AMS 250: An Introduction to High Performance Computing

Parallel Performance Theory

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Applied Mathematics & Statistics

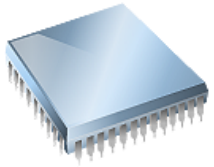
University of California, Santa Cruz

Outline

- Performance and Scalability
- Analytical performance measures
- Amdahl's law and Gustafson's law
- DAG Model of Computation

What is Performance?

- In computing, performance is defined by 2 factors
  - Computational requirements
  - Computing resources
- Computational problems translate to requirements
- Computing resources:



Hardware

Time

Energy

… and ultimately

Money

$$Performance \sim \frac{1}{Resources\ for\ solution}$$

Why do we care about Performance?

- Performance itself is a measure of how well the computational requirements can be satisfied
- We evaluate performance to understand the relationships between requirements and resources
  - Decide how to change “solutions” to target objectives
- Performance measures reflect decisions about how and how well “solutions” are able to satisfy the computational requirements

> *“The most constant difficulty in contriving the engine has arisen from the desire to **reduce the time** in which the calculations were executed to the shortest which is possible.”*
>
> *Charles Babbage, 1791 – 1871*

What is Parallel Performance?

- Here we are concerned with performance issues when using a parallel computing environment
  - Performance with respect to parallel computation
- Performance is the *raison d'être* for parallelism
  - Parallel performance versus sequential performance
  - If the "performance" is not better, parallelism is not necessary
- *Parallel processing* includes techniques and technologies necessary to compute in parallel
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …
- Parallelism must deliver performance
  - How? How well?

Performance Expectation (Loss)

- If each processor is rated at *k* MFLOPS and there are *p* processors, should we see *k*p* MFLOPS performance?
- If it takes 100 seconds on 1 processor, shouldn't it take 10 seconds on 10 processors?
- Several causes affect performance
  - Each must be understood separately
  - But they interact with each other in complex ways
    - Solution to one problem may create another
    - One problem may mask another
- Scaling (system & problem size) can change conditions

Embarrassingly Parallel Computations

- An embarrassingly parallel computation is one that can be obviously divided into completely independent parts that can be executed simultaneously
  - In a truly embarrassingly parallel computation there is no interaction between separate processes
  - In a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way
- Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms
  - If it takes *T* time sequentially, there is the potential to achieve *T/P* time running in parallel with *P* processors
  - What would cause this not to be the case always?

Performance and Scalability

- Evaluation
  - *Sequential* runtime (T_{seq}) is a function of
    - problem size and architecture
  - *Parallel* runtime (T_{par}) is a function of
    - problem size and parallel architecture
    - # of processing elements (processors, cores, etc.) used in the execution
  - Parallel performance affected by
    - algorithm + architecture
- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a *p* processor system
- S_p is the *speedup*

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

- E_p is the *efficiency*

$$E_p = \frac{S_p}{p}$$

- C_p is the *cost*

$$C_p = p \times T_p$$

- Parallel algorithm is *cost-optimal,* if *parallel time = sequential time*

  $C_p = T_1$ & $E_p = 100\%$

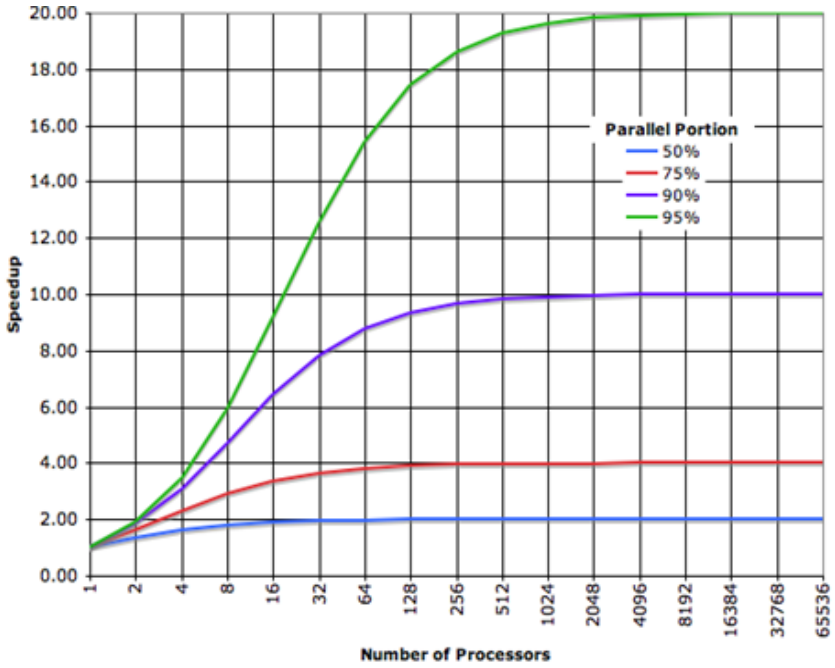
Amdahl's Law

- **Fixed** problem size
- Let f be the fraction of a program that is sequential, then $1-f$ is the fraction that can be parallelized

$$T_1 = f\,T_1 + (1-f)\,T_1$$
$$T_p = f\,T_1 + (1-f)\,T_1/p$$
$$S_p = T_1 / T_p$$
$$= T_1 / (fT_1 + (1-f)T_1/p))$$
$$= 1 / (f + (1-f)/p))$$

