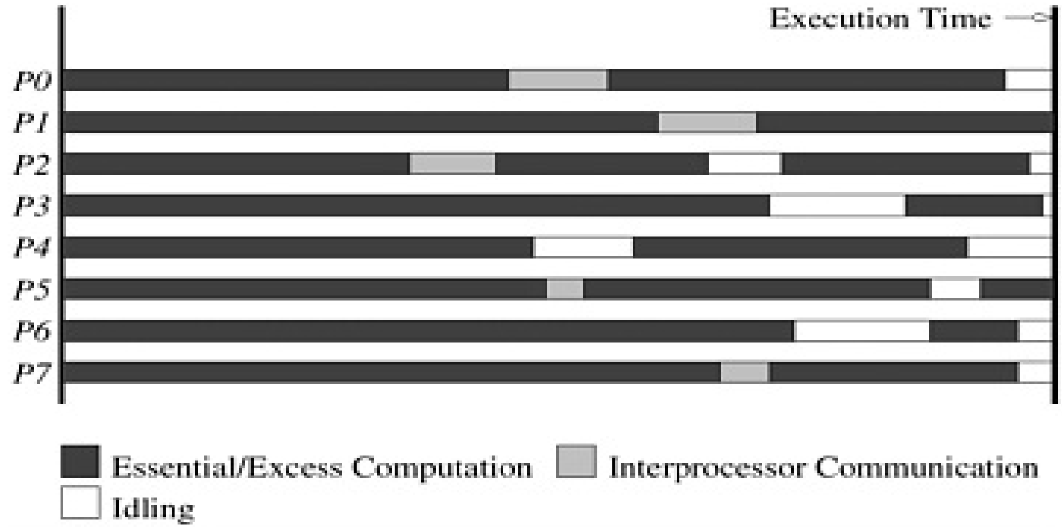
CSE 590: Special Topics Course
(Supercomputing)

Lecture 2
(Analytical Modeling of Parallel Programs)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2016

Parallel Execution Time & Overhead



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

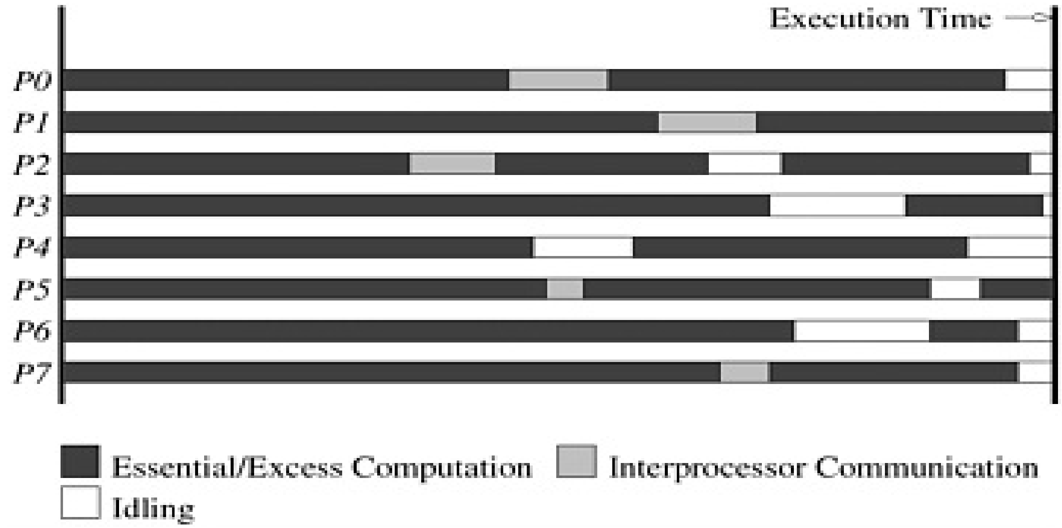
Parallel running time on p processing elements,

$$T_P = t_{end} - t_{start},$$

where, t_{start} = starting time of the processing element that starts first

t_{end} = termination time of the processing element that finishes last

Parallel Execution Time & Overhead

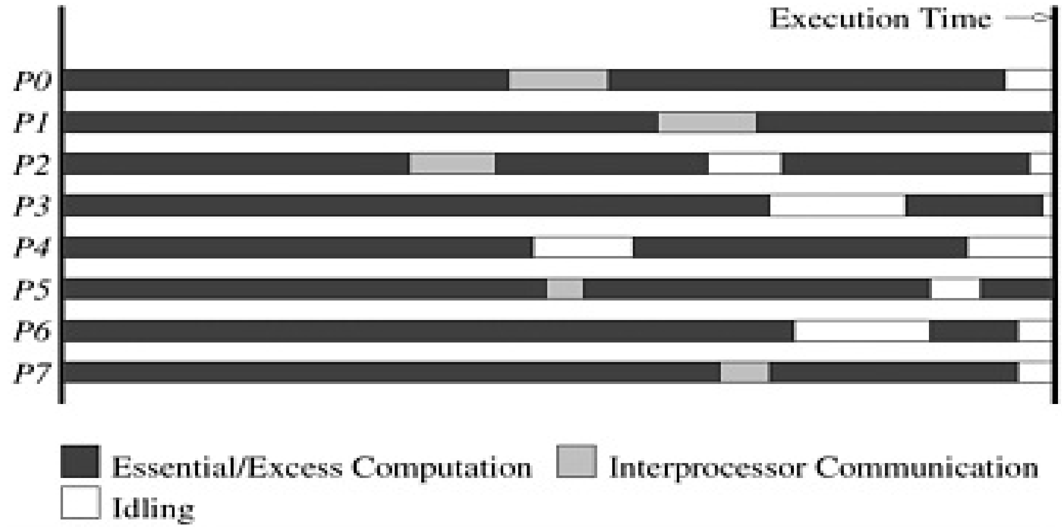


Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition

Sources of overhead (w.r.t. serial execution)

