Many O/S provide a `dtime` or `etime` system function that returns the CPU time
  - Timing resolution varies with system
  - Pass 2-element real array, returns user and system time

```
real, dimension(2) :: time !default 4-byte real here
real               :: UserTime, SystemTime, TotTime

TotTime  = dtime(time)
UserTime = time(1): SystemTime = time(2)

< insert some highly important stuff here to be timed >

TotTime    = dtime(time) - TotTime
UserTime   = time(1) - UserTime
SystemTime = time(2) - SystemTime

print *,"Total time ",TotTime," secs"
print *,"System time",SystemTime," secs"
print *,"User time  ",UserTime ," secs"
```

Can also do this in C, C++ or any other language.

Using F95 Intrinsics

- Fortran 95 provides an intrinsic `CPU_TIME(time)` to make timing code portable and simple
  - where `time` is a real scalar that is assigned a system-dependent approximation to the time in seconds (or a negative value if there is no clock) :

```
real :: start, end, elapsed_time

call cpu_time(start)

< important stuff to be timed >

call cpu_time(end)
elapsed_time = end - start
```

Subprogram Sampling

- With a large, complex code it can be very cumbersome to do this for every individual subprogram (might be useful if interested in just one or two chunks)
- Many compilers can help by *instrumenting* your code, so that the *program counter* will be periodically sampled when the code runs.
  - See where code is spending different proportions of time – but increases actual run-time due to sampling overhead so do not do this with production version!
- This is known as *profiling*...

Enabling Sampling

- In order to get this profiling information, need to tell the compiler to instrument your code.
  - typically use `-pg` flag.
  - Add debug symbols for line by line information (`-g`)
  - Code optimisations may mess up profile, disable with `-O0`
  - Newer versions of gcc/gfortran have `-Og` option
- Run the code as normal, and it will produce a *trace file* called `gmon.out` for later analysis by the `gprof` tool.
- `gprof` will produce a summary of the time spent in each subprogram and a call graph profile
  - Use `gcov` to see line-by-line analysis
  - There are also O/S or vendor dependent tools ...

gprof example

```
% gcc -o ctest -pg ctest.c
% ./ctest
% gprof ctest
```

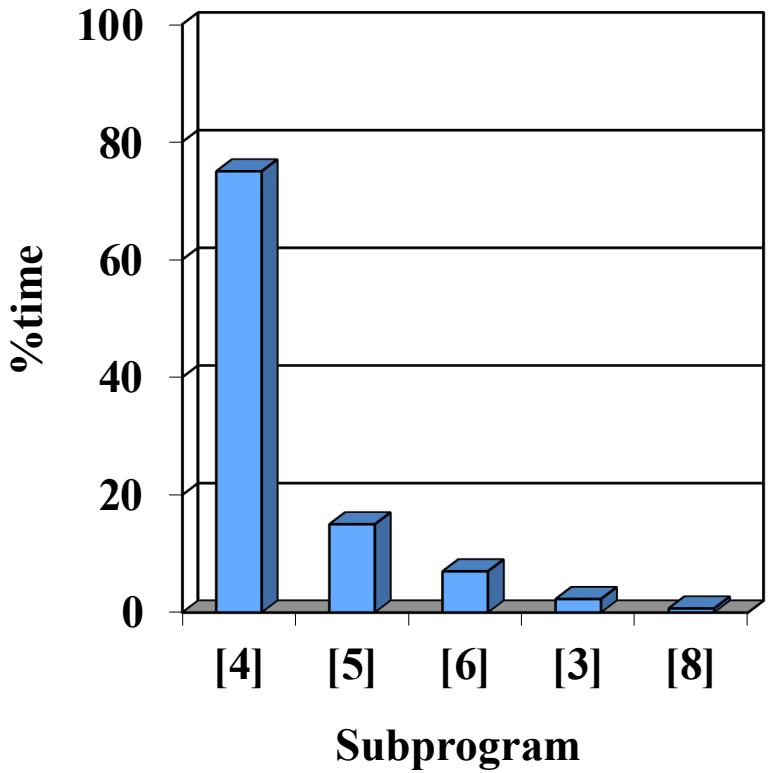
(A very stupid list-based program to read sequence of integers from file, sort into order and print out result.)

