Multi-Queues Can Be State-of-the-Art Priority Schedulers

Anastasiia Postnikova[1], **Nikita Koval**[2],
Giorgi Nadiradze[3], Dan Alistarh[3]

[1] ITMO University
[2] **JetBrains**
[3] IST Austria

Priority Schedulers – when do we need them?

Priority Schedulers – when do we need them?

Parallel iterative algorithms!

Priority Schedulers – when do we need them?

Parallel iterative algorithms!

- Parallel Graph Algorithms (Dijkstra, A*, BFS, Boruvka, …)
- Delaunay Triangulation
- PageRank Algorithm
- …

Parallel Dijkstra: A Brief Overview

```
val Q := PriorityQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)
```

Parallel Dijkstra: A Brief Overview

```
val Q := PriorityQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)

while Q.isNotEmpty() {
  u := Q.delete()
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = u.distance + v.weight
    Q.insert(v)
  }
}
```

Parallel Dijkstra: A Brief Overview

```
val Q := ConcurrentPriorityQueue<Node>()
start.distance = 0 // INF for others
Q.add(start)

while Q.isNotEmpty() {
  u := Q.delete()
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = u.distance + v.weight
    Q.insert(v)
  }
}
```

T threads

Parallel Dijkstra: A Brief Overview

```
val Q := ConcurrentPriorityQueue<Node>()
start.distance = 0
Q.add(start)
```

The Priority Scheduler

T threads

```
while Q.isNotEmpty() {
  u := Q.delete()
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = u.distance + v.weight
    Q.insert(v)
  }
}
```

Parallel Dijkstra: A Brief Overview

```
val Q := ConcurrentPriorityQueue<Node>()
start.distance = 0
Q.add(start)
activeNodes := 1
```

T threads

```
while activeNodes > 0 {
  u := Q.delete()
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = u.distance + v.weight
    activeNodes.inc(); Q.insert(v)
  }
  activeNodes.dec()
}
```

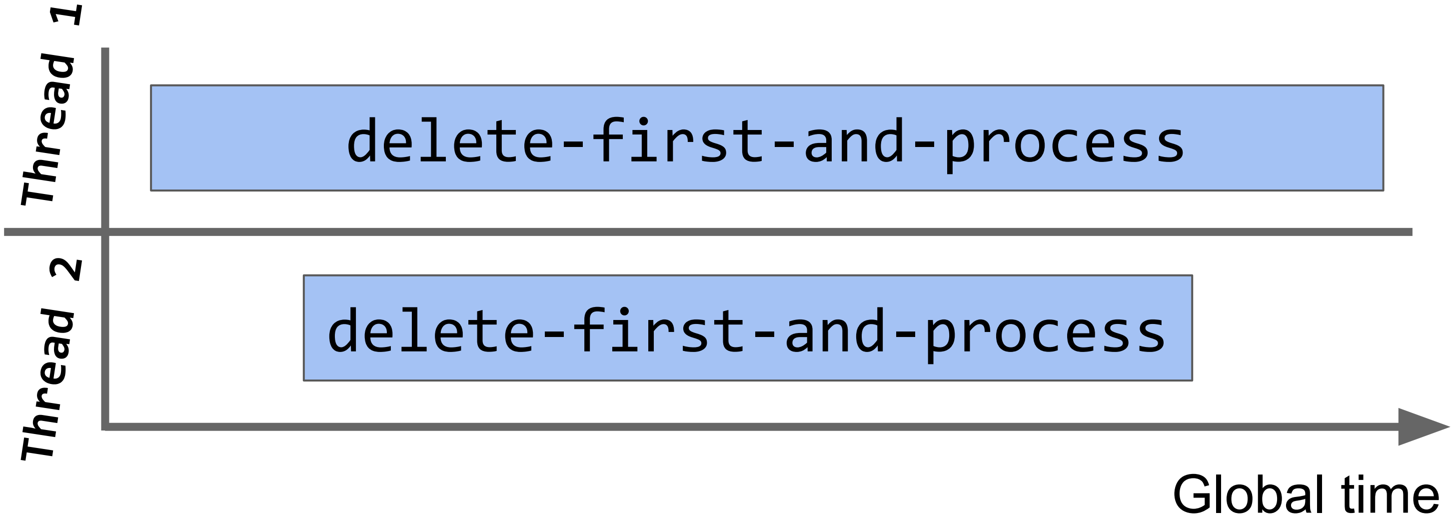
Parallel Dijkstra: A Brief Overview

```
val Q := ConcurrentPriorityQueue<Node>()
start.distance = 0
Q.add(start)
activeNodes := 1

while activeNodes > 0 {
  u := Q.delete()
  for (v : u.edges) {
    if v.distance != INF: continue
    v.distance = u.distance + v.weight
    activeNodes.inc(); Q.insert(v)
  }
  activeNodes.dec()
}
```

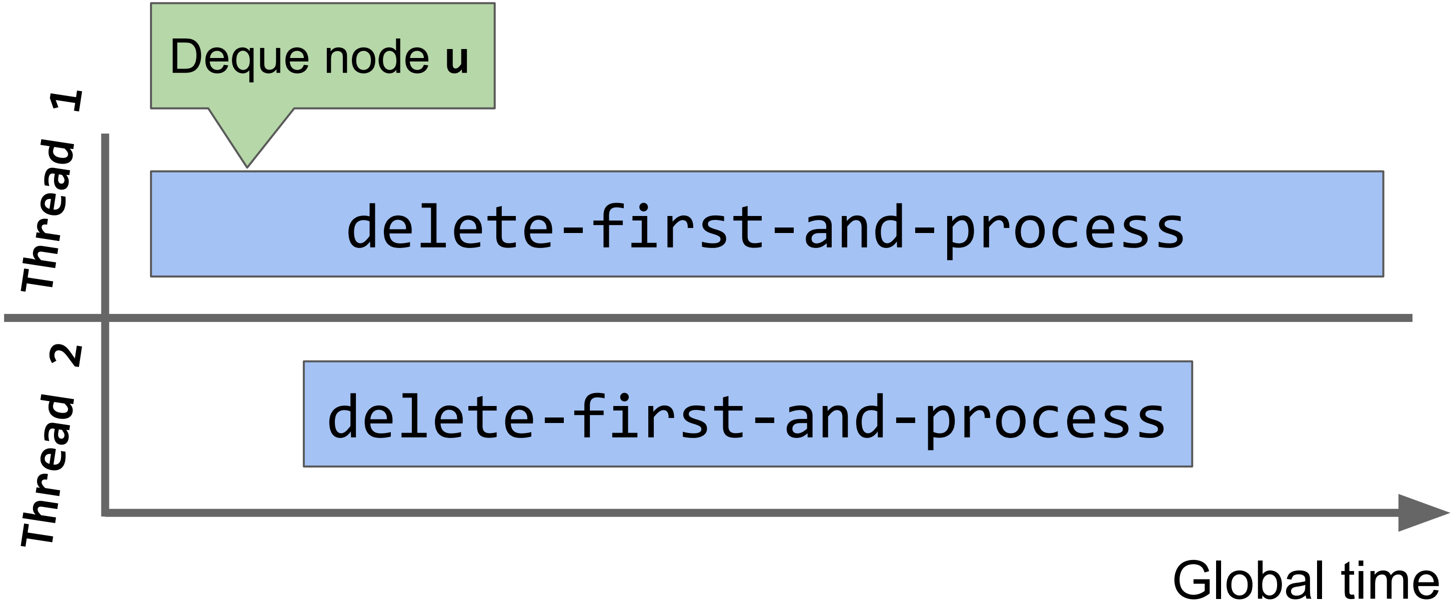
T threads

Is this algorithm correct?

