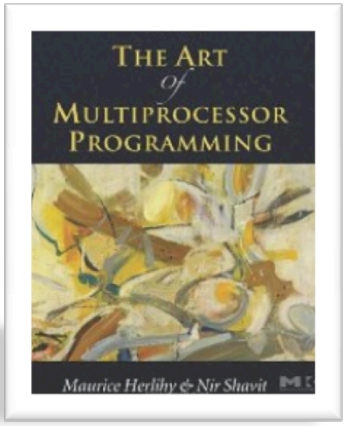
WORK STEALING SCHEDULER

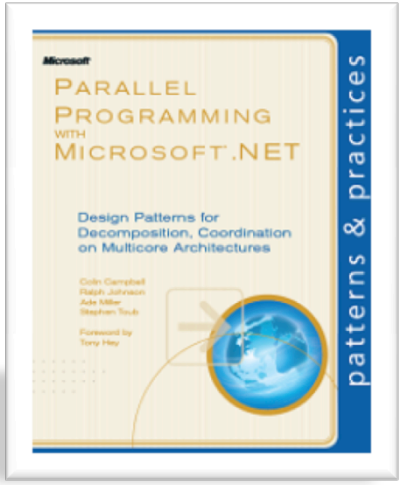
Announcements

- Text books
- Assignment 1
- Assignment 0 results
- Upcoming Guest lectures

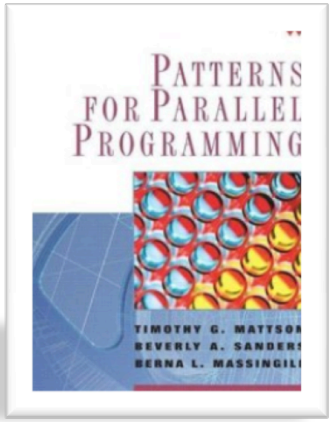
Recommended Textbooks



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.

