Speedup

Thoai Nam

High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Engineering

HCMC University of Technology

Outline

- Speedup & Efficiency
- Amdahl's Law
- Gustafson's Law
- Sun & Ni's Law

Speedup & Efficiency

- **Speedup**:

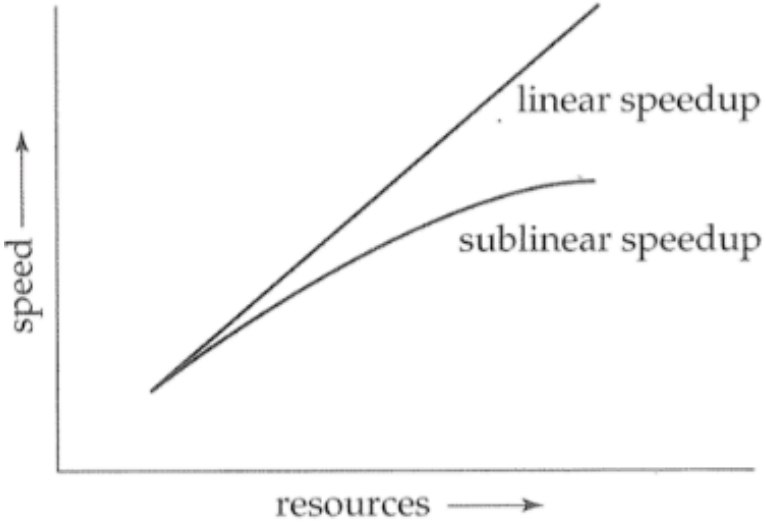
  S = Time(*the most efficient sequential algorithm*) / Time(*parallel algorithm*)

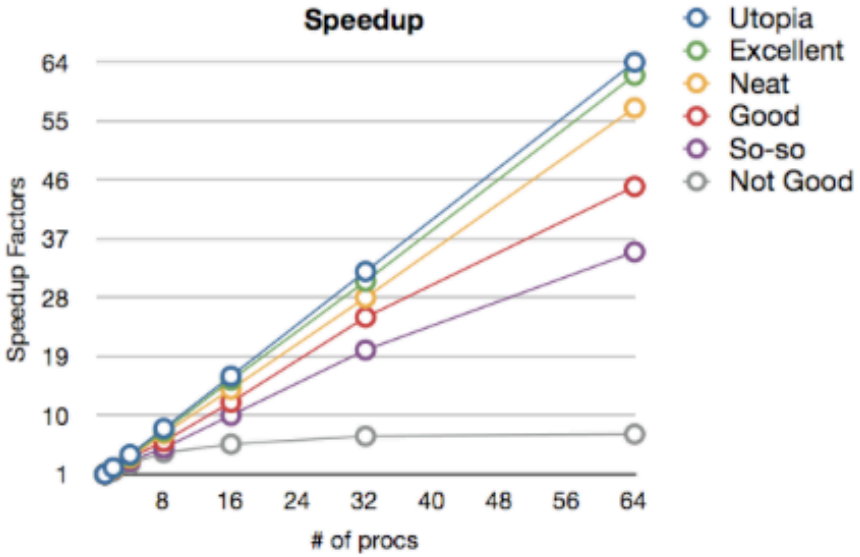
- **Efficiency**:

  E = S / N with N is the number of processors

Speedup

- The fundamental concept in parallelism
  - T(1) = time to execute task on a single resource
  - T(n) = time to execute task on n resources
  - Speedup = T(1)/T(n)