Parallel Dijkstra: A Brief Overview

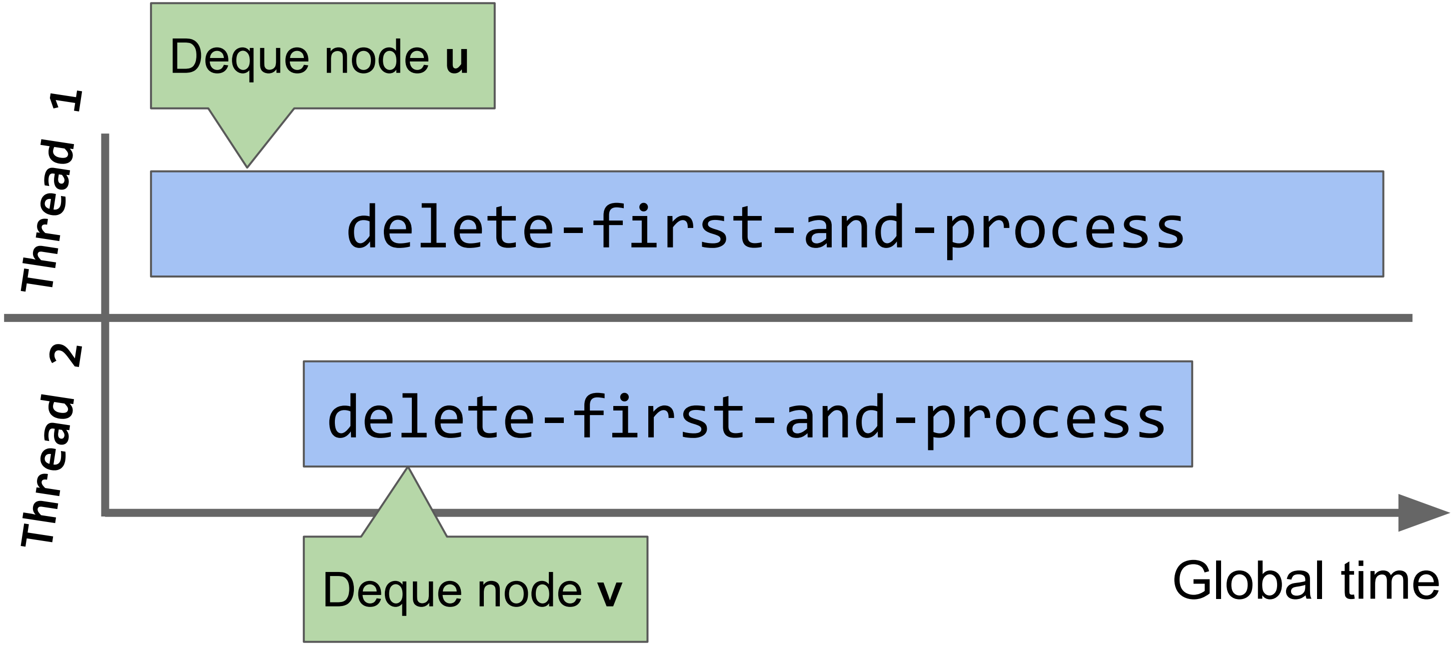


Parallel Dijkstra: A Brief Overview

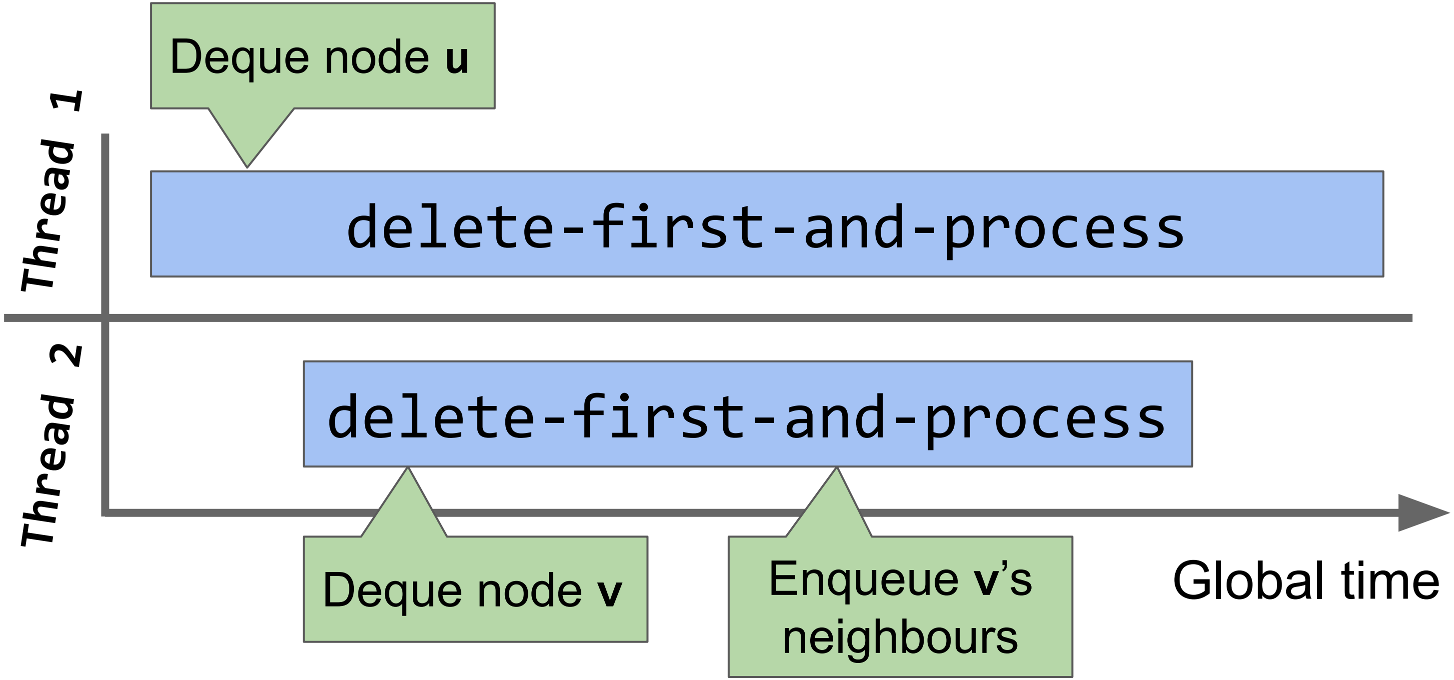


Deque node **u**

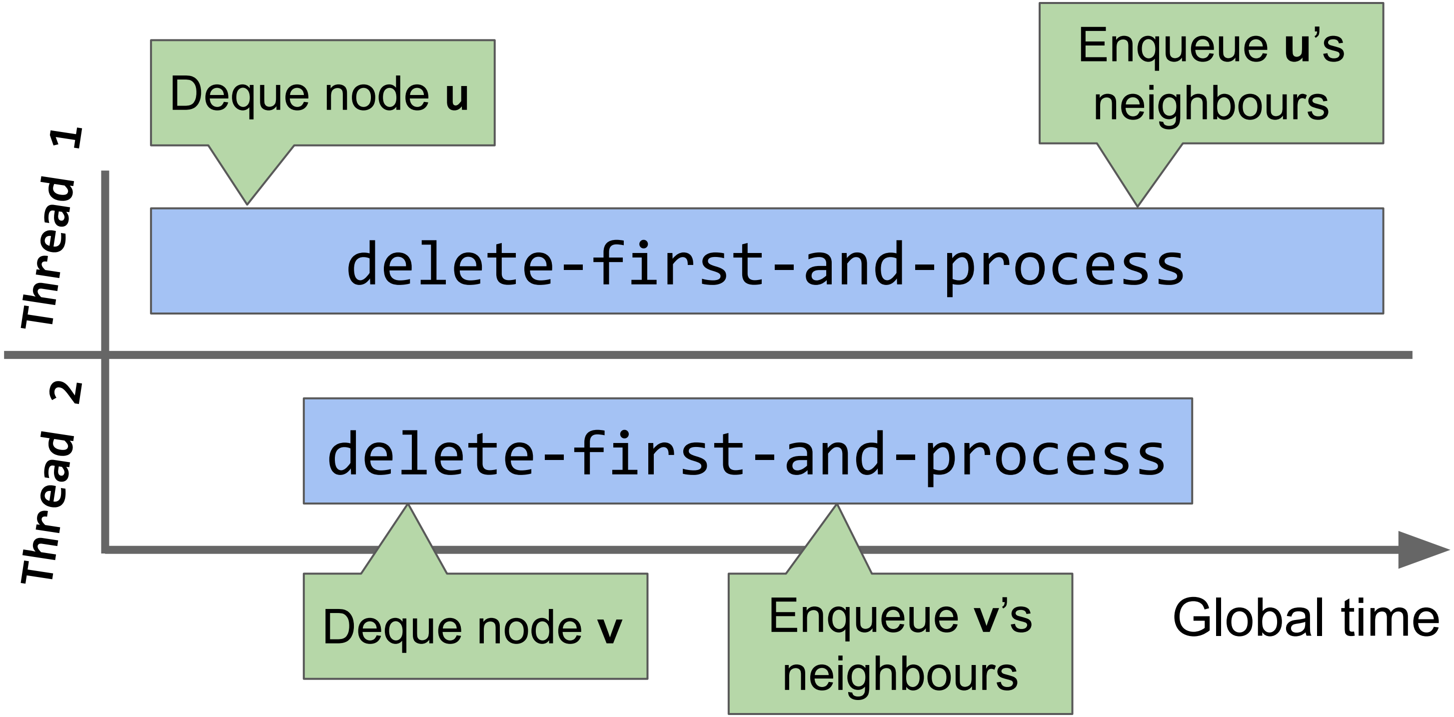
Parallel Dijkstra: A Brief Overview



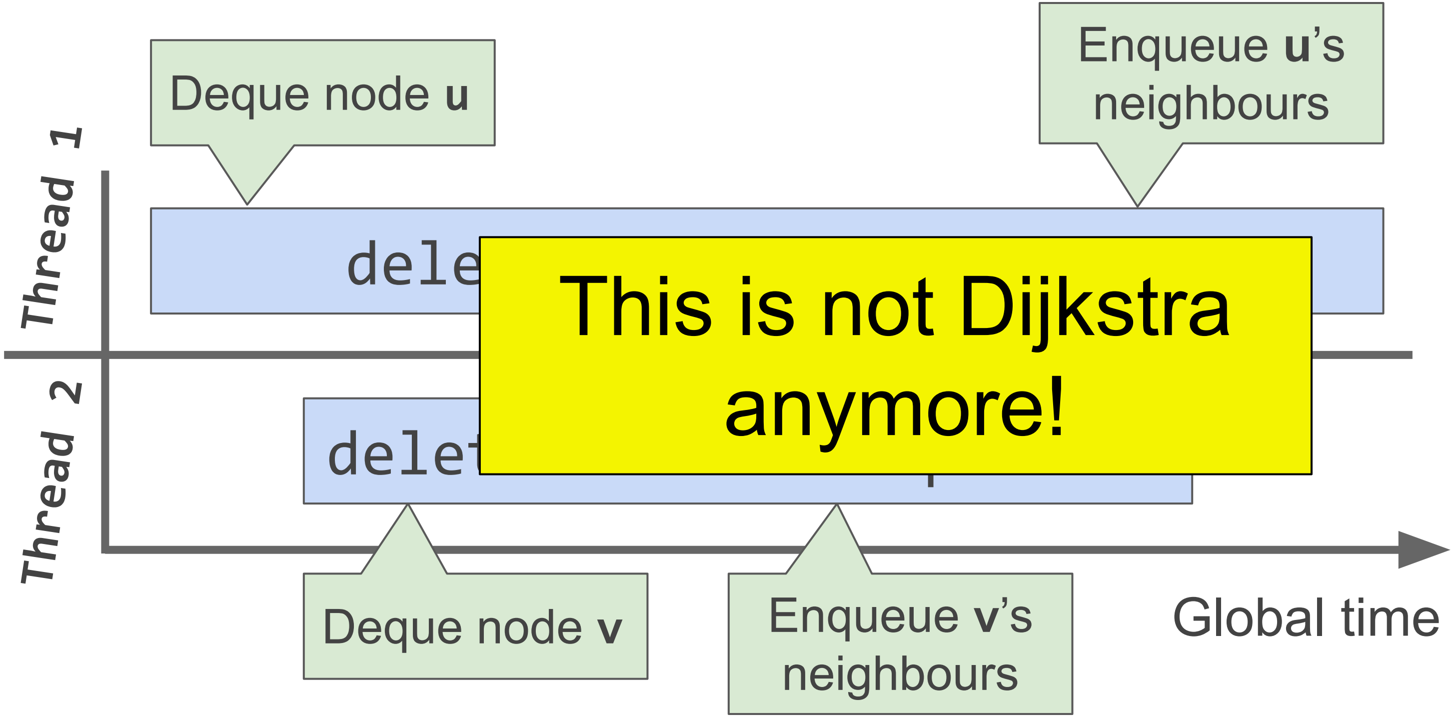
Parallel Dijkstra: A Brief Overview



Parallel Dijkstra: A Brief Overview



Parallel Dijkstra: A Brief Overview



Deque node **u**

Enqueue **u**’s neighbours

Thread 1

`dele`

This is not Dijkstra anymore!

Thread 2

`delet`

Deque node **v**

Enqueue **v**’s neighbours

Global time

Parallel Dijkstra: A Brief Overview

```
val Q = ConcurrentPriorityQueue<Node>()
start.distance = 0
Q.add(start)
activeNodes := 1
```

T threads

```
while activeNodes > 0 {
  u := Q.delete()
  for (v : u.edges) {
    d := u.distance + v.weight
    relaxed := v.updateDistIfLower(d)
    if relaxed { activeNodes.inc(); Q.insert(v) }
  }
  activeNodes.dec()
}
```

Parallel Dijkstra: A Brief Overview

```
val Q = ConcurrentPriorityQueue<Node>()
start.distance = 0
Q.add(start)
activeNodes := 1

