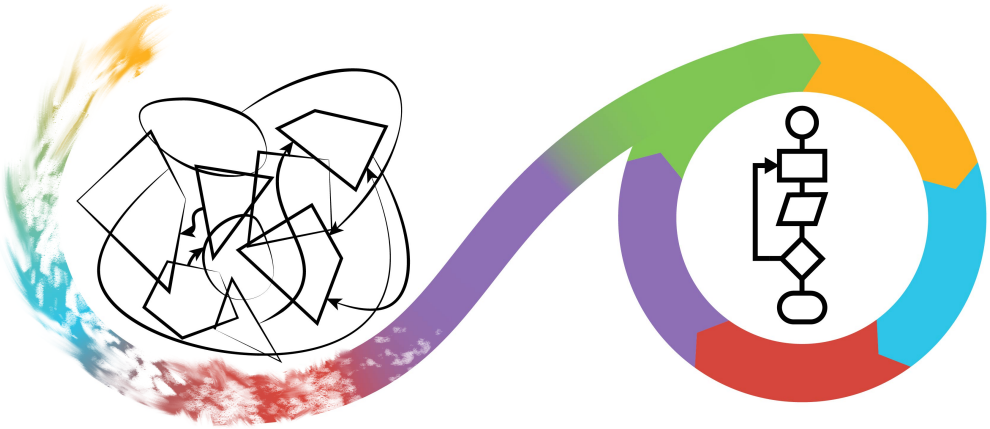
Optimisation

Richèl Bilderbeek

Optimization



Problem

What to do, when your code is too slow?

How often should we try to make our code faster?

When to optimize for speed?

- C++ Core Guidelines: Per.1: Don't optimize without reason
- C++ Core Guidelines: Per.2: Don't optimize prematurely
- C++ Core Guidelines: Per.3: Don't optimize something that's not performance critical

Famous quote

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.
>
> Donald Knuth

Source: Wikipedia

Two measurements

- C++ Core Guidelines: Per.6: Don't make claims about performance without measurements
- Big O complexity
- Run-time speed profile

Big O

How your (combination of) algorithms scales with more complex input.

- Counting the words in a book: O(n)
- Looking up a word in a dictionary: O(log2(n))

Exercise

Measure big-O complexity of https://www.pythonpool.com/check-if-number-is-prime-in-python/

Measure big-O complexity of DNA alignment at https://johnlekberg.com/blog/2020-10-25-seq-align.html

Figure 1: Donald Knuth

Break 1

Run-time speed profile

- 95% of all time, the program spends in 5% of the code
- Even expert programmers are bad at predicting this
- Without measuring where the program spends its cycles, your time is wasted

Misconceptions

- Do not assume you need to learn Assembly/C/C++/Rust
- C++ Core Guidelines: Per.4: Don't assume that complicated code is necessarily faster than simple code
- C++ Core Guidelines: Per.5: Don't assume that low-level code is necessarily faster than high-level code

Exercise

Create speed profile of https://www.pythonpool.com/check-if-number-is-prime-in-python/

Create speed profile of DNA alignment

Break 2

Recap quote

> It is far, far easier to make a correct program fast, than it is to make a fast program correct.
>
> Herb Sutter

Recap

- Tested and clean code always comes first
- Measure where the problem is
- Conclude what to do

Figure 2: Herb Sutter

Source Wikimedia

Ruthless copy-paste of here:

“Premature optimisation is the root of all evil”

Overview of the module:

```
Performance bottlenecks - what might cause slowness?
Profiling - where is slow?
Performance analysis - why is it slow?
Optimisation - how do we make it better?
Parallelisation - a chapter of its own...
```

Ruthless copy-paste of here:

Performance bottlenecks

The rate of execution of given process (i.e. a running program) will typically be limited by one bit of hardware, perhaps two. To increase the performance, you have to know what this bottleneck is.

