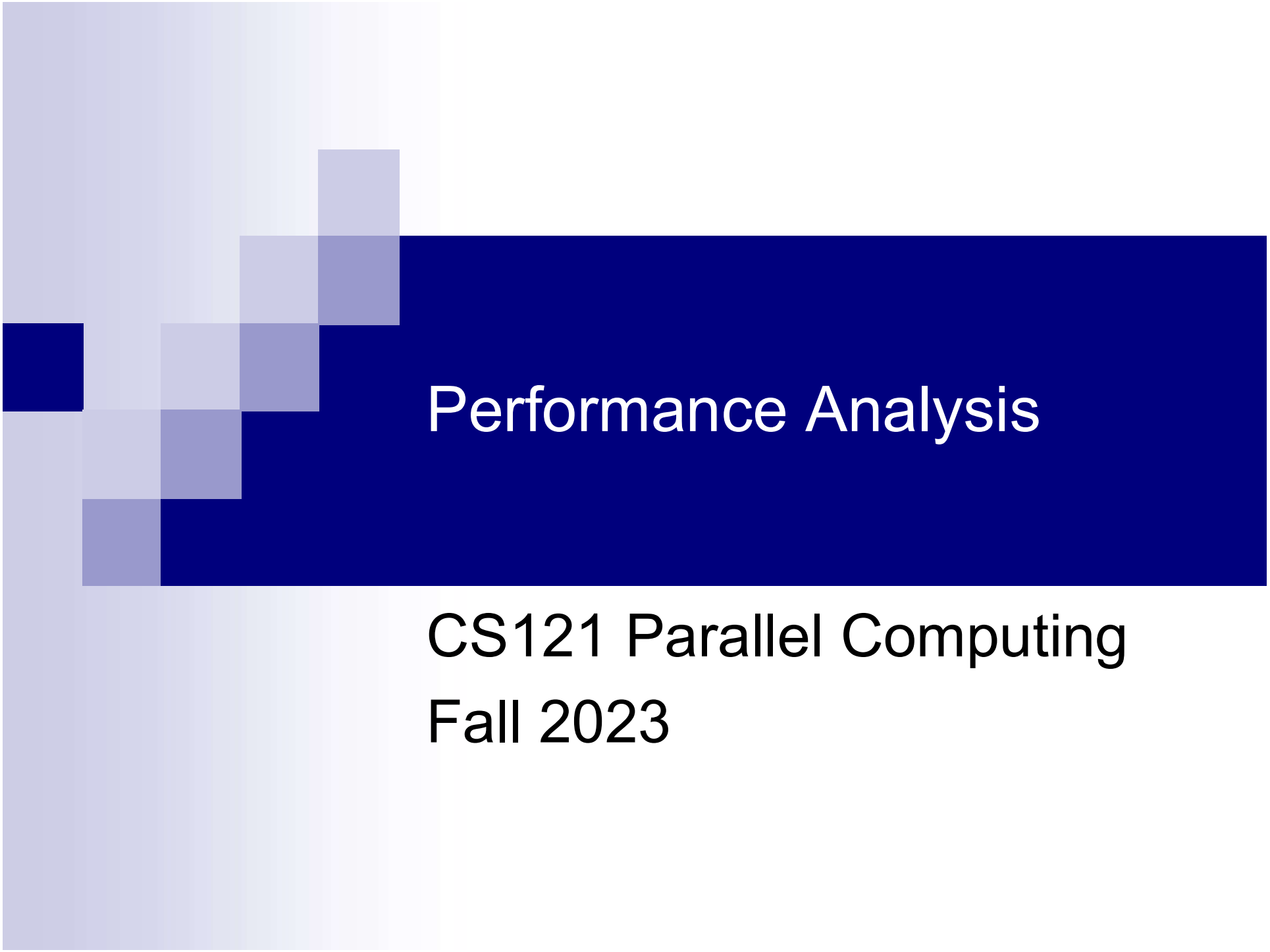
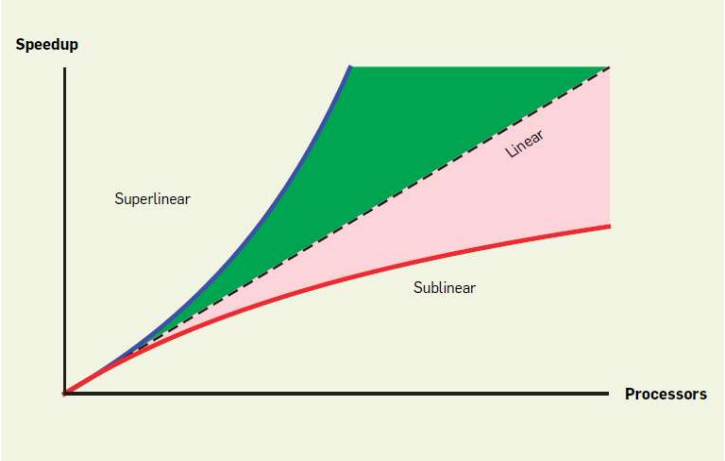
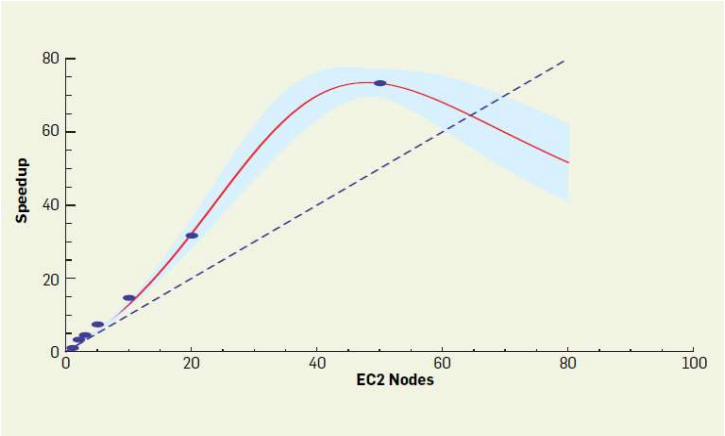

