CSCI 5451: Introduction to Parallel Computing

Lecture 16: Analytical Modeling

computer science
& engineering

University of Minnesota
Driven to Discover®

Announcements (10/27)

Project Groups → If you are not a part of a project group, or only have 1/2 people, you must form a group of 3-5

- If this is not done by Wednesday, we will form larger groups as needed

Lecture Overview

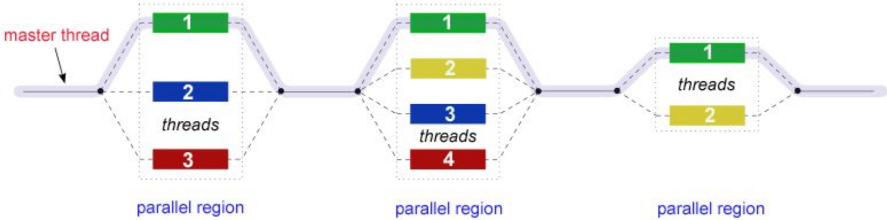
- Background
- Overheads
- Definitions
  - Serial Runtime/Parallel Runtime/Parallel Overhead
  - Speedup
  - Efficiency
  - Cost
- Granularity

Lecture Overview

- **Background**
- Overheads
- Definitions
  - Serial Runtime/Parallel Runtime/Parallel Overhead
  - Speedup
  - Efficiency
  - Cost
- Granularity

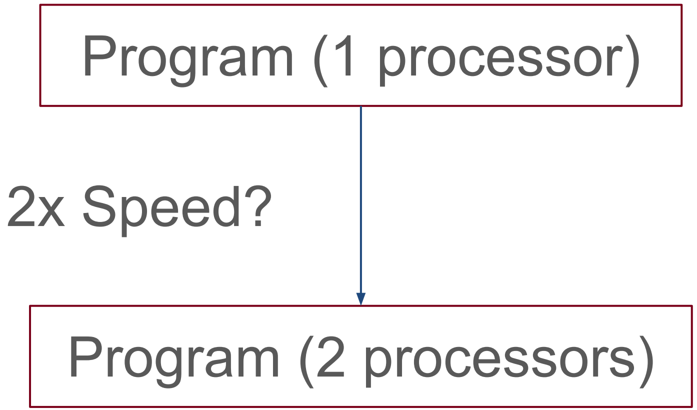
Analysis of Parallel Programs

- Serial programs can make use of asymptotic runtimes to define the relative ‘goodness’ of different algorithms
- Parallel programs introduce the use of potentially many processors at once
- We need to define terms which show the relative ‘goodness’ of different parallel algorithms beyond just runtimes



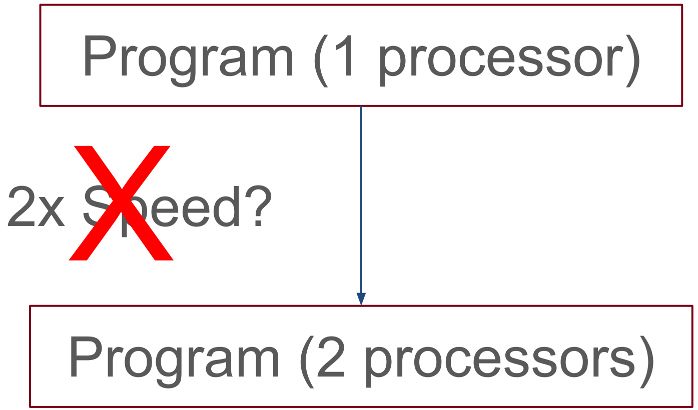
Desired Speedups

- Suppose we have some parallel program which we have chosen to run with twice as much hardware as was run in the serial case
- We would hope that this will lead to twice the speedups



Desired Speedups

- Suppose we have some parallel program which we have chosen to run with twice as much hardware as was run in the serial case
- We would hope that this will lead to twice the speedups



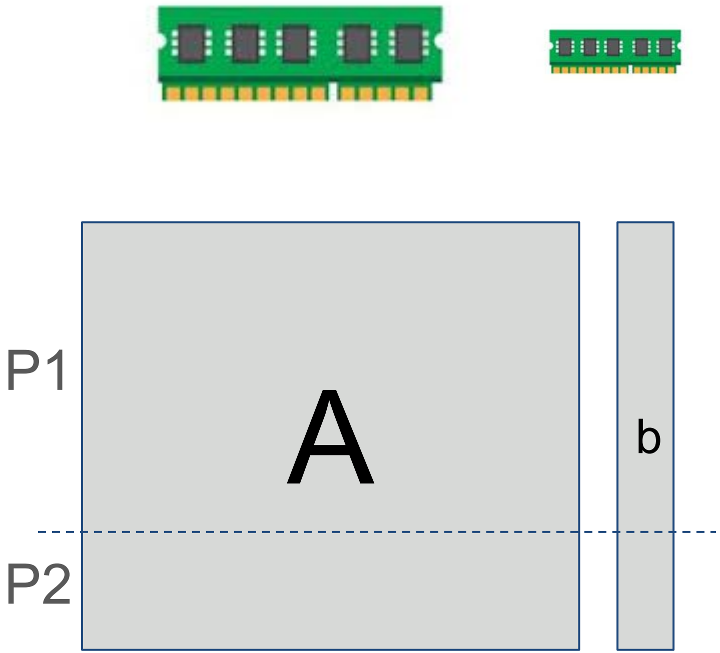
The overheads of running a program will usually result in less than the desired speedups

Lecture Overview

- Background
- **Overheads**
- Definitions
  - Serial Runtime/Parallel Runtime/Parallel Overhead
  - Speedup
  - Efficiency
  - Cost
- Granularity

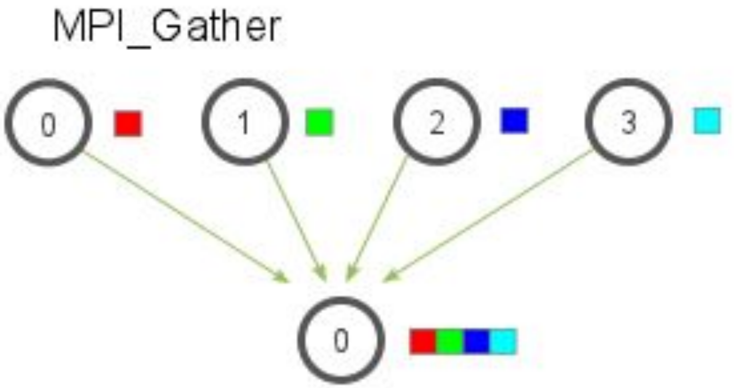
Overhead In Programs (Idling)

- Load imbalances → If your program distributes work unevenly, then some processes will finish early & wait (e.g. sparse matrix multiplication)
- Uneven hardware → Uneven processing speeds/networking speeds/memory/memory locality
- Resource Contention on locks



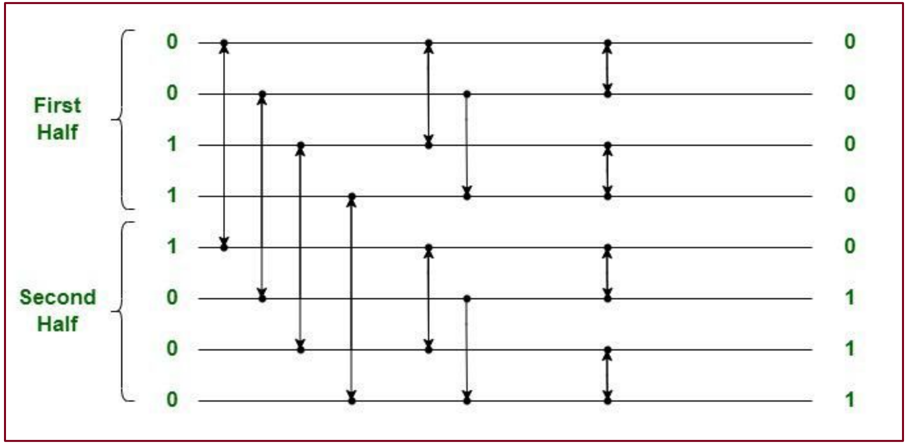
Overhead In Programs (Communication)

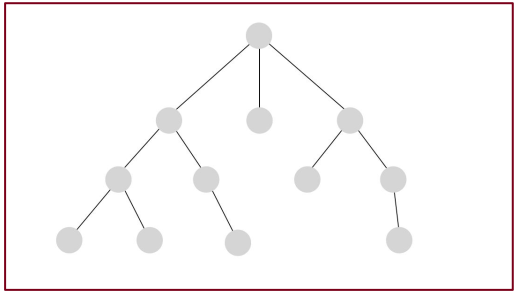
- Distributed Memory → Any communication among processors will take excess time away from actual computation
- Shared Memory → Even though there may be shared memory, each thread will likely be running on separate cores - each with their own separate cache. If they need to share information, it must still be updated here.



Overhead In Programs (Redundant Computation)

- Sometimes, we can accept performing some degree of excess computation in order to reduce extra communication or idling
- Parallel variants of sorting, graph traversal, FFT all make use of this
- Implies that we are computing more things than in the serial case

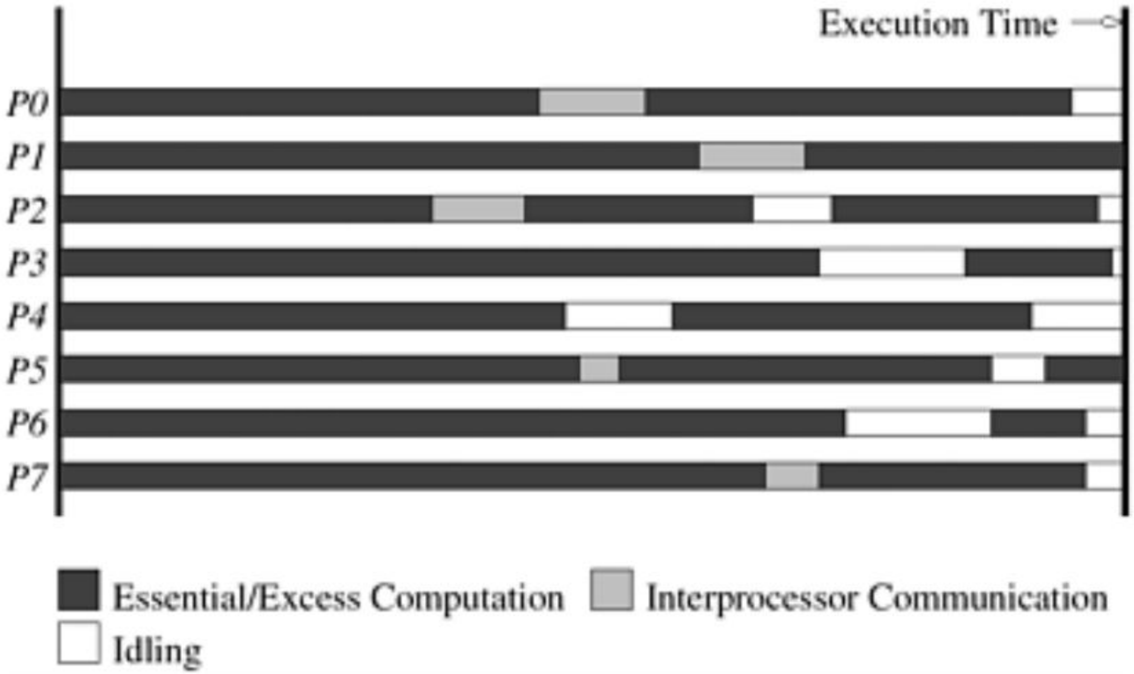




Overhead In programs

Overheads are not necessarily always a bad thing - they are a necessary component of parallel programming.

We will discuss how to asymptotically define our programs in terms of overhead



Lecture Overview

- Background
- Overheads
- **Definitions**
    - **Serial Runtime/Parallel Runtime/Parallel Overhead**
    - Speedup
    - Efficiency
    - Cost
- Granularity

Serial Runtime

- The time elapsed between the beginning & ending of a serial program
- You should ***always*** use the fastest algorithm to solve the given problem you are examining
- We will compare this speed to that of a hypothetical parallel program - ***don't*** compare to a slower algorithm, compare to the best serial algorithm

Serial Runtime: T_s

Parallel Runtime

- Analogous to the serial runtime (T_s)
- The time that elapses between the start of parallel computation to the moment the last processing element completes execution
- Note that this term is a function of the number of processes p

Parallel Runtime: T_p

Parallel Overhead

- Gathers all the previously discussed overheads into a single term
- Defines the excess processing time taken up across all processes

$$T_o = pT_p - T_s$$

Parallel Overhead

You want to minimize this term. More overheads means wasting processing power

- Gathers all the previously discussed overheads into a single term
- Defines the excess processing time taken up across all processes

$$T_o = pT_p - T_s$$

Lecture Overview

- Background
- Overheads
- **Definitions**
  - Serial Runtime/Parallel Runtime/Parallel Overhead
  - **Speedup**
  - Efficiency
  - Cost
- Granularity

Speedup

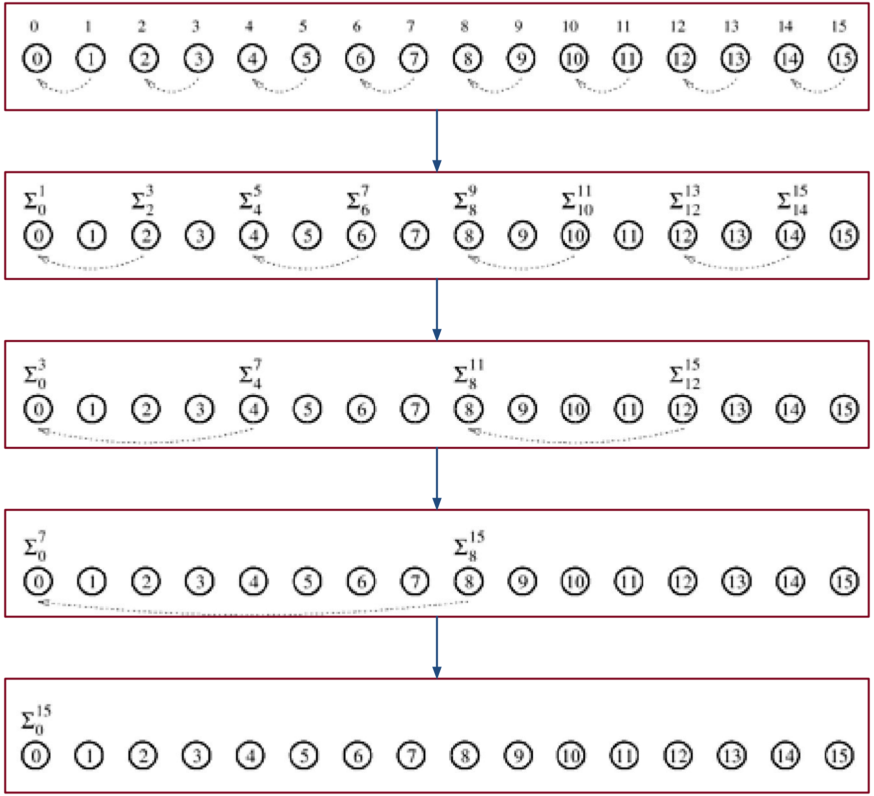
- Defines the ratio of time taken in the serial program to the parallel program
- If there are *p* processes, we want *S* to be closer to *p*
- That is, multiplying the hardware by *p* should get us close to *p* times speedups
- Assume that the serial program uses the same hardware as the parallel version

$$S = T_s / T_p$$

Speedup Example

How many processes can we use?

Sum up *n* numbers

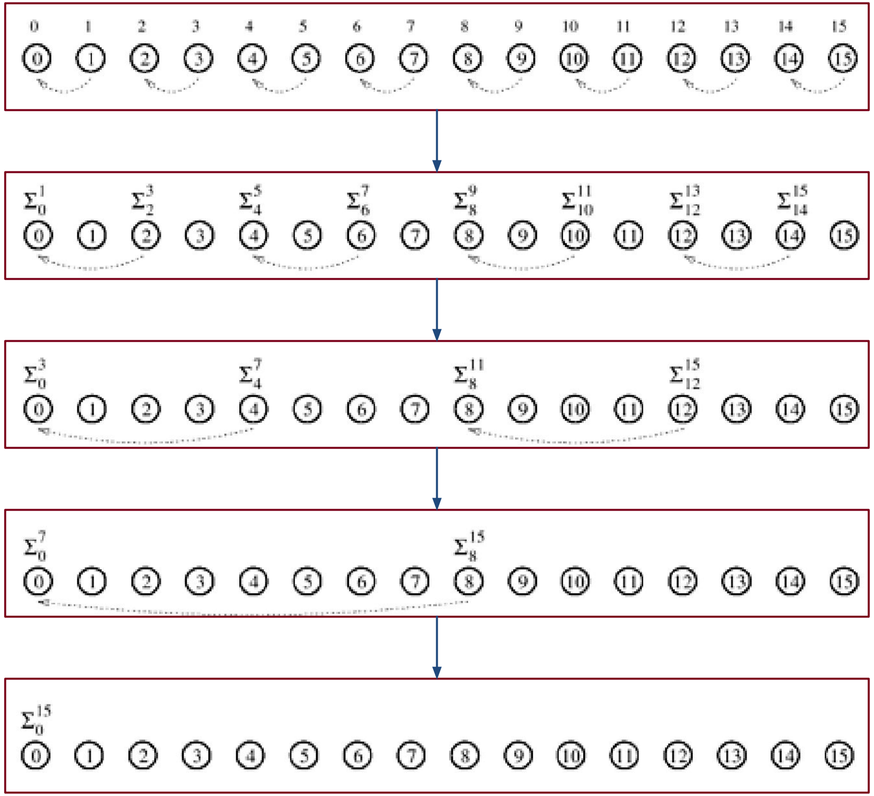


Speedup Example

How many processes can we use?

$p = n$

Sum up n numbers



Speedup Example

How many processes can we use?

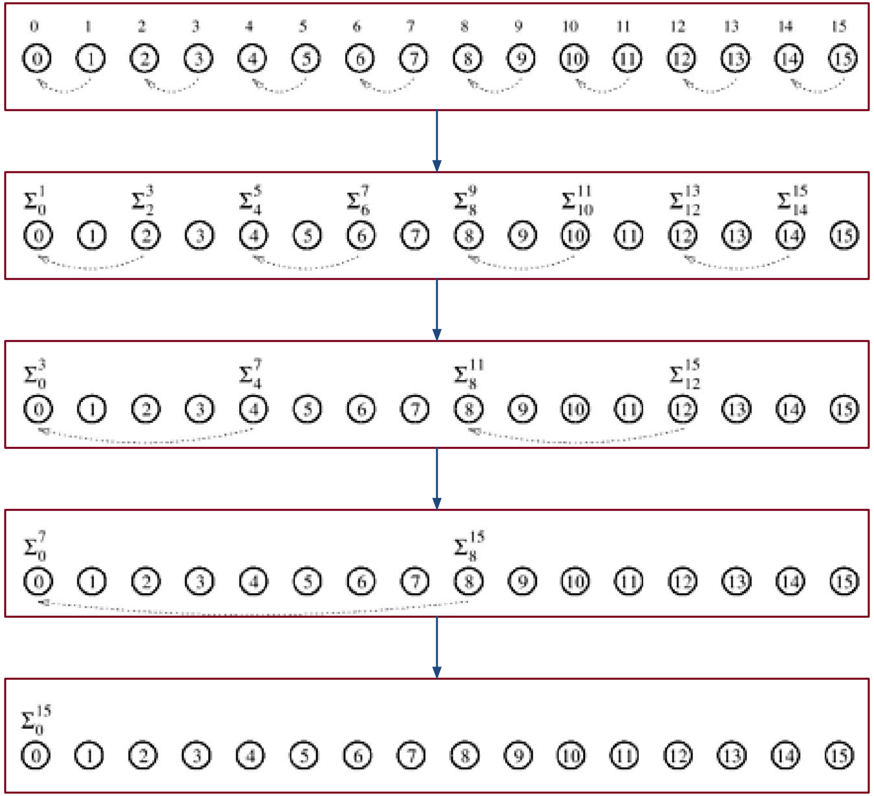
$p = n$

Big-Θ runtimes?

$T_s = ?$

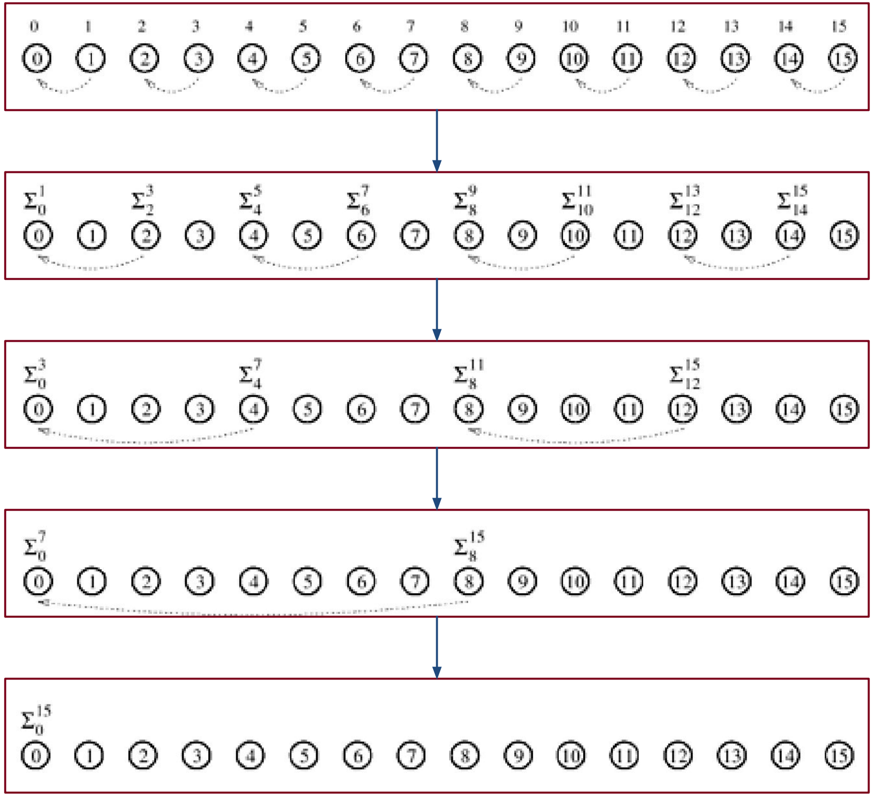
$T_p = ?$

Sum up *n* numbers



Speedup Example

Sum up n numbers



How many processes can we use?

$p = n$

Big-Θ runtimes?

$T_s = \Theta(n)$

$T_p = \Theta(\log n)$

Speedup Example

How many processes can we use?

$p = n$

Big-Θ runtimes?

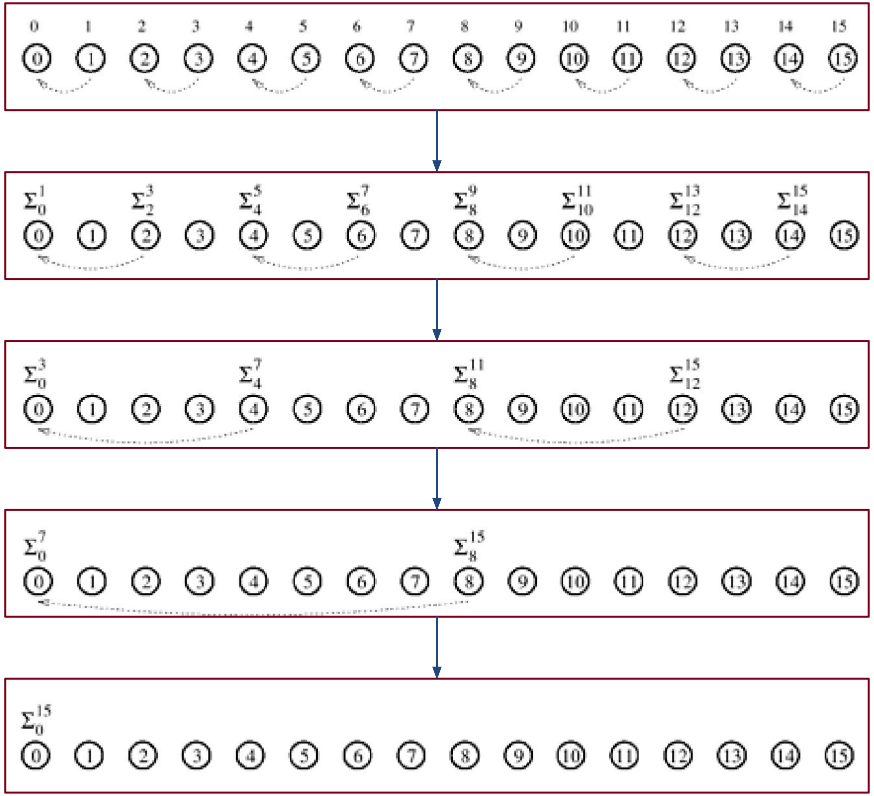
$T_s = \Theta(n)$

$T_p = \Theta(\log n)$

Speedup?

$S = \Theta(?)$

Sum up n numbers



Speedup Example

Sum up n numbers

How many processes can we use?

$p = n$

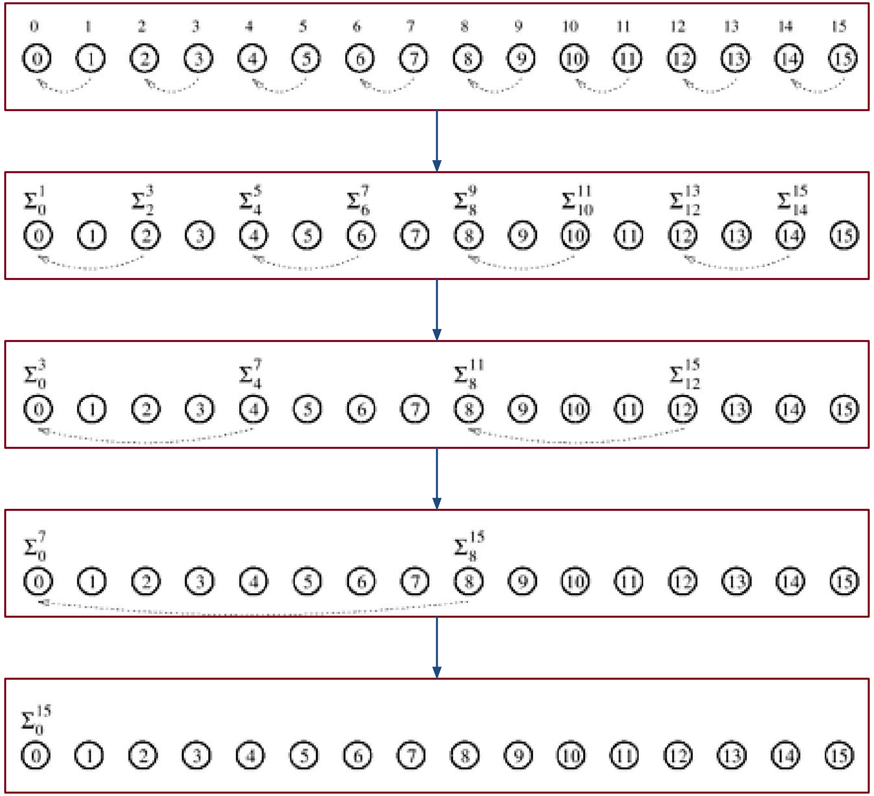
Big-Θ runtimes?

$T_s = \Theta(n)$

$T_p = \Theta(\log n)$

Speedup?

$S = \Theta(n/\log n)$



Sorting Speedups Example

- Assume serial bubble sort takes 150 seconds
- Serial quicksort takes 30 seconds
- Parallel odd-even sort takes 40 seconds on 4 processes

$T_p = ?$

$T_s = ?$

$S = ?$

$T_o = ?$

Sorting Speedups Example

- Assume serial bubble sort takes 150 seconds
- Serial quicksort takes 30 seconds
- Parallel odd-even sort takes 40 seconds on 4 processes

$T_p = 40$

$T_s = 30$

$S = .75$

$T_o = 130$

Sorting Speedups Example

Always use best serial algorithm - parallelizing in this example is pointless as the best serial algorithm is better

- Assume serial bubble sort takes 150 seconds
- Serial quicksort takes 30 seconds
- Parallel odd-even sort takes 40 seconds on 4 processes

$T_p = 40$

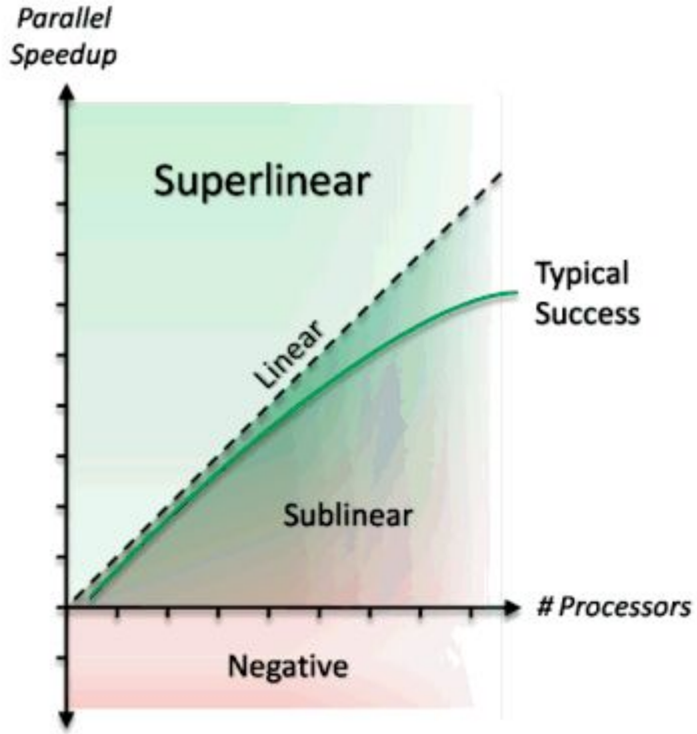
$T_s = 30$

$S = .75$

$T_o = 130$

Superlinear Speedups

- It is generally not possible to get greater than *p* speedup
- When this occurs, the program is said to exhibit superlinear speedup
- This is most commonly observed in exploration & caching



Caching Example

- Serial Version
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - 80% hit rate
  - Average access time:

Caching Example

- Serial Version
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - 80% hit rate
  - Average access time:
    .8*2 + .2*100= 21.6ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 21.6ns → 1/(21.6 *1e-9) = 46.3 Megaflops

Caching Example

- Serial Version
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - 80% hit rate
  - Average access time:
    .8*2 + .2*100= 21.6ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 21.6ns → 1/(21.6 *1e-9) = 46.3 Megaflops

- Parallel Version (2 threads)
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - Remote DRAM latency: 400ns
  - 90% cache hit, 8% DRAM, 2% Remote DRAM

Caching Example

- Serial Version
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - 80% hit rate
  - Average access time: .8*2 + .2*100= 21.6ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 21.6ns → 1/(21.6 *1e-9) = 46.3 Megaflops

- Parallel Version (2 threads)
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - Remote DRAM latency: 400ns
  - 90% cache hit, 8% DRAM, 2% Remote DRAM

The cache hit rate can increase if (a) the total problem size is large enough to not fit in cache on 1 thread, but is closer when on 2 threads ***and/or*** (b) there is a highly irregular access pattern from memory that has improved locality on a larger number of threads

Caching Example

- ❑ Serial Version
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - 80% hit rate
  - Average access time:
    .8*2 + .2*100= 21.6ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 21.6ns → 1/(21.6 *1e-9) = 46.3 Megaflops

- ❑ Parallel Version (2 threads)
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - Remote DRAM latency: 400ns
  - 90% cache hit, 8% DRAM, 2% Remote DRAM

Caching Example

- Serial Version
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - 80% hit rate
  - Average access time: .8*2 + .2*100= 21.6ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 21.6ns → 1/(21.6 *1e-9) = 46.3 Megaflops

- Parallel Version (2 threads)
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - Remote DRAM latency: 400ns
  - 90% cache hit, 8% DRAM, 2% Remote DRAM
  - Average access time:

Caching Example

- Serial Version
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - 80% hit rate
  - Average access time: .8*2 + .2*100= 21.6ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 21.6ns → 1/(21.6 *1e-9) = 46.3 Megaflops

- Parallel Version (2 threads)
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - Remote DRAM latency: 400ns
  - 90% cache hit, 8% DRAM, 2% Remote DRAM
  - Average access time: .9*2+.08*100+.02*400= 17.8ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 17.8ns → 1/(17.8 *1e-9) = 56.18 Megaflops
  - 2 threads → 112.36 MegaFlops

Caching Example

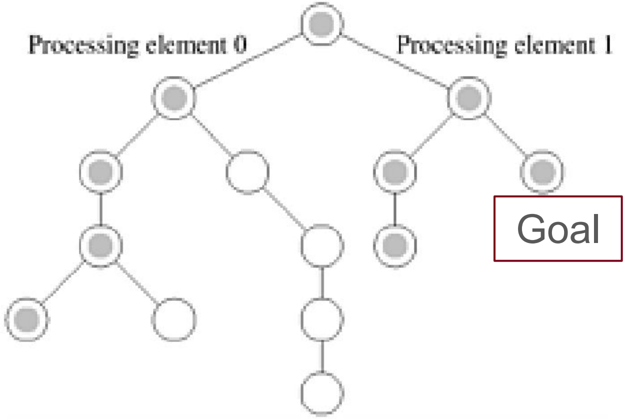
More than 2x!

- Serial Version
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - 80% hit rate
  - Average access time: .8*2 + .2*100= 21.6ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 21.6ns → 1/(21.6 *1e-9) = $\boxed{\text{46.3 Megaflops}}$

- Parallel Version (2 threads)
  - Cache Latency: 2ns
  - DRAM latency: 100ns
  - Remote DRAM latency: 400ns
  - 90% cache hit, 8% DRAM, 2% Remote DRAM
  - Average access time: .9*2+.08*100+.02*400= 17.8ns
  - Assume program is memory bottlenecked & only performs one FLOP/memory access
  - 1 FLOP every 17.8ns → 1/(17.8 *1e-9) = 56.18 Megaflops
  - 2 threads → $\boxed{\text{112.36 MegaFlops}}$

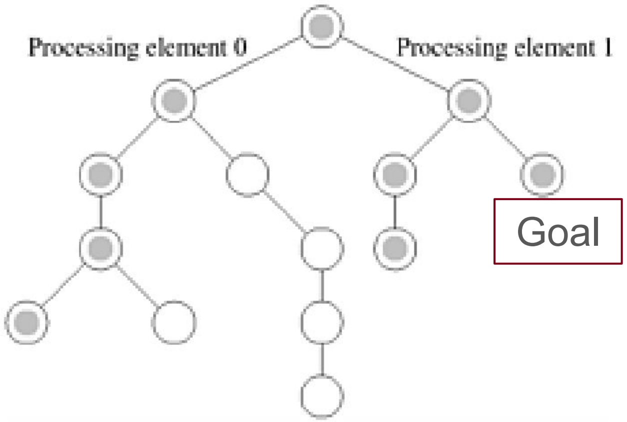
Exploratory Decomposition Example

- In search problems, sometimes the goal state can be expanded *much* more quickly in parallel
- Serial, depth-first search of the graph at right will take $14t_c$ where t_c is the cost of traversing one node ($T_s = 14t_c$)
- In parallel…



Exploratory Decomposition Example

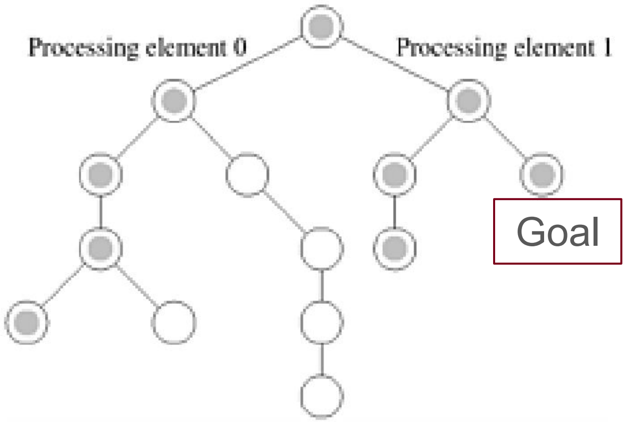
- In search problems, sometimes the goal state can be expanded *much* more quickly in parallel
- Serial, depth-first search of the graph at right will take $14t_c$ where t_c is the cost of traversing one node ($T_s = 14t_c$)
- In parallel with *p=2*, we have $T_p = 5t_c$
- S = ?



Exploratory Decomposition Example

- In search problems, sometimes the goal state can be expanded *much* more quickly in parallel
- Serial, depth-first search of the graph at right will take $14t_c$ where t_c is the cost of traversing one node ($T_s = 14t_c$)
- In parallel with $p=2$, we have $T_p = 5t_c$
- $\boxed{S = 14/5 > 2}$

Superlinear



Lecture Overview

- ❑ Background
- ❑ Overheads
- ❑ **Definitions**
  - Serial Runtime/Parallel Runtime/Parallel Overhead
  - Speedup
  - **Efficiency**
  - Cost
- ❑ Granularity

Efficiency

- Measures the fraction of time a processor is usefully employed
- One of the more important metrics for allocating resources efficiently
- The most efficient parallel algorithms have E close to 1
- E should usually lie in (0, 1)

$$E = S/p$$

Efficiency (Adding n Numbers)

$$S = \Theta(n/\log n)$$

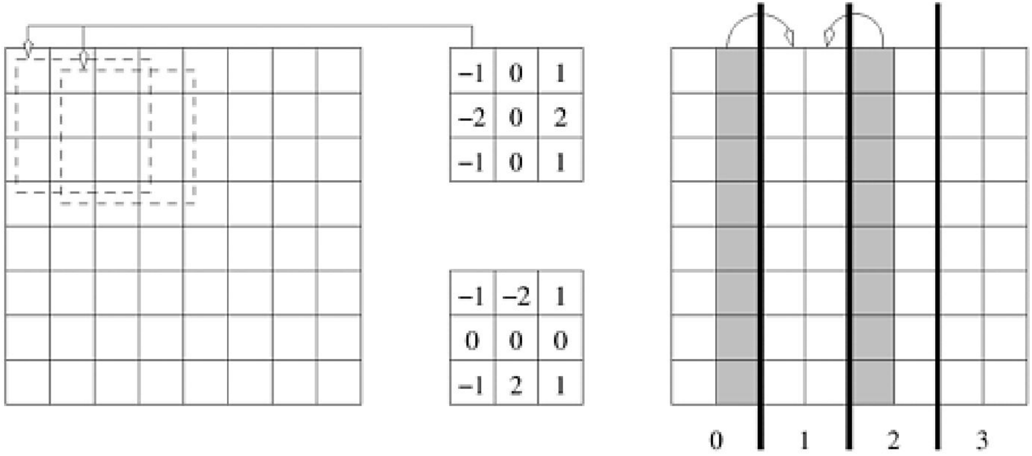
$$p = n$$

$$E = \Theta(n/\log n)/n = \boldsymbol{\Theta(1/\log n)}$$

Efficiency (Edge Detection Example)

Edge-Detection

- Compute edge feature map
- Input is n x n image
- With 3x3 kernels, $9n^2$ computations (assume each computation takes t_c seconds)
- Assume column-wise distribution as in right-most figure
- Let t_s and t_w be the startup time and per-word transfer time for communication, respectively
- Let p be the total number of processes



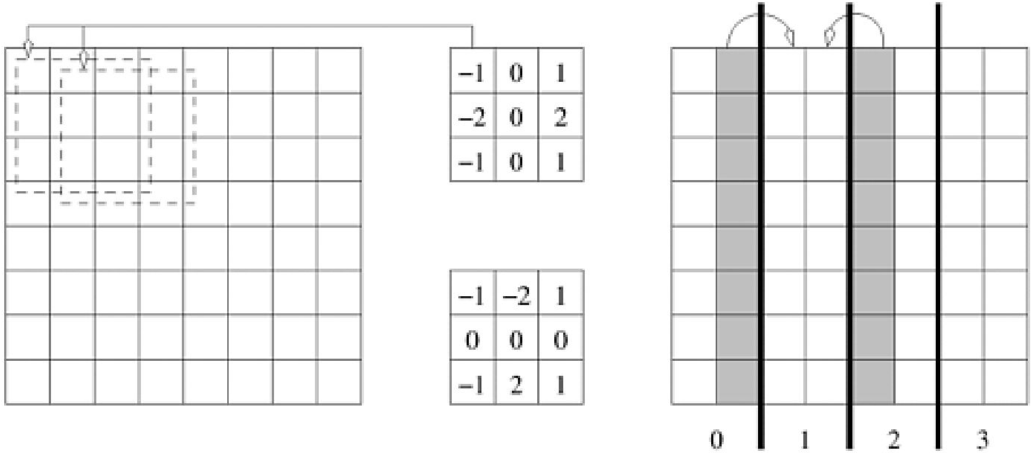
Efficiency (Edge Detection Example)

$T_s =$

$T_p =$

$S =$

$E =$



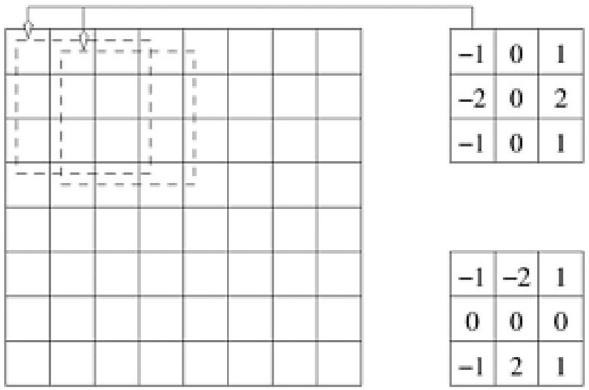
Efficiency (Edge Detection Example)

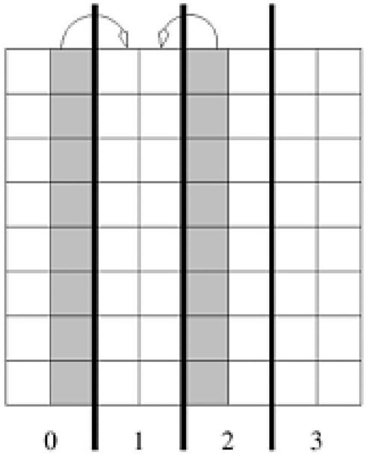
$T_s = 9t_c n^2$

$T_p =$

$S =$

$E =$





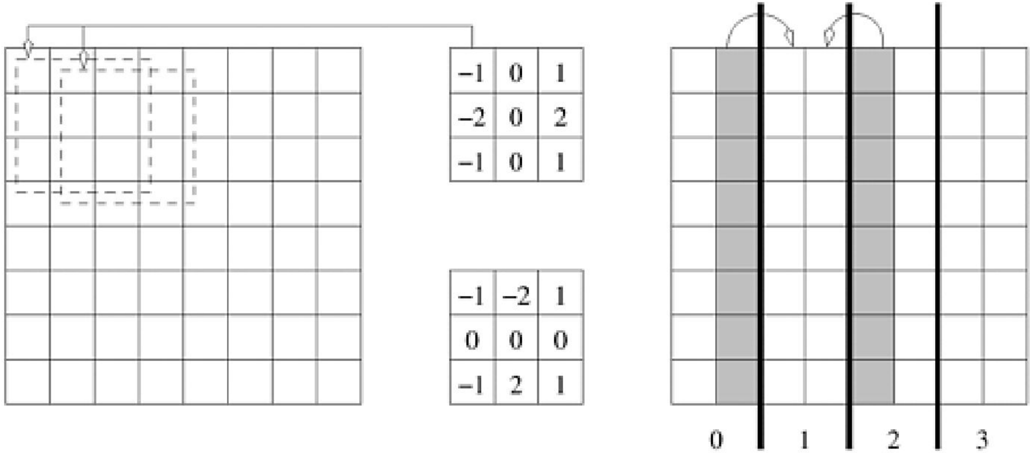
Efficiency (Edge Detection Example)

$T_s = 9t_c n^2$

$T_p = 9t_c\frac{n^2}{p} + 2(t_s + t_w n)$

S =

E =



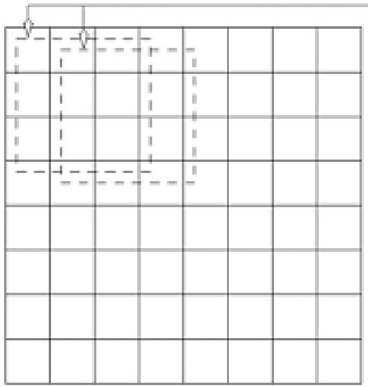
Efficiency (Edge Detection Example)

$T_s = 9t_c n^2$

$T_p = 9t_c \frac{n^2}{p} + 2(t_s + t_w n)$

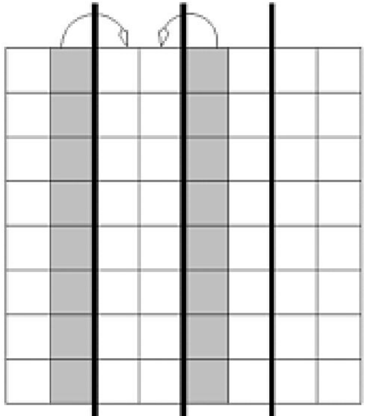
$S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)}$

$E =$



$$\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|}\hline -1 & -2 & 1 \\ \hline 0 & 0 & 0 \\ \hline -1 & 2 & 1 \\ \hline \end{array}$$



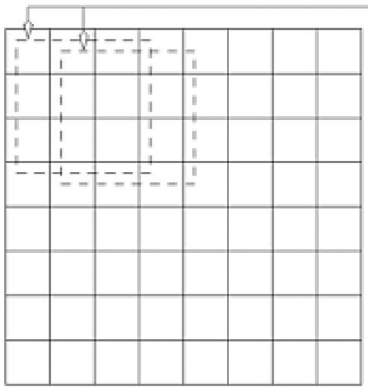
Efficiency (Edge Detection Example)

$$T_s = 9t_c n^2$$

$$T_p = 9t_c \frac{n^2}{p} + 2(t_s + t_w n)$$

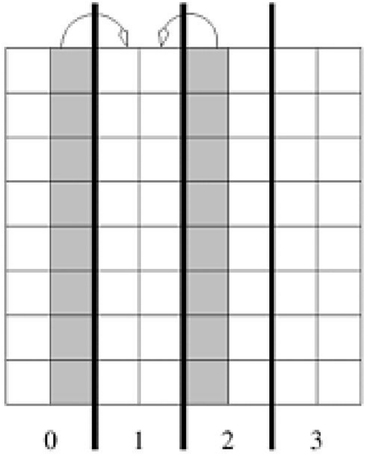
$$S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)}$$

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.$$



$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{bmatrix}$$



Lecture Overview

- Background
- Overheads
- **Definitions**
  - Serial Runtime/Parallel Runtime/Parallel Overhead
  - Speedup
  - Efficiency
  - **Cost**
- Granularity

Cost

- Sum of time spent on the program across all processes
- We can reformulate efficiency as $E = S/p = (T_s/T_p)/p = T_s/(pT_p) = T_s/Cost$
- Also sometimes called *work* or *processor-time product*
- A program is *cost-optimal* if it has the same big-Θ complexity as a function of input size as the fastest known sequential algorithm on a single processing element

$$Cost = pT_p$$

Cost of adding n numbers

- Recall from our adding example that
  - $p=n$
  - $T_p = \log n$
  - $T_s = n$
- Cost = $\Theta(n \log n)$
- Is this cost-optimal?

Cost of adding n numbers

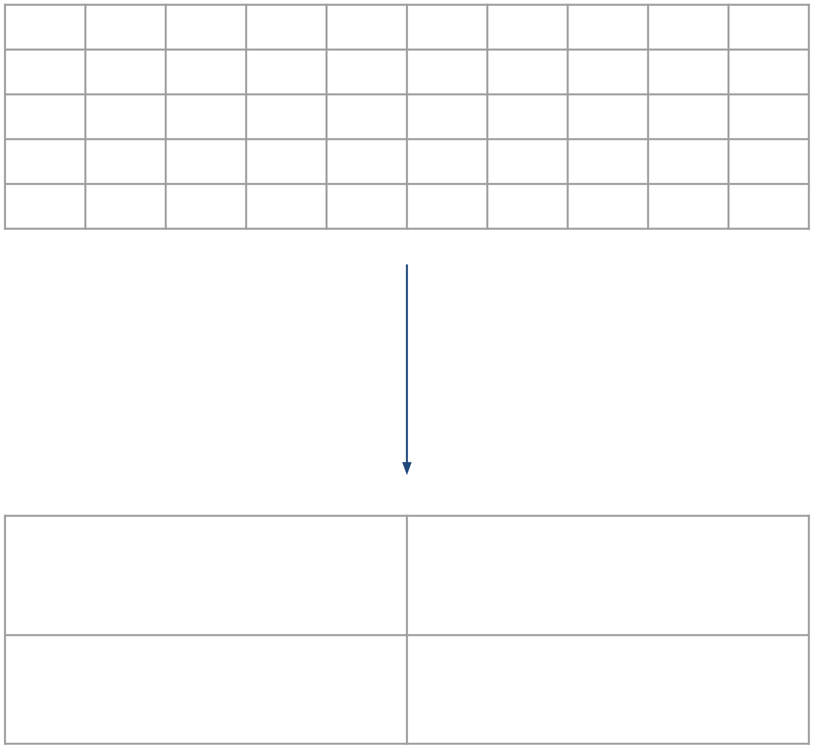
- Recall from our adding example that
  - $p = n$
  - $T_p = \log n$
  - $T_s = n$
- Cost = $\Theta(n \log n)$
- Is this cost-optimal?
  - No - cost grows asymptotically faster than the serial time of execution

Lecture Overview

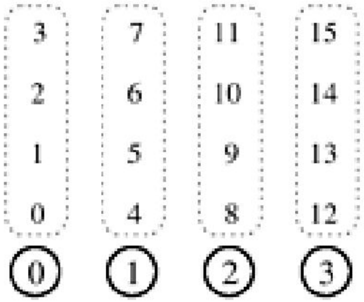
- Background
- Overheads
- Definitions
  - Serial Runtime/Parallel Runtime/Parallel Overhead
  - Speedup
  - Efficiency
  - Cost
- **Granularity**

Granularity

- Using fewer processes than the maximum possible is often more practical given the overheads introduced by idling/communication
- If we choose an appropriate level of granularity, we can create cost-optimal programs with fewer processes

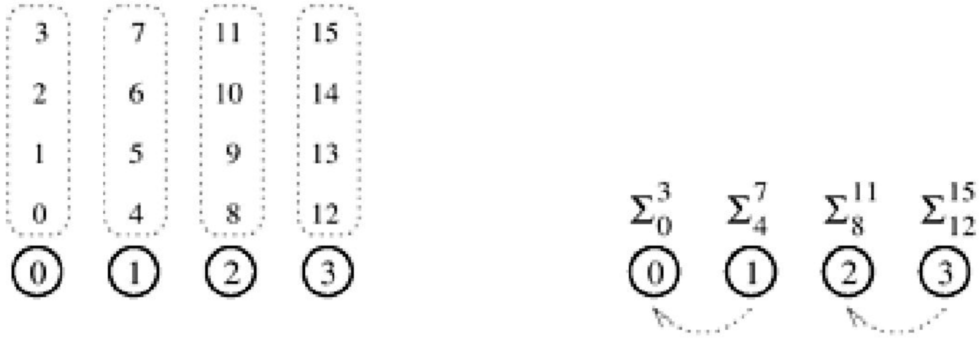


Cost Optimal Addition



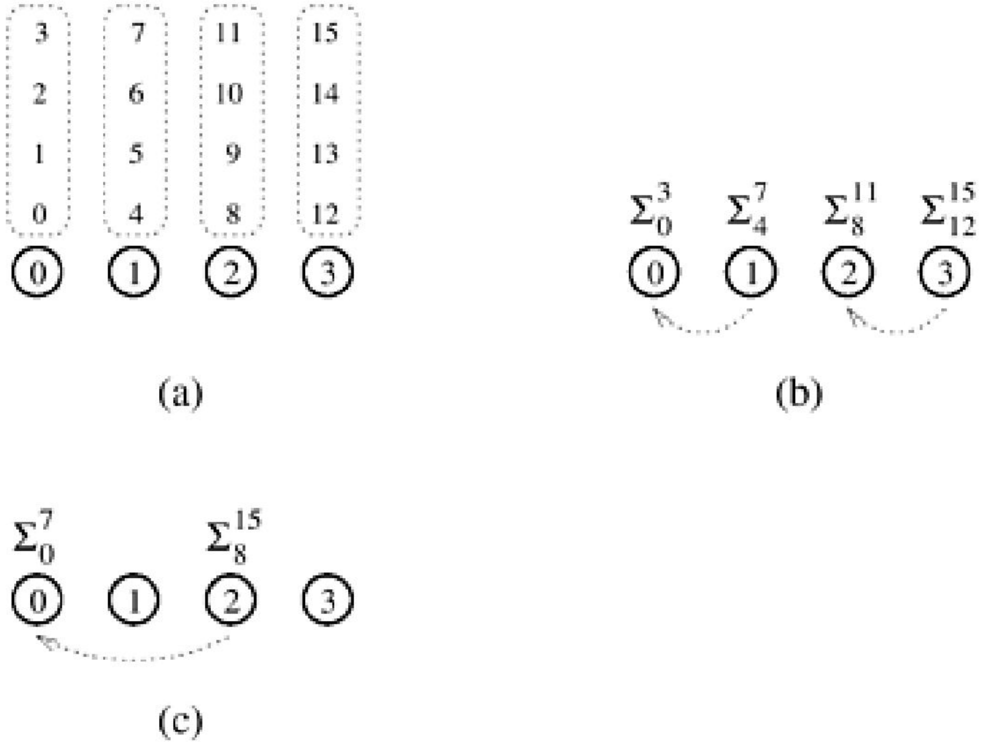
(a)

Cost Optimal Addition

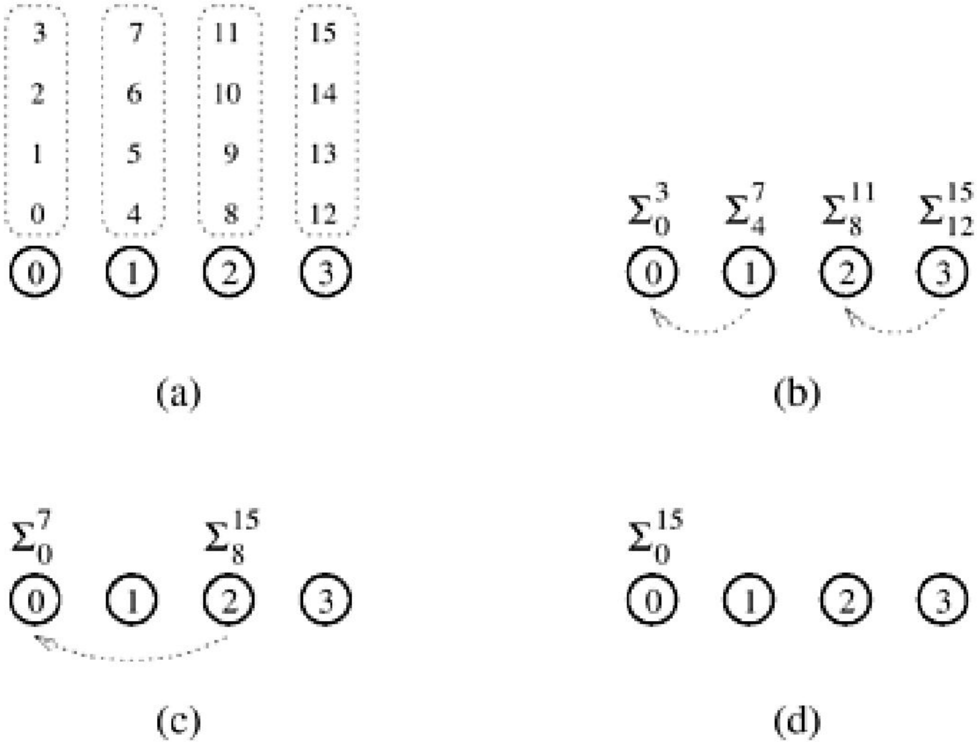


(a) (b)

Cost Optimal Addition



Cost Optimal Addition

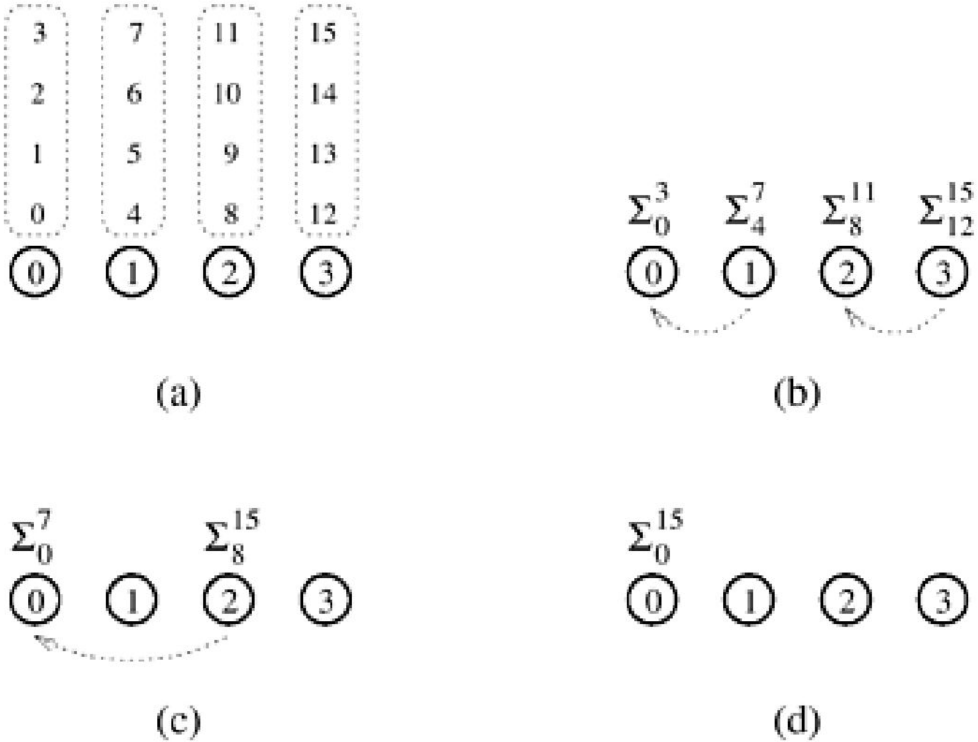


Cost Optimal Addition

$T_p = ?$

Cost = ?

$T_s = ?$

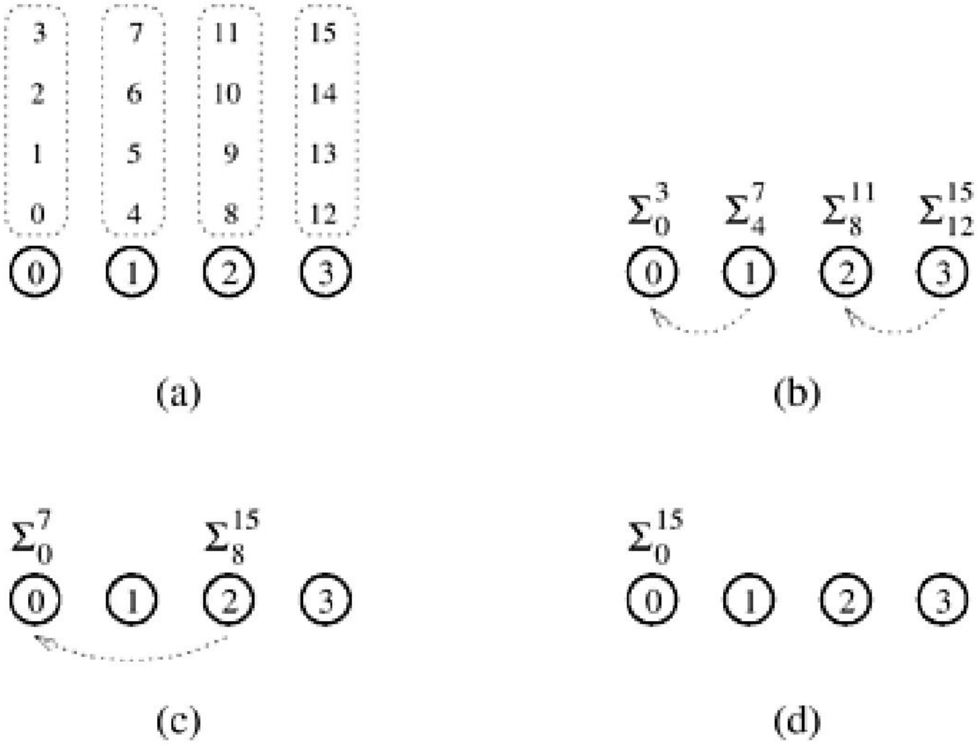


Cost Optimal Addition

$T_p = \Theta(n/p + \log p)$

$\text{Cost} = \Theta(n + p\log p)$

$T_s = \Theta(n)$



Cost Optimal Addition

$T_p = \Theta(n/p + \log p)$

$\text{Cost} = \Theta(n + p\log p)$

$T_s = \Theta(n)$

As long as $n = \Omega(p\log p)$, this program is cost-optimal