ARCHER Single Node Optimisation

Introduction to performance optimisation

Slides contributed by Cray and EPCC

Overview

- Why do we optimise
- What is performance optimisation
- When do we optimise
- The optimisation cycle
- Trends in computer architecture
- Types of optimisation

Why?

- Large computer simulations are becoming common in many scientific disciplines.
- These often take a significant amount of time to run.
  - Sometimes they take *too long*.
- There are three things that can be done
  - Change the science (compromise the research)
  - Change the computer (spend more money)
  - Change the program (this is performance optimisation)

What?

- There are usually many different ways you can write a program and still obtain the correct results.
- Some run faster than others.
  - Interactions with the computer hardware.
  - Interactions with other software.
- **Performance optimisation is the process of making an existing working computer program run faster in a particular Hardware and Software environment.**
  - Converting a sequential program to run in parallel is an example of optimisation under this definition!

When?

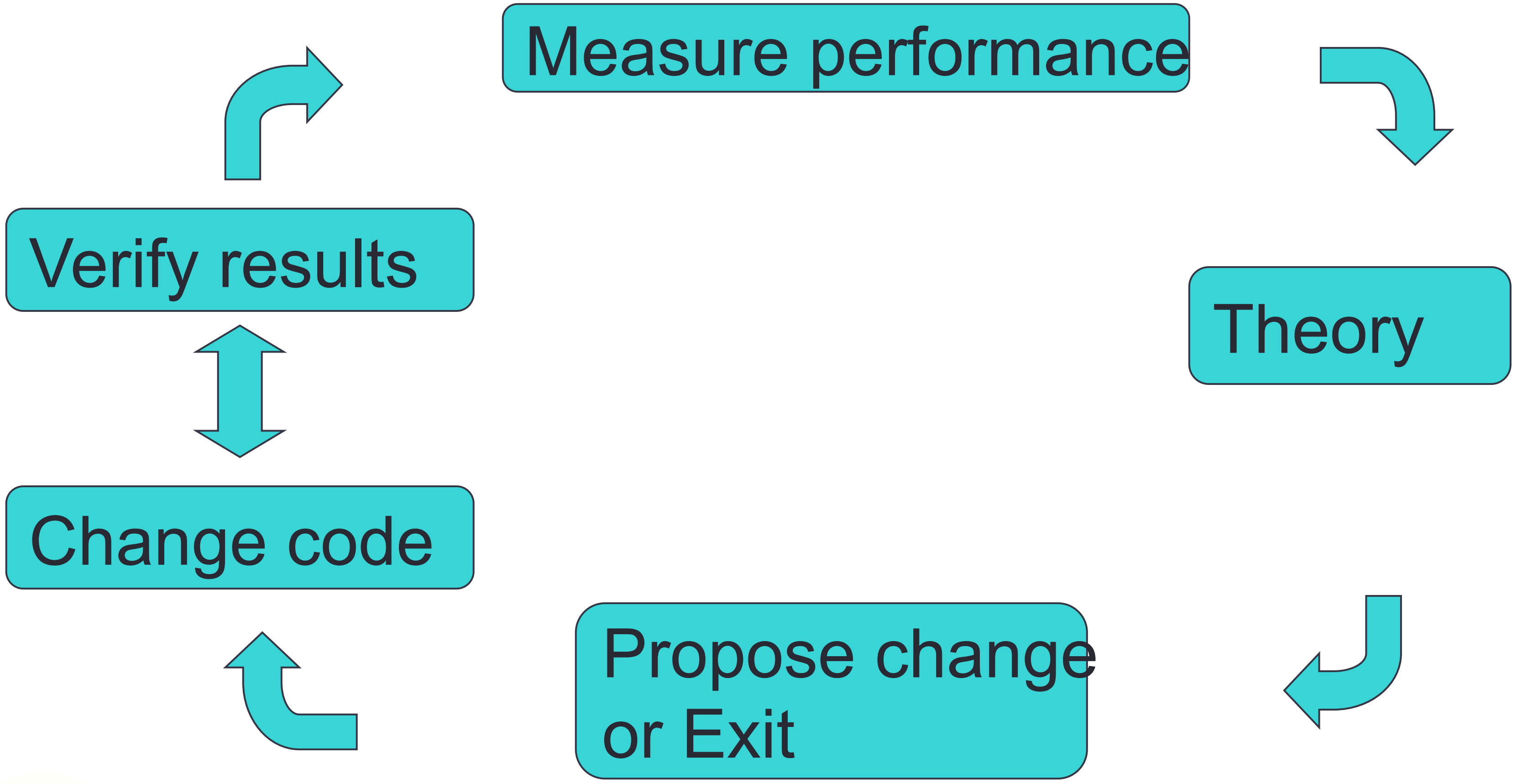
- Performance optimisation can take large amounts of development time.
- Some optimisations improve program speed at the cost of making the program harder to understand (increasing the cost of future changes)
- Some optimisations improve program speed at the cost of making the program more specialised and less general purpose.
- It is always important to evaluate the relative costs and benefits when optimising a program
  - This requires the ability to estimate potential gains in advance