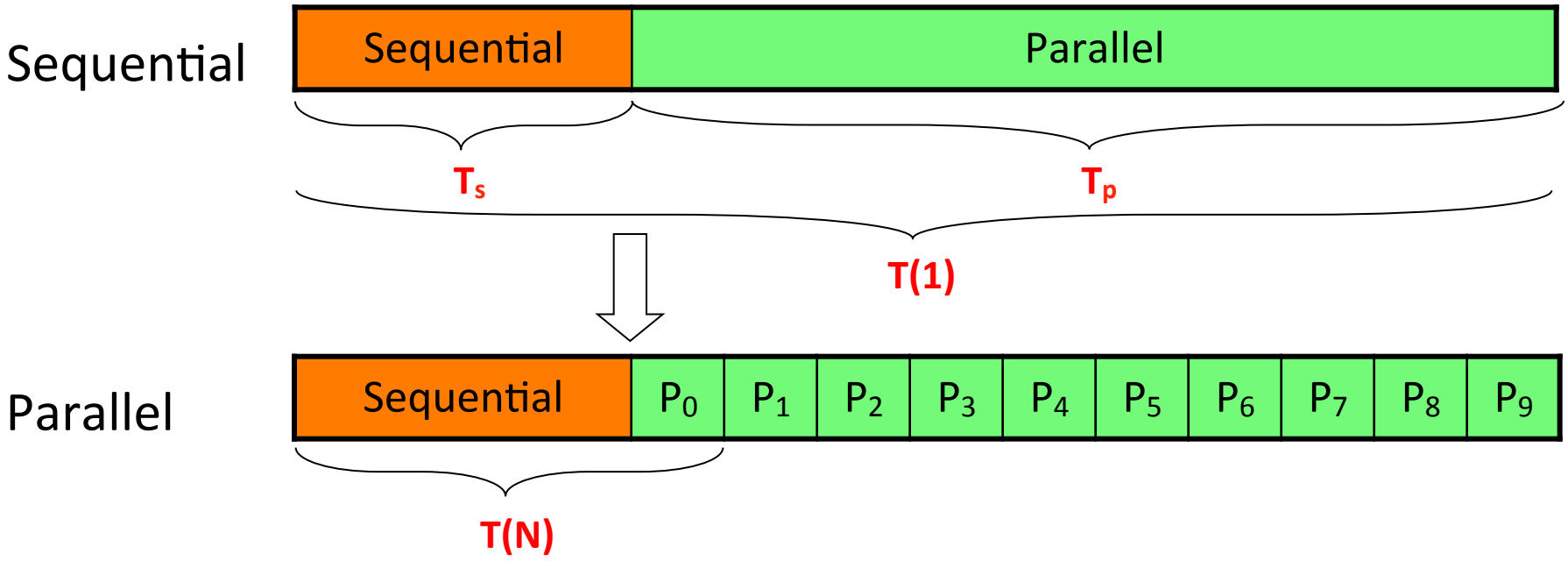
http://web.eecs.utk.edu/~huangj/hpc/hpc_intro.php

From Silberschatz, Korth, and Sudarshan. *Database Systems Concepts, 4th Ed.*

Amdahl's Law – Fixed Problem Size (1)

- The main objective is to produce the results as soon as possible
  - (ex) video compression, computer graphics, VLSI routing, etc
- Implications
  - Upper-bound is
  - Make Sequential bottleneck as small as possible
  - Optimize the common case
- Modified Amdahl's law for **fixed problem size** including the overhead

Amdahl's Law – Fixed Problem Size (2)



$T_s = \alpha T(1) \Rightarrow T_p = (1-\alpha)T(1)$

$$T(N) = \alpha T(1) + (1-\alpha)T(1)/N$$

Number of processors

Amdahl's Law – Fixed Problem Size (3)

