XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 4: Theory of parallel performance

Previous lecture

In the last lecture we started to look at solving problems in parallel:

- **Vector addition**, which can be parallelised for shared memory systems by using the **fork-join** construct.
- Implemented in OpenMP as a single line just before the loop: `#pragma omp parallel for`
- **Mandelbrot set**, which has a **nested loop**.
- Both **data parallel** problems ('**maps**') as calculations in the loop are independent.
- Still difficult to achieve good performance for the Mandlebrot set.

This lecture

Now we will look at some general considerations for **parallel performance**:

- Introduce common **parallel overheads**.
- Common **metrics** for parallel performance.
- Classic **models** for predicting parallel speed-up and highlighting potential pitfalls.

- How these relate to **scaling**, *i.e.* how performance varies with the number of processors and the problem size.

Notation

For this lecture we will use the following notation:

Symbol	Meaning	Notes
n	Problem size	*e.g.* vector size, list length, image size, ...
p	No. processing units	*e.g.* cores, threads, processes, ...
t_s	Serial execution time	'Optimal'
t_p	Parallel execution time	
f	Serial fraction	Amdahl, Gustafson-Barsis

What we are trying to achieve

Assume a problem can be solved by a **serial** algorithm in a time t_s.

- We assume this is **optimal**, *i.e.* cannot be improved (in serial).

In practice the optimal t_s is rarely employed.

- Optimal solution may not be known.
- May be known, but take too long to implement.

Usually consider the serial algorithm which is 'equivalent' to the parallel one.

- For instance, if developing a parallel bubblesort, would probably compare to **serial bubblesort** (rather than quicksort, mergesort, heapsort *etc.*).

Parallel acceleration

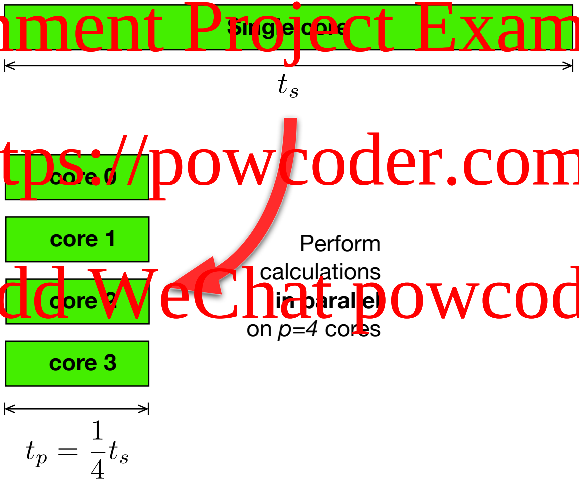
One way to improve on the **serial** execution time t_s is to implement a **parallel** solution on **parallel** hardware.

- May be possible to 'beat' t_s by exploiting **simultaneous calculations**.
- Can also make better use of shared **memory cache**.

Denote the (not necessarily optimal) parallel execution time t_p.

- Measured in same units as t_s.
- On 'as similar as possible' hardware.
- Sometimes known as the **wall clock** time, as it is what 'a clock on the wall' would measure.

Simultaneous calculations (ideally)

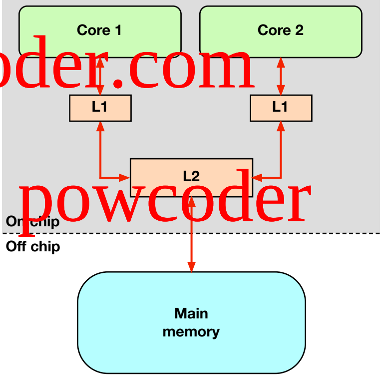


Multi-core memory cache

Recall from Lecture 2 that using multiple cores can make better use of **memory cache**.

Fewer **cache misses**:

- Cache lines pulled up by one core may include data required by another core.
- Depending on how data is arranged in memory and accessed, a parallel code may result in **fewer cache misses** overall that the equivalent serial algorithm.