So let’s look at some typical hardware:

High-level diagram of a compute node

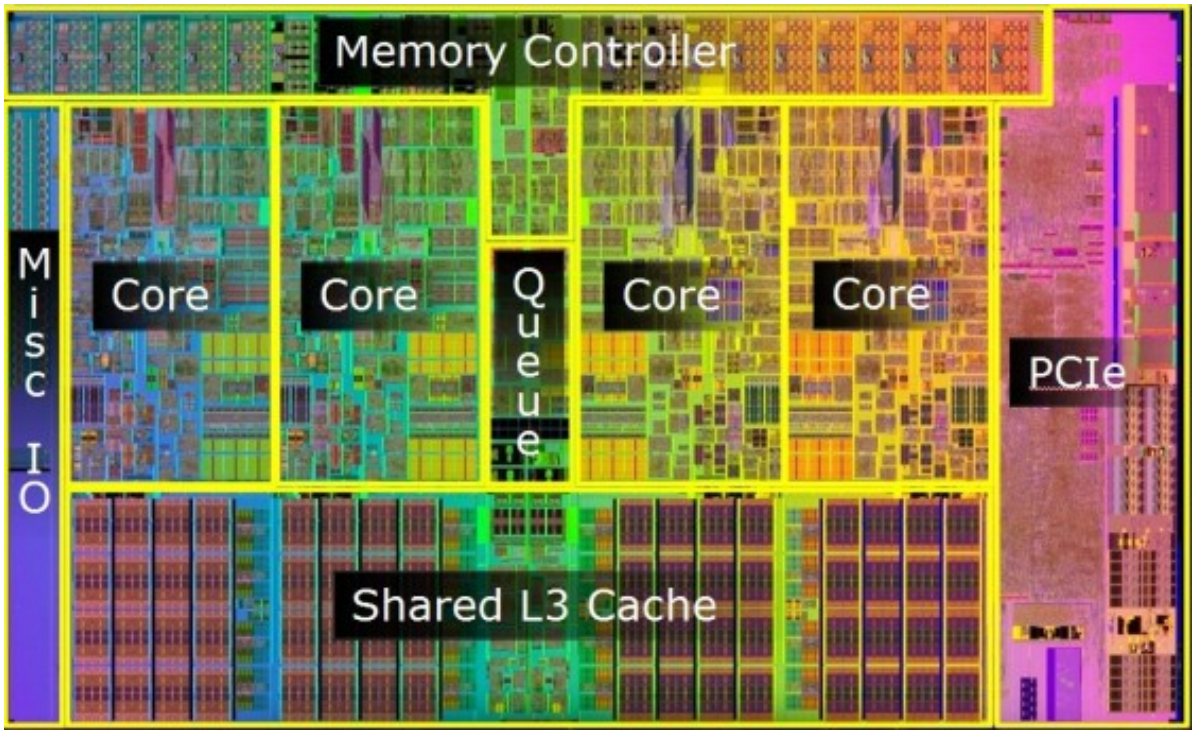
In order to do any work, the CPU cores must execute instructions and move data to/from memory, disk, and network. Disk, network, and other system call bottlenecks

We will only briefly cover the scenario when a program is limited by network and file I/O, because if there is a solution it is usually pretty obvious. Other system calls, like printing to console, are similarly simple. When we get to profiling, we’ll also show a couple of ways of discovering that this is the bottleneck.

A common cause of disk I/O bottlenecks is opening and closing files rapidly. Every time a process needs to ask the operating system to open or close a file, it takes about 0.1 ms. Just writing to the file is 100x faster. We’ll see a demonstration of this later.

An exception to the rule that these bottlenecks are easy to see and resolve can be found in large distributed memory codes, where time spent communicating between nodes can limit the performance. If you find yourself in such a situation (a parallel profiler will tell you this), find an expert to help you and be prepared to spend a few months or years developing a new code… Cache

Before we can discuss the optimisation of the actual instructions that do work, we must examine the CPU a bit closer.



As this diagram of a 4-core Intel i7 shows, a fair amount of CPU space is dedicated to a large cache, which is used to invisibly (to the programmer) preload data from RAM onto the CPU. This image actually understates the space dedicated to cache, because each core has L2 and L1 caches as well! Probably >50% of the chip is occupied by cache.

This should tell you that proper use of cache is important. The details of how caches work in practice varies a lot between chips, but there are general rules that we can follow to improve most codes. In addition, when a byte of data is accessed from RAM, an entire cache line is fetched, usually 64 or 128 bytes.

Note that cache, while present on the chip, is not immediately accessible for operations. The CPU does arithmetic operations on registers, which get loaded with the output of other operations or data from cache/memory. The CPU pipeline

"We have to deeper."

