MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 1 - Introduction

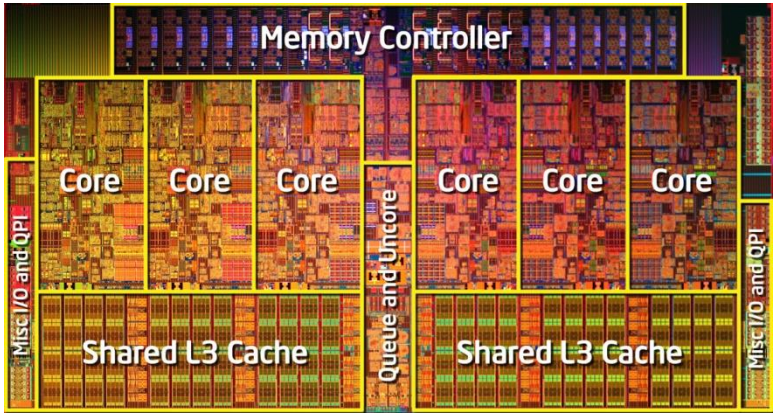
Graham Riley

Logistics

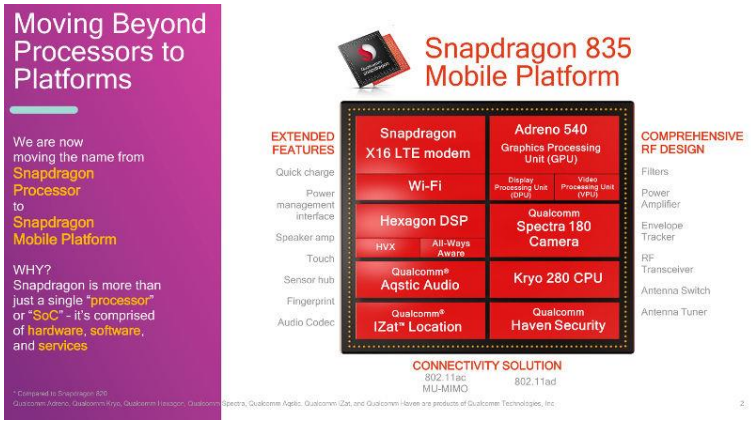
- Lectures – 15 (perhaps one guest spot)
  - Tuesday 1500 (Kilburn 1.5), Thursday 0900 (Kilburn 1.5)
- Labs – 6 (scheduled hours)
  - starting Wednesday 20th February (NOTE: the day!)
  - 3 basic exercises in multiprocessor programming
- Pre-requisites
  - COMP25212 (COMP25111, COMP15111)
- Assessment
  - 75% Exam – Closed book, answer all three questions
  - 25% Lab - Lab submission is via Blackboard
- http://studentnet.cs.manchester.ac.uk/ugt/2018/COMP35112/

Chip Multiprocessors?

Transistors



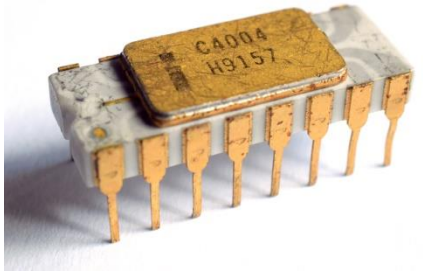
6 Core Intel i7
2011, ~1.1 Billion transistors



Qualcomm Snapdragon 835
SoC (Quad core, GPU, DSP…)
2017, ~3 Billion transistors

Course Summary

- For over 40 years we have seen a continual increase in the power of processors
- This has been driven primarily by improvements in (silicon) integrated circuit (IC) technology
- Circuits will continue to get bigger (more transistors on a single ‘circuit’)
- But the basic circuit speed has now become limited
- Architectural approaches to increase single processor speed have been exhausted



Intel 4004
1st commercial IC processor, 1971
2,300 transistors!