Challenges to parallel performance

These potential benefits must be offset by the many challenges to achieving good parallel performance.

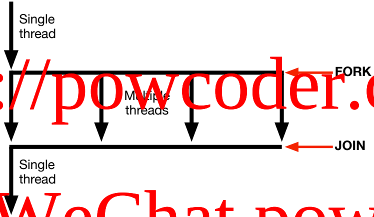
In Lecture 2 we saw one example: **false sharing**.

- Hardware performance loss in maintaining **cache coherency** when two cores repeatedly write to the same cache line, **even though they never read the other core's data**.

Over the coming lectures we will see two important, general challenges: **synchronisation** and **load balancing**.

Synchronisation

In the **fork-join** construct from last lecture, multiple threads complete before the main thread continues.



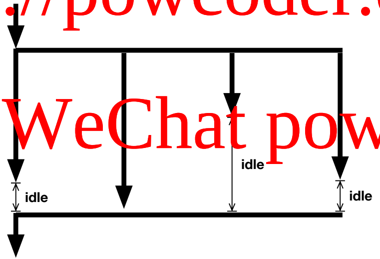
This **joining** requires resources.

- Main thread may repeatedly **probe** worker thread status.
- Alternatively, workers may **signal** their completion to main.
- An example of **synchronisation**.

Load balancing

A related issue is **load balancing**:

- What if the worker threads did not all finish **at the same time**?
- Some would be **idle**, waiting for others to finish.
- Poor use of available resources.



This happens in the Mandelbrot set since each thread performs different numbers of calculations *[cf. last lecture; Lecture 13]*.

Parallel overheads

Even if these challenges could be overcome, there are inevitable **overheads**. For example:

- Time and resources to **create**, **schedule** and **destroy** threads and/or processes during runtime (e.g. fork-join).
- **Communication** between threads/processes not present in the serial equivalent.
- **Computation** not present in serial, e.g. when partitioning the problem size between threads.

The impact may be small or large depending on parallel algorithm and hardware architecture.

Metrics for parallel performance

There are various measurements of parallel *vs.* serial performance.

The most common is the **speedup** S:

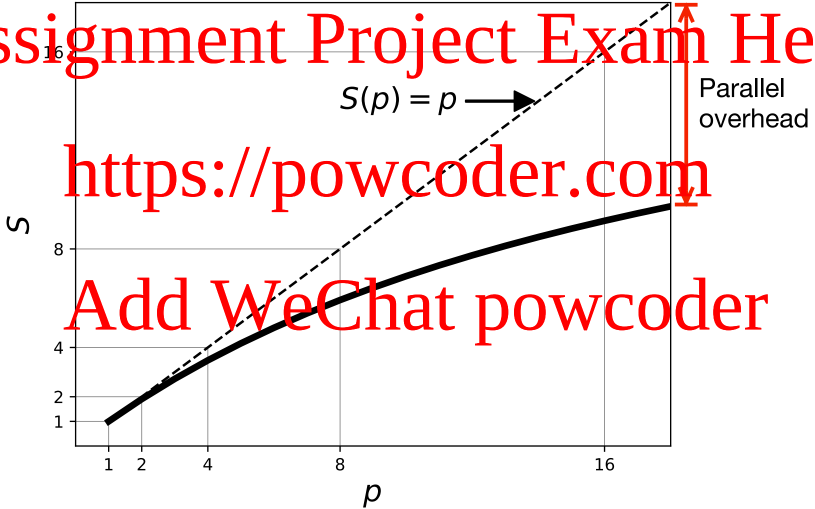
$$S = \frac{t_s}{t_p}$$

- If the parallel version was p times faster than the serial:

$$t_p = \frac{1}{p} t_s \quad \Longrightarrow \quad S = \frac{t_s}{\frac{1}{p} t_s} = p$$

- Rarely realised in practice due to **parallel overheads**.