- As $p \rightarrow \infty$

$$S_p = 1 / f$$



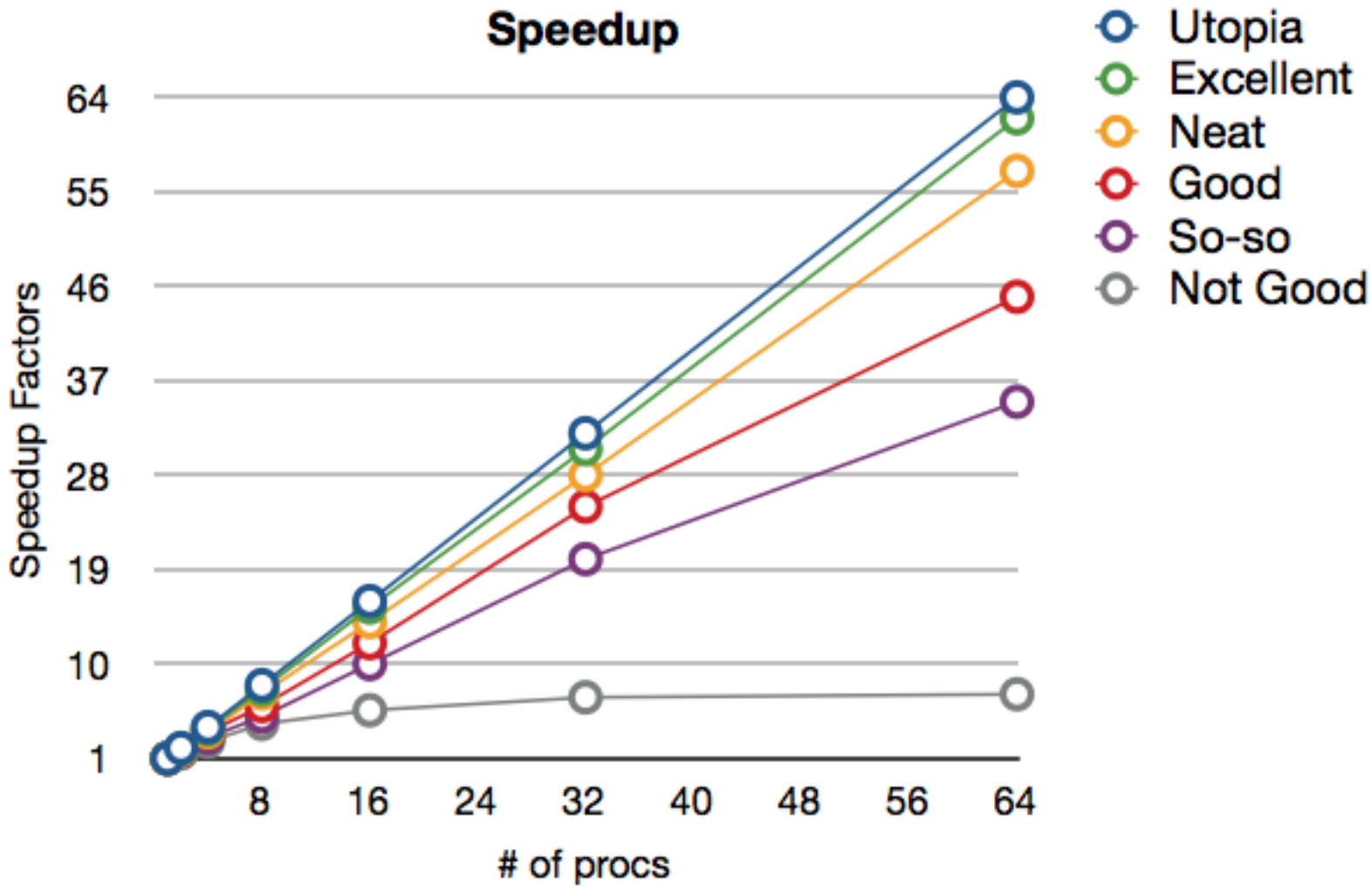
Amdahl's Law and Scalability

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- When does Amdahl's Law apply?
  - When the problem size is **fixed**
  - ***Strong scaling***
  - Speedup bound is determined by the degree of sequential execution time in the computation, not # of processors!