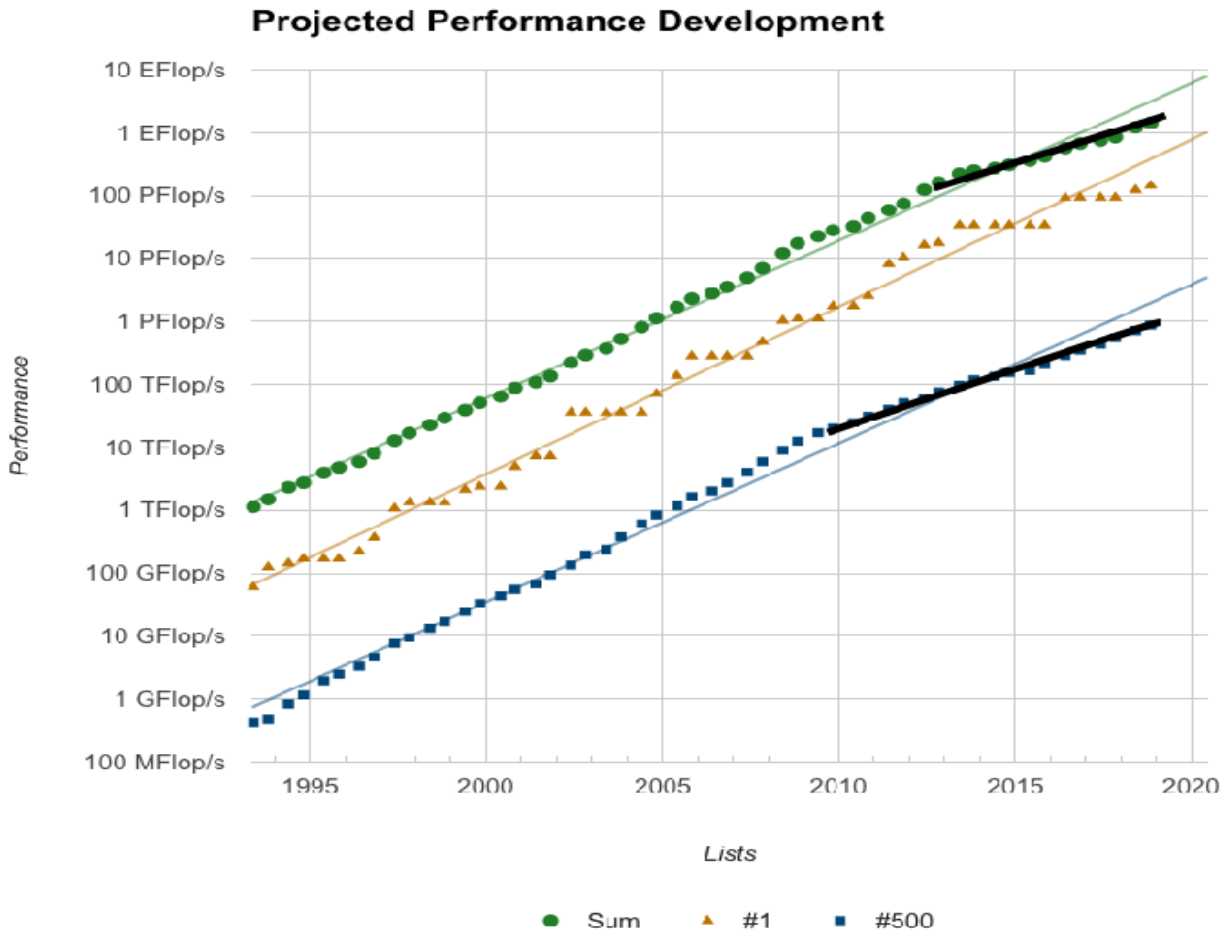
By Thomas Nguyen - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=47684767

Course Summary (cont.)