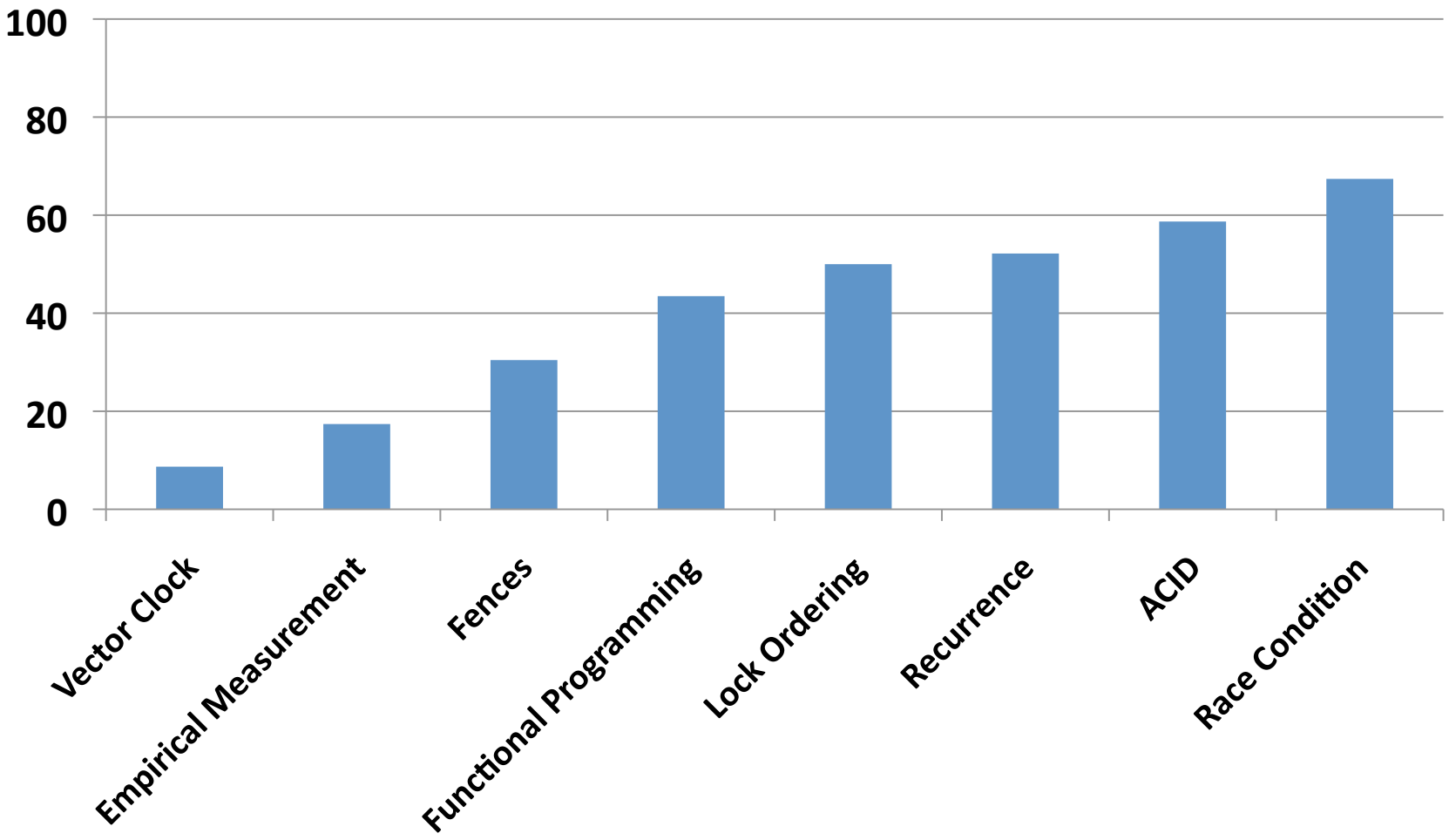
Available on Website

- Assignment 1 (due two weeks from now)
- Paper for Reading Assignment 1 (due next week)