Example:

- Lattice QCD
  - Simulation of the underlying theory of particle physics.
- Requires ***very*** large amounts of computer time.
  - A single simulation can take months of supercomputer time.
- Over 90% of the runtime is in a very small number of kernel routines
  - Almost any amount of optimisation effort expended on the kernel routines will be justified.

How?

- Performance optimisation usually follows a cycle:



Measuring performance

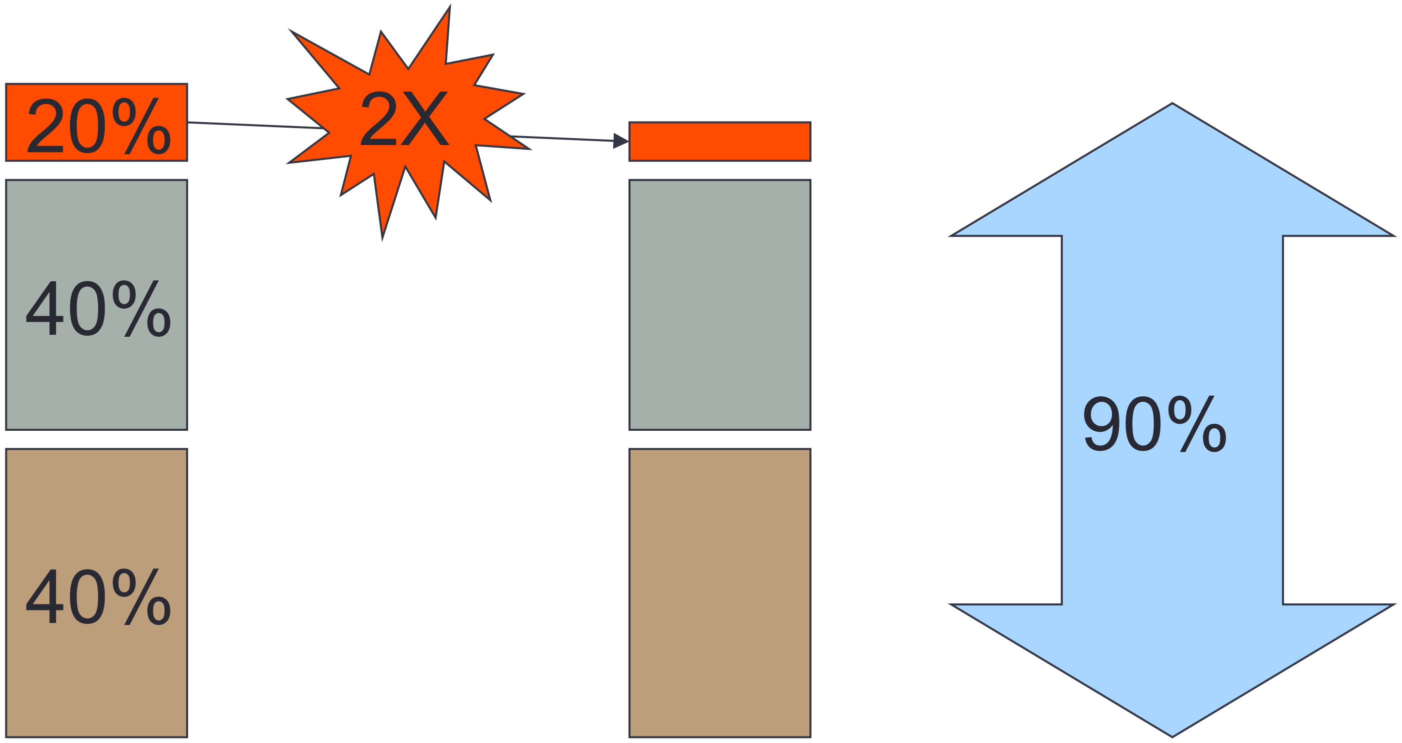
- It is not enough to just measure overall speed
  - You need to know where the time is going.
- There are tools to help you do this
  - They are called profiling tools.
- They give information about:
  - Which sections of code are taking the time
    - Sometimes line by line but usually only subroutines.
  - Sometimes the type of operation
    - memory access
    - floating point calculations
    - file access
- Make sure you understand how variable your results are
  - Are the results down to my changes or just random variation?

