

Introduction to profiling and optimisation

**Stewart Martin-Haugh, Nikos Konstantinidis (UCL), Tim Scanlon
(UCL)**

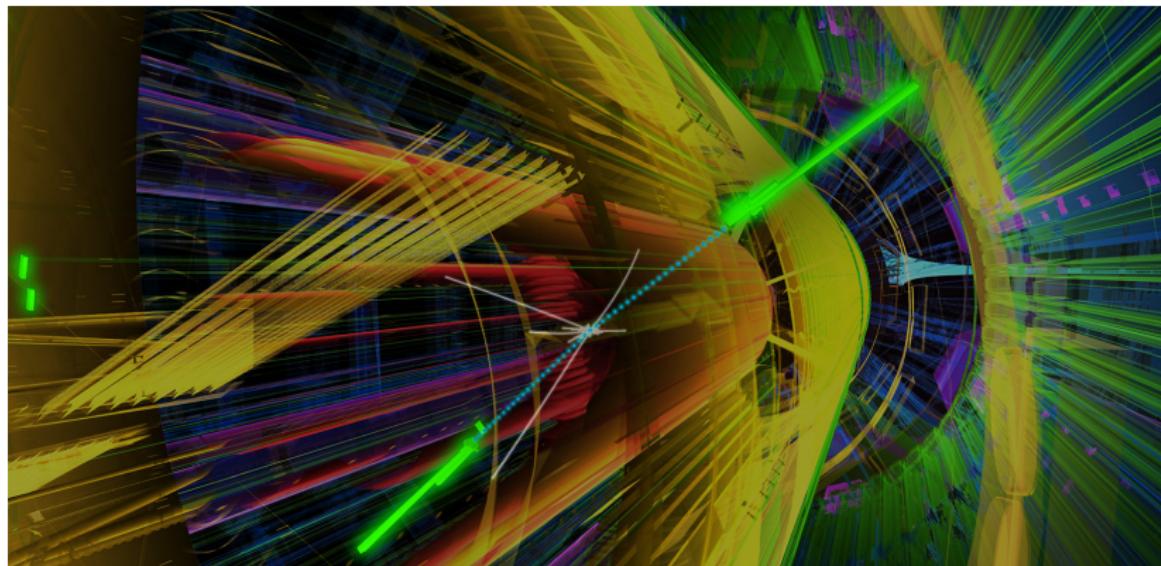
**Data Intensive Science Centres for Doctoral
Training Launch Event, Cardiff**

27/28 October 2017

Particle collider experiments

Large Hadron Collider is the best-known collider experiment, but there are others

- ▶ Collide protons together at high energy
- ▶ Measure fragments of proton-proton collision in specialised detectors
- ▶ Most collisions are low-energy - really interesting physics happens only 1 in a billion collisions



Data at collider detectors

Take e.g. ATLAS detector at LHC

- ▶ Each proton-proton collision record (output from all sub-detectors) \approx 1 megabyte
- ▶ \approx 40 million collisions every second \rightarrow 40 terabyte/second output rate
- ▶ Google data storage \approx 15 exabytes¹

$$\frac{15 \text{ exabytes}}{40 \text{ terabytes/second}} = 4.34 \text{ days}$$

- ▶ Even if we fill up Google's data centres, we only produce \approx 1000 Higgs boson events, and not many in "golden channels"
- ▶ In any case, we cannot read out the detector at 40 terabytes per second!