Performance Analysis
CS121 Parallel Computing
Fall 2023

Parallel performance

- Ideally, a program runs k times faster on k processors than on one.
  - Almost never achieved in practice. Why?
- Goal Given a parallel program, understand its performance under increasing levels of parallelism.
  - Identify performance bottlenecks.
  - Compare different algorithms for a problem.
- Try to abstract away machine details.
  - Find performance properties that hold regardless of architecture.
  - Use asymptotic analysis.





Speedup

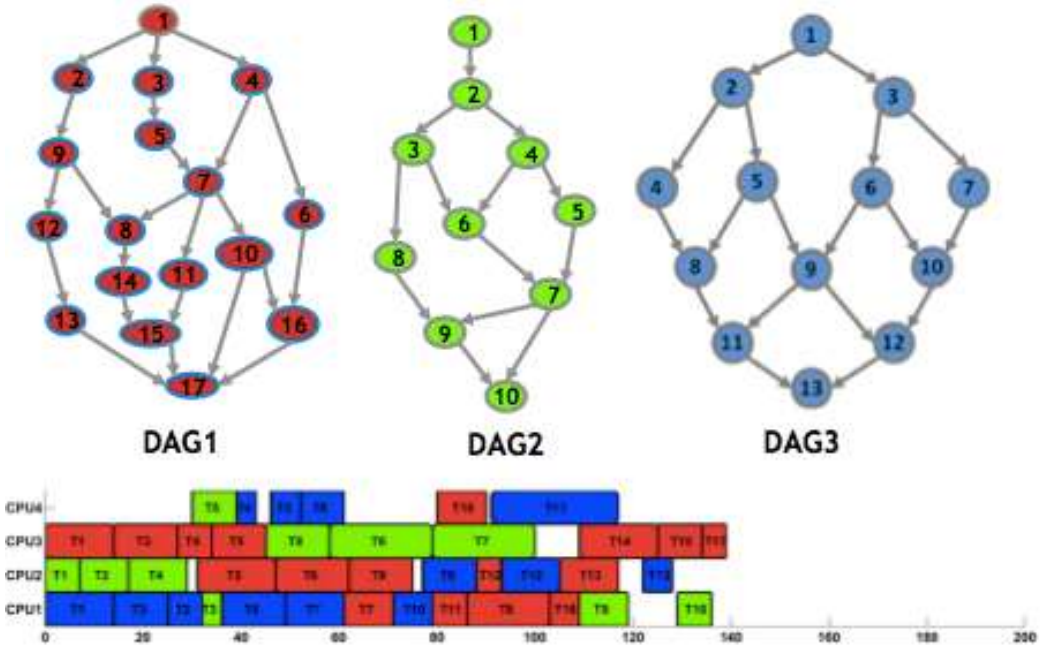
- For a given problem X and parallel algorithm A solving X, let
  - T_s = minimum time to solve X on one processor, i.e. the time for the best sequential algorithm.
  - T_1 = time algorithm A takes using one processor.
    - $T_1 \geq T_s$.
  - T_p = time algorithm A takes using p processors.
- Absolute speedup $S^*_p = T_s / T_p$.
  - Compare A with the best sequential algorithm.
- Relative speedup, aka scalability $S_p = T_1 / T_p$.
  - Compare A with itself on different machine sizes.
  - Focus on scalability, since hard to know what best sequential algorithm is.
- Work $W_p = p\ T_p$ = "total cycles burned" by p processors.
  - $p\ T_p \geq T_s$, because parallel system has to do at least as much total work as the best sequential algorithm.
- Efficiency $E_p = S_p / p$ = speedup per processor.
  - Typically ≤ 1 due to overhead.
  - But can be > 1 in practice in special circumstances.