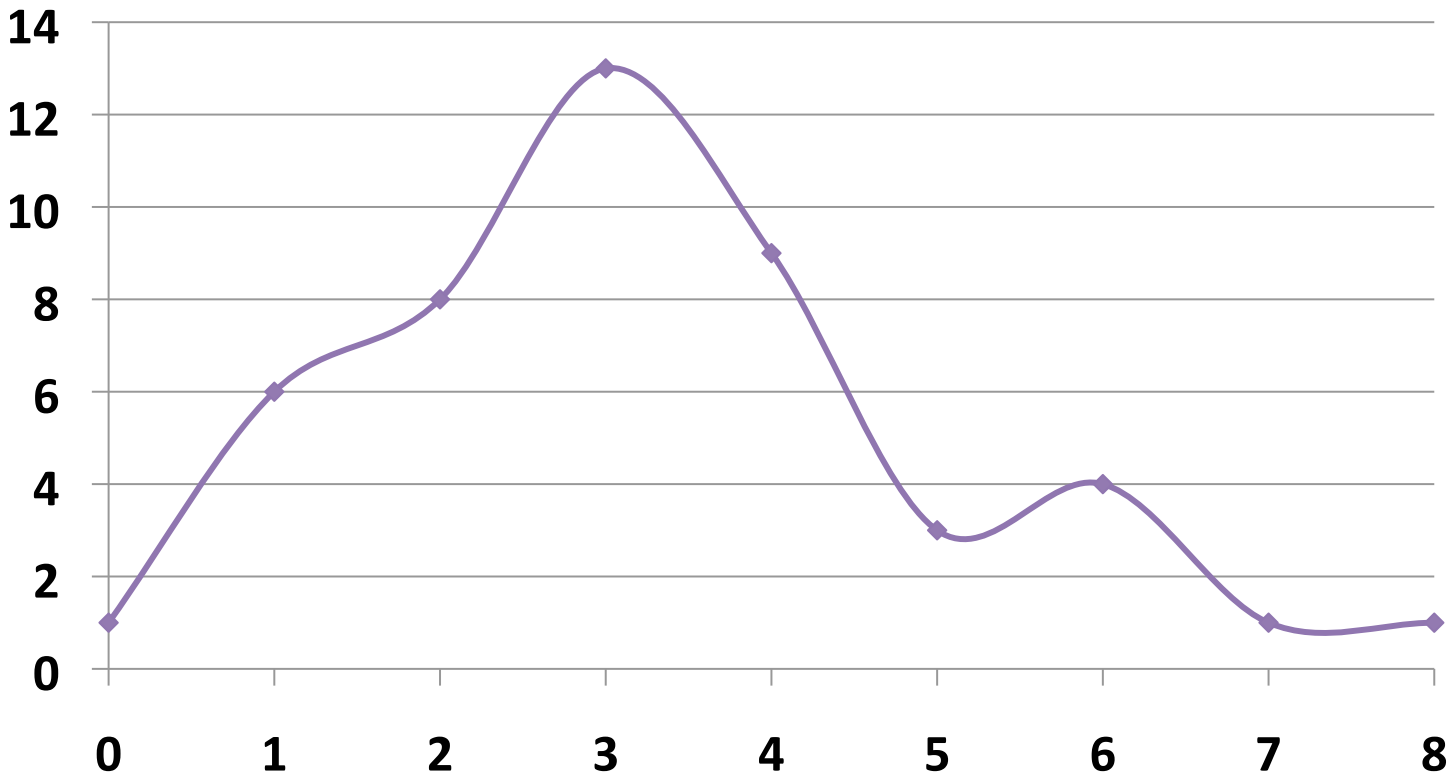
Assignment 0

- Thanks for submitting
- Provided important feedback to us

Percentage Students Answering Yes



Number of Yes Answers Per Student



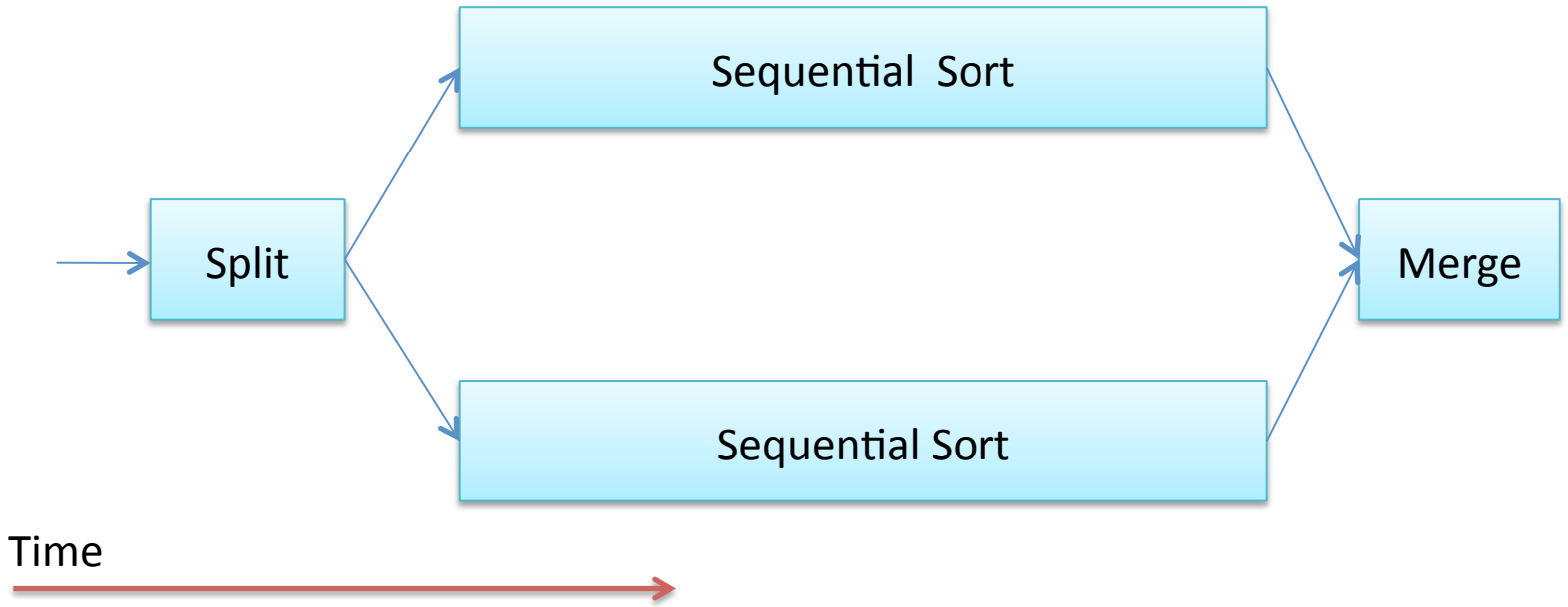
Upcoming Guest Lectures

- Apr 12: Todd Mytkowicz
  - How to (and not to) measure performance
- Apr 19: Shaz Qadeer
  - Correctness Specifications and Data Race Detection

Last Lecture Recap

Last Lecture Recap

- Parallel computation can be represented as a DAG
  - Nodes represent sequential computation
  - Edges represent dependencies



