ENEL464 - Profiling

1 Introduction

When writing a program to solve a problem, you often want to make the program run faster. The process of improving the runtime, or memory usage, or various resource usage, is called *optimising*. There are many ways to optimise your code: rewriting complex math expressions; using clever data structures to speed memory access; enabling built-in compiler optimisations; etc. However, before any of that can occur, you must first identify *where* to optimise. For a program that takes two hours, there is no point optimising a chunk of code that is responsible for 10 seconds of it (see Amdahl's law). You must first *profile* your code to identify where the CPU is spending most of its time (so called *hot-spots*) and *why* it takes so much time.

2 GProf

GProf is a Unix profiling system. It uses a combination of compile-time instrumentation (compiled into your program by GCC), and run-time sampling (pauses your program and records the program counter) to build reports on the execution details of a program.

There are three steps to using GProf:

1. Compile your code with profiling enabled.
2. Execute your program to collect runtime data.
3. Process the execution report and analyse the code.

2.1 Compiling with profiling enabled

From the man page of gcc:

```
-pg : Generate extra code to write profile information suitable for the
analysis program gprof. You must use this option when compiling the source
files you want data about, and you must also use it when linking.
```

So following this, we could modify the compilation command for the poisson program to be:

```
$ g++ -pg -g -o poisson poisson.c -lpthread
```

The `-g` option is required to include debugging symbols for annotating the program.

2.2 Executing with profiling

This is the same as executing the program normalling. Simply run

```
$ ./poisson -n 101 -i 100
```

Note: when using statistical profilers, the more snapshots taken, the more accurate the profiling will be.

After executing the program, a new file called 'gmon.out' should be present. This contains the runtime data and will be used in the next step.

2.3 Processing the runtime data

The GProf tool can now be used to analyse the output from the previous step.

```
$ gprof poisson gmon.out [> file_to_dump_to.txt]
```

This will list the most expensive functions in your program with total time used, time spent in child calls, number of calls, etc... GProf can also use this data to annotate the original source files:

```
$ gprof -A poisson gmon.out
```

Note: this does not modify the original files but prints the annotated version to stdout, just like the previous command.

Now that you know where the program is spending its time, you need to figure out *why*? Is it a computer architectural reason? Perhaps you could write your code differently to reduce the runtime? Can you organise your data differently?

3 Cachegrind

Cachegrind is a tool within the Valgrind suite. It works by simulating a modern processor cache and recording all of the memory accesses performed by your program. It will then list the various accesses to the level 1 data and instruction caches as well as the lower level shared cache.

Note: because cachegrind simulates *your cache, it runs slowly so do not run large computations with it.*

Cachegrind can be run using:

```
$ valgrind --tool=cachegrind ./poisson -n 51 -i 10
```

This will generate and display some statistics about the cache usage. Usually, for a computational problem, there will be very few instruction misses as the code is short and run many times. However, there will be significant data misses as your program traverses your simulation volume. Running cachegrind will also generate a 'cachegrind.out.<pid>' file, similar to that of the GProf output. This can be used to annotate your source code using

```
$ cg_annotate cachegrind.out.<pid> [> cg_output.txt]
```

The output from this command is quite wide, so storing it in a file is recommended.

4 Profiling with compiler optimisations

Compiler optimisations work by stripping out excess instructions, reordering, and simlifying the generated machine instructions. The only rule the optmiser follows is that the generated code must have the same output "as if" it had not been optimised at all. This gives the compiler a lot of leeway and often results in very confusing assembly outputs, see the note on optimisations for examples.

Because of this, certain functions may not exist in your compiled program anymore, and instructions no longer correspond to specific source code lines. This can be tricky when trying to profile as it becomes difficult to determine where the hot-spot is located. However, profiling without optimisations does not give you an accurate picture of real world program usage.

TLDR: you should profile your programs both with, and without, optimisation as this will help give you a full picture of how your program runs.