- Interprocess interaction
  - Interact and communicate data (e.g., intermediate results)
- Idling
  - Due to load imbalance, synchronization, presence of serial computation, etc.
- Excess computation
  - Fastest serial algorithm may be difficult/impossible to parallelize

Parallel Execution Time & Overhead



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

Overhead function or total parallel overhead,

$$T_O = pT_p - T,$$

where, p = number of processing elements

T = time spent doing useful work

(often execution time of the fastest serial algorithm)

Speedup

Let T_p = running time using p identical processing elements

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

Theoretically, $S_p \le p$ (why?)

Perfect or *linear* or *ideal* speedup if $S_p = p$

Speedup

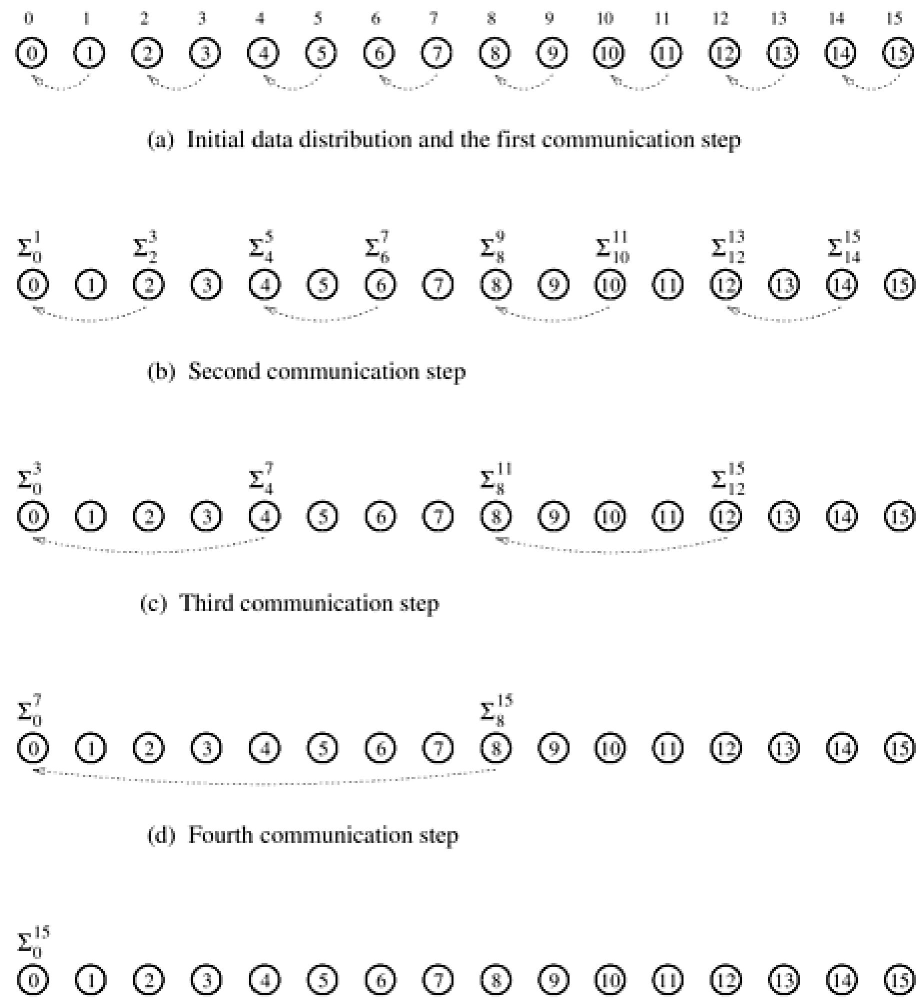
Consider adding n numbers using n identical processing elements.

Serial runtime, $T_1 = \Theta(n)$

Parallel runtime, $T_n = \Theta(\log n)$

Speedup, $S_n = \frac{T_1}{T_n} = \Theta\left(\frac{n}{\log n}\right)$

Speedup not ideal.



(a) Initial data distribution and the first communication step

(b) Second communication step

(c) Third communication step

(d) Fourth communication step

(e) Accumulation of the sum at processing element 0 after the final communicatior

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

Superlinear Speedup

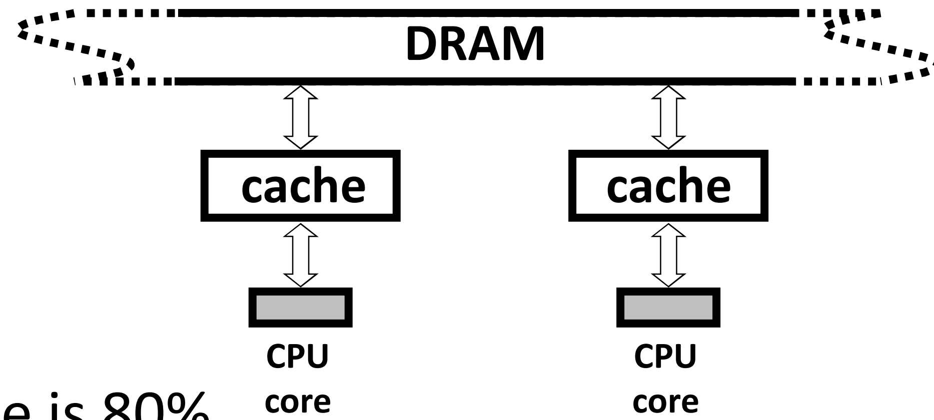
Theoretically, $S_p \leq p$

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 $2 \times 0.8 + 100 \times 0.2 = 21.6$ ns to execute.