Input dependence

- Many codes perform differently with different input data.
- Use multiple sets of input data when measuring performance.
- Make sure these are representative of the problems where you want the code to run quickly.

Only optimise important sections

- Its only worth working on parts of the code that take a lot of time.
- Large speed-up of unimportant sections have little impact on the overall picture.
  - Amdahl’s law is this concept applied to parallel processing.
  - Same insight applies to other forms of optimisation.


20%
2X
40%
40%
90%

Theory

- Optimisation is an experimental process.
- You propose reasons why a code section is slow.
- Make corresponding changes.
- The results may surprise you
  - Need to revise the theory
- Never “optimise” without measuring the impact.

Exit ?

- It is important to know when to stop.
- Each time you propose a code change consider:
  - The likely improvement
    - Code profile and Amdahl`s law helps here.
    - Take account of how long much use you expect for the optimised code. Single use programs are rarely worth optimising.
  - The likely cost
    - Programming/debugging time.
    - Delay to starting simulation
    - “Damage” to the program

Changing code

- Many proposed changes will turn out not be useful.
- You may have to undo your changes.
  - At the very least keep old versions
  - Better to use revision control software.
- Always check the results are still correct !!
  - No point measuring performance if the results are wrong
  - A good test framework will help a lot

Damaging code

- Performance changes can damage other desirable aspects of the code.
  - Loss of encapsulation.
  - Loss of clarity
  - Loss of flexibility
- Think about down-side of changes.
- Look for alternative changes with same performance benefit but less damage.

Back-tracking

- Just because a code change made the code faster does not mean you have to keep it.
- Some performance problems can be addressed in many different ways.
  - Cache conflicts can be addressed by array padding or loop re-ordering.
  - Loop re-ordering is often the cleaner solution but will give little benefit once the arrays have been padded.
  - Be prepared to back-track and apply optimisations to earlier versions.
- If you find a code change with good speed-up
  - Ask how you can persuade the compiler to make an equivalent change.

Key points

- Optimisation tunes a code for a particular environment
  - Not all optimisations are portable.
- Optimisation is an experimental process.
- Need to think about cost/benefit of any change.
- Always verify the results are correct.

Experimental frameworks

- Like any experiment, you need to keep good records.
- You will be generating large numbers of different versions of the code.
  - You need to know exactly what the different version were.
  - How you compiled them.
  - Did they get the correct answer.
  - How did they perform.
- You may need to be able to re-run or reproduce your experiments
  - You discover a bug
  - A new compiler is released
  - A new hardware environment becomes available.
  - Etc.

Making things easier

- Keep everything under version control (including results)
- Script your tests so they are easy to run and give a clear yes/no answer.
- Write timing data into separate log-files in easily machine readable format.
- Keep notes.

Types of optimisation

- Compiler Optimisation
- Auto Tuning
- Hand Optimisation

Compiler Optimisation

- Automatic Optimisation performed by the compiler.
- Compiler takes the source code you give it and tries to find best machine code implementation for the target hardware.
  - Compiler only has limited information about the target hardware
  - Constrained by the source code you give it. Can only make changes allowed within the language specification.
  - Compiler is not intelligent but is much better at some kinds of optimisation than a human programmer.
  - Compiler can afford to optimise the entire program.

Auto tuning

- Experiment driven automatic optimisation.
- A wide range of possible implementations are run on the target hardware to determine the fastest.
  - Different implementation parameters
  - Alternative algorithms.
- May be pure run-time parameters or may generate new source code for each experiment.
- Can only look at the options coded into the auto-tuning.
- Usually specific to one application or library.
- Augments compiler based optimisation.

Hand optimisation

- Changing the source code to improve performance.
- Used to augment compiler/auto-tuning, not as a substitute.
  - Compiler and auto-tuning code is part of the environment you optimise for.