Last Lecture Recap

- Parallel computation can be represented as a DAG
- T_1 = Work = time on a single processor
- T_∞ = Depth = time assuming infinite processors
- T_P = time on P processor using optimal scheduler

Last Lecture Recap

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

Last Lecture Recap

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

- Greedy scheduler is optimal within a factor of 2

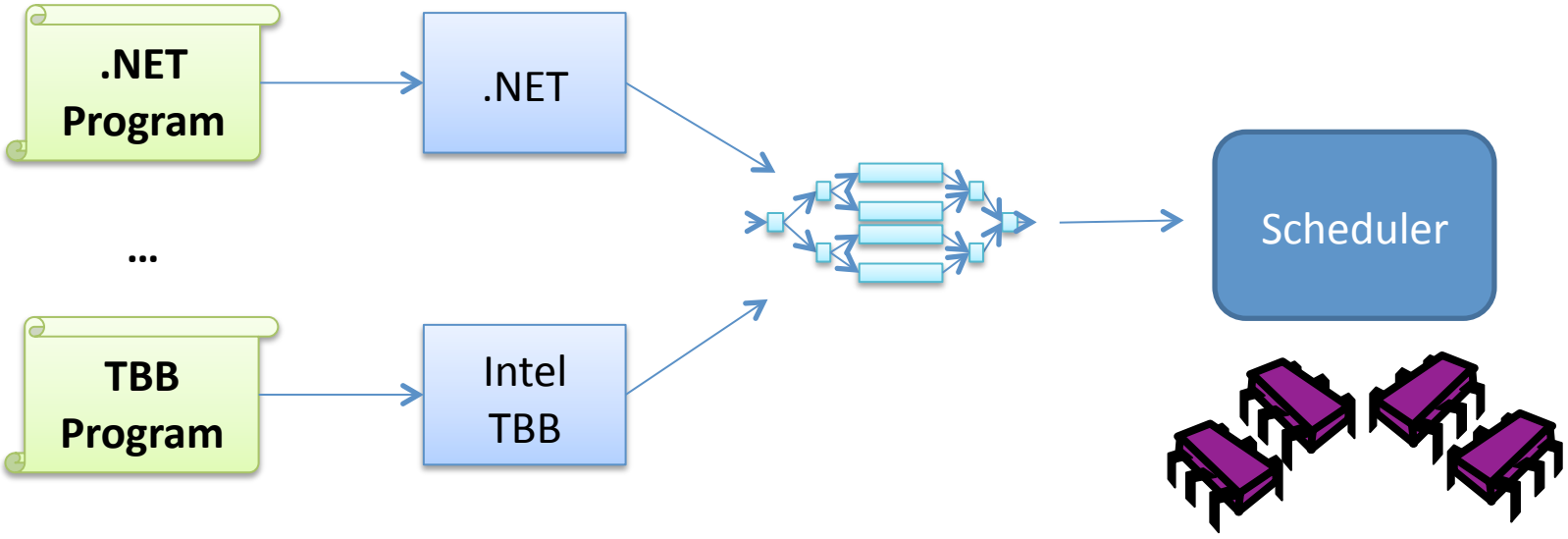
$$T_P \leq \quad T_P(Greedy) \leq \frac{T_1}{P} + T_\infty \quad \leq 2 * T_P$$

