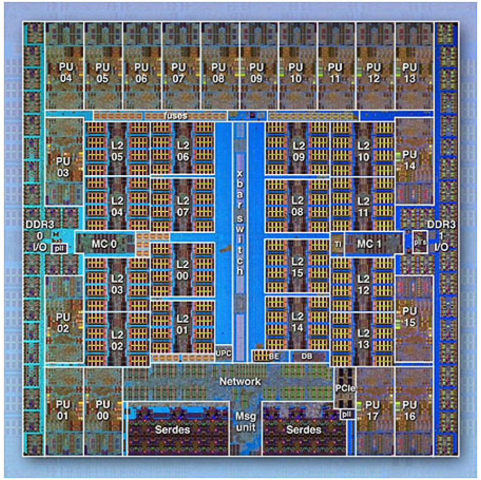
Computer architecture: parallelization

EEE8087

Dr Fei Xia and Dr Alex Bystrov

Why Parallel systems?

- Use multiple processors in parallel to compute faster than a single processor
- Example parallel computing systems:
  - Cluster computers (like supercomputers) contain multiple (few hundreds) PCs combined together with a high speed network – each PC has one or more processors
  - Modern high performance workstations have multiple (up to a dozen or more) multiprocessors
  - Chip multiprocessors (CMP) contains multiple cores on a single chip [check your handheld phones]
- State-of-the-art multicore systems:
  - AMD Opteron (4/8 64-bit x86 cores 3L cache), Intel Nehalem (4/8 64-bit cores 3L cache), 80-core Teraflops Research Processor , Intel Xeon Phi (61x4 processors)
  - IBM Power5/6 processors with dual cores, Sun UltraSparc T2 with eight cores etc.
  - ARM big.LITTLE (in your smart phones in 2+4, 4+4, 2+2+4, etc.), Intel Xeon Phi (61 co-processors)
- Traditionally driven by performance needs; energy consumption is an emerging need



Energy/power advantage

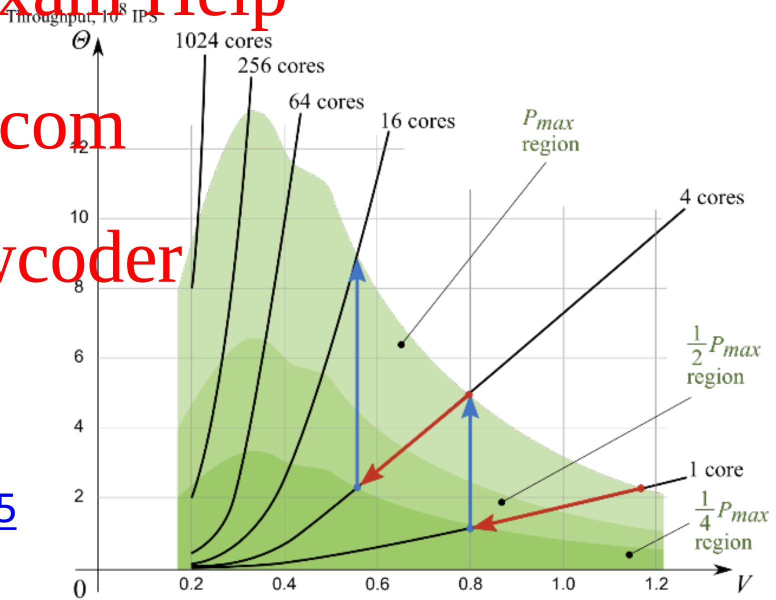
- Dynamic power of MOSFET circuits

  $P = AFV^2$ (from 1st principles - can you derive this?)

  If we run double the number of cores (parallelize by 2) we may get the same performance with $A_2 = 2A, F_2 = {}^F/_2$, but also $V_2 < V$

  Therefore we have $P_2 < P$ for the same execution speed – the same workload will complete in the same amount of time

- In other words, parallelization can help improve performance, or power, or both together
  - https://ieeexplore.ieee.org/document/7999145



Speedup

- How much does the speed of a system improve after an attempt at improving the system:
  - Proportional concept
  - Speed of system after improvement divided by speed of system before improvement

$$Speedup_{improvement} = \frac{Speed_{after\ improvement}}{Speed_{before\ improvement}}$$

$$= \frac{Time_{before\ improvement}}{Time_{after\ improvement}} \in [1, \infty)$$

  - "Improvement" implies that we have better speed after than before
- Improvements may include

  - Increase operating frequency (e.g. overclocking)
  - Use accelerator (e.g. graphics card)
  - Use pipelining (e.g. Intel Hyperthreading)
  - Use multiple/more cores
  - Change hard disk drive to solid state drive
  - Use cache memory
  - But do they automatically guarantee $Speedup \geq 1$?

