DAG EXECUTION MODEL, WORK AND DEPTH

Computational Complexity of (Sequential) Algorithms

- Model: Each step takes a unit time

- Determine the time (/space) required by the algorithm as a function of input size

Sequential Sorting Example

- Given an array of size n
- MergeSort takes O(n . log n) time
- BubbleSort takes $O(n^2)$ time

Sequential Sorting Example

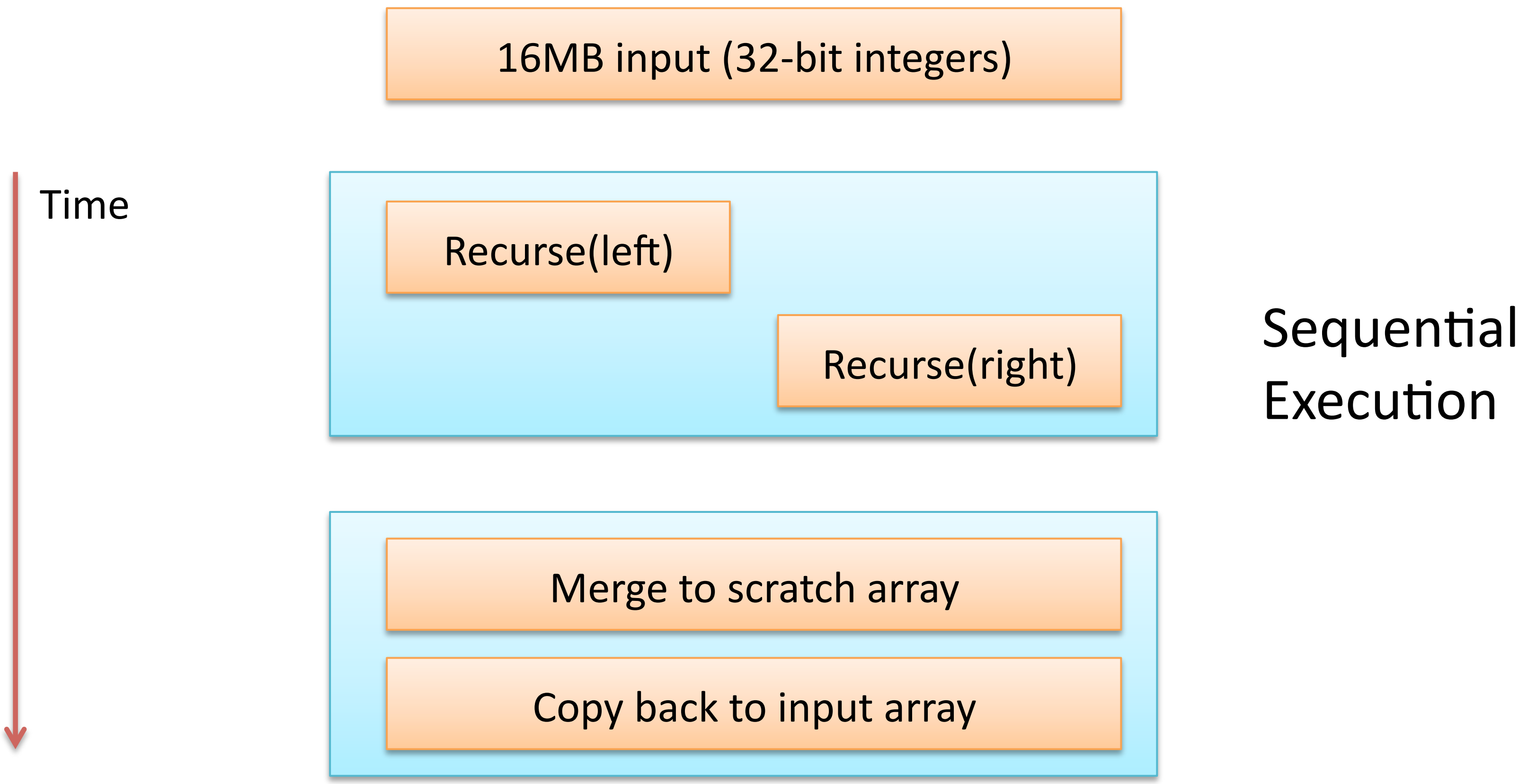
- Given an array of size n
- MergeSort takes O(n . log n) time
- BubbleSort takes $O(n^2)$ time
- But, a BubbleSort implementation can sometimes be faster than a MergeSort implementation
- Why?

Sequential Sorting Example

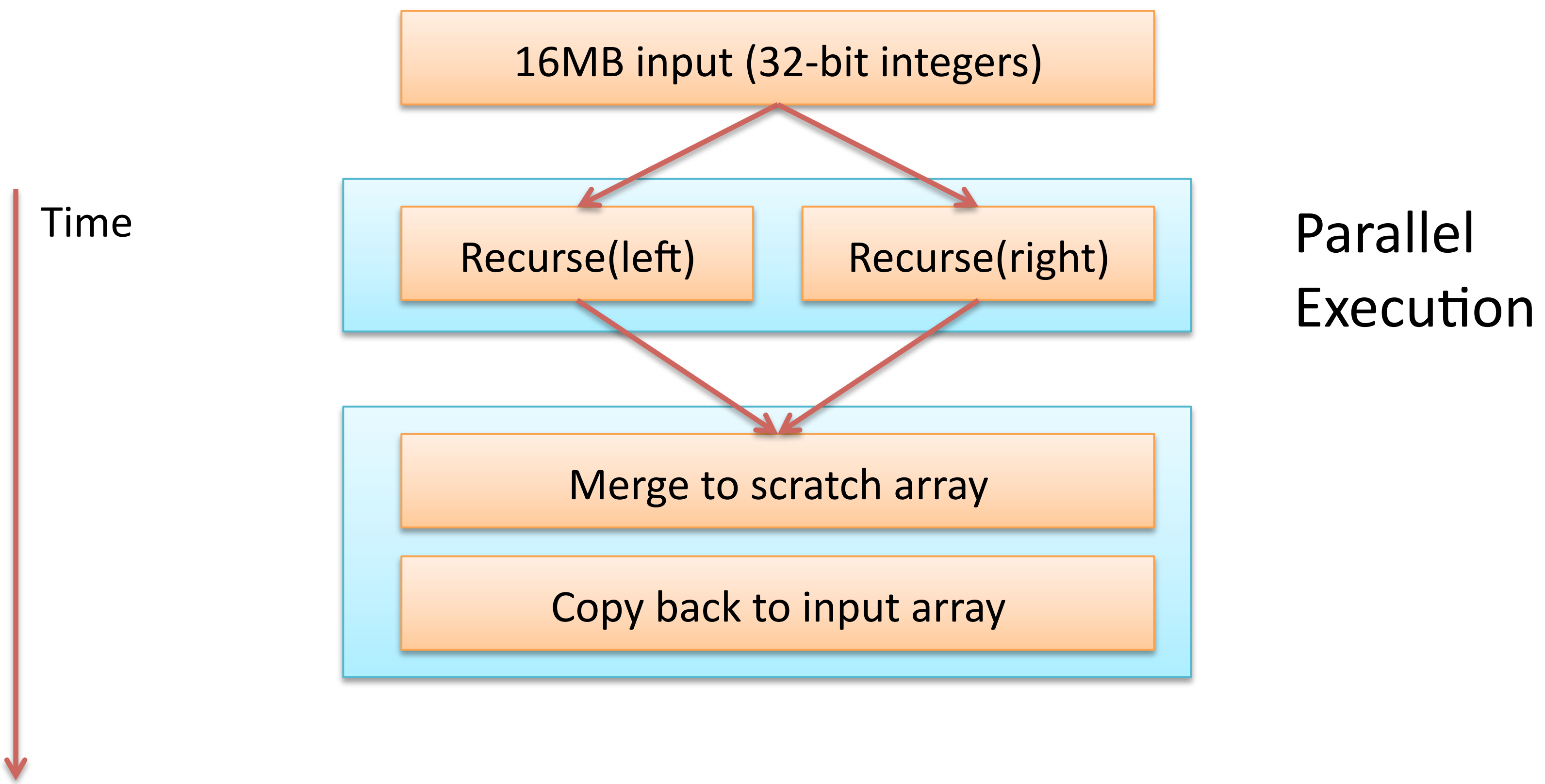
- Given an array of size n
- MergeSort takes O(n . log n) time
- BubbleSort takes $O(n^2)$ time
- But, a BubbleSort implementation can sometimes be faster than a MergeSort implementation
- The model is still useful
  - Indicates the scalability of the algorithm for large inputs
  - Lets us prove things like a sorting algorithm requires at least O(n. log n) comparisions

We need a similar model for parallel algorithms

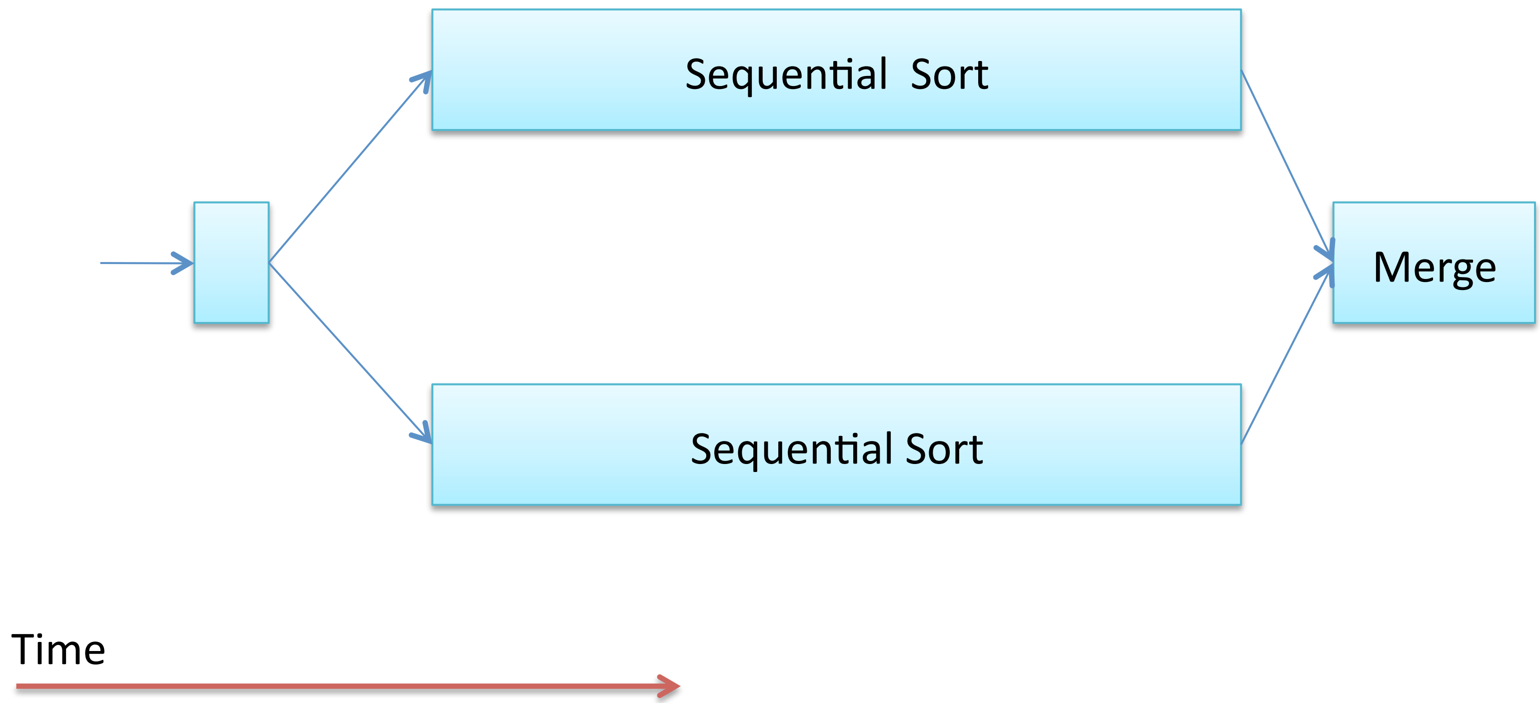
Sequential Merge Sort



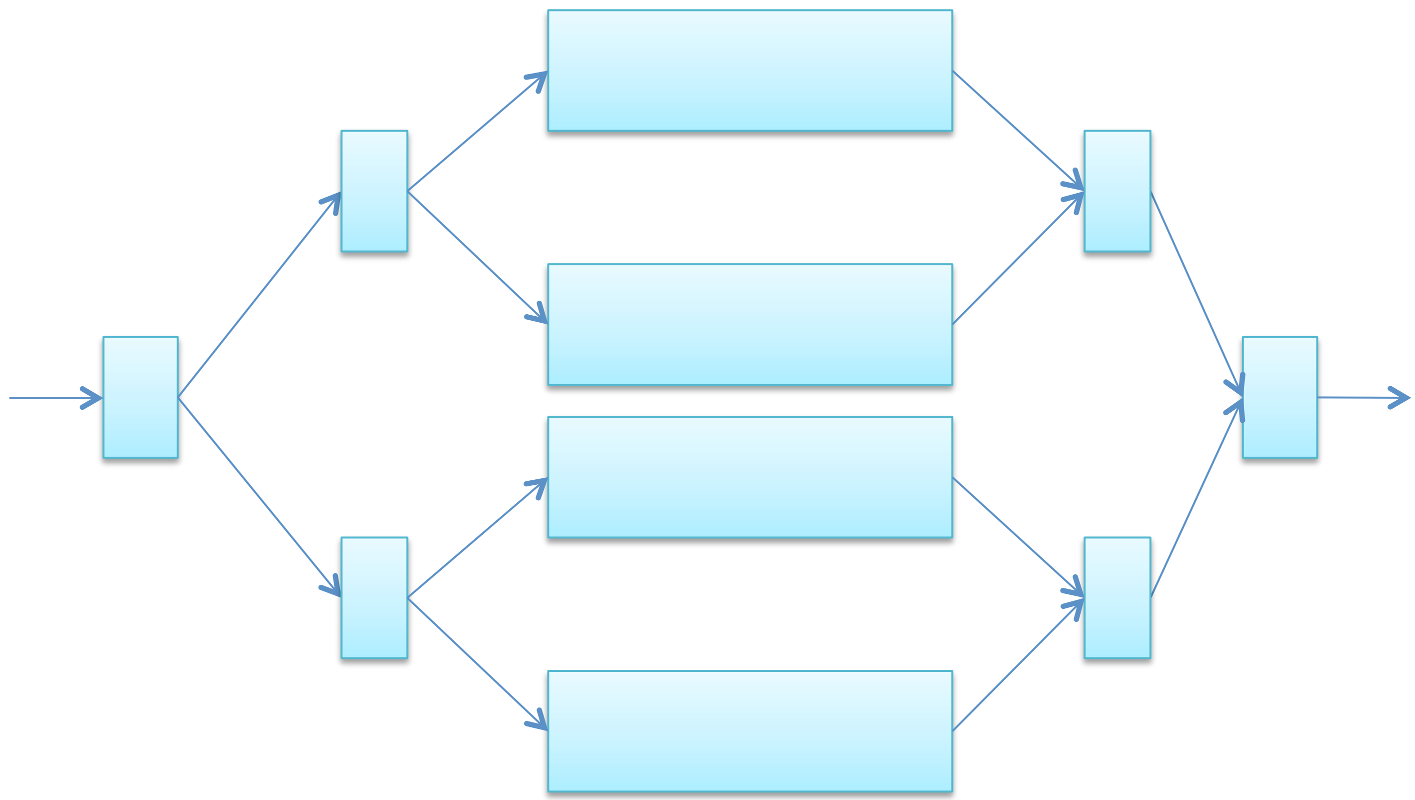
Parallel Merge Sort (as Parallel Directed Acyclic Graph)



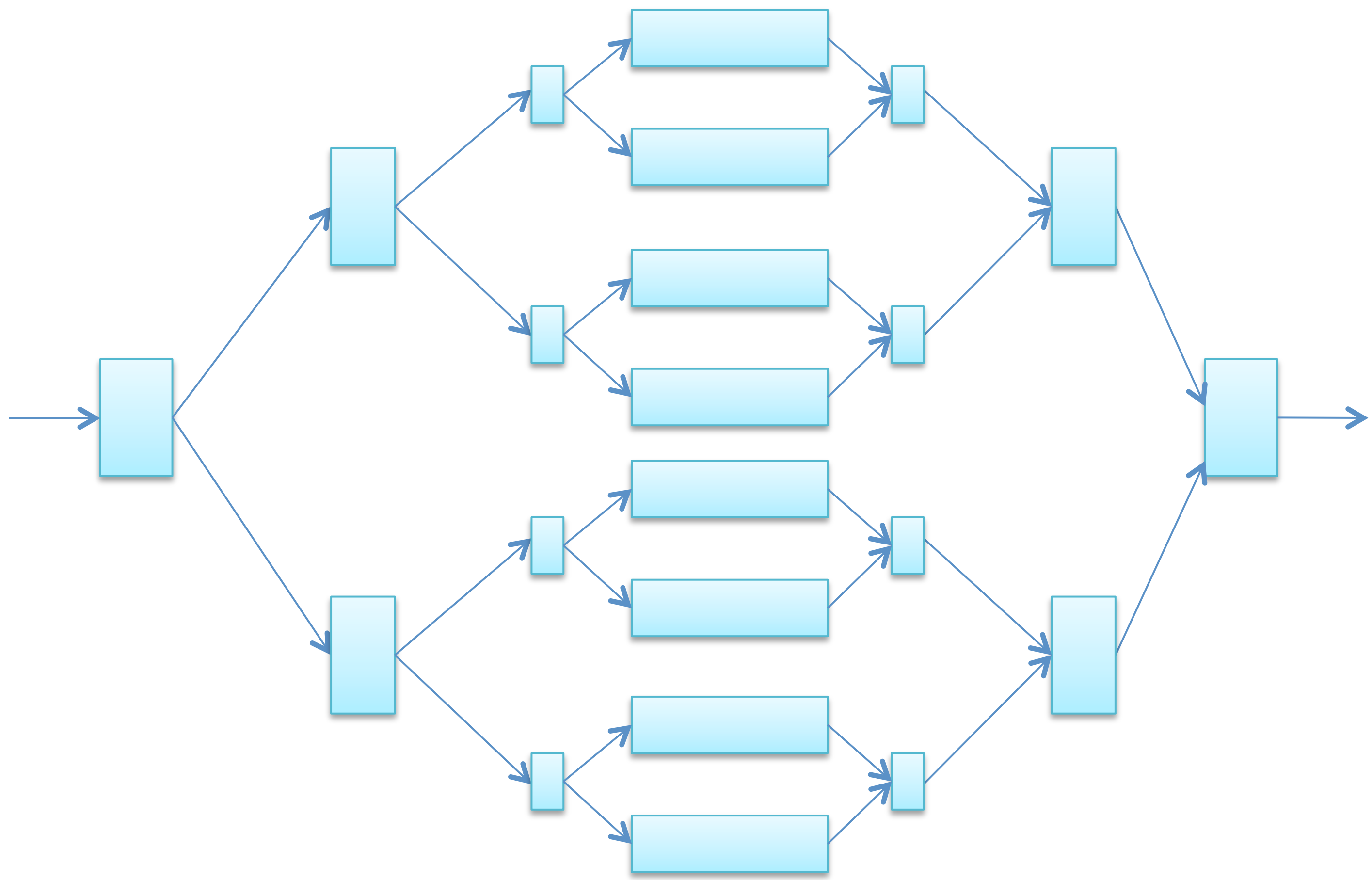
Parallel DAG for Merge Sort (2-core)



Parallel DAG for Merge Sort (4-core)



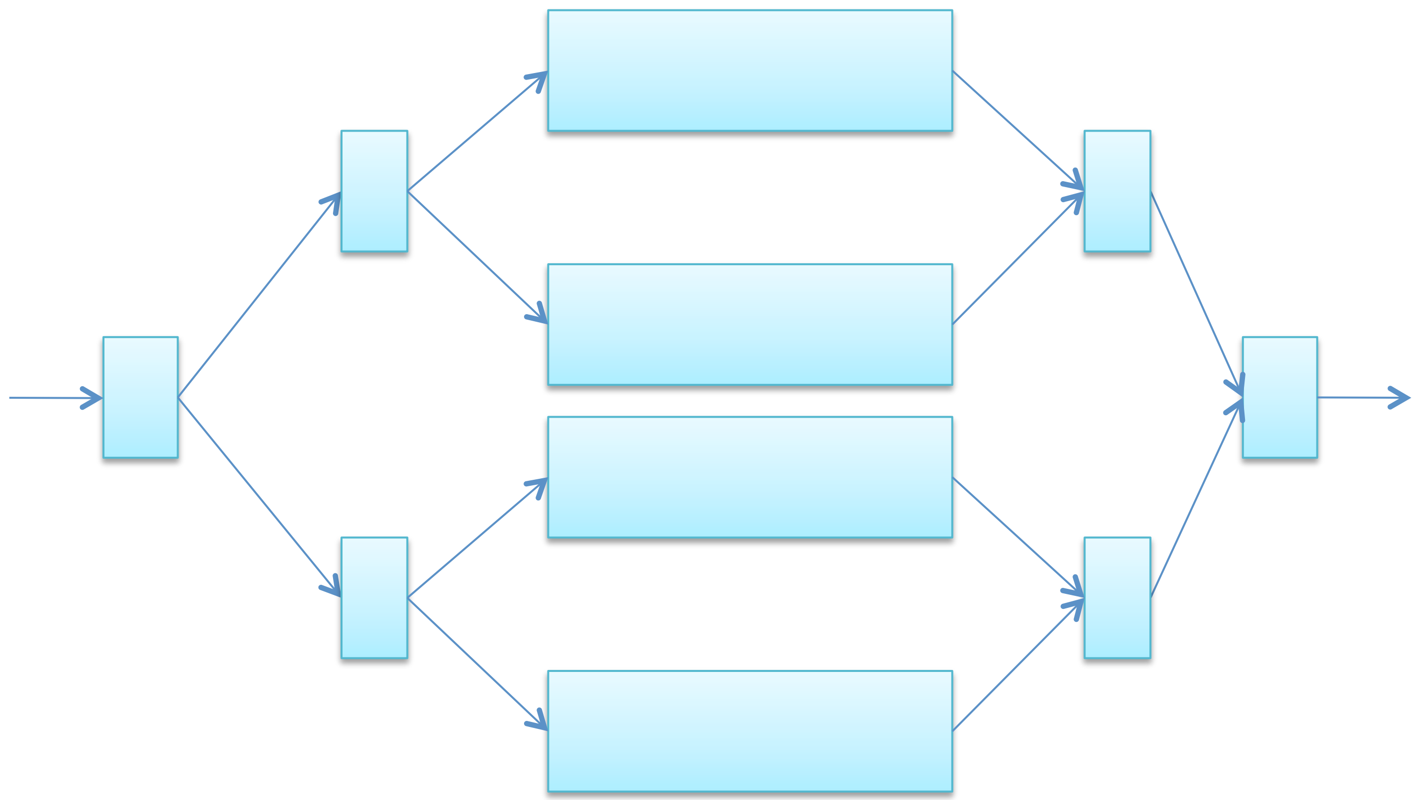
Parallel DAG for Merge Sort (8-core)



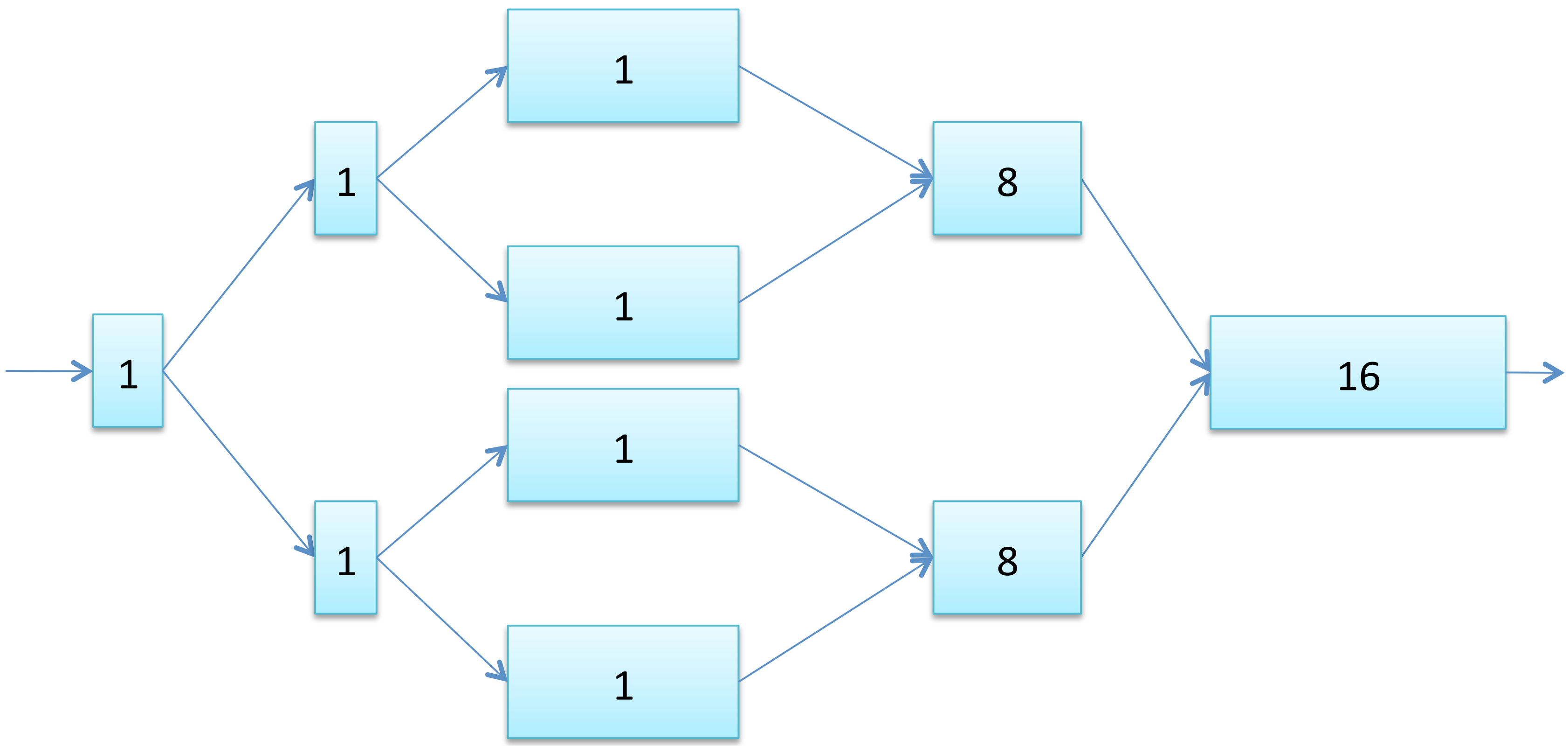
The DAG Execution Model of a Parallel Computation

- Given an input, dynamically create a DAG
- Nodes represent sequential computation
  - Weighted by the amount of work
- Edges represent dependencies:
  - Node A → Node B means that B cannot be scheduled unless A is finished

Sorting 16 elements in four cores



Sorting 16 elements in four cores (4 element arrays sorted in constant time)



Performance Measures

- Given a graph G, a scheduler S, and P processors

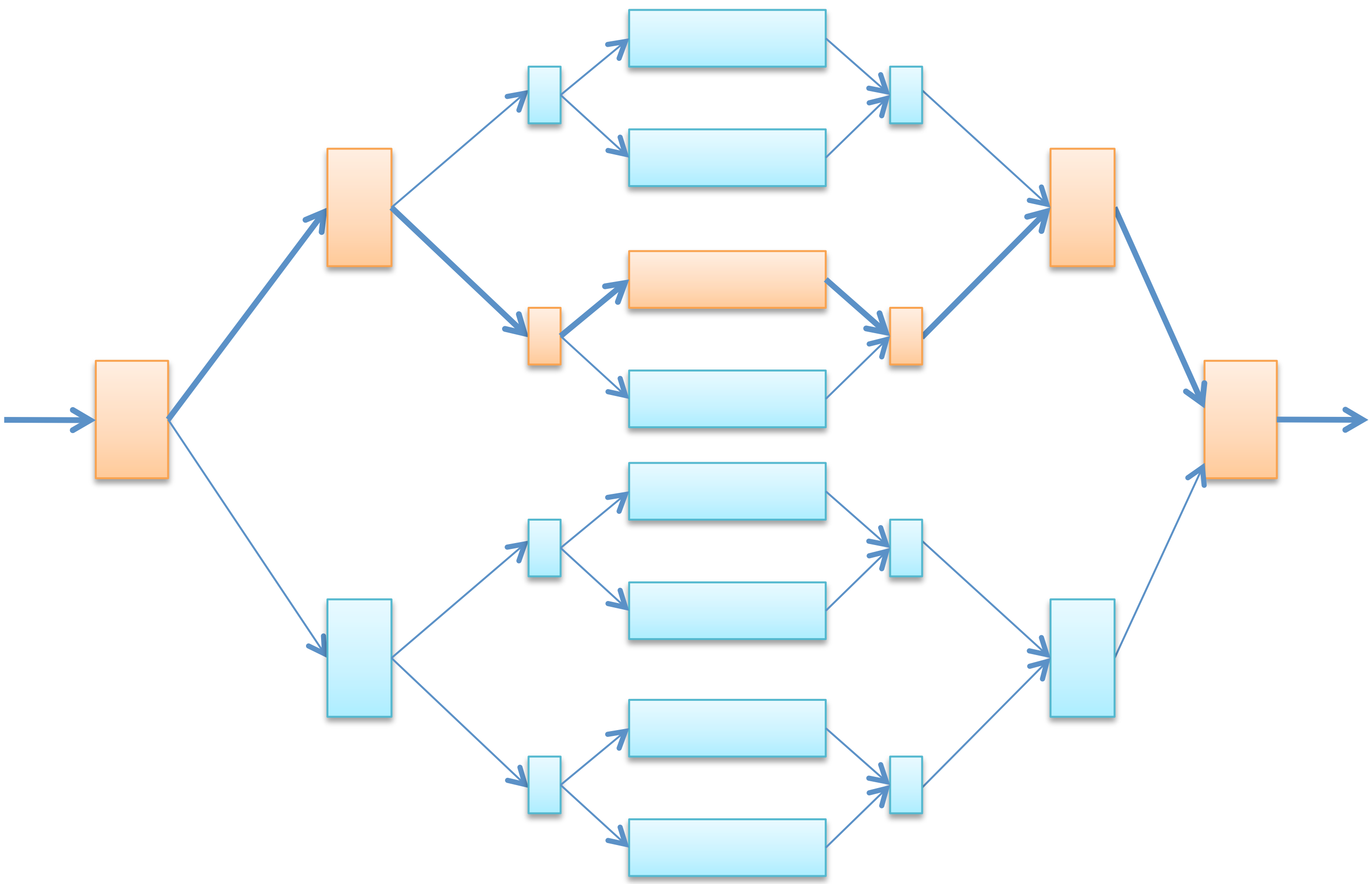
- $T_P(S)$: time on P processors using scheduler S
- T_P : time on P processors for the best scheduler
- T_1 : time on a single processor (sequential cost)
- T_∞ : time assuming infinite resources

Work and Depth

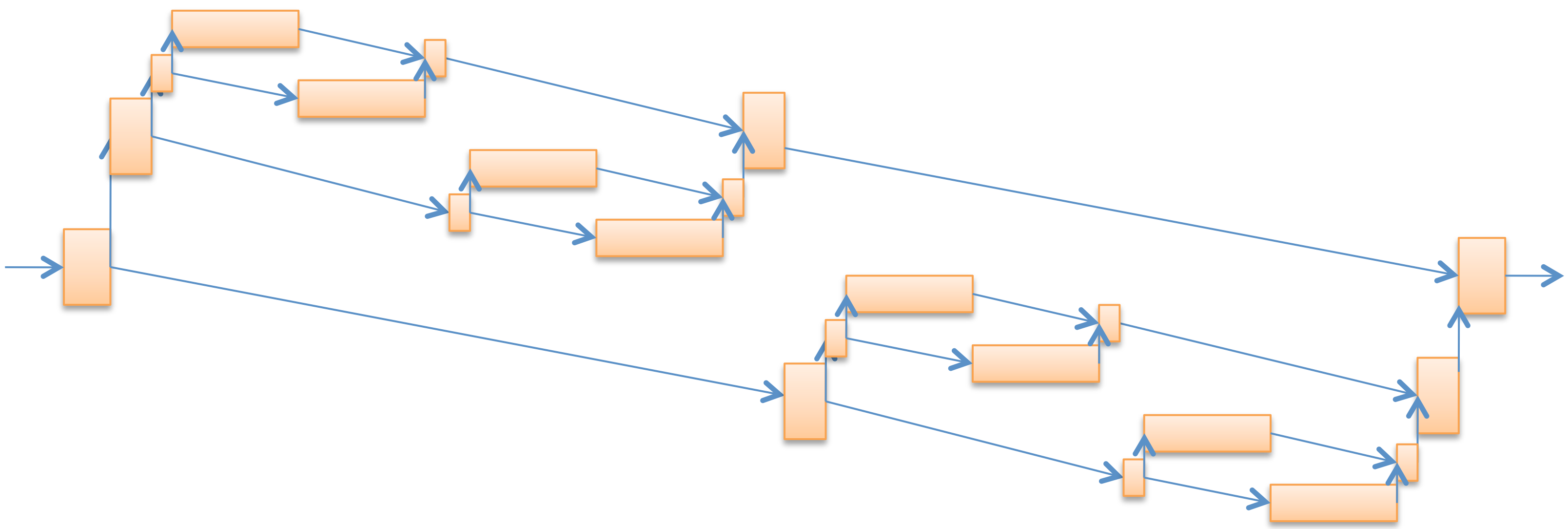
- T_1 = Work
  - The total number of operations executed by a computation

- T_∞ = Depth
  - The longest chain of sequential dependencies (critical path) in the parallel DAG
  - Also called as Span

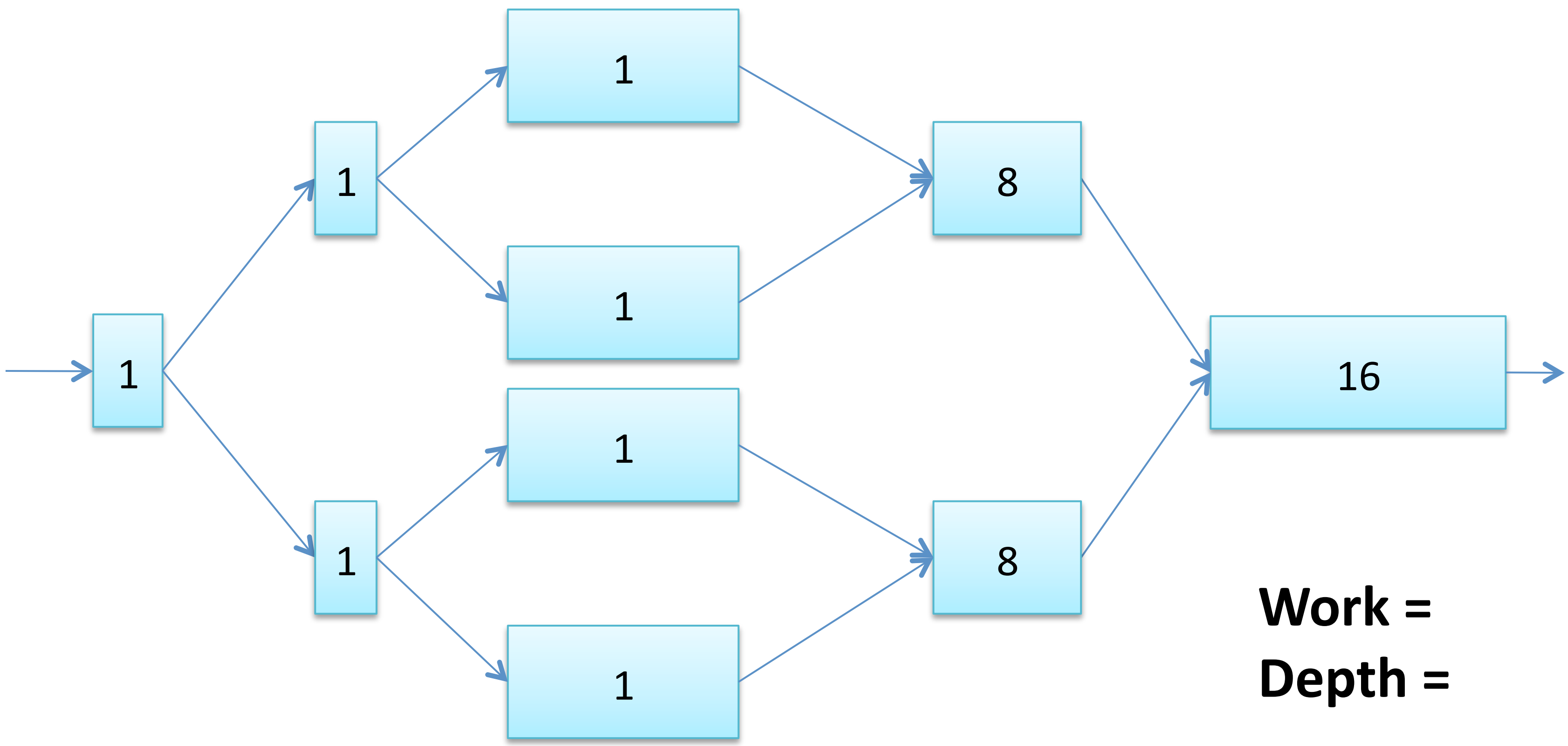
T_∞ (Depth): Critical Path Length (Sequential Bottleneck)



T_1 (work): Time to Run Sequentially



Sorting 16 elements in four cores (4 element arrays sorted in constant time)



Work =
Depth =

Some Useful Theorems

Work Law

- "You cannot avoid work by parallelizing"

$$\frac{T_1}{P} \leq T_P$$

Work Law

- "You cannot avoid work by parallelizing"

$$\frac{T_1}{P} \leq T_P$$

- Speedup = $\dfrac{T_1}{T_P} \leq \mathrm{P}$

Work Law

- "You cannot avoid work by parallelizing"

$$\frac{T_1}{P} \leq T_P$$

- Speedup = $\frac{T_1}{T_P} \leq \mathrm{P}$
- Can speedup be more than 2 when we go from 1-core to 2-cores, in practice?

Depth Law

- More resources should make things faster
- You are limited by the sequential bottlenec

$$T_P \geq T_\infty$$

Amount of Parallelism

-

$$\text{Parallelism} = \frac{T_1}{T_\infty}$$

Maximum Speedup Possible

Speedup $$\frac{T_1}{T_P} \leq \frac{\boldsymbol{T_1}}{\boldsymbol{T_\infty}}$$ **Parallelism**

"speedup is bounded above
by available parallelism"

Greedy Scheduler

- If more than P nodes can be scheduled, pick any subset of size P
- If less than P nodes can be scheduled, schedule them all

Performance of the Greedy Scheduler

- $$T_P(Greedy) \leq \frac{T_1}{P} + T_\infty$$

Performance of the Greedy Scheduler

- $$T_P(Greedy) \le \frac{T_1}{P} + T_\infty$$

Note:

Work law: $\frac{T_1}{P} \le T_P$

Depth law: $T_\infty \le T_P$

Greedy is optimal within a factor of 2

- $$T_P \leq T_P(Greedy) \leq 2.T_P$$

Note:

Work law: $\frac{T_1}{P} \leq T_P$

Depth law: $T_\infty \leq T_P$

Work/Depth of Merge Sort (Sequential Merge)

- Work $\boldsymbol{T_1}$: $O(n \log n)$
- Depth $\boldsymbol{T_\infty}$: $O(n)$
  - Takes $O(n)$ time to merge n elements

- Parallelism:
  - $\frac{\boldsymbol{T_1}}{\boldsymbol{T_\infty}}$: $O(\log n)$ - really bad!

Main Message

- Analyze the Work and Depth of your algorithm
- Parallelism is Work/Depth
- Try to decrease Depth
  - the critical path
  - a <u>*sequential*</u> bottleneck
- If you increase Depth
  - better increase Work by a lot more!

Amdahl's law

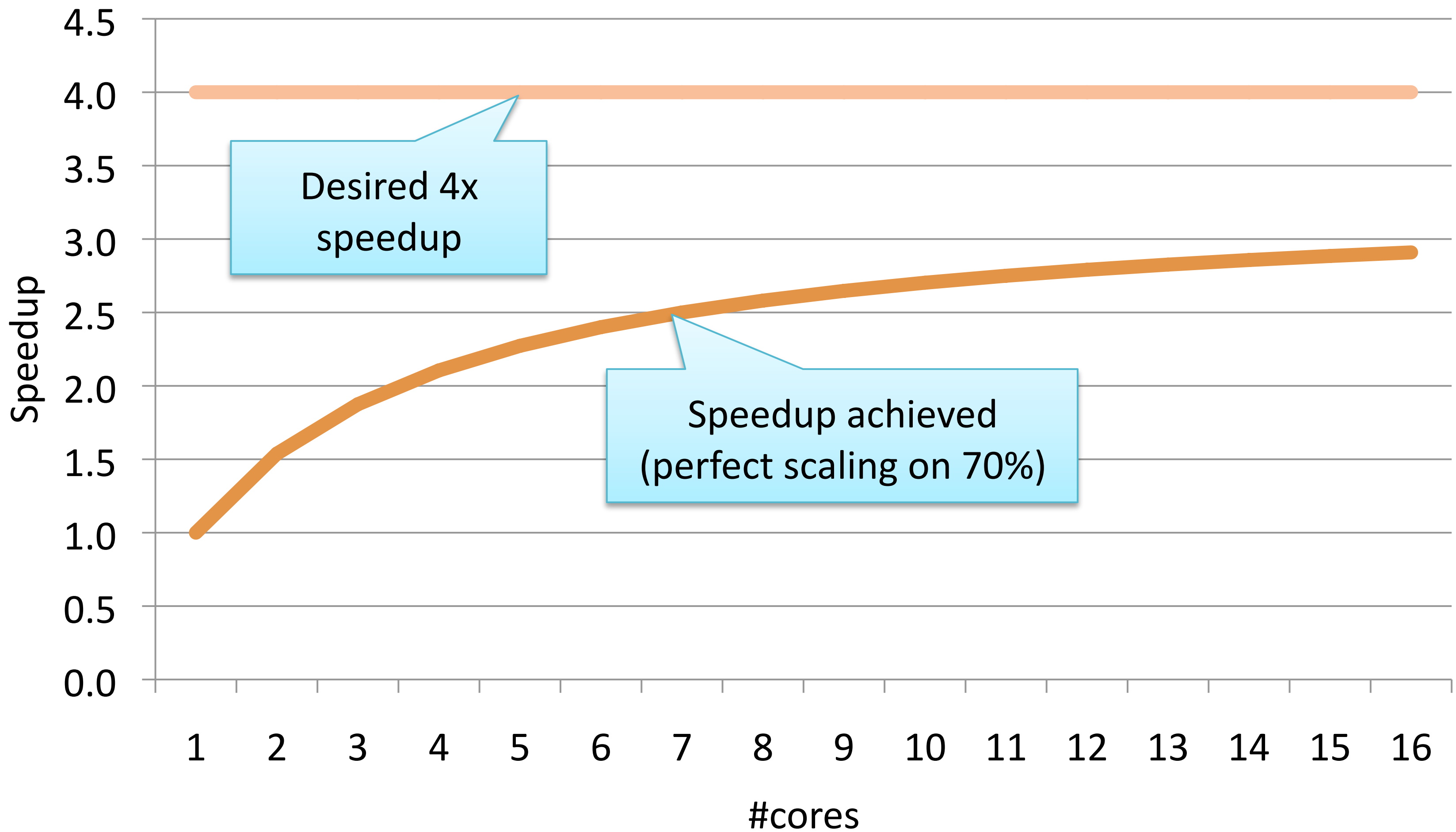
- Sorting takes 70% of the execution time of a sequential program
- You replace the sorting algorithm with one that scales perfectly on multi-core hardware
- How many cores do you need to get a 4x speed-up on the program?

Amdahl's law, $f = 70\%$

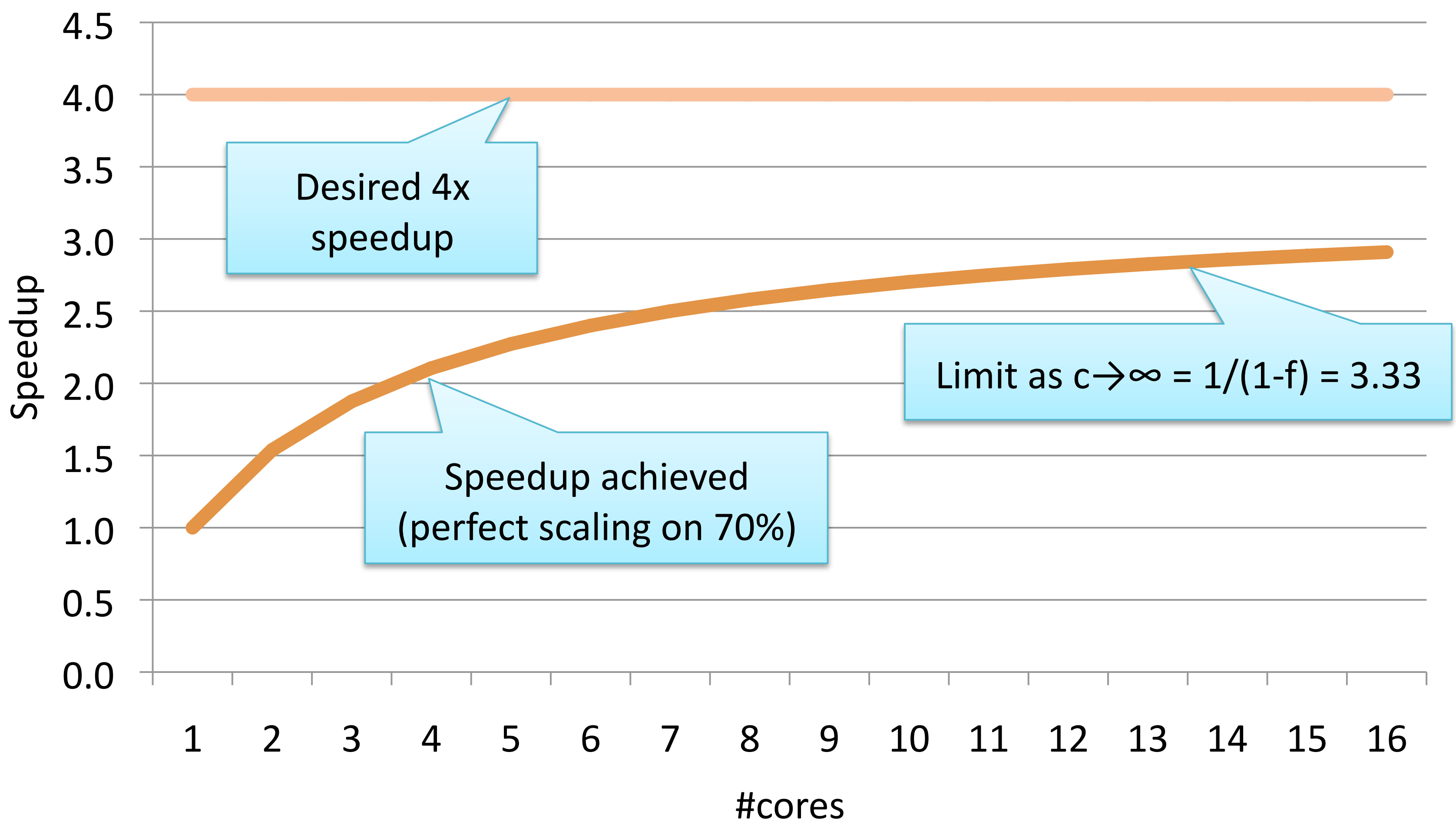
$$Speedup(f, c) = \frac{1}{(1-f) + \frac{f}{c}}$$

f	= the parallel portion of execution
$(1-f)$	= the sequential portion of execution
c	= number of cores used

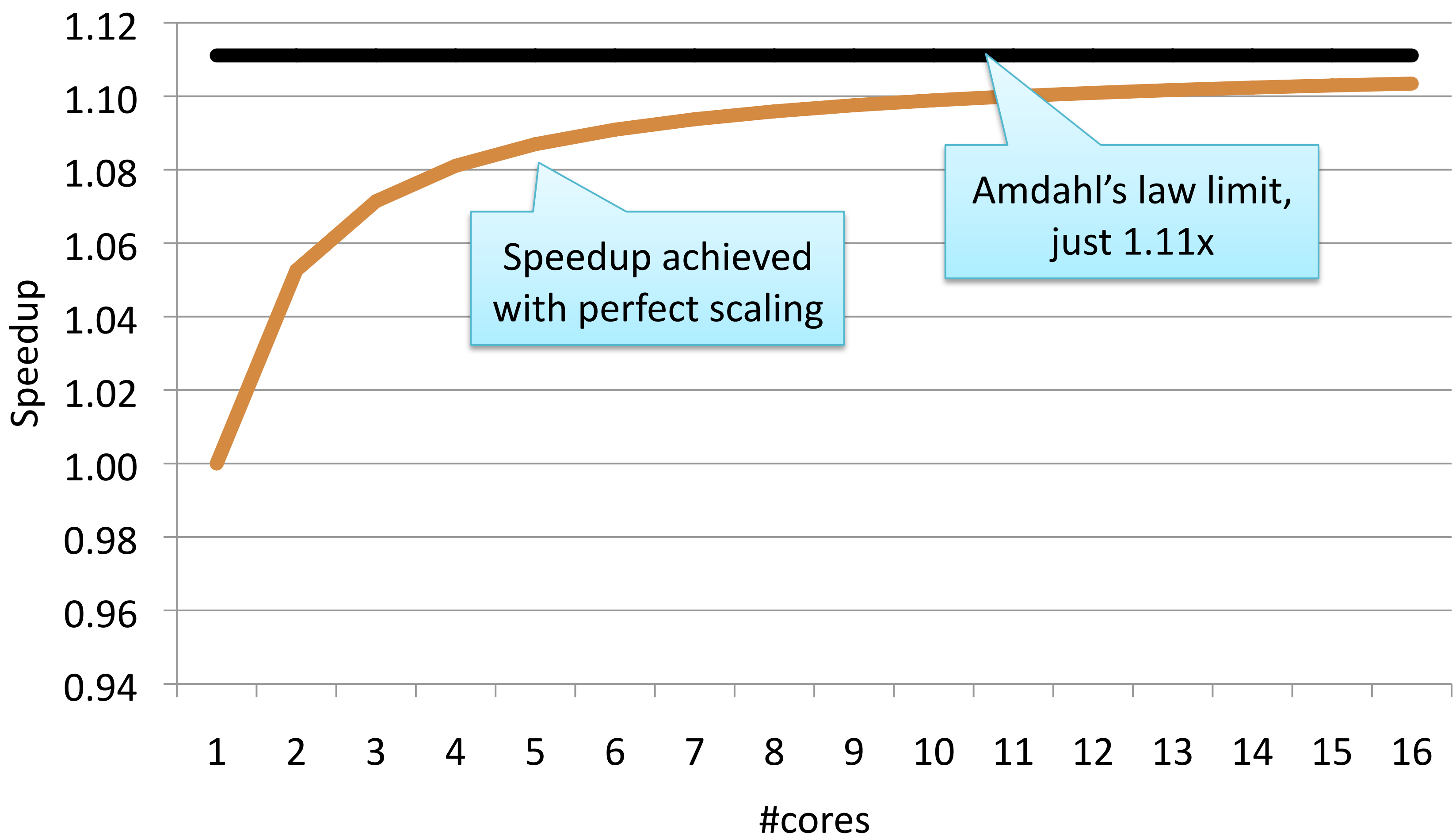
Amdahl's law, $f = 70\%$



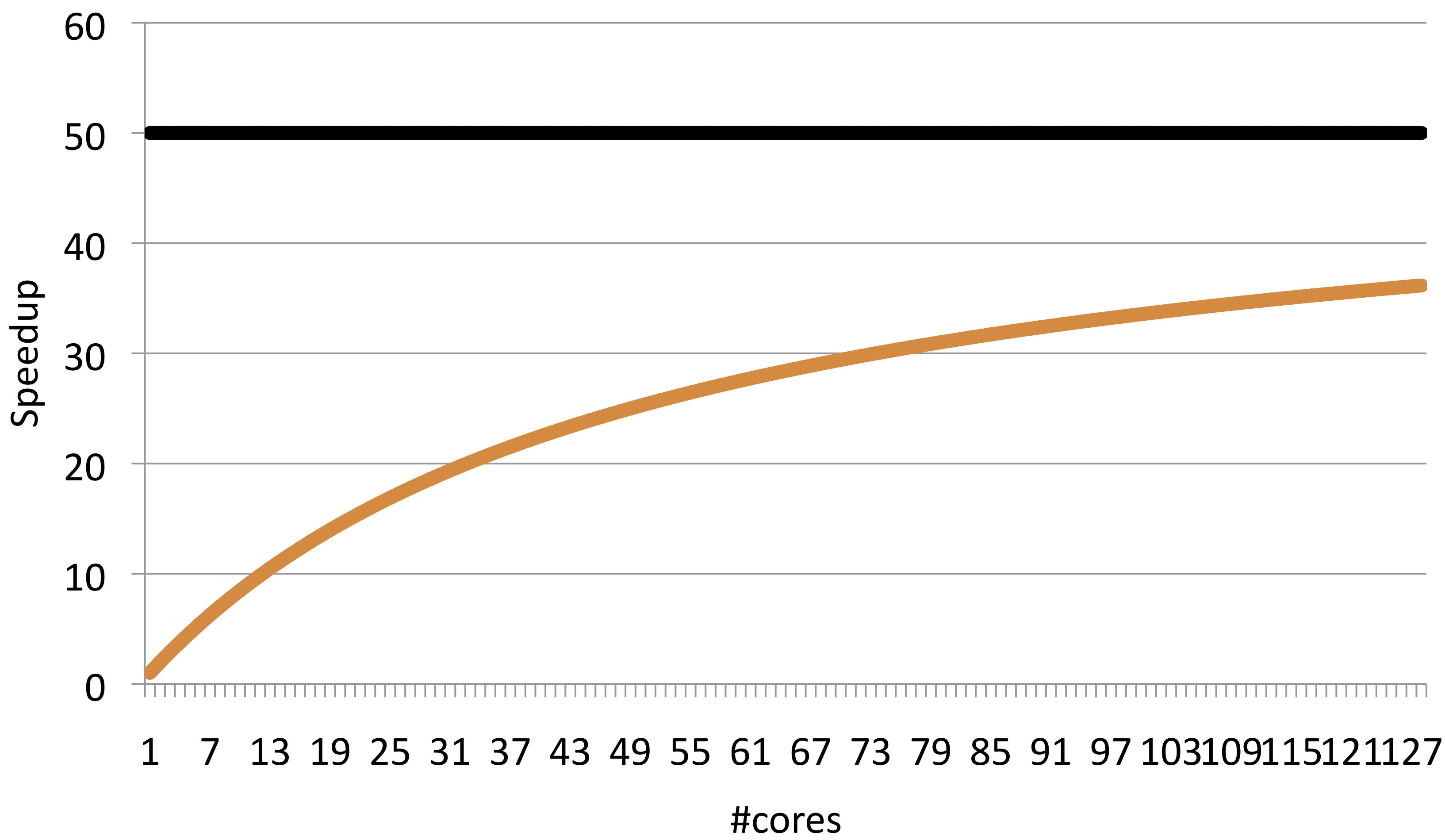
Amdahl's law, $f = 70\%$



Amdahl's law, $f = 10\%$



Amdahl's law, $f = 98\%$



Lesson

- Speedup is limited by <u>sequential</u> code

- Even a small percentage of <u>sequential</u> code can greatly limit potential speedup

Gustafson's Law

Any sufficiently large problem can be parallelized effectively

$$Speedup(f, c) = fc + (1 - f)$$

f	= the <u>parallel</u> portion of execution
$(1 - f)$	= the <u>sequential</u> portion of execution
c	= number of cores used

Key assumption: f increases as problem size increases