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

Suppose cache hit rate is now 90%.

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

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

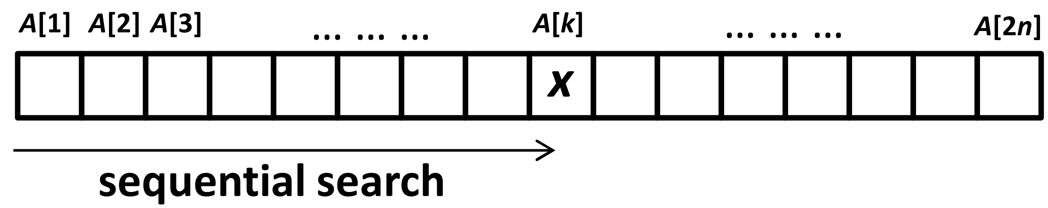
Speedup, $S_2 = \frac{k \times 21.6}{(k/2) \times 11.8} \approx 3.66 > 2$

Superlinear Speedup (Due to Exploratory Decomposition)

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

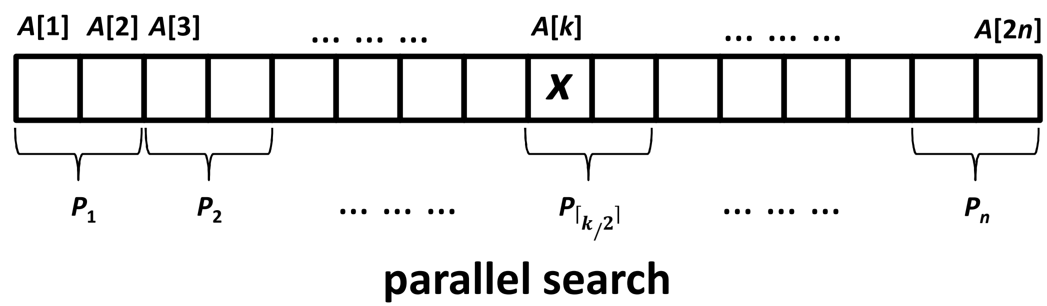
Suppose x is located at array location $k > n$ and k is odd.

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!

Parallelism & Span Law

We defined, T_p = runtime on p identical processing elements

Then span, T_∞ = runtime on an infinite number of identical processing elements

Parallelism, $P = \frac{T_1}{T_\infty}$

Parallelism is an upper bound on speedup, i.e., $S_p \leq P$ (why?)

Span Law

$$\boxed{T_p \geq T_\infty}$$

Efficiency

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

Efficiency is a measure of the fraction of time for which a processing element is usefully employed.

In an ideal parallel system, $S_p = p$ and $E_p = 1$.

Consider again the example of adding *n* numbers using *n* identical processing elements.

Speedup, $S_n = \frac{T}{T_n} = \Theta\left(\frac{n}{\log n}\right)$

Efficiency, $E_n = \frac{S_n}{n} = \Theta\left(\frac{1}{\log n}\right)$

Cost or Work

The cost of solving (or work performed for solving) a problem:

On a Serial Computer: is the execution time T of the fastest known sequential algorithm for solving the problem.

On a Parallel Computer: is given by pT_p.

A parallel algorithm is *cost-optimal* or *work-optimal* provided

$$pT_p = \Theta(T)$$

For a work-optimal parallel algorithm: $E_p = \frac{S_p}{p} = \frac{T}{pT_p} = \Theta(1)$

Our algorithm for adding *n* numbers using *n* identical processing elements is clearly not cost optimal.

Work Law

The cost of solving (or work performed for solving) a problem:

On a Serial Computer: is given by T_1

On a Parallel Computer: is given by pT_p

Work Law

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

Work Optimality

Let T_s = runtime of the optimal or the fastest known serial algorithm

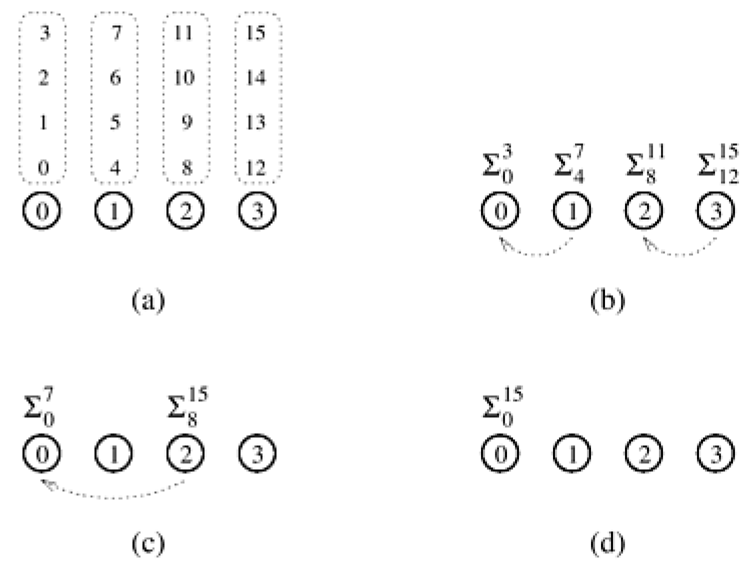
A parallel algorithm is *cost-optimal* or *work-optimal* provided

$$pT_p = \Theta(T_s)$$

Our algorithm for adding n numbers using n identical processing elements is clearly not work optimal.

