Profiling Code

Before applying clever tricks it is often useful to find out what your program spends its time doing.

Tools for profiling:

profiler	cc flags	invocation
time		`/usr/bin/time prog`
top		`top`
prof	`-p`	`prof prog`
gprof	`-pg`	`gprof prog`
tcov	`-a`	`prog` then `tcov prog`
gcov	`-fprofile-arcs` `-ftest-coverage`	`prog` then `gcov prog`

Time is the simplest to use. Look out for low $\frac{\text{user}+\text{system}}{\text{real}}$ ratios or high % system times.

You can use top to see if your program is competing with others or to see if it goes through phases of paging or being IO bound.

Prof and gprof provide a summary of what functions are executed by sampling the program counter regularly. Gives an estimate of how long your program spends in each function.

Tcov and gcov tell you how many times each line of code was executed, but not how long was spent executing them.

Other profilers are available as a part of compiler suites. It can also be interesting to look at the system calls your program makes with `strace`, `ktrace` or `truss`.

Optimisation

- Concentrate on what takes time.
- Use a better algorithm/data structure.
- Experiment with compiler optimisation.
- Tune code.
- Parallelise code.
- Consider buying an upgrade/new computer.

Code Tuning

- Keep code simple - handle special cases separately.
- Consider using binary data with `read` and `write` for internal data files. Maybe even use `mmap`.
- Don’t make huge directories, make multilevel directories instead.
- Store what you can’t easily recompute.
- Don’t store what you can easily recompute.
- Try collecting subexpressions, especially if they involve function calls.

- If the compiler supports hints try using them.
- Factor constant code out of loops.
- Consider unrolling loops.
- Would macros or inline code help?
- It may be quicker to do calculations in local variables than to use their final destinations.
- Don’t be mean with variables - compilers are good at figuring out when they are no longer needed.
- Check it actually does some good!

Benchmarking

Benchmarking: measuring how good a computer is at a *particular* job.

Usually not the job you're doing.

There are various ill defined benchmarks such as MIPS and FLOPS.

Benchmarks age badly as CPU & compiler vendors figure out how to optimise for that code, or as cache sizes increase.

HPC benchmarks: Linpack, SPLASH, NAS, STREAM & HINT.

Industry benchmarks: SPECmarks, Winstones, Quake, . . .

Other benchmarks: Webstones, Worldstones, lmbench, TP, NFS, X, . . .

If you are going to benchmarking:

- the benchmark should represent what you do in the conditions it is done,
- make sure your problem/code isn’t optimised away,
- use identical code/problems on each system,
- set the rules and make sure everyone knows them,
- take quality of system, tools and vendor into account,
- check the output isn’t nonsense,
- benchmark your old system.

Assignment

Produce a table of how long each of the following takes on a machine of your choice.

Ints	`i++`
	`i = j + k`
	`i = j - k`
	`i = j * k`
	`i = j / k`
	`i = j % k`
Floats	`x = y + z`
	`x = y - z`
	`x = y * z`
	`x = y / z`
Doubles	`x = y + z`
	`x = y - z`
	`x = y * z`
	`x = y / z`
Convert	`x = i`
	`i = x`
	`sscanf("3.14159", "%lf", &pi)`
	`sprintf("%6.2f", pi)`
Functions	`x = drand48()`
	`x = sin(x)`
	`x = sqrt(x)`
	`free(malloc(sizeof(double)))`