$$Speedup = \frac{Time(1)}{Time(N)}$$

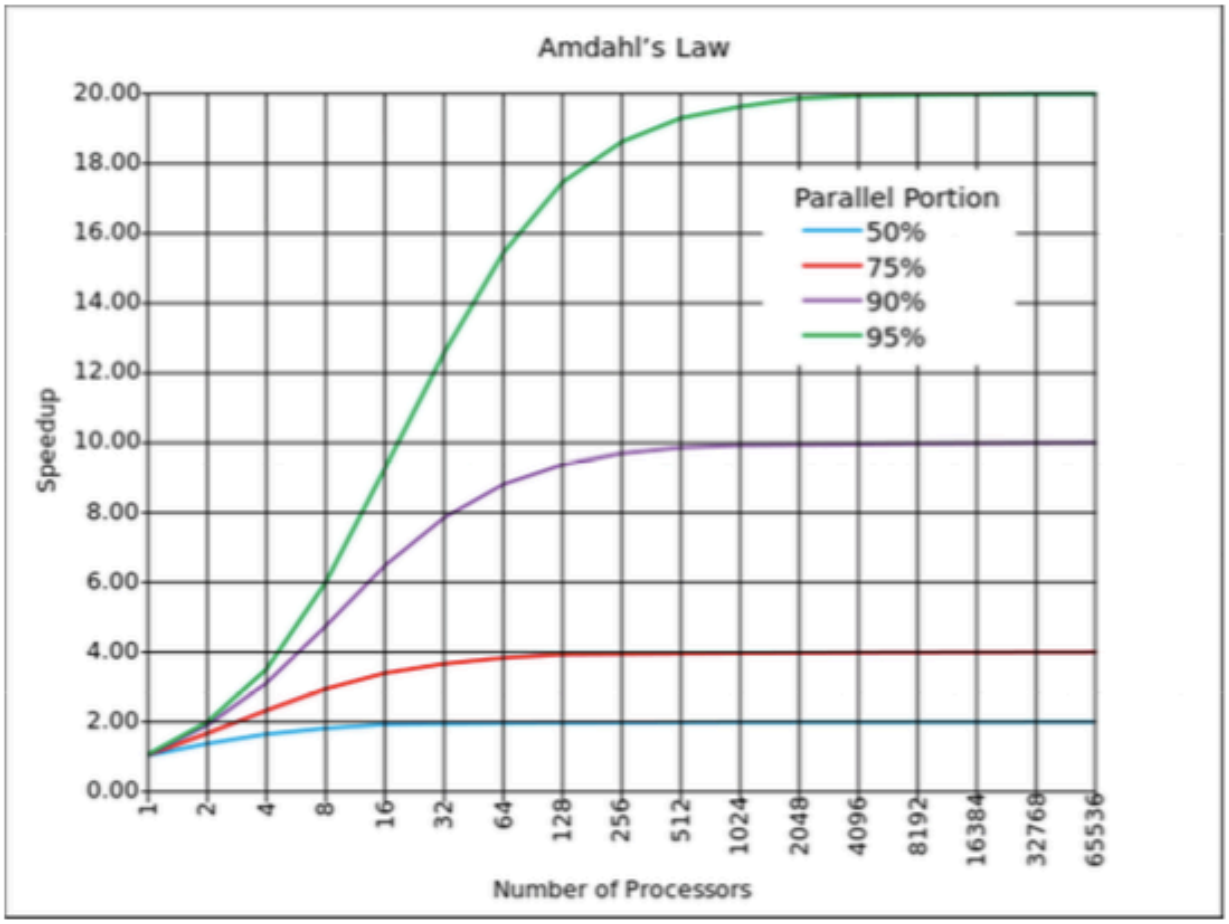
$$Speedup = \frac{T(1)}{\alpha T(1) + \frac{(1-\alpha)T(1)}{N}} = \frac{1}{\alpha + \frac{(1-\alpha)}{N}} \rightarrow \frac{1}{\alpha} \; as \; N \rightarrow \infty$$

Enhanced Amdahl's Law

The overhead includes parallelism and interaction overheads

$$Speedup = \frac{T(1)}{\alpha T(1) + \frac{(1-\alpha)T(1)}{N} + T_{overhead}} \rightarrow \frac{1}{\alpha + \frac{T_{overhead}}{T(1)}} \; as \; N \rightarrow \infty$$

Amdahl's Law



Source: Wikipedia

Gustafson's Law – Fixed Time (1)