¹<https://what-if.xkcd.com/63/>

Data at collider detectors

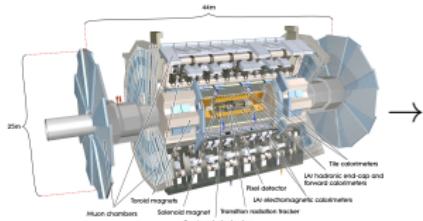
To summarise

- ▶ We physically **cannot** read out LHC data at the rate the LHC can provide
- ▶ If we could record all the data produced by the LHC, we would run out of worldwide disk space in a few days
- ▶ Once we filled up all the disk space worldwide, we still wouldn't have found the Higgs boson

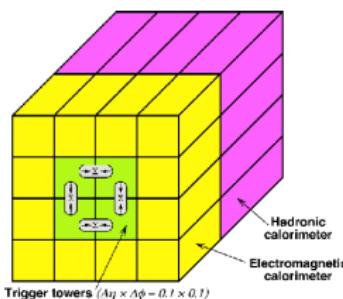
We need a **trigger** system - decide on the fly which collisions to accept and which to reject

Trigger systems for particle physics

- ▶ Goal: reduce the output data rate from 40 terabytes/second to e.g. ≈ 1 gigabyte/second
- ▶ Need hardware and software systems that read the detector output and decide whether the collision contains interesting physics



ATLAS detector



Fast hardware -
microsecond decision
time



Fast software -
millisecond decision
time

- ▶ For a more detailed introduction, see. e.g [here](#)

Software trigger systems for particle physics

- ▶ 40,000 CPU cores needed to read collisions and reduce the data rate to an acceptable output rate (1 gigabyte/second)
- ▶ Strong requirements for trigger software
 - ▶ Fast, smart pattern recognition algorithms - excellent rejection
 - ▶ Robust software - rare/no crashes
 - ▶ Reproducible/correct software - free/mostly free of bugs, no spurious rejection of interesting physics
- ▶ Trigger systems at LHC are **the** canonical data intensive science system!
- ▶ Focus on CPU and RAM performance today
- ▶ But feel free to ask me about robustness/reproducibility later on

This workshop

- ▶ You have an algorithm you want to use for a data intensive science application
- ▶ But you want the algorithm to run faster or use less RAM
 - ▶ Allow for more refined selection
 - ▶ Reduce computing requirements
- ▶ Need to measure and improve speed
- ▶ Focus on Linux, C++ applications

Warning

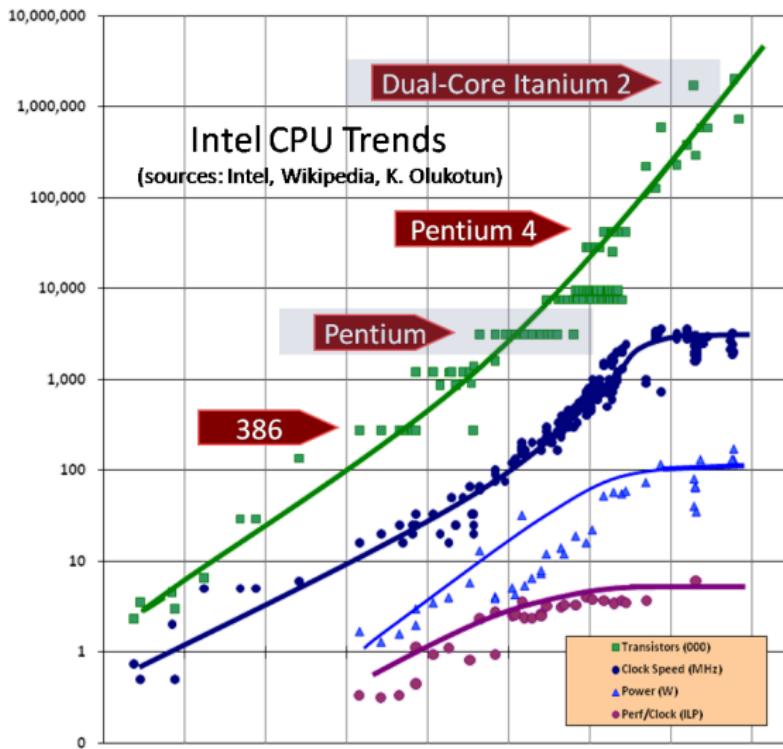
- ▶ Always implement something correct and readable (and save it to your version control system) first
- ▶ **Then** you can have fun optimising