Speedup example



Superlinear speedup

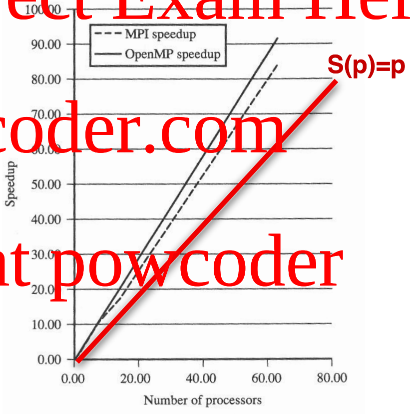
$S(p) > p$ is possible.

- Usually due to memory cache (or suboptimal t_s).

This is **super-linear speedup**.

Example (right): Benchmark computational fluid dynamics algorithm.

However, this is rare - most commonly see $S(p) < p$.



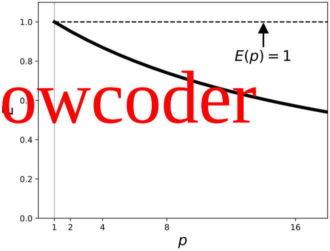
From *Parallel Programming in OpenMP*, Chandra *et al.* (Academic, 2001).

Efficiency

Another common parallel performance metric is the **efficiency** E:

$$E = \frac{t_s}{p t_p} = \frac{S}{p}$$

- 'Ideal' speedup $S = p$ corresponds to $E = 1$.
- Often expressed as a percentage: $E = 1 = 100\%$.
- Typically $E < 1$ due to parallel overheads.
- Superlinear speedup gives $E > 1$.



Models for parallel performance

Desirable to theoretically predict t_p for parallel algorithms.

- Select the 'best' without development and testing.
- Identify 'bottlenecks' for further investigation.

Challenging to derive **precise** equations for t_p:

- Need to include *e.g.* memory cache, thread scheduler *etc.*
- Involve many unknown parameters requiring calibration.
- Would need re-calibration for new hardware.

However, even **simple** models can predict **trends**.

- **Parallel scaling**, which refers to the variation with p.

Amdahl's law

Suppose a fraction f of t_s cannot be parallelised.

$$
\begin{aligned}
t_s &= f t_s + (1-f) t_s \\
\implies t_p &\geq f t_s + \frac{(1-f) t_s}{p} \\
\implies S = \frac{t_s}{t_p} &\leq \frac{t_s}{f t_s + \frac{(1-f) t_s}{p}} = \frac{1}{f + \frac{1-f}{p}}
\end{aligned}
$$

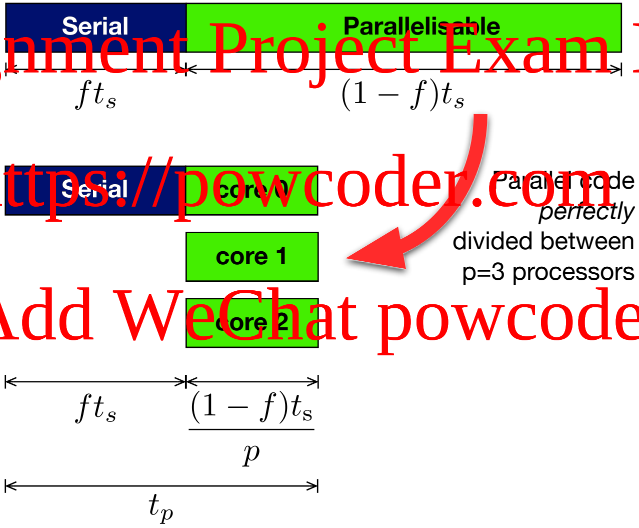
This is **Amdahl's law**[1] (1967).

For large p it predicts $S \leq \frac{1}{f}$ **regardless of** p.

- *e.g.* $f = 0.2$, maximum speedup of 5, **even for p**$=\infty$!

[1] Amdahl, *AFIPS Conference Proceedings* **30**, 483 (1967).