Adding *n* Numbers Work-Optimality

We reduce the number of processing elements which in turn increases the granularity of the subproblem assigned to each processing element.



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

Suppose we use p processing elements.

First each processing element locally adds its $\frac{n}{p}$ numbers in time $\Theta\left(\frac{n}{p}\right)$.

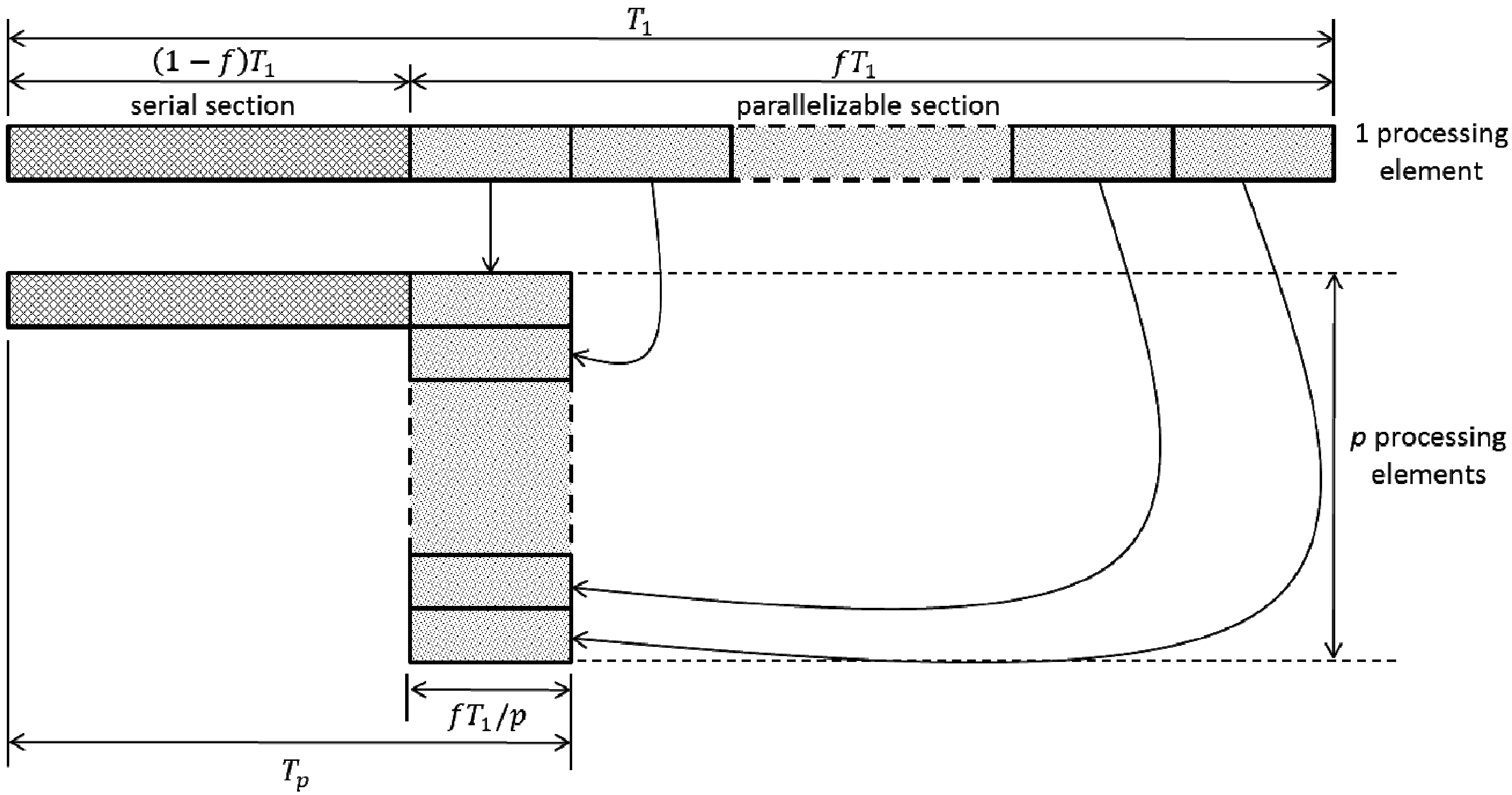
Then p processing elements adds these p partial sums in time $\Theta(\log p)$.

Thus $T_p = \Theta\left(\frac{n}{p} + \log p\right)$, and $T_s = \Theta(n)$.

So the algorithm is work-optimal provided $n = \Omega(p \log p)$.

Scaling Laws

Scaling of Parallel Algorithms (Amdahl's Law)



Suppose only a fraction *f* of a computation can be parallelized.