All modern CPU's are pipelined. At the end of the single-core era, some CPU's had 30-stage pipelines, but modern chips have reduced pipelines down to less than 20. We can illustrate this with a generic 4-step "Fetch-decode-execute-write" pipeline:

a generic 4-stage instruction pipeline

Because parts of a CPU are anyway designed to do different things, several instructions can be executed at once by fetching a new instruction (reading the next line of assembly) while the previous instruction is being decoded (translating the instruction's hexadecimal value into

operations). This is known as Instruction-level parallelism (ILP) The advantage of this setup is better utilisation of the CPU's hardware and (in this case) up to four sequential instructions of a single program executing in parallel. In real systems, a CPU core is only running at full performance if 10-20 instructions are in the pipeline at once.

In order to fill the pipeline, the CPU must be able guess at the instructions that will follow e.g. an "if" statement. Additionally, all of the instructions must have their data ready (i.e. in register if doing arithmetic, in L1 cache if loading a register, etc). If there is a cache miss, or the CPU guesses wrong, all the stuff in the pipeline must wait or be purged entirely, reducing the effective ILP. Modern CPUs ameliorate the impossibility of this to some extent by executing some instructions out of order. Vector/SIMD operations

One of the main workhorses of a CPU core is the Arithmetic Logical Units (ALU), which does the math. Modern ALUs are equipped with wide "vector" registers, which perform SIMD parallelism. Typical high-precision floating point numbers use 4 bytes (32 bits). A vector register can store up to 16 doubles, and associated vector instructions can do operations on all the elements simultaneously.

Needless to say, this is super handy if you have an algorithm that does a lot of the same operations on data that lies adjacent. But even if you do, the compiler often needs to be coaxed to issue such instructions, if you don't do it by hand. In either case, you will be subject to the whims of machine spirits — prayers and offerings can go a long way.

Parallelisation

Of course, if you only issue instructions to one CPU core at a time with a single-threaded program, you're potentially missing out on a lot of performance. We'll discuss this more later.

Ruthless copy-paste of here:

Profiling

In many cases, the system will spend a majority of its time in a small amount of code. Clearly, this is the part of the code you need to improve. Amdahl's Law

This law usually comes up in the context of parallelisation, but it is equally applicable to any other optimisation technique. Amdahl's law points out that the overall speedup gained by optimising a single section of a program is limited by the fraction of time that the improved section required.

More specifically, if you are optimising a section of code that takes p % of the total runtime, you can achieve maximum 100/(100-p) times speedup.

Knowing the expected outcome of an optimisation effort is helpful when deciding whether to spend time on it. Time measurement

A simple method for timing is using the UNIX time function.

```
user\@system:\~/mydir\$ time ./myprogram real 0m0.008s user 0m0.000s sys 0m0.002s
```

This is super handy for checking whether a change made a difference, but can also hint at potential bottlenecks.

real is the "wall-time", the actual physical time that the program ran. This is what you want to improve.

user is time spent in "user mode", normal program execution. Multithreaded processes will report the sum of the times of all threads.

sys is time spent in system calls. Programs that spend their time here are usually limited by memory I/O, disk accesses, network, console output, or the like.

To see this in action, create a short python program with the following lines:

```
for i in range(10000): with open("test.txt","w") as f: f.write(str(i)+"\n")
```

Try running `time python3 fiotest.py` and examine the output. Then move the "with open…" line to the top of the script and compare.

The timeit module in Python is useful for timing parts of your code. The equivalent in C is using the <time.h> library. Profilers

Profilers are tools, either internal or external, that can tell you more details about where your program is spending its time.

Try profiling the fiotest.py script:

```
marcusl\$ python3 -m cProfile -s 'tottime' filetest.py 50003 function calls in 1.587 seconds

Ordered by: internal time

ncalls tottime percall cumtime percall filename:lineno(function) 10000 0.957 0.000 0.989 0.0
```

On SNIC systems, you have access to Intel VTune, which can do the same thing for C, Fortran, and other codes. Many IDE's have a profiler, including XCode.

Basically all Linux systems with gcc also have the profiling tool gprof and the memory checker valgrind, which may not be super nice to use but are perfectly serviceable in a pinch.

Valgrind is actually an entire collection of tools. If you want to know something about the behaviour of your program, chances are good that something in valgrind can tell you. For instance, the callgrind tool can give a clear report on cache misses and branch prediction misses on the scale of a function.