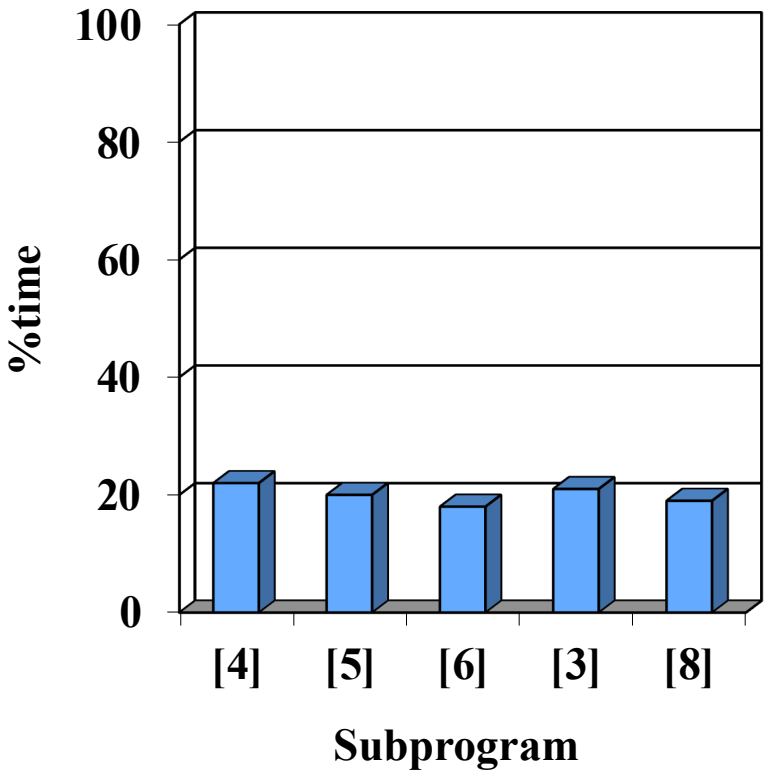
Timing (flat) profile with 0.01 sec sampling:

% time	Cumulative seconds	Self seconds	Calls	Self s/call	Total s/call	Name
75.12	15.94	15.94	53843	0.00	0.00	insert_list [4]
15.65	19.26	3.32	328408081	0.00	0.00	make_nonempty [5]
6.79	20.70	1.44	328515774	0.00	0.00	is_empty [6]
2.31	21.19	0.49	1	0.49	21.19	sort_list [3]
0.07	21.20	0.01	2	0.01	0.01	print_list [8]
0.05	21.21	0.01	4	0.00	0.00	append_lists [10]
0.02	21.22	0.01	2	0.00	0.01	print_data [7]
etc						

"`%time`" is contribution of this routine to total runtime. "`Cumulative sec`" is sum of all preceding routines + this one. "`Self secs`" is this routines contribution. "`Calls`" is the no. of times routine called. "`Self s/call`" is the average time in this routine. "`Total s/call`" is average time in this routine + all its descendants.

Subprogram Profiles





Sharp profile – dominated by routine [4] – hence easy to see where to target code optimisation.

Flat profile – no routines dominate – hence hard to optimise – need to work on whole code optimisation.

Amdahl's Law

- Gene Amdahl (designer of some of the early parallel computers) came up with a "law" for performance potential of parallel computers – which can also be applied to profiling.
- If your code contains portions that can be optimised (A), and some that cannot (B), then even if you make (A) infinitely fast, the runtime will be dominated by (B).
  - E.g. in the "sharply-peaked" profile, if subprogram [4] was optimised to go 75 times faster, the runtime would only be improved by ~4 times. Doing the same in the "flat profile" would result in an overall 1.25x speedup!
  - Hence there is a finite return on effort – no point over-optimising one routine if run-time contribution is small.

gprof call-graph profile

Index	% time	Self	Children	Called	Name
[1]	100.0	0.00	21.22		main [1]
		0.00	21.19	1/1	sort_data [2]
		0.01	0.02	2/2	print_data [7]
		0.00	0.01	1/1	get_data [9]
---	---	---	---	---	---
		0.00	21.19	1/1	main [1]
[2]	99.9	0.00	21.19	1	sort_data [2]
		0.49	20.70	1/1	sort_list [3]
---	---	---	---	---	---
		0.49	20.70	1/1	sort_data [2]
[3]	99.9	0.49	20.70	1+53843	sort_list [3]
		15.94	4.76	53843/53843	insert_list [4]
		0.00	0.00	53844/328515774	is_empty [6]
etc					

"Index" is a cross-reference for locating given routine in listing. "%time" is time in routine + all its children. "Self" is time in this routine and "Children" is time in all the called routines within. "Called" indicates number of calls of child from this parent, with "+" indicating recursive calls.

Interpreting Profile Output

- From the timing profile can quickly identify where your program is spending its time.
  - Will often see system routines not explicitly called in your program, e.g. `memset`, `fpsetsticky`, etc. and may also see routines of the type `mcount` listed – these are part of the profiling implementation and are also measured.
- NB `gprof` can produce a large amount of data – best to limit it to the most significant routines only (e.g. those that contribute >5% to runtime) – see man page for relevant switches.
- NB The `gprof` call-graph also gives an estimate of how much time was spent in the subroutines of each routine. This can suggest places where you might try to eliminate function calls that use a lot of time.