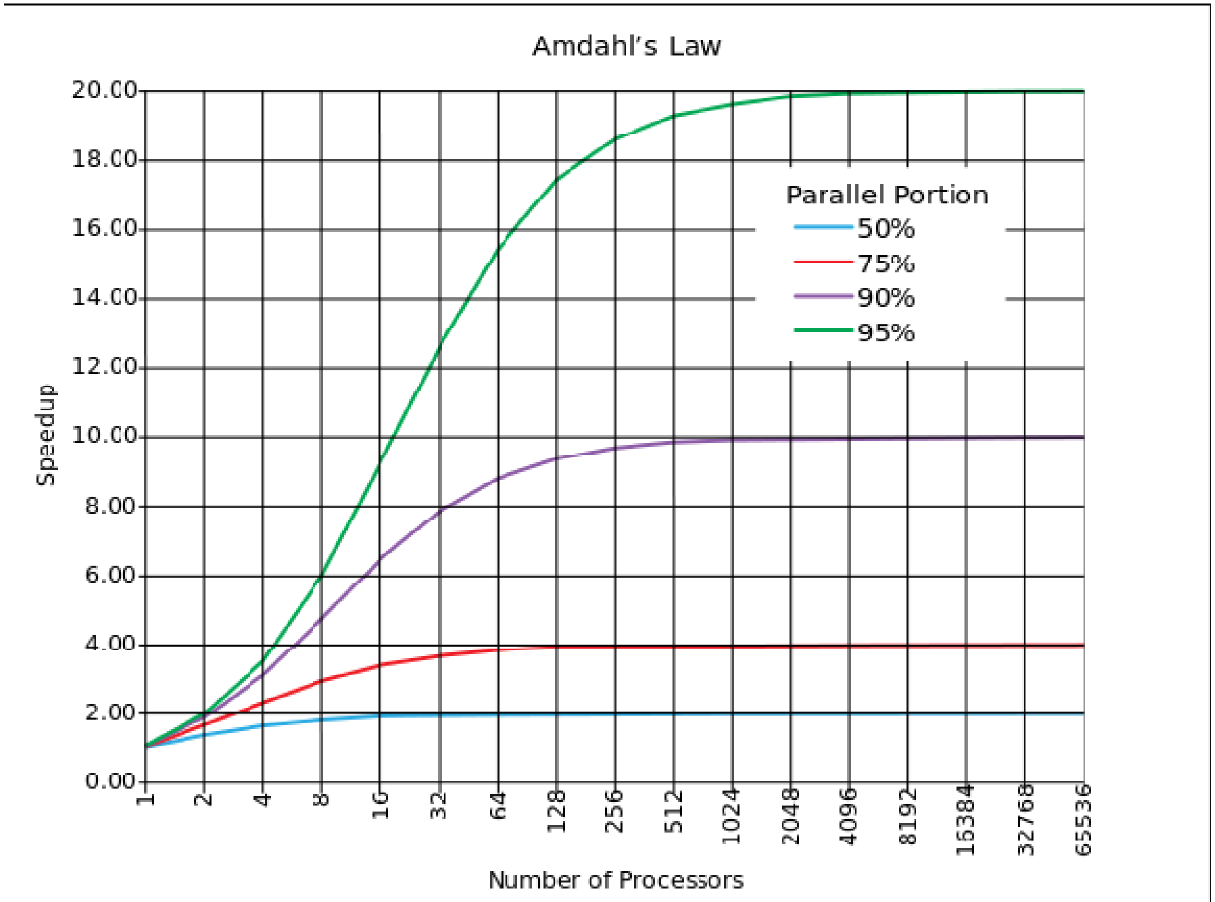
Then parallel running time, $T_p \geq (1-f)T_1 + f\,\frac{T_1}{p}$

Speedup, $S_p = \frac{T_1}{T_p} \leq \frac{p}{f+(1-f)p} = \frac{1}{(1-f)+\frac{f}{p}} \leq \frac{1}{1-f}$

Scaling of Parallel Algorithms (Amdahl's Law)