Ruthless copy-paste of here:

Performance analysis

Once you know the hotspots in your code, it may not be immediately evident what is making it slow.

Generally speaking, the execution of code is limited by the rate of arithmetic operations or by the rate of data flowing into the CPU. But which is it, and how can we tell? The Roofline model

https://dl.acm.org/doi/10.1145/1498765.1498785

Links to an external site.

https://www.sciencedirect.com/topics/computer-science/roofline-model

Links to an external site.

roofline_model.jpeg

In the Roofline model, the arithmetic intensity of a given code kernel is calculated (or estimated) and plotted on a roofline graph for the test hardware. Focus is put on the kernel of a code, the small contents of the inner-most loop where the CPU is spending most of its time. For performance issues that stem from other sources, the roofline model is not useful. Arithmetic intensity (AI)

AI, sometimes called numeric intensity or operational intensity, is typically given in flops/word or flops/byte. This is simply the number of operations in the ALU divided by the number of words or bytes transferred from memory. A high AI corresponds to a code that does a lot of arithmetic on every data element, which will usually lead it to be limited by the speed of the arithmetic operations and is present somewhere on the flat part of the graph. A low AI means that the ALU is partially idle during execution, and the code is somewhere on the banked part of the roofline.

The operational intensity of a code is a consequence of the algorithm and the CPU's instruction set (i.e. assembly language). Sometimes, poor coding can artificially inflate or deflate it, but (for example) a correctly written kernel that sums two arrays will have an OI of 0.5 flops/word and there's nothing you can do about it.

Often, the algorithm will tell you roughly what the OI of your kernel is, but you can be even more sure by looking at the assembly code and counting instructions. This requires some knowledge of assembly… which isn't actually so bad.

From a naive matrix-matrix multiply (double precision):

[assembly code]

Total (ignoring bookkeeping): 4 ops and 12 bytes, giving us 0.333 ops/byte or 1.33 ops/word.

The same program had an auto-SIMDised version of the function:

[assembly code]

Total (ignoring bookkeeping)l: 8 ops, 48 bytes, giving us 0.333 ops/byte or 1.33 ops/word. Drawing a roofline graph

This can be challenging, especially if you lack certain information about your computer's hardware. On Linux machines, you can use /proc/cpuinfo. On Mac, the command sysctl -n machdep.cpu is helpful. You need information about the clock speed, but also the vector instruction sets (e.g. avx2 among "flags"), then a search on the web can tell you about the supported SIMD instructions.

You also need to find the maximum bandwidth to memory, and this information is not always available to look up. I've found that a simple test in C using memcpy() provides usable info.

Roofline model showing three levels of computational ceiling

In the example above, borrowed from the best Roofline article ("Roofline:An insightful Visual Performance model for multicore Architectures"), you can see an example of computational ceilings for a certain computer. With "just" perfect thread-level parallelism (TLP), a program can attain at most 2 GFlops. With SIMD instructions doing 4 effective operations every instruction, the max performance increases accordingly. The last doubling of speed comes from "balancing" arithmetic operations, allowing the compiler to issue fused multiply-add instructions that effectively do two operations at once. Note that the height of some of these ceilings depends on whether you're doing single-precision or double-precision math.

Roofline mode showing 3 levels of memory bandwidth

This example shows memory bandwidth ceilings. A code that only relies on decent memory performance from accessing memory addresses sequentially, which means entire cache lines are consumed efficiently and cache misses are minimised, will reach the peak floating point capacity only if it requires 4 floating point operations on each byte (16 operations for each double!). By making the code NUMA-aware, ensuring that threads are accessing only the nearest memory, performance can be more than doubled. The last bit of efficiency is gained by introducing a clever prefetching scheme that uses your knowledge of the algorithm to ensure that the right data is always in the right place when needed.

Roofline model showing different optimisation regions

A naive implementation of a code, even if nicely multithreaded, will typically perform somewhere in the light blue region of the above plot.

Ruthless copy-paste of here:

Optimisation

As you've seen, the efficiency of a code can be placed on a very wide spectrum. In this section, we'll try to start from the very worst of situations and discuss how to improve.

At every stage, keep your goals in mind. You're probably looking to solve a particular problem or set of problems — don't spend more time optimising than you need to. Optimising a naive script

"Naive", in this context, means a code that was written off the top of your head to solve the problems in front of you. This code was probably written iteratively, based around the set of commands you issued to a Python or R prompt.