Courtesy of reddit

```
//I don't know what I did but it works
//Please don't modify it
private int square(int n)
{
    int k = 0;
    while (true)
    {
        if (k == n*n)
        {
            return k;
        }
        k++;
    }
}
```

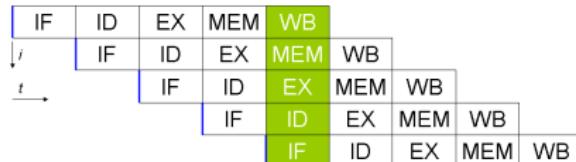
CPUs 101

- ▶ Moore's Law no longer holding for CPU clock speed (since \approx 2006)
 - ▶ Before then, could reasonably expect code to run faster on newer machines
- ▶ Memory has fallen behind CPU - big bottleneck frequently memory access
- ▶ More processing



Pipelining

- ▶ Pipelining allows processors to execute multiple instructions per clock cycle

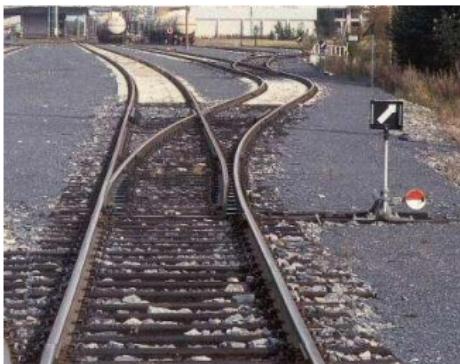


Five stage Instruction pipeline

- ▶ Only works for linear code
- ▶ Branching (e.g. `if` and `else`) is a problem
- ▶ Naively, can only execute one instruction to test branch

Branch prediction

- ▶ Module within CPU decides which branch to take (details [here](#))
- ▶ Allows CPU to pipeline code with branches
- ▶ Significant penalty if you take an unexpected branch - CPU has to load new code into pipeline



- ▶ Solution: remove branches if possible
- ▶ Unbalance your branches - 50/50 **if else** is harder to predict than e.g. 90/10 **if else**

Measuring runtimes

- ▶ Basic solution: time command
- ▶ user = time spent in your code
- ▶ sys = time spent in (Linux) kernel code
- ▶ real = sum of user + sys = **Walltime**

```
>time factor
 123456789098765432112345678987654321123456789
123456789098765432112345678987654321123456789: 3 3 3 3 3
    7 300239 122516477 13357815735863 147711262685101
real  0m1.172s
user  0m1.162s
sys  0m0.011s
```

- ▶ You (probably) only care about user
- ▶ If you're worried about system calls, you can use strace to see which ones are used (see e.g. [Julia Evans strace zine](#))

Walltime

- ▶ Walltime is the most important number for profiling, but also the most difficult to measure accurately
 - ▶ Varies with CPU
 - ▶ Some variation from operating system
 - ▶ Penalty for running in a virtual machine



Measuring runtimes

- ▶ Next level in complication: debugger
- ▶ Start your program, then randomly interrupt it a few times and see which function it's in

```
^C
Thread 1 "program" received signal SIGINT, Interrupt.
0x00007f8d81f09b55 in SiSpacePointsSeedMaker_ATLxk::
    production3Sp() ()
    from libSiSpacePointsSeedTool_xk.so
(gdb) bt
#0  0x00007f8d81f09b55 in production3Sp() ()
    from libSiSpacePointsSeedTool_xk.so
#1  0x00007f8d81f0baaa in production3Sp() ()
    from libSiSpacePointsSeedTool_xk.so
#2  0x00007f8d81f0bc0b in find3Sp() ()
    from libSiSpacePointsSeedTool_xk.so
```

- ▶ This is the **callstack**
- ▶ If your program spends 90% of its time in function X, you have a 90% chance of catching it

