Writing Efficient Code in C(++)

Petr Ročkai

Organisation

- theory: 20-30 minutes every week
- coding: all the remaining time
- passing the subject: collect 7 points
- most points come from assignments
- showing up 10 times gets you 1 point

Assignments

- one assignment every 2 weeks, 5 in total
- missing the deadline or failing is the same

Deadlines

1. 14 days (Wed by midnight), fetches 2 points
2. end of semester (17.12.), fetches 1.5 points
3. end of the exam period (12.2.), fetches 1 point

Assignments (cont'd)

- you can use git, mercurial or darcs
- put everything that you want me to see on `master`
- write a simple `Makefile` (no cmake, autotools, …)
- each homework gets a target (`make hw1` through `hw5`)
- use the same repo for in-seminar work (`make ex1` …)

Competitions

- we will hold 3 competitions in the seminar
- you'll have 40 minutes to do your best on a small problem
- the winner gets 1 point, second place gets .5 point
- all other working programs get .2 points
- we'll dissect the winning program together

Preliminary Plan

19.9.	today computational complexity	
26.9.	microbenchmarking & statistics	
3.10.	cancelled	
10.10.	the memory hierarchy	hw01 due
17.10.	using `callgrind`	
24.10.	tuning for the compiler/optimiser	hw02 due
31.10.	competition 1	
7.11.	understanding the CPU	hw03 due
14.11.	exploiting parallelism	
21.11.	using `perf` + competition 2	hw04 due
28.12.	Q&A, homework recap	
5.12.	semester recap + competition 3	hw05 due

Efficient Code

- computational complexity
- the memory hierarchy
- tuning for the compiler & optimiser
- understanding the CPU

- exploiting parallelism

Understanding Performance

- writing and evaluating benchmarks
- profiling with `callgrind`
- profiling with `perf`
- the law of diminishing returns

- premature optimisation is the root of all evil
- (but when is the right time?)

Tools

- on a POSIX operating system (preferably not in a VM)
- `perf` (Linux-only, sorry)
- `callgrind` (part of the `valgrind` suite)
- `kcachegrind` (for visualisation of `callgrind` logs)
- maybe `gnuplot` for plotting performance data

Compilers

- please stick to C++14 and C11 (or C99)
- the reference compiler will be clang 5.0.1
- you can use other compilers locally
- but your code has to build with clang 5

Part 1: Computational Complexity

Complexity and Efficiency

- this class is not about asymptotic behaviour
- you need to understand complexity to write good code
- performance and security implications

- what is your expected input size?
- complexity vs constants vs memory use

Quiz

- what's the worst-case complexity of:
  - a bubble sort? (standard) quick sort?
  - inserting an element into a RB tree?
  - inserting an element into a hash table?
  - inserting an element into a sorted vector?
  - inserting an element into a dynamic array?
- what are the amortised complexities?
- how about expected (average)?

Hash Tables

- often the most efficient data structure available
- poor theoretical worst-case complexity
  - what if the hash function is really bad?
- needs a fast hash function for efficiency
  - rules out secure (cryptographic) hashes

Worst-Case Complexity Matters

- CVE-2011-4815, 4838, 4885, 2012-0880, …
- apps can become unusable with too many items
- use a better algorithm if you can (or must)
- but: simplicity of code is worth a lot, too
- also take memory complexity and constants into account

Constants Matter

- n ops if each takes 1 second
- $n \log n$ ops if each takes .1 second
- n^2 ops if each takes .01 second

Picking the Right Approach

- where are the crossover points?
- what is my typical input size?
- is it worth picking an approach dynamically?
- what happens in pathological cases?

Exercise 1

- set up your repository and a `Makefile`
- implement a bounded priority buffer
  - holds at most n items
  - holds at most one copy of a given item
  - forgets the smallest item if full
  - fetch/remove the largest item
  - API: `insert`, `top` and `remove`
- two versions: sorted array and a sorted list

Exercise 1 (cont'd)

- write a few unit tests
- write a benchmark that inserts ($\sim 10^7$) random values
- the benchmark can use `clock(3)` or `time(1)`
- compare the approaches for $n = 5, 10, 10000$

- what are the theoretical complexities?
- what are your expectations on performance?
- can you think of a better overall solution?

