Computing

Stewart Martin-Haugh (RAL)

RAL Graduate Lectures
13 May 2020

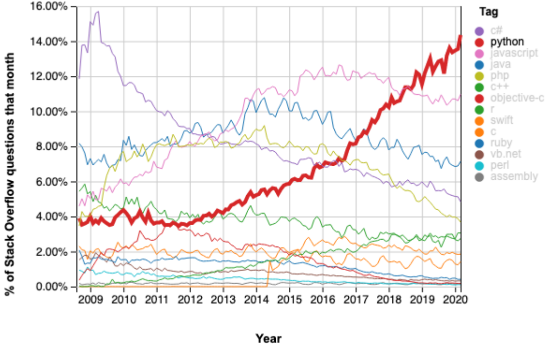
Science and
Technology
Facilities Council

Introduction

- Computers offer two principal advantages over humans
  - Correctness
  - Speed

Programming languages

- Python and C++ most popular in particle physics
- Python increasingly popular in the outside world
- C++ isn't going anywhere, but doesn't seme to be growing much



Source: Stack Overflow

Python

- Python is particularly good for scripts
  - Easy to read and write
  - Slower than C++ (interpreted vs. compiled) but can call C++, Fortran if speed is necessary
- Most experiment frameworks use Python to glue bits of C++ together (Athena, CMSSW)
- Increasing interest in common and open-source tools - see e.g. PyHEP workshops

C++

- Big, complicated language - multiple ways to do things
  - C (arrays, pointers, functions)
  - Classic C++ (`new, delete, classes`)
  - Templates (`template <class T>`)
  - Modern C++ (`std::unique_ptr, for (auto x: y)`)
- In large codebases (e.g. experiment offline software) all of these will co-exist (happily/unhappily)
- Evolving language - C++11 standard brought modern C++, new standards every 3 years
- C and C++ are probably the best-supported ways to write fast code

Other languages

My views:

- Julia: aims to be like a fast Python
- Rust: aims to be like a correct-by-default C++

A rustacean

Correctness

- How do you verify a calculation that you can't do any other way?
  - Find a version of your calculation where you know what the answer **should be**
  - Change something that should not change the answer and check the result

Software engineering[1]

- Everything around the actual writing of code
- Debugging, testing, packaging, operating environments
- Software engineering is an entire discipline, but just one of the skills you need in HEP

[1] Term coined by Margaret Hamilton, lead programmer for the Apollo Mission guidance computer

Debugging thoughts

Brian Kernighan (C, Unix etc):

- The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.
- Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Rubber ducks and teddy bears:

- Explain your problem to anyone at all: doesn't have to be an expert programmer
- Doesn't have to be a living being: rubber ducks and teddy bears
- Draft an email to an expert explaining what you think is happening

Debugging thoughts

1. Reproduce the problem and generate exact setup instructions (environment, command, extra software on top)
2. Find a fix
3. Establish that the fix has no obvious side effects (e.g. an unrelated test still gives the same result)

```
#This fails with an error - try it out if you're on
    ATLAS
asetup master, r2020-05-07T2140, Athena
test_trig_data_v1Dev_deps_build.py
```

Static analysis

- There is no way to tell, in general, what a program will do beyond running it (Turing completeness)
- But you can tell that some things will crash
- Automated tools to help

Python

- flake8 (combination of pep8 style guide and static analysis)
- vulture

C++

- Compiler warnings (enable as many warnings as possible, don't consider your code complete until they are all fixed)
- cppcheck (fast!)
- clang-analyzer
- Coverity (slow...)

- Add these to your testing (more on testing later)

Dynamic analysis

Add extra information to your compiled binary that makes it crash or warn when things have gone wrong

- Extra print statements
- Debug symbols: add line numbers etc to your crashes `g++ -g`
- GDB and friends: learn to use a debugger, read stack traces
- Valgrind: run in a virtual environment that flags memory errors
- Sanitizer tools (part of Clang project): tell you if you're writing outside allowed memory, using uninitialised data

Memory errors (C++)

- Most common: reading/writing beyond an array

```
vector<float> vec;
vec.push_back(1);
std::cout << vec[2] << std::endl;
```

- This will read random memory: could crash, could print 0, could print `1.1755-e38`
- Cause irreproducible results
- Can catch by compiling with AddressSanitizer or running under Valgrind
- `vec.at(i)` will throw an exception if you go past the end of the vector, but it's a bit slower (unlikely to matter for you)