Speedup

- How much does the speed of a system improve after an attempt at improving the system:
  - Proportional concept
  - Speed of system after change divided by speed of system before change

$$Speedup_{change} = \frac{Speed_{after}}{Speed_{before}}$$

$$= \frac{Time_{before}}{Time_{after}} \in [0, \infty)$$

  - General modification does not guarantee improvement
- Improvements may include

  - Increase operating frequency (e.g. overclocking)
  - Use accelerator (e.g. graphics card)
  - Use pipelining (e.g. Intel Hyperthreading)
  - Use multiple/more cores
  - Change hard disk drive to solid state drive
  - Use cache memory
  - None of these automatically guarantee $Speedup \geq 1$!

The parallelizability of software

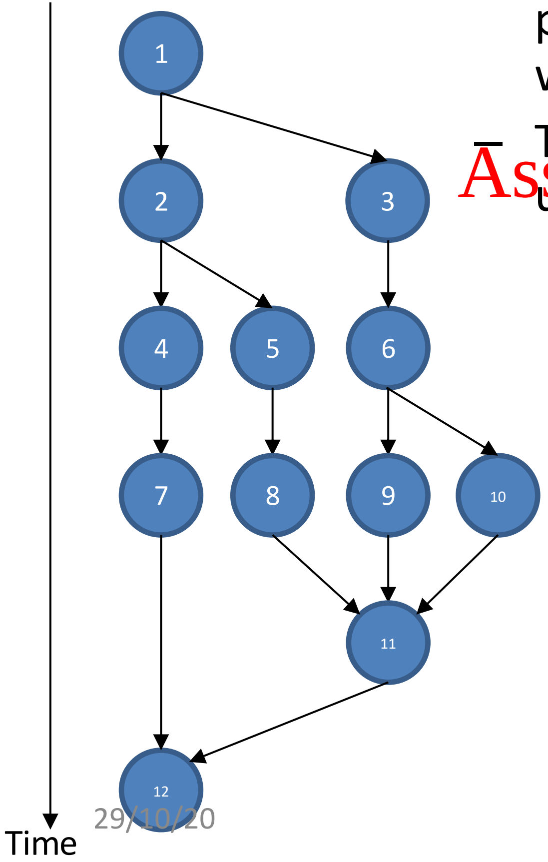
- “Standard” matrix addition code found in books:

```
for (c = 0; c < m; c++) {
    for (d = 0 ; d < n; d++) {
      sum[c][d] = first[c][d] + second[c][d];
        }
    }
```


The additions themselves are fully parallelizable
(zero cross-dependency between different additions in the same step)
But the code is fully sequential!
(does not take advantage of multi-core hardware)

Almost all matrix operations are highly parallelizable – this is why GPUs and NPUs are highly parallel machines

- Parallelism
  - Parallelism indicates how parallelizable a workload/program/job is
  - This can be illustrated by using task graphs