Part 2: Microbenchmarking & Statistics

Motivation

- there's a gap between high-level code and execution
- the gap has widened over time
  - higher-level languages & more abstraction
  - more powerful optimisation procedures
  - more complex machinery inside the CPU
  - complicated cache effects
- it is very hard to predict actual performance

Challenges

- performance is very deterministic in theory
- this is not the case in practice
  - time-sharing operating systems
  - cache content and/or swapping
  - power management, CPU frequency scaling
  - program nondeterminism; virtual machines
- both micro (unit) and system benchmarks are affected

Unit vs System Benchmarking

- a benchmark only gives you one number
- it is hard to find causes of poor performance
- unit benchmarks are like unit tests
  - easier to tie causes to effects
  - faster to run (minutes or hours vs hours or days)
  - easier to make parametric

Isolation vs Statistics

- there are many sources of measurement errors
- some are systematic, others are random (noise)
- noise is best fought with statistics
- but statistics can’t fix systematic errors
- benchmark data is not normally distributed

Repeated Measurements

- you will need to do repeat measurements
- more repeats give you better precision
  - the noise will average out
  - execution time vs precision tradeoff
- the repeat runs form your input sample
  - this is what you feed into bootstrap

Bootstrap

- usual statistical tools are distribution-dependent
- benchmark data is distributed rather oddly
- idea: take many random re-samplings of the data
- take the 5th and 95th percentiles as a confidence interval
- this is a very robust (if stochastic) approach

Implementing Bootstrap

- inputs: a sample, an estimator and iteration count
- outputs: a new sample
- in each iteration, create a random resample
  - add a random item from the original sample
  - we do not care about repeats
  - size of the resample should be the same as the original

Estimators

- most useful estimators are the mean (average)
- and various percentiles (e.g. median)
- you can also estimate standard deviation
  - but keep in mind the original data is not normal

Output Distribution

- the output of bootstrap is another distribution
- you can expect this one to be normal
- it is the distribution of the estimator result
- you can compute the mean and σ of the bootstrap

Confidence Intervals

- assume your estimator is the mean (average)
- you get a normal distribution of averages
  - each of them is more or less likely correct
  - you can pick the average one as your estimate
  - and take a 2σ interval for the CI

Confidence Interval on Performance

- the above gives you a CI on average speed
- you may want a confidence interval on actual speed
- you can use a 5th or 95th percentile as estimator

Precise Clocks

- available in POSIX via `clock_gettime`
  - the resulting time is in nanoseconds
- your best bet is `CLOCK_MONOTONIC` (maybe `_RAW`)
- you can ask `clock_getres` for clock resolution

Homework 1: Benchmarking

- implement a simple benchmarking tool
- allow for repeat measurements
  - make the time limit and precision configurable
- you will use this tool for the rest of the semester
- the API is up to you

Reducing Systematic Errors

- you can use `fork()` to get fresh processes
  - the testcase might leak memory
  - other effects may cause systematic slowdowns
- consider the effect of cache content
  - hot vs cold cache benchmarking

Output

- for each unit benchmark, print a single line of output
- it should contain an average & a CI on the average
  - also a 90% CI on actual runtime
- also allow each measurement to be printed out separately
- exact format will be decided in the seminar

Part 3: The Memory Hierarchy

- many levels of ever bigger, ever slower memories
- CPU registers: very few, very fast (no latency)
- L1 cache: small (100s of KiB), plenty fast (~4 cycles)
- L2 cache: still small, medium fast (~12 cycles)
- L3 cache: ~2-32 MiB, slow-ish (~36 cycles)
- L4 cache: (only some CPUs) ~100 MiB (~90 cycles)
- DRAM: many gigabytes, pretty slow (~200 cycles)

- NVMe: ~10k cycles
- SSD: ~20k cycles
- spinning rust: ~30M cycles
- RTT to US: ~450M cycles

Paging vs Caches

- page tables live in slow RAM
- address translations are very frequent
- and extremely timing-sensitive
- TLB → small, very fast address translation cache

- process switch → TLB flush
- but: Tagged TLB, software-managed TLB
- typical size: 12 - 4k entries
- miss penalties up to 100 cycles

Additional Effects

- some caches are shared, some are core-private
- out of order execution to avoid waits
- automatic or manual (compiler-assisted) prefetch
- speculative memory access
- ties in with branch prediction

Some Tips

- use compact data structures (vector > list)
- think about locality of reference
- think about the size of your working set
- code size, not just speed, also matters

See Also

- `cpumemory.pdf` in study materials
  - somewhat advanced and somewhat long
  - also very useful (the title is not wrong)
  - don't forget to add 10 years
  - oprofile is now `perf`
- `http://www.7-cpu.com` CPU latency data

Exercise 2

- write benchmarks that measure cache effects

Some Ideas

- walk a random section of a long `std::list`
- measure time per item in relation to list size
- same but with a `std::vector`
- same but access randomly chosen elements (vector only)

Some Issues

- `uniform_int_distribution` has odd timing behaviour
- but we don't really care about uniformity

- you may need to fight the optimiser a bit
- especially make sure to avoid undefined behaviour

- indexing vs iteration have wildly different behaviour
- shuffling your code slightly can affect the results a lot

Homework 2: Matrix Multiplication

- implement a real-valued matrix data structure
- implement 2 matrix multiplication algorithms
  - natural order
  - cache-efficient order
- compare the implementations using benchmarks

Part 4: Profiling I, `callgrind`

Why profiling?