Floating point errors

- `1.0/0.0, 0.0/0.0, sqrt(-1): inf, NaN, NaN` - all of these are floating point exceptions (FPEs)
- Since any mathematical operation involving `NaN` gives `NaN`, it can pollute all your results and should be fixed
- Floats are most precise close to 0 - avoid using very small and very large numbers
- Strongly recommend reading Floating point demystified for a more thorough understanding

Testing

Types of tests

- System/integration tests: test that different components work together, usually at large-scale (e.g. run full reconstruction over a large dataset)
  - Easiest to write: mirror what the code is eventually meant to o
- Regression tests: check that a fixed bug does not reappear
  - Harder to write: need to keep track of test cases
- Unit tests: check functionality at $\approx$ function level - should be quick to run
  - Hardest to write: think about what each function does, may need extra code, fake input data
- Add static analysis to your tests

Writing any tests at all is good, and you get better at writing them with practice

Unit test example

```
def squ(x):
  return x**2 + 1

def test_squ():
  assert squ(42)==1764
test_squ()

#Output:
Traceback (most recent call last):
  File "square.py", line 7, in <module>
    test_squ()
  File "square.py", line 5, in test_squ
    assert(squ(42)==1764)
AssertionError
```

- This kind of strategy is useful for checking that a change that shouldn't change the output, doesn't change the output
- Python `assert` is very useful in general: litter your code with it
- Consider unit testing libraries (e.g. GoogleTest, Python unittest)

Random and comprehensive testing

- Testing with truly random inputs hasn't been used much in particle physics
- But we do run code over large MC and data samples with built-in randomness
  - So we are testing with a random distribution - nature/theory + systematic uncertainties + detector response
- Fuzz testing is truly random testing, very important for computer security

▲ Margaret Hamilton in 1969 with the source code her team developed for the Apollo missions. Photograph: Science History Images/Alamy

Did your life as a software engineer and a mother ever collide?
Often in the evening or at weekends I would bring my young daughter, Lauren, into work with me. One day, she was with me when I was doing a simulation of a mission to the moon. She liked to imitate me - playing astronaut. She started hitting keys and all of a sudden, the simulation started. Then she pressed other keys and the simulation crashed. She had selected a program which was supposed to be run prior to launch - when she was already "on the way" to the moon. The computer had so little space, it had wiped the navigation data taking her to the moon. I thought: my God - this could inadvertently happen in a real mission. I suggested a program change to prevent a prelaunch program being selected during flight. But the higher-ups at MIT and Nasa said the astronauts were too well trained to make such a mistake. Midcourse on the very next mission - Apollo 8 - one of the astronauts on board accidentally did exactly what Lauren had done. The Lauren bug! It created much havoc and required the mission to be reconfigured. After that, they let me put the program

Continuous integration

- Run as many tests as you can for each change you make (i.e. a merge or pull request)
- Catch problems before they're part of the main codebase
- GitLab CI, Jenkins, Travis CI

```
#example gitlab CI for LaTeX
stages:
    - build

build:
    image: thomasweise/docker-texlive-full
    stage: build
    script:
    - apt update -y
    - apt install -y biber
    - make
    artifacts:
        paths:
            - "*.pdf"
        expire_in: 1 week
```

Containers

Useful for many reasons: debugging, versioning, sharing code

- Run a lightweight virtual operating system on top of your real OS
- Similar to a virtual machine
- Easily run Linux programs on Mac, Windows
- A description of an environment that someone else can run
- Can run on the Grid
- Can run Unix v1 (1972)

Conclusions about correctness

- Have only had time to cover some basics
- Questions?

Now on to speed!

Fast computing

High throughput computing

- Can parallelise and buffer data for later processing
- LHC, SKA - **this talk**
- Maximise throughput = events/second

Low latency computing

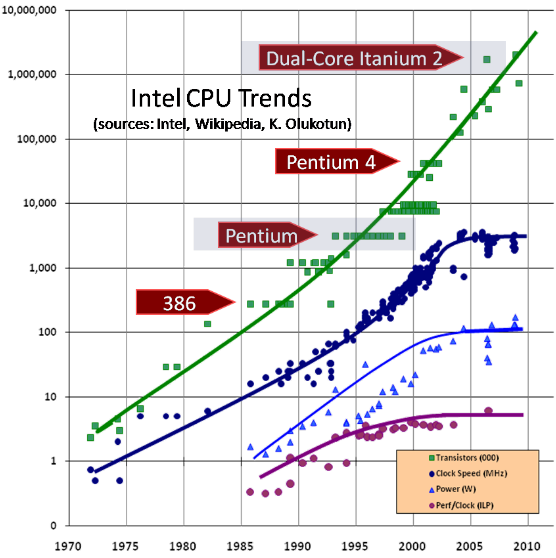
- Pointless or impossible to buffer
- High frequency trading, autonomous vehicles