Overheads

- Linear scalability $S_p = \Theta(p)$.
- Hard to achieve due to overheads.
  - Communication and synchronization time between processors.
  - Processors can idle due to poor partitioning or load imbalance.
  - Sometimes cheaper for a processor to redo a computation than get the result from another processor. But this increases overall computation.
- Sometimes the best sequential algorithm is not parallelizable.
  - Must choose more work intensive (higher W_p) but more parallelizable algorithm.
  - Ex Dijkstra's shortest path algorithm does less work than Bellman-Ford, but BF is more parallelizable.

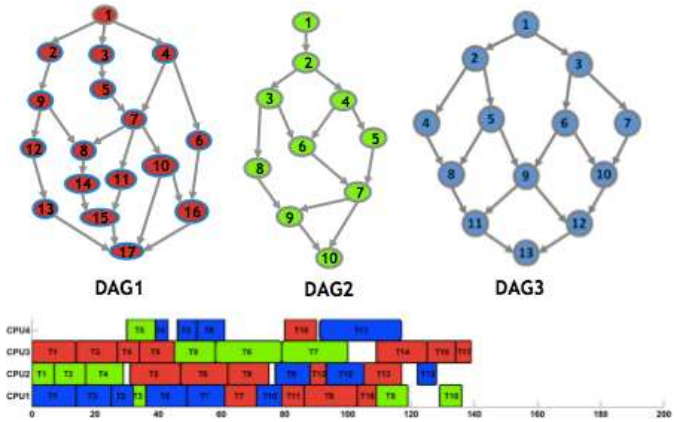
DAG model

- Nodes represent computations / tasks.
  - Can be weighted to represent different task sizes.
- Directed edges represent dependencies between tasks.
  - (u,v) indicates task u must finish before task v completes.
  - Can be weighted to indicate communication, startup cost etc. for task.
    - Assume default weight is 0.
- Graph can't contain cycles, i.e. computation must eventually finish.
- Critical path is length of longest directed path in graph.



DAG model

- Given a dag G, let
  - C = sum of node and edge weights in G.
  - D = length of critical path in G.
  - T_p = time taken by p processors to execute G.
  - T_∞ = minimum time to execute G using arbitrary number of processors.
- Assume p processors, each doing one unit of work per time step.
- Work law p $T_p \geq C$.
  - I.e. $T_p \geq C / p$. Places lower bound on parallel running time.
  - Holds because processors must do all the work in G.
- Span law $T_\infty \geq D$.
  - Due to dependency, tasks along critical path must be done sequentially, regardless of number of processors.