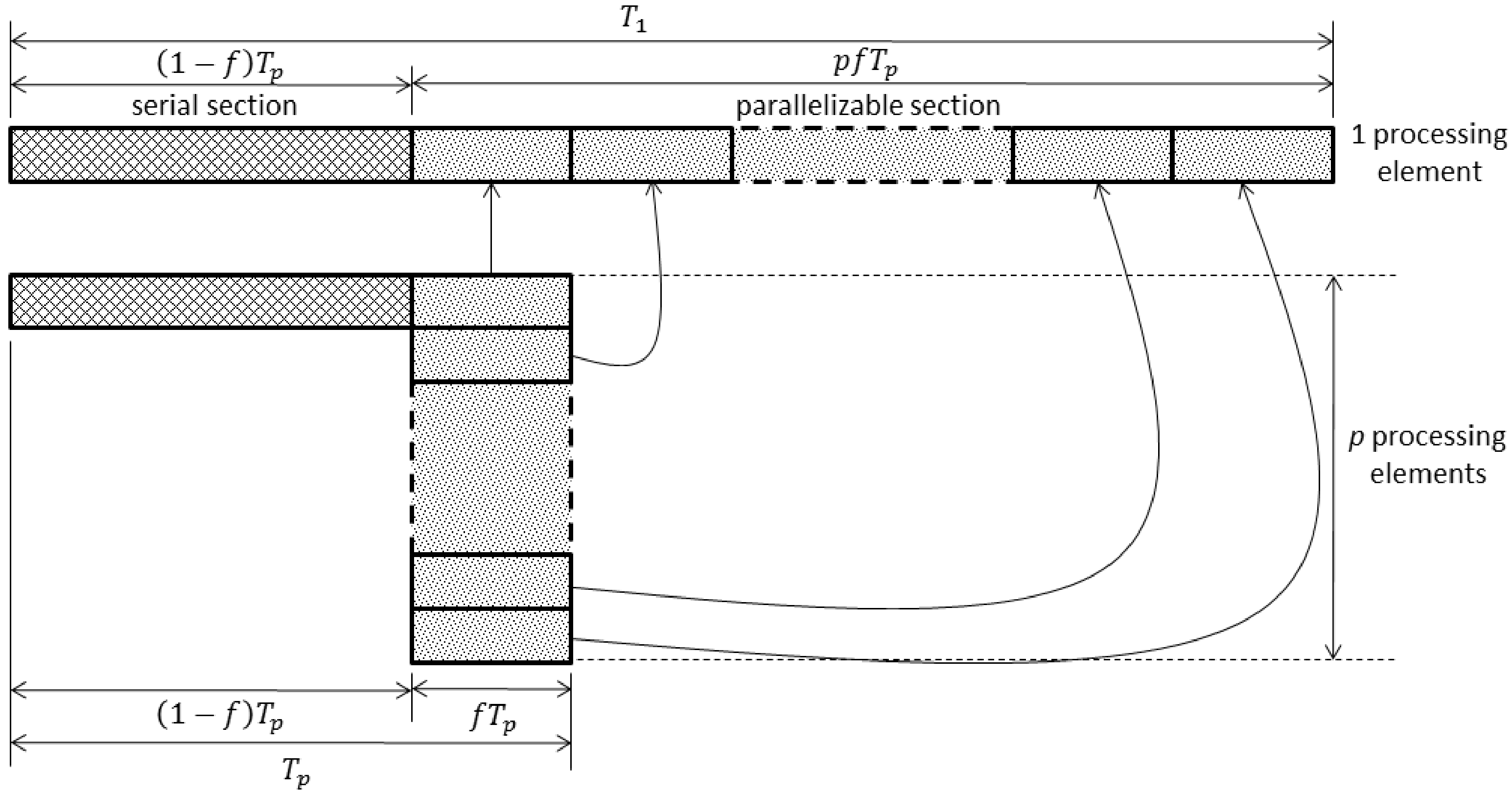
Suppose only a fraction f of a computation can be parallelized.

Speedup, $S_p = \frac{T_1}{T_p} \le \frac{1}{(1-f)+\frac{f}{p}} \le \frac{1}{1-f}$



Source: Wikipedia

Scaling of Parallel Algorithms (Gustafson-Barsis' Law)



Suppose only a fraction f of a computation was parallelized.

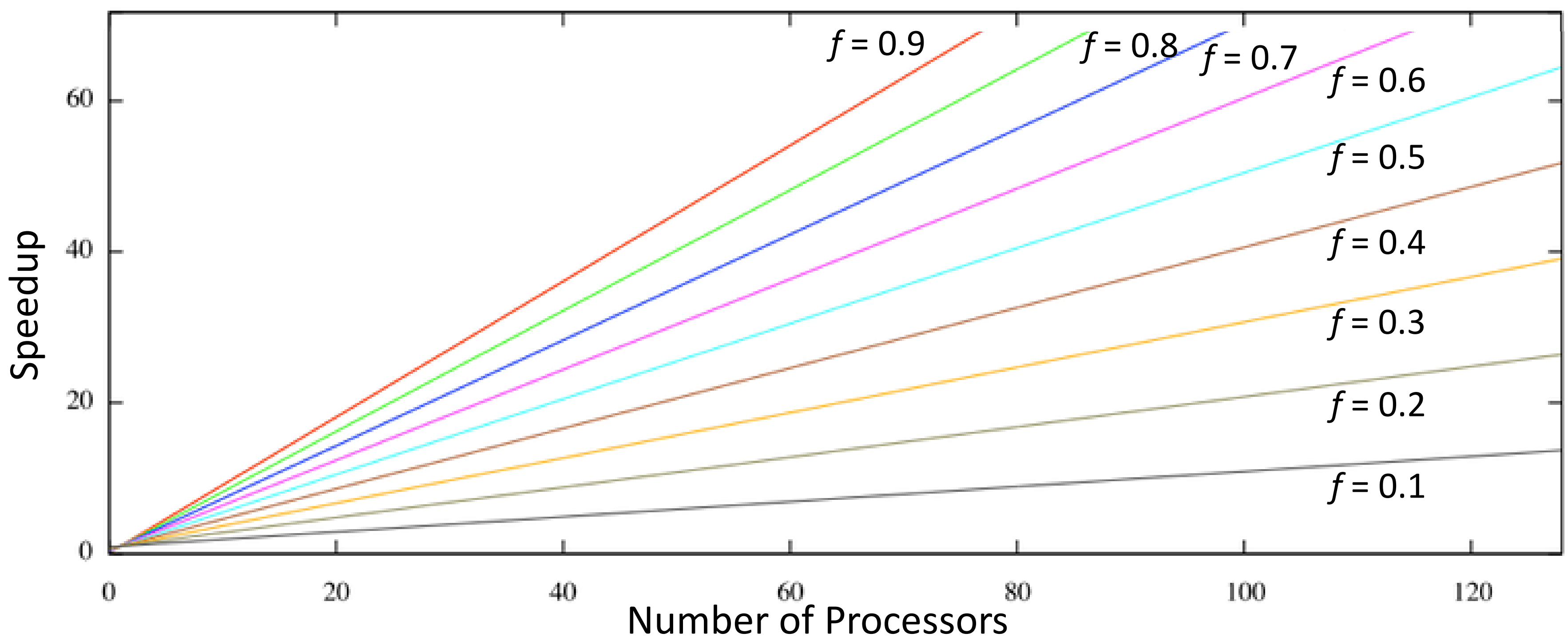
Then serial running time, $T_1 = (1-f)T_p + pfT_p$