while activeNodes > 0 {
  u := Q.delete()
  for (v : u.edges) {
    d := u.distance + v.weight
    relaxed := v.updateDistIfLower(d)
    if relaxed { activeNodes.inc(); Q.insert(v) }
  }
  activeNodes.dec()
}
```

T threads

Finally correct!

Parallel Dijkstra: Trade-Offs

- *Sequential* Dijkstra: visits each node **exactly once**
- *Parallel* Dijkstra: may process nodes **multiple** times

Parallel Dijkstra: Trade-Offs

- *Sequential* Dijkstra: visits each node **exactly once**
- *Parallel* Dijkstra: may process nodes **multiple** times

- What do we win and lose?
    - **Win:** parallel edge processing
    - **Loss:** additional waste work

Parallel Dijkstra: Trade-Offs

- *Sequential* Dijkstra: visits each node **exactly once**
- *Parallel* Dijkstra: may process nodes **multiple** times

- What do we win and lose?
    - **Win:** parallel edge processing
    - **Loss:** additional waste work

On the real-world graphs, **Win >> Loss**

Priority Scheduler for Iterative Algorithms

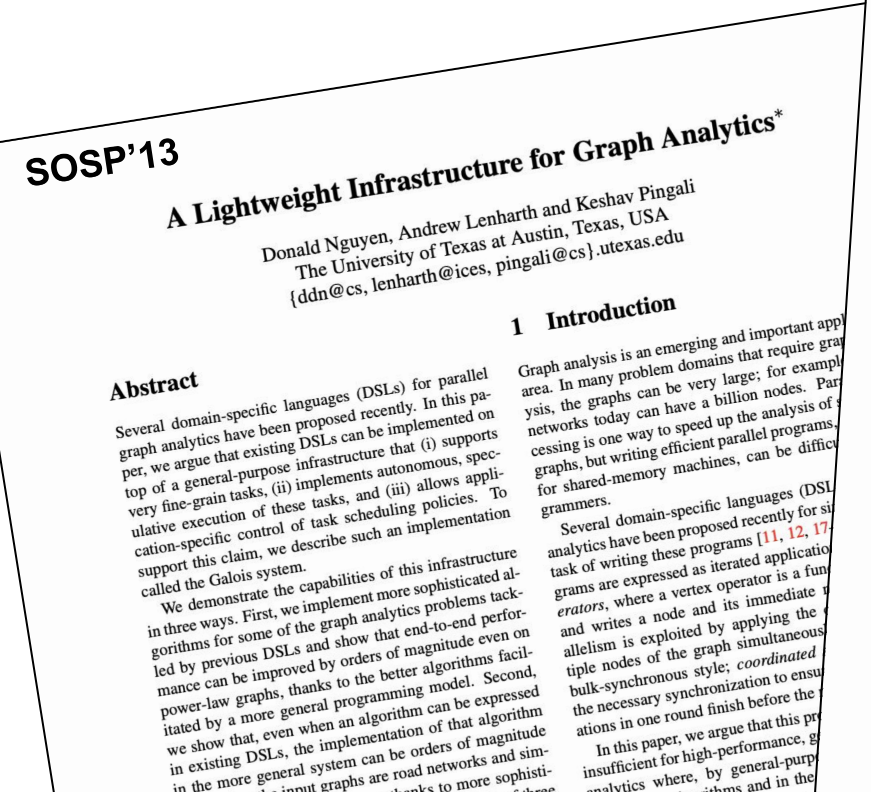
- Should be *fast*
- Should be *scalable*