    $p \to \infty, S_p = S_\infty \to 1 / f$
  - Uhh, this is not good … Why?
  - Perfect efficiency is hard to achieve

http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

Sample Strong Scaling Plot



http://web.eecs.utk.edu/~huangj/hpc/hpc_intro.php

Gustafson's Law

- We often increase the size of problems to fully exploit the available computing power
- Let f be the fraction of a program that is sequential, then $f_{par}=1-f$ is the fraction that can be parallelized. The workload on one processor is:

  $$W_1 = f\,W_1 + (1-f)\,W_1$$

- Assuming parallel time is kept constant, the total workload on p processors is:

  $$W_p = f\,W_1 + (1-f)\,W_1 * p$$

- Scaled speedup:

  $$S_p = W_p \,/\, W_1$$
  $$= f + (1-f)*p$$
  $$= 1-f_{par} + f_{par}*p = 1 + (p-1)\,f_{par}$$

- Efficiency: $E_p = S_p \,/\, p = (1-f_{par})/p + f_{par}$

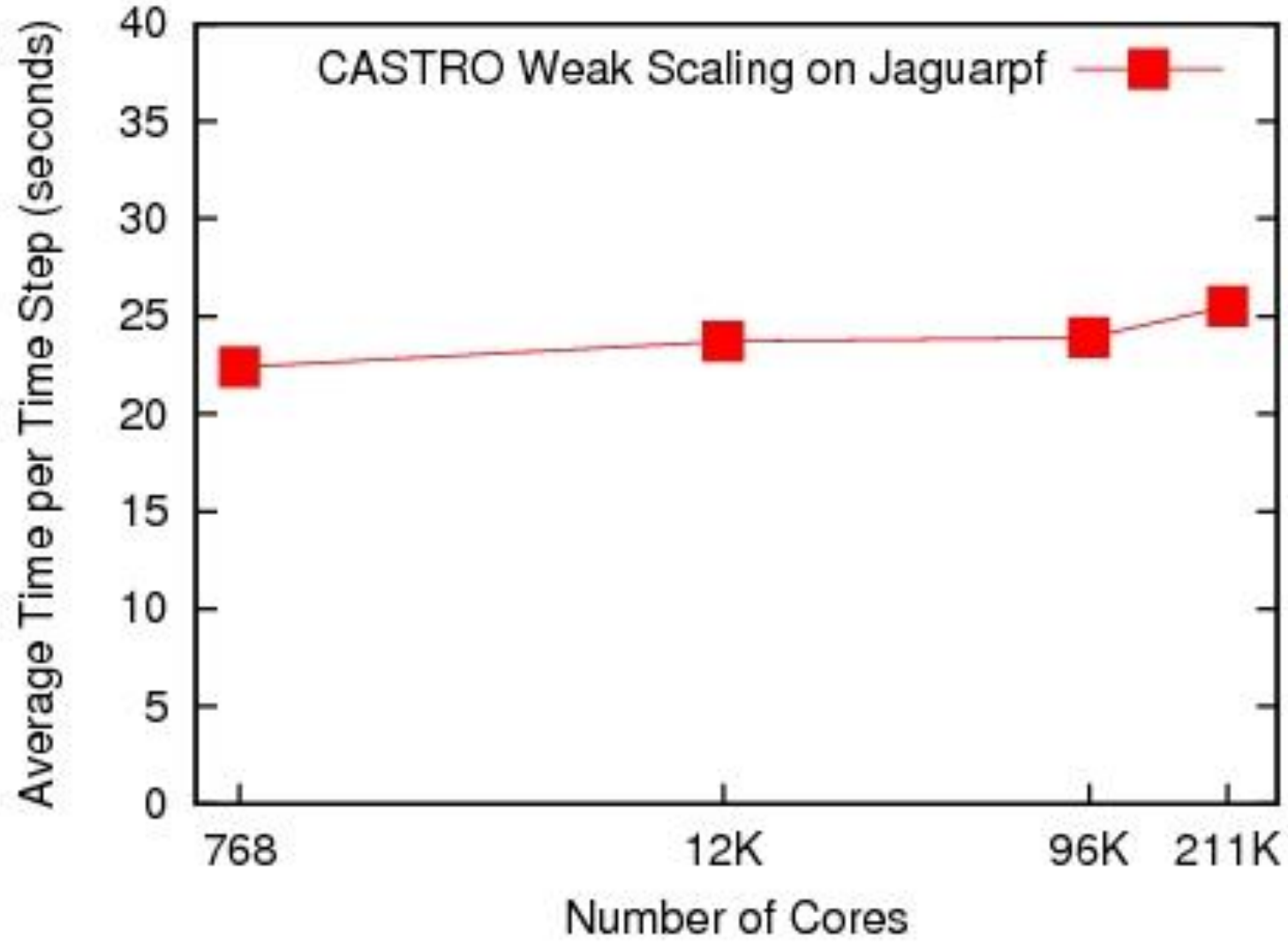
Gustafson's Law and Scalability

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
- When does Gustafson's Law apply?
  - When the problem size can increase as the number of processors increases
  - *Weak scaling*
  - Speedup function includes the number of processors!