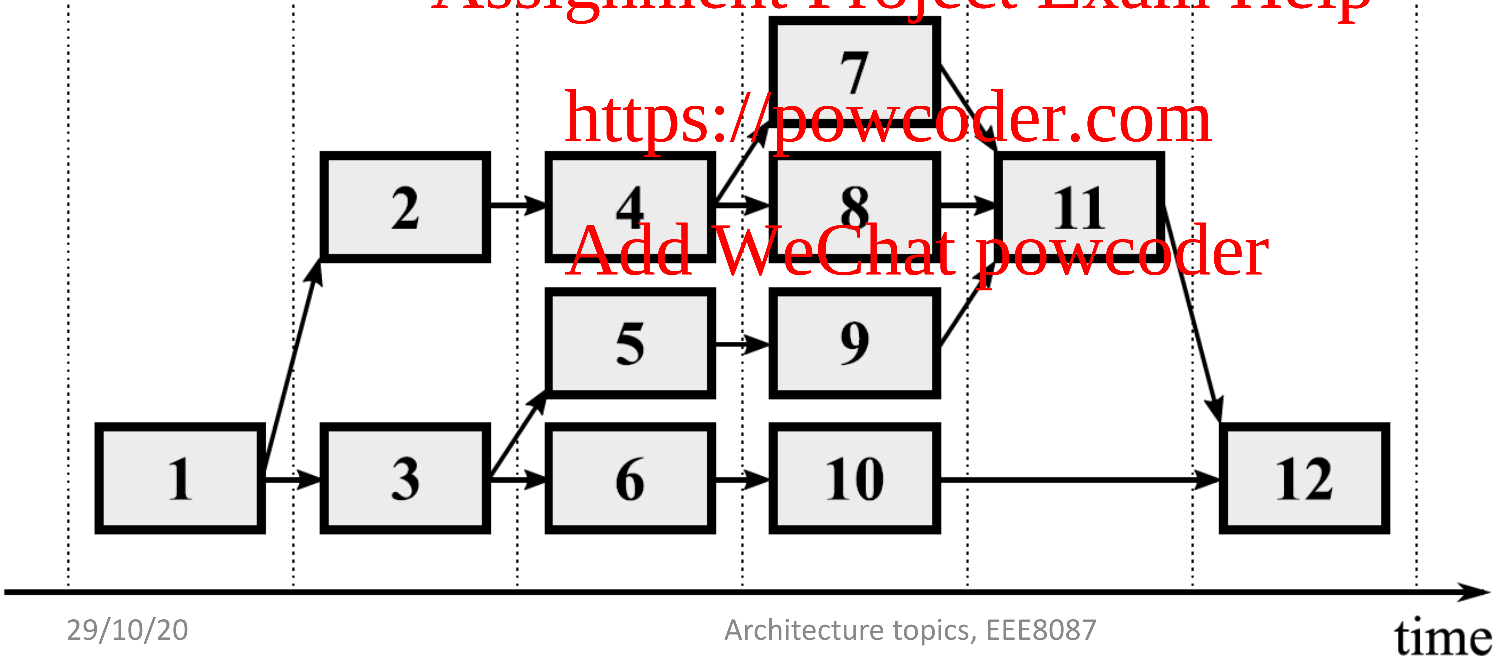
Tasks are represented by the nodes (vertices) – circles here, could be rectangles or other shapes

Task dependencies are represented by the directed arcs (edges) – an arc connects two tasks, when the first completes the second may start

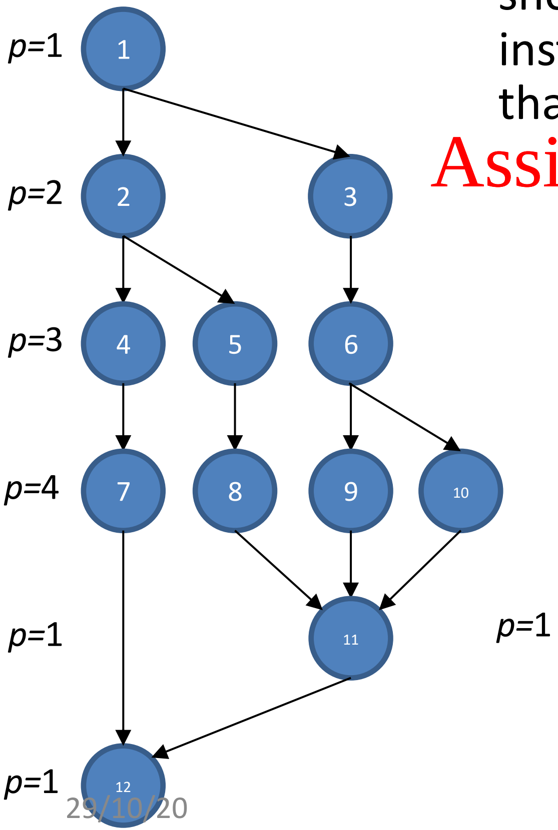
The general direction of all arcs indicates the direction of time progression - from top to bottom in this case