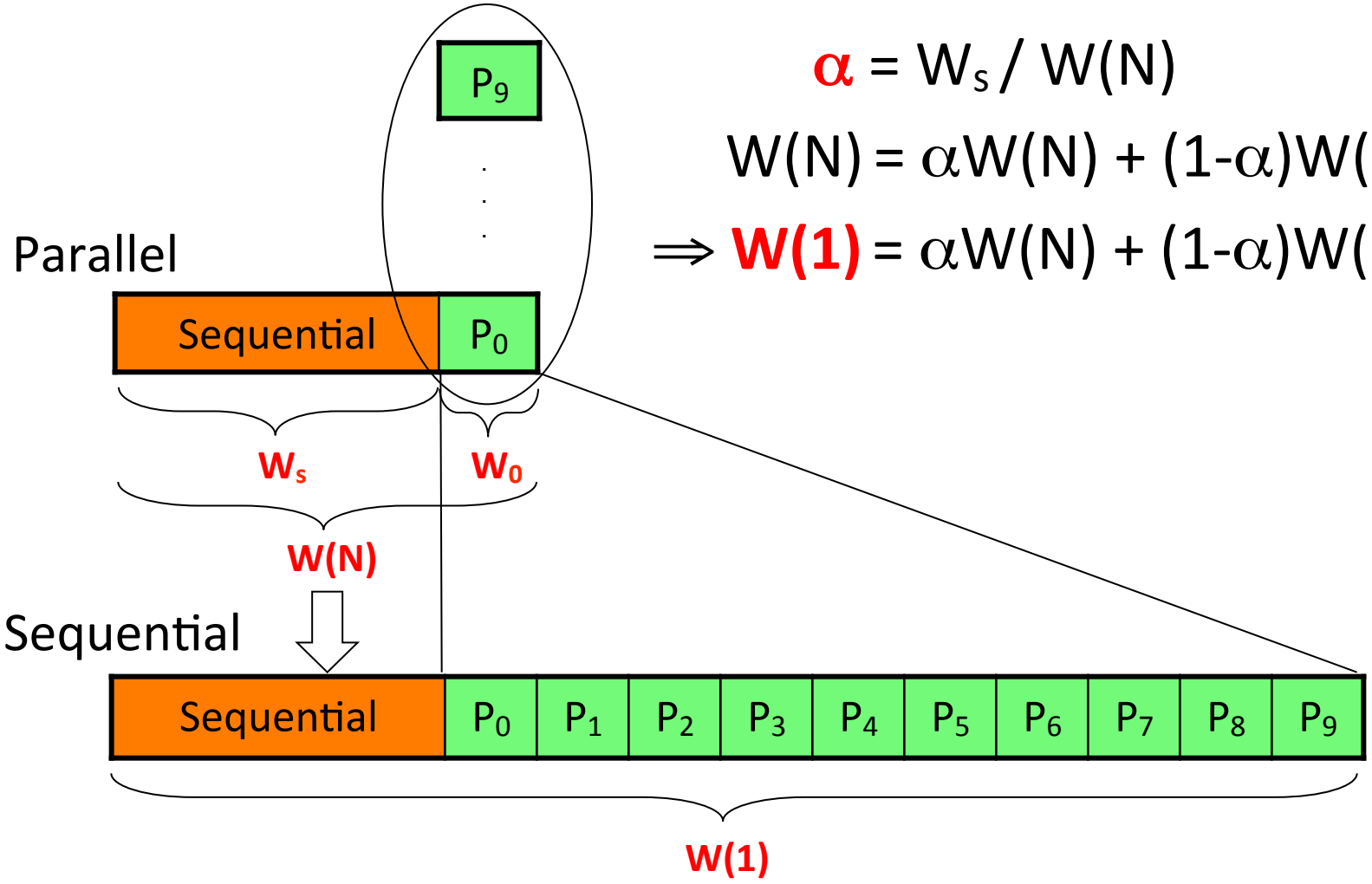
- User wants more accurate results within a time limit
  - Execution time is fixed as system scales
  - (ex) FEM (Finite element method) for structural analysis, FDM (Finite difference method) for fluid dynamics
- Properties of a work metric
  - Easy to measure
  - Architecture independent
  - Easy to model with an analytical expression
  - No additional experiment to measure the work
  - The measure of work should scale linearly with sequential time complexity of the algorithm
- Time constrained seems to be most generally viable model!