Sampling profilers

- ▶ Congratulations, you've made a basic sampling profiler!
- ▶ Sample = interrupt, look at the call stack

```
^C
Thread 1 "python" received signal SIGINT, Interrupt.
0x00007f8d81f09b55 in costlyFunction() ()
    from costlyNumerics.so
(gdb) bt
#0  0x00007f8d81f09b55 in costlyFunction() ()
    from costlyNumerics.so
#1  0x00007f8d81f0baaa in frameworkCode() ()
    from frameworkCode.so
#2  0x00007f8d81f0bc0b in main() ()
    from program.so
```

Cost

- ▶ costlyFunction() (top of the stack trace): where program was when halted
 - ▶ “Self cost”
- ▶ frameworkCall(), main(): call the function doing the work
 - ▶ “Total cost”
- ▶ Self cost \leq total cost
- ▶ Focus optimisation efforts on functions with highest self-cost

```
#0 0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
#1 0x00007f8d81f0baaa in frameworkCall() ()
   from frameworkCode.so
#2 0x00007f8d81f0bc0b in main() ()
   from program.so
```

- ▶ Mysteriously, Python stack traces start from the bottom
- ▶ Some would argue this is the one true profiler

Exercise 1: basic gdb

- ▶ Find number of primes below 1 million
- ▶ What is the bottleneck?

```
gdb ./Exercise_1
...
(gdb) run
*Press Ctrl-C*
(gdb) backtrace
#0 0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
#1 0x00007f8d81f0baaa in frameworkCode() ()
   from frameworkCode.so
(gdb) continue
*Press Ctrl-C*
(gdb) backtrace
#0 0x00007f8d81f09b55 in costlyFunction() ()
   from costlyNumerics.so
#1 0x00007f8d81f0baaa in frameworkCode() ()
   from frameworkCode.so
```

