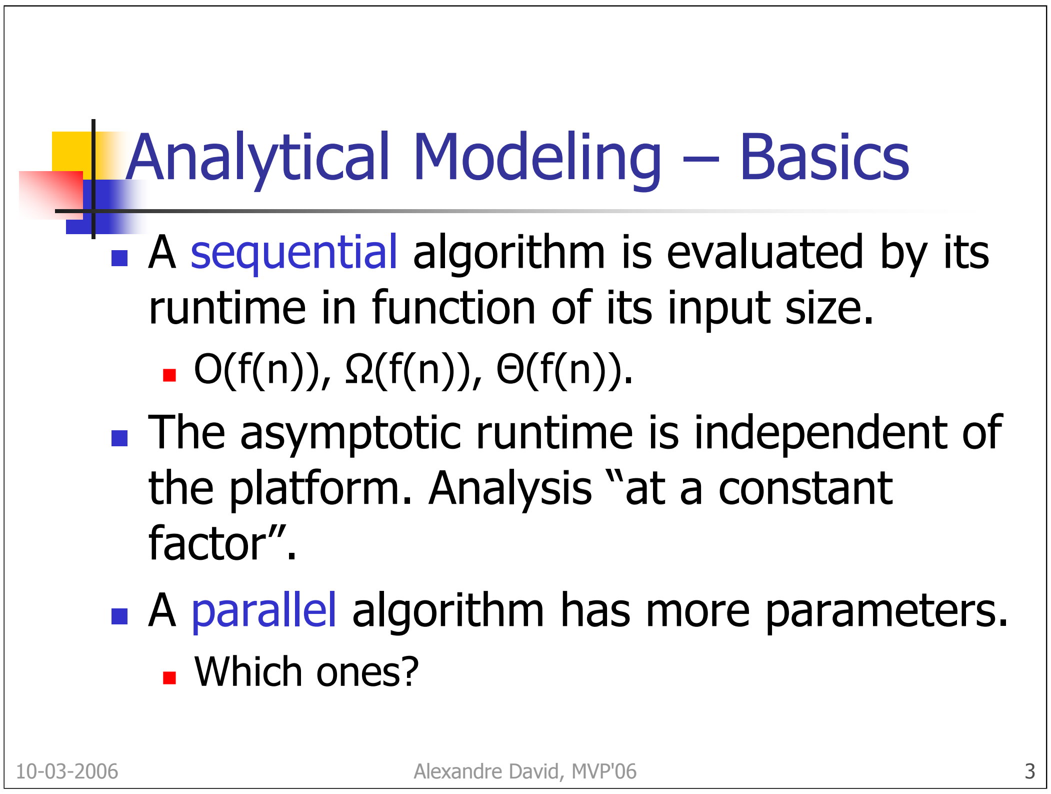
# Analytical Modeling of Parallel Programs (Chapter 5)

Alexandre David
B2-206

## Topic Overview

- Sources of overhead in parallel programs.
- Performance metrics for parallel systems.
- Effect of granularity on performance.
- Scalability of parallel systems.
- Minimum execution time and minimum cost-optimal execution time.
- Asymptotic analysis of parallel programs.
- Other scalability metrics.

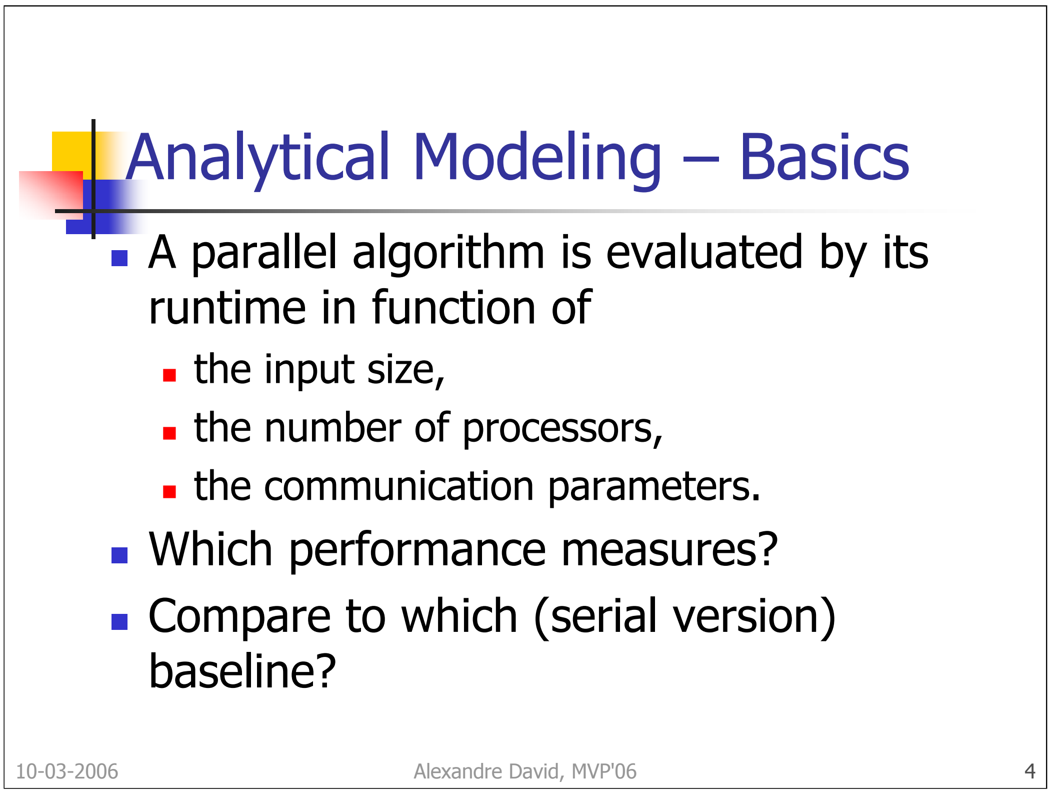


## Analytical Modeling – Basics

- A sequential algorithm is evaluated by its runtime in function of its input size.
  - O(f(n)), Ω(f(n)), Θ(f(n)).
- The asymptotic runtime is independent of the platform. Analysis "at a constant factor".
- A parallel algorithm has more parameters.
  - Which ones?

10-03-2006 Alexandre David, MVP'06 3

Reminder O-notation, Ω-notation, Θ-notation.



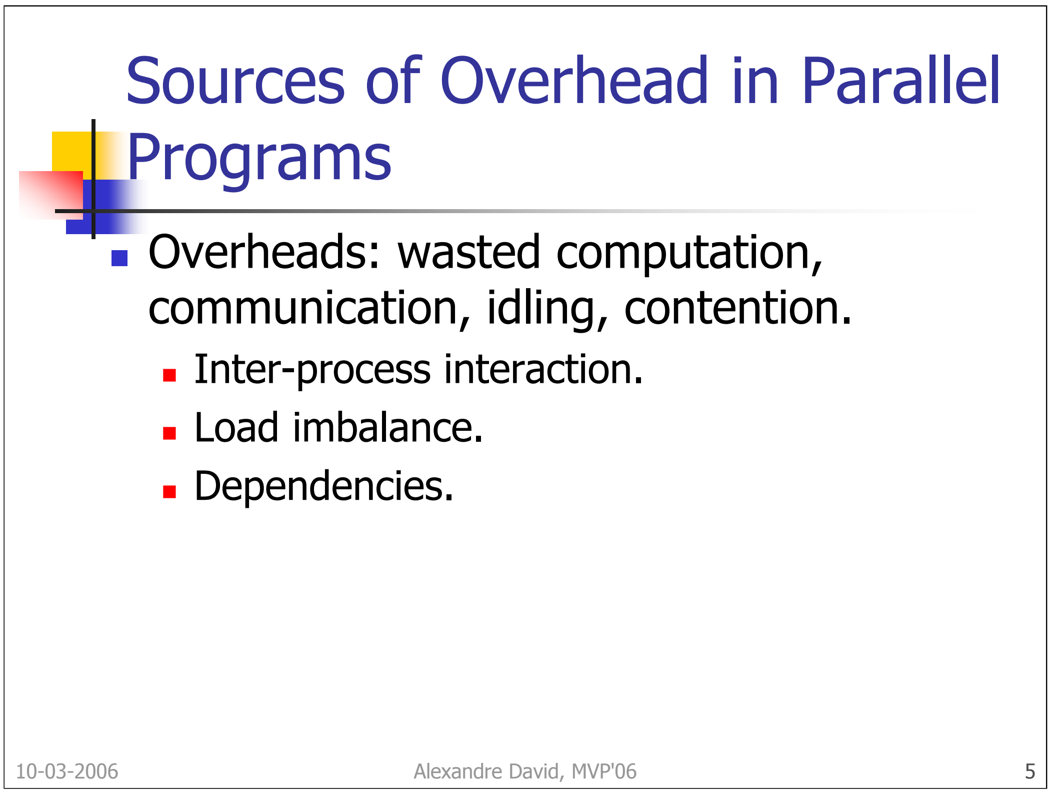
## Analytical Modeling – Basics

- A parallel algorithm is evaluated by its runtime in function of
  - the input size,
  - the number of processors,
  - the communication parameters.
- Which performance measures?
- Compare to which (serial version) baseline?

Note: The underlying RAM model may play a role, keep in mind that they are equivalent and the more powerful models can be emulated by the weaker ones in polynomial time.

Parallel system = parallel algorithm + underlying platform, which what we analyze.

Performance measures: time obvious, but how does it scale?

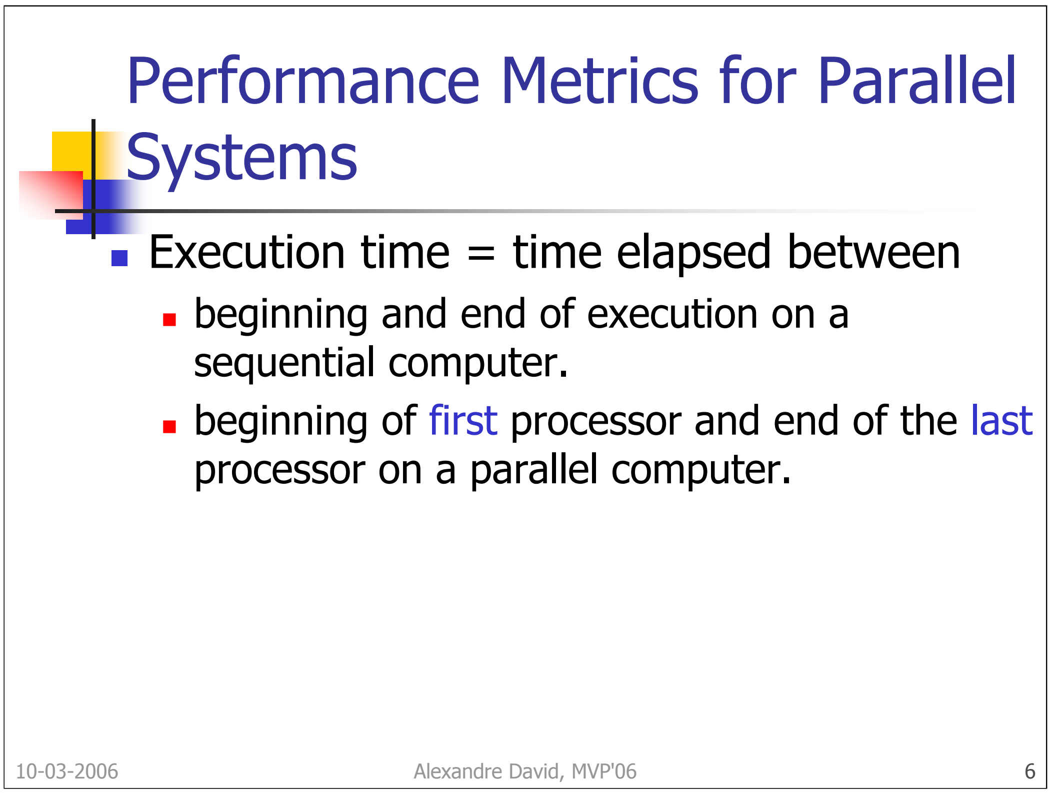


## Sources of Overhead in Parallel Programs

- Overheads: wasted computation, communication, idling, contention.
  - Inter-process interaction.
  - Load imbalance.
  - Dependencies.

Naïve question: Shouldn't my program run twice faster if I use two processors?

Different sources of overhead: We have already seen them. Wasted computation = excess computation (speculative execution for example, or duplicate work).