Gustafson's Law – Fixed Time (2)

$$\alpha = W_s / W(N)$$

$$W(N) = \alpha W(N) + (1-\alpha)W(N)$$

$$\Rightarrow \mathbf{W(1)} = \alpha W(N) + (1-\alpha)W(N)*N$$



Gustafson's Law – Fixed Time without overhead

Time = Work * k

W(N) = W

$$Speedup = \frac{T(1)}{T(N)} = \frac{W(1)*k}{W(N)*k} = \frac{\alpha W + (1-\alpha)NW}{W} = \alpha + (1-\alpha)N$$

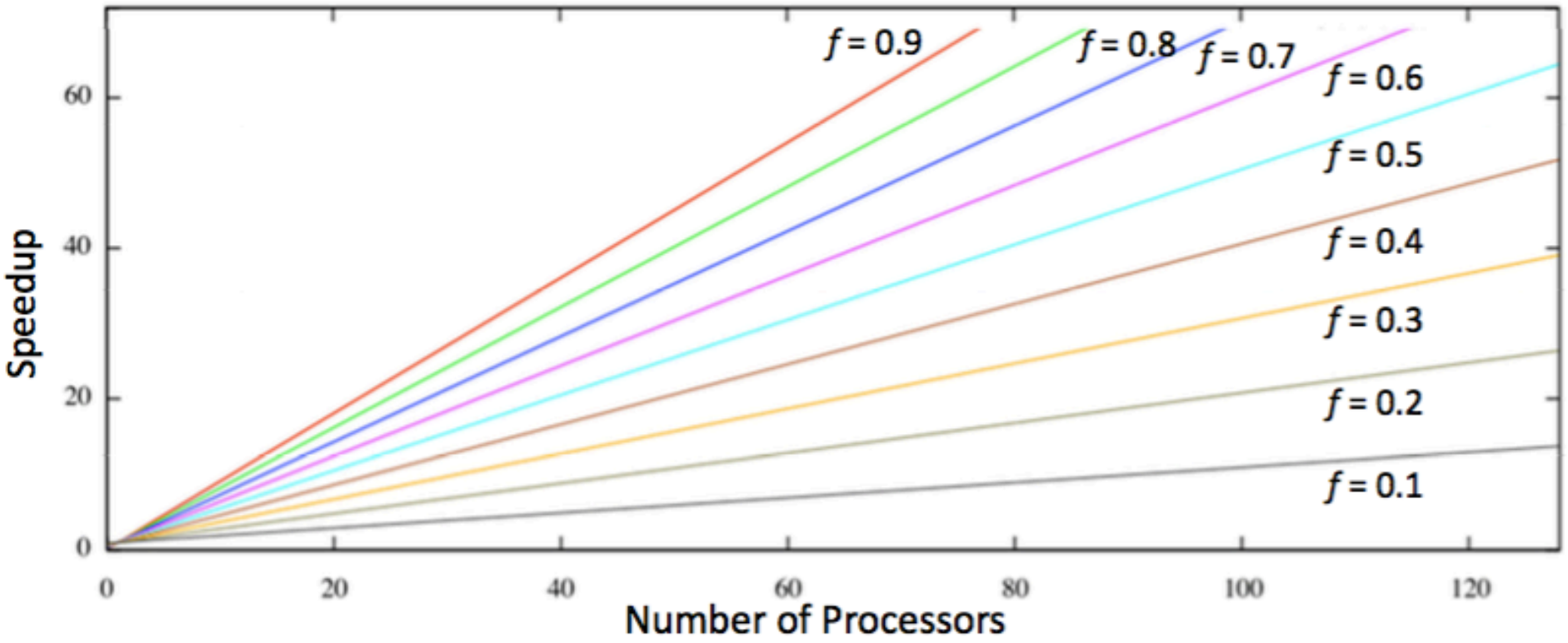
Gustafson's Law – Fixed Time with overhead