High performance computing

- Problems that don't parallelise easily - supercomputer
- Climate modelling
- Fast connections between processors, lots of RAM

CPUs 101

- Moore's Law no longer holding for CPU clock speed (since ≈ 2006)
- Memory has fallen behind CPU - big bottleneck frequently memory access
- More processing power available through parallelism

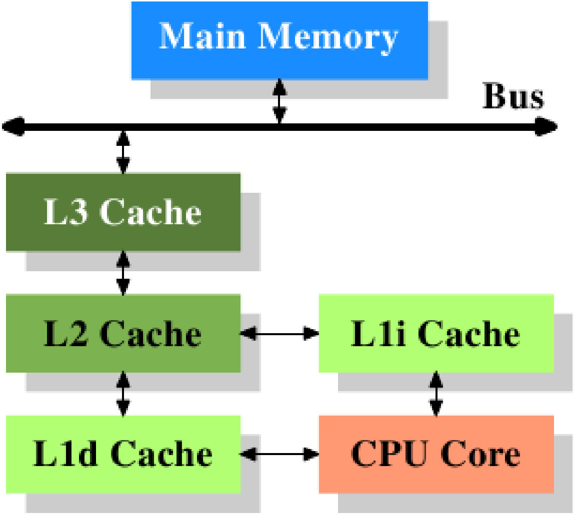


Source: Herb Sutter

CPUs 101

Fast memory is expensive

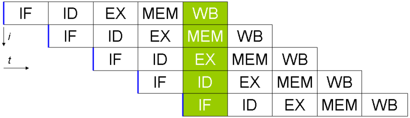
- Fastest memory kept in L1 cache, slower in L2 etc
- Slow memory in RAM
- Slowest of all is hard disk
- Cache miss = retrieving data from a different cache



Source: What Every Programmer Should Know About Memory

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
- Can't load anything past the branch point

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`**
- Sort data (see most popular ever Stack Overflow question)

Parallelism/concurrency

- An entirely parallelisable calculation is referred to as **embarrassingly parallel**
  - Event generation: every collision has no dependency on the previous
  - Simulate each collision on a separate CPU: scale to number of CPUs available
- Most calculations have a parallel and serial component, which limits the speedup

Parallel and serial components

Silly example

- Making a car requires 1000 identical parts: each part takes 1 minute to make
- The 1000 identical parts must be assembled in a final step: this takes 60 minutes

Serial path

- 1000 parts assembled by a single worker + final assembly = 1000 + 60 minutes = 1060 minutes

Parallel path

- 1000 parts assembled by different workers simultaneously + car assembly = 1 + 60 minutes = 61 minutes

Maximum speedup

- $1060/61 = 17.4$

No further gains without improving the final assembly (serial part)

- Amdahl's law turns this kind of reasoning into a formal statement

Parallelism

Flynn's taxonomy (1966)

- SISD (Single instruction, single data) single-threaded operation
- SIMD (Single instruction, multiple data) vector operations
- MISD (Multiple instruction, single data) fairly rare in practice
- MIMD (Multiple instruction, multiple data) multi-threaded operation

Vectorisation/SIMD

- Modern CPUs can execute the same instructions on multiple data simultaneously

```
std::array<int, 100> int_array;
for (unsigned int i = 0; i < 100; i++) {
        int_array[i] = i*4;
}
```

- Different architectures depending on CPU generation: MMX, SSE, AVX
  - Code generated for one instruction set will not work with another

Auto-vectorisation

- Compiler can generate appropriate vector instructions for loops etc
- Will not always apply it if not beneficial - see backup for an example where GCC and Clang disagree

```cpp
#include <array>
#include <iostream>

int main() {
        std::array<int, 100> int_array;
        for (unsigned int i = 0; i < 100; i++) {
                int_array[i] = i*4;
        }
        std::cout << int_array[10] << std::endl;
        return 0;
}
```

```
clang++ -msse4.2 -std=c++11 vec.cpp -O2 -Rpass=loop-vectorize
vec.cpp:6:2: remark: vectorized loop (vectorization width: 4,
    interleaved count: 1) [-Rpass=loop-vectorize]
        for (unsigned int i = 0; i < 100; i++) {
        ^
#similarly: g++ -std=c++11 vec.cpp -O3 -fopt-info-vec
```