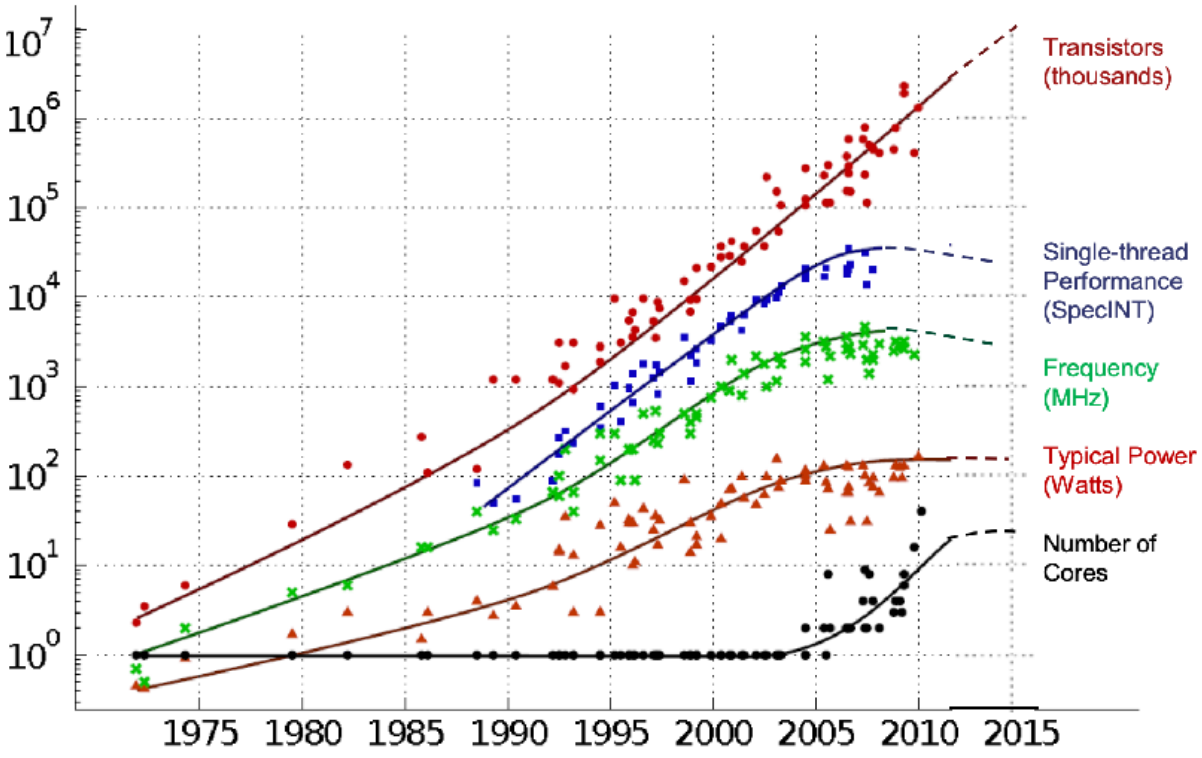
- But if we have lots of transistors on a single circuit, we can build lots of closely coupled processors to work together
- Sounds easy, but .....
  - Architectural Issues
    - What kind of processor(s) to use?
    - How are processors connected?
    - How is the memory organised?
  - Software Issues
    - How do we program general purpose applications to run on new parallel hardware?

End of Moore's Law?

(from top500 November 2018, with my trend-lines)



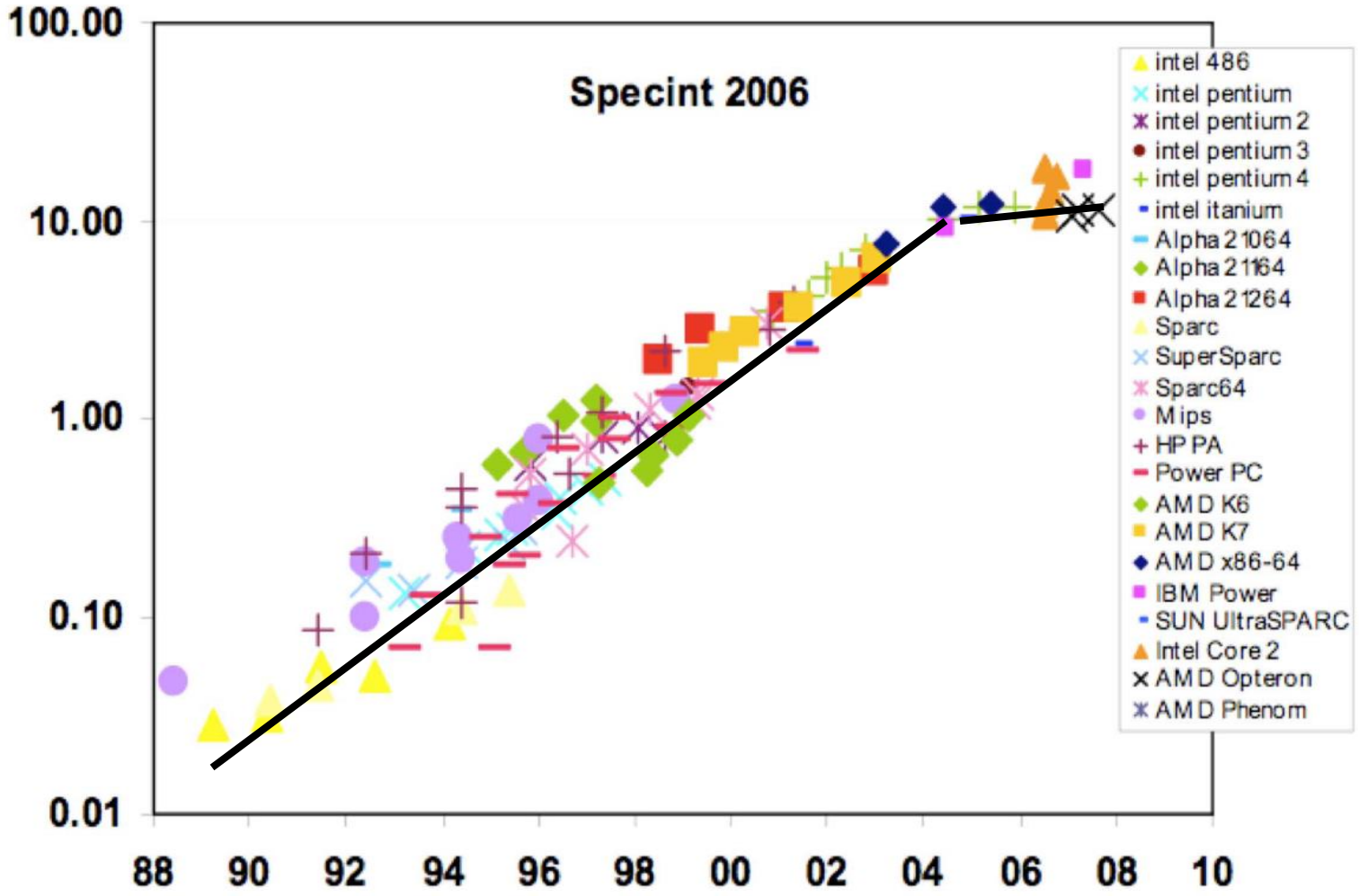
End of Dennard Scaling



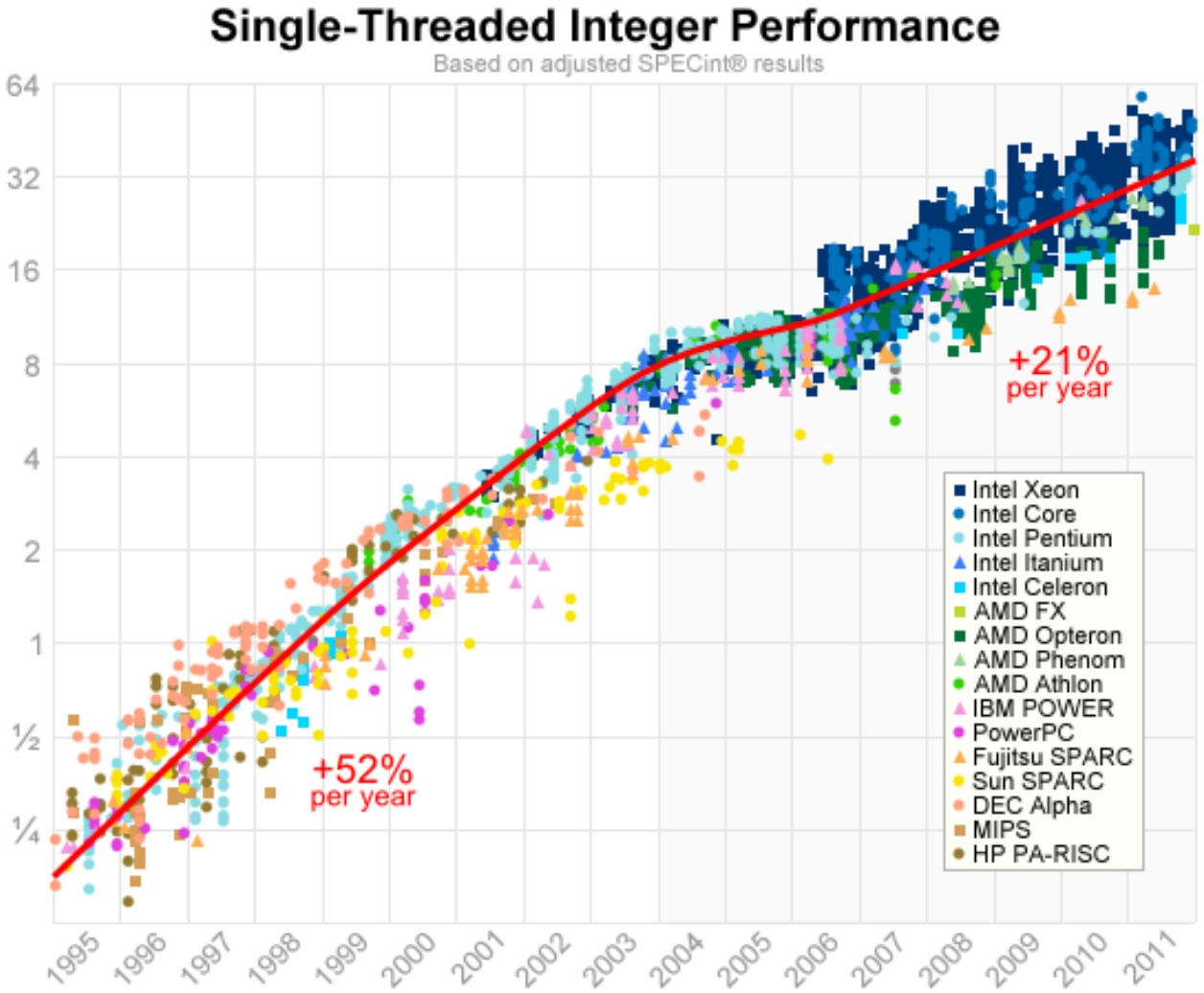
Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