Priority Scheduler for Iterative Algorithms

- Should be *fast*
- Should be *scalable*

- **DOES NOT NEED TO BE FAIR!**
  - Yet, should be *fair enough*

The State-of-the-Art: OBIM and PMOD



SOSP'13

A Lightweight Infrastructure for Graph Analytics*

Donald Nguyen, Andrew Lenharth and Keshav Pingali
The University of Texas at Austin, Texas, USA
{ddn@cs, lenharth@ices, pingali@cs}.utexas.edu

Abstract

Several domain-specific languages (DSLs) for parallel graph analytics have been proposed recently. In this paper, we argue that existing DSLs can be implemented on top of a general-purpose infrastructure that (i) supports very fine-grain tasks, (ii) implements autonomous, speculative execution of these tasks, and (iii) allows application-specific control of task scheduling policies. To support this claim, we describe such an implementation called the Galois system.

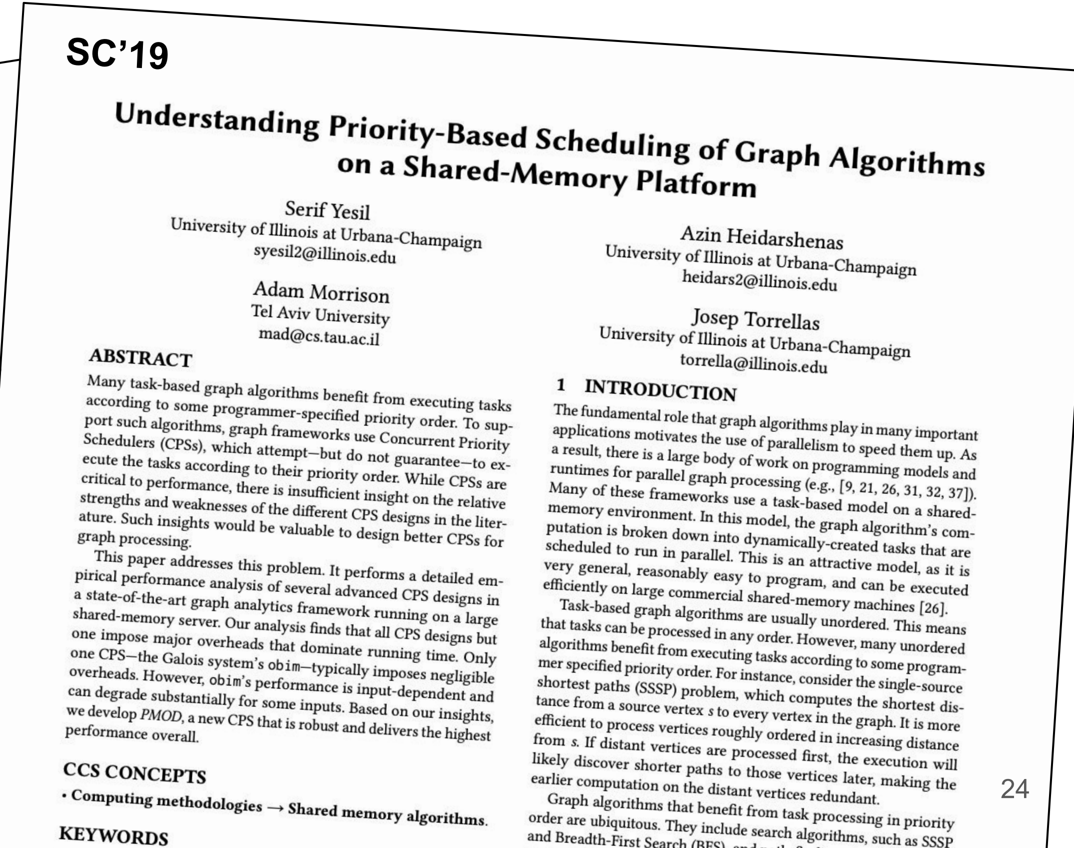
We demonstrate the capabilities of this infrastructure in three ways. First, we implement more sophisticated algorithms for some of the graph analytics problems tackled by previous DSLs and show that end-to-end performance can be improved by orders of magnitude even on power-law graphs, thanks to the better algorithms facilitated by a more general programming model. Second, we show that, even when an algorithm can be expressed in existing DSLs, the implementation of that algorithm in the more general system can be orders of magnitude

1 Introduction

Graph analysis is an emerging and important appl
area. In many problem domains that require grap
ysis, the graphs can be very large; for exampl
networks today can have a billion nodes. Par
cessing is one way to speed up the analysis of
graphs, but writing efficient parallel programs,
for shared-memory machines, can be difficu
grammers.

Several domain-specific languages (DSL
analytics have been proposed recently for si
task of writing these programs [11, 12, 17
grams are expressed as iterated applicatio
erators, where a vertex operator is a fun
and writes a node and its immediate
allelism is exploited by applying the
tiple nodes of the graph simultaneous
bulk-synchronous style; *coordinated*
the necessary synchronization to ensu
ations in one round finish before the

In this paper, we argue that this pr
insufficient for high-performance, g



SC'19

Understanding Priority-Based Scheduling of Graph Algorithms on a Shared-Memory Platform

Serif Yesil
University of Illinois at Urbana-Champaign
syesil2@illinois.edu

Adam Morrison
Tel Aviv University
mad@cs.tau.ac.il

Azin Heidarshenas
University of Illinois at Urbana-Champaign
heidars2@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

ABSTRACT

Many task-based graph algorithms benefit from executing tasks according to some programmer-specified priority order. To support such algorithms, graph frameworks use Concurrent Priority Schedulers (CPSs), which attempt—but do not guarantee—to execute the tasks according to their priority order. While CPSs are critical to performance, there is insufficient insight on the relative strengths and weaknesses of the different CPS designs in the literature. Such insights would be valuable to design better CPSs for graph processing.

This paper addresses this problem. It performs a detailed empirical performance analysis of several advanced CPS designs in a state-of-the-art graph analytics framework running on a large shared-memory server. Our analysis finds that all CPS designs but one impose major overheads that dominate running time. Only one CPS—the Galois system's `obim`—typically imposes negligible overheads. However, `obim`'s performance is input-dependent and can degrade substantially for some inputs. Based on our insights, we develop *PMOD*, a new CPS that is robust and delivers the highest performance overall.

CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms.**

KEYWORDS

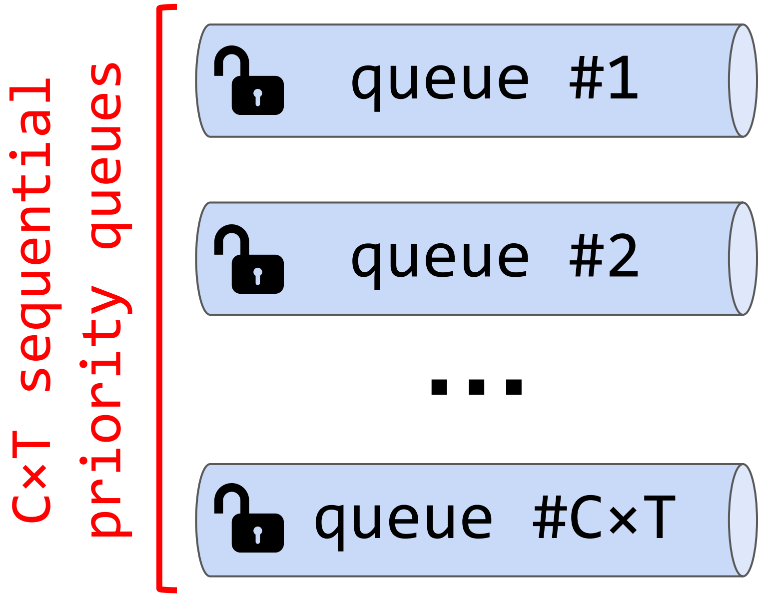
1 INTRODUCTION

The fundamental role that graph algorithms play in many important applications motivates the use of parallelism to speed them up. As a result, there is a large body of work on programming models and runtimes for parallel graph processing (e.g., [9, 21, 26, 31, 32, 37]). Many of these frameworks use a task-based model on a shared-memory environment. In this model, the graph algorithm's computation is broken down into dynamically-created tasks that are scheduled to run in parallel. This is an attractive model, as it is very general, reasonably easy to program, and can be executed efficiently on large commercial shared-memory machines [26].

Task-based graph algorithms are usually unordered. This means that tasks can be processed in any order. However, many unordered algorithms benefit from executing tasks according to some programmer specified priority order. For instance, consider the single-source shortest paths (SSSP) problem, which computes the shortest distance from a source vertex *s* to every vertex in the graph. It is more efficient to process vertices roughly ordered in increasing distance from *s*. If distant vertices are processed first, the execution will likely discover shorter paths to those vertices later, making the earlier computation on the distant vertices redundant.

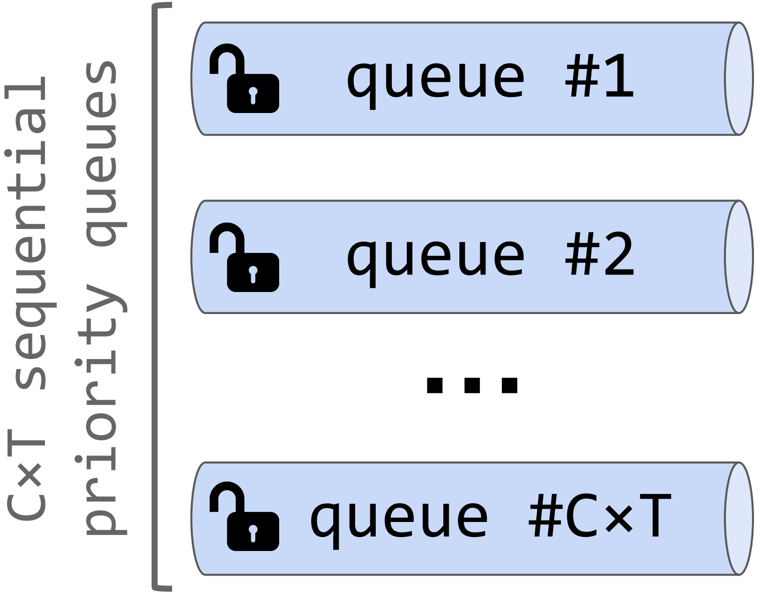
Graph algorithms that benefit from task processing in priority order are ubiquitous. They include search algorithms, such as SSSP

Multi-Queue Priority Scheduler



Multi-Queue Priority Scheduler

```
val queues := PQ<E>[C×T]
```



C×T sequential priority queues

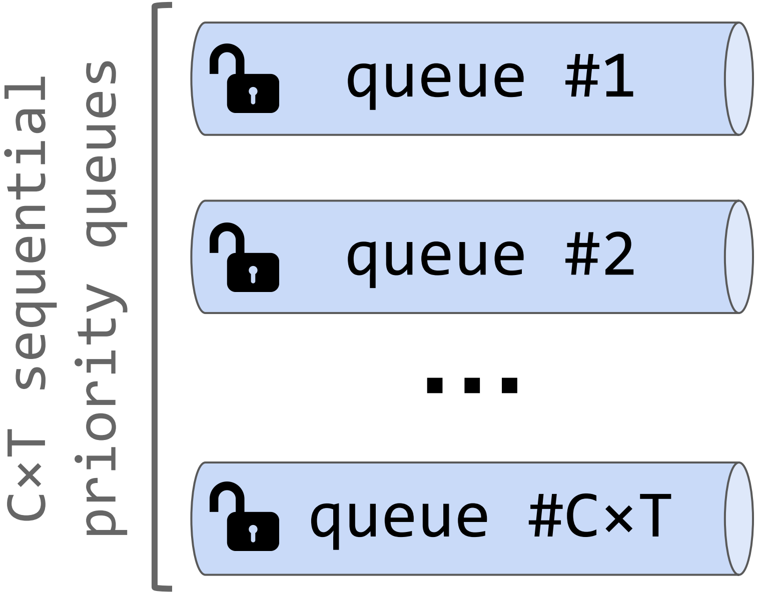
queue #1

queue #2

...

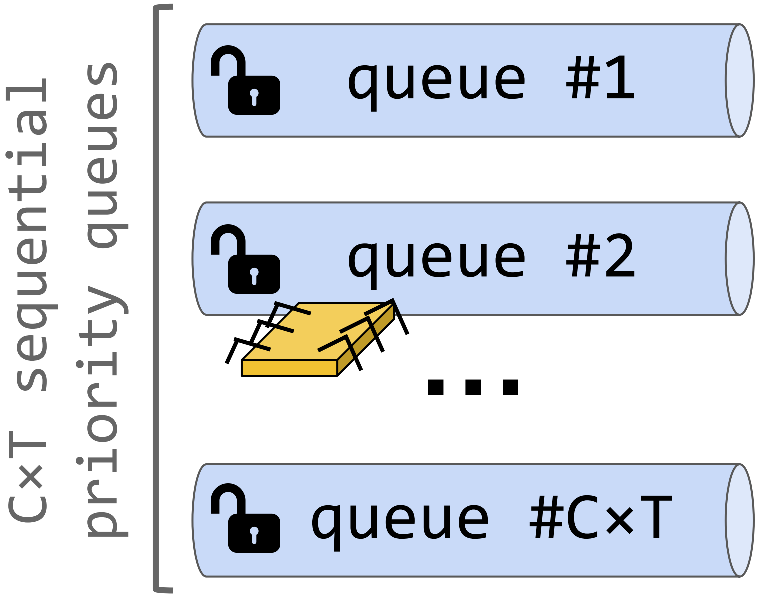
queue #C×T

Multi-Queue Priority Scheduler



```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}
```

Multi-Queue Priority Scheduler



C×T sequential priority queues

queue #1

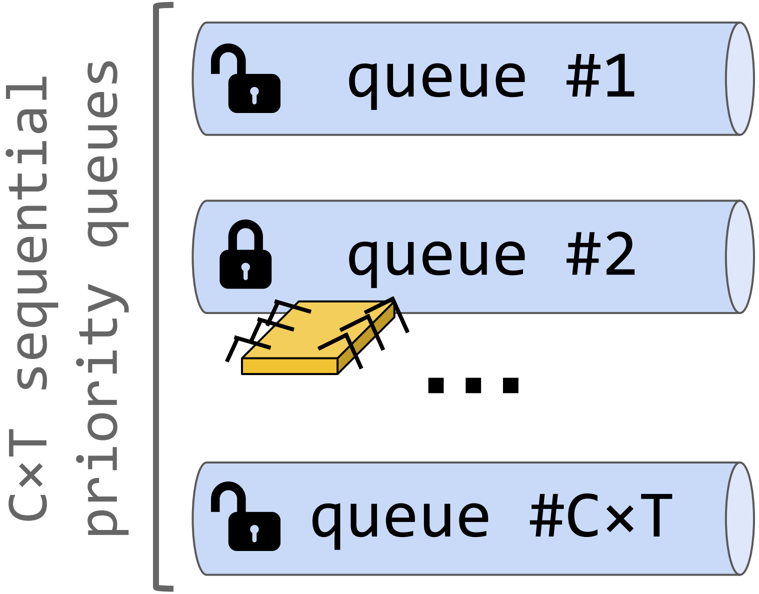
queue #2

...