This type of graph is known as a ***dag*** – directed acyclic graph – a task graph does not have loops, must ***unfold*** all loops to generate a task graph

- Parallelism
  - Task graphs do not have to have circles or go top-down



- Parallelism
  - Each step of a task graph shows the workload's instantaneous parallelism in that step

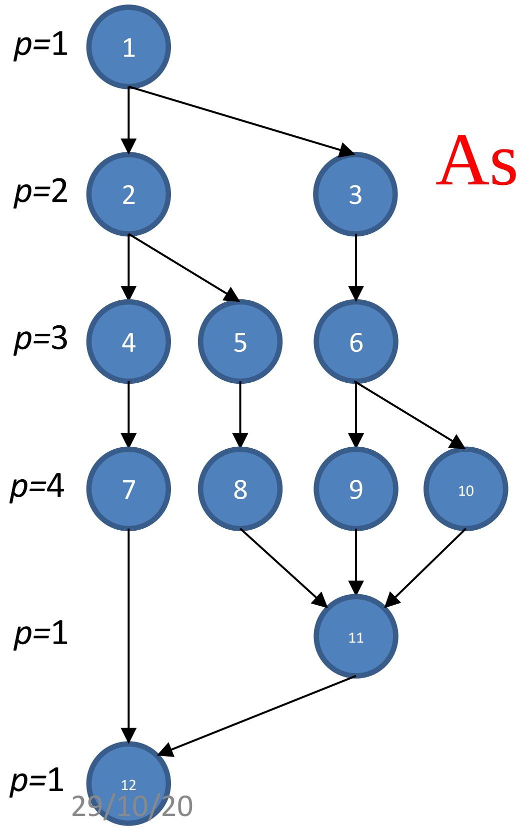


Intuitively, Instantaneous parallelism is how many independent threads a workload has at a time step, which measures how parallelizable the workload is at that step.

For instance, you don't want to give a workload 4 cores if its parallelism is only 2, in some time step, because you'd be wasting 2 cores. On the other hand, if you only have a single core available in that time step, the parallelism of 2 cannot be exploited

We normally assume that a task is an indivisible thread – atomic element of a workload. We also assume that each task takes the same time to run on a core (for ease of reasoning, not necessary).

- Parallelism
  - The maths



Parallelism is the speedup of a workload if you run it on an infinite number of cores, compared to running it on one core:

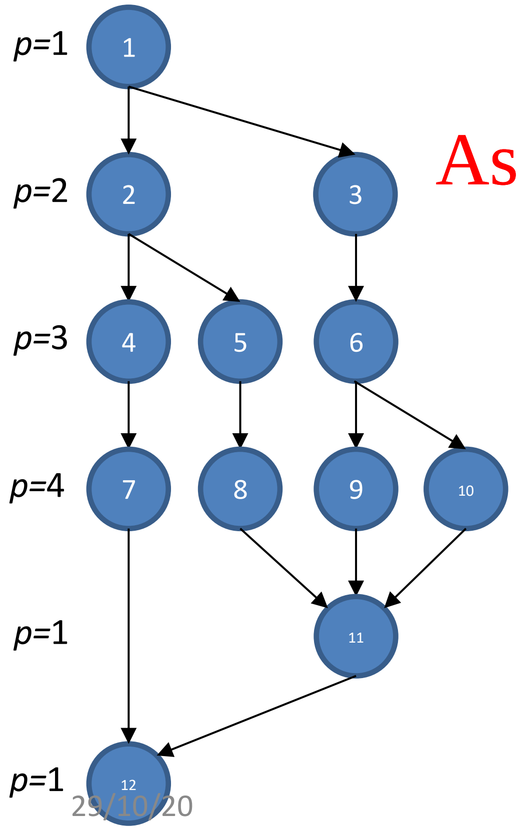
$$p = \frac{T_1}{T_\infty}$$

It therefore is the inherent parallelizability of software, regardless of hardware. Because extra cores will be wasted, we also have

$$p = \frac{T_1}{T_p}$$

For instance, at time step 3 on the left, if we give the workload an infinite number of cores, we'd achieve a speedup of 3, which can also be obtained by giving it 3 cores.