Schematic for Amdahl's law ($p = 3$)



Gustafson-Barsis law

However, Amdahl assumed t_s — and hence n — was fixed.

- Suppose instead n increases with p such that t_p is fixed.

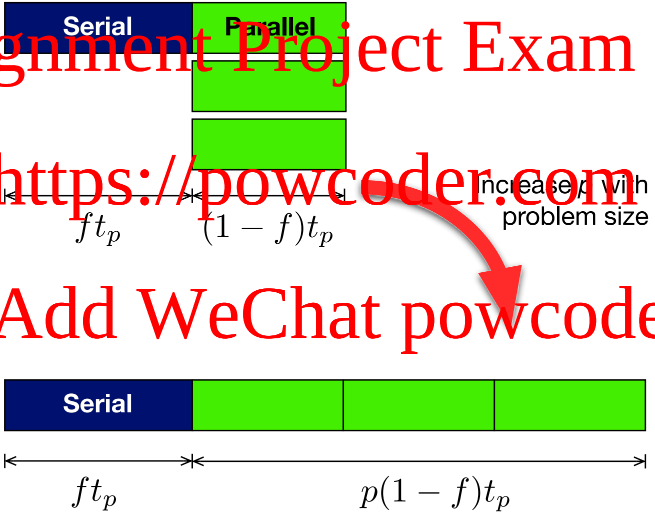
$$\begin{aligned} t_p &= f t_p + (1-f) t_p \\ \implies t_s &\leq f t_p + p(1-f) t_p \\ \implies S &\leq f + p(1-f) = p + f(1-p) \end{aligned}$$

Now $S \leq (1-f)p$ for large p - **no upper bound** as $p \to \infty$.

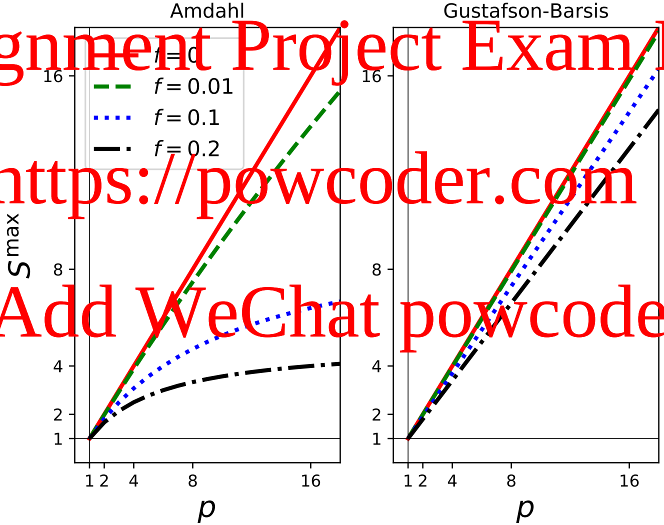
This is the **Gustafson-Barsis law**, or just **Gustafson's law**[1].

[1] Gustafson, *Comm. ACM* **31**, 532 (1988).

Schematic for Gustafson-Barsis law (p=3)



Amdahl *versus* Gustafson-Barsis



Weak *versus* strong scaling

The differences are encapsulated in **weak** *versus* **strong** scaling:

Strong scaling: Increasing p with n fixed.

- **Amdahl's law**.
- Cannot control system size.
- *e.g.* data analysis/mining.

Weak scaling: Increasing n with p.

- **Gustafson-Barsis law**.
- Have freedom to vary n.
- *e.g.* higher resolution meshes for scientific/engineering applications; more/larger layers in neural networks.

Summary and next lecture

Today we have looked at **parallel performance**:

- Two common metrics: **speedup** and **efficiency**.
- Challenging to achieve ideal speedup due to various **parallel overheads**.
- Classic models known as **Amdahl's law** and the **Gustafson-Barsis** law.
- Correspond to **strong** and **weak** scaling, respectively.

Next time we will look more closely at **data dependencies** in parallel loops.