Parallel Programming Can be Hard

Thinking about all possible orderings of concurrent actions and how threads may interact/interfere

Some types of errors:

- ***Race conditions:***
  outcome depends on arbitrary OS scheduling order
- ***Deadlock:***
  errors in resource allocation prevent forward progress
- ***Livelock / Starvation / Fairness***:
  arbitrary scheduling can prevent tids progress

These can be difficult to find or difficult to fix

Parallel Programming Benefits

- Can greatly improve performance of program if can divide computation into parts that multiple threads can run simultaneously on multiple cores.

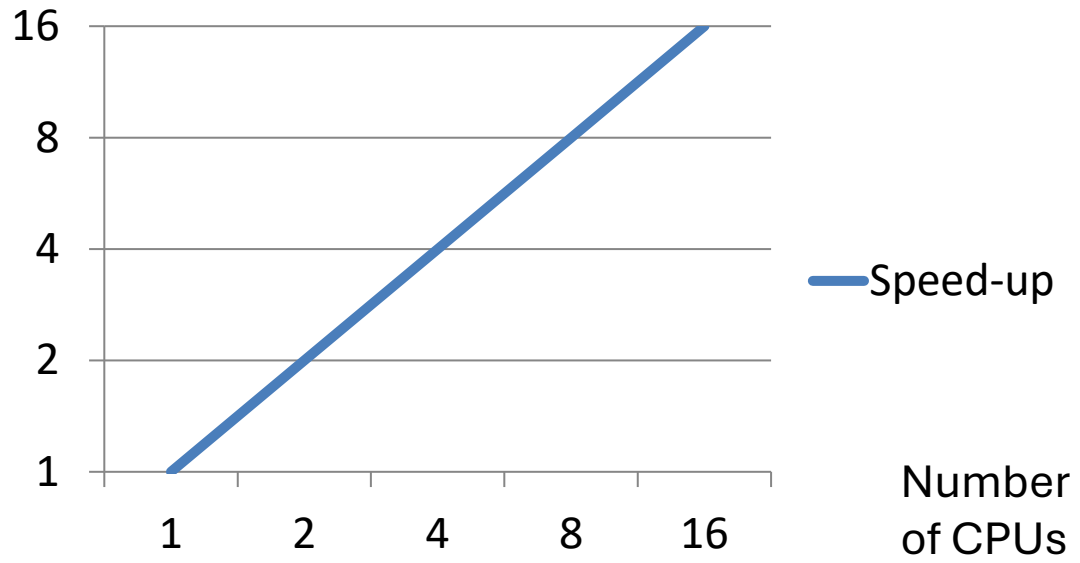
- How much faster can we get?
  - With 4 cores? 8 cores? 16 cores? …

It Depends.

Parallel Performance Metrics

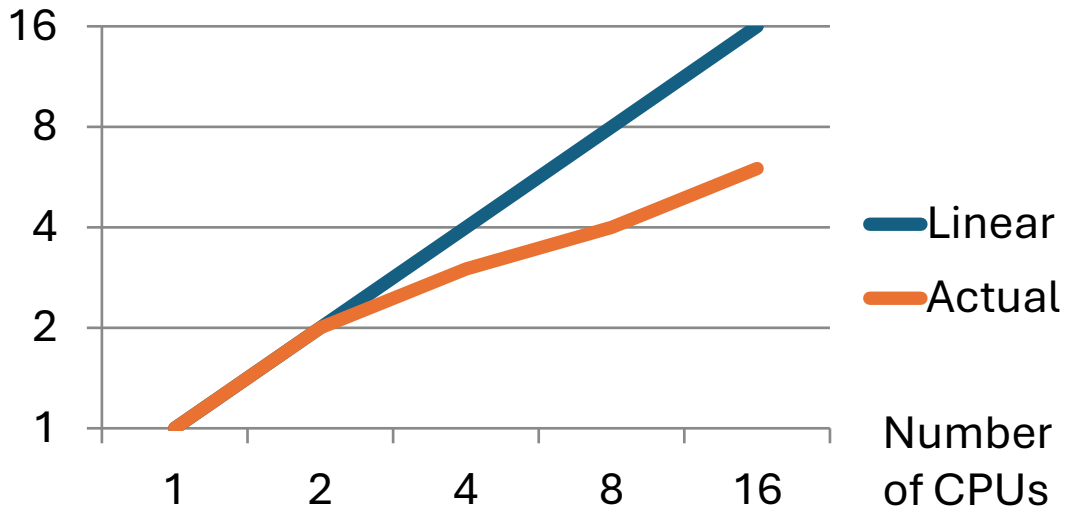
- Speed-up: Ratio of Sequential Time to Parallel
  Ts/Tp: Ts: time to execute sequential version
  Tp: time to execute a parallel version

- Ideal: linear speed-up as increase the number of processors (degree of parallelism)



Speed-up Reality

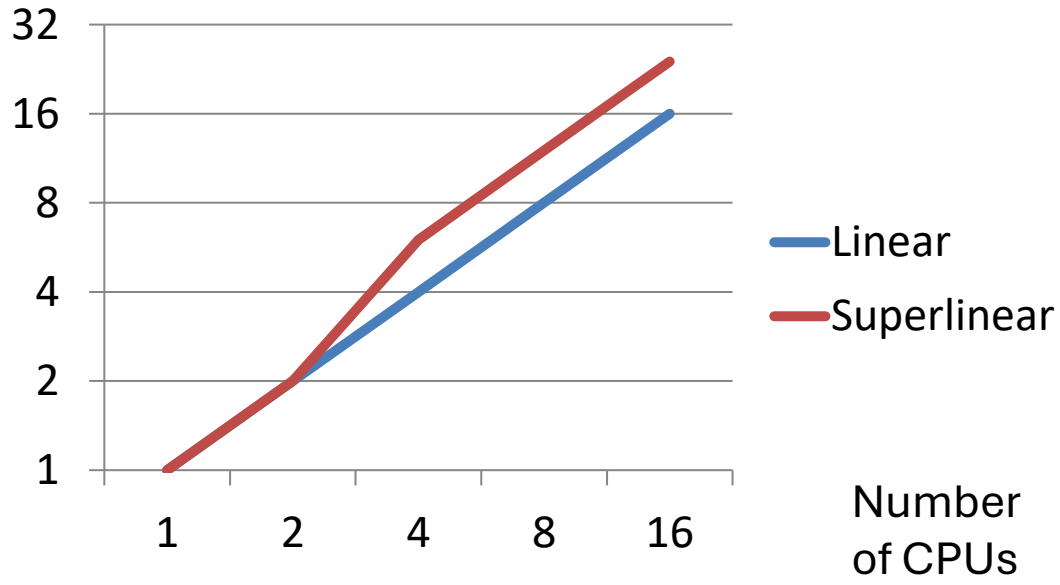
- Often unable to achieve ideal linear speedup



- Why?
  - Added overheads in parallel version of code:
    - thread create/join, synchronization, thread CXS
  - Often limits to parallelism
    - Need good amount of parallel computation between synch
    - Number CPUs limit number threads actually running at once

Speed-up Awesomeness!

- Sometimes parallel program can actually do better than linear speed-up



- Why?
  - Better locality with smaller problem sizes
  - More cache hits (faster memory accesses!)

Performance Limits

- Usually linear speed-up is hard to achieve
  - Embarrassingly parallel solutions often can achieve
    - Parallelism requiring basically no synchronizing actions
  - But most parallel solutions have some parallel overheads that interfere with achieving the ideal
    - Typically some points where threads need to synchronize their actions
- Can we quantify the limits to parallel speed-up?

Parallel program performance

How can we model the **speedup** if we use *c* cores?

In an ideal world, the speedup on *n* cores would be *n*

- Sometimes, more cores is slower. Why?
- Sometimes, it is faster (**superlinear** speedup). Why?

The speedup is the serial time divided by the parallelized time: T_1/T_c

T_1 = time our program takes with 1 core
T_c = time our program takes with *c* cores

Speedup

A serial program takes 10 seconds to run but only 2 seconds with 4 cores. What is the speed up?

A serial program takes 10 seconds to run but only 0.5 seconds with 8 cores. What is the speed up?

Efficiency

Efficiency is the speedup per core

$$\text{Efficiency}_c = \frac{T_1}{T_c \times c} = \frac{\text{speedup}_c}{c}$$

Varies from 0 to 1

- 0 => no benefit to parallelism
- 1 => all cores used perfectly

Exercise: Efficiency

Exercise: $T_1 = 60s$, $T_2 = 30s$. Compute efficiency.

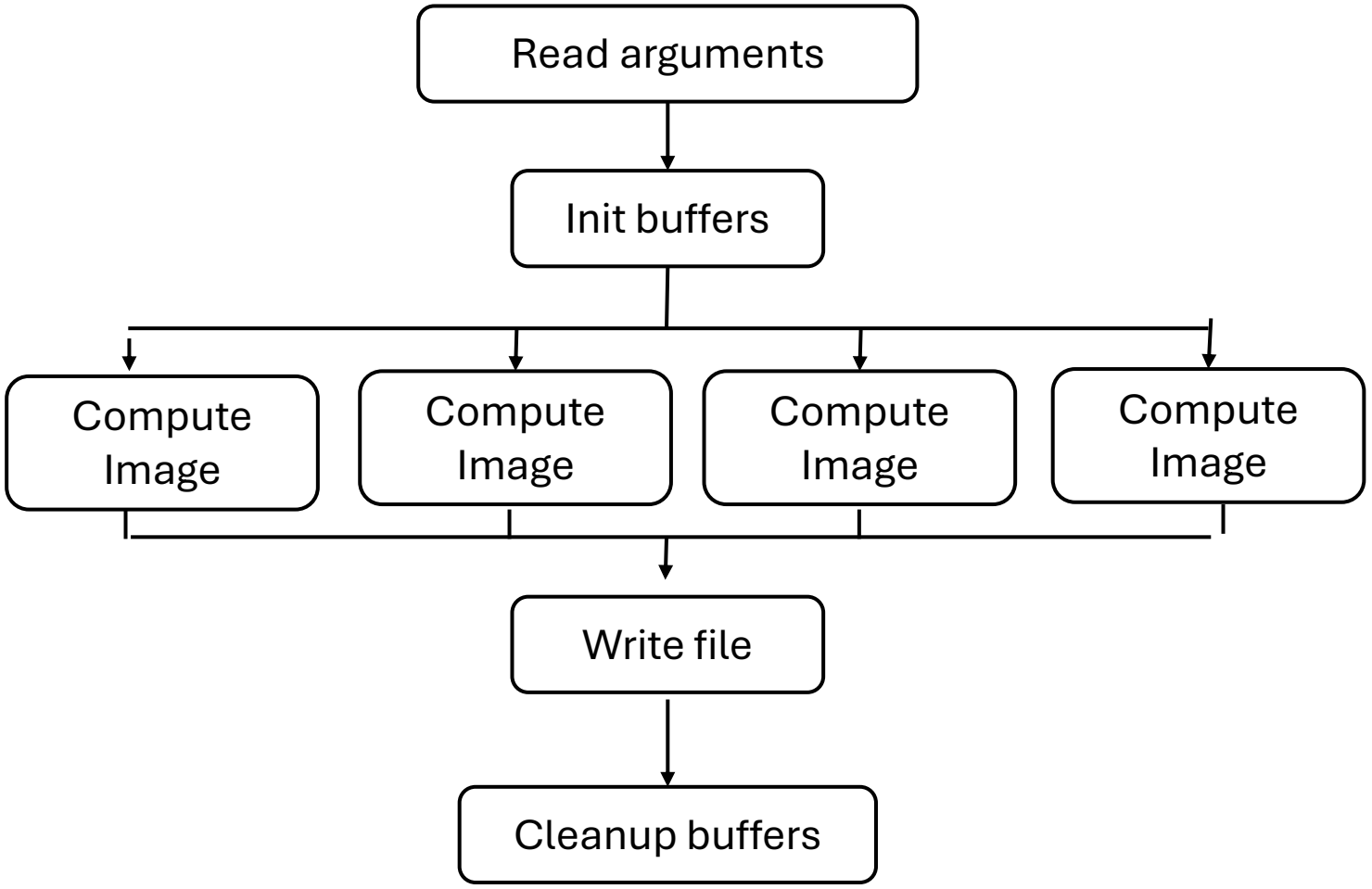
Exercise: $T_1 = 60s$, $T_4 = 30s$. Compute efficiency.

Critical path

In practice, some parts of the code have to be done serially

The **critical path** is the largest sequence of serial commands in the program

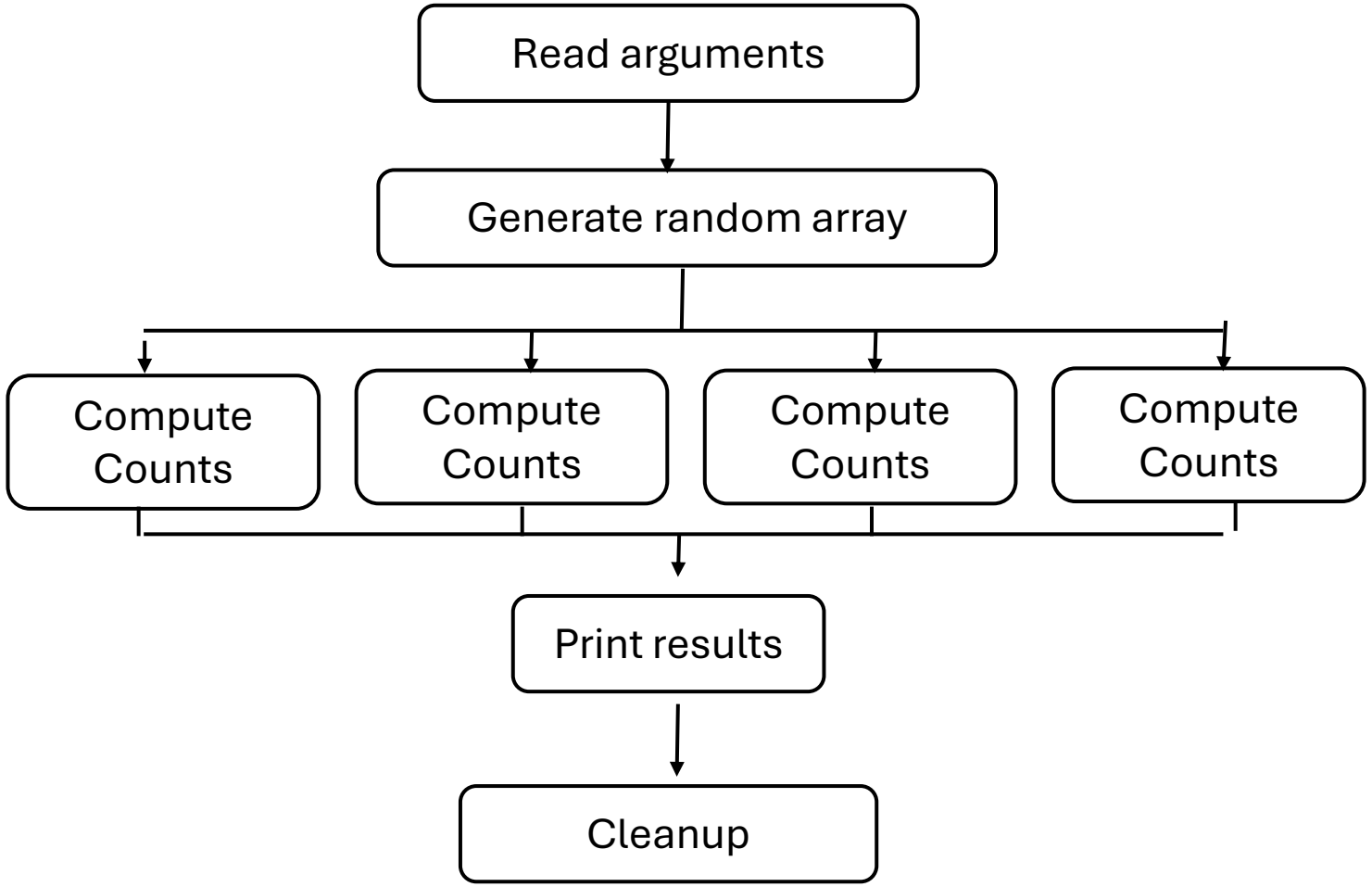
E.g. the lower bound on how much parallelism can speed up our program



Example: Multi-Mandelbrot

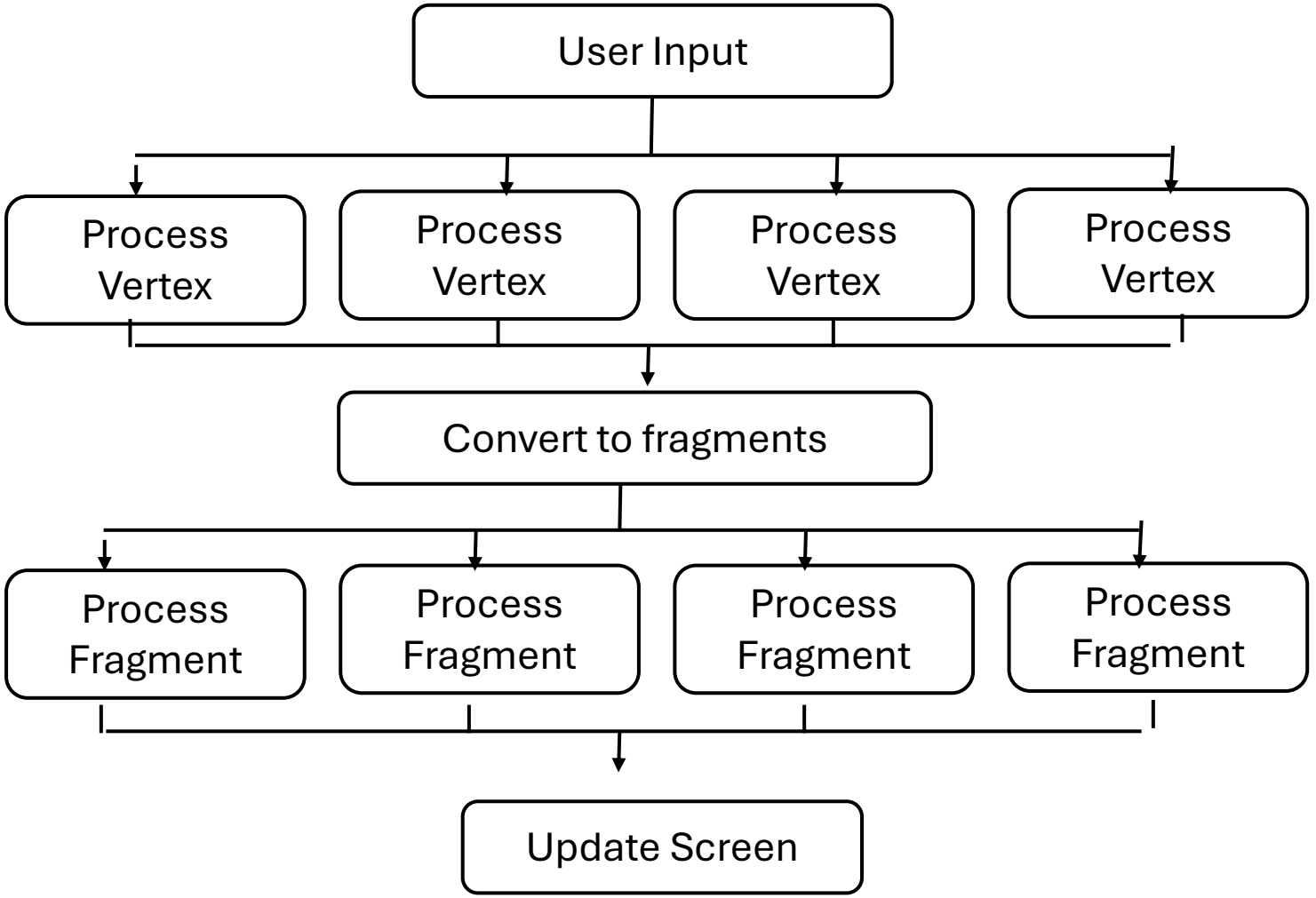
countElems

What is the critical path?



What is the critical path?

Graphics Pipeline



Example: countElems

A threaded example that implements a CountSort algorithm

CountSort

- Data includes digits 0 through 9
- Count number of occurrences for each digit
- List each digit in order using the counts

CountSort Example

A = [9, 0, 2, 7, 9, 0, 1, 4, 2, 2, 4, 5, 0, 9, 1]

counts =

Sorted A =

Exercise: countElems algorithm

Write a parallel algorithm that implements CountSort for the digits 0 through 9

1. What data structures do we need?
2. What can be done in parallel?
3. What can be done serially?

Sketch a design for a parallel program.

Example: countElems

A threaded example that implements a CountSort algorithm

Table 130. Performance Benchmarks

Number of threads	**2**	**4**	**8**
Speedup	1.68	2.36	3.08
Efficiency	0.84	0.59	0.39

Ideal speedup is not met

- Thread overhead
- Serial components of algorithm start to dominate the performance

Amdahl's Law

Tells us to focus our efforts on the hot spots:

Goal: Look to parallelize (or optimize in any way) the
portion of code that accounts for the largest portion of execution time

The critical path will put an upper bound on the performance gain

- If one part accounts for 10% of execution time, the best we can do is completely optimize this portion away
- At best, the overall effects on performance are reducing total runtime by 10%

Where is all the time?

- Usually in loops
- Could also be implicit and explicit I/O costs

Ahmdahl's Law

Models the speedup of adding more cores *when the problem size remains the same*

Let S = fraction of the program that is serial
Let P = fraction of the program that is parallel

} S+P = 1

$$T_c = S \times T_1 + \frac{P}{c} \times T_1$$

} Models the diminishing returns of having more threads

What happens to this measure when S >> P?
When P >> S?
What happens as c (number of cores) reaches infinity?

Amdahl's Law

Execution time after improvement:

```
(time effected by parallelism)/(amount of improvement)
 +  (time not effected by improvement)
```

Example: 90% of application can be parallelized

10 processors:

$T_p = .9*T_s/10 + .1*T_s = .19*T_s$

Speed-up = T_s/T_p = **~5.3**

Compute the speed-up for 100 processors:

T_p =

Speed-up = T_s/T_p =

Compute the speed-up for 1000 processors:

T_p =

Speed-up =

Example

Suppose a program is 90% parallelizable and T1 = 10 seconds

#Cores	Serial Time (s)	Parallel Time (s)	Total Time (s)	Speedup: T1/Tc
1	1	9	10	1
10				
100				
1000				

Ahmdahl’s Law - Speedup

Using Ahmdahl’s Law, we can compute the theoretical best case speedup if we had infinite cores

Compute a formula for the speedup

Suppose S = 0.1 and c -> infinity, what is the speedup?