Vectorisation by hand

- Autovectorisation is fragile: re-order your code and it can disappear
- Can write using vector intrinsics: functions that act on arrays of data and operate accordingly
- Resulting code will not compile on different CPU type (e.g. ARM, older Intel/AMD)
- More complicated to write

void _mm_2intersect_epi32 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)	vp2intersectd
void _mm256_2intersect_epi32 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)	vp2intersectd
void _mm512_2intersect_epi32 (__m512i a, __m512i b, __mmask16* k1, __mmask16* k2)	vp2intersectd
void _mm_2intersect_epi64 (__m128i a, __m128i b, __mmask8* k1, __mmask8* k2)	vp2intersectq
void _mm256_2intersect_epi64 (__m256i a, __m256i b, __mmask8* k1, __mmask8* k2)	vp2intersectq
void _mm512_2intersect_epi64 (__m512i a, __m512i b, __mmask8* k1, __mmask8* k2)	vp2intersectq
__m512i _mm512_4dpwssd_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)	vp4dpwssd
__m512i _mm512_mask_4dpwssd_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)	vp4dpwssd
__m512i _mm512_maskz_4dpwssd_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)	vp4dpwssd
__m512i _mm512_4dpwssds_epi32 (__m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)	vp4dpwssds
__m512i _mm512_mask_4dpwssds_epi32 (__m512i src, __mmask16 k, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)	vp4dpwssds
__m512i _mm512_maskz_4dpwssds_epi32 (__mmask16 k, __m512i src, __m512i a0, __m512i a1, __m512i a2, __m512i a3, __m128i * b)	vp4dpwssds
__m512 _mm512_4fmadd_ps (__m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)	v4fmaddps
__m512 _mm512_mask_4fmadd_ps (__m512 src, __mmask16 k, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)	v4fmaddps
__m512 _mm512_maskz_4fmadd_ps (__mmask16 k, __m512 src, __m512 a0, __m512 a1, __m512 a2, __m512 a3, __m128 * b)	v4fmaddps
__m128 _mm_4fmadd_ss (__m128 src, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)	v4fmaddss
__m128 _mm_mask_4fmadd_ss (__m128 src, __mmask8 k, __m128 a0, __m128 a1, __m128 a2, __m128 a3, __m128 * b)	v4fmaddss

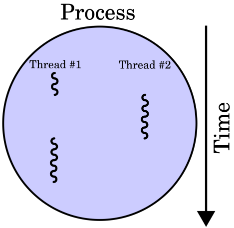
Vectorisation by someone else's hand

- Easiest solution: use a library written by an expert
- E.g. for `cos()`, `exp()`, `atan2()`
  - CERN VDT
  - Intel and AMD mathematical function libraries (recipe in backup)
- For matrix algebra
  - Eigen

Multiple instructions, multiple data (MIMD)

MIMD: multi-threading

- A thread is a sub-program controlled by your main program
- Operating system decides when and on which CPU they run
- Thread order is non-deterministic: can lead to difficult bugs
  - Race conditions, deadlocks, irreproducible output
- Usually one thread per CPU
- Swapping between threads on one CPU: "hyper-threading"



from Wikipedia

Threading example with OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
  #pragma omp parallel num_threads(4)
  {
    int thread = omp_get_thread_num();
    int total = omp_get_num_threads();
    std::cout << "Greetings from thread " << thread << "
         out of " << total << std::endl;
  }
  std::cout << "parallel for ends." << std::endl;
  return 0;
}
g++ -fopenmp test_omp.cpp && ./a.out
Greetings from thread 1 out of 4
Greetings from thread 0 out of 4
Greetings from thread 3 out of 4
Greetings from thread 2 out of 4
```

- Must program in a way that avoids dependence on thread order or data local to each thread

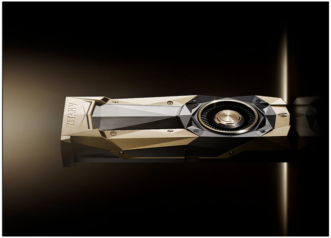
Threading libraries

- Intel Threading Building Blocks: used by ATLAS, CMS, LHCb for multi-threaded data processing
- OpenMP is best for simpler situations with limited relationships between threads
- Other frameworks exist: e.g. HPX is gaining some momentum

Many more details on threading available in Graeme Stewart's concurrency lectures (with worked examples)

Brief digression on GPU programming

- Initially designed for graphics calculations: matrix and vector operations
- Tailored towards embarrassingly parallel problems
- Increasingly used for scientific applications, machine learning
- Several competing options for programming
  - CUDA for NVidia
  - HIP for AMD
  - OpenCL, SYCL: multi-platform
- HEP software moving towards these, but difficult/labour-intensive to port
- Increasingly popular for supercomputers etc: dragging HEP that way



Parallelism conclusions

- Vectorisation and multi-threading are harder to work with than single-threaded programming
  - But necessary if you want to get the highest possible performance
- Even if you don't need the best performance, you can still apply some of this through libraries

Next topic: measuring performance

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 1234567890987654321123456789333333333
1234567890987654321123456789333333333: 3 3 3 23 43
   27062723775121 1708375824282413291

real    0m0.363s
user    0m0.321s
sys     0m0.000s
```