Architecture trends

- Optimisation is the process of tuning a code to run faster in a particular Hardware and Software environment.
- The hardware environment consists of many different resources
  - FPU
  - Cache
  - Memory
  - I/O
- Any of these resources could be the limiting factor for code performance
  - Which one depends on the application

CPU resources

- In the early days of computing memory accesses were essentially free.
  - Optimisation consisted of reducing the instruction count.
- This is no longer the case, and is getting worse
  - CPU performance increases at approx. 80% per year (though recently this has been due to increasing core count rather than clock speed)
  - memory speed increases at approx. 7% per year
- Most HPC codes/systems are memory bandwidth limited.

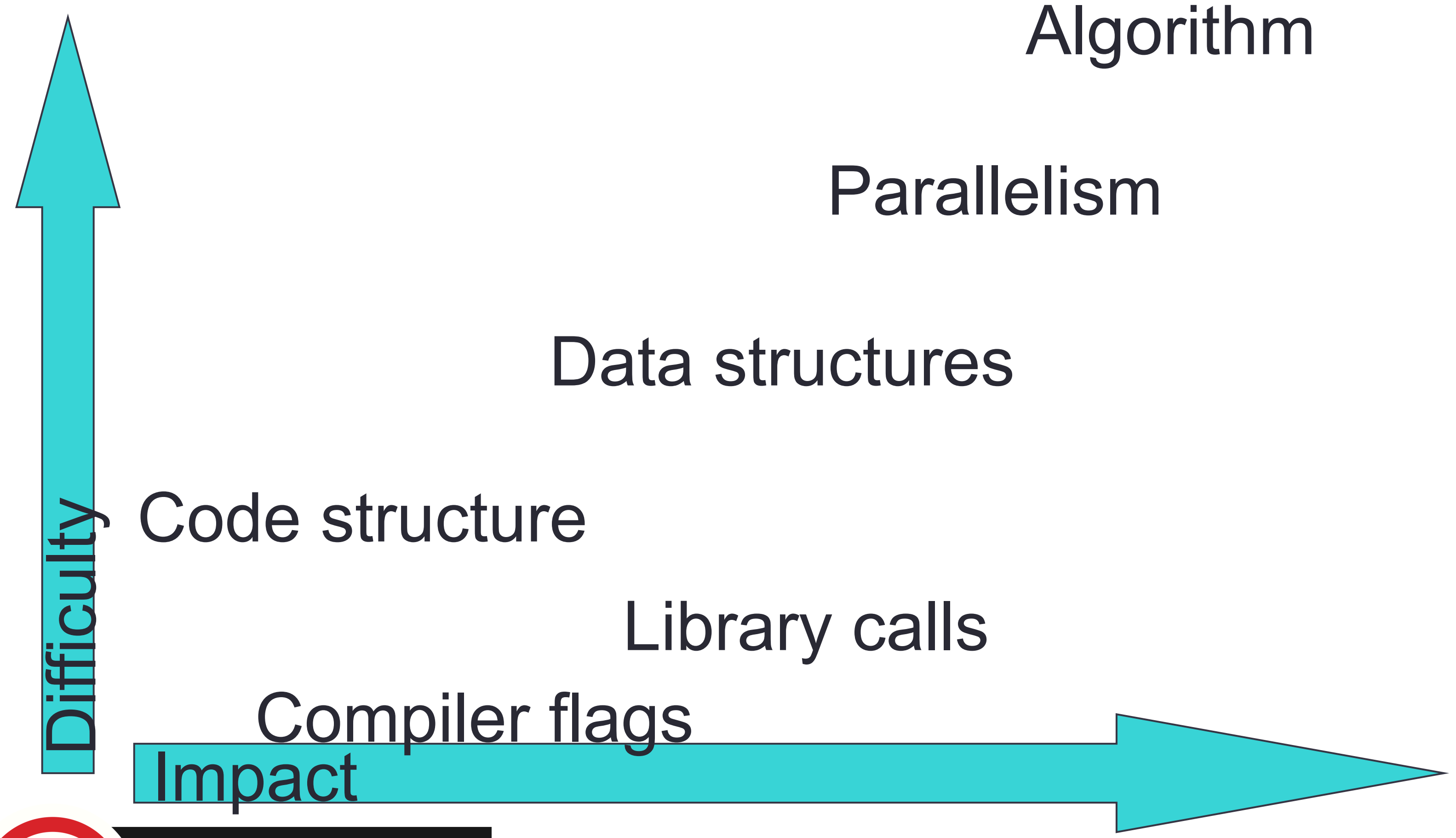
Commercial factors

- Computer design is all about balancing performance and cost.
  - We know how to build faster computers than we can afford.
- Cheaper technologies used where “good enough”
- HPC computing is a very small market compared to desktop/business systems.
  - HPC systems are often built out of components designed for a different purpose.
  - Economies of scale make mainstream technology cheaper