This Lecture

- Design of a greedy scheduler
- Task abstraction
- Translating high-level abstractions to tasks

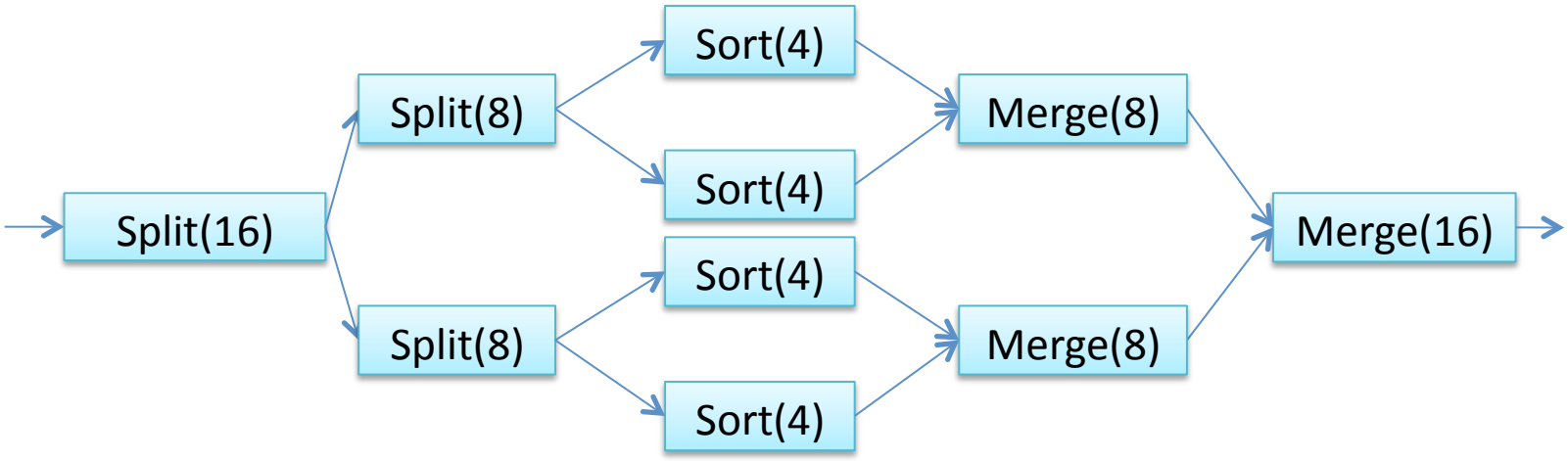
This Lecture

- Design of a greedy scheduler
- Task abstraction
- Translating high-level abstractions to tasks



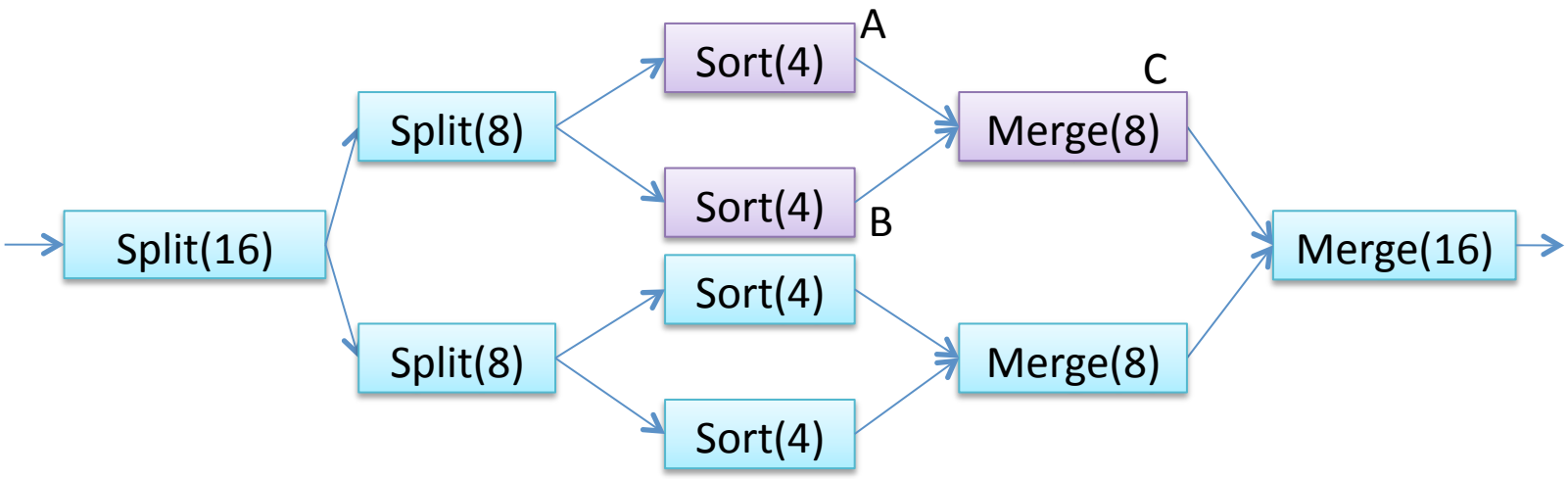
(Simple) Tasks

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



(Simple) Tasks

- A node in the DAG
- Executing a task generates dependent subtasks
  - Note: 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, kernel

Design Requirements of the Scheduler

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 cachelines

Attempt 0: Centralized Scheduler

- "Manager distributes tasks to others"
- 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

- "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

- "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
  - "Perform 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

- "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

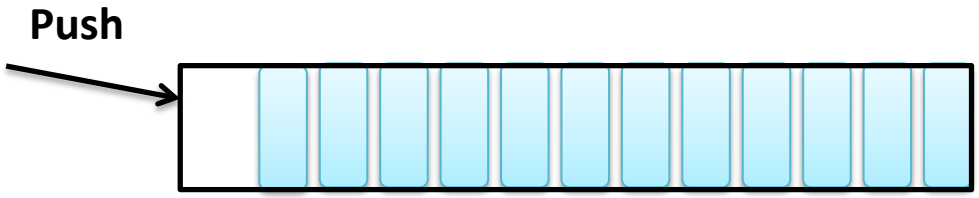
- Communication done only when idle
  - No communication when all processors are in full throttle
- Each task is stolen at most once
- This scheduler is greedy, assuming stealers always succeed
- Limited communication
  - $O(P.T_\infty)$ steals on average for some stealing strategies

Nice Properties of Work Stealing

- Communication done only when idle
  - No communication when all processors are in full throttle
- 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
- Assignment 1 explores different stealing strategies