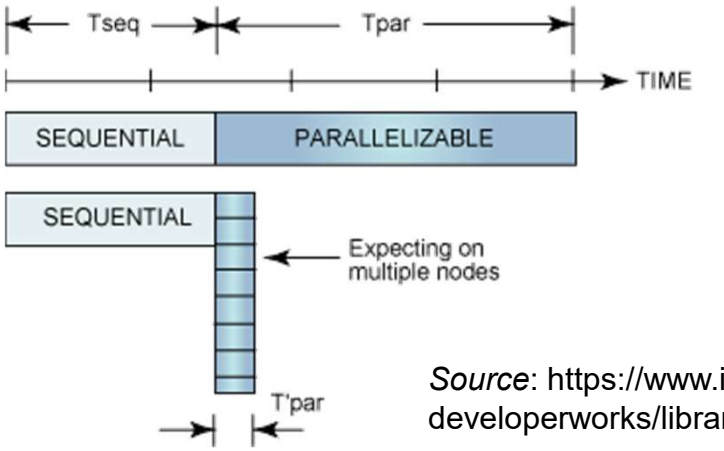


Amdahl's Law

- Another upper bound on the maximum speedup achievable by a program.
- Let T_1 be time to run a program using one processor.
  - Assume f fraction of it the program is inherently sequential, i.e. cannot be parallelized.
  - f T_1 amount of work is sequential.
  - (1-f) T_1 amount of work is parallelizable.
- On parallel computer with p processors
  - Sequential part still takes f T_1 time.
  - Parallel part takes (1-f) T_1 / p time.
  - Total time T_p = f T_1 + (1-f) T_1 / p.

Amdahl's Law

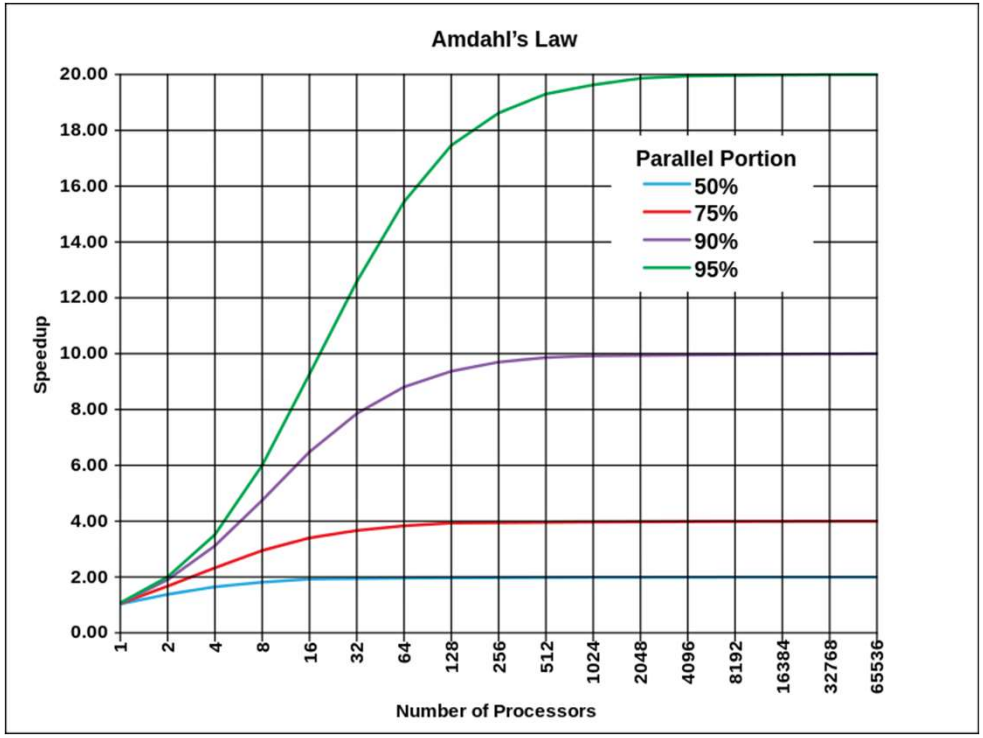
- $S(p) = \frac{T_1}{T_P} = \frac{T_1}{fT_1 + \frac{(1-f)T_1}{p}} = \frac{1}{f + \frac{1-f}{p}}$



Source: https://www.ibm.com/developerworks/library/l-cluster1/

- As p increases, second term in denominator gets smaller, but first term always stays the same.
- So S(p) < 1/f always.
  - Speedup never exceeds one divided by the fraction of sequential work.

Amdahl's Law



Source: Wikipedia

- Even with a small proportion of sequential work, parallelism is very limited.
- Suggests it's pointless to build large computers with thousands of cores, since they can't improve performance much.
- But, in real world large scale parallelism is very useful. Why?