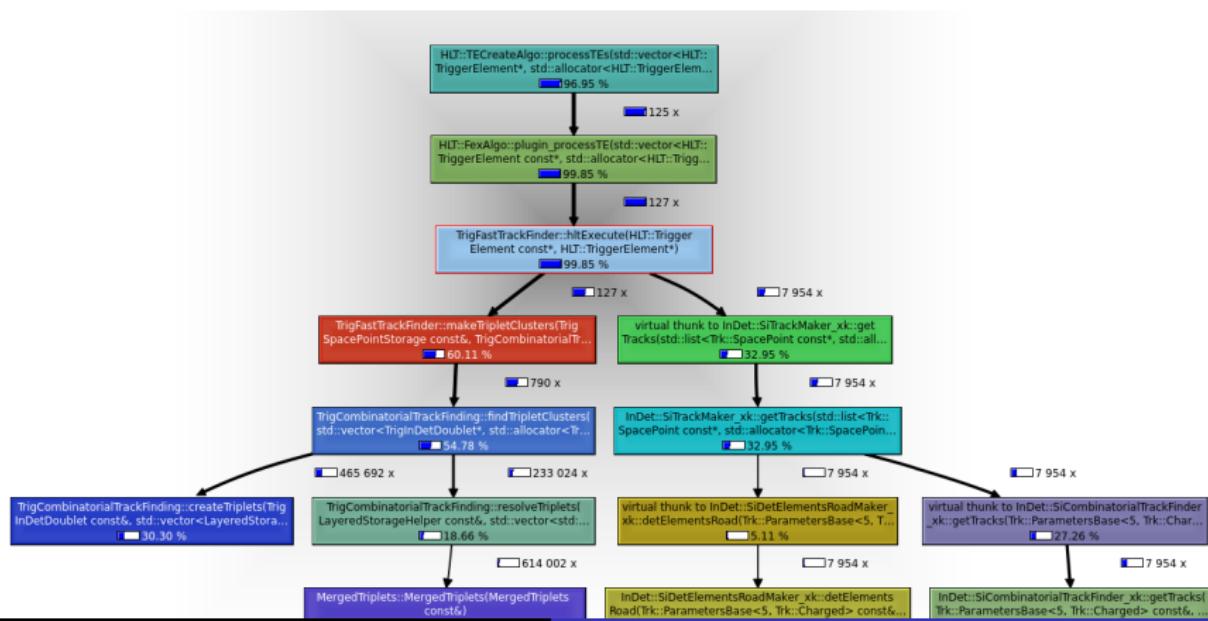
Exercise 1

Things to try

- ▶ Change optimisation flags -O0, -O1, -O2, -O3
- ▶ Swap clang for g++
- ▶ Keep code the same for now (even if you think/know you can improve it)
- ▶ Time code, interrupt in gdb

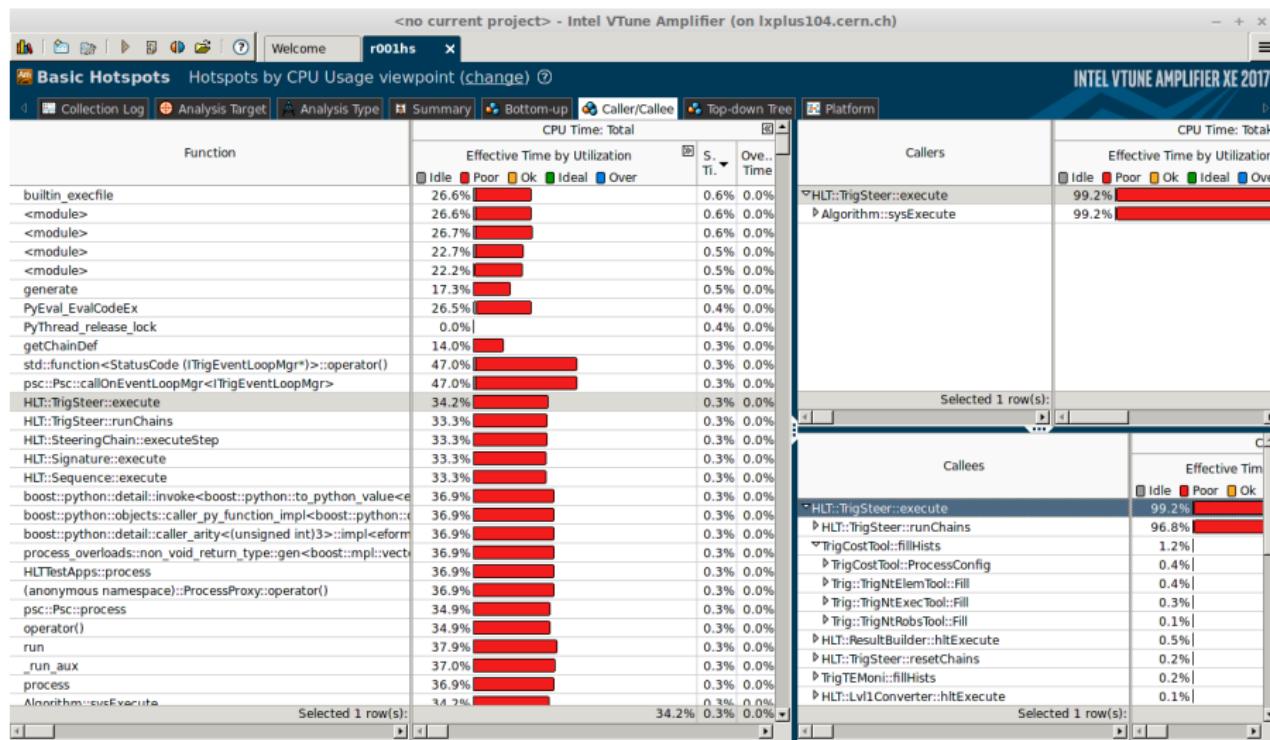
Sampling profilers

- ▶ Automate the call stack sampling procedure, generate a call graph (can be nicely explored in KCacheGrind)
- ▶ gperftools, Intel VTune, igprof
- ▶ Can also assign cost to lines of code (but take with a pinch of salt)