$W(N) = W + W_0$

$$Speedup = \frac{T(1)}{T(N)} = \frac{W(1)*k}{W(N)*k} = \frac{\alpha W + (1-\alpha)NW}{W + W_0} = \frac{\alpha + (1-\alpha)N}{1 + \frac{W_0}{W}}$$

Gustafson's Law – Fixed Time

Suppose only a fraction ***f*** of a computation was parallelized



Source: Wikipedia

Sun and Ni's Law – Fixed Memory (1)

- Scale the largest possible solution limited by the memory space. Or, fix memory usage per processor
- Speedup
  - Time(1)/Time(N) for scaled up problem is not appropriate
  - For simple profile, and **G(N)** is the increase of parallel workload as the memory capacity increases N times

Sun and Ni's Law – Fixed Memory (2)

- W = αW+(1- α)W
- Let M be the memory capacity of a single node
- *N* nodes:
  - the increased memory N*M
  - The scaled work: W = αW+(1- α)W*G(N)

$$Speedup_{MC} = \frac{\alpha + (1-\alpha)G(N)}{\alpha + (1-\alpha)\frac{G(N)}{N}}$$

Sun and Ni's Law – Fixed Memory (3)

- Definition:

  A function g is homomorphism if there exists a function such that $\overline{g}$ for any real number *c* and variable *x*,

$$g(cx) = \overline{g}(c) * g(x)$$

- Theorem:

  If W = $g(M)$ for some homomorphism function g, then with all data being shared by all available processors, the simplified memory-bounced speedup is

$$S_N^* = \frac{W_1 + \overline{g}(N)W_N}{W_1 + \frac{\overline{g}(N)}{N}W_N} = \frac{\alpha + (1-\alpha)G(N)}{\alpha + (1-\alpha)\frac{G(N)}{N}}$$

Sun and Ni's Law – Fixed Memory (4)

Proof:

Let the memory requirement of W_n be M, $W_n = g(M)$.

M is the memory requirement when 1 node is available.

With N nodes available, the memory capacity will increase to N*M.

Using all of the available memory, for the scaled parallel portion W_N^*: $W_N^* = g(N*M) = \bar{g}(N)*g(M) = \bar{g}(N)*W_N$

.

$$S_N^* = \frac{W_1^* + W_N^*}{W_1^* + \frac{W_N^*}{N}} = \frac{W_1 + \bar{g}(N)W_N}{W_1 + \frac{\bar{g}(N)}{N}W_N}$$

Speedup

$$S_N^* = \frac{W_1 + G(N)W_N}{W_1 + \frac{G(N)}{N}W_N}$$

- When the problem size is independent of the system, the problem size is fixed, *G(N)=1* $\Rightarrow$ Amdahl's Law.
- When memory is increased N times, the workload also increases N times, *G(N)=N* $\Rightarrow$ Gustafson's Law
- For most of the scientific and engineering applications, the computation requirement increases faster than the memory requirement, *G(N)>N*.

Scalability

- Parallelizing a code does not always result in a speedup; sometimes it actually slows the code down! This can be due to a poor choice of algorithm or to poor coding
- The best possible speedup is **linear**, i.e. it is proportional to the number of processors: T(N) = T(1)/N where N = number of processors, T(1) = time for serial run.
- A code that continues to speed up reasonably close to linearly as the number of processors increases is said to be **scalable**. Many codes scale up to some number of processors but adding more processors then brings no improvement. Very few, if any, codes are indefinitely scalable.

Factors That Limit Speedup

- Software overhead

  Even with a completely equivalent algorithm, software overhead arises in the concurrent implementation. (e.g. there may be additional index calculations necessitated by the manner in which data are "split up" among processors.) i.e. there is generally more lines of code to be executed in the parallel program than the sequential program.

