Toward HPC

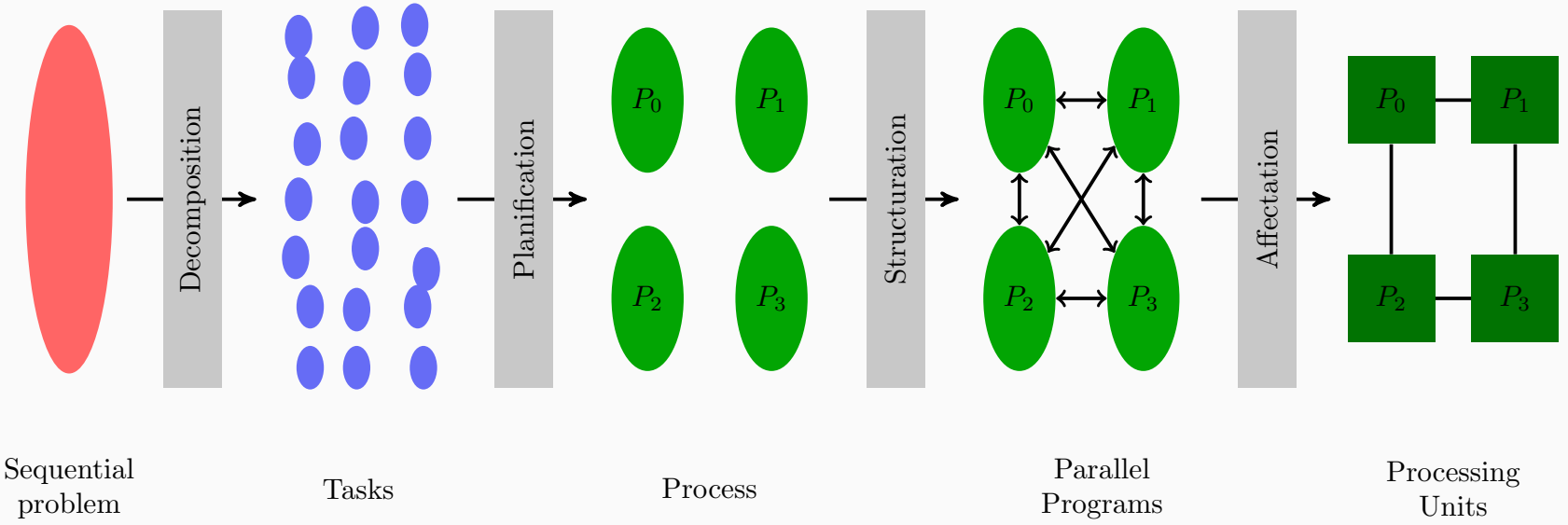
Chapter 5 – Patterns

M1 – MSIAM

April 4, 2017

So far . . .

Foster design



Scalability

Computing performance

Performance is defined by 2 factors

- Computational requirements (what needs to be done)
- Computing resources (what it costs to do it)

Computational problems translate to requirements

Factors: hardware, time, energy, money

Why is it important

- 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 $\Longrightarrow$ Decide how to change methodology to target objectives
- Performance measures reflect decisions about how and how well approaches are able to satisfy the computational requirements

Some definition of parallelism

Performance issues when using a parallel computing environment: Performance with respect to parallel computation

- Performance is why we do 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?

What can we expect?

If each processor is rated at $k - GFlops$ and there are p processors, should we see $kpGFlops$ 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 or one problem may mask another

Scaling (system, problem size) can change conditions

Need to understand performance space

Analytical measure

Embarrassingly Parallelism

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?

Scalability

- A program can scale up to use many processors: What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation: if double the number of processors, what to expect? Is scalability linear?
- Use parallel efficiency measure: is efficiency retained as problem size increases?
- Apply performance metrics

A measure of performance

Evaluation

- Sequential runtime T_{seq} is a function of problem size and architecture
- Parallel runtime T_{par} is a function of problem size, parallel architecture and number of processors used in the execution
- Parallel performance affected by algorithm and architecture

Definition (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

$$S(p) = T_1/T_p$$

- $E(p)$ is the efficiency

$$Efficiency = S_p/p$$

- $Cost(p)$ is the cost

$$Cost = pT_p$$

- Parallel algorithm is cost-optimal
  Parallel time = sequential time ($C(p) = T_1, E(p) = 100\%$)

Laws on performances

Amdahl's law

Let f be the fraction of a program that is sequential. $1 - f$ is the fraction that can be parallelized

Let T_1 be the execution time on 1 processor

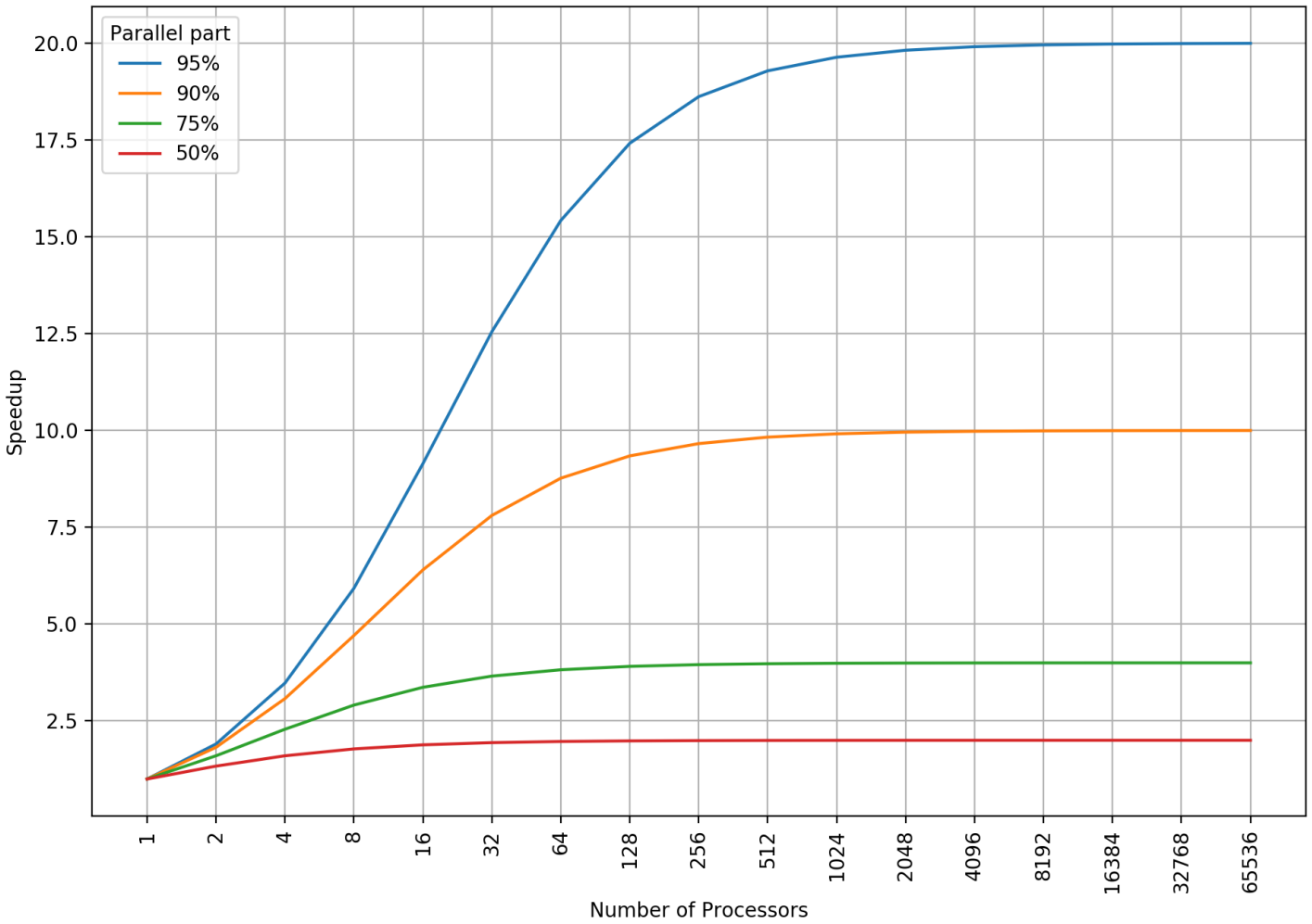
Let T_p be the execution time on p processors

S_p is the speedup

$$\begin{aligned} S_p &= T_1/T_p \\ &= T_1/(fT_1 + (1-f)T_1/p) \\ &= 1/(f + (1-f)/p) \end{aligned}$$

As $p \to \infty$, $S_p = 1/f$

Amdahl's law


Parallel part
95%
90%
75%
50%
Speedup
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Gustafson-Barsis' Law

Definition (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 ($p \to \infty$, $S_p = S_\infty \to 1/f$)
- Speedup bound is determined by the degree of sequential execution time in the computation, not number of processors!!!
- Perfect efficiency is hard to achieve

Gustafson-Barsis' Law

- Often interested in larger problems when scaling
  - How big of a problem can be run
  - Constrain problem size by parallel time
- Assume parallel time is kept constant

$$T(p) = C = (f + (1 - f)) * C$$

- What is the execution time on one processor? Let $C = 1$, then $T(s) = f_{seq} + p(1-f_{seq}) = 1 + (p-1)f_{par}$
- What is the speedup in this case?

$$S(p) = T_s/T_p = T_s/1 = f_{seq} + p(1-f_{seq}) = 1 + (p-1)f_{par}$$

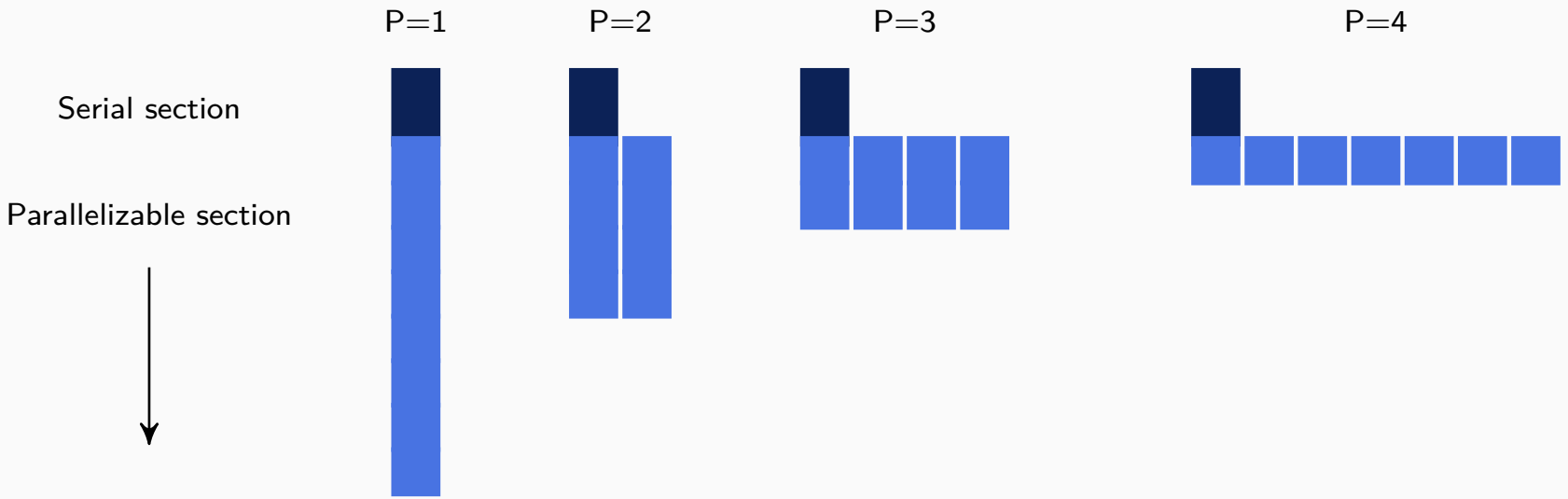
Definition (Scalability)

Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

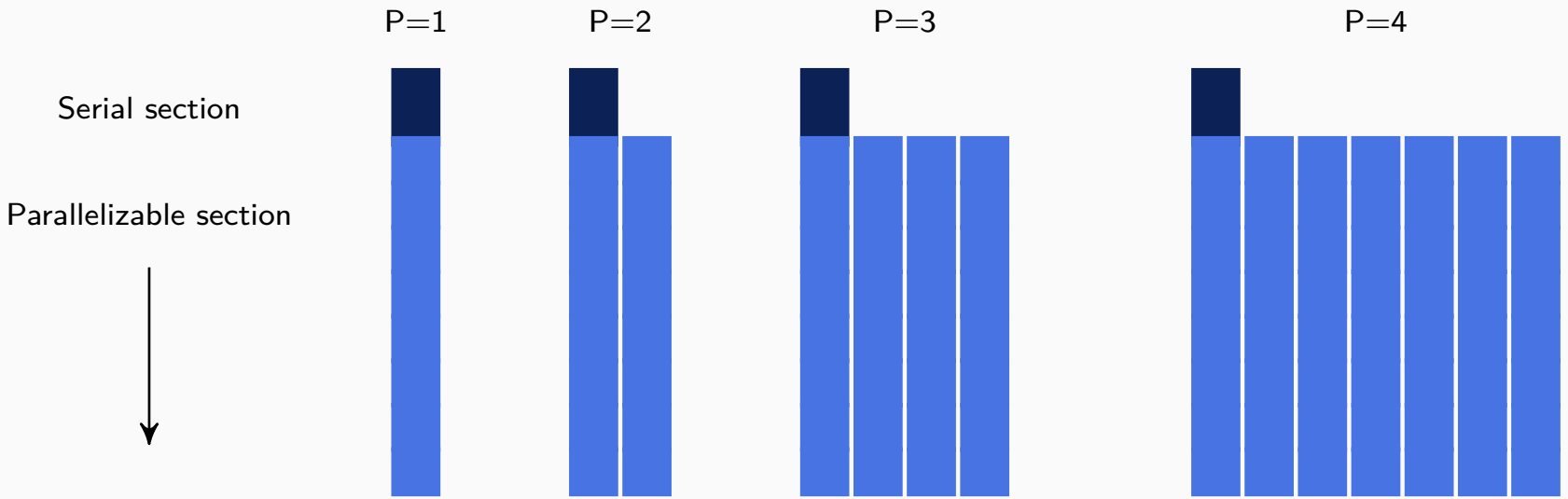
When does Gustafson's Law apply:

- When the problem size can increase as the number of processors increases
- Weak scaling ($S_p = 1 + (p-1)f_{par}$)
- Speedup function includes the number of processors!!!
- Can maintain or increase parallel efficiency as the problem scales

Amdahl's law vs Gustafson



Amdahl's law vs Gustafson



DAG Model of Computation

A program seen as directed acyclic graph (DAG) of tasks

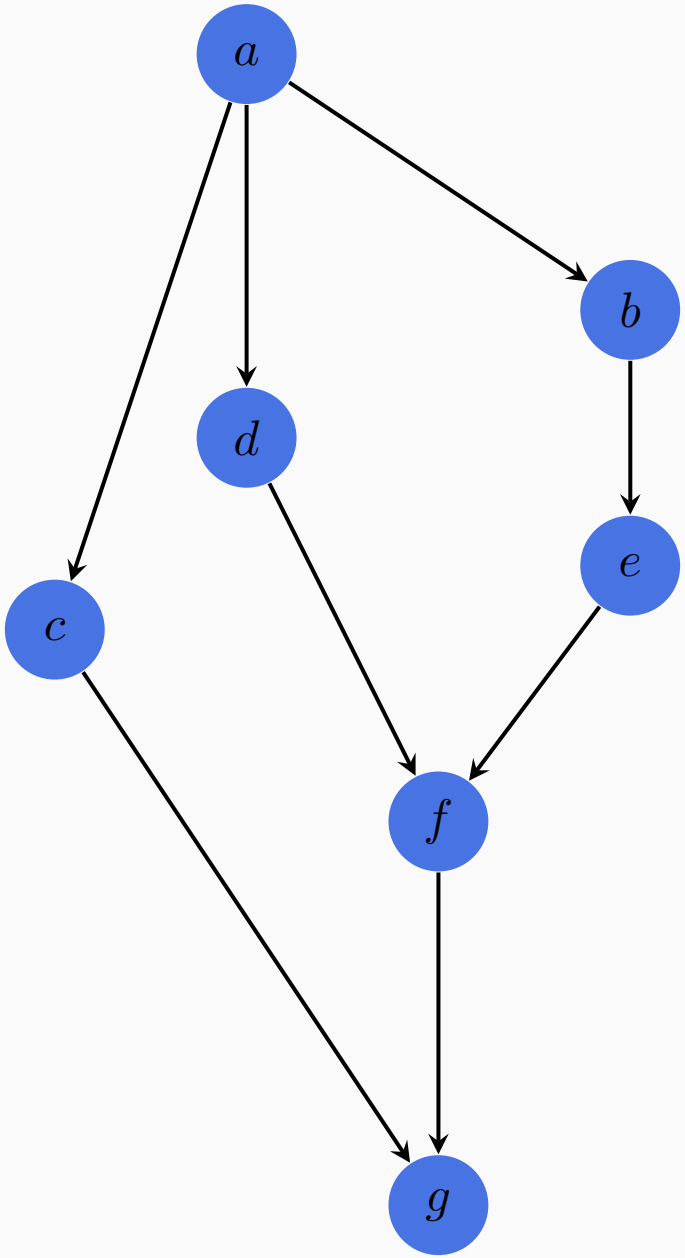
- A task can not execute until all the inputs to the tasks are available
- These come from outputs of earlier executing tasks
- DAG shows explicitly the task dependencies

Hardware consists of workers (processing units)

We consider a greedy scheduler of the DAG tasks to workers

$\Longrightarrow$ No worker is idle while there are tasks still to execute

Example of DAG



Execution time

T_P is time to execute with P workers

T_1 is time for serial execution. It is the sum of all tasks

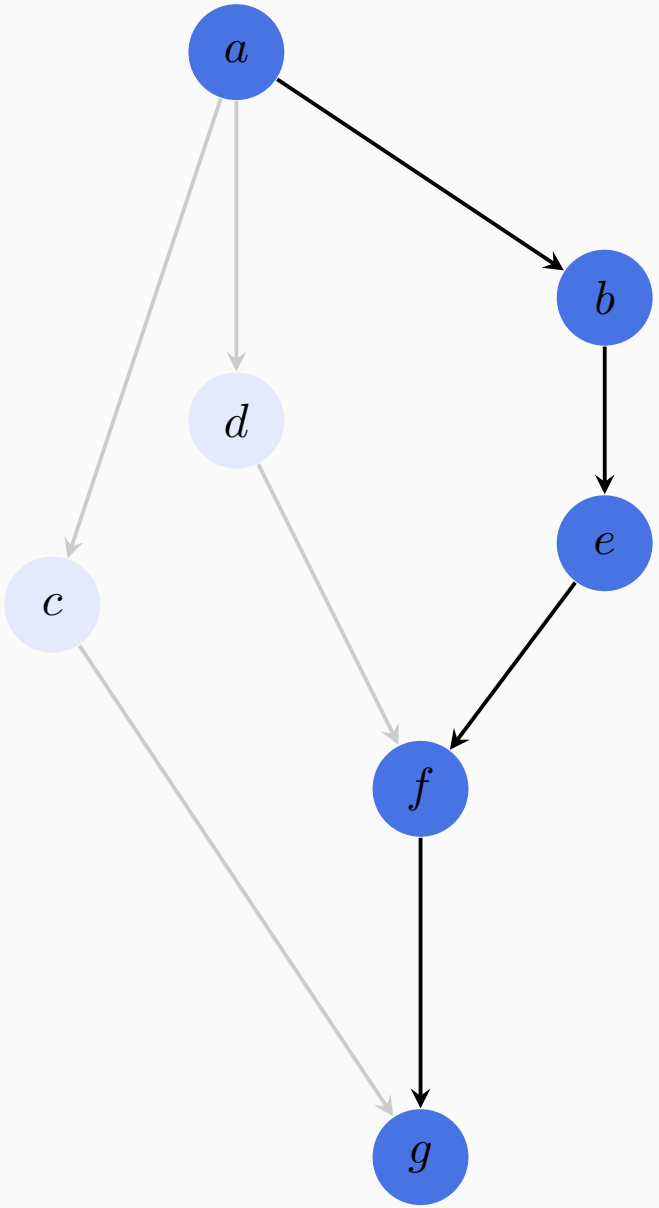
T_∞ is time along the critical path

- Sequence of task execution (path) through DAG that takes the longest time to execute
- Assumes an infinite number workers available

Example

Let each task take 1 unit of time

- $T_1 = 7$: All tasks have to be executed and in serial 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



Bounds

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

Theorem (Brent)

Capture the additional cost executing the other tasks not on the critical path.

Assume can do so without overhead. Then

$$T_P \leq (T_1 - T_\infty)/P + T_\infty$$

Application

- $T_1 = 7$
- $T_\infty = 5$
- For $P = 2$

$$\begin{aligned} T_2 &\leq (T_1 - T_\infty)/P + T_\infty \\ &\leq (7 - 5)/2 + T_\infty \\ &\leq 6 \end{aligned}$$

Estimation of running time

- Scalability requires that T_∞ be dominated by T_1

$$T_P \simeq T_1/P + T_\infty \text{ if } T_\infty \ll T_1$$

- Increasing work hurts parallel execution proportionately
- The span impacts scalability, even for finite P

Available parallelism

Sufficient parallelism implies linear speedup

$$T_p \sim T_1/P \text{ if } T_1/T_\infty \gg P$$

Applications

Conclusion

Neural Networks