Speedup, $S_p = \frac{T_1}{T_p} = \frac{(1-f)T_p + pfT_p}{T_p} = 1 + (p-1)f$

Scaling of Parallel Algorithms (Gustafson-Barsis' Law)

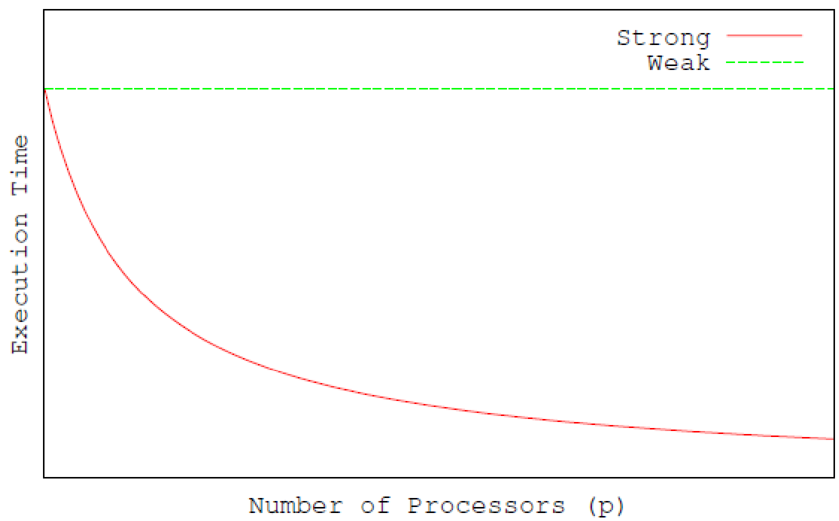
Suppose only a fraction f of a computation was parallelized.

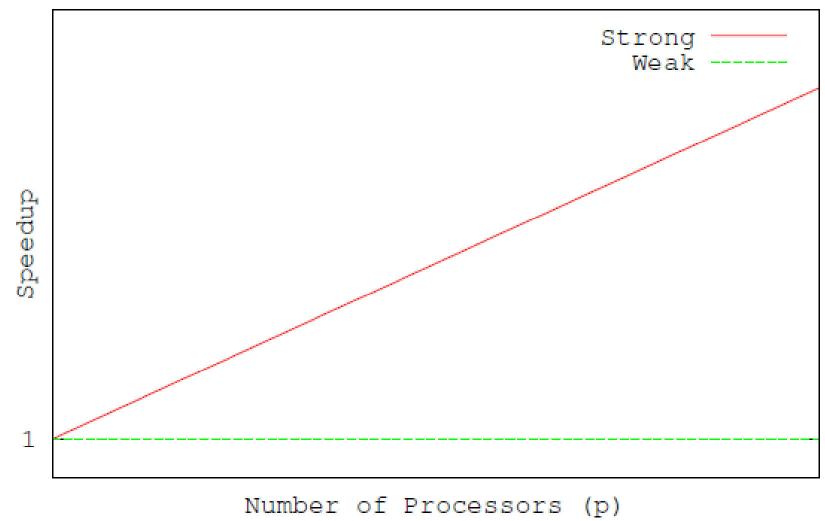
Speedup, $S_p = \frac{T}{T_p} \leq \frac{T_1}{T_p} = \frac{(1-f)T_p + pfT_p}{T_p} = 1 + (p-1)f$



Source: Wikipedia

Strong Scaling vs. Weak Scaling





Source: Martha Kim, Columbia University

Strong Scaling

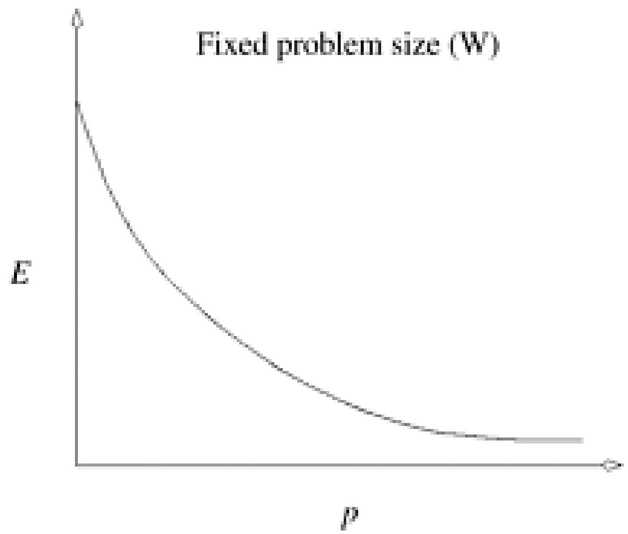
How T_p (or S_p) varies with p when the problem size is fixed.