- Load balancing
- Communication overhead

Superlinear Speedup

- Theoretically, $S_p < p$ (p is the number of processors)

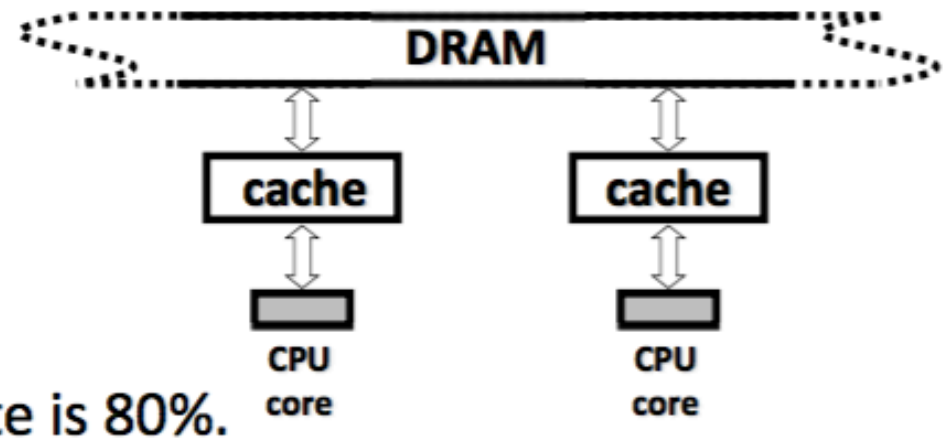
But in practice superlinear speedup is sometimes observed, that is $S_p > p$, (why?)

- Reasons for superlinear speedup
  - Cache effects
  - Exploratory decomposition

Superlinear Speedup
(Cache Effects)

Let cache access latency = 2 ns

DRAM access latency = 100 ns



Suppose we want solve a problem instance that executes *k* FLOPs.

With 1 Core: Suppose cache hit rate is 80%.

If the computation performs 1 FLOP/memory access, then each FLOP will take 2 × 0.8 + 100 × 0.2 = 21.6 ns to execute.

With 2 Cores: Cache hit rate will improve. (why?)

Suppose cache hit rate is now 90%.

Then each FLOP will take 2 × 0.9 + 100 × 0.1 = 11.8 ns to execute.

Since now each core will execute only *k* / 2 FLOPs,

$$\text{Speedup, } S_2 = \frac{k \times 21.6}{(k/2) \times 11.8} \approx 3.66 > 2$$

Superlinear Speedup
(Due to Exploratory Decomposition)

Consider searching an array of 2*n* unordered elements for a specific element *x*.

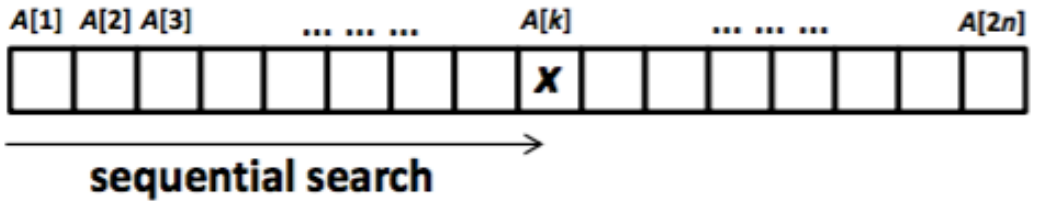
Suppose *x* is located at array location $k > n$ and *k* is odd.

