Durham University

PHYS52015 Core Ib: Introduction to High Performance Computing (HPC)

Session I: Introduction to parallelism

Christopher Marcotte

Michaelmas term 2023

Outline

- The lecture concept
- No free lunch
- Flynn's taxonomy
- Speedup laws
- Parallelisation concepts

pollev.com/christophermarcotte820

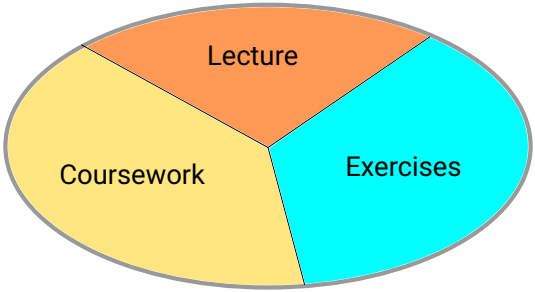
Disclaimer

Durham University

You need *C* for this course – not that all high-performance computing is done in *C*, but it is a standard.

```c
#include <stdio.h>
int main() {
        printf("hello, world");
        return 0;
}
```

The three didactic pillars



- 4 weeks $\times$2 per week sessions
- Lectures are split up into theory part plus hands-on
- Hands-on: BYOD, group work *encouraged*
- Notes: plenty from last year, can adapt as we go along
- Third didactic pillar is *coursework* – to be released later

Tools

- Experiments/source code online on Blackboard Ultra
- It is all plain C/C++
- Using a Unix environment is strongly recommended
- Access to Hamilton/COSMA can be granted to students
  Please check *Week 0* on Blackboard
- Tools used:
  - Up-to-date GCC (`gcc --version`)
  - OpenMPI – already on Hamilton, but may want for local testing
  - Intel Parallel Studio (free for students)

Moore's law

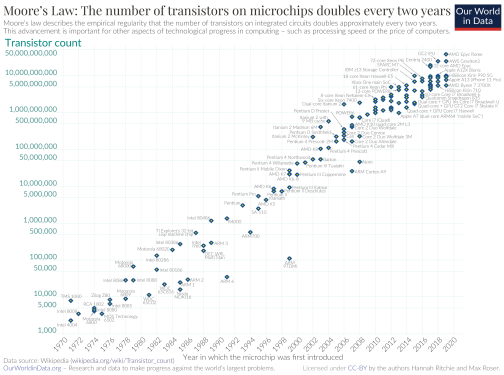
Gordon Moore, 2004

> Moore's Law: The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.

- G. Moore: Co-founder of Intel (R&D director of Fairchild Semiconductor)
- G. Moore: *Cramming more components onto integrated circuits*. Electronics Magazine (1965)
- Cf. Intel's tick-tock policy: change micro architecture vs. die shrink
- Carver Mead (CalTech) coined the term *Moore's law* (1975)