- Parallelism
  - The maths



The overall (or average) parallelism of a workload can be derived from averaging all the instantaneous parallelisms across all of its steps, but there is a much easier way.

The 'work' of a workload is the sum of all of its tasks – this assumes that each task takes exactly the same amount of time to run for convenience – hence

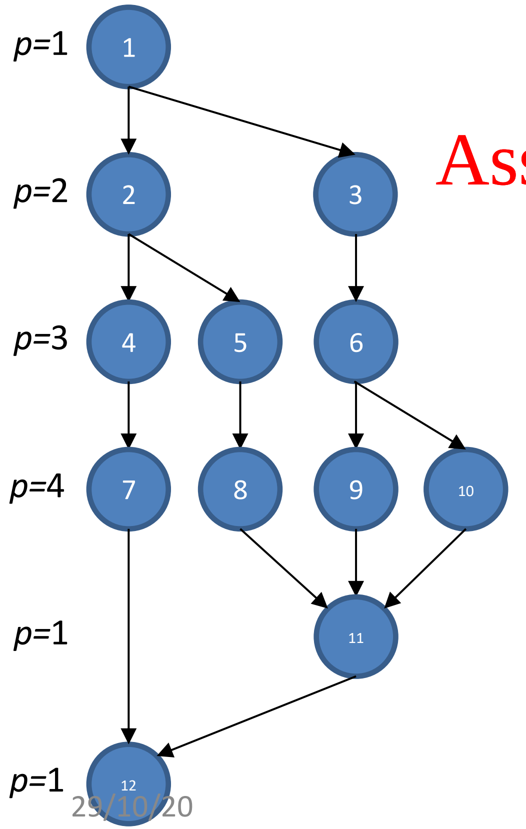
$$work = T_1 = 12$$

The 'span' of a workload is the number of steps in the task graph of a workload – again assuming equal task size in terms of time for convenience – hence

$$span = T_\infty = 6$$

Hence dividing work with span gets us the overall parallelism of a workload. 2 for this example.

- Influence of hardware



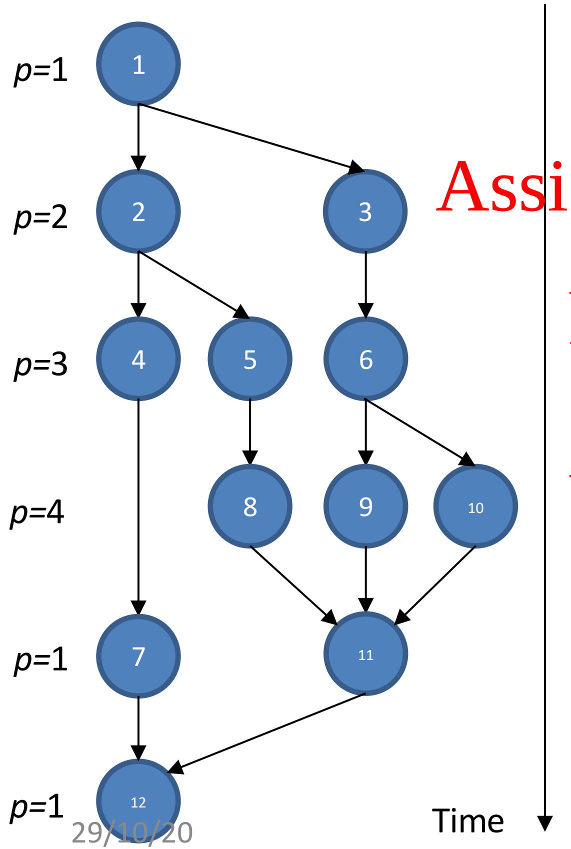
You may not have enough hardware to take full advantage of the parallelism of a software. For instance, you need 4 cores in order to fully exploit $p_{max} = 4$ in the example on the left, but what if you only have 3 cores?

With 3 cores, step 4 cannot be fully parallelized as in the task graph.

Usually you need to modify the task graph to reflect the hardware reality and scheduling constraints.

Scheduling refers to task-to-core mapping. This is usually done in system software, without the knowledge of the user, but it can also be intentionally controlled in some operating systems.