VTune

- ▶ Intel VTune is an excellent (but proprietary) tool
- ▶ Access possible through e.g. CERN



Emulation

- ▶ Valgrind has the callgrind tool in particular
- ▶ Emulates a basic modern CPU, with level 1, level 2 caches,
branch prediction (somewhat configurable)
- ▶ Runs slowly
- ▶ Information about cache misses and branch misprediction
- ▶ Produces output suitable for KCacheGrind

Instrumentation

- ▶ **perf** is now the gold standard - sampling and instrumenting
- ▶ Part of Linux kernel (best results with new kernels)
- ▶ Monitor performance monitoring counters (PMCs)
- ▶ VTune also has access to these
 - ▶ Some features require root access

```
perf stat -d program
      10 152 172 182      cycles:u          #
            3,451 GHz          (49,86%)
      14 584 154 073      instructions:u      #
            1,44  insn per cycle      (62,43%)
      2 318 605 154      branches:u          #
            788,130 M/sec      (74,93%)
      44 768 463      branch-misses:u      #
            1,93% of all branches      (75,00%)
      4 116 170 377      L1-dcache-loads:u      #
            1399,150 M/sec      (74,18%)
      167 821 302      L1-dcache-load-misses:u  #
            4,08% of all L1-dcache hits      (25,06%)
      45 252 042      LLC-loads:u          #
            15,382 M/sec      (24,89%)
      8 794 669      LLC-load-misses:u      #
```

Profiling thoughts

- ▶ It's a cliche, but the biggest improvements usually come from changing algorithm, not minor changes to code
- ▶ [Compiler Explorer](#) is a great tool to get a feel for what the compiler is doing to your code
- ▶ Don't second-guess modern compilers
- ▶ Many profilers available
- ▶ Measure and benchmark

Compiler optimisation

- ▶ Standard compilers (GCC, clang) can do a lot of optimising for you!
 - ▶ -O0 = no optimisations applied
 - ▶ -O1, -O2 = basic, safe optimisations applied
 - ▶ -O3 = more experimental optimisations applied (for the brave)
- ▶ O2 is a good optimisation reference level
- ▶ Measure at O2/O3 **before** optimising by hand
- ▶ Fine-tuned optimisation options available - check GCC/clang documentation for details

CPU optimisation

- ▶ Once you've identified which part of your code takes the most time, you can start optimising
- ▶ Strategies are somewhat language-dependent, but some general points always true
- ▶ Compiled languages (C++, Fortran) faster than interpreted (Python, Ruby)
- ▶ Standard libraries (FFTW, BLAS, Eigen) likely faster than your own code - don't reinvent the wheel!

Floating-point operations

- ▶ Avoid square root if possible
- ▶ Avoid division if possible

```
y=x/5.0; //Bad  
y=x*0.2; //Good
```

- ▶ Rearrange calculations to minimise number of operations
- ▶ Compiler won't necessarily do this for you (floating point rules)

```
y = d*x*x*x + c*x*x + b*x + a; //Bad  
y = x*(x*(x*d+c)+b) + a; //Good
```

Higher mathematical functions

- ▶ Trigonometric functions are slow
 - ▶ Consider using an optimised library (see e.g. [VDT](#))
 - ▶ Trigonometric identities can help you
- ▶ For linear algebra, definitely use a library (e.g. [Eigen](#))

Loops

- ▶ Don't recalculate within loops: move code outside
- ▶ Consider storing frequently calculated values

```
for (i = 0; i < 50; i++) {  
    for (j = 0; j < 50; j++) {  
        x = sin(5*i) + cos(6*j);  
        //Can move sin() into earlier loop  
    }  
}
```