$\Rightarrow$ It is about chip complexity, not about speed

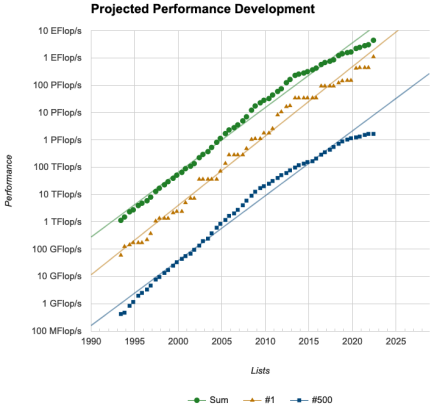
Wikipedia validates Moore's law



Source: Wikipedia `https://en.wikipedia.org/wiki/Transistor_count`

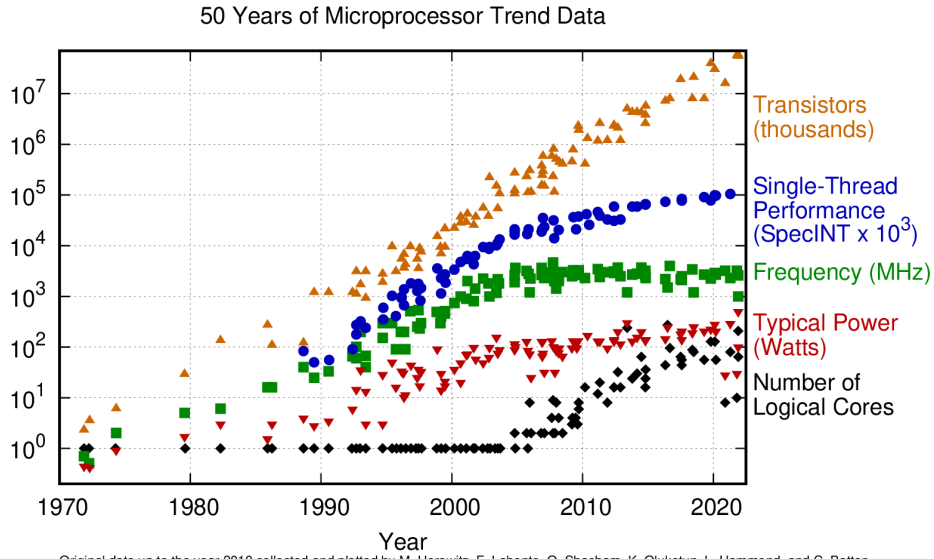
- Moore's law is a statement on transistor count/hardware complexity for fixed price
- Moore's law used to translate directly into performance for decades
- Let's study performance growth for the biggest machines in the world

(Super)Computer speed follows Moore's law



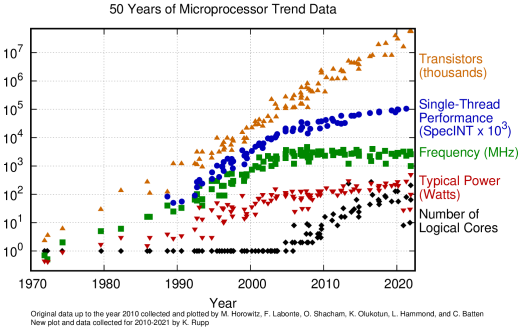
Source: www.top500.org

- Source: Top-500 (`www.top500.org` from SC and ISC)
- Moore's law seems to continue to hold for speed, too
- However . . .



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

The free lunch is over . . .



Karl Rupp *et al*, `https://github.com/karlrupp/microprocessor-trend-data`

- These curves include data across all vendors
- Seems that the “free lunch” (single-threaded performance) is over
- Increase of performance stems from **increase in concurrency**

$\Rightarrow$ learn how to exploit concurrency

. . . though we rely on performance improvements

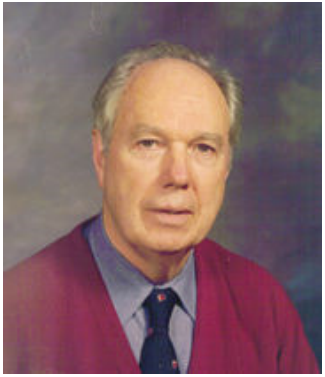
Three classic objectives of HPC/science:

- Throughput & efficiency
  Reduce cost as results are available faster; scientists and engineers can do more experiments in a given time frame or study more parameters (UQ)
- Response time
  Urgent computing as required for tsunami prediction or in medical applications, e.g.; interactive parameter studies (computational steering)
- Problem size
  New insight through never-seen resolution/zoom into scales

> Solution: If chips don't become faster anymore, we have to squeeze more of them into one computer.

> Implication: If computers become "more parallel", we have to be able to write "more parallel" codes.