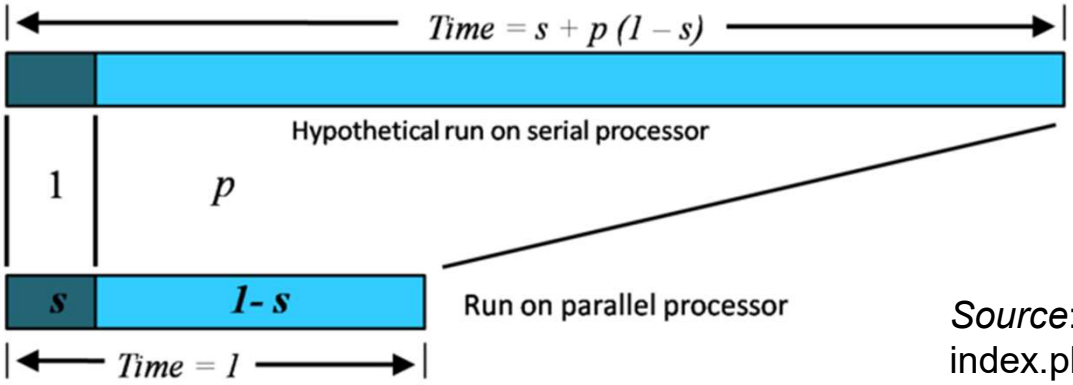
Gustafson's Law

- Gustafson's law tries to explain why real parallel systems often get good speedup.
- Amdahl's Law assumes a fixed problem size, and looks at runtime as we increase parallelism.
- But when parallelism increases, we often try to solve a larger (scaled) problem.
  - Ex Given few processors, we try to solve a coarse weather model. Given more processors, we try to solve a finer model.
- Gustafson's Law assumes size of scaled problem is chosen so the parallel running time stays the same as the running time of the original problem.
- Gustafson's Law also assumes the sequential work in the scaled and unscaled problems are the same.
  - Ex For weather prediction, sequential work may be initialization, which is roughly same in coarse and fine model.
  - This assumption may or may not be valid in practice.
- The parallel work increases in the scaled problem.
  - Ex Parallel work (the actual weather simulation) increases in finer model.

Gustafson's Law

- Given p processors
  - Sequential work is still f T_1.
  - Parallel work chosen to be p(1-f) T_1.
  - T_p = f T_1 + p(1-f) T_1 / p = f T_1 + (1-f) T_1 = T_1.
    - Parallel running time on larger problem equals sequential running time for original small problem, as per assumption.
- Size of scaled problem is f T_1 + p(1-f) T_1.
- So a sequential processor takes T'_1 = f T_1 + p(1-f) T_1 time to solve scaled problem.
- Scaled speedup ratio

$$S_p = \frac{T'_1}{T_p} = \frac{fT_1 + p(1-f)T_1}{T_1} = f + p(1-f) = p - (p-1)f$$



Source: http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2012/4b_rs

Comparison and limitations

- Gustafson's Law predicts much better speedup than Amdahl's Law.
- Ex For f = 0.05, p = 50.
  - Amdahl's Law gives speedup = 1 / (0.05+.95/50) = 14.5
  - Gustafson's Law gives speedup = 50-49*0.05 = 47.6
- Amdahl's Law also called strong scaling, i.e. performance improvement for fixed problem size and increasing processors.
- Gustafson's Law also called weak scaling, i.e. performance improvement for fixed problem size per processor.
- Which assumption is more appropriate for given problem determines which law applicable.
- Important shortcoming of Amdahl's and Gustafson's Laws is they ignore overhead as parallelism increases.

Karp-Flatt metric

- Capture parallelism overhead using empirical observations.
- Given a program, let T_q and T_r be the sequential and parallelizable part, resp.
  - $T_1 = T_q + T_r$.
- Experimentally determined serial fraction $e = T_q / T_1$.
  - A priori we don't know e. But we can determine it through other measurable quantities.
- $T_p = T_q + \frac{T_r}{p} = T_1 e + \frac{T_1(1-e)}{p}$
- $\frac{T_p}{T_1} = e + \frac{(1-e)}{p} = \frac{1}{S_p}$
- $e = (\frac{1}{S_p} - \frac{1}{p}) / (1 - \frac{1}{p})$
- S_p can be determined experimentally by running the program. Then we can use S_p to determine the serial fraction e.

Karp-Flatt metric

$$e = \left(\frac{1}{S_p} - \frac{1}{p}\right) / \left(1 - \frac{1}{p}\right)$$

- Suppose observed speedups for a program are

p	2	3	4	5	6	7	8
S_p	1.87	2.61	3.23	3.73	4.14	4.46	4.71
e	0.07	0.075	0.08	0.085	0.09	0.095	0.1

- Since e increases with p, this says overhead is increasing with parallelism.

p	2	3	4	5	6	7	8
S_p	1.82	2.50	3.08	3.57	4.00	4.38	4.71
e	0.1	0.1	0.1	0.1	0.1	0.1	0.1

- Since e is constant with p, overhead does not increase.