    $S_p = 1 + (p-1)f_{par}$
  - Can maintain or increase parallel efficiency as the problem scales

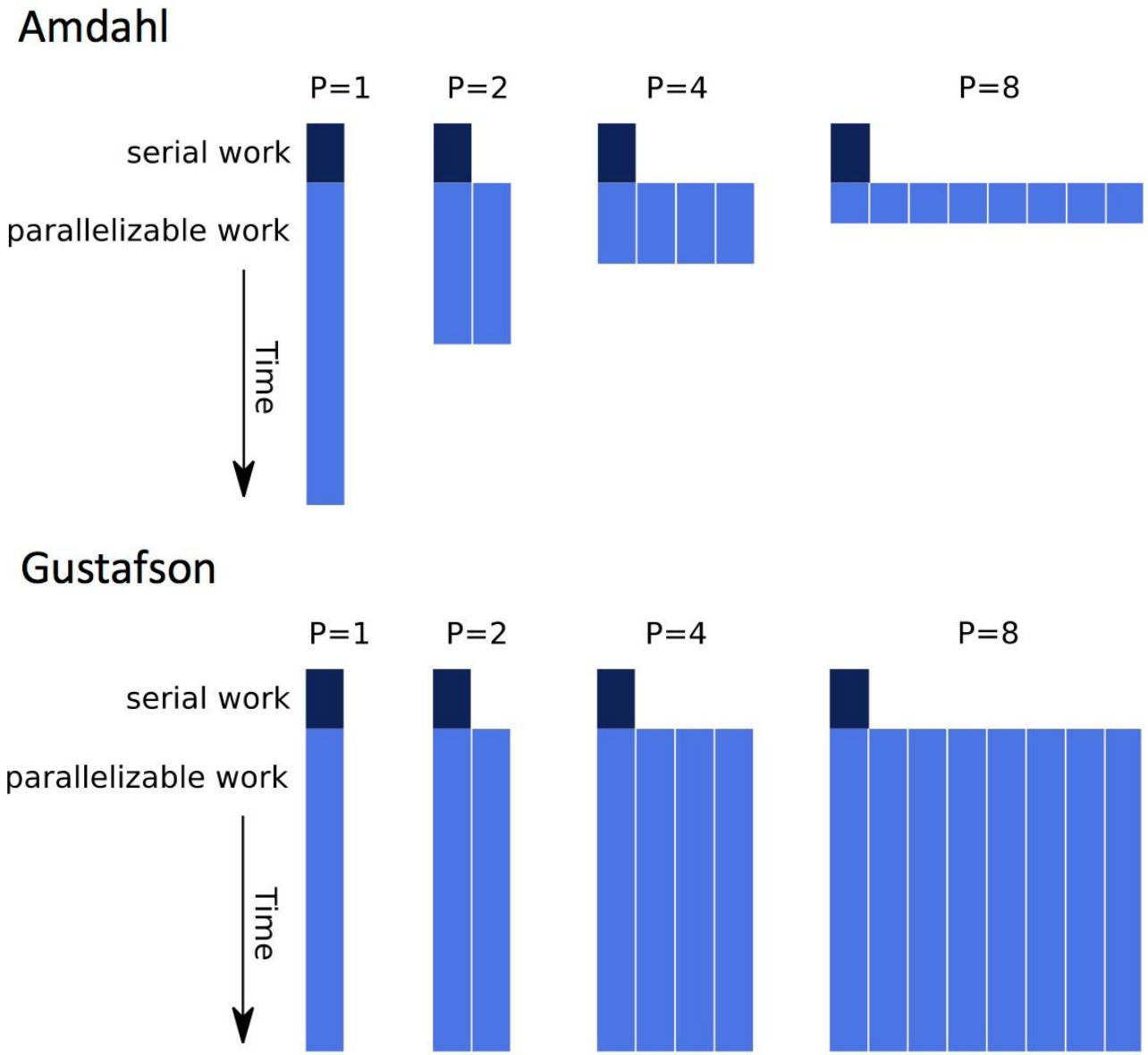
http://www.johngustafson.net/pubs/pub13/amdahl.htm