- it's not always obvious what is the bottleneck
- benchmarks don't work so well with complex systems
- performance is not quite composable
- the equivalent of `printf` debugging isn't too nice

Workflow

1. use a profiler to identify expensive code
   - the more time program spent doing X,
   - the more sense it makes to optimise X
2. improve the affected section of code
   - re-run the profiler, compare the two profiles
   - if satisfied with the improvement, goto 1
   - else goto 2

What to Optimise

- imagine the program spends 50 % time doing X
  - optimise X to run in half the time
  - the overall runtime is reduced by 25 %
  - good return on investment
- law of diminishing returns
  - now only 33 % of time is spent on X
  - cutting X in half again only gives 17 % of total
  - and so on, until it makes no sense to optimise X

Flat vs Structured Profiles

- flat profiles are easier to obtain
- but also harder to use
  - just a list of functions and cost
  - the context & structure is missing
- call stack data is a lot harder to obtain
  - endows the profile with very rich structure
  - reflects the actual control flow

`cachegrind`

- part of the `valgrind` tool suite
- dynamic translation and instrumentation
- based on simulating CPU timings
  - instruction fetch and decode
  - somewhat abstract cost model
- can optionally simulate caches
- originally only flat profiles

`callgrind`

- records entire call stacks
- can reconstruct call graphs
- very useful for analysis of complex programs

`kcachegrind`

- graphical browser for callgrind data
- demo

Exercise 3

- there's a simple BFS implementation in study materials
- you can also use/compare your own BFS implementation
- don't forget to use `-O2 -g` or such when compiling
- generate a profile with `cachegrind`
- load it up into `kcachegrind`
- generate another, using `callgrind` this time & compare

Exercise 3 (cont'd)

- add cache simulation options &c.
- explore the knobs in `kcachegrind`
- experiment with the size of the generated graph
- optimise the BFS implementation based on profile data

Part 5: Tuning for the Compiler

Goals

- write high-level code
- with good performance

What We Need to Know

- which costs are easily eliminated by the compiler?
- how to make best use of the optimiser (with minimal cost)?

How Compilers Work

- read and process the source text
- generate low-level intermediate representation
- run IR-level optimisation passes
- generate native code for a given target

Intermediate Representation

- for C++ compilers typically a (partial) SSA
- reflects CPU design / instruction sets
- symbolic addresses (like assembly)
- explicit control and data flow

IR-Level Optimiser

- common sub-expression elimination
- loop-invariant code motion
- loop strength reduction
- loop unswitching
- sparse conditional constant propagation
- (regular) constant propagation
- dead code elimination

Common Sub-expression Elimination

- identify redundant (& side-effect free) computation
- compute the result only once & re-use the value
- not as powerful as equational reasoning

Loop-Invariant Code Motion

- identify code that is independent of the loop variable
- and also free of side effects
- hoist the code out of the loop
- basically a loop-enabled variant of CSE

The Cost of Calls

- prevents CSE (due to possible side effects)
- prevents all kinds of constant propagation

Inlining

- removes the cost of calls
- improves all intra-procedural analyses
- inflates code size
- only possible if the IR-level definition is available

See also: link-time optimisation

The Cost of Abstraction: Encapsulation

- API or ABI level?
- API: cost quickly eliminated by the inliner
- ABI: not even LTO can fix this
- ABI-compatible setter is a call instead of a single store

The Cost of Abstraction: Late Dispatch

- used for `virtual` methods in C++
- indirect calls (through a vtable)
- also applies to C-based approaches (`gobject`)
- prevents (naive) inlining
- compilers (try to) devirtualise calls

Exercise 4: Variant 1

- start with `bfs.cpp` from study materials
- make a version where `edges()` is in a separate C++ file
- you will need to use `std::function`
- try a compromise using a visitor pattern
- compare all three approaches using benchmarks

Exercise 4: Variant 2

- compare the cost of a direct and indirect call
- write a `foreach` function that takes a function pointer
  - use separate compilation to prevent inlining
- compare to a loop with a direct call
  - the function to be called should be simple-ish

Homework 3: Sets of Integers

- implement a set of `uint16_t` using a bitvector
  - with insert, erase, union and intersection
- the same using a nibble-trie
  - a trie with out-degree 16 (4 bits)
  - should have a maximum depth of 4
  - implement insert and union
- compare the two implementations

Part 6: Understanding the CPU

The Simplest CPU

- in-order, one instruction per cycle
- sources of inefficiency
  - most circuitry is idle most of the time
  - not very good use of silicon
- but it is reasonably simple

Design Motivation

- silicon (die) area is expensive
- switching speed is limited
- heat dissipation is limited
- transistors cannot be arbitrarily shrunk
- “wires” are not free either