queue #C×T

```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}
```

Multi-Queue Priority Scheduler


C×T sequential priority queues

queue #1

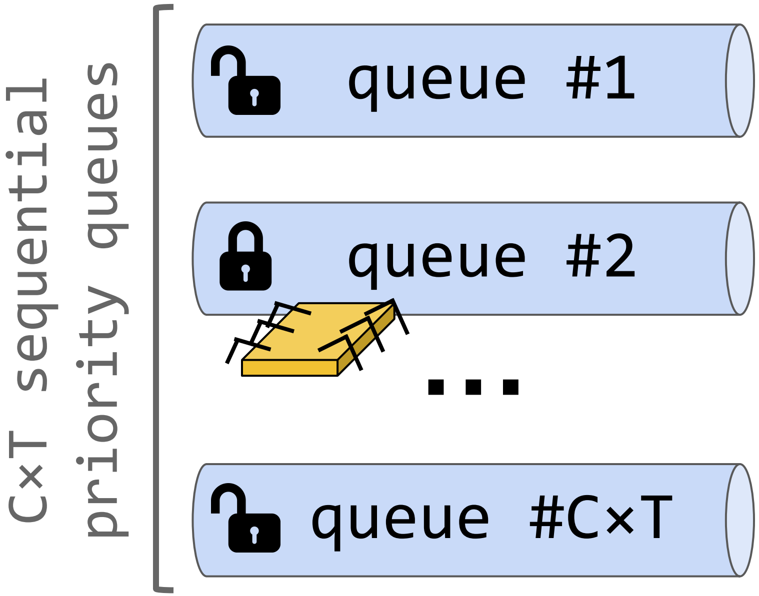
queue #2

...

queue #C×T

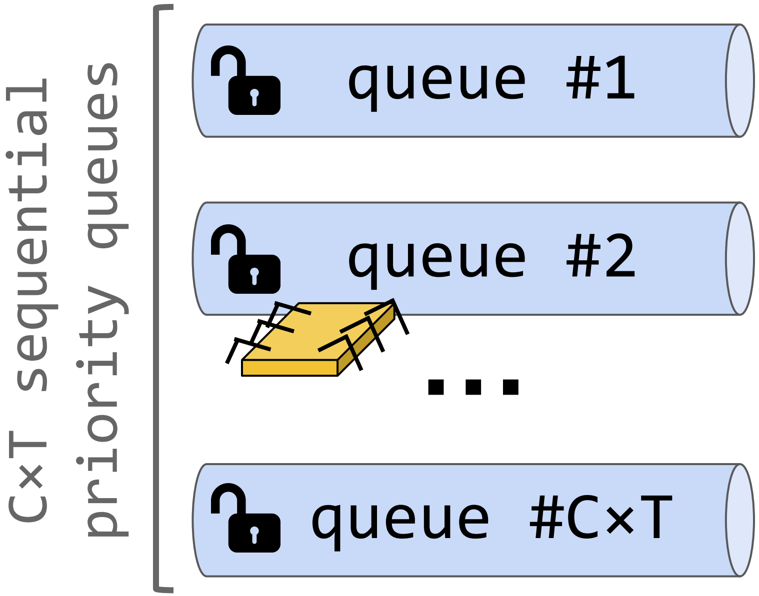
```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}
```

Multi-Queue Priority Scheduler



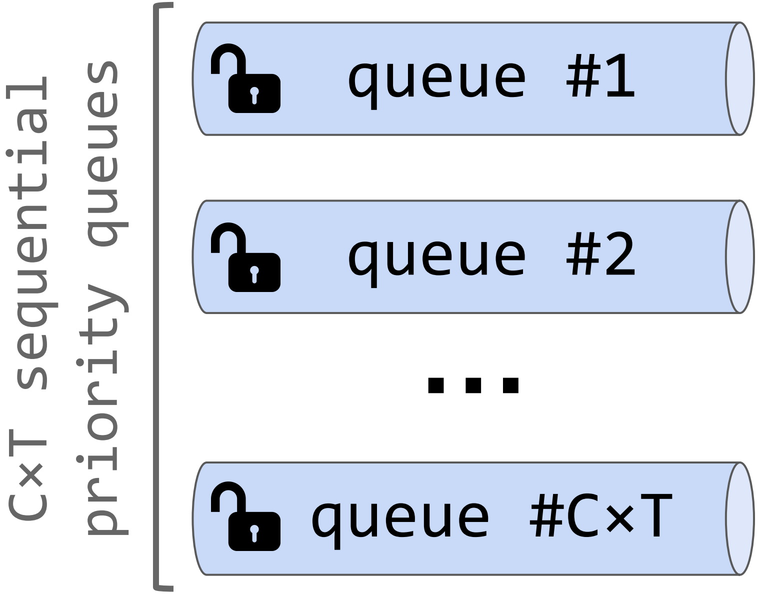
```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}
```

Multi-Queue Priority Scheduler

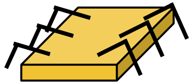


```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}
```

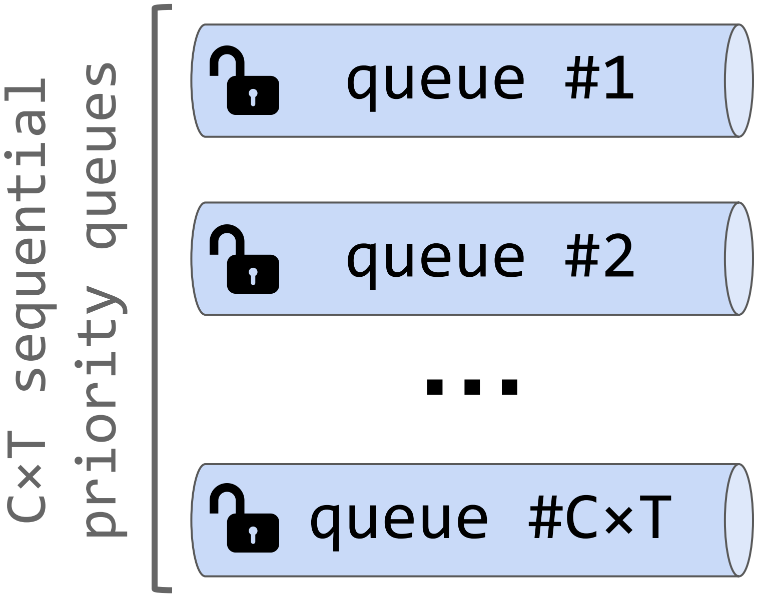
Multi-Queue Priority Scheduler



```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}
```