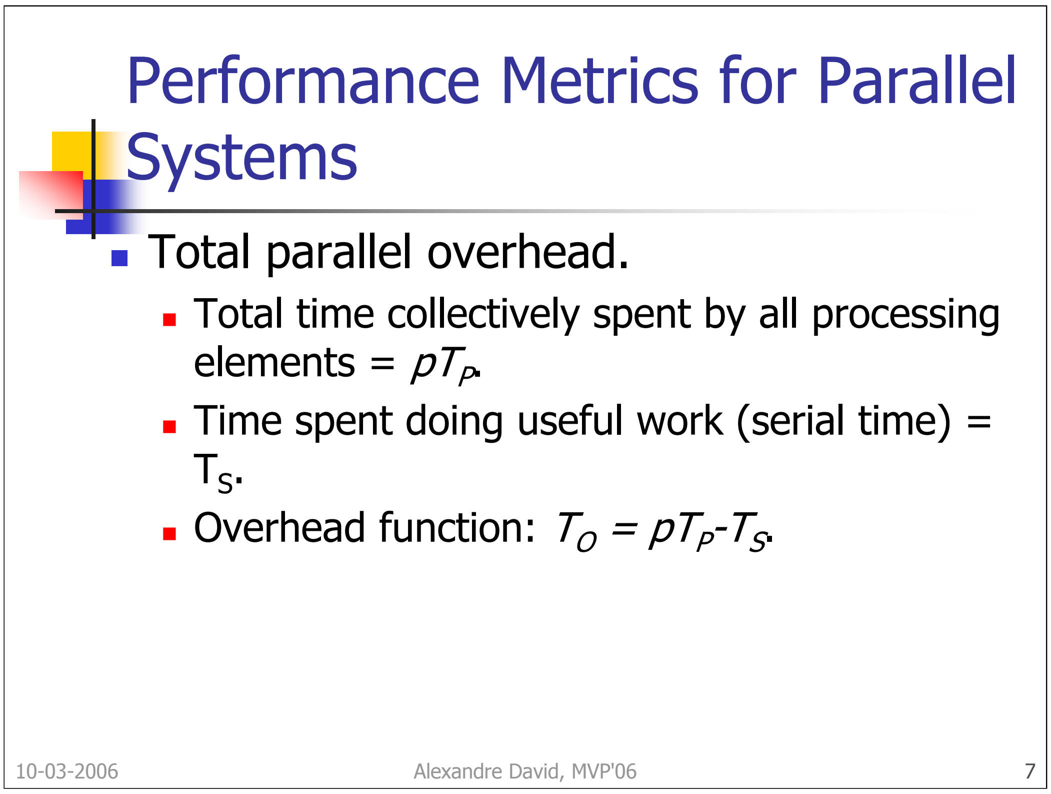


## Performance Metrics for Parallel Systems

- Execution time = time elapsed between
  - beginning and end of execution on a sequential computer.
  - beginning of first processor and end of the last processor on a parallel computer.

Intuitive for sequential programs but be careful for parallel programs.

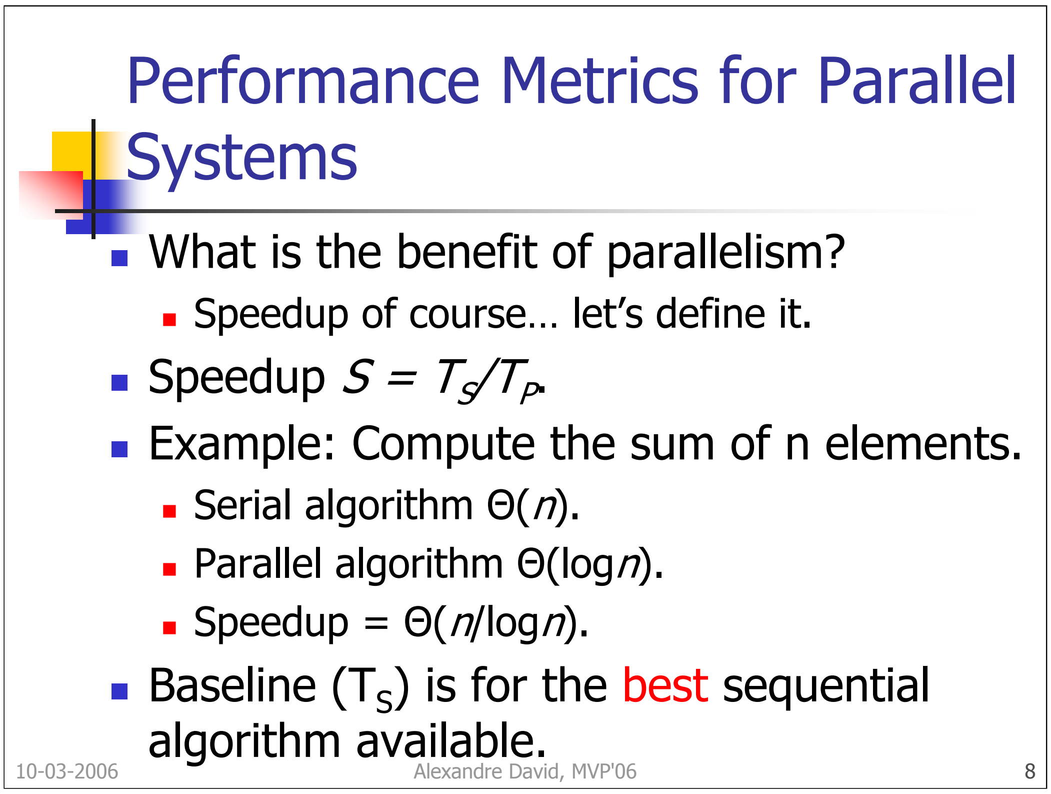
Execution time denoted $T_P$.



## Performance Metrics for Parallel Systems

- Total parallel overhead.
  - Total time collectively spent by all processing elements = $pT_P$.
  - Time spent doing useful work (serial time) = $T_S$.
  - Overhead function: $T_O = pT_P - T_S$.

Quantitative way of measuring overheads, this metric contains all kinds of overheads.



## Performance Metrics for Parallel Systems

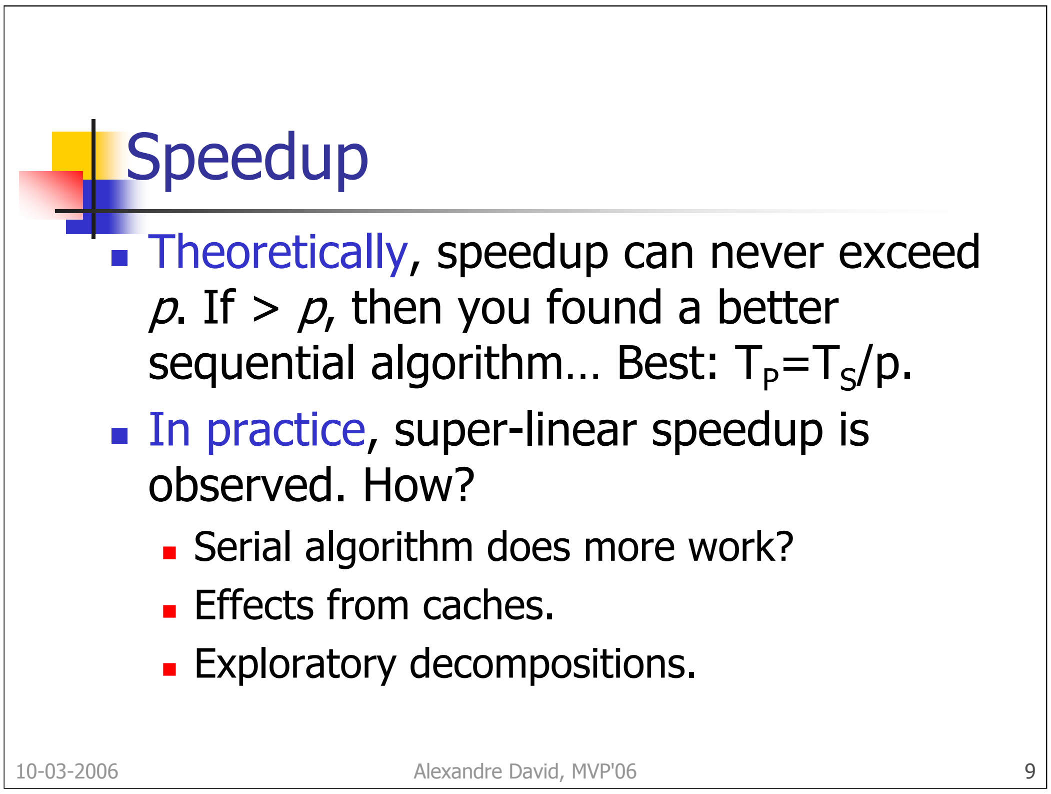
- What is the benefit of parallelism?
  - Speedup of course… let's define it.
- Speedup $S = T_S/T_P$.
- Example: Compute the sum of n elements.
  - Serial algorithm $\Theta(n)$.
  - Parallel algorithm $\Theta(\log n)$.
  - Speedup = $\Theta(n/\log n)$.
- Baseline ($T_S$) is for the best sequential algorithm available.

10-03-2006 Alexandre David, MVP'06 8

And by the way speedup is one benefit, you can find others like simpler hardware architectures (several simple CPUs better than one big complex) and heat issues.

Adding 2 elements and communication time are constants.

Question: Compare to what? Which $T_S$ to take? All the sequential algorithm are not equally parallelizable and do not perform the same.

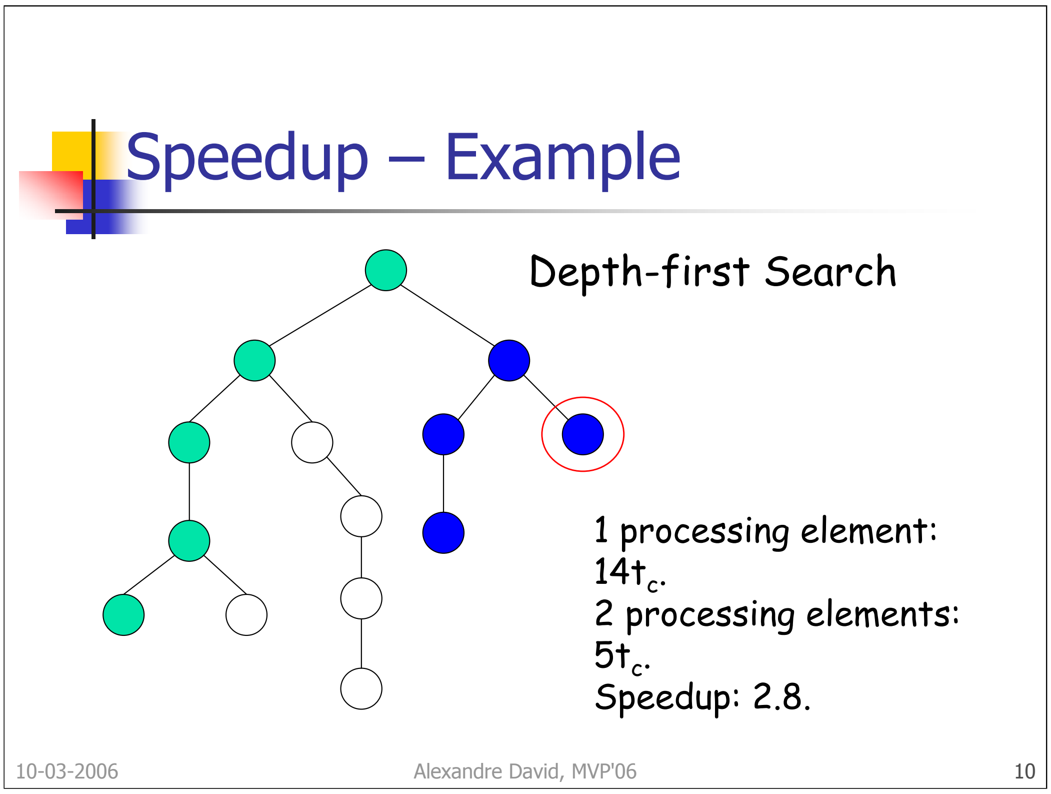


## Speedup

- Theoretically, speedup can never exceed *p*. If > *p*, then you found a better sequential algorithm… Best: $T_P=T_S/p$.
- In practice, super-linear speedup is observed. How?
  - Serial algorithm does more work?
  - Effects from caches.
  - Exploratory decompositions.

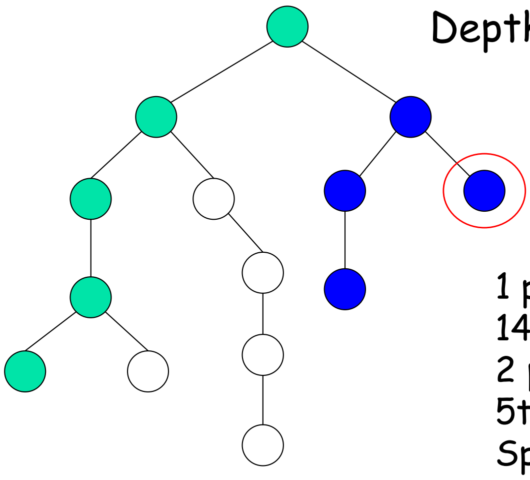
Serial algorithm may do more work compared to its parallel counterpart due to features in parallel hardware.

Caches: aggregate amount of caches is larger, so “more data can fit in the cache”, if the data is partitioned appropriately.



## Speedup – Example

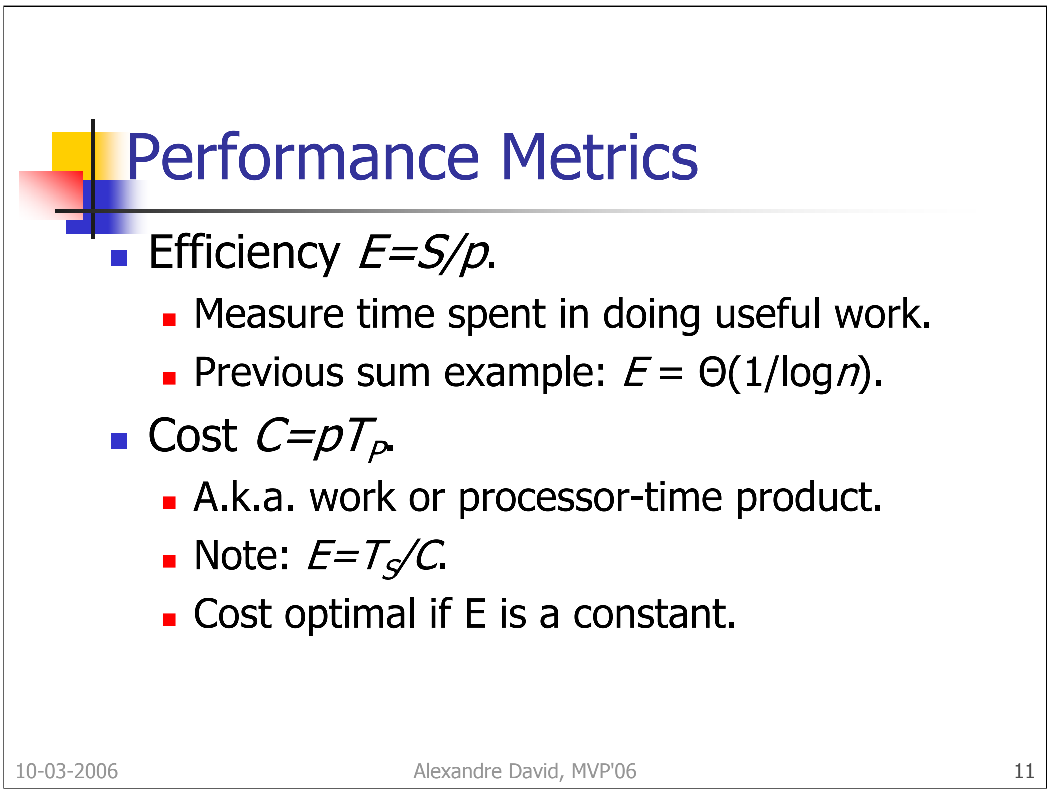
Depth-first Search



1 processing element: $14t_c$.
2 processing elements: $5t_c$.
Speedup: 2.8.

10-03-2006 Alexandre David, MVP'06 10

The works performed by the serial and the parallel algorithms are different. If we simulate 2 processes on the same processing element then we get a better serial algorithm for this **instance** of the problem but we cannot generalize it to all instances. Here the work done by the different algorithms depends on the input, i.e., the location of the solution in the search tree.



## Performance Metrics

- Efficiency $E=S/p$.
  - Measure time spent in doing useful work.
  - Previous sum example: $E = \Theta(1/\log n)$.
- Cost $C=pT_P$.
  - A.k.a. work or processor-time product.
  - Note: $E=T_S/C$.
  - Cost optimal if E is a constant.

Speedup/number of processing elements. Ideally it is 1 with S = p.

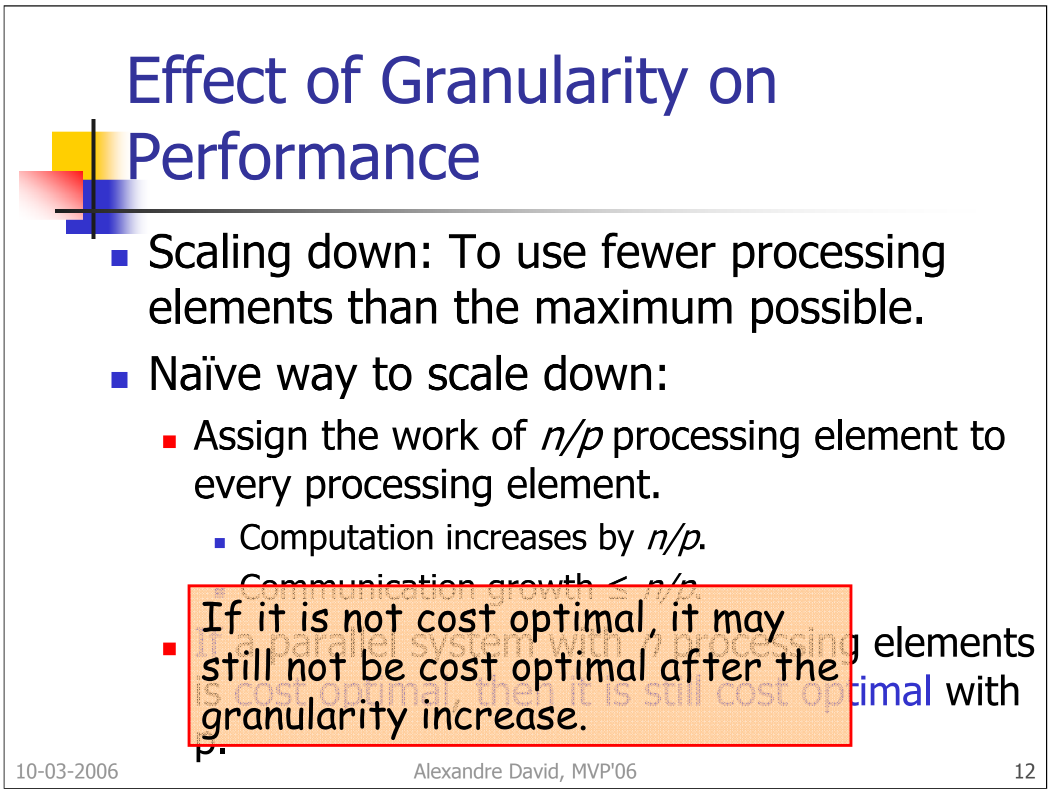
Comment for 1/logn: efficiency (and speedup too) goes down with n. If the problem size increases you win less by using more processors.

Check yourself edge detection example in the book.

Cost = parallel runtime * number of processing elements = total time spent for all processing elements.

C is a constant = $T_S$ and $T_P$ have the same asymptotic growth function (at a constant factor).

Related to previous lecture on Brent’s scheduling principle.



## Effect of Granularity on Performance

- Scaling down: To use fewer processing elements than the maximum possible.
- Naïve way to scale down:
  - Assign the work of *n/p* processing element to every processing element.
    - Computation increases by *n/p*.
    - Communication growth ≤ *n/p*.
  - If a parallel system with *n* processing elements is cost optimal, then it is still cost optimal with *p*.

If it is not cost optimal, it may still not be cost optimal after the granularity increase.

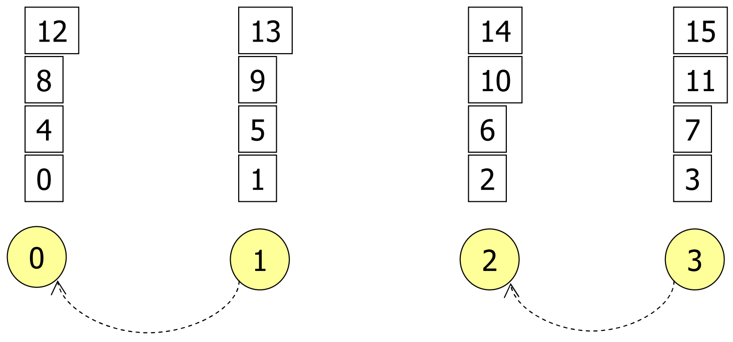
10-03-2006 Alexandre David, MVP'06 12

Communication growth bounded if the mapping is appropriate.

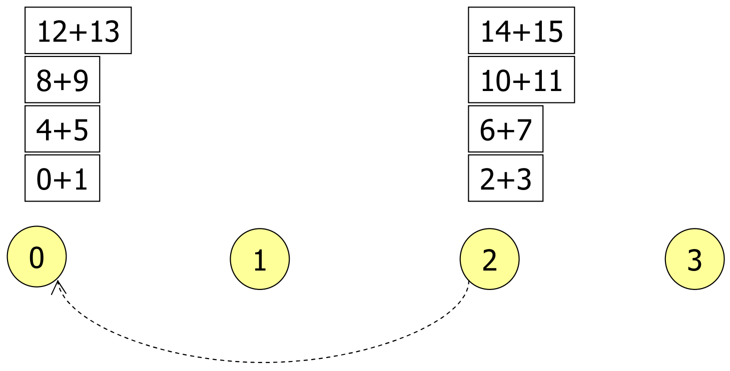
Recall Brent's scheduling algorithm: Re-schedule tasks on processes. It doesn't do miracles, it's only a re-scheduling algorithm.

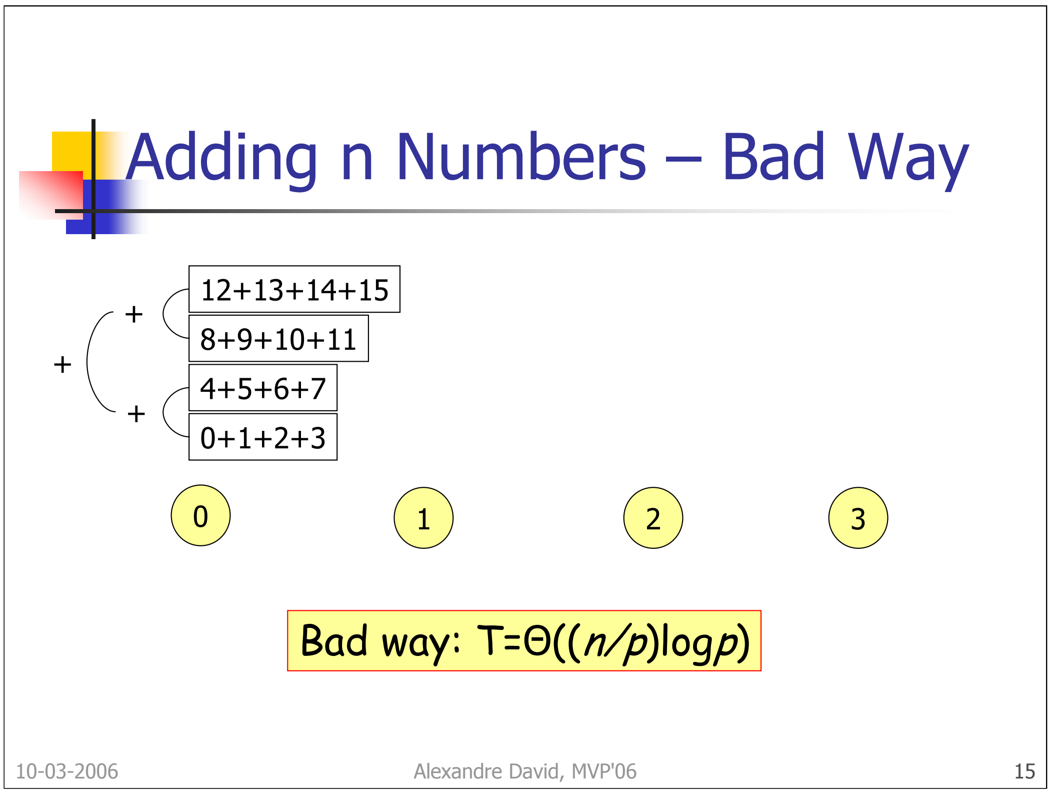
Reason for improvement in increasing the granularity (coarse grained vs. fine grained): Decrease of global communication (instead of growing with *n*, it should grow with *n/p*) because tasks mapped on the same process communicate together without overhead.

## Adding n Numbers – Bad Way



## Adding n Numbers – Bad Way





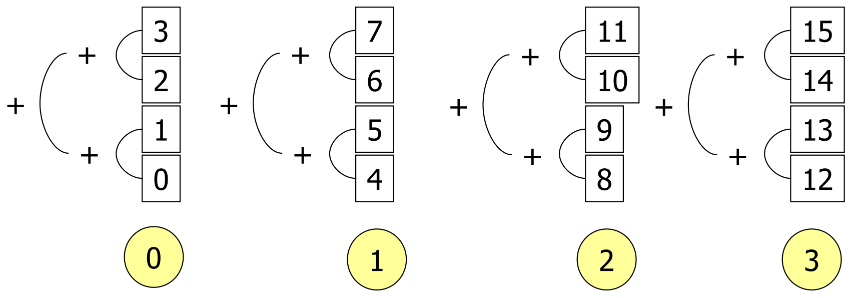
## Adding n Numbers – Bad Way

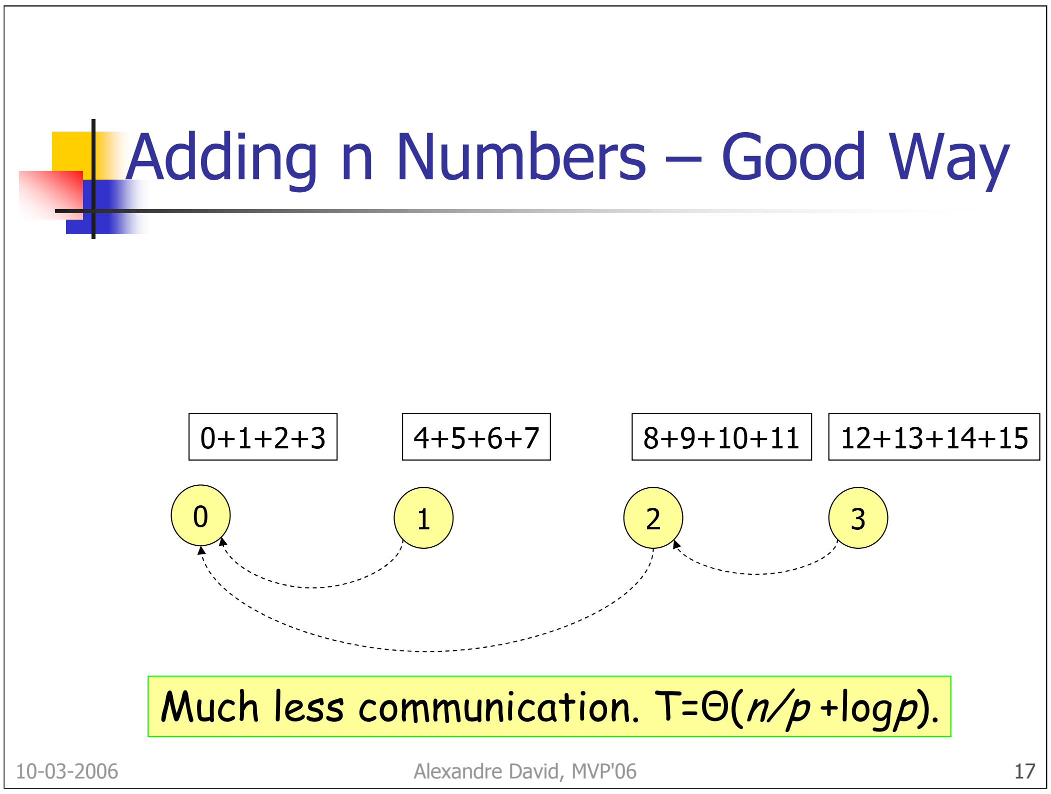
+
+
+
12+13+14+15
8+9+10+11
4+5+6+7
0+1+2+3
0
1
2
3

Bad way: T=Θ((*n*/*p*)log*p*)

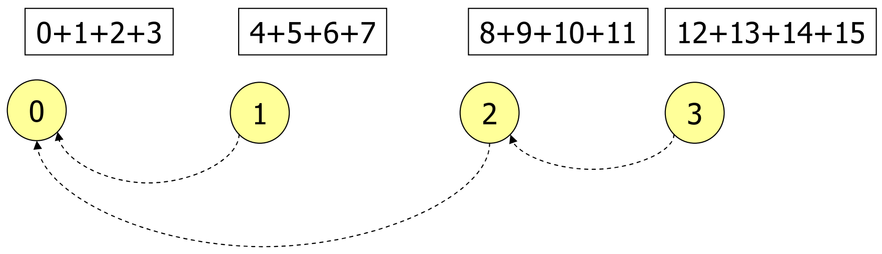
Incrementing the granularity does not improve compared to log*n*. We need to distribute better.

## Adding n Numbers – Good Way





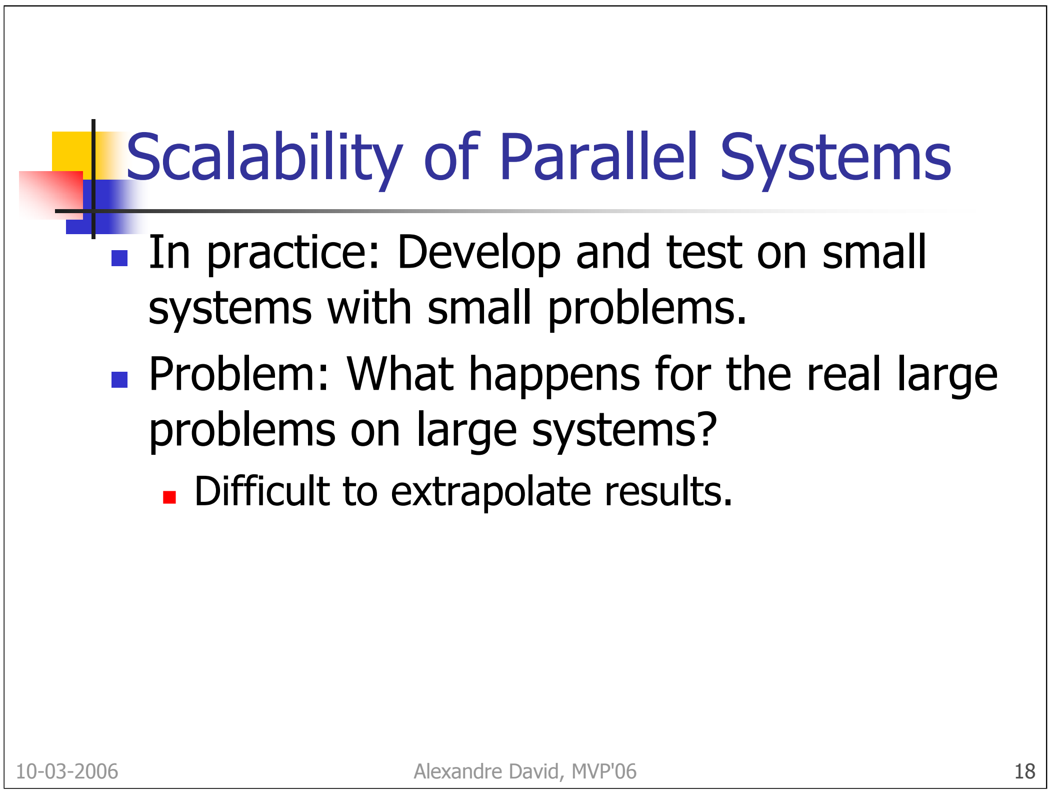
## Adding n Numbers – Good Way



Much less communication. T=Θ(*n*/*p* +log*p*).

10-03-2006 Alexandre David, MVP'06 17

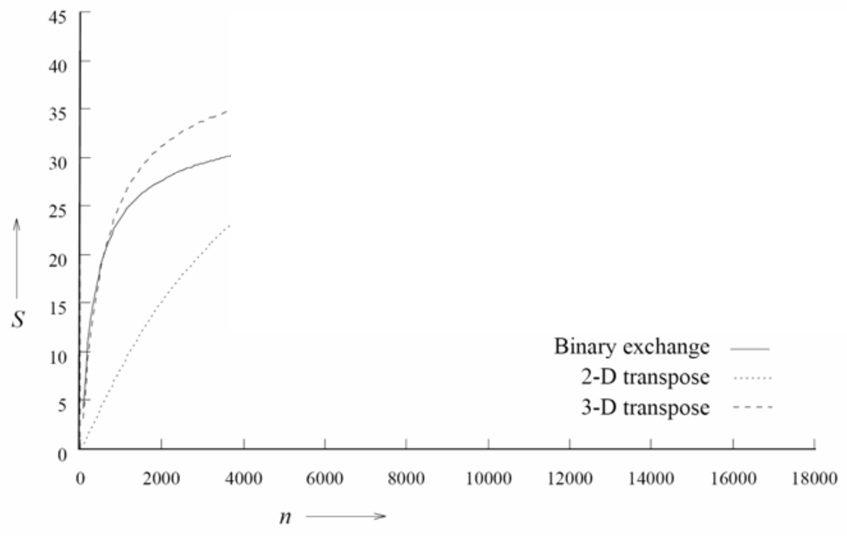
Is it optimal? As long as n=Ω(*p*log*p*), the cost is Θ(n), which is the same as the serial runtime.



## Scalability of Parallel Systems

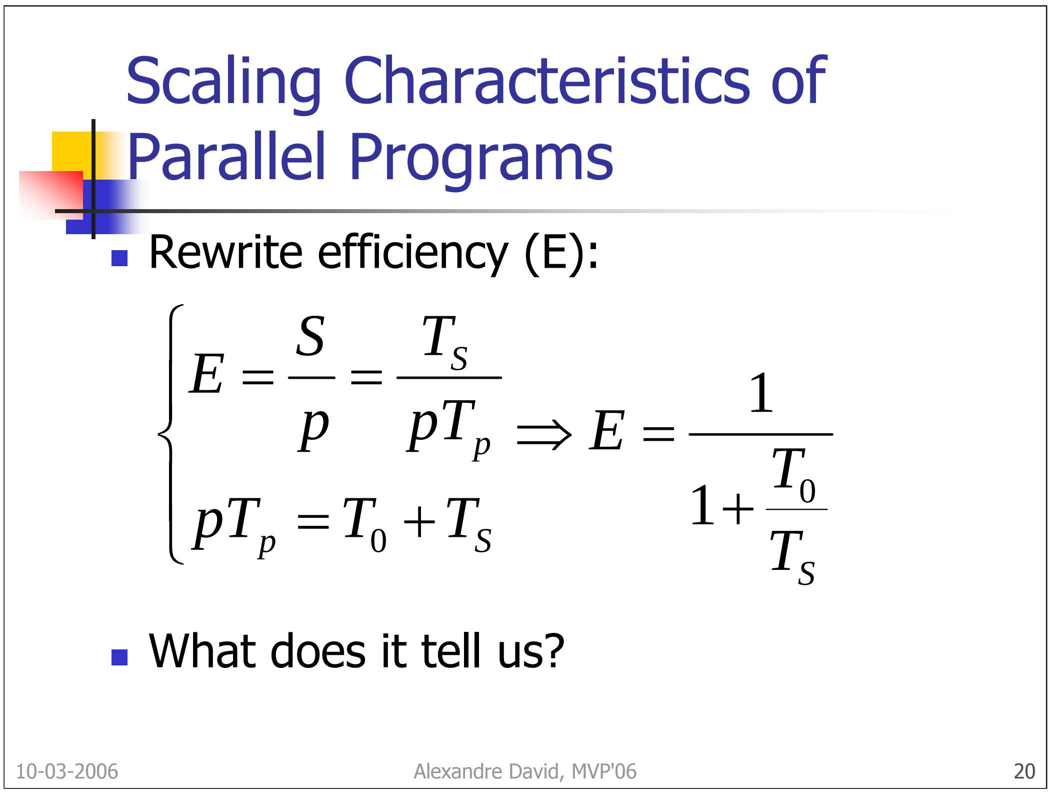
- In practice: Develop and test on small systems with small problems.
- Problem: What happens for the real large problems on large systems?
  - Difficult to extrapolate results.

## Problem with Extrapolation



19

Problem: It’s always like this and it’s always difficult to predict. You can fix the size of the problem and vary the number of processors, it will be similar.



## Scaling Characteristics of Parallel Programs

- Rewrite efficiency (E):

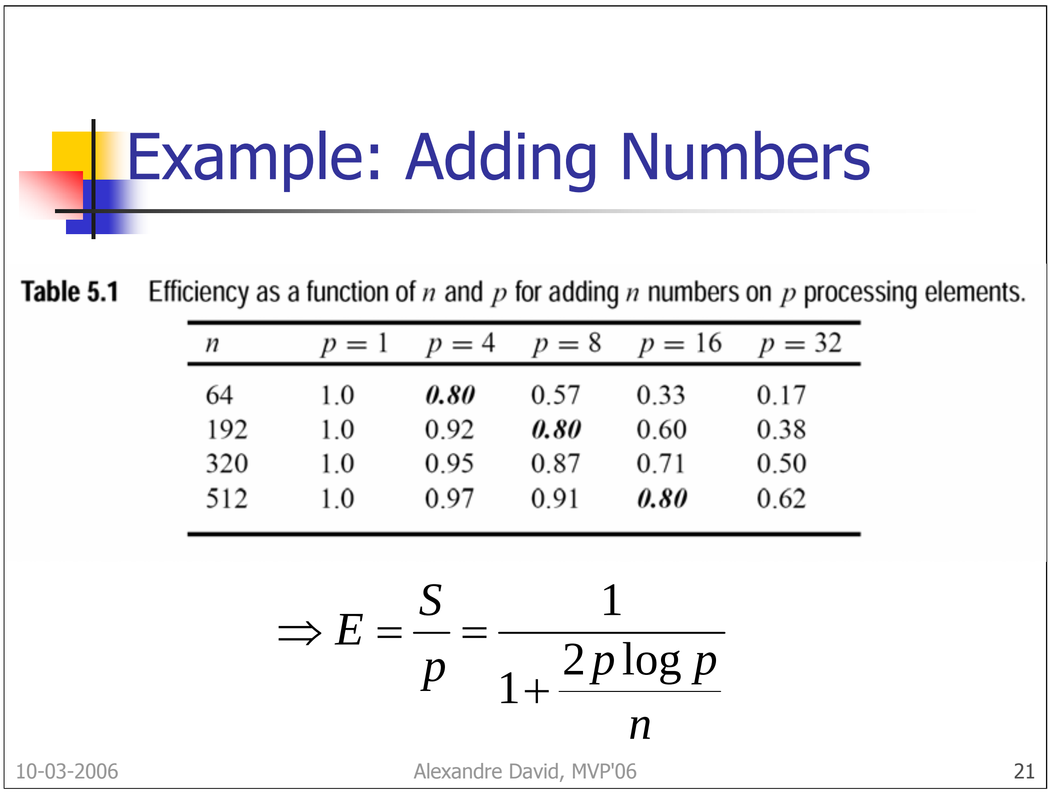
$$\begin{cases} E = \dfrac{S}{p} = \dfrac{T_S}{pT_p} \\ pT_p = T_0 + T_S \end{cases} \Rightarrow E = \frac{1}{1 + \dfrac{T_0}{T_S}}$$

- What does it tell us?

Note: The total overhead $T_0$ is an increasing function of *p*. So E decreases in function of *p*. Every program has some serial component that will limit efficiency: idling = *(p-1)*t*, increases in function of *p*. So it is **at least** linear in function of *p*.

Size fixed, $T_S$ fixed, if *p* increases, E decreases.

Number of processors fixed, $T_0$ fixed, if size increases, E increases.



## Example: Adding Numbers

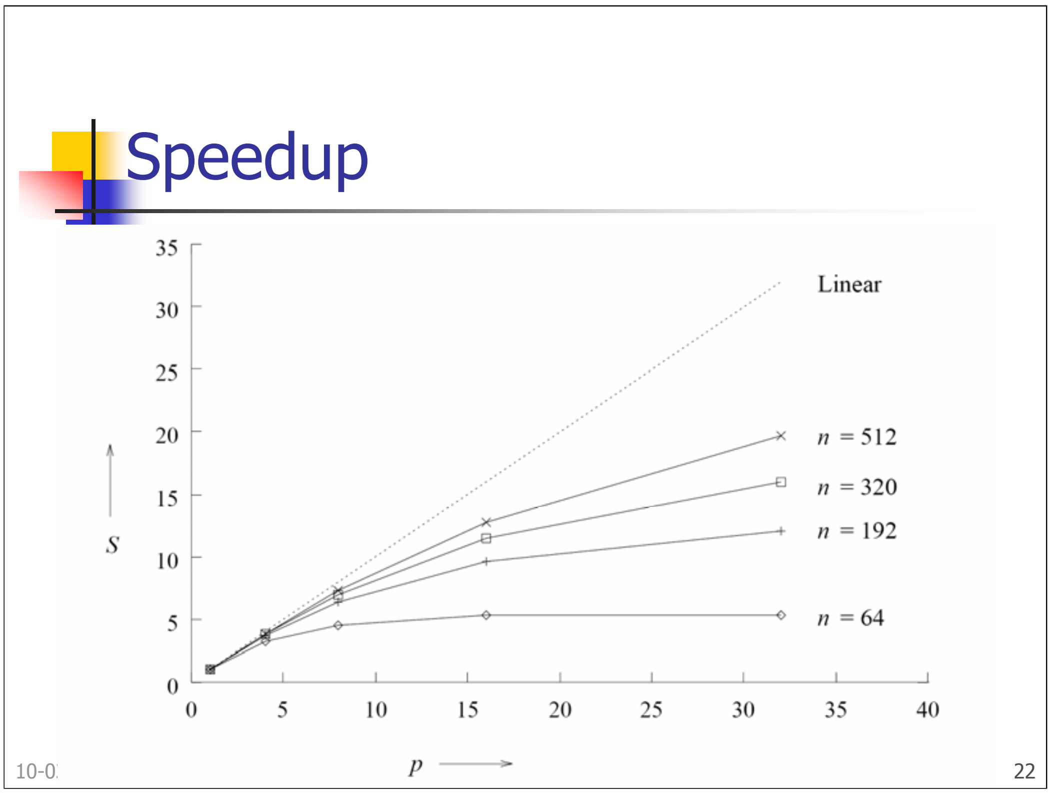
**Table 5.1** Efficiency as a function of $n$ and $p$ for adding $n$ numbers on $p$ processing elements.

| $n$ | $p = 1$ | $p = 4$ | $p = 8$ | $p = 16$ | $p = 32$ |
|---|---|---|---|---|---|
| 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 |

$$\Rightarrow E = \frac{S}{p} = \frac{1}{1 + \frac{2p\log p}{n}}$$

Since $T_S$=n here, you can see the overhead.

0.80: We can keep the same efficiency if we increase the problem size and the number of processors.

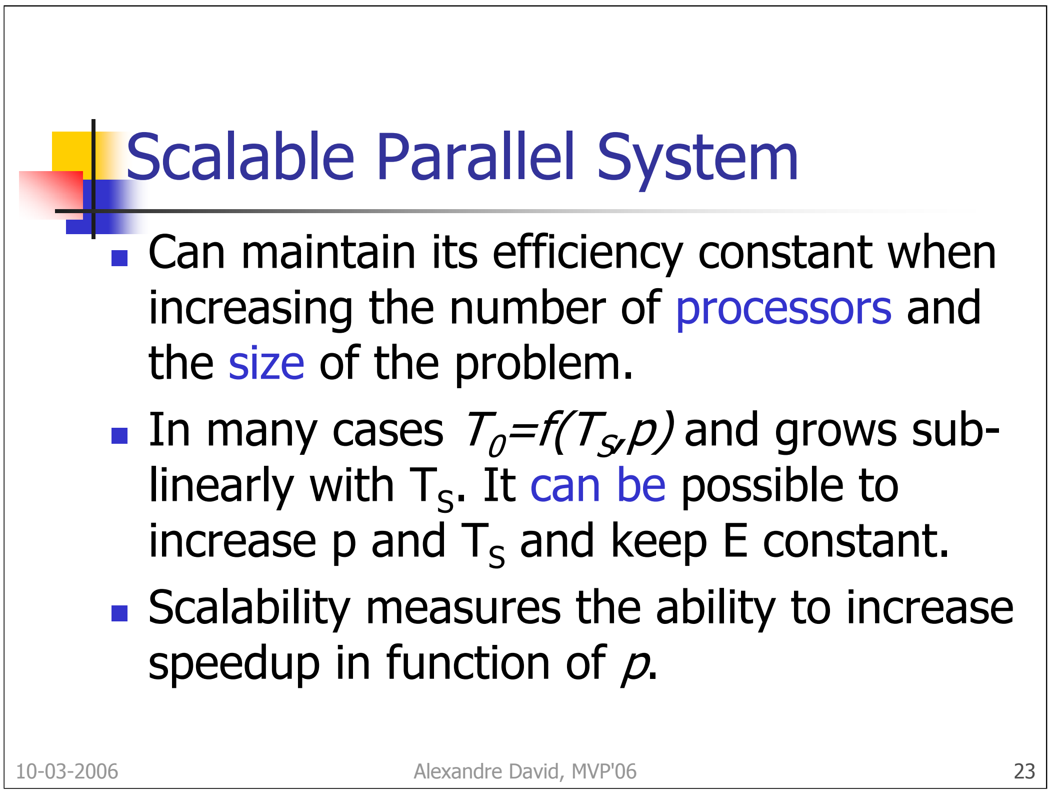


## Speedup

Fix n, efficiency decreases when p increases.

Fix p, efficiency increases when n increases.

Consequence of Amdahl’s law (exercise 5.1).



## Scalable Parallel System

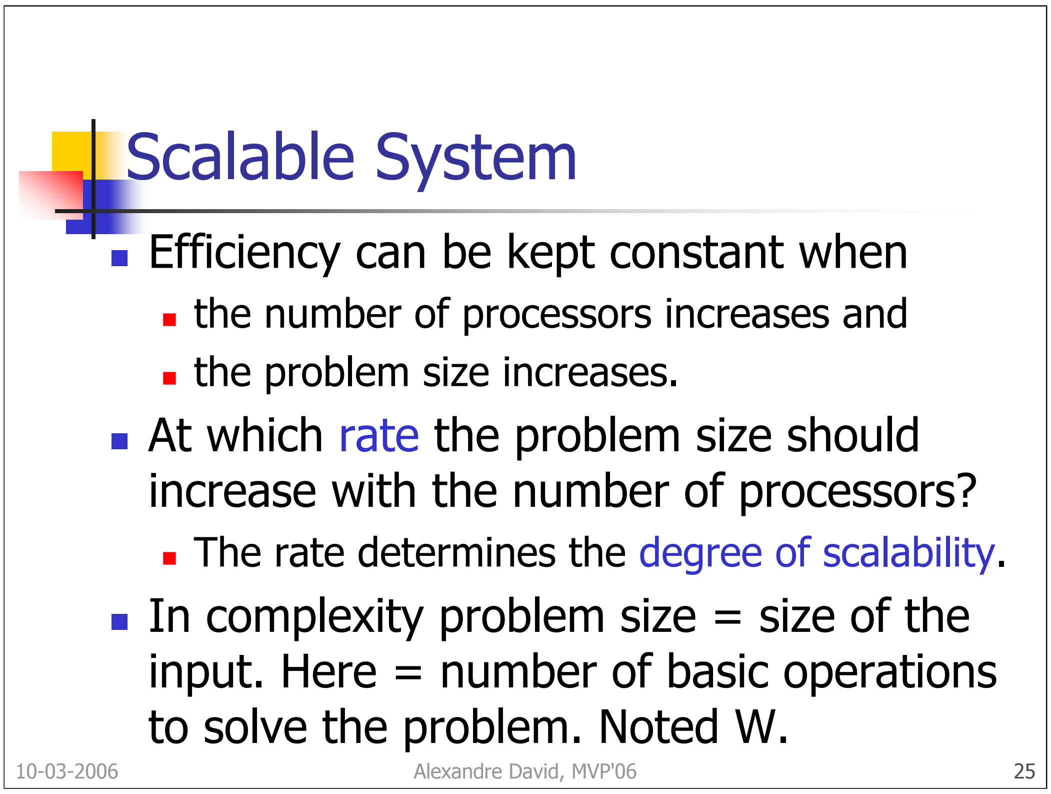
- Can maintain its efficiency constant when increasing the number of processors and the size of the problem.
- In many cases $T_0=f(T_S,p)$ and grows sub-linearly with $T_S$. It can be possible to increase p and $T_S$ and keep E constant.
- Scalability measures the ability to increase speedup in function of *p*.

10-03-2006 Alexandre David, MVP'06 23

Scalability: ability to use efficiently increasing processing power.

## Cost-Optimality

- Cost optimal parallel systems have efficiency $\Theta(1)$.
- So scalability and cost-optimality are linked.
- Adding number example: becomes cost-optimal when $n=\Omega(p \log p)$.



## Scalable System

- Efficiency can be kept constant when
  - the number of processors increases and
  - the problem size increases.
- At which rate the problem size should increase with the number of processors?
  - The rate determines the degree of scalability.
- In complexity problem size = size of the input. Here = number of basic operations to solve the problem. Noted W.

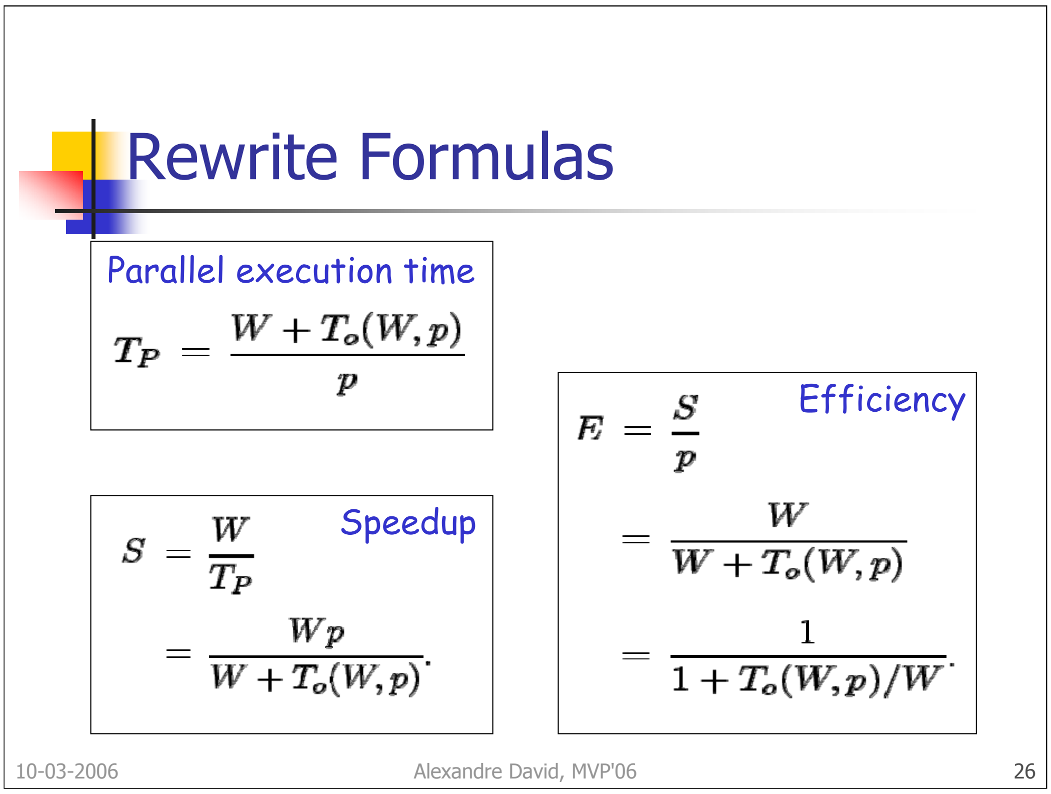
10-03-2006 Alexandre David, MVP'06 25

Note on the increase of the rate: the slower the better.

Motivation for change of definition: When doubling the problem size we wish to double the amount of computation. However, doubling the input size has very different impact on the amount of computations depending on the kind of algorithm you have.

Number of basic operations in the **best sequential algorithm**.

$W=T_S$ of the fastest known algorithm to solve the problem.



## Rewrite Formulas

Parallel execution time

$$T_P = \frac{W + T_o(W,p)}{p}$$

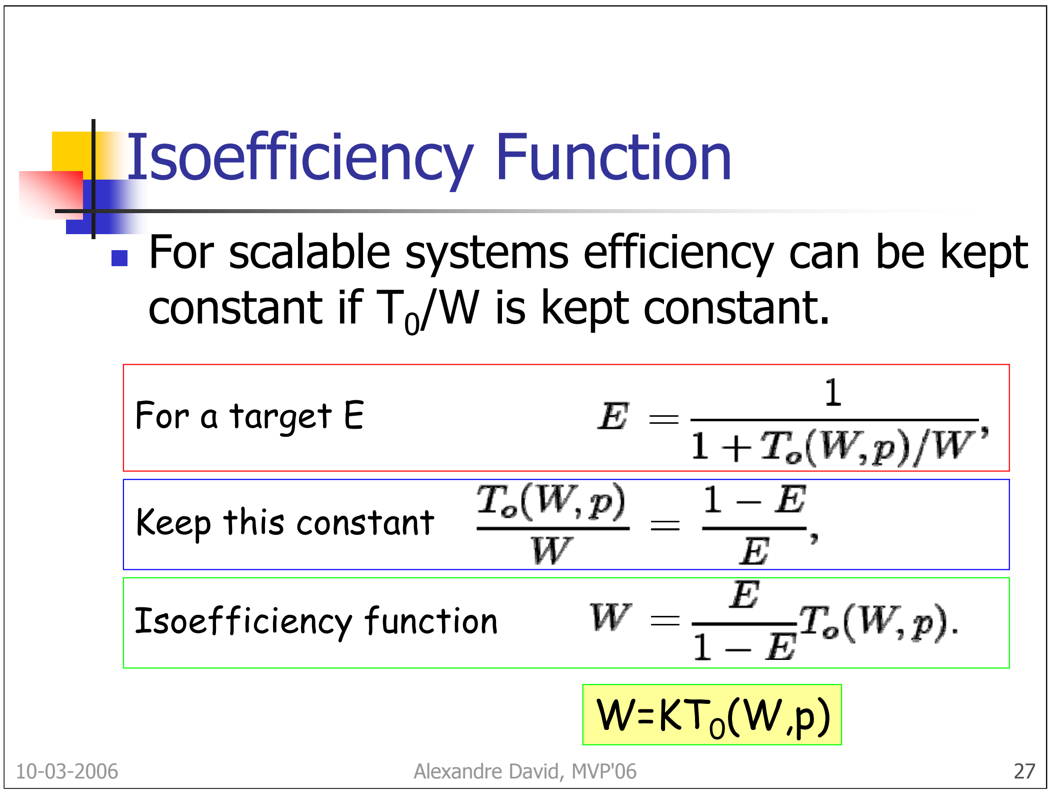
Speedup

$$S = \frac{W}{T_P} = \frac{Wp}{W + T_o(W,p)}.$$

Efficiency

$$E = \frac{S}{p} = \frac{W}{W + T_o(W,p)} = \frac{1}{1 + T_o(W,p)/W}.$$

$W=T_S$



## Isoefficiency Function

- For scalable systems efficiency can be kept constant if $T_0/W$ is kept constant.

For a target E $$E = \frac{1}{1 + T_o(W,p)/W},$$

Keep this constant $$\frac{T_o(W,p)}{W} = \frac{1-E}{E},$$

Isoefficiency function $$W = \frac{E}{1-E} T_o(W,p).$$

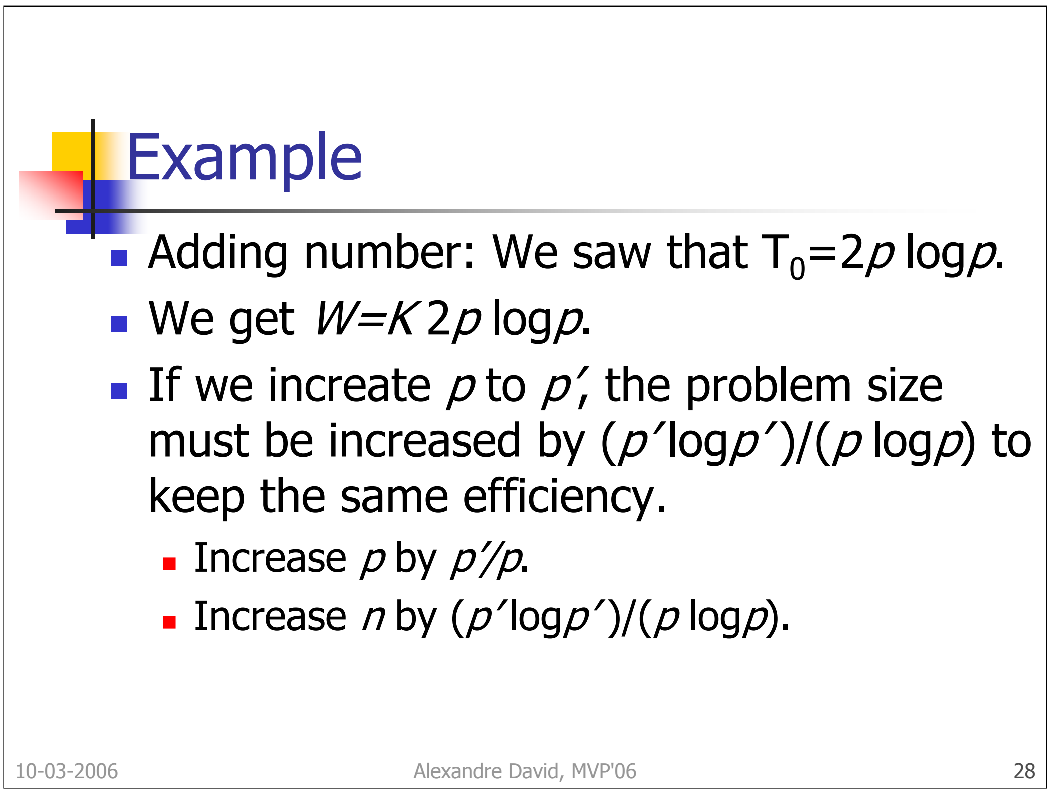
$W=KT_0(W,p)$

10-03-2006 Alexandre David, MVP'06 27

What it means: The isoefficiency function determines the ease with which a parallel system can maintain its efficiency in function of the number of processors. A small function means that small increments of the problem size are enough (to compensate the increase of p), i.e., the system is scalable. A large function means the problem size must be incremented dramatically to compensate p, i.e., the system is poorly scalable.

Unscalable system do not have an isoefficiency function.

Isoefficiency function is in function of *p*.



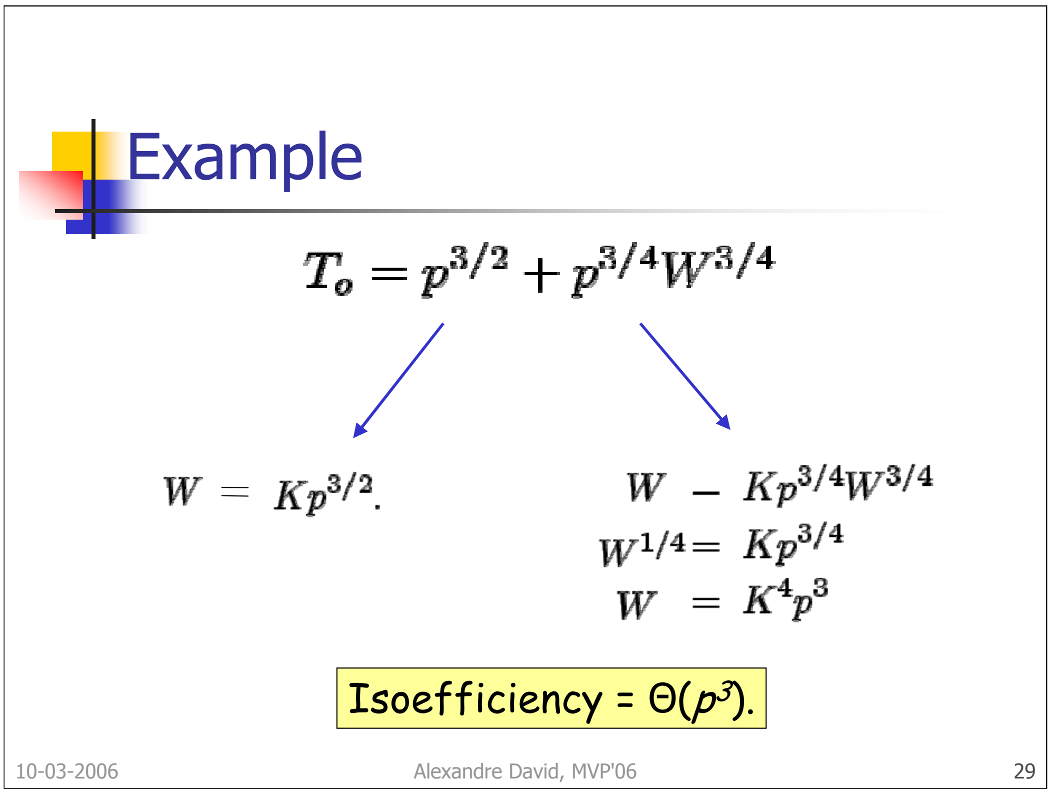
## Example

- Adding number: We saw that $T_0=2p\log p$.
- We get $W=K2p\log p$.
- If we increate $p$ to $p'$, the problem size must be increased by $(p'\log p')/(p\log p)$ to keep the same efficiency.
  - Increase $p$ by $p'/p$.
  - Increase $n$ by $(p'\log p')/(p\log p)$.

10-03-2006 Alexandre David, MVP'06 28

Here the overhead depends on $p$ only but in general it depends on $n$ as well.

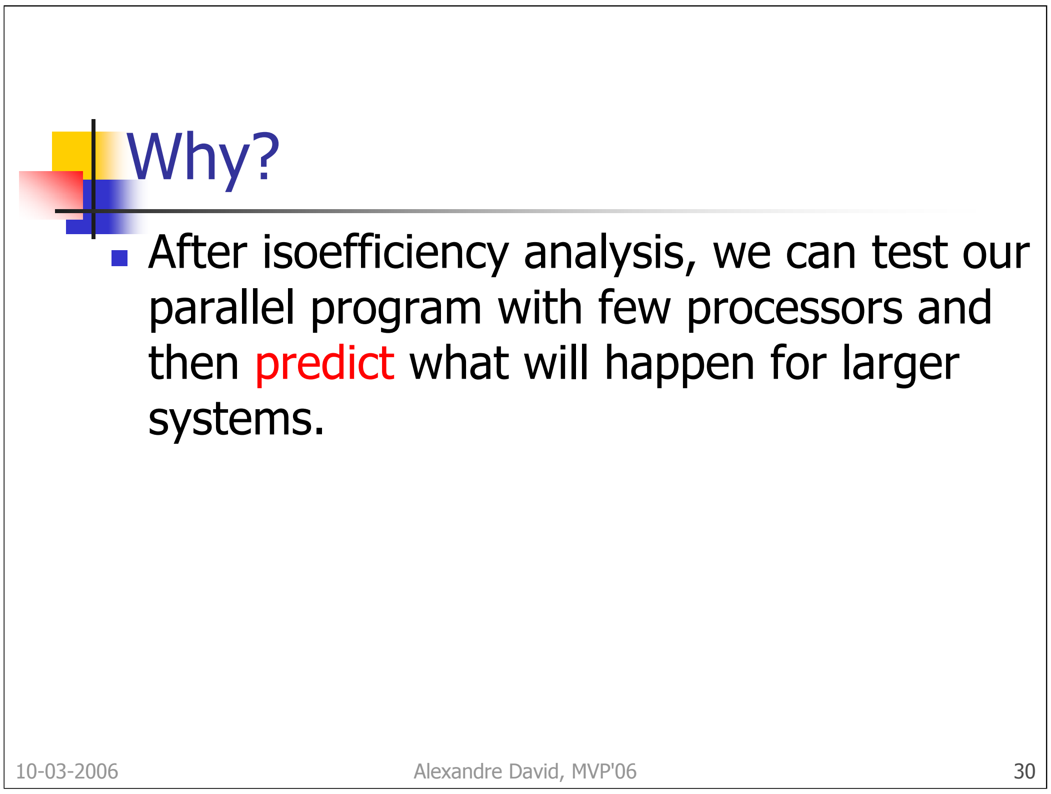
For more complex expressions of $T_0$, decompose and solve individually each term, and keep the asymptotically dominant term for the isoefficiency.



## Example

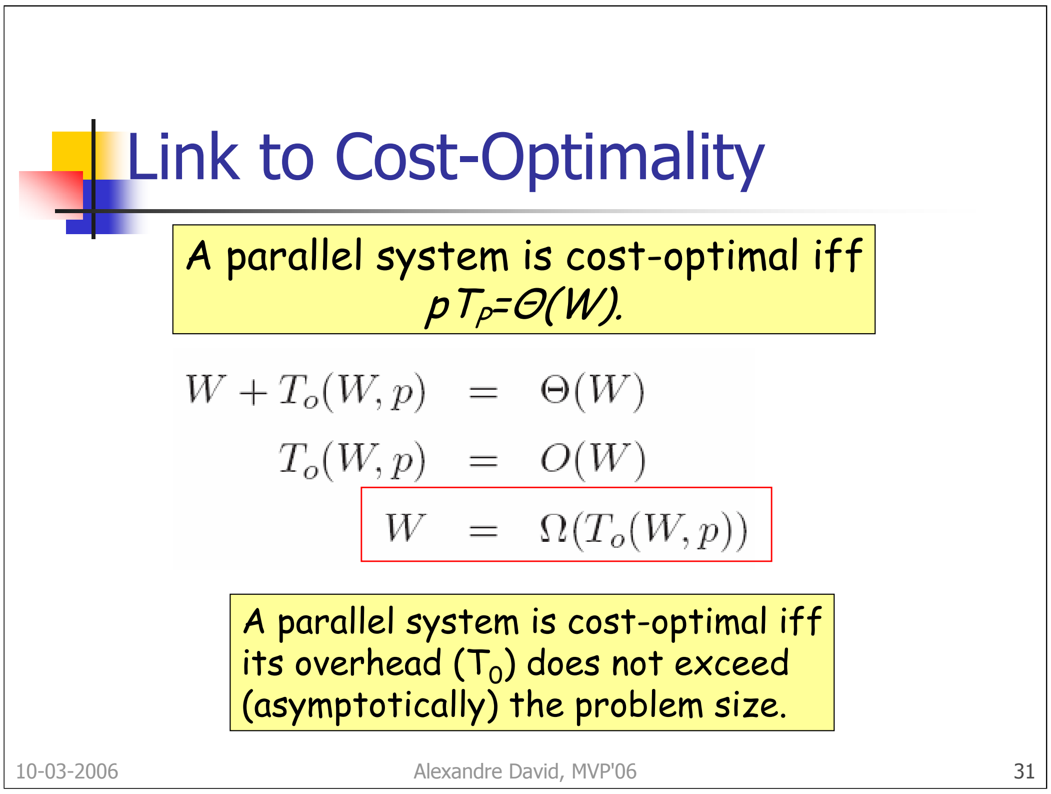
$$\begin{tikzcd}
 & T_o = p^{3/2} + p^{3/4}W^{3/4} \arrow[dl] \arrow[dr] & \\
W = Kp^{3/2}. & & \begin{array}{rl} W & - Kp^{3/4}W^{3/4} \\ W^{1/4} & = Kp^{3/4} \\ W & = K^4p^3 \end{array}
\end{tikzcd}$$

Isoefficiency = $\Theta(p^3)$.



## Why?

- After isoefficiency analysis, we can test our parallel program with few processors and then predict what will happen for larger systems.



## Link to Cost-Optimality

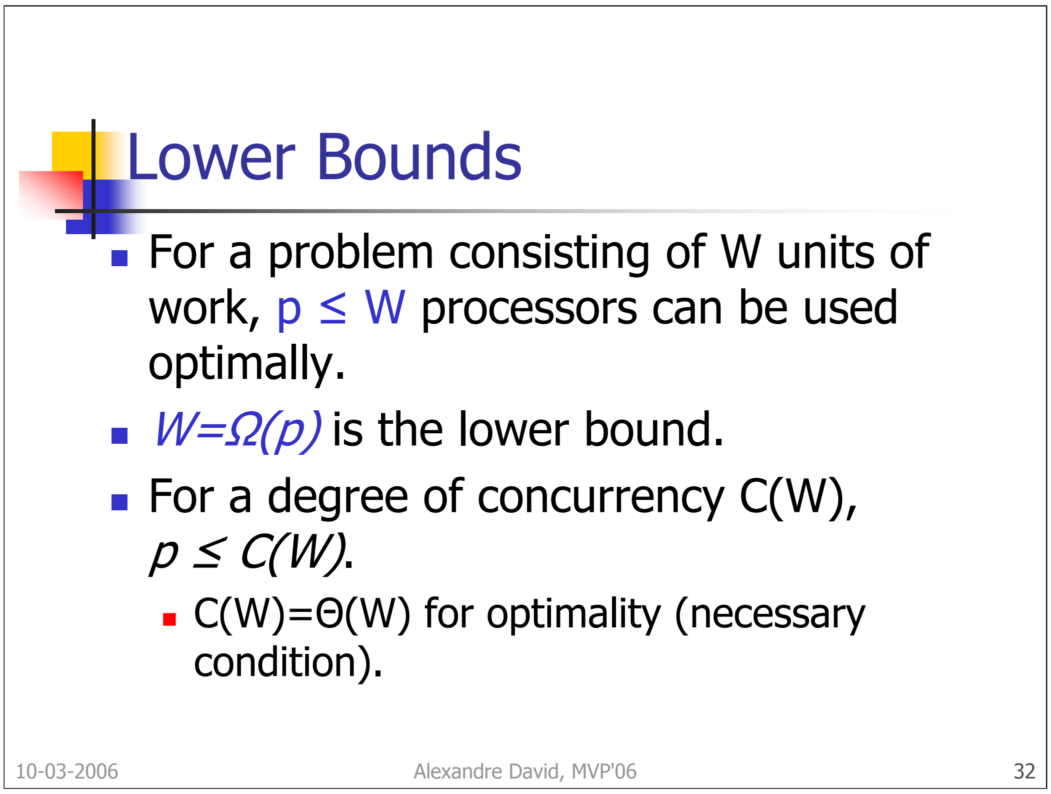
A parallel system is cost-optimal iff *$pT_P = \Theta(W)$.*

$$\begin{aligned} W + T_o(W,p) &= \Theta(W) \\ T_o(W,p) &= O(W) \\ \boxed{W = \Omega(T_o(W,p))} \end{aligned}$$

A parallel system is cost-optimal iff its overhead ($T_0$) does not exceed (asymptotically) the problem size.

10-03-2006 Alexandre David, MVP'06 31

Recall for cost-optimality. We saw this previously for the example of adding numbers.



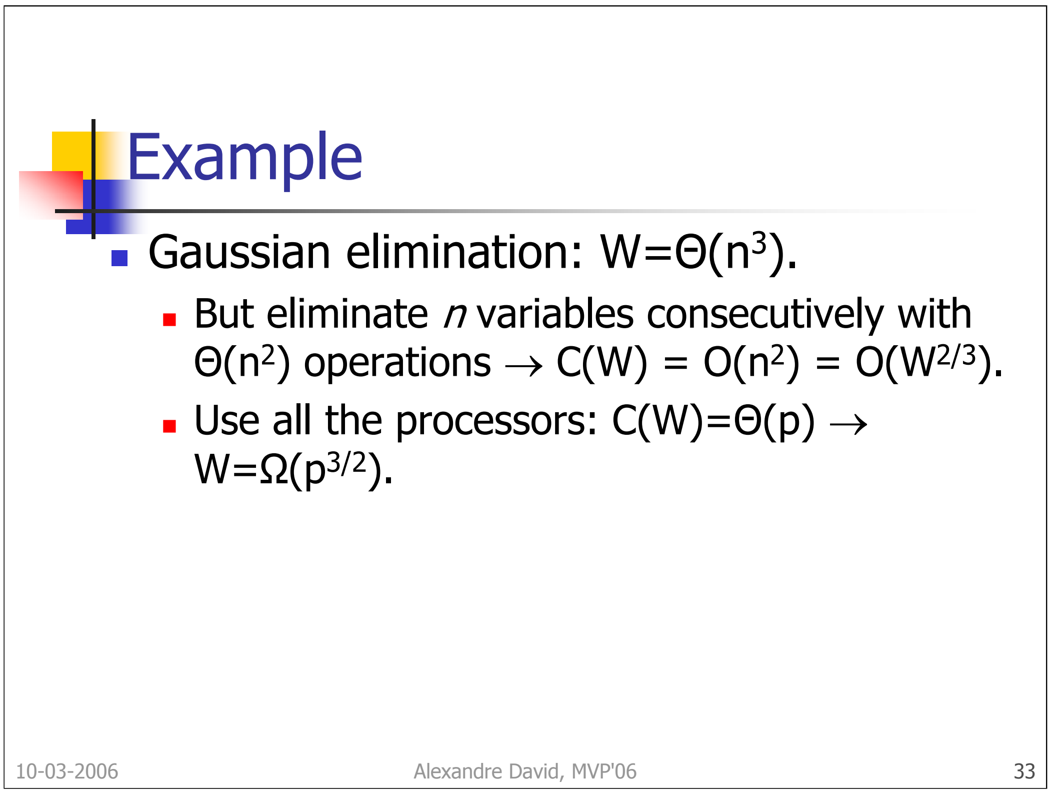
## Lower Bounds

- For a problem consisting of W units of work, $p \leq W$ processors can be used optimally.
- *$W=\Omega(p)$* is the lower bound.
- For a degree of concurrency C(W), *$p \leq C(W)$*.
  - $C(W)=\Theta(W)$ for optimality (necessary condition).

10-03-2006 Alexandre David, MVP'06 32

Degree of concurrency (chapter 5) = maximal degree of concurrency (chapter 3).

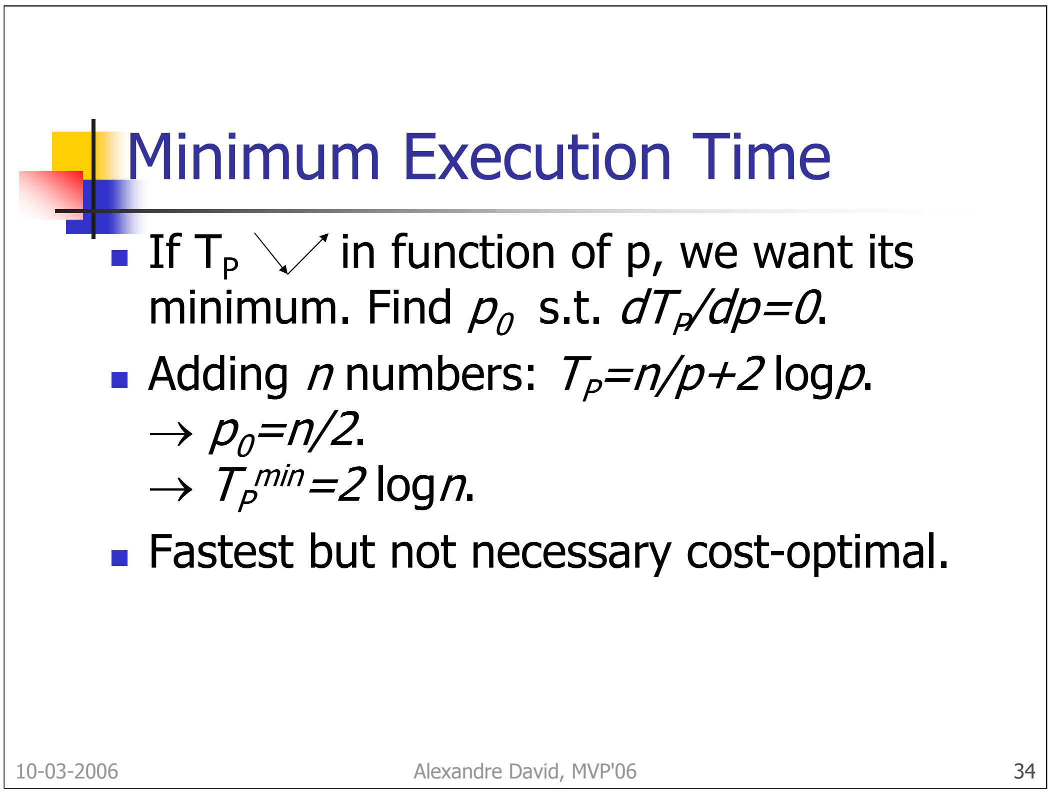
Optimal if $W=\Theta(p)$. If $C(W)<\Theta(W)$ (order of magnitude) then not optimal.



## Example

- Gaussian elimination: $W=\Theta(n^3)$.
  - But eliminate *n* variables consecutively with $\Theta(n^2)$ operations $\rightarrow C(W) = O(n^2) = O(W^{2/3})$.
  - Use all the processors: $C(W)=\Theta(p) \rightarrow W=\Omega(p^{3/2})$.

Isoefficiency function not optimal here.



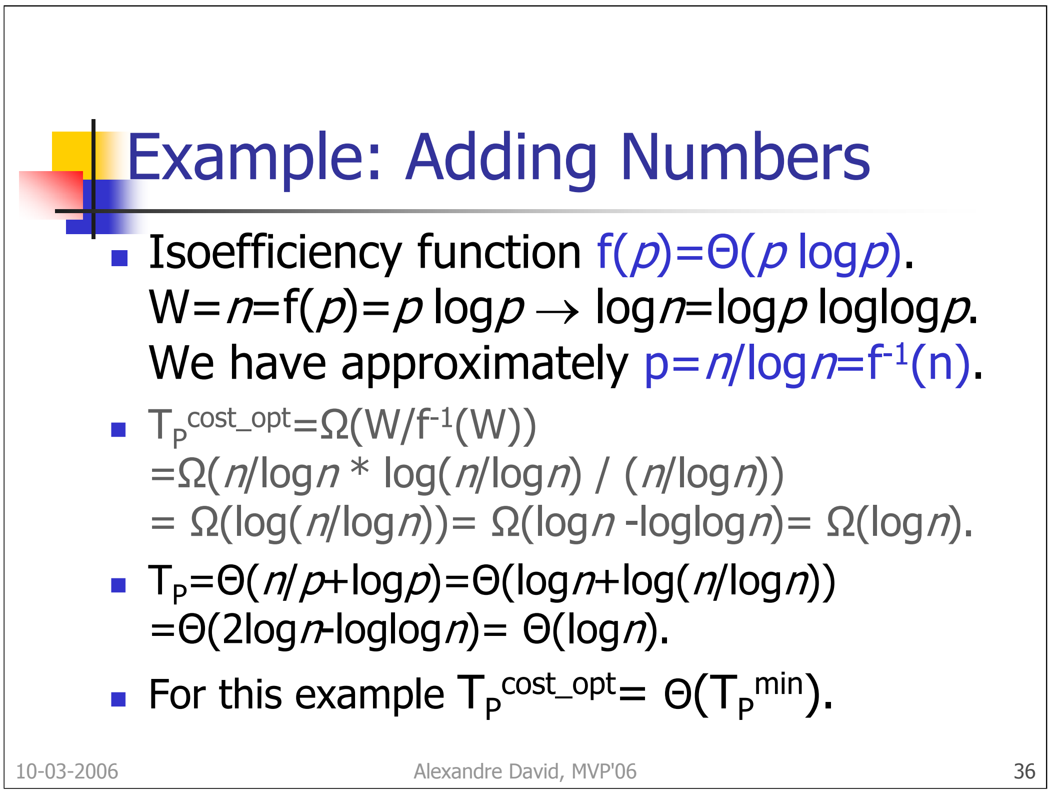
## Minimum Execution Time

- If $T_P$ $\searrow\nearrow$ in function of p, we want its minimum. Find $p_0$ s.t. $dT_P/dp=0$.
- Adding $n$ numbers: $T_P=n/p+2\log p$.
  $\rightarrow p_0=n/2$.
  $\rightarrow T_P^{min}=2\log n$.
- Fastest but not necessary cost-optimal.

Often what we are interested in = minimum execution time.

## Cost-Optimal Minimum Execution Time

- If we solve cost-optimally, what is the minimum execution time?
- We saw that if isoefficiency function = $\Theta(f(p))$ then a problem of size W can be solved optimally iff $p=\Omega(f^{-1}(W))$.
- Cost-optimal system: $T_P=\Theta(W/p)$
  $\rightarrow T_P^{cost\_opt}=\Omega(W/f^{-1}(W))$.



## Example: Adding Numbers

- Isoefficiency function $f(p)=\Theta(p \log p)$.
  $W=n=f(p)=p \log p \rightarrow \log n=\log p \log\log p$.
  We have approximately $p=n/\log n=f^{-1}(n)$.
- $T_P^{cost\_opt}=\Omega(W/f^{-1}(W))$
  $=\Omega(n/\log n * \log(n/\log n) / (n/\log n))$
  $= \Omega(\log(n/\log n))= \Omega(\log n - \log\log n)= \Omega(\log n)$.
- $T_P=\Theta(n/p+\log p)=\Theta(\log n+\log(n/\log n))$
  $=\Theta(2\log n-\log\log n)= \Theta(\log n)$.
- For this example $T_P^{cost\_opt}= \Theta(T_P^{min})$.

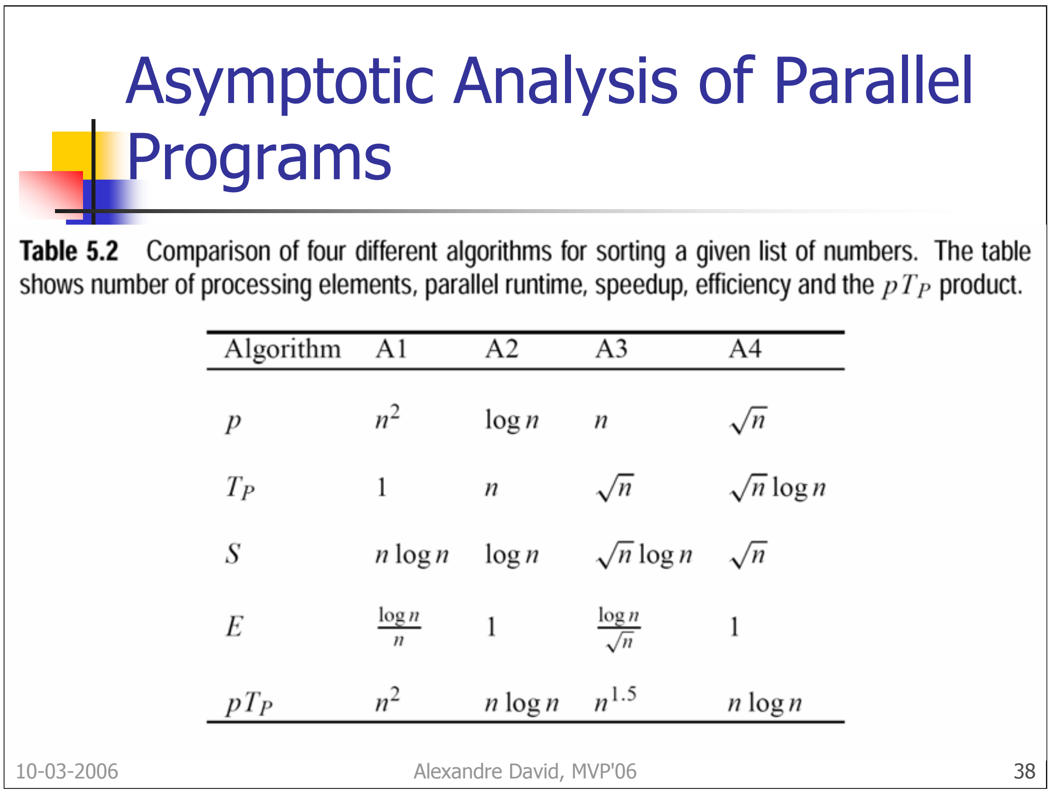
10-03-2006 Alexandre David, MVP'06 36

Equation 5.5 should be used, not 5.2.

In general it is possible to have $T_P^{cost\_opt} > \Theta(T_P^{min})$.

## Remark

- If $p_0$ > C(W) then its value is meaningless. $T_P^{min}$ is obtained for p=C(W).

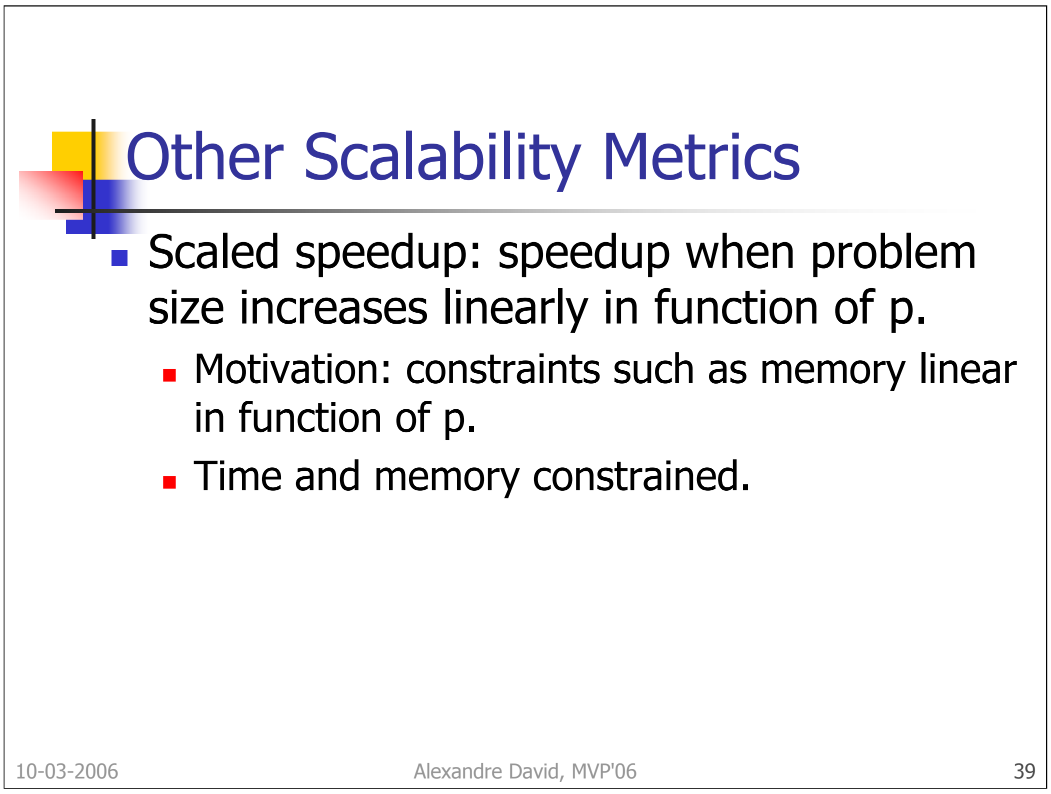


## Asymptotic Analysis of Parallel Programs

**Table 5.2** Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the $pT_P$ product.

| Algorithm | A1 | A2 | A3 | A4 |
|---|---|---|---|---|
| $p$ | $n^2$ | $\log n$ | $n$ | $\sqrt{n}$ |
| $T_P$ | $1$ | $n$ | $\sqrt{n}$ | $\sqrt{n}\log n$ |
| $S$ | $n\log n$ | $\log n$ | $\sqrt{n}\log n$ | $\sqrt{n}$ |
| $E$ | $\frac{\log n}{n}$ | $1$ | $\frac{\log n}{\sqrt{n}}$ | $1$ |
| $pT_P$ | $n^2$ | $n\log n$ | $n^{1.5}$ | $n\log n$ |

10-03-2006 Alexandre David, MVP'06 38



## Other Scalability Metrics

- Scaled speedup: speedup when problem size increases linearly in function of p.
  - Motivation: constraints such as memory linear in function of p.
  - Time and memory constrained.

The constraints link p and n.