Putting it all together

- ▶ Sometimes you can remove branches and reduce number of operations all at once

```
//OLD
if ( h >= 0.) {
    h = min( max( 0.25*h, pow((x / y), 0.25)*h), 4.*h);
} else {
    h = max( min( 0.25*h, pow((x / y), 0.25)*h), 4.*h);
}
//NEW
h = h*min( max( 0.25, pow((x / y), 0.25)), 4.);
```

Algorithmic complexity

- ▶ If possible, stick to standard algorithms (e.g. C++ `std::sort`) instead of writing your own
- ▶ If the algorithm is a hotspot, consider trying out different algorithms (note, flashing lights)

Data structures

- ▶ Worth thinking about which data format fits your problem
- ▶ In C++, `std::vector` is probably a good fit

Points to remember

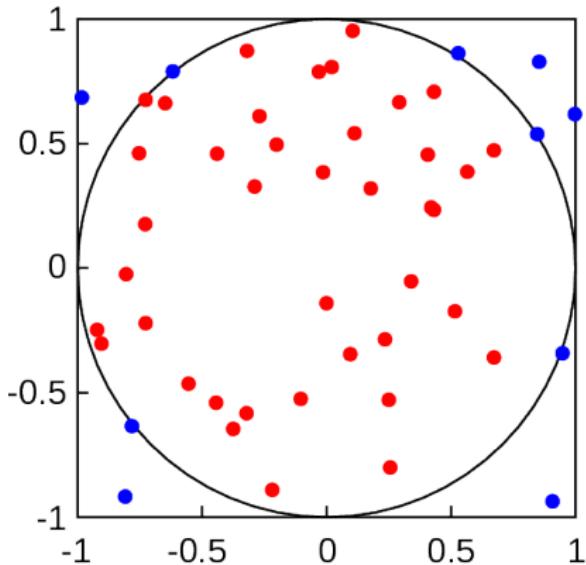
- ▶ Profiling and reasoning about code cannot tell you if you're using the wrong algorithm for your problem
- ▶ Writing your own implementation of something is an excellent way to learn, even if you never use it
- ▶ Correctness must come before optimisation

Back to exercise 1

- ▶ Fix identified bottleneck
- ▶ Measure result

Exercise 2

- ▶ Classic demonstration of Monte Carlo integration
- ▶ Throw darts at a circle inscribed in a square
- ▶ Ratio of darts hitting the blackboard tends to $(\pi r^2)/(4r^2) = \pi/4$
- ▶ Big difference in O0 vs. O2 runtime - where is the bottleneck?



Memory 101

Programs have access to two pools of memory: stack and heap

- ▶ Stack:

- ▶ Small amount of memory associated with program
- ▶ Fast to access - can be e.g. in CPU L1 cache
- ▶ E.g. variables in a function

```
int f(int x) {  
    int i = 55;  
    return x + i;  
}
```

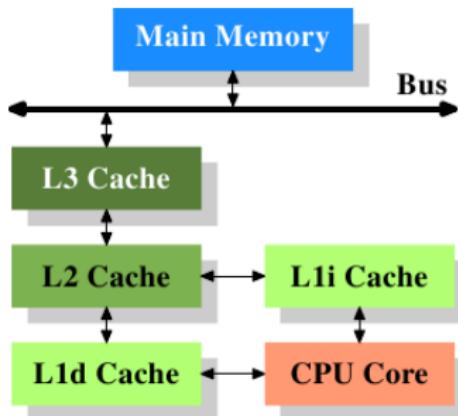
- ▶ Heap:

- ▶ Slower to access than stack
- ▶ Can be **dynamically** allocated
- ▶ If you don't free up memory, this is where
- ▶ All the RAM available on the machine (if it runs out, it will use hard drive - v slow!)

```
int g(int x) {  
    int* i = new int(55); //On heap  
    return x + *i;  
    //Memory for i not given back to OS - leak  
}
```

Memory 101

- ▶ RAM has not improved at the same speed as CPU clock speed
- ▶ Fast RAM is still very expensive - store recently used data in series of caches before checking main RAM
- ▶ Ideal case - all data found in L1 cache
- ▶ Worst case - all data retrieved from main memory
- ▶ Improving memory access can make a big difference in speed - unfortunately, very difficult in practice