End of Dennard Scaling: Frequency can't keep increasing because power can't be removed from the chips, they'd melt!

Single Core Performance



Single Core Performance



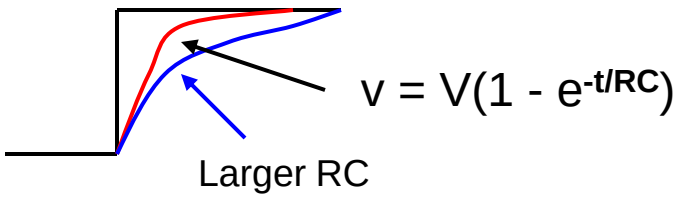
Moore's Law

Not really a law, more an observation/prediction

- Transistor count doubles every 18 months
  - Feature size – basically transistor size
    - Intel 386 (1985) : 10^{-6} metres (275,000 transistors)
    - Intel Core 2 (2008) : 45 x 10^{-9} metres (820 million)
    - Factor (size ratio)^2 = 500
  - Die Size
    - 386 : 100mm^2
    - Core 2 : 280mm^2
    - Factor = 3
- Mostly due to feature size

Smaller ⇒ Faster

- Transistor switching speeds are determined by
  - The transistor (gate) capacitance (C)
  - The resistance of the driving circuit (R)



  - Delay determined by the product RC (time constant)
  - Reducing wire length & width: R stays the same
  - Reducing gate area: C reduces
  - So RC reduces making circuit faster

But We Hit Limits

- This cannot continue
  - Previous analysis is too simple – things like interconnect capacitance start to prevent further speed increases
  - Power density increasing (more watts per unit area) ⇒ cooling a serious problem (See: “Dennard Scaling”, 1974)
  - Small transistors have less predictable characteristics as transistor sizes start to approach the atomic structure (impurity density) of the semiconductor ⇒ cannot build reliable circuits

How To Go Faster?