Now you're running it on a bigger problem, or you're going to run it on a thousand data sets, and you realise it's taking forever.

Step 0 really should be to think about your algorithm and what you're really doing. When you started, you maybe did not have a clear understanding of the problem and your approach, but now that the script is written and is working, you should consider things like:

```
whether you're going for an optimal result when an approximation would suffice,
whether you're recomputing results that you could reuse (dynamic programming),
whether you know something about the data that you're not using,
whether you're using the right data structures
```

Assume that you're happy with all this and you simply want to speed up your code, here are some tips:

```
Keep your kernel clean. Move all unnecessary stuff out of the inner working loop of your scri
Reduce disk I/O by avoiding it. You can pipe data directly between programs, avoiding the pe
Improve your disk I/O latency by using local disk instead network. On most clusters, there w
Use high-level constructs instead of for-loops. In Python, for instance, you can use list co
Use the right data type/data structure. There is a huge difference in performance between e.g
Parallellise. More on this later, but if you're not I/O limited (and sometimes even then) you
Convert the kernel to a compiled language such as C. If all else fails, and especially if the
```

Optimising a multithreaded C code

Suppose you have done all this and have a solid implementation in a compiled language such as C, Fortran, Rust, Go, or the like. But you're still not happy with the performance. What now?

Again, revisiting the algorithm side of things is probably wise. Having done that, make a Roofline plot and follow these tips:

```
Fool around with compiler flags. Your compiler can do magic, especially if you specify the ar
Improve disk I/O with memory mapping. Memory mapped files appear to the programmer and progra
Increase ILP with loop unrolling. By making the inside of a loop longer, more gets done befor
SIMDize the code. Check the assembly for SIMD operations. Ask the compiler for an autovectori
this is surprisingly often more effective than autovectorisation.
Improve cache efficiency by accessing data with a regular stride. Avoid random jumps in memor
Improve cache efficiency by reaccessing data more quickly. If you reorder accesses to data so
Implement software prefetching. The prefetcher will retrieve a cache line when you access a l
```

Ruthless copy-paste of here:

Parallelisation

Multithreading can be done in multiple ways, even using a single programming language.

The easiest and best solution in many cases is to use a library that does what you need and is already parallelised. Then someone else has done the hard work and hopefully has done it well. With that said: Measure the scaling/speedup of your parallel program

Don't trust people's words about what is fast and what is parallel. They're probably lying or mistaken or talking about a different problem size or assumed different settings. You must time your workload.

```
Get a baseline timing with a single thread on a representative input set. Ideally, use a good
Run with 2, 4, 8, etc threads using the same input. This gives you the strong scaling, the sp
Run with 2, 4, 8, etc threads, also doubling the size of the input. This shows you the weak s
```

A Parallel boid simulation

Here I try to present an instructive example of how a relatively simple program turned out to be difficult to parallelise.

Boids are self-propelled particles in a 2D or 3D space. They move about in the space and change direction when they interact with nearby boids.

A boid simulation looks like this:

boid algorithm

Finding the neighbours naively involves checking whether pair of boids is close, which is O(N2). Since boids only interact with nearby boids, a spatial subdivision scheme that uses a hierarchy to discard entire groups of boids from consideration at once seems like a better algorithm. In 2D, a quadtree reduces the complexity to O(N log N).

Such a tree can be built with

quadtree algorithm

Then, finding the nearest neighbour looks like this:

nearest neighbours search algorithm

The fast boid algorithm is then:

fast boid algorithm

Profiling a C implementation of this shows that 96% of the runtime is spent in the force interaction loop between lines 7-11. Amdahl's Law suggests that we should be able to achieve a maximum speedup of 25x with infinite parallelism, and about 6.25x speedup on 8 threads or 10x speedup on 16 threads. Moreover, this loop is embarrassingly parallel.

This was quickly done, and we measured the speedup with glee.

Of course, this wouldn't be a lesson in this course if it turned out as we'd hoped…

plot of speedup of parallel fast boids, showing poor results.

What happened??

More profiling showed that multithreading the interaction loop slowed down the tree building step by 65%. Additionally, the multithreaded code segment itself had 4% inefficiency. Some of this time is overhead from starting and waiting for threads. The bottom line

Test, test, and test your assumptions about what makes things quicker (or not).