- Influence of hardware



Task graph modification to fit workload onto 3 cores. The best (why?) option is to delay task 7 until the next step. This does not modify the logical relations within the task graph – causality is fully preserved.

Now with $p_{max} = 3$ the workload can be executed on 3 cores.

Interestingly, neither work nor span changes from this modification. The workload still has the same overall parallelism 2!

In other words, with 3 cores you can achieve the same speedup with this workload as with 4 cores, with the right task graph modification. The relative speedup obtained by moving from 3 to 4 cores is $S(4)/S(3) = 1$.

Homework: what happens if your system only has 2 cores?

Software parallelizability – an alternative view

- Amdahl's law
  - For studying in the abstract, i.e. what if you do not know the structure of the software and its inner workings, and a task graph is not available?
  - Gene Amdahl, in the 1960s, proposed an abstract view of a software workload as consisting of a fully parallelizable part and a fully non-parallelizable part. In other words, a piece of software is regarded as one part with $p = 1$ and another part with $p = \infty$.
  - The time taken by executing such a parallel workload on a single core can be divided into two parts, that taken by the sequential part and that taken by the parallel part: $T_w = T_s + T_p$
  - Executing such a workload on *n* cores means that in the total time T_n, T_s stays the same, as you use one of the cores to execute the sequential part leaving the other *n*-1 cores idle. However the parallel part will take time T_p/n as you get to use all *n* cores because $p = \infty$ and you get to fill all the cores with no idle.
  - The speedup achieved by moving from one core to *n* cores is then

$$S(n) = \frac{T_1}{T_n} = \frac{T_s + T_p}{T_s + \frac{T_p}{n}}$$

Amdahl’s law

- An important parameter for Amdahl’s Law is what’s known as the ‘parallel fraction’.
- Fraction of the workload that is parallel, defined as

$$f = \frac{T_p}{T_w}$$

The fraction of time taken by the parallel part of the workload

- Amdahl’s Law says that the speedup, with the parallel fraction, is then

$$S(n) = \frac{T_1}{T_n} = \frac{T_s + T_p}{T_s + \frac{T_p}{n}} = \frac{1}{(1-f) + \frac{f}{n}}$$

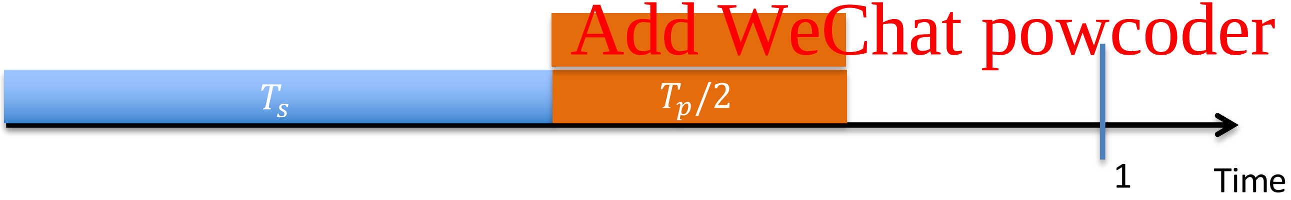
- The parallel part achieves linear speedup with the number of cores and the sequential part stays sequential.
- Typical scheduling decision: if you have a workload with small f and another one with large f, give the latter more cores.

$$S(n) = \frac{1}{(1-f) + \frac{f}{n}}$$

- **Amdahl's Law examples**
  - Fixed workload
  - 50% sequential part
  - On a single core takes 1 unit of time to complete

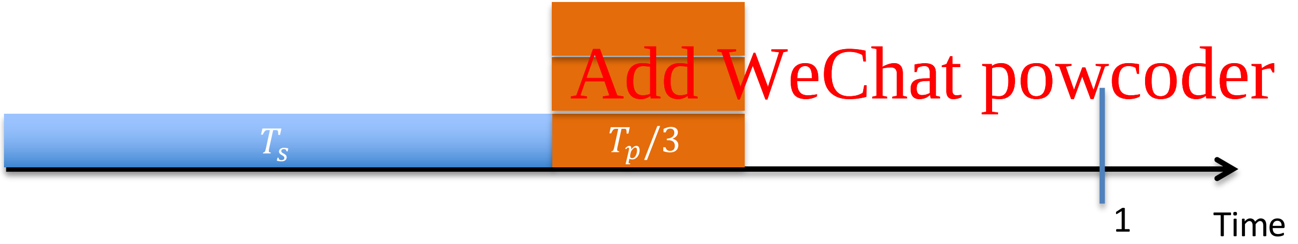
- **Amdahl's Law examples**
  - With two cores …
  - Parallelizable part is distributed between the two cores
  - Total time 0.75