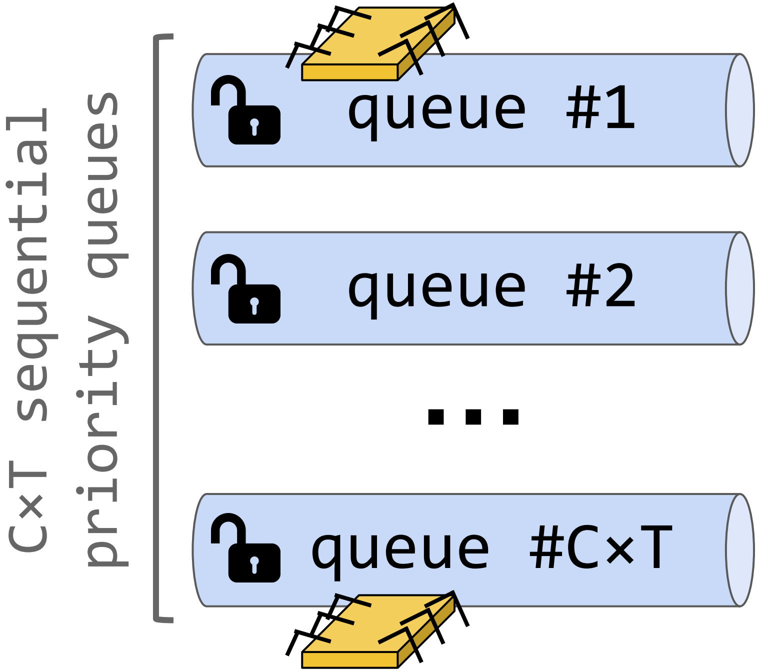


Multi-Queue Priority Scheduler



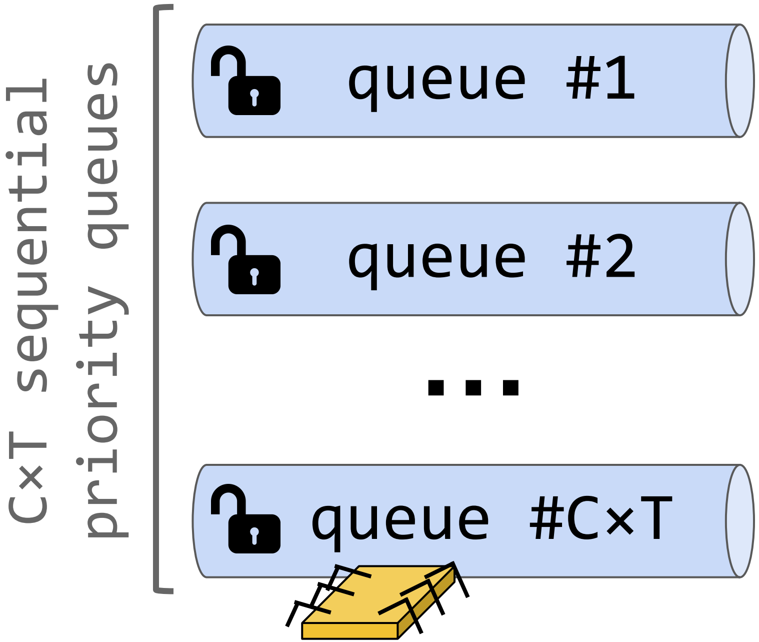
```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}

fun delete(): E? = while(true) {
  i1, i2 := distinctRandom(0, C×T)
  q1 := queues[i1]; q2 := queues[i2]
  q := q1.top() < q2.top() ? q1 : q2
  if !tryLock(q): continue
  task := q.extractTop()
  unlock(q)
  return task
}
```

Multi-Queue Priority Scheduler



```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}

fun delete(): E? = while(true) {
  i1, i2 := distinctRandom(0, C×T)
  q1 := queues[i1]; q2 := queues[i2]
  q := q1.top() < q2.top() ? q1 : q2
  if !tryLock(q): continue
  task := q.extractTop()
  unlock(q)
  return task
}
```

Multi-Queue Priority Scheduler



C×T sequential priority queues

queue #1

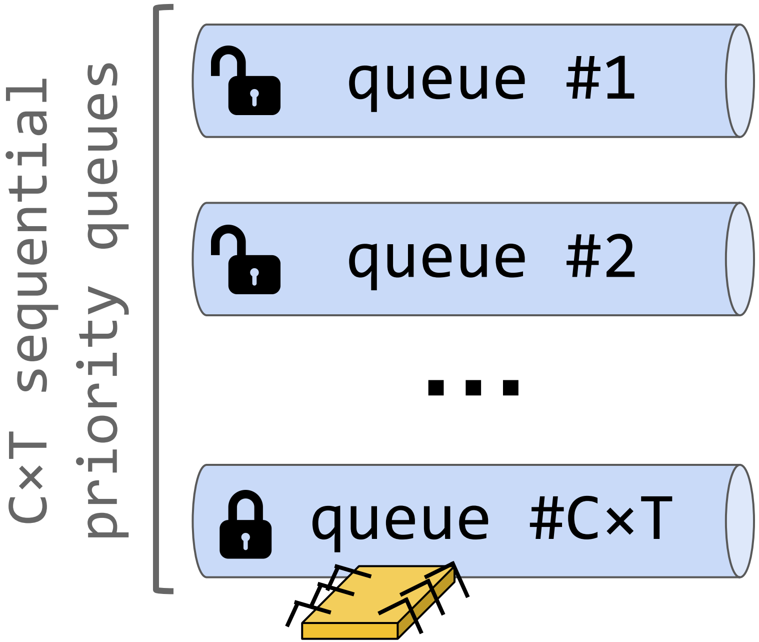
queue #2

...

queue #C×T

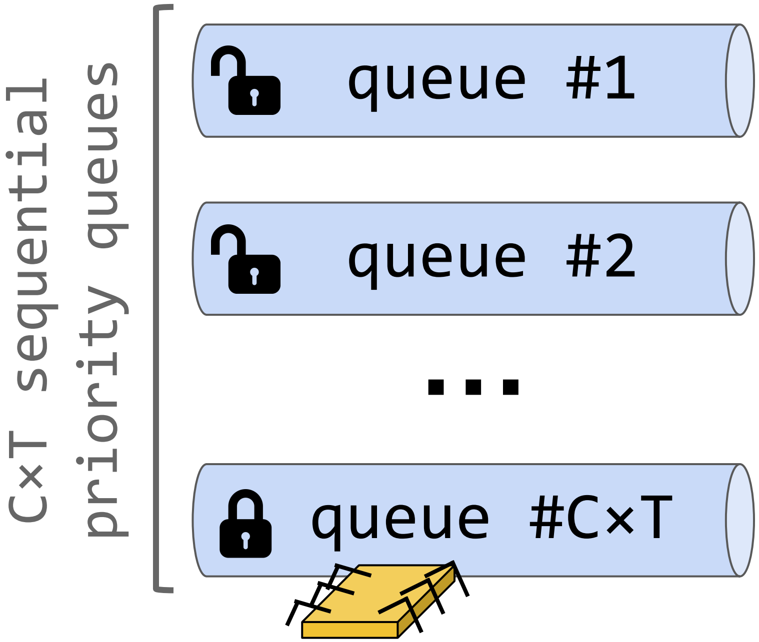
```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}

fun delete(): E? = while(true) {
  i1, i2 := distinctRandom(0, C×T)
  q1 := queues[i1]; q2 := queues[i2]
  q := q1.top() < q2.top() ? q1 : q2
  if !tryLock(q): continue
  task := q.extractTop()
  unlock(q)
  return task
}
```

Multi-Queue Priority Scheduler



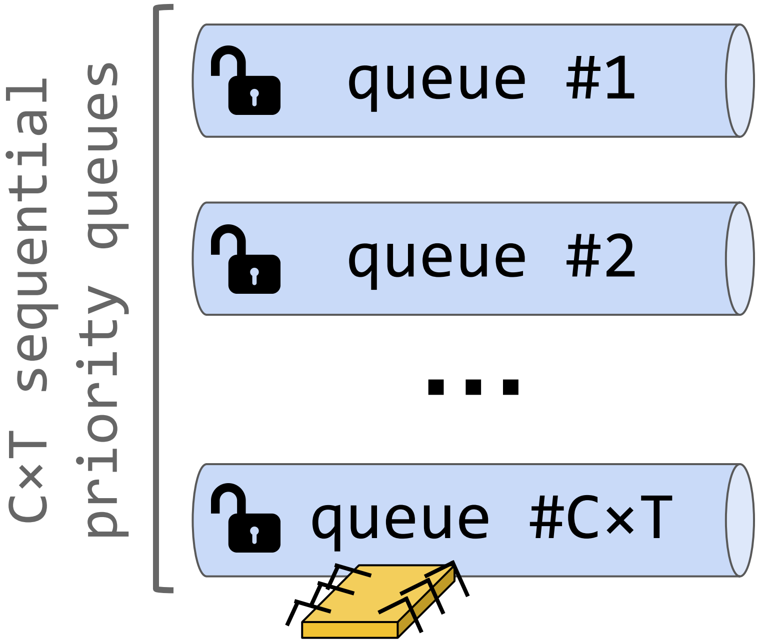
```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}

fun delete(): E? = while(true) {
  i1, i2 := distinctRandom(0, C×T)
  q1 := queues[i1]; q2 := queues[i2]
  q := q1.top() < q2.top() ? q1 : q2
  if !tryLock(q): continue
  task := q.extractTop()
  unlock(q)
  return task
}
```