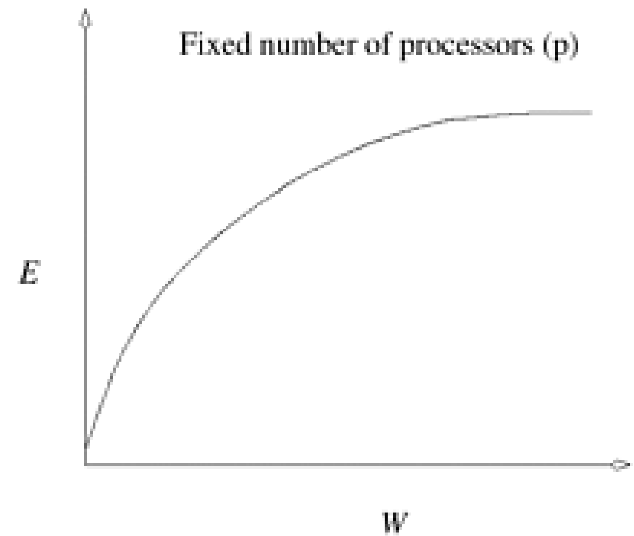
Weak Scaling

How T_p (or S_p) varies with p when the problem size per processing element is fixed.

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

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.

Efficiency, $E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} = \frac{T_1}{T_1 + T_O} = \frac{1}{1 + \frac{T_O}{T_1}}$

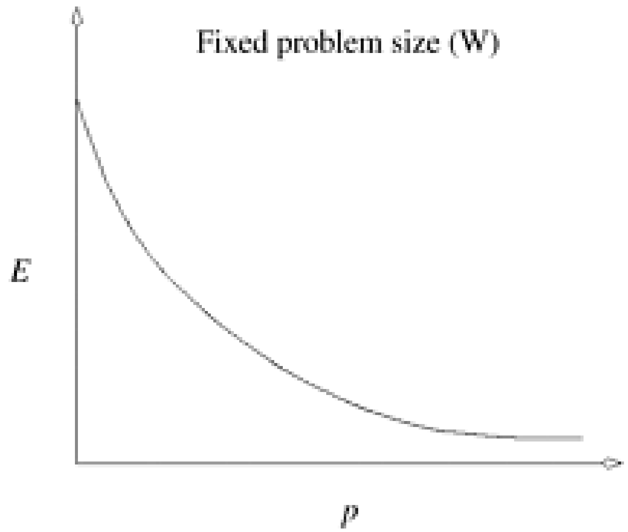
Observe that if the problem size is fixed, T_O increases with p. (why?)

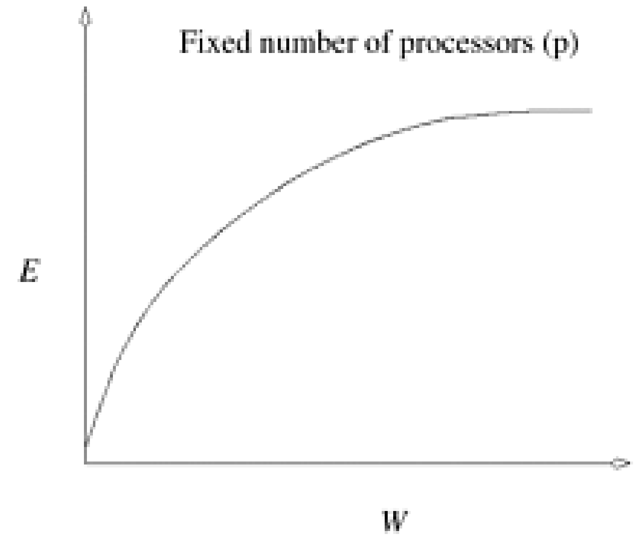
So, E_p drops as *p* increases.

On the other hand, for many algorithms T_O grows sublinearly w.r.t. T_1.

For such algorithms E_p can be kept fixed by increasing the problem size and p simultaneously.

Scalable Parallel Algorithms





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

Efficiency, $E_p = \frac{S_p}{p} = \frac{T}{pT_p} = \frac{T}{T+T_O} = \frac{1}{1+\frac{T_O}{T}}$

Observe that if the problem size is fixed, T_O increases with *p*. (why?)

So E_p drops as *p* increases.

On the other hand, for many algorithms T_O grows sublinearly w.r.t. *T*.

For such algorithms E_p can be kept fixed by increasing the problem size and *p* simultaneously.

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

The Isoefficiency Function

For a given problem, we define *problem size W* as the number of basic computation steps in the fastest sequential algorithm that solves the problem on a serial machine.

Thus $W = T$.

We have, $E_p = \frac{S_p}{p} = \frac{T}{pT_p} = \frac{T}{T+T_O} = \frac{1}{1+\frac{T_O}{T}} = \frac{1}{1+\frac{T_O(W,p)}{W}}$

Rearranging, $W = \frac{E_p}{1-E_p} T_O(W,p) = KT_O(W,p)$, where $K = \frac{E_p}{1-E_p}$

We have already seen how to obtain the isoefficiency function for adding *n* numbers using *p* processing elements.

Isoefficiency for Complex Overhead Functions

Suppose, $T_O = p^{3/2} + p^{3/4}W^{3/4}$.

We balance W against each term of T_O, and the component of T_O that requires W to grow at the highest rate w.r.t. p gives the overall asymptotic isoefficiency function for the algorithm.

Using only the 1st term, $W = Kp^{3/2}$

Using only the 2nd term, $W = K^4p^3$

Hence, the overall isoefficiency function is $\Theta(p^3)$.