Michael J. Flynn



- Michael J. Flynn
- May 20, 1934 in New York City
- Stanford University (emeritus)
- Flynn, M.J.: Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computing C, 1972, pp. 948–960

Flynn's taxonomy

- **Instruction stream**: sequence of commands (top-down classification)
- **Data stream**: sequence of data (left-right classification)

Flynn's taxonomy

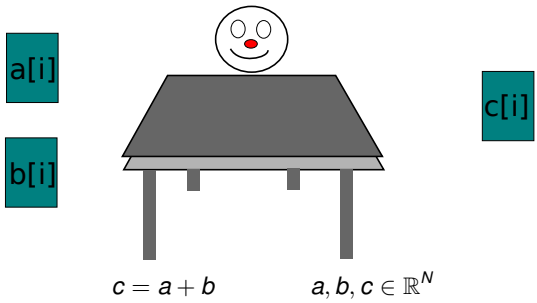
- **Instruction stream**: sequence of commands (top-down classification)
- **Data stream**: sequence of data (left-right classification)

The combination of data stream and instruction stream yields four combinations:

- SISD
- SIMD
- MISD
- MIMD

		Instruction streams	
		single	multiple
Data streams	single	SISD	MISD
	multiple	SIMD	MIMD

SISD



- SISD architecture: One ALU and one activity a time
- Runtime: $(2 \cdot load + add + store) \cdot N$
- Usually *load*, *add*, *store* require multiple cycles (depend both on hardware and environment such as caches; cf. later sessions)

Example: (almost) assembler code in SISD

$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

```
for (int i=0; i<N; i++) {
  load a[i]
  load b[i]
  add
  store c[i]
}
```

Example: (almost) assembler code in SISD

$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

```
for (int i=0; i<N; i++) {
  load a[i]
  load b[i]
  add
  store c[i]
}
```

This is the model of a computer most people will typically employ when reasoning about their program – a sequence of operations, completing in order, without the details of data locality or hardware.

Assembler code running on MIMD hardware

$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

```
for (int i=0; i<N/2; i++) { // Processor 1
  load a[i]
  load b[i]
  add
  store c[i]
}
for (int i=N/2; i<N; i++) { // Processor 2
  load a[i]
  load b[i]
  add
  store c[i]
}
```

Assembler code running on MIMD hardware

$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

```
for (int i=0; i<N/2; i++) { // Processor 1
  load a[i]
  load b[i]
  add
  store c[i]
}
for (int i=N/2; i<N; i++) { // Processor 2
  load a[i]
  load b[i]
  add
  store c[i]
}
```

Thus MIMD is frequently thought about as multiple SISD computers working collaboratively on sub-problems.

MIMD remarks

MIMD describes many processors working on their data independently.

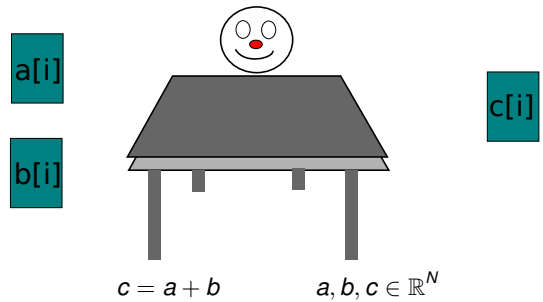
SPMD – Single Program Multiple Data

- Not a hardware type but a programming paradigm
- All chips run same instruction stream, but they might process different steps at the same time (no sync)
- But: MIMD doesn't say that we have to run the same program/algorithm everywhere

Classification

- **SISD**
  - von Neumann's principle
  - classic single processors with one ALU
  - one instruction on "one" piece of data a time
- **MIMD**
  - classic multicore/parallel computer: multiple SISD
  - asynchronous execution of several instruction streams (though it might be copies of the same program)
  - they might work on same data space (shared memory), but the processors do not (automatically) synchronise

SIMD



- ALU: One activity a time
- Registers: Hold two pieces of data (think of two logical regs)
- Arithmetics: Update both sets at the same time
- Runtime: $(4 \cdot \mathit{load} + \mathit{add} + 2 \cdot \mathit{store}) \cdot N/2$
- *load*, *add*, *store* slightly more expensive than in SIMD
- *load* and *store* can grab two pieces of data in one rush when they are stored next to each other

Almost assembler code running on SIMD hardware

$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

where $N/n \in \mathbb{N}$.

```
for (int i=0; i<N; i+=n) { // Processor 1
  load a[i], a[i+1], ..., a[i+n]
  load b[i], b[i+1], ..., b[i+n]
  vadd
  store c[i], c[i+1], ..., c[i+n]
}
```

Almost assembler code running on SIMD hardware

$$c = a + b \qquad a, b, c \in \mathbb{R}^N$$

where $N/n \in \mathbb{N}$.

```
for (int i=0; i<N; i+=n) { // Processor 1
  load a[i], a[i+1], ..., a[i+n]
  load b[i], b[i+1], ..., b[i+n]
  vadd
  store c[i], c[i+1], ..., c[i+n]
}
```

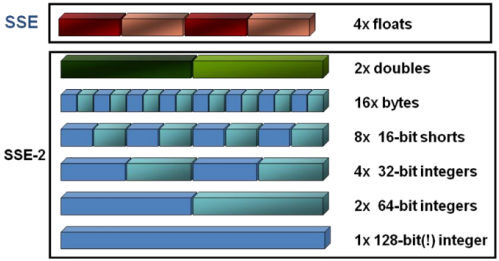
SIMD operates at a lower level of hardware parallelism than the CPU core – it operates at the level of execution units (ALU & FPU).
Likewise, it is frequently preferable to implement SIMD parallelism automatically, using compiler optimization or `#pragma`s.

AVX/SSE

Vectorisation: Rewrite implementation to make it work with small/tiny vectors.

- Extension to four or eight (single precision) straightforward
- Technique: Vectorisation/vector computing
- Hardware realisation: Large registers holding multiple (logical) registers

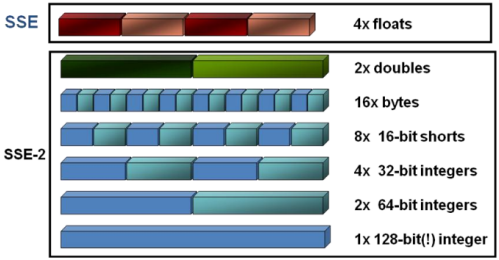
SSE



(C) Intel

- SSE = Streaming SIMD Extension
- SIMD instruction instruction set introduced with Pentium III (1999)
- Answer to AMD's 3DNow (computer games)
- Concept stems from the days when Cray was a big name
- Around 70 single precision instructions
- AMD implements SSE starting from Athlong XP (2001)

AVX



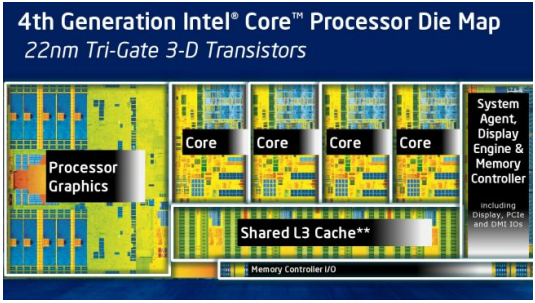
(C) Intel

- Double number of registers
- However, multiple cores might share registers (MIC)
- Operations may store results in third register (original operand not overwritten)
- AVX 2–AVX512: Gather and scatter operations and fused multiply-add

Classification

- **SISD**
  - von Neumann's principle
  - classic single processors with one ALU
  - one instruction on "one" piece of data a time
- **MIMD**
  - classic multicore/parallel computer: multiple SISD
  - asynchronous execution of several instruction streams (though it might be copies of the same program)
  - they might work on same data space (shared memory), but the processors do not (automatically) synchronise
- **SIMD**
  - synchronous execution of one single instruction
  - vector computer: (tiny) vectors
- **MISD**
  - pipelining (we ignore it here)

Flynn's taxonomy: lessons learned/insights



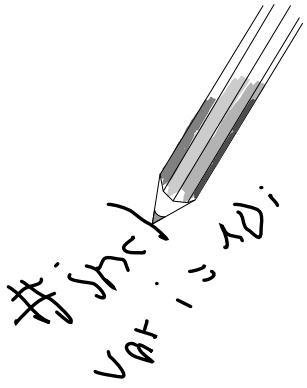
- All three paradigms are often combined (see Haswell above)
- There is more than one flavour of hardware parallelism
- There is more than one parallelisation approach
- MIMD substypes:
  - shared vs. distributed memory
  - races vs. data inconsistency
  - synchronisation vs. explicit data exchange

Gene Amdahl

Gene Amdahl, 2008

- Gene Amdahl
- November 16, 1922 in South Dakota
- Computer architect (own company and IBM)
- Amdahl, G. M. (1967): *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. AFIPS Conference Proceedings (30): 483–485

A thought experiment



- A single-core computer requires 64s to complete a job.
- How much time should a dual core computer require?
- How much time should a quadcore require?
- How would you define the terms speedup and efficiency?
- Write down three reasons why the speedup might actually be lower than expected.

Speedup and efficiency

We are usually interested in two metrics:

Speedup: Let $t(1)$ be the time used by one processor. $t(p)$ is the time required by p processors. The speedup is

$$S(p) = \frac{t(1)}{t(p)}.$$

Efficiency: The efficiency of a parallel application is

$$E(p) = \frac{S(p)}{p}.$$

Amdahl's Law

$$t(p) = f \cdot t(1) + (1 - f)\frac{t(1)}{p}$$

Ideas:

- Focus on simplest model, i.e. neglect overheads, memory, ...
- Assume that code splits up into two parts: something that has to run serially (f) with a remaining code that scales
- Assume that we know how long it takes to run the problem on one core.
- Assume that the problem remains the same but the number of cores is scaled up

Remarks:

- We do not change anything about the setup when we go from one to multiple nodes. This is called **strong scaling**.
- The speedup then derives directly from the formula.
- In real world, there is some concurrency overhead (often scaling with p) that is neglected here.

Speedup and efficiency—Strong Scaling

Durham University

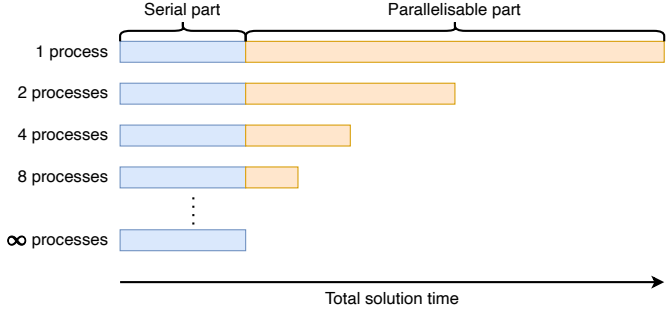
> Speedup: Let $t(1)$ be the time used by one processor. $t(p)$ is the time required by p processors. The speedup is $S(p) = t(1)/t(p)$.

- What is the scaling of the speedup if $f = 0$?
- Are speedups smaller than 1 possible?
- What is superlinear speedup?

> Efficiency: The efficiency of a parallel application is $E(p) = S(p)/p$.

- What is an upper bound for the efficiency?
- What is a lower bound for the efficiency?

Strong Scaling – intuition



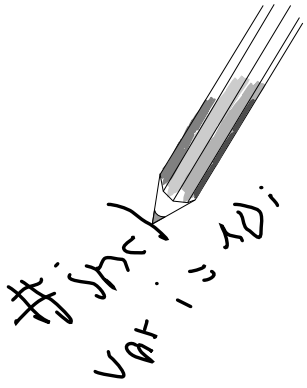
- Given f as the serial fraction, and $t(p) = t(1)(1 + \frac{1-f}{p})$
- Then $S(p) = 1/(f + (1-f)/p)$,
- The speedup is limited by the serial fraction: $\lim_{p\to\infty} S(p) = 1/f$
- Want to maintain a parallel efficiency of 80% – how does the parallel fraction of the code need to scale as we add more processors?

John Gustafson

John Gustafson, 2008

- John L. Gustafson
- January 19, 1955
- Computer scientists (Intel, AMD and others)
- John L. Gustafson: *Reevaluating Amdahl's Law*, Communications of the ACM 31(5), 1988

A thought experiment



- A supercomputer with 64 nodes requires 2s to complete a job.
- How much time would a single node computer require if the program has a sequential fraction of instructions f?
- Write down the speedup.

Gustafson's Law

$$t(1) = f \cdot t(p) + (1 - f)t(p) \cdot p$$

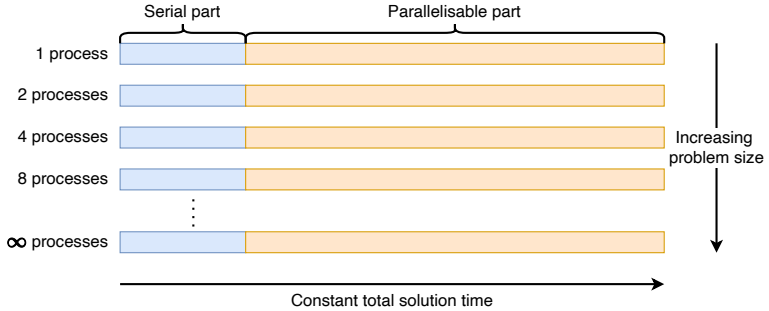
Ideas:

- Focus on simplest model, i.e. neglect overheads, memory, ...
- Assume that code splits up into two parts (cf. BSP with arbitary cardinality): something that has to run serially (f) and the remaining code that scales.
- Assume that we know how long it takes to run the problem on p ranks.
- Derive the time required if we used only a single rank.

Remarks:

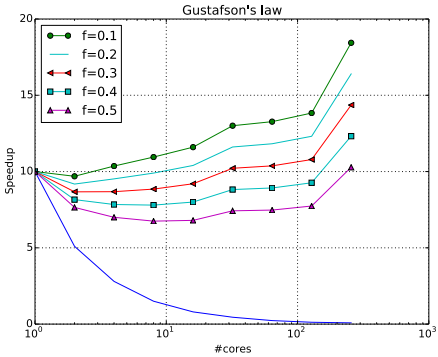
- Single node might not be able to handle problem and we assume that original problem is chosen such that whole machine is exploited, i.e. problem size is scaled. This is called **weak scaling**.
- The speedup then derives directly from the formula.
- In real world, there is some concurrency overhead (often scaling with p) that is neglected here.

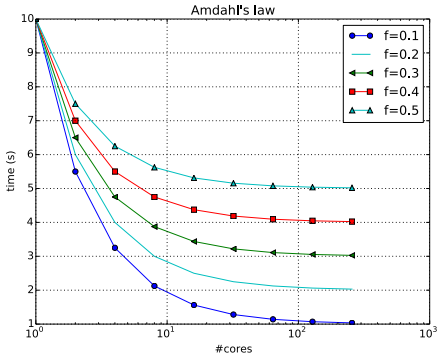
Weak Scaling – intuition



- Perfect *weak scaling* matches the number of processes to the amount of work to do – giving a *constant* run time rather than shorter with more processors
- Given, $t(1) = t(p)f + p(1-f)t(p)$, then $S(p) = f + (1-f)p$
- What is the efficiency as $\lim_{p\to\infty} E(p)$? $\lim_{p\to 1} E(p)$?

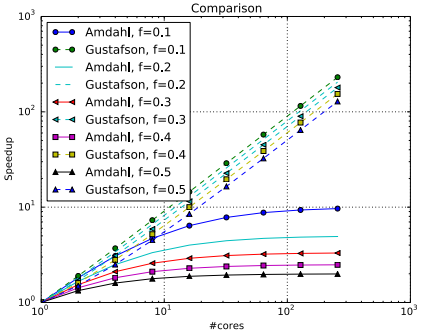
Python Exercise





Disclaimer: We will typically not be using Python!

Python exercise: Comparison of the speedup laws



- It depends on whether you fix the problem size.
- It is crucial to clarify assumptions a priori.
- It is important to be aware of shortcomings.

$$t(p) = f \cdot t(1) + (1-f)\frac{t(1)}{p} \qquad t(1) = f \cdot t(p) + (1-f)t(p) \cdot p$$

$$S(p) = t(1)/t(p) = \frac{1}{f + (1-f)/p} \qquad S(p) = t(1)/t(p) = f + (1-f)p$$

Shortcomings of simple performance models

- These are very simple performance models with a number of assumptions.
  - That a program can be split into two separate parts which can be treated independently.
  - That changing a program to run in parallel will not affect the serial part.
  - That there is no communication overhead, or parallel management which scales with p.
  - That every processor in p is precisely the same, and experiencing the same environment.
- Frequently, real programs will change behaviour significantly when made parallel on the same hardware – e.g., due to frequency scaling or cache effects.
  - Computers are much more than a collection of loose CPU cores!
  - Phone SoCs are a great example of heterogeneity and 'the other parts' leading to dramatically improved performance.
- Is *time* the most relevant metric anymore?
- Some problems simply react differently to parallelisation!

Shortcomings of simple performance models

- These are very simple performance models with a number of assumptions.
  - That a program can be split into two separate parts which can be treated independently.
  - That changing a program to run in parallel will not affect the serial part.
  - That there is no communication overhead, or parallel management which scales with p.
  - That every processor in p is precisely the same, and experiencing the same environment.
- Frequently, real programs will change behaviour significantly when made parallel on the same hardware – e.g., due to frequency scaling or cache effects.
  - Computers are much more than a collection of loose CPU cores!
  - Phone SoCs are a great example of heterogeneity and 'the other parts' leading to dramatically improved performance.
- Is *time* the most relevant metric anymore?
- Some problems simply react differently to parallelisation!

What about parallel search? If you and p friends need to find a phone number in a phone book (before your time!), does Amdahl's law give us a reasonable model for the time to find the number $t(p)$ given a known $t(1)$?

You have a problem... How do you parallelize it?

Typically in HPC, we have a problem which is well-specified (e.g. in a serial program) but the current approach is insufficient for some reason:

1. Serial is too slow; we need a fast solution
2. Problem is too small; Need to solve a larger or more accurate version
3. Solution is inefficient; we need it solved more efficiently

You have a problem... How do you parallelize it?

Typically in HPC, we have a problem which is well-specified (e.g. in a serial program) but the current approach is insufficient for some reason:

1. Serial is too slow; we need a fast solution
2. Problem is too small; Need to solve a larger or more accurate version
3. Solution is inefficient; we need it solved more efficiently

There are typically three approaches to applying parallel computing to the serial problem (not mutually exclusive):

- Better align with existing hardware capabilities (e.g., SIMD, caches)
- Split the computation so that multiple cores can work on subproblems
- Split the data so that multiple machines can be used to solve parts of the problem simultaneously (distributed memory parallelism)

You have a problem... How do you parallelize it?

Typically in HPC, we have a problem which is well-specified (e.g. in a serial program) but the current approach is insufficient for some reason:

1. Serial is too slow; we need a fast solution
2. Problem is too small; Need to solve a larger or more accurate version
3. Solution is inefficient; we need it solved more efficiently

There are typically three approaches to applying parallel computing to the serial problem (not mutually exclusive):

- Better align with existing hardware capabilities (e.g., SIMD, caches)
- Split the computation so that multiple cores can work on subproblems
- Split the data so that multiple machines can be used to solve parts of the problem simultaneously (distributed memory parallelism)

In every approach, some degree of *collaborative* of *competitive* concurrency or parallelism is possible.

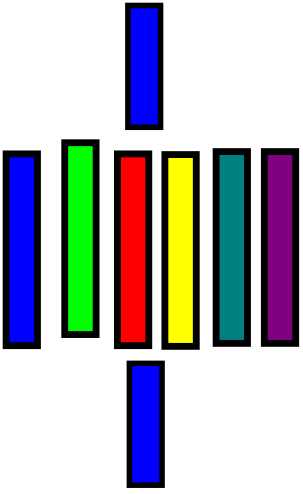
Programming techniques

Durham University

Programming Techniques: For each parallelisation paradigm, there are many programming techniques/patterns/paradigms.

- We will more or less ignore the competition approach.
  - *Competitive* parallelism is mostly relevant for GUIs and games, where responsiveness to user action is paramount.
  - *Cooperative* parallelism is useful for solving large compute-bound problems without interactivity.
  - These are not necessarily antagonistic concepts: Muller, *et al.*, *Competitive Parallelism: Getting Your Priorities Right*; `https://www.cs.cmu.edu/~rwh/papers/priorities/paper.pdf`
- We will today quickly run through three important patterns.
- More patterns will arise throughout the course.

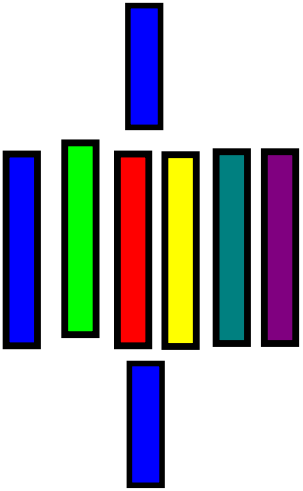
Technique 1: BSP



BSP: Bulk-synchronous parallelism is a type of coarse-grain parallelism where inter-processor communication follows the discipline of strict barrier synchronization. Depending on the context, BSP can be regarded as a computation model for the design and analysis of parallel algorithms, or a programming model for the development of parallel software.

Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, 1990

BSP terminology



Remarks:

- Other name: Fork-join model
- No communication in original model
- Superstep

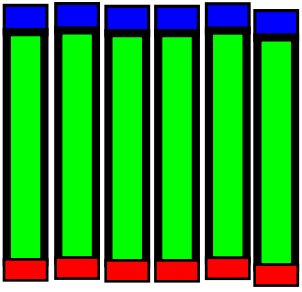
Terminology:

- Bulk
- Fork
- Join/Barrier/Barrier synchronisation
- Grain size (how many subproblems per thread)
- Superstep

Analysis:

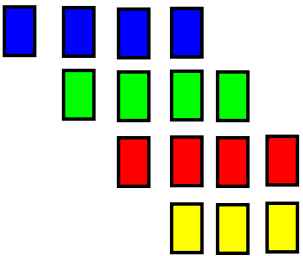
- Parallel fraction f
- Max. concurrency
- Fit to strong and weak scaling

Technique 2: SPMD



SPMD: Single program multiple data is a type of coarse-grain parallelism where every single compute unit runs exactly the same application (though usually on different data). The may communicate with each other at any time. Depending on the context, SPMD can be regarded as a computation model for the design and analysis of parallel algorithms, or a programming model for the development of parallel software.

Technique 3: Pipelining



Functional decomposition does not yield parallelism on its own, but . . .

> Pipelining: In computing, a pipeline is a set of data processing elements connected in series, where the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel.

Applications:

- Assembly line
- Graphics pipeline
- Instruction pipeline (inside chip) / RISC pipeline
- Now coming back into fashion (PyTorch, Gpipe; Huang *et al*) for large ML model training

Summary, outlook & homework

Concepts discussed:

- Motivation of parallel programming through hardware constraints
- Course structure
- Parallel Scaling laws – Amdahl & Gustafson
- Basics of parallelisation strategies

Next:

- OpenMP!