- Use extra transistors to build more complex single core processors?
- e.g. more parallel pipelines to exploit more ILP (Instruction Level Parallelism)
  - Experience shows that exploitation of ILP produces significantly diminished returns beyond about 4 pipelines
- e.g. bigger caches – lower miss rates
  - Experience shows that payback for bigger caches also diminishes rapidly

The 'Solution' ⇒ Multiple Cores

- Put multiple CPUs (cores) on a single integrated circuit (chip) ⇒ "multicore chip" or "chip multiprocessor"
- Use the multiple CPUs in parallel to achieve higher performance
- Simpler to design than a more complex single processor (basically replication)
- Need more computing power? – just add more cores
- Simple in principle, but the practice is a bit more difficult

The Multicore 'Roadmap'

year, cores per chip, feature size

- 2006, ~2 cores, 65nm
- 2008, ~4 cores, 45nm
- 2010, ~8 cores, 33nm
- 2012, ~16 cores, 23nm
- **2014**, ~32 cores, 16nm
  sharing discontinuity
- 2016, ~64 cores, 12nm
- **2018**, ~128 cores, 8nm
- 2020, ~256 cores, 6nm
- 2022, ~512 cores, 4nm
- **2024**, ~1024 cores, 3nm
- 2026, ~2048 cores, 2nm
  scale discontinuity?
- 2028, ~4096 cores
- 2030, ~8192 cores
- **2032**, ~16384 cores

Can We Use Multiple Cores?

- Small numbers of cores can be used for separate tasks – e.g. run a virus checker on one core and an application on another – i.e. (OS) process-level parallelism
- But, if we want increased performance *on a single application*, we need to use **parallel programming** for each application
- Need collections of pieces of code (threads?) **all working together** to solve a single problem

ILP vs. TLP

- ILP = Instruction Level Parallelism
- TLP = Thread Level Parallelism
- ILP : Program is a single sequence of basic instructions, but we may be able to execute some of these in parallel, even out of order. However, the overall result must be exactly the same as that produced when the whole sequence was executed in order
- TLP : Program is a collection of 'threads', each of which is a sequence of basic instructions. Many threads may execute in parallel and/or in any order. The overall result must be 'the same' (deterministic) whatever the details of execution (?)

Thread Level Parallelism

- You should be familiar with the basics from Java
- Programmer must divide program into sections which can execute as threads
- Main problems arise when multiple threads share variables which they want to update; care is needed if the overall result of the program is to be independent of the exact order of execution (see previous slide)
- In Java this is handled by the use of the **synchronized** keyword – more later

Thread Execution

- Thread based programs (e.g. in Java) can be made to run, apparently in parallel, on a single core processor by switching between execution of threads (exact mechanism is implementation dependent), but overall time taken would be total serial time of threads
- Obviously, if we had multiple cores available, it would be possible to run threads truly in parallel (up to the number of cores, P) and, ideally, execution time would be reduced (by a factor of P)

Forms of Parallelism

- ILP is a very limited form of parallelism extracted automatically from a serial instruction stream
- TLP can be used to generate large amounts of parallelism by writing an appropriate parallel program – but this, of course, assumes that the algorithm is suitable to express in a parallel form

- TLP is ‘general purpose’ in that it makes no assumption about the structure of the parallelism, i.e. parallel threads can each be doing something completely different

Data Parallelism

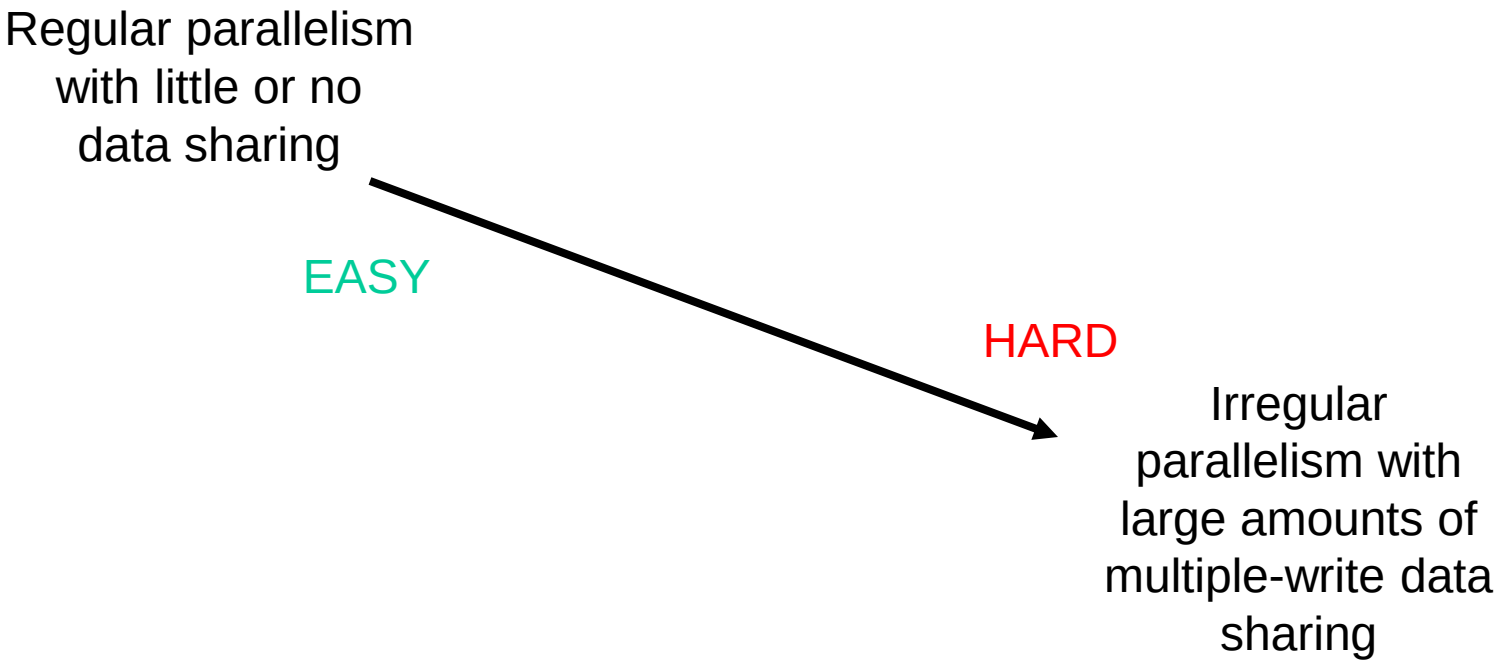
- Some programs may contain more structured parallelism
- If this exists, it is often worthwhile using its properties to simplify implementation
- Data parallelism is a prime example – usually associated with computations on a multi-dimensional array
- Many array computations perform the same (or very similar) computation on all elements of an array, often using only ‘adjacent’ elements as input values

Data Parallelism Examples

- General
  - Matrix multiply (used heavily in CNNs, for example)
  - Fourier transform
- Graphics
  - Anti-aliasing
  - Texture mapping
  - Illumination and shading
- Differential Equations
  - Weather/climate forecasting
  - Engineering simulation (and “Physics” in Games)
  - Financial modelling

Complexity of Parallelism

- Parallel programming is generally considered to be difficult, but depends a lot on the program structure



Chip Multiprocessor Issues

- How should we build the hardware?
  - How are cores connected?
  - How are they connected to memory?
  - Should they reflect particular parallel programming patterns (e.g. data parallelism)?
  - Simple vs. complex cores?
  - General vs. Special Purpose (e.g. graphics processors)?
- How should we program them?
  - Extended ‘conventional’ languages?
  - Domain specific languages?
  - Totally new approaches?

Overview of Lectures

- Introduction – parallel options in hardware & software
- Thread-based programming
- Homogeneous shared memory multiprocessors
- Hardware support for threads
- Alternative programming views
- Speculation and transactional memory
- Memory consistency
- Heterogeneous processors/cores and programs
- Radical approaches

Overview of Labs

- Scheduled sessions during Wednesday slots @ 1100
- Will probably require some 'unscheduled' time!
- Exercises use Java as a parallel programming language
- 1: Delivering speedup (for vector addition)
  - 2 hours scheduled time (20th February/27th February)
- 2: Mergesort (one approach to parallel sorting)
  - 2 hours scheduled time (13th March/20th March)
- 3: Relaxation (to solve Poisson's equation)
  - 2 hours scheduled time (27th March/3rd April)

Lab Deadlines

- As per normal …
- Deadline is 10:00am one week after the last of the scheduled lab sessions for each exercise
  - Exercise 1 – 10:00 Wednesday 6th March
  - Exercise 2 – 10:00 Wednesday 27th March
  - Exercise 3 – 10:00 Wednesday 10th April
- We shall endeavour to get feedback to you within one week of these deadlines
- Lab submission is via the COMP35112 Blackboard site, where details of the exercises can also be found once you have registered your details