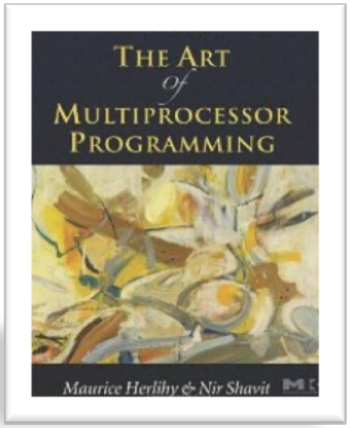
DAG EXECUTION MODEL, WORK STEALING

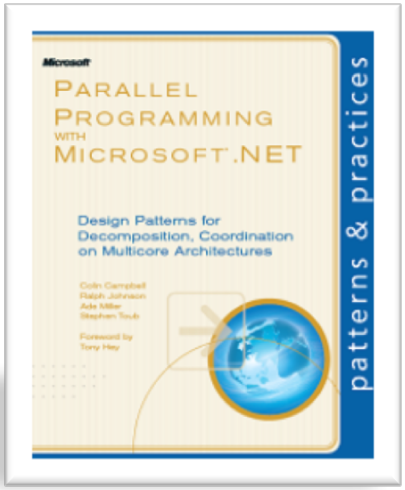
CISTER Summer Internship 2017

Luís Nogueira lmn@isep.ipp.pt

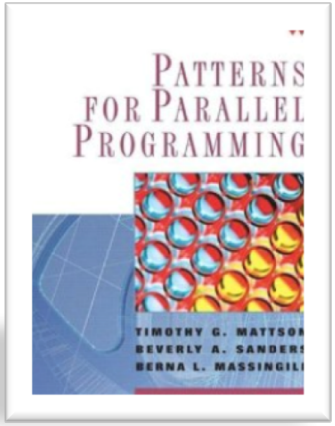
Recommended textbooks



The Art of Multiprocessor Programming
Maurice Herlihy, Nir Shavit



Parallel Programming with Microsoft .NET
http://parallelpatterns.codeplex.com/



Patterns for Parallel Programming
Timothy Mattson, et.al.

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
- Merge Sort takes O(n . log n) time
- Bubble Sort takes $O(n^2)$ time

Sequential sorting example

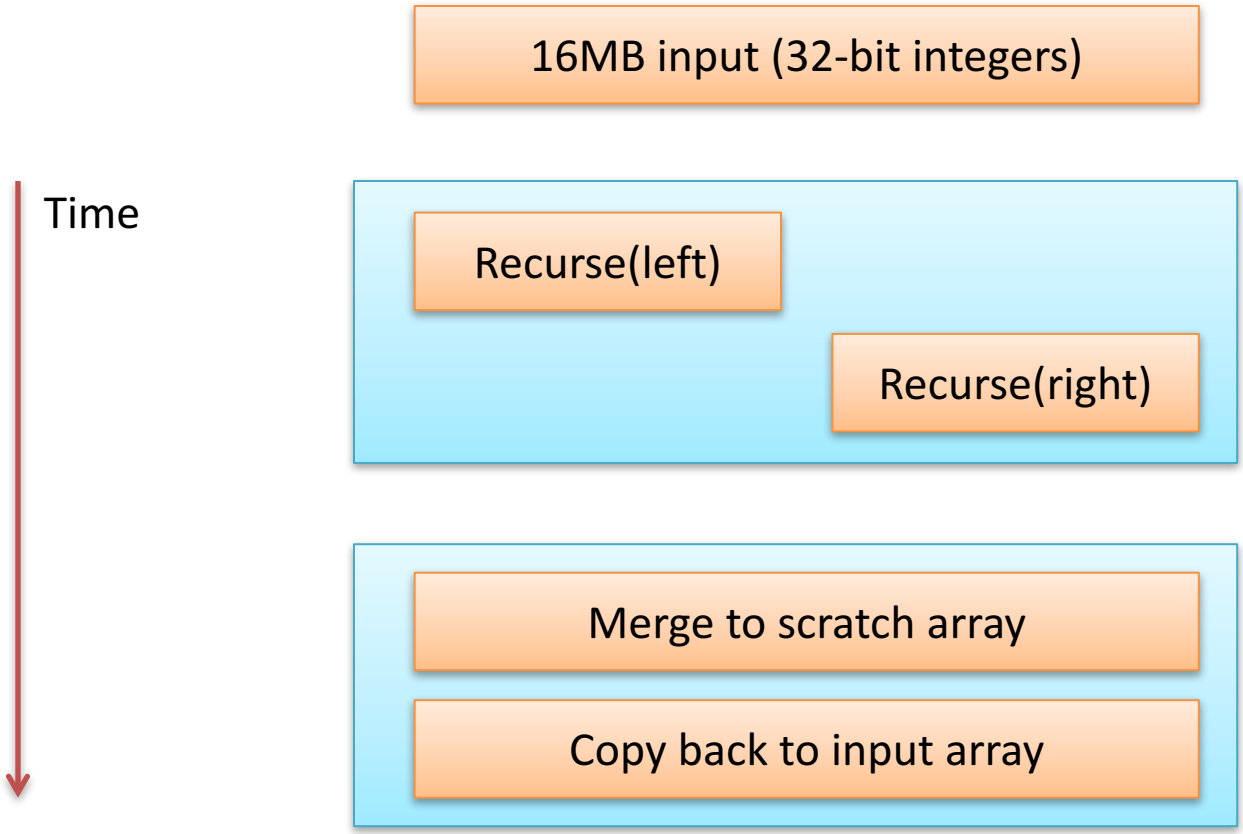
- Given an array of size n
- Merge Sort takes O(n . log n) time
- Bubble Sort takes $O(n^2)$ time
- But, a Bubble Sort implementation can sometimes be faster than a Merge Sort implementation
- Why?

Sequential sorting example

- Given an array of size n
- Merge Sort takes O(n . log n) time
- Bubble Sort takes $O(n^2)$ time
- But, a Bubble Sort implementation can sometimes be faster than a Merge Sort 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) comparisons