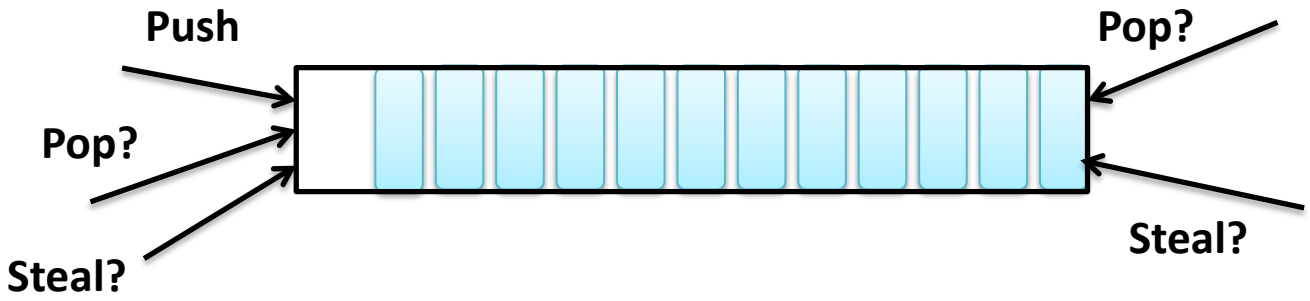
Work Stealing Queue Datastructure

- 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



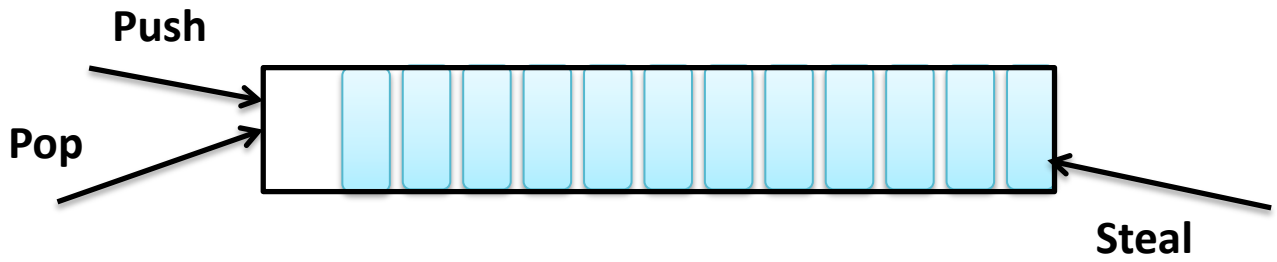
Work Stealing Queue Datastructure

- 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



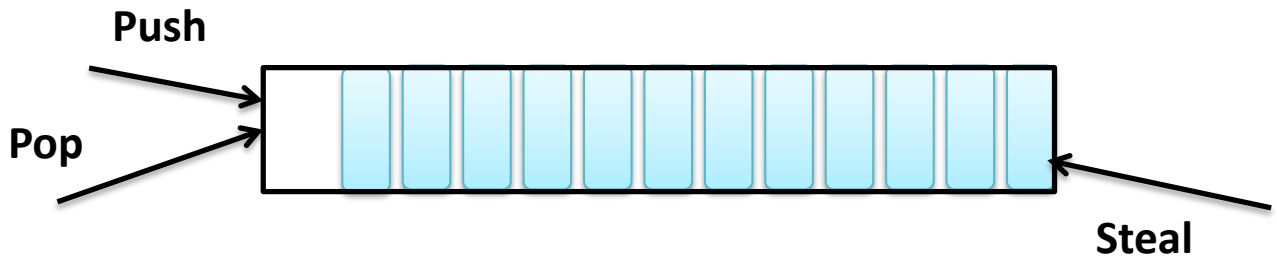
Work Stealing Queue Datastructure

- 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)



For Assignment 1

- We provide an implementation of Work Stealing Queue
- This implementation is thread-safe
  - Clients can concurrently call the push, pop, steal operations
  - You don't need to use additional locks or other synchronization
- Implement the scheduler logic and stealing strategy
  - Hint: this implementation is single threaded