- You care about real, but you can only affect 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
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
Program 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

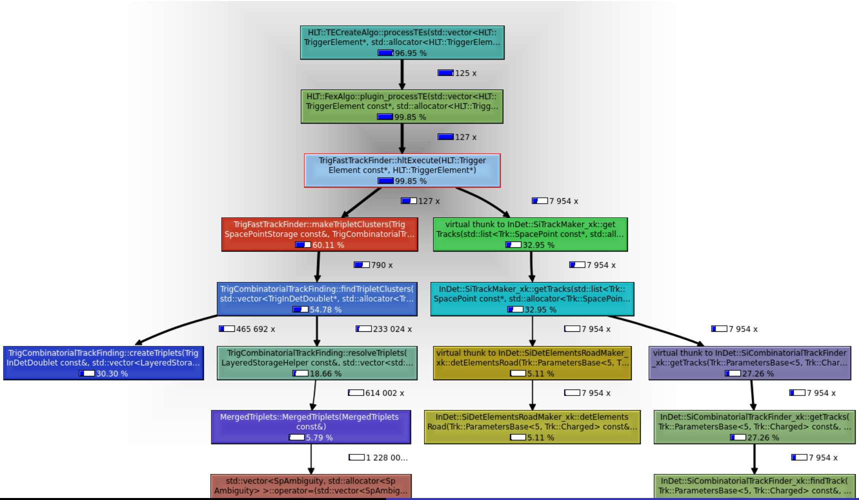
- costlyFunction() (top of the stack trace): where program was when halted
  - "Self cost"
- frameworkCall(), main(): call the function doing the work
  - "Total cost"
- Self cost $\leq$ total cost
- Focus optimisation efforts on functions with highest self-cost

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

- Some would argue this is the one true profiler

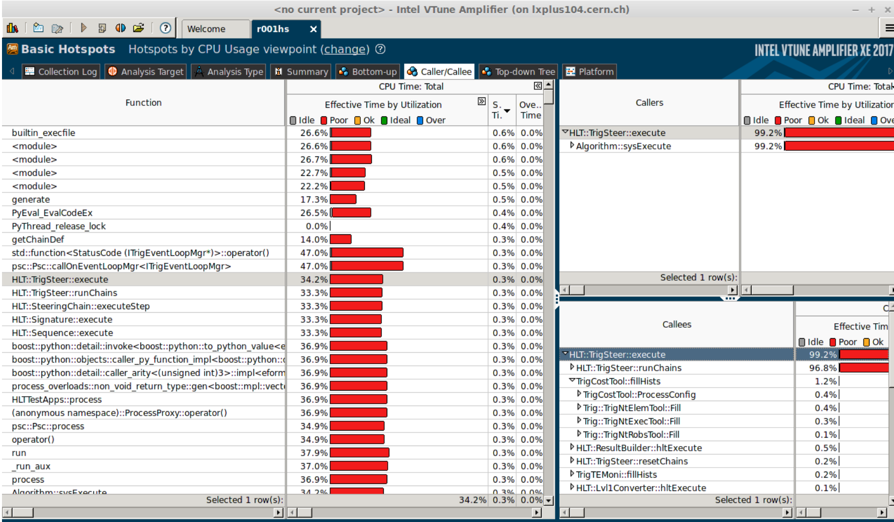
Sampling profilers

- Automate the call stack sampling procedure, generate a call graph (can be nicely visualised 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 tool
- Free to download, if you register with Intel