Manufacturing trade-off example

- Processors need complex manufacturing processes
  - E.g. many metal layers.
  - Expensive to manufacture
- DRAM can be manufactured on a much simpler process.
  - Fewer metal layers
  - Much less expensive
- DRAM and processors are manufactured as separate components for cost reasons.
  - Simple DRAM process force interface to be simple too.
  - Separation of RAM and processor impacts performance

Types of optimisation



Compiler flags

- Easiest thing to change are the compiler flags
- Most compilers will have good optimisation by default.
  - Some compiler optimisations are not always beneficial and need to be requested explicitly.
  - Some need additional guidance from user (e.g. inter-file in-lining)
  - Some break language standards and need to be requested explicitly
    - E.g. a/2 -> a*0.5 is contrary to Fortran standard but is usually safe.
    - Usually worthwhile to read compiler manual pages before optimising.

Debugging flags

- Most compilers have flags to enable debugging
- Traditionally debugging flags disable some optimisations
  - this is to make it easier to associate machine instructions with lines of source code.
- Many compilers now support optimisation and debugging together
  - May have alternative flags to allow some debugging with optimisation enabled.

Library calls

- The easiest way to make a big impact on code performance is to re-use existing optimised code.
- Libraries represent large amount of development effort
  - Somebody else has put in the effort so you don’t have to,
- Code libraries exist for commonly used functionality (e.g. linear algebra, FFTs etc.).
  - Often possible to access highly optimised versions of these libraries.
  - Even if the application code does not use a standard library it is often easy to re-write to use the standard calls.

Using libraries

- Don’t just use libraries blindly.
- Read the documentation.
  - Learn what the library is capable of.
  - Libraries often have multiple ways of doing the same thing. You need to understand which is best for your code.
- Many modern libraries support auto-tuning.
  - Can run internal experiments to automatically optimise the implementation for a given problem.
  - Application programmer needs to cooperate on this.

Algorithm

- The biggest performance increases typically require a change to the underlying algorithm.
  - Consider changing an O(N) sort algorithm to a O(log(N)) algorithm.
  - This is a lot of work as the relevant code section usually needs a complete re-write.
- A warning
  - The complexity of an algorithm O(N), O(log(N)), O(N log(N)) etc. is related to number of operations and is not always a reliable indication of performance.
    - Pre-factor may make a "worse" algorithm perform better for the value of N of interest.
    - The "worse" algorithms may have much better cache re-use

Data structures

- Changing the programs data structures can often give good performance improvements
  - These are often global changes to the program and therefore expensive.
    - Code re-writing tools can help with this.
    - Easier if data structures are reasonably opaque, declared once
      - objects, structs, F90 types, included common blocks.
  - As memory access is often the major performance bottleneck the benefits can be great.
    - Improve cache/register utilisation.
    - Avoid pointer chasing
  - May be able to avoid memory access problems by changing code structure in key areas instead.

Code structure

- Most optimisations involve changes to code structure
  - Loop unrolling
  - Loop fusion
  - routine in-lining.
- Often overlap with optimisations attempted by the compiler.
  - Often better to help the compiler to do this than perform change by hand.
- Easier to implement than data changes as more localised.
  - Performance impact is often also smaller unless the code fragment is a major time consumer.
- Performance improvement often at the expense of code maintainability.
  - Try to keep the unoptimised version up to date as well.

Parallelism

- Parallel computing is usually an optimisation technique
  - Multiple processors are used to improve run-time
- Multiple cores within a node
  - Can be exploited by running one MPI process per core
  - May be better to run fewer MPI processes and use threads (e.g. OpenMP) to make use of the other cores.

Summary

- There are many ways to optimise a program
  - need to select the most appropriate for the situation
  - doing nothing is an option
- Must always weigh the benefit (faster execution) against costs:
  - programmer effort
  - loss of maintainability
  - loss of portability
- Performance measurement, analysis and testing are key parts of the process.