Basic cache architecture diagram from
[What Every Programmer Should Know About Memory](#)

Memory profiling

- ▶ Using too much memory is bad for two reasons
 - ▶ Eventually you run out (e.g. memory leak)
 - ▶ Allocating memory has a significant CPU cost - higher if your data doesn't fit in e.g. L1 cache
 - ▶ A single large allocation is cheaper than several small allocations
- ▶ Better to access memory in order - data-locality
 - ▶ Appropriate data structures help with this

Different allocators

- ▶ Your program will not just receive the memory it asks for when it asks for it
- ▶ Allocator decides how much to request at a time and how much should be contiguous
- ▶ **glibc** used by default
- ▶ Others available, particularly **jemalloc** (Facebook) and **tcmalloc** (Google)
- ▶ No need to recompile, just preload
- ▶ May work better for your memory access pattern than glibc - free speedup!

```
LD_PRELOAD=/usr/lib/libtcmalloc.so.4 ./my_program
```

Finding big allocations

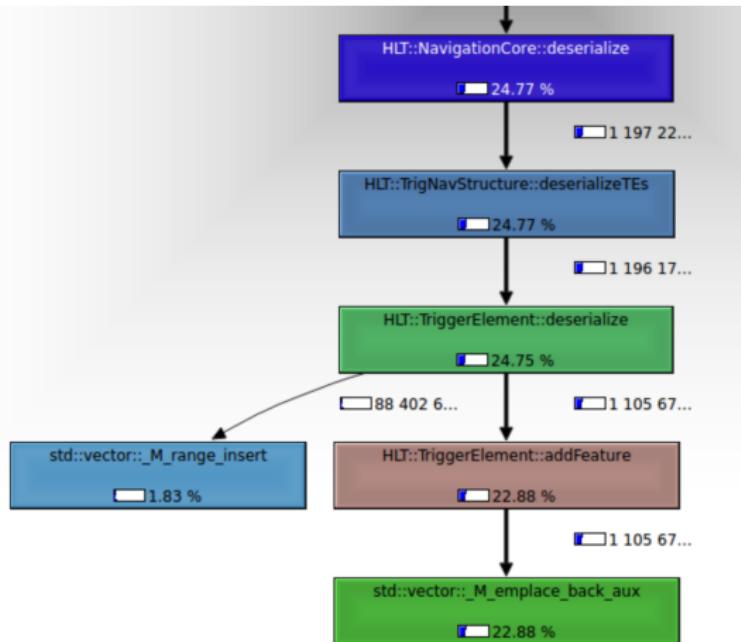
- ▶ Scenario: your program is running out of memory
- ▶ How to track down large (e.g. 1 GB) allocations?
- ▶ `tcmalloc` provides a printout when this happens

```
tcmalloc: large alloc 2720276480 bytes == 0x73eda000 @  
tcmalloc: large alloc 2720276480 bytes == 0x2a96f0000 @  
tcmalloc: large alloc 2720276480 bytes == 0x34b932000 @
```

- ▶ Add a breakpoint at `ReportLargeAlloc`

Heap profilers

- ▶ `jemalloc` and `tcmalloc` both come with low-overhead profilers to analyse which functions allocate most memory
- ▶ Output can be interpreted much as with a call-graph



Conclusions

- ▶ A small amount of profiling/optimisation knowledge can dramatically improve your application performance
 - ▶ Profiling is more important than optimisation
- ▶ Advanced techniques useful once you've done the easy bits

Useful references

- ▶ Numerical Recipes in C++, Second Edition, Press, Teukolsky, Vetterling and Flannery
- ▶ Brendan Gregg [Linux performance resources](#)
- ▶ Agner Fog [optimisation resources](#) (C++ focused)
- ▶ Ulrich Drepper
[What Every Programmer Should Know About Memory](#)