We need a similar model for parallel algorithms

Sequential Merge Sort

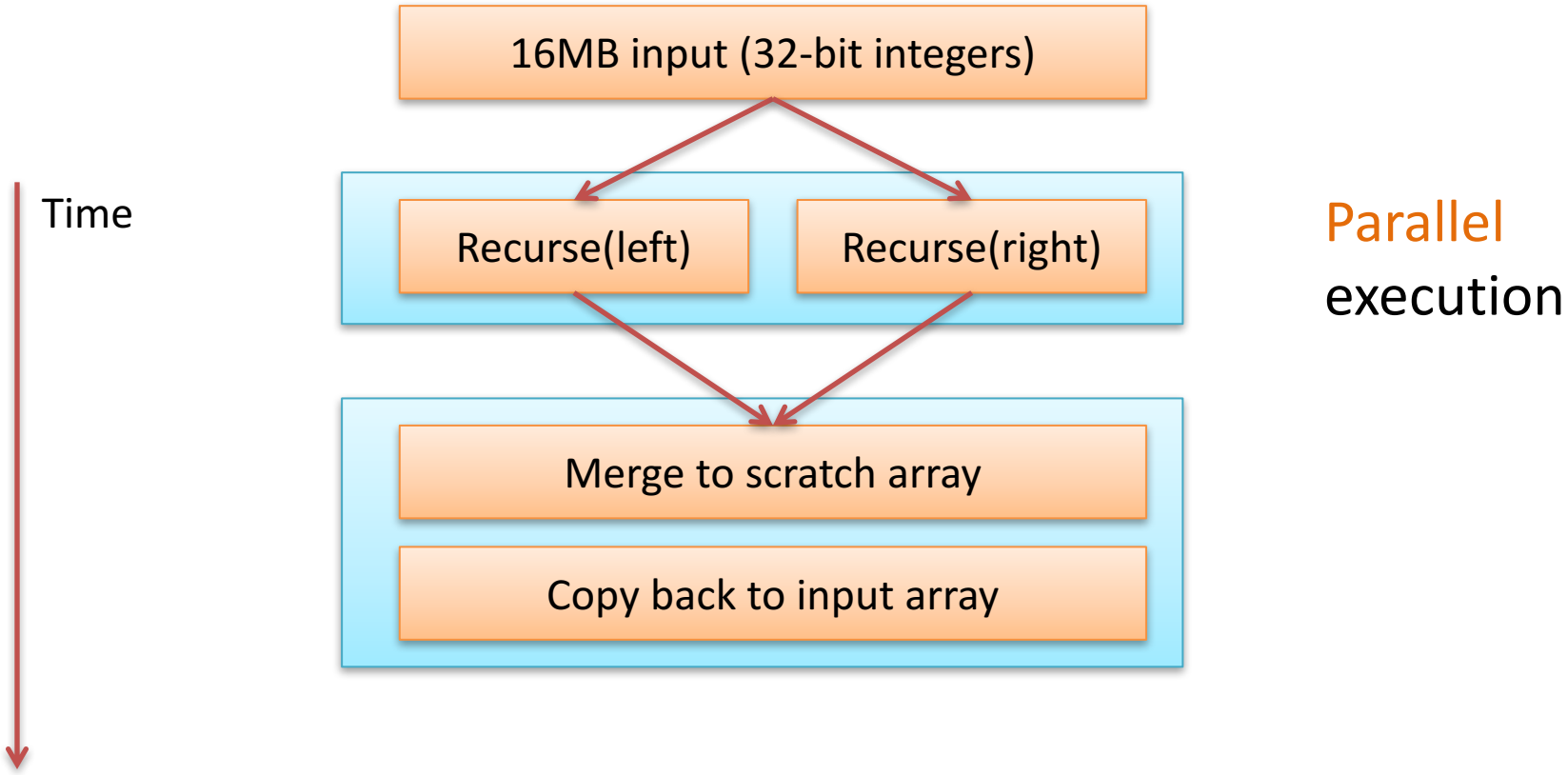


Sequential execution

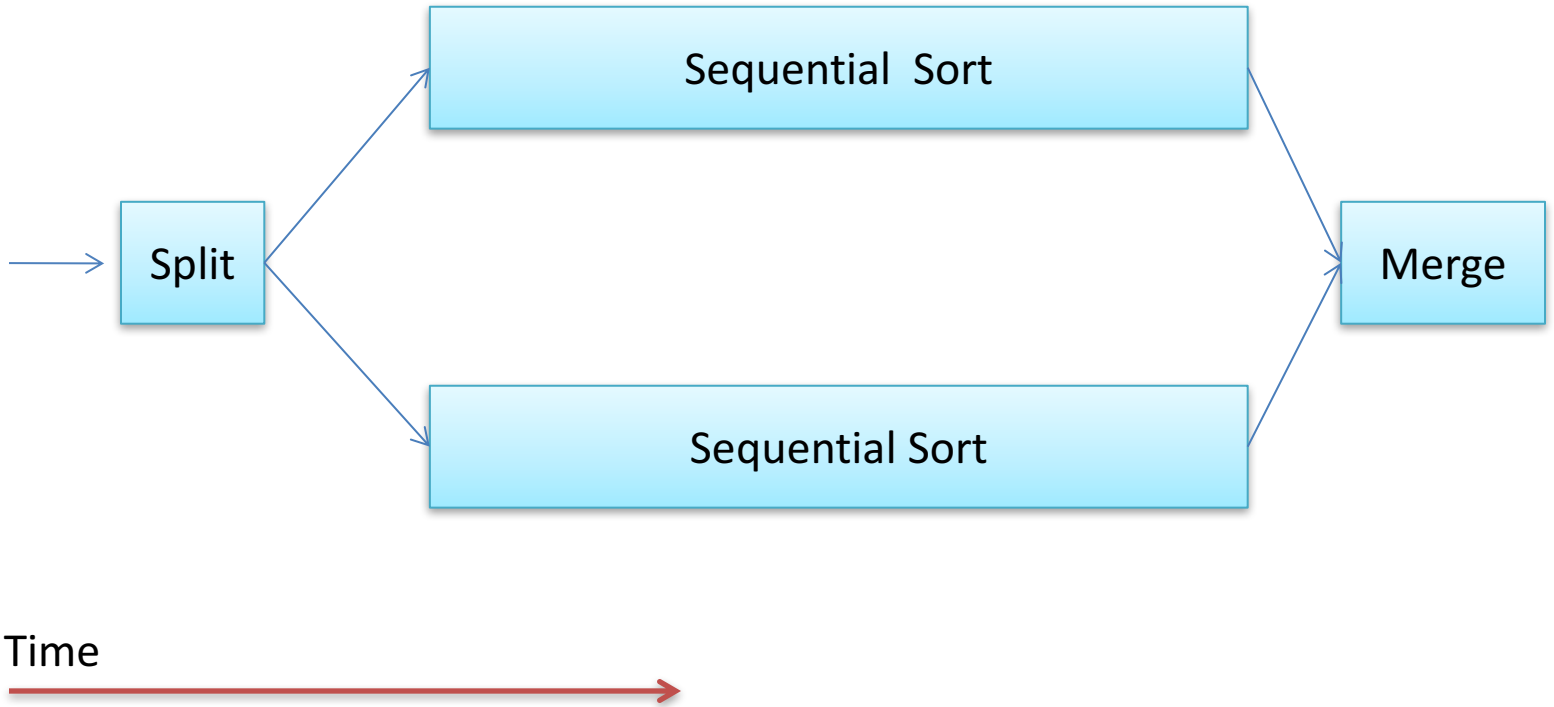
The DAG Execution Model of a parallel computation

- Given an input, dynamically create a DAG
  - Directed acyclic graphs representations of partial orderings have many applications in systems of tasks with ordering constraints
- 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

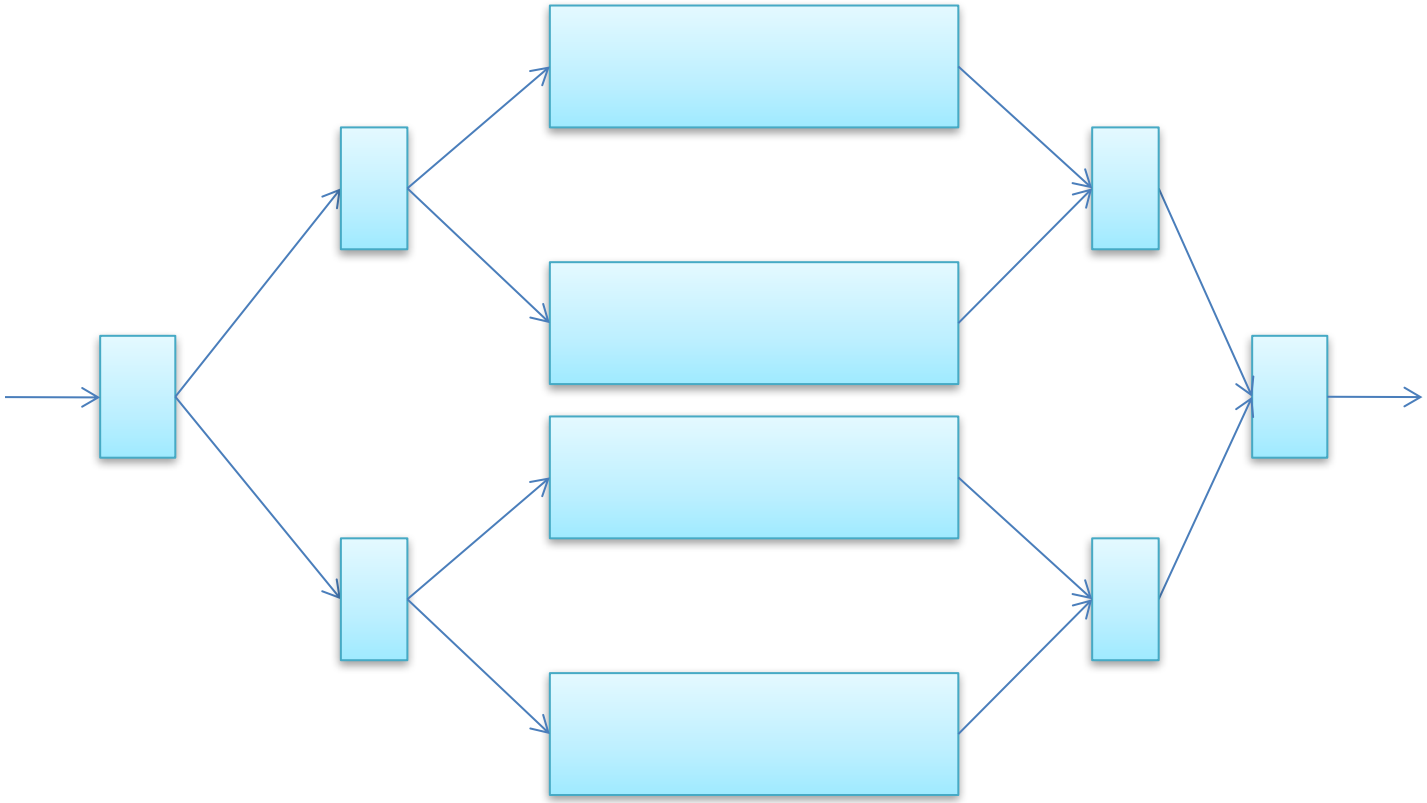
Parallel Merge Sort (as Parallel Directed Acyclic Graph)



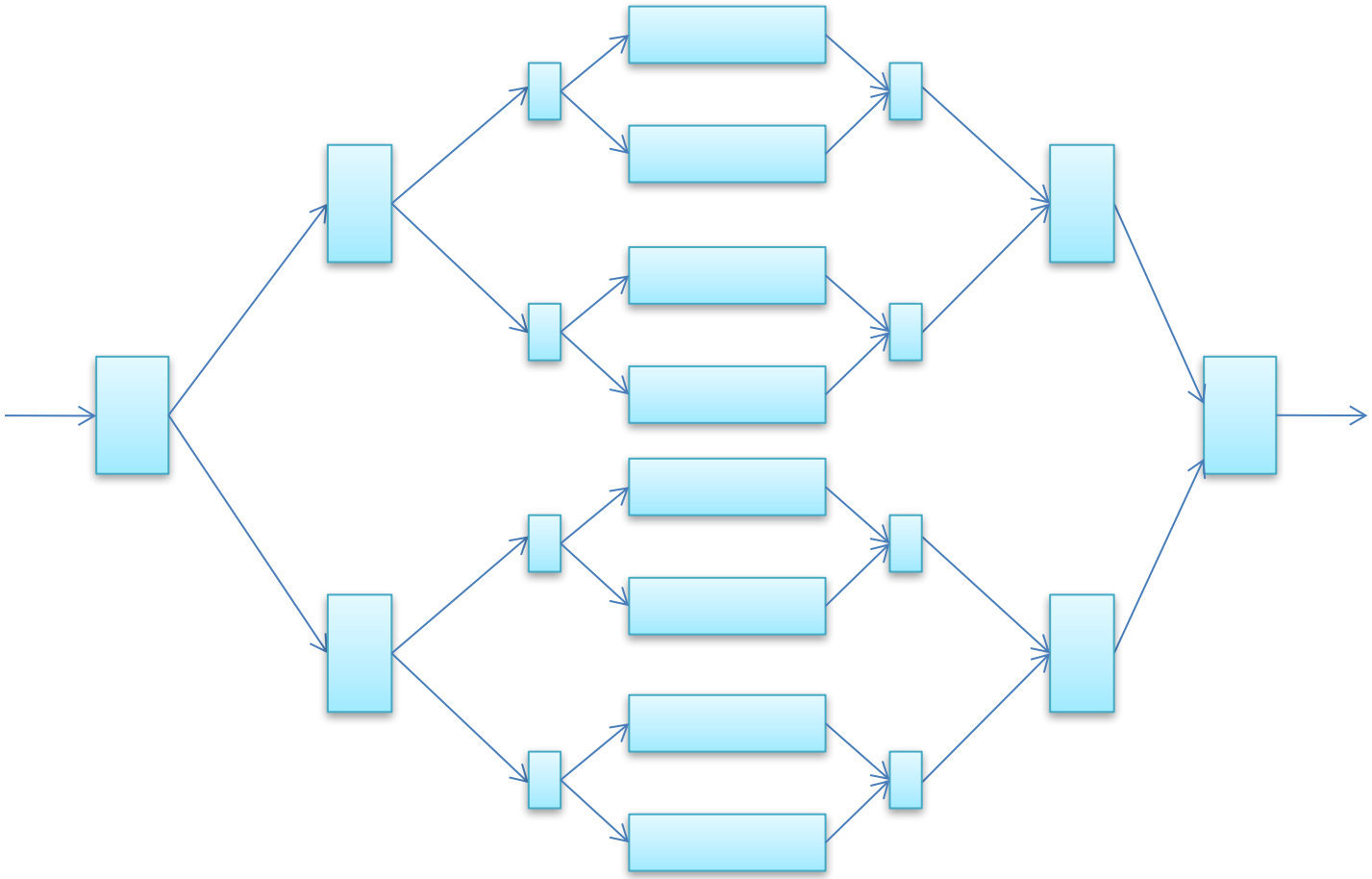
Parallel DAG for Merge Sort (2-core)



Parallel DAG for Merge Sort (4-core)



Parallel DAG for Merge Sort (8-core)



Performance measures

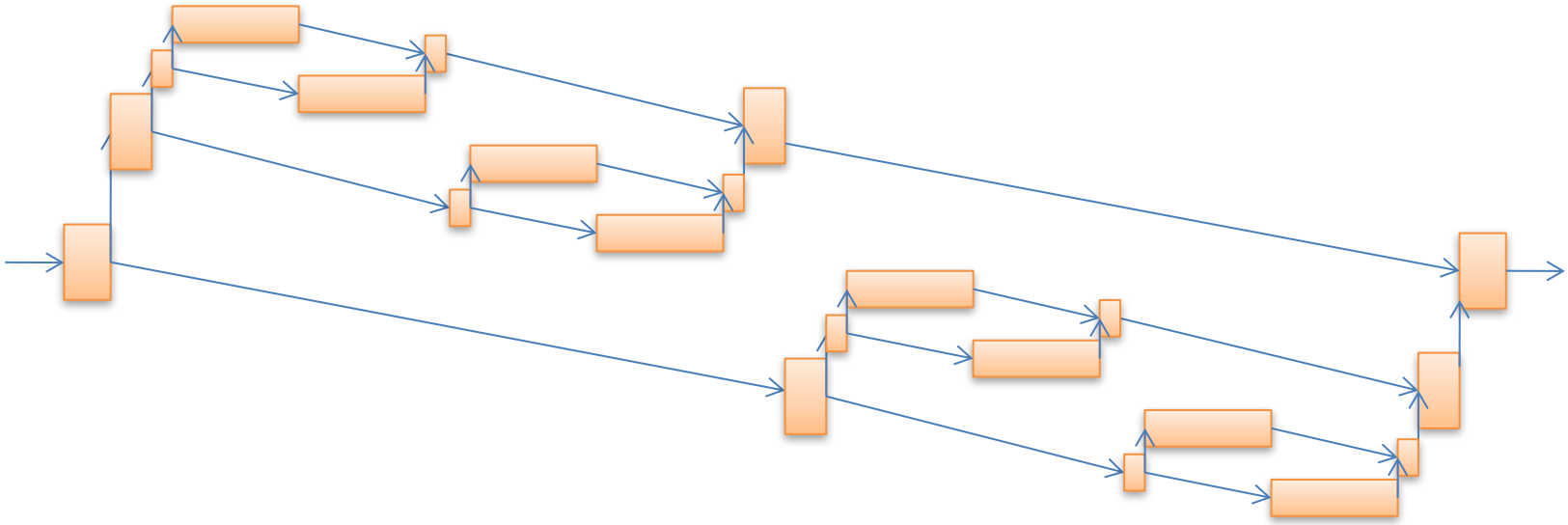
- Given a graph G, a scheduler S, and P processors

- $T_P(S)$: 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

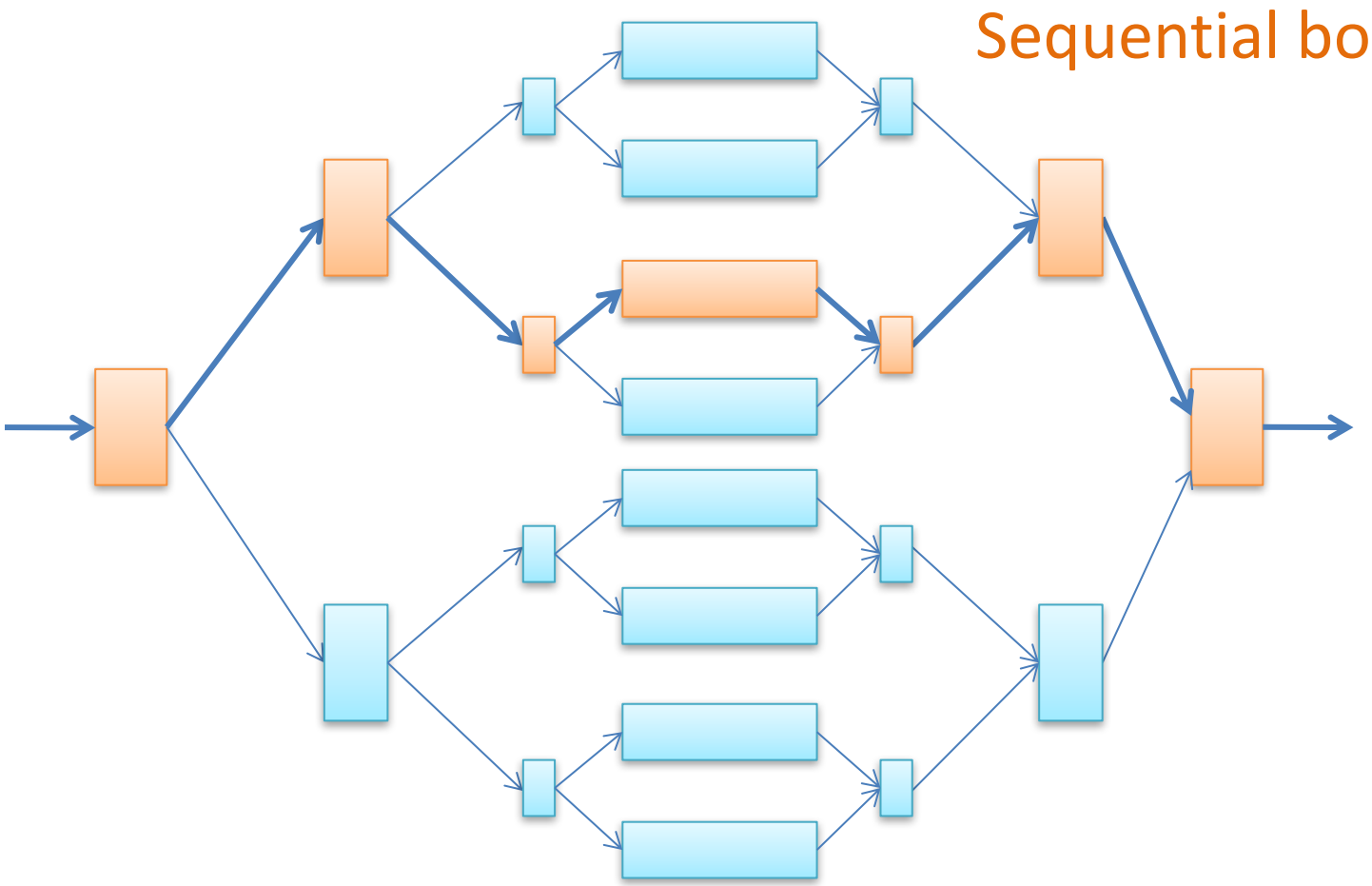
Work and Depth

- 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_1 (work): Time to run sequentially



T_∞ (Depth): Critical path length



Work Law

- You cannot avoid work by parallelizing

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

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

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

- Unfortunately, you are limited by the sequential bottleneck

$$T_P \geq T_\infty$$

Amount of parallelism

- A metric which indicates how many operations can be simultaneously executed in every step of the critical path

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

Maximum speedup possible

- Speedup is bounded above by available parallelism

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

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

- All programs contain:
  - Parallel sections (we hope!)
  - Sequential sections (we despair!)

- Sequential sections limit the parallel effectiveness

Main message

- Analyse the Work and Depth of your algorithm
- Parallelism is Work/Depth
- Try to decrease Depth
  - The critical path: a sequential bottleneck
- If you increase Depth, better increase Work by a lot more!

Limits of parallel computation

- Theoretical limits
  - Amdahl's Law
  - Gustafson's Law
- Practical limits
  - Load balancing (waiting), Scheduling (shared processors or memory), Cost of Communications, I/O…
- Other considerations
  - Time to re-write code

Amdahl's law

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

f = the parallel portion of execution

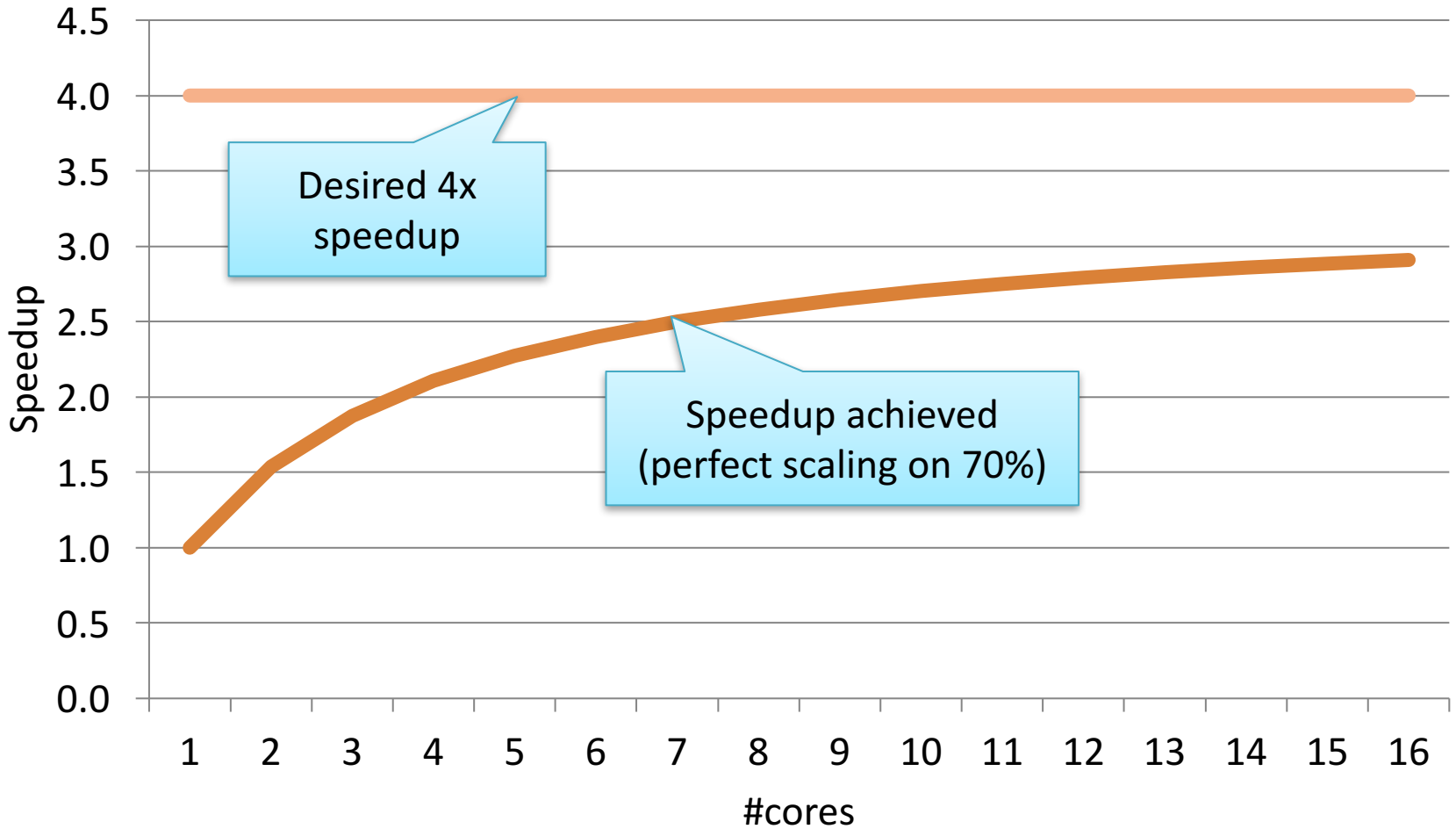
$(1 - f)$ = the sequential portion of execution

c = number of cores used

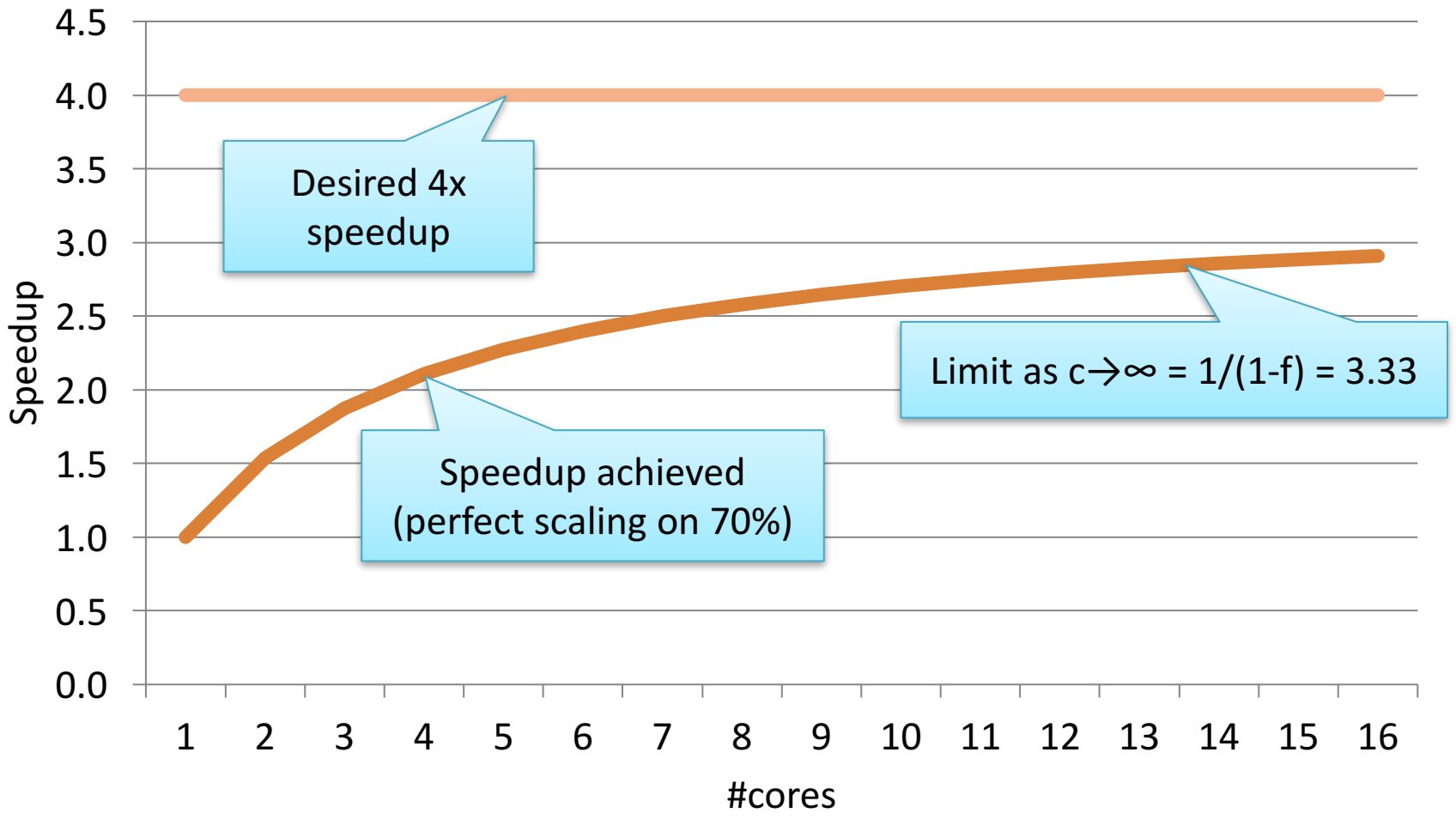
Amdahl's law, $f = 70\%$

- 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?

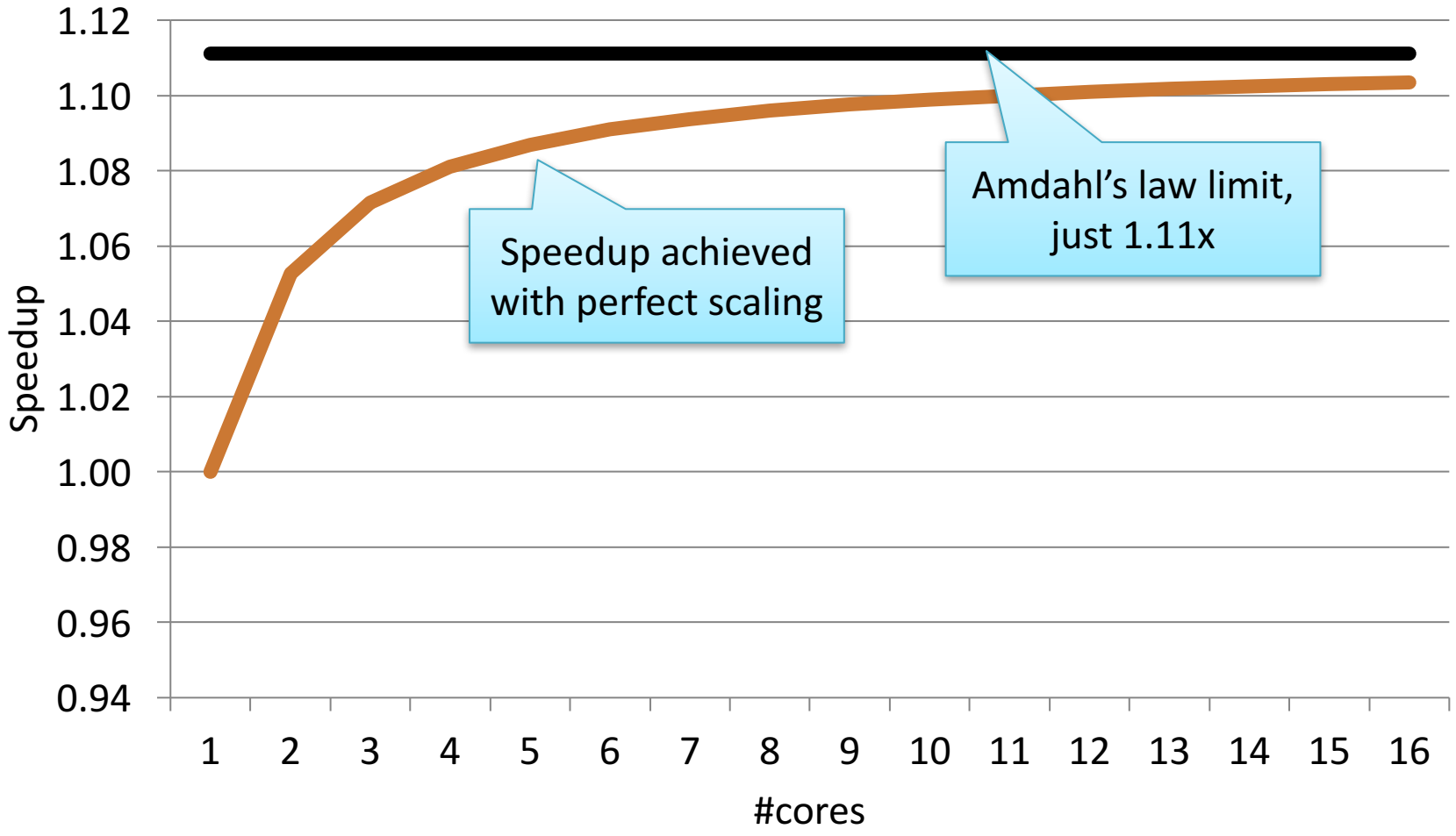
Amdahl's law, $f = 70\%$



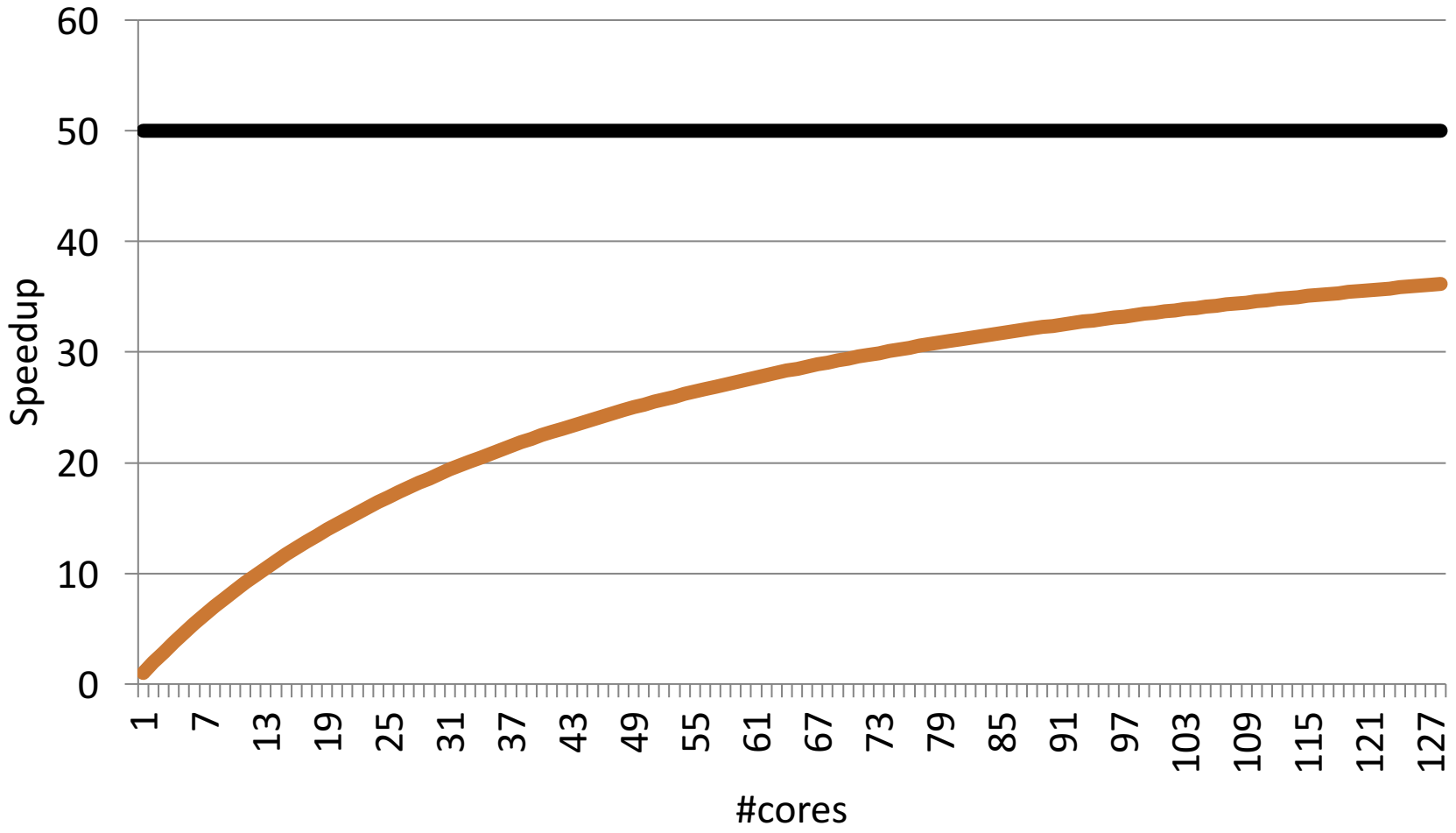
Amdahl's law, $f = 70\%$



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



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



Lesson

- Speedup is limited by sequential code
- Even a small percentage of sequential code can greatly limit potential speedup
- In reality, the situation is usually even worse than predicted by Amdahl's Law
  - Load balancing (waiting), Scheduling (shared processors or memory), Cost of Communications, I/O…

Gustafson's Law

Any sufficiently large problem can be parallelized effectively

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

f = the parallel portion of execution

$(1 - f)$ = the sequential portion of execution

c = number of cores used

Key assumption: f increases as problem size increases

Scaling: Strong vs Weak

- We want to know how quickly we can complete analysis on a particular data set by increasing P
  - Amdahl's Law
  - Known as "strong scaling"

- We want to know if we can analyse more data in the same amount of time by increasing P
  - Gustafson's Law
  - Known as "weak scaling"

And now…

- Design of a greedy scheduler
- Task abstraction

Greedy Scheduler

- A scheduler which attempts to do as much work as possible at every step
- 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

- Given P processors, a greedy scheduler executes any computation with work T_1 and critical path length T_∞ in time:

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

- Remember:
  - Work law: $\frac{T_1}{P} \leq T_P$
  - Depth law: $T_\infty \leq T_P$

Performance of the Greedy Scheduler

- Given P processors, a greedy scheduler executes any computation with work T_1 and critical path length T_∞ in time:

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

- The idea is to operate in the range where T_1 / P dominates
  - Adding more processors reduces T_1 / P while T_∞ remains unchanged
  - Once T_∞ becomes significant in comparison to T_1 / P, the ability to increase parallelism is reduced

Case study: *Socrates

- The *Socrates massively parallel chess program had two versions, one including an optimisation that significantly improved its performance on 32 processors
  - The performance reduced the total work, but increased the critical path length

- T_{32} = 65 sec → T_1 = 2048 sec, T_∞ = 1 sec
- T'_{32} = 40 sec → T_1 = 1024 sec, T_∞ = 8 sec

Case study: *Socrates

- On 512 processors the optimisation becomes a problem!

- $T_{512} = T_1 /512 + T_\infty$ = 5 sec
- $T'_{512} = T'_1 /512 + T_\infty$ = 10 sec

- The optimization that sped up the program on 32 processors would have made the program twice as slow on 512 processors

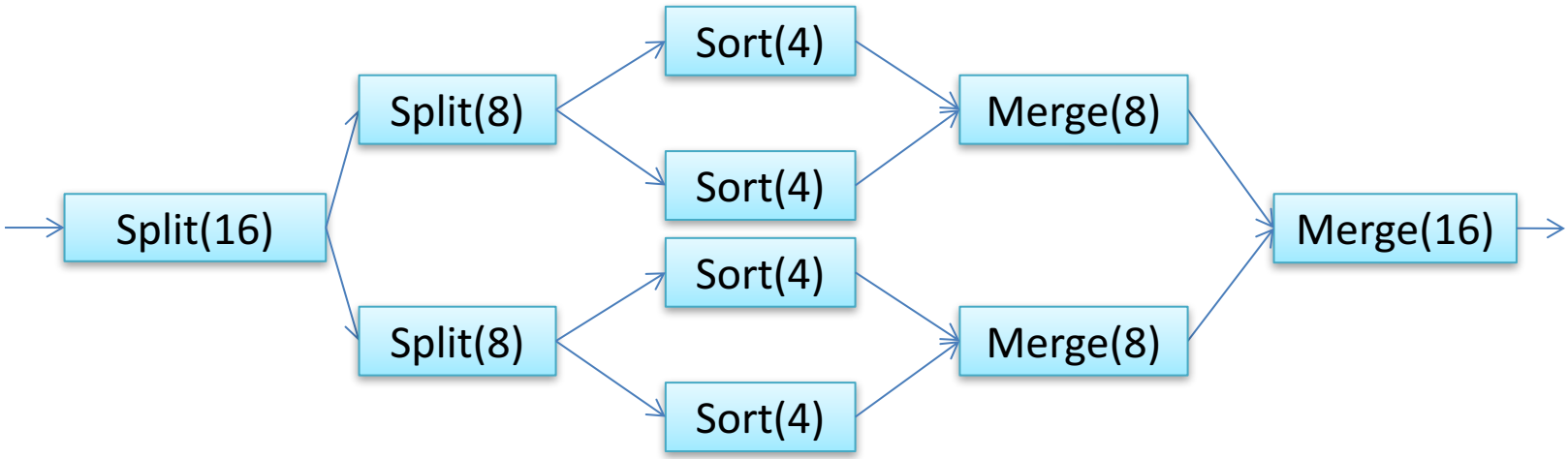
Case study: *Socrates

- The optimized version's depth of 8, which was not the dominant term in the running time on 32 processors, became the dominant term on 512 processors, nullifying the advantage from using more processors

- Work and Span can provide a better means of extrapolating performance than can measured running times

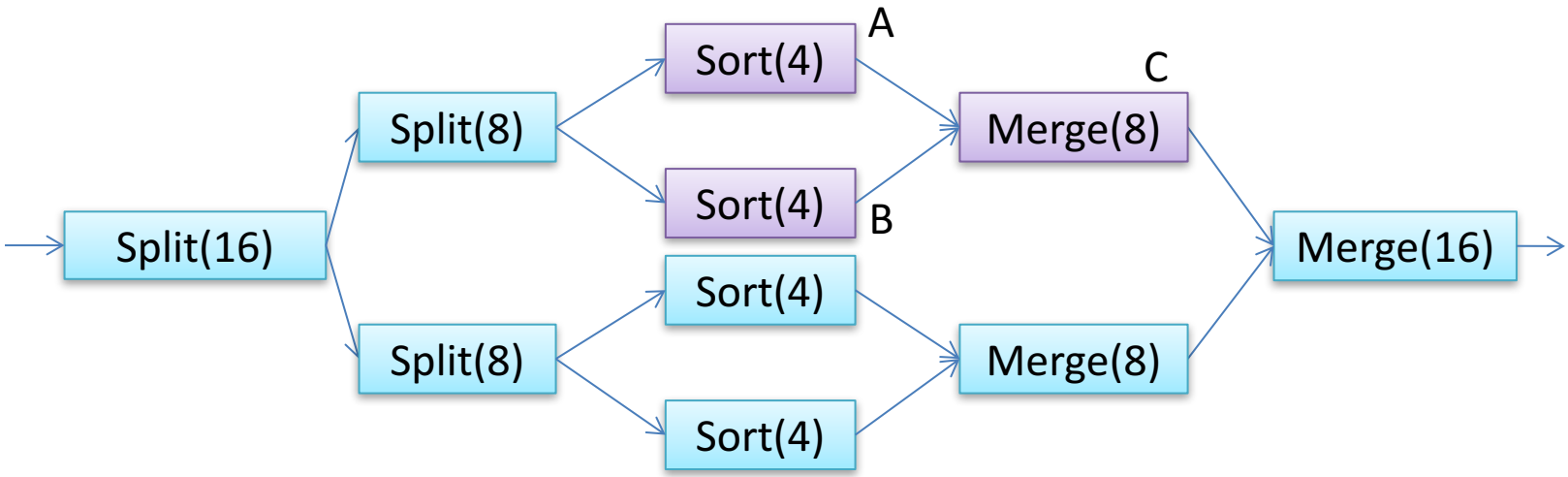
(Simple) Tasks

- A node in the DAG
- Executing a task generates dependent subtasks



(Simple) Tasks

- A node in the DAG
- Executing a task generates dependent subtasks
  - Task C is generated by A or B, whoever finishes last



Design constraints of the scheduler

- The DAG is generated dynamically
  - Based on inputs and program control flow
  - The graph is not known ahead of time
- The amount of work done by a task is dynamic
  - The weight of each node is not know ahead of time
- Number of processors P can change at runtime
  - Hardware processors are shared with other processes and kernel

Design requirements of the scheduler

- Should be greedy
  - A processor cannot be idle when tasks are pending
- Should limit communication between processors
- Should schedule related tasks in the same processor
  - Tasks that are likely to access the same cache lines

Attempt 0: Centralized Scheduler

- Approach
  - Manager distributes tasks to workers
- Manager
  - Assigns tasks to workers, ensures no worker is idle
- Workers
  - On task completion, submit generated tasks to the manager

Attempt 1: Centralized Work Queue

- Approach
  - All processors share a common work queue
- Every processor dequeues a task from the work queue
- On task completion, enqueue the generated tasks to the work queue

Attempt 2: Work Sharing

- Approach
  - Loaded workers share
- Every processor pushes and pops tasks into a local work queue
- When the work queue gets large, send tasks to other processors

Disadvantages of Work Sharing

- If all processors are busy, each will spend time trying to offload, performing communication when busy
- Difficult to know the load on processors
  - A processor with two large tasks might take longer than a processor with five small tasks
- Tasks might get shared multiple times before being executed
- Some processors can be idle while others are loaded
  - Not greedy

Attempt 3: Work Stealing

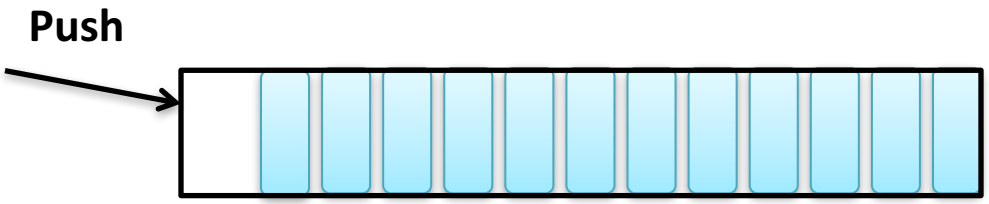
- Approach
  - Idle workers steal
- Each processor maintains a local work queue
- Pushes generated tasks into the local queue
- When local queue is empty, steal a task from another processor

Nice properties of Work Stealing

- Communication done only when idle
- Each task is stolen at most once
- This scheduler is greedy, assuming stealers always succeed
- Limited communication
  - $O(P.T_\infty)$ steals on average for some stealing strategies

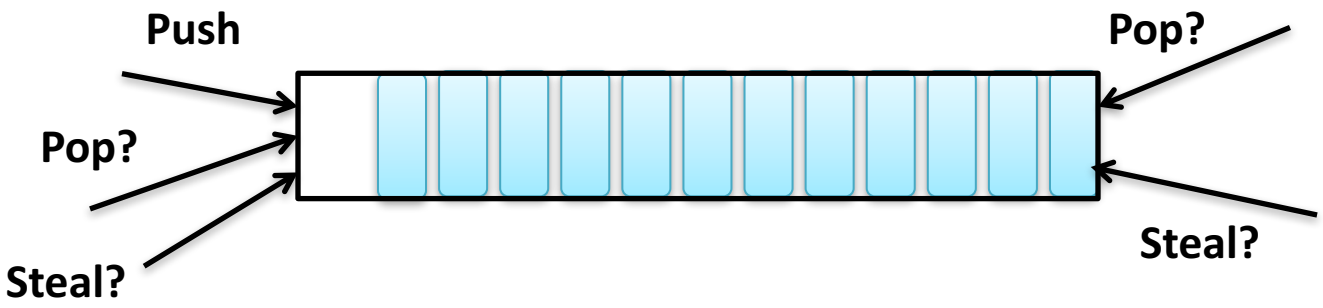
Work Stealing Queue data structure

- A specialized deque (Double-Ended Queue) with three operations:
  - Push : Local processor adds newly created tasks
  - Pop : Local processor removes task to execute
  - Steal : Remote processors remove tasks



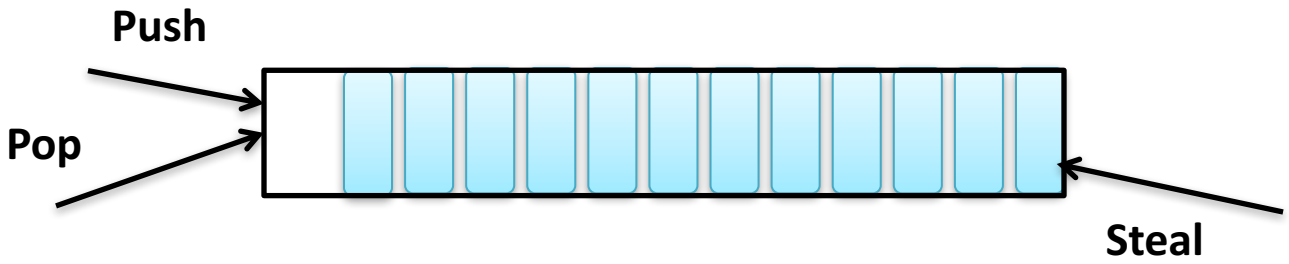
Work Stealing Queue data structure

- A specialized deque (Double-Ended Queue) with three operations:
  - Push : Local processor adds newly created tasks
  - Pop : Local processor removes task to execute
  - Steal : Remote processors remove tasks



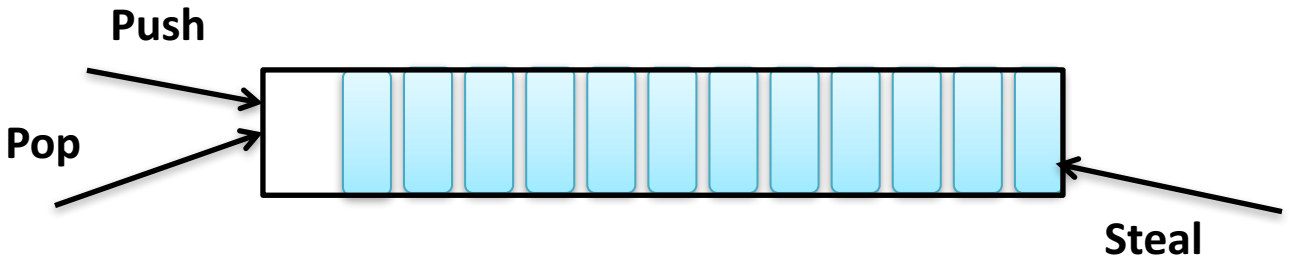
Work Stealing Queue data structure

- A specialized deque (Double-Ended Queue) with three operations:
  - Push : Local processor adds newly created tasks
  - Pop : Local processor removes task to execute
  - Steal : Remote processors remove tasks



Advantages

- Stealers don't interact with local processor when the queue has more than one task
- Popping recently pushed tasks improves locality
- Stealers take the oldest tasks, which are likely to be the largest (in practice)



Assignment

- Study the paper “Scheduling Multithreaded Computations by Work Stealing” by Blumofe and Leiserson

- Discuss how the work stealing approach can be extended to real-time systems