Combined Profile Output

- Another really useful property of `gprof` is that you can combine the output from several runs together
  - Hence get a more accurate depiction of where your program spends its time when different sets of parameters are used.
  - Need to rename the `gmon.out` files after each run of program to unique names and then call `gprof` with "`-s`" flag and multiple files for an overall summary, e.g.

```
gprof mycode -s gmon.1 gmon.2
  gmon.3 > mycode.gprof_summary
```

gcov example

```
% gcc --coverage ctest.c
% ./a.out
% gcov ctest.c
File 'ctest.c'
Lines executed:97.06% of 68
ctest.c:creating 'ctest.c.gcov'
% more ctest.c.gcov
 -:    0:Source:ctest.c
 -:    0:Graph:ctest.gcno
 -:    0:Data:ctest.gcda
 -:    0:Runs:1
 -:    0:Programs:1
 -:    1:/*simple program for time profiling. */
 <snip>
 -:   20:List make_nonempty(int first, List rest)
50132169:   21:{
50132169:   22: List list = (List) malloc(sizeof(struct
  AList));
50132169:   23: if (list == NULL) {
    #####:   24:   fprintf(stderr,"Couldn't allocate.\n");
```

Use '-f' to get function level coverage

Preamble – line numbers=0

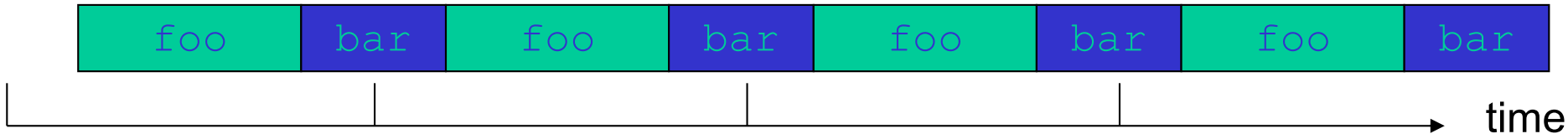
Not executable

Number of times executed

Not executed

Sampling Problems

- For a profile to be useful, the sampling must be sufficiently frequent
  - How frequent depends on clock speed – if processor is running at 2 GHz then will have 20 million clock cycles between 0.01 ms samples
  - As well as low resolution, this quantization can also cause sampling errors due to aliasing:



So although `foo` takes much more time than `bar`, the sampling frequency closely matches the cycling frequency of the two routines and hence we get a quantization error – sampling would suggest all the time is spent in `bar`!

Profile Problems

- For a profile to be useful, the code is usually compiled without optimisation. But with a good optimising compiler, the relative costs of certain routines can change significantly!
  - E.g. On hardware with fused multiply-add instruction pipes, an optimising compiler will get much better performance using these specialised instructions (typically turned on at `-O3` and above) – might be a factor of 6 or more! This may make the results of profiling (`-O0 -g`) misleading when applied to the production code (`-O3`) hence new `-Og` option .
- There are also issues when using dynamic libraries – is the time spent in the library visible to the profiler?

Profile Guided Optimisation

- Nevertheless, profilers ARE very useful
- Can give a quick overall representation of hot-spots in your code
  - Or your compiler (2-pass process)
    - Compile a special version to generate the data (intel: `-prof-gen` or `-fprofile-generate` for gnu) and then run to generate data
    - Recompile using this data (intel: `-prof-use` or `-fprofile-use` for gnu)
    - This is known as *profile guided optimisation* or dynamic optimisation – can produce substantial speedups compared to static optimisation

Know Your Compiler

- The compiler is of primary importance in coding
  - A good optimising compiler will know about the underlying hardware and how best to re-write your code for optimal efficiency – see next lecture
- Know what your compiler can do
  - Typically can have very large and complex set of flags/switches to fine-tune behaviour
  - READ THE MAN PAGE! On my old Alpha ‘man f90’ gave ~4000 lines of terse description of the command, including 93 principle switches and 493 sub-options!
  - Check carefully what is the default behaviour
    - Does your compiler automatically initialise all undeclared variables to zero or not? Common practice in old compilers, but rare today – is your code robust to this? Do you attempt to use any variables before assigning values to them?

SUMMARY

- Choose your algorithms carefully
  - And implement them carefully!
  - Will cover this next week …
- Profile your code and test for hotspots
  - Clue as to where to spend more effort
  - Use the compiler and other tools to help

Further Reading

- Chapter 2 of “Introduction to High Performance Computing for Scientists and Engineers”, Georg Hager and Gerhard Wellein, CRC Press (2011).
- `gprof2dot` converts `gprof` output at https://github.com/jrfonseca/gprof2dot
- `lcov` is a wrapper to `gcov` at http://ltp.sourceforge.net/coverage/lcov.php
- Google code: `gperftools` at https://github.com/gperftools/gperftools/wiki