Asymptotic complexity

- Compare trends in the sizes of two functions.
- Given functions f(n) and g(n), there are 5 comparison operators.
  - Analogous to the 5 comparison operators for numbers.

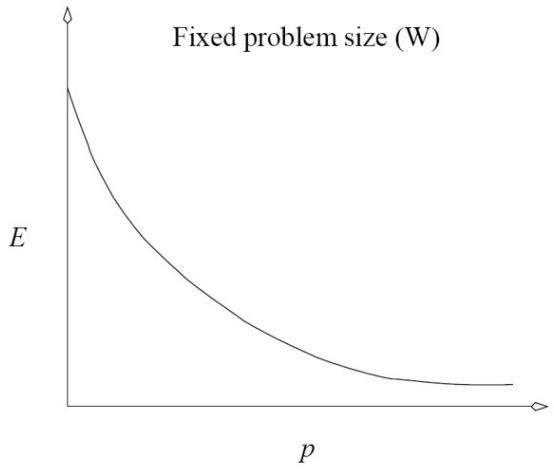
Notation	Analogy	Formal definition
f(n) = O(g(n))	$f \leq g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
f(n) = Ω(g(n))	$f \geq g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
f(n) = Θ(g(n))	$f = g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ 0 < c < \infty$
f(n) = o(g(n))	$f < g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
f(n) = ω(g(n))	$f > g$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

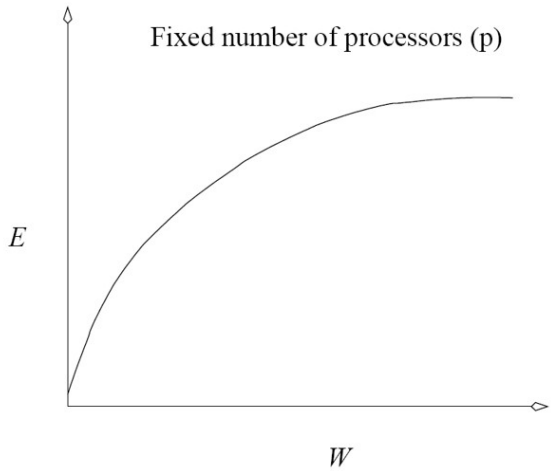
Isoefficiency

- Another way to measure scalability, by looking at the amount of useful work vs overhead as we scale the problem and system.
- Given a problem of size n and p processors, let
  - W(n) = useful work done by processors.
  - γ(n,p) = overhead from parallelism.
  - Both W and γ increase in n and p.
- Efficiency $E = \frac{W(n)}{W(n)+\gamma(n,p)} = \frac{1}{1+\frac{\gamma(n,p)}{W(n)}}$.
- Goal is to maintain the same (i.e. iso) efficiency as the number of processors increases.
  - Whether this is possible depends on the relative rate of increase of W vs γ.

Work vs overhead

- As p increases but n stays constant, γ typically increases faster than W.
  - More processors have to do more communication, synchronization, etc.
  - Efficiency decreases for constant problem size and increasing number of processors.
- As n increases but p stays constant, W typically increases faster than γ.
  - There's more work per processor, so they can spend more time computing instead of communicating.
  - Efficiency increases for constant number of processors and increasing problem size.
- Thus, can use increase in problem size to balance increase in processor count to maintain efficiency, i.e. achieve isoefficiency.





Examples

- Ex $W(n) = n \log n$, $\gamma(n,p) = n \log p$
  - $E = \frac{1}{1+\frac{n \log p}{n \log n}} = \frac{1}{1+\frac{\log p}{\log n}}$
  - So if log n = Ω(log p), i.e. n = Ω(p), can maintain isoefficiency.
- Ex $W(n) = n$, $\gamma(n,p) = p^{3/2} + p^{3/4} n^{3/4}$
  - Find isoefficiency problem size for each term separately.
  - For first term, need $n = \Omega(p^{3/2})$.
  - For second term, need $n = \Omega(p^{3/4} n^{3/4})$, so $n^{1/4} = p^{3/4}$, so $n = \Omega(p^3)$.
  - Thus, $n = \Omega(p^3)$ maintains isoefficiency.
- Ex $W(n) = n^2$, $\gamma(n,p) = n^2 \log p$
  - $E = \frac{1}{1+\log p}$, i.e. E decreases in p, regardless of n.
  - So not possible to maintain isoefficiency.