$$S(2) = 1/0.75 = 1.333$$



- **Amdahl’s Law examples**
  - With three cores …

$$S(3) = \frac{1}{0.5 + \frac{0.5}{3}} = 1.5$$

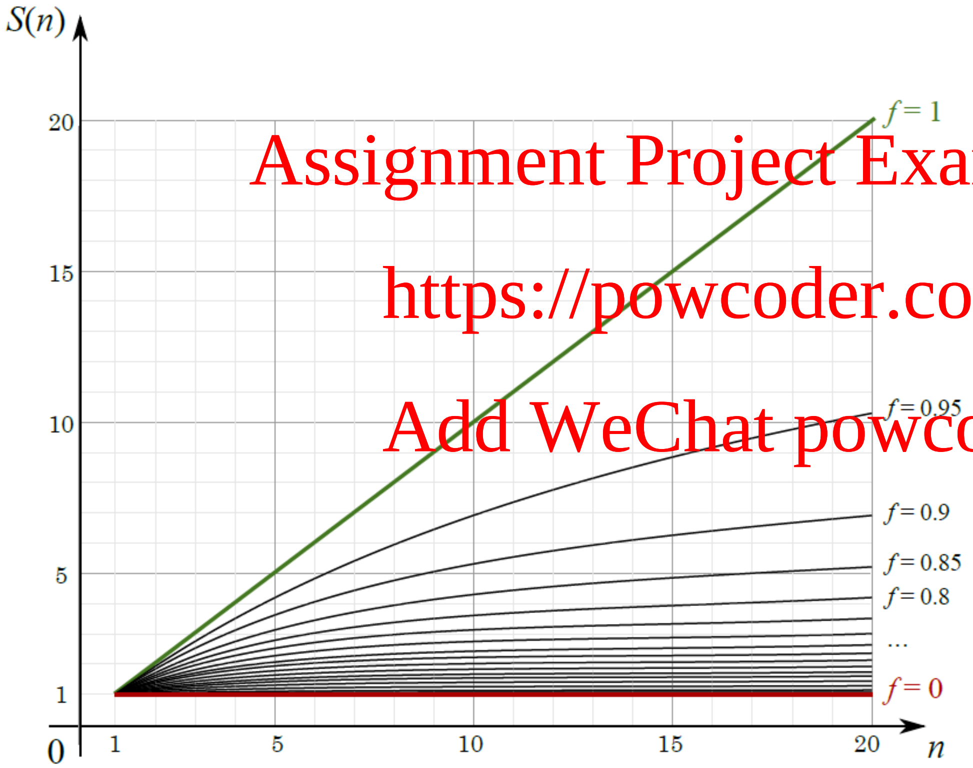


- **Amdahl's Law examples**
  - With an infinite number of cores …

$$S(\infty) = \frac{1}{1 - f + \frac{f}{\infty}} = \frac{1}{1-f} = 2$$

$$T_\infty = T_s$$

Amdahl’s law



$$S(n) = \frac{1}{(1-f) + \frac{f}{n}}$$

Amdahl’s Law vs task graphs

- Both methods can be used for modelling the behaviour of workloads with regard to the hardware on which they run
- Scheduling decisions can be made to optimize speedup, energy, or a combination of both based on these models
- Task graphs are usually derivable by programmers who write workloads – could be very complicated for large workloads
- Users usually do not have access to task graphs
- The parallel fraction *f* can be obtained through experiments
- Task graphs can also be extracted from experiments (running a workload and monitoring certain system sensors)
- Which one to use in what situation is the user’s decision to make
- Models can be unified – see

  http://async.org.uk/tech-reports/NCL-EEE-MICRO-TR-2018-211.pdf
- These all assume ideal hardware and software, with full scalability other than what’s limited by parallelism and parallel fraction