Multi-Queue Priority Scheduler



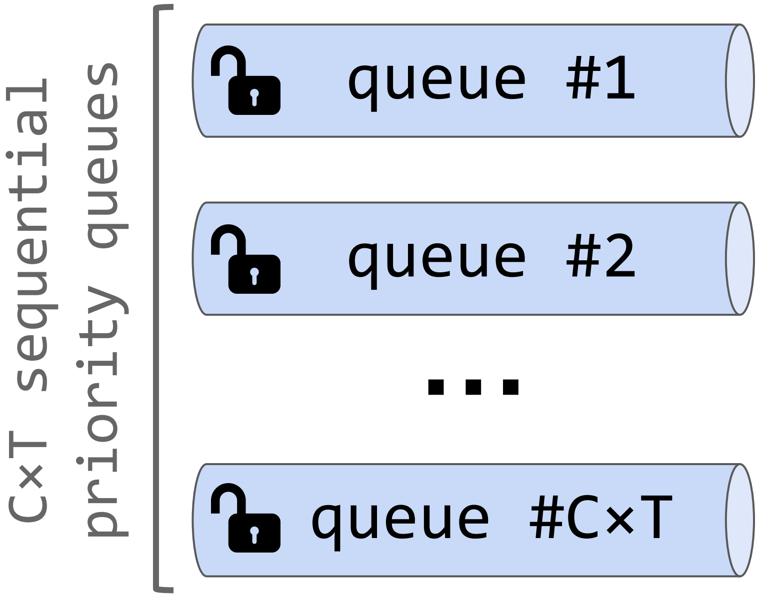
```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}

fun delete(): E? = while(true) {
  i1, i2 := distinctRandom(0, C×T)
  q1 := queues[i1]; q2 := queues[i2]
  q := q1.top() < q2.top() ? q1 : q2
  if !tryLock(q): continue
  task := q.extractTop()
  unlock(q)
  return task
}
```

Multi-Queue Priority Scheduler

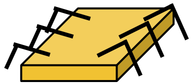


```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}

fun delete(): E? = while(true) {
  i1, i2 := distinctRandom(0, C×T)
  q1 := queues[i1]; q2 := queues[i2]
  q := q1.top() < q2.top() ? q1 : q2
  if !tryLock(q): continue
  task := q.extractTop()
  unlock(q)
  return task
}
```

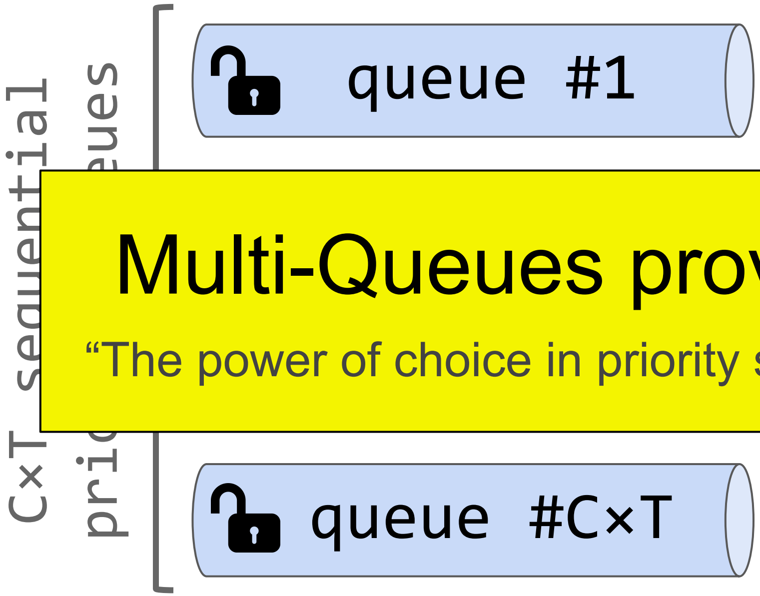
Multi-Queue Priority Scheduler



```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)
  unlock(q)
  return
}

fun delete(): E? = while(true) {
  i1, i2 := distinctRandom(0, C×T)
  q1 := queues[i1]; q2 := queues[i2]
  q := q1.top() < q2.top() ? q1 : q2
  if !tryLock(q): continue
  task := q.extractTop()
  unlock(q)
  return task
}
```



Multi-Queue Priority Scheduler

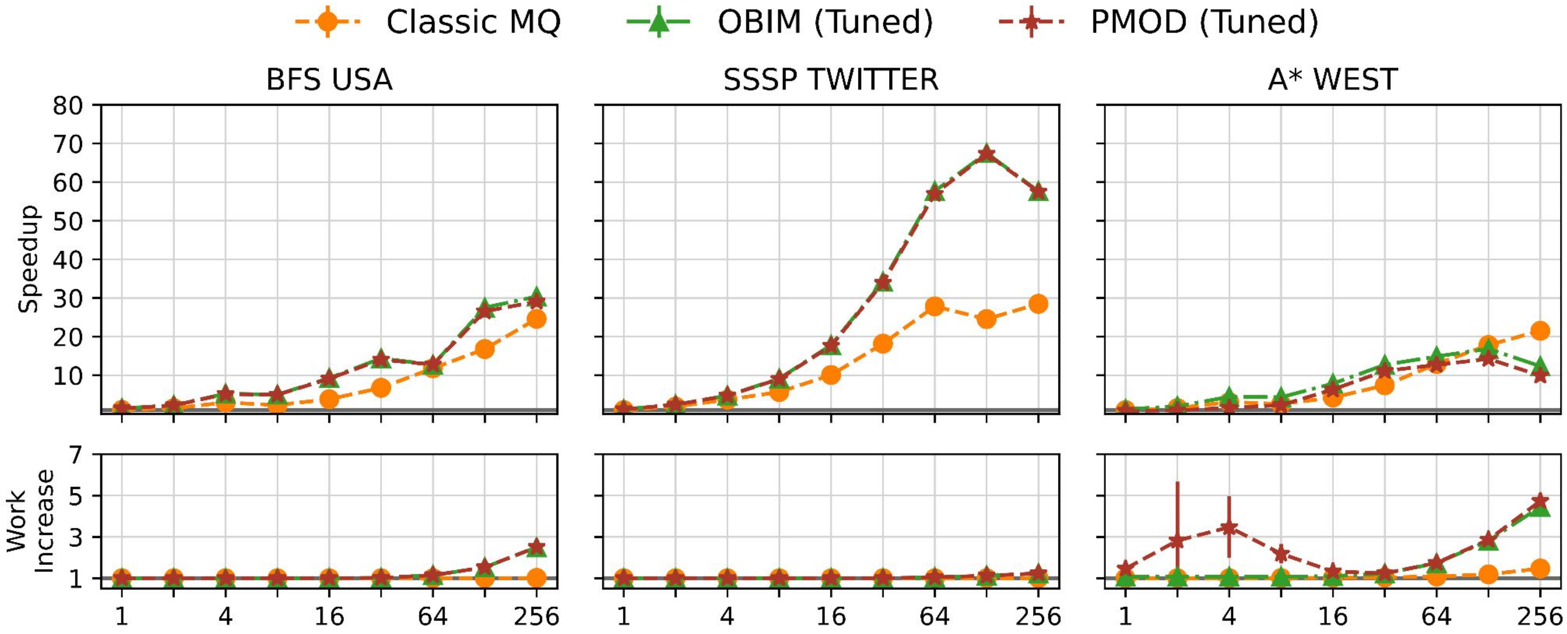


```
val queues := PQ<E>[C×T]

fun insert(task: E) = while(true) {
  q := queues[random(0, C×T)]
  if !tryLock(q): continue
  q.add(task)

  q1 := queues[i1]; q2 := queues[i2]
  q := q1.top() < q2.top() ? q1 : q2
  if !tryLock(q): continue
  task := q.extractTop()
  unlock(q)
  return task
}
```