Sample Weak Scaling Plot



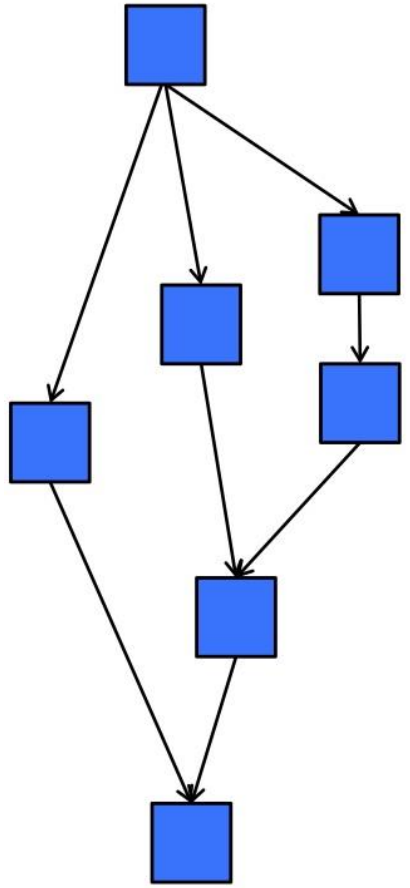
https://ccse.lbl.gov/BoxLib/

Amdahl versus Gustafson



DAG Model of Computation

- Think of a program as a directed acyclic graph (DAG) of tasks
  - A task can not execute until all the inputs to the tasks are available
  - These come from outputs of earlier executing tasks
  - DAG shows explicitly the task dependencies
- Think of the hardware as consisting of workers (processors)
- Consider a *greedy* scheduler of the DAG tasks to workers
  - No worker is idle while there are tasks still to execute

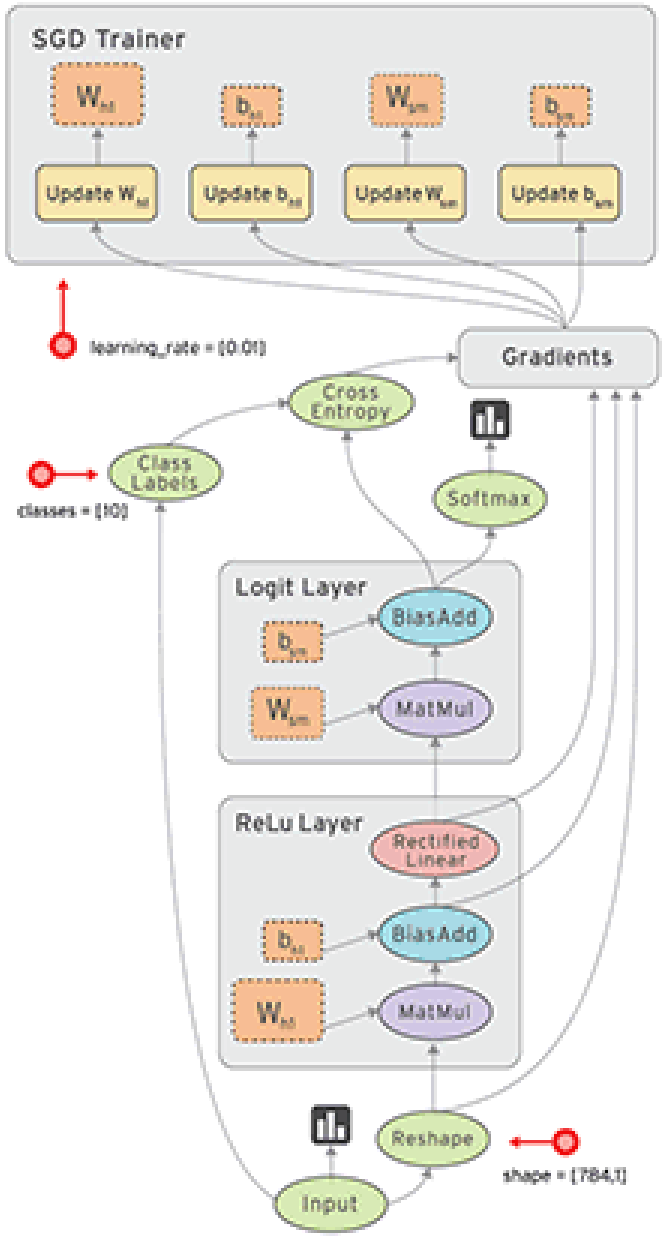


Example: TensorFlow Dataflow Graph

https://www.tensorflow.org/programmers_guide/graphs

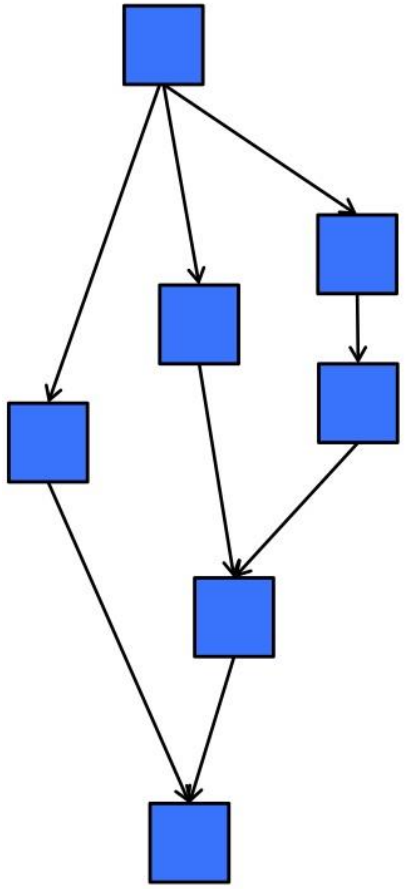
Dataflow Graph has several advantages that TensorFlow leverages:

- **Parallelism**: by using explicit edges to represent dependencies between operations, it is easy for the system to identify operations that can execute in parallel.
- **Distributed execution**: by using explicit edges to represent the values that flow between operations, it is possible for TensorFlow to partition your program across multiple devices (CPUs, GPUs, and TPUs) attached to different machines. TensorFlow inserts the necessary communication and coordination between devices.
- Etc.



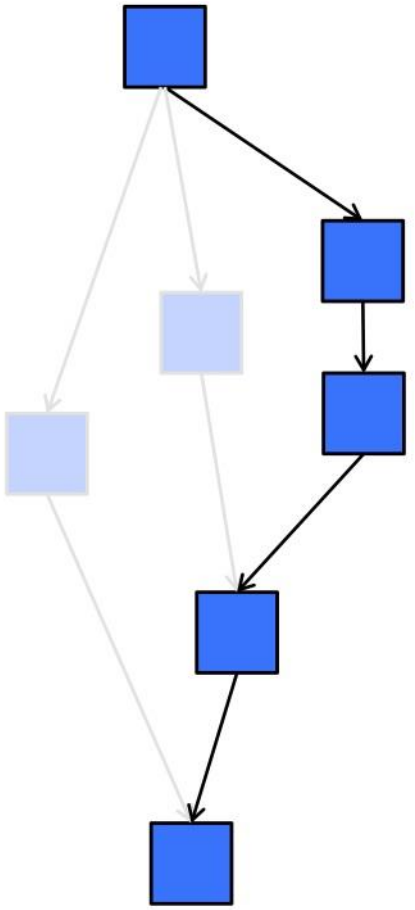
Work-Span Model

- T_P = time to run with P workers
- T_1 = *work*
  - Time for serial execution
    - execution of all tasks by 1 worker
  - Sum of all work
- T_∞ = *span*
  - Time along the *critical path*
- Critical path
  - Sequence of task execution (path) through DAG that takes the longest time to execute
  - Assumes an infinite # workers available



Work-Span Example

- Let each task take 1 unit of time
- DAG at the right has 7 tasks
- $T_1 = 7$
  - All tasks have to be executed
  - Tasks are executed in a serial order
  - Can they execute in any order?
- $T_\infty = 5$
  - Time along the *critical path*
  - In this case, it is the longest path length of any task order that maintains necessary dependencies



Lower/Upper Bound on Greedy Scheduling

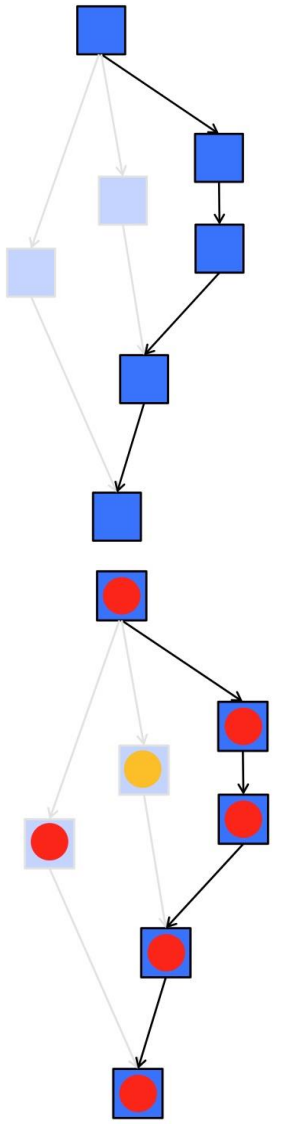
- Suppose we only have P workers
- We can write a work-span formula to derive a lower bound on T_p

  $Max(T_1 / p , T_\infty) \leq T_P$

- T_∞ is the best possible execution time
- **Brent's Lemma** derives an upper bound
  - Capture the additional cost executing the other tasks not on the critical path
  - Assume we can do so without overhead

  $T_p \leq (T_1 - T_\infty) / p + T_\infty$

$$\boldsymbol{Max(T_1 / p , T_\infty) \leq T_p \leq (T_1 - T_\infty) / p + T_\infty}$$



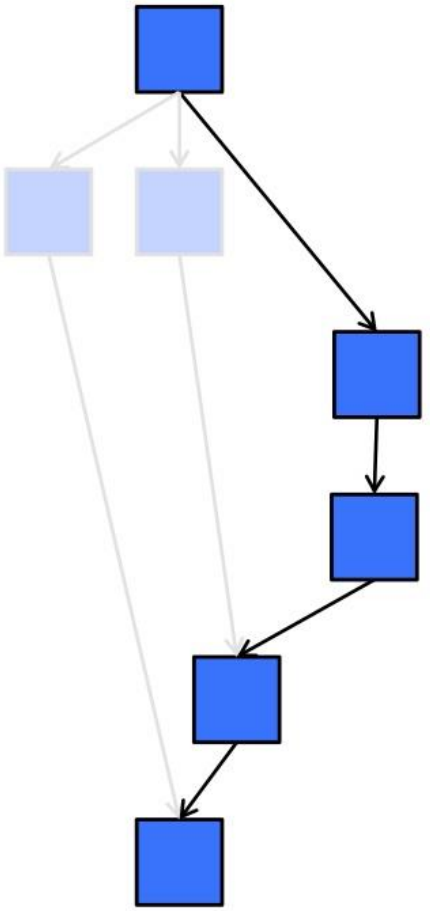
https://maths-people.anu.edu.au/~brent/pub/pub022.html

Consider Brent's Lemma for 2 Processors

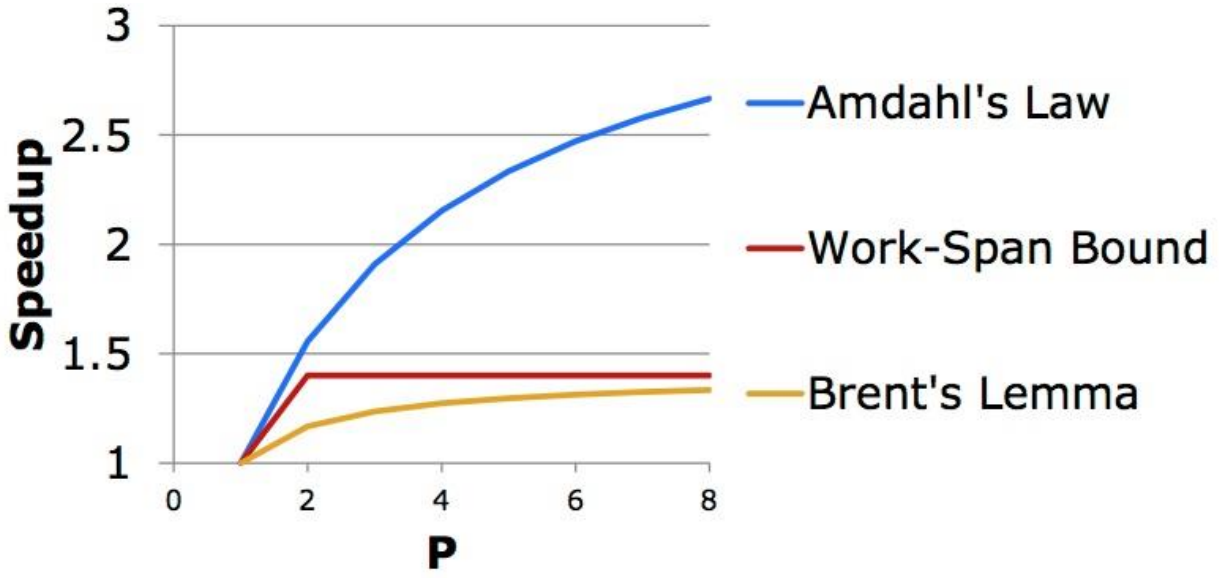
- $T_1 = 7$
- $T_\infty = 5$
- $T_2 \leq (T_1 - T_\infty) / P + T_\infty$

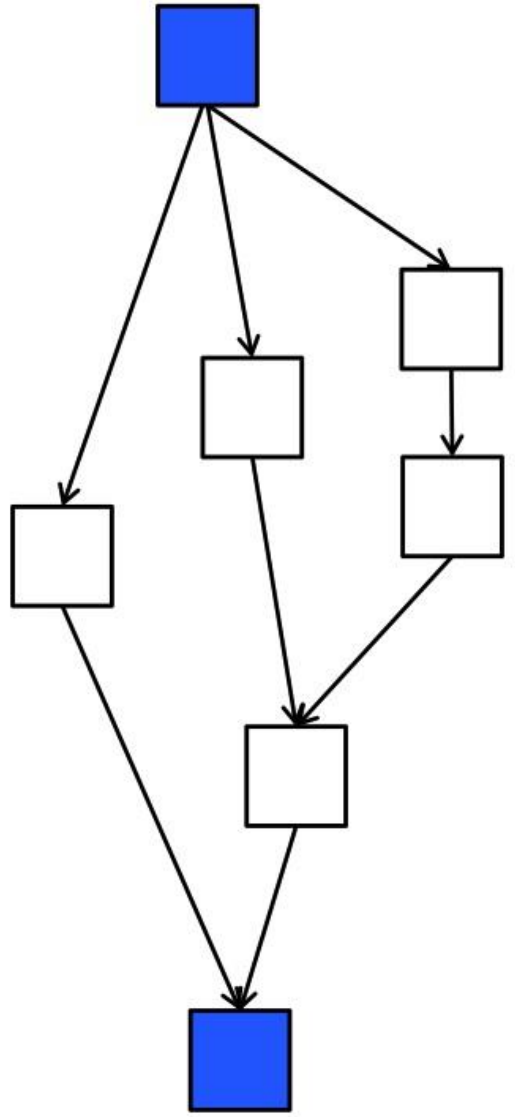
  $\leq (7 - 5) / 2 + 5$

  ≤ 6



Amdahl was an optimist!





f = 2/7, $T_1 = 7$, $T_\infty = 5$

Amdahl's Law: $S_p = 1 / (f + (1-f)/p)) = 7p/(5+2p)$

$Max(T_1 / p , T_\infty) \leq T_p \leq (T_1 - T_\infty) / p + T_\infty$

$\Rightarrow Max(7/p, 5) \leq T_p \leq 2/p + 5$

$\Rightarrow 7p/(2+5p) \leq S_p \leq 7/ Max(7/p, 5)$

Estimating Running Time

- Scalability requires that T_∞ be dominated by T_1

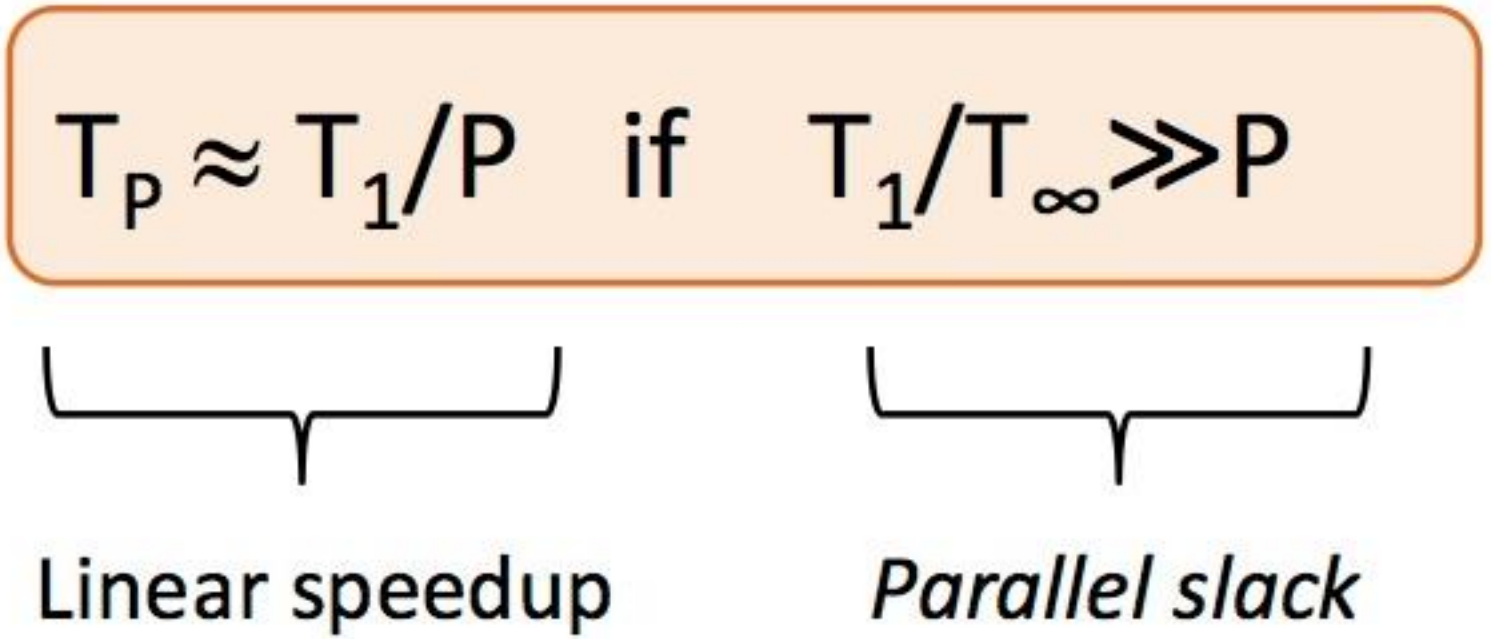
$$Max(T_1 / p , T_\infty) \leq T_p \leq (T_1 - T_\infty) / p + T_\infty$$

$$\Rightarrow T_p \approx T_1 / p + T_\infty \text{ if } T_\infty << T_1$$

- Increasing work (T_1) hurts parallel execution proportionately
- The span (T_∞) impacts scalability, even for finite p

Parallel Slack

- Sufficient parallelism implies linear speedup



- Is it possible to have superlinear speedup ($S_p > p$)?