The Classic RISC Pipeline

- fetch – get instruction from memory
- decode – figure out what to do
- execute – do the thing
- memory – read/write to memory
- write back – store results in the register file

Instruction Fetch

- pull the instruction from cache, into the CPU
- the address of the instruction is stored in PC
- traditionally does branch "prediction"
  - in simple RISC CPUs always predicts not taken
  - this is typically not a very good prediction
  - loops usually favour taken heavily

Instruction Decode

- not much actual decoding in RISC ISAs
- but it does register reads
- and also branch resolution
  - might need a big comparator circuit
  - depending on ISA (what conditional branches exist)
  - updates the PC

Execute

- this is basically the ALU
  - ALU = arithmetic and logic unit
- computes bitwise and shift/rotate operations
- integer addition and subtraction
- integer multiplication and division (multi-cycle)

Memory

- dedicated memory instructions in RISC
  - load and store
  - pass through execute without effect
- can take a few cycles
- moves values between memory and registers

Write Back

- write data back into registers
- so that later instructions can use the results

Pipeline Problems

- data hazards (result required before written)
- control hazards (branch misprediction)
- different approaches possible
  - pipeline stalls (bubbles)
  - delayed branching
- structural hazards
  - multiple instructions try to use a single block
  - only relevant on more complex architectures

Superscalar Architectures

- more parallelism than a scalar pipeline
- can retire more than one instruction per cycle
- extracted from sequential instruction stream
- dynamically established data dependencies
- some units are replicated (e.g. 2 ALUs)

Out-of-order execution

- tries to fill in pipeline stalls/bubbles
- same principle as super-scalar execution
  - extracts dependencies during execution
  - execute if all data ready
  - even if not next in the program

Speculative Execution

- sometimes it's not yet clear what comes next
- let's decode, compute etc. something anyway
- fills in more bubbles in the pipeline
- but not always with actual useful work
- depends on the performance of branch prediction

Take-Away

- the CPU is very good at utilising circuitry
- it is somewhat hard to write “locally” inefficient code
- you should probably concentrate on non-local effects
  - non-local with respect to instruction stream
  - like locality of reference
  - and organisation of data in memory in general
  - also higher-level algorithm structure

Exercise 6

- implement a brainfuck interpreter
- try to make it as fast as possible
- see wikipedia for some example programs

Homework 4

- implement sub-string search algorithms
- a naive one (with full restarts)
- one based on a failure table (KMP)
- one that uses a DFA
- write benchmarks, find cross-over points

Part 7: Exploiting Parallelism

Hardware vs Software

- hardware is naturally parallel
- software is naturally sequential
- something has to give
  - depends on the throughput you need
  - eventually, your software needs to go parallel

Algorithms

- some algorithms are inherently sequential
  - typically for P-complete problems
  - for instance DFS post-order
- which algorithm do you really need though?
  - topological sort is much easier than post-order
- some tasks are trivially concurrent
  - think map-reduce

Task Granularity

- how big are the tasks you can run in parallel?
  - big tasks = little task-switching overhead
  - small tasks = easier to balance out
- how much data do they need to share?
  - shared memory vs message passing

Distributed Memory

- comparatively big sub-tasks
- not much data structure sharing (small results)
- scales extremely well (millions of cores)

Shared Memory

- small, tightly intertwined tasks
- sharing a lot of data
- scales quite poorly (hundreds of cores)

Caches vs Parallelism

- different CPUs are connected to different caches
- caches are normally transparent to the program
- what if multiple CPUs hold the same value in cache
  - they could see different versions at the same time
  - need cache coherence protocols

Cache Coherence

- many different protocols exist
- a common one is MESI (4 cache line states)
  - modified, exclusive, shared, invalid
  - snoops on the bus to keep up to date
- cheap until two cores hit the same cache line
  - required for communication
  - also happens accidentally

Locality of Reference

- comes with a twist in shared memory
- compact data is still good, but
  - different cores may use different pieces of data
  - if they are too close, this becomes costly
  - also known as false sharing

Distribution of Work

- want to communicate as little as possible
- also want to distribute work evenly
- randomised, spread-out data often works well
  - think hash tables
- structures with a single active point are bad
  - think stacks, queues, counters &c.

Shared-Memory Parallelism in C++

- `std::thread` – create threads
- `std::future` – delayed (concurrent) values
- `std::atomic` – atomic (thread-safe) values
- `std::mutex` and `std::lock_guard`

Exercise 7

- implement shared-memory map-reduce
- make the number of threads a runtime parameter
- check how this scales (wall time vs number of cores)
- use this for summing up a (big) array of numbers
- can you improve on this by hand-rolling the summing loop?

Homework 5

- implement parallel matrix multiplication
- compare to your sequential versions
  - try with 2 and 4 threads in your benchmarks
- you can use `std::thread` or OpenMP