Instrumentation

- High-level languages (e.g. C++) have inbuilt timing facilities:

```
using namespace std;
using namespace std::chrono;
auto start_time = high_resolution_clock::now();
doSomething();
auto end_time = high_resolution_clock::now();
cout << "Time: " << duration_cast<microseconds>(end_time -
    start_time).count() << endl;
```

- Known as “instrumenting” your code
- Useful, but has some cost - don’t e.g. try to measure within tight loops
- Google Benchmark builds this into a useful framework to benchmark functions

Emulation

- Callgrind tool (part of Valgrind[2])
- 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

[2]Very useful suite of tools for debugging and profiling

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
- 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
  - -03 = expensive optimisations (take a long time, may actually make code slower) applied
- O2 is a good optimisation reference level - try also O3
- Measure at O2/O3 **before** optimising by hand
- Fine-tuned optimisation options available - check GCC/clang documentation for details

Optimisation example

- GCC and Clang compilers can reduce square example[3] down to something sensible

```
int square(int n)
{
  int k = 0;
  while (true)
  {
    if(k == n*n)
    {
      return k;
    }
    k++;
  }
}
```

→

```
int square2(int n)
{
    return n*n;
}
```

- Optimising compilers are amazing - you only need to care when automatic optimisation fails

[3] NB: Don't write a square function, just square numbers in the code

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

- Addition is faster than multiplication (usually compiler will do this for you if needed)
- Multiplication is faster than division

```
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
```

- Some of these rearrangements lose clarity
- Only do this if it's genuinely a bottleneck

Mathematical functions

- Square root is slow
- Trigonometric functions, exp, log, 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
  }
}
```

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 (but make sure you reserve enough size in advance!)
  - `std::map` and `std::unordered_map` are also useful

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

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 it leaks
  - 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 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

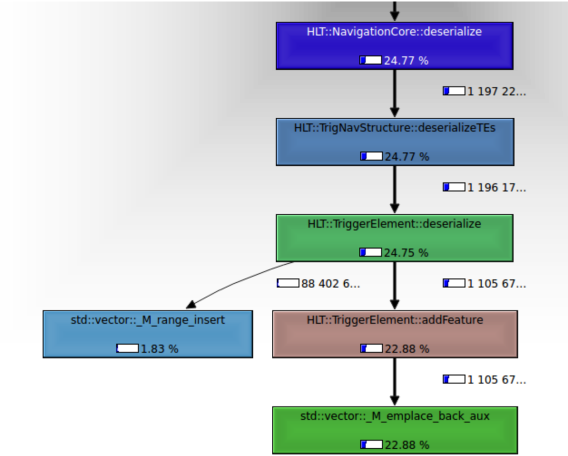
- Scenario: your program is running out of memory
- How to track down large (e.g. 1 GB) allocations?
- `tcmalloc` provides a printout when this happens

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

- Add a breakpoint at `(anonymous namespace)::ReportLargeAlloc(unsigned long, void*)`

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



Memory profiling thoughts

- Memory profiling is more difficult than CPU profiling - tools less advanced/convenient
  - But improving all the time
- Can make a big difference if you're using a lot of memory

Profiling and optimisation 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
- More detail and worked examples in workshop

Conclusions

- A lot to cover in a single (long) lecture
- Continuing developments, particularly in concurrency and memory safe programming
- Good debugging and profiling skills can help you in a lot of areas throughout your PhD
- Particularly for C++, books (e.g. by Herb Sutter, Scott Meyers) and videos (e.g. from CppCon)

Backup: Meltdown and Spectre

- Security vulnerabilities in branch predictors: discovered January 2018
- Allow an attacker to read information from a non-executed branch
- More details here, here and here
- Fixes will slow down certain types of program

MELTDOWN

Backup: vec2.cpp

```
//Clang 6.0 thinks it's not worth it to vectorise, GCC 7.5 thinks it is
//g++ -msse4.2 -std=c++11 vec2.cpp -O3 -fopt-info-vec
//clang++ -msse4.2 -std=c++11 vec2.cpp -O2 -Rpass-missed=loop-vectorize
#include <array>
#include <iostream>

struct particle {
        float x; float y; float z; float t;
};

int main() {

        std::array<particle, 100> part_array;
        for (unsigned int i = 0; i < 100; i++) {
                particle part;
                part.x = i; part.y = i*2; part.z = i*3; part.t = i*4;
                part_array[i] = part;
        }
        std::cout << part_array[10].x << std::endl;
}
```

```
clang++ -msse4.2 -std=c++11 vec2.cpp -O2 -Rpass-missed=loop-
```