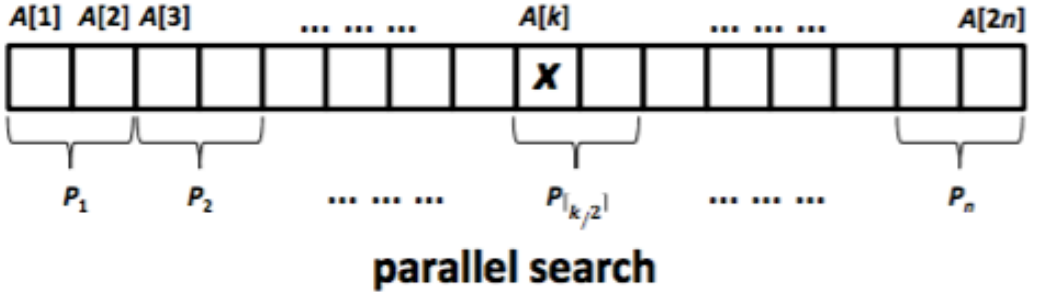
Serial runtime, $T_1 = k$

Parallel running time with *n* processing elements, $T_n = 1$

Speedup, $S_n = \frac{T_1}{T_n} = k > n$

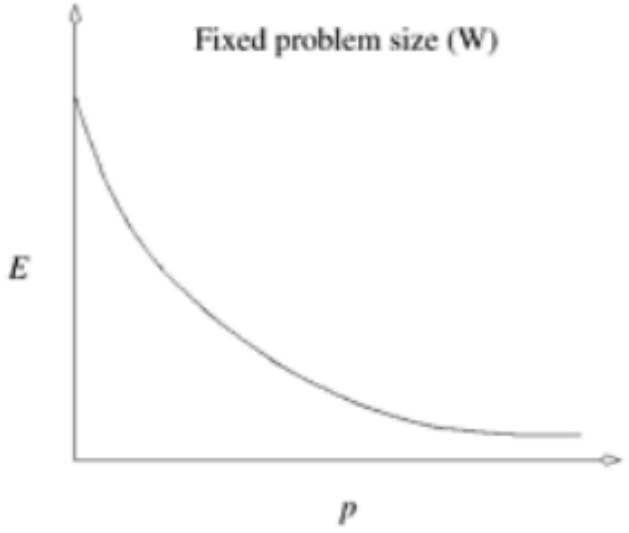
Speedup is superlinear!

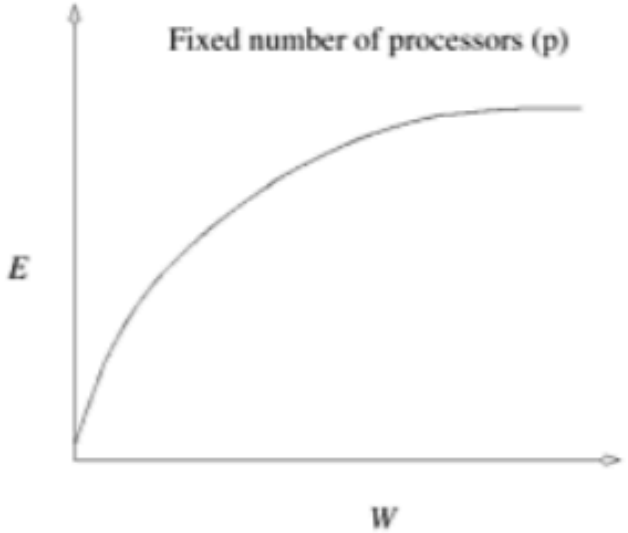




Scalable Parallel Algorithms

Efficiency, $E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$





Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

A parallel algorithm is called *scalable* if its efficiency can be maintained at a fixed value by simultaneously increasing the number of processing elements and the problem size.

Scalability reflects a parallel algorithm's ability to utilize increasing processing elements effectively.

Scalable Parallel Algorithms

In order to keep E_p fixed at a constant *k*, we need

$$E_p = k \Rightarrow \frac{T_1}{pT_p} = k \Rightarrow T_1 = kpT_p$$

For the algorithm that adds *n* numbers using *p* processing elements:

$$T_1 = n \text{ and } T_p = \frac{n}{p} + 2\log p$$

So in order to keep E_p fixed at *k*, we must have:

$$n = kp\left(\frac{n}{p} + 2\log p\right) \Rightarrow n = \frac{2k}{1-k} p \log p$$

n	*p* = 1	*p* = 4	*p* = 8	*p* = 16	*p* = 32
64	1.0	***0.80***	0.57	0.33	0.17
192	1.0	0.92	***0.80***	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	***0.80***	0.62

Fig: Efficiency for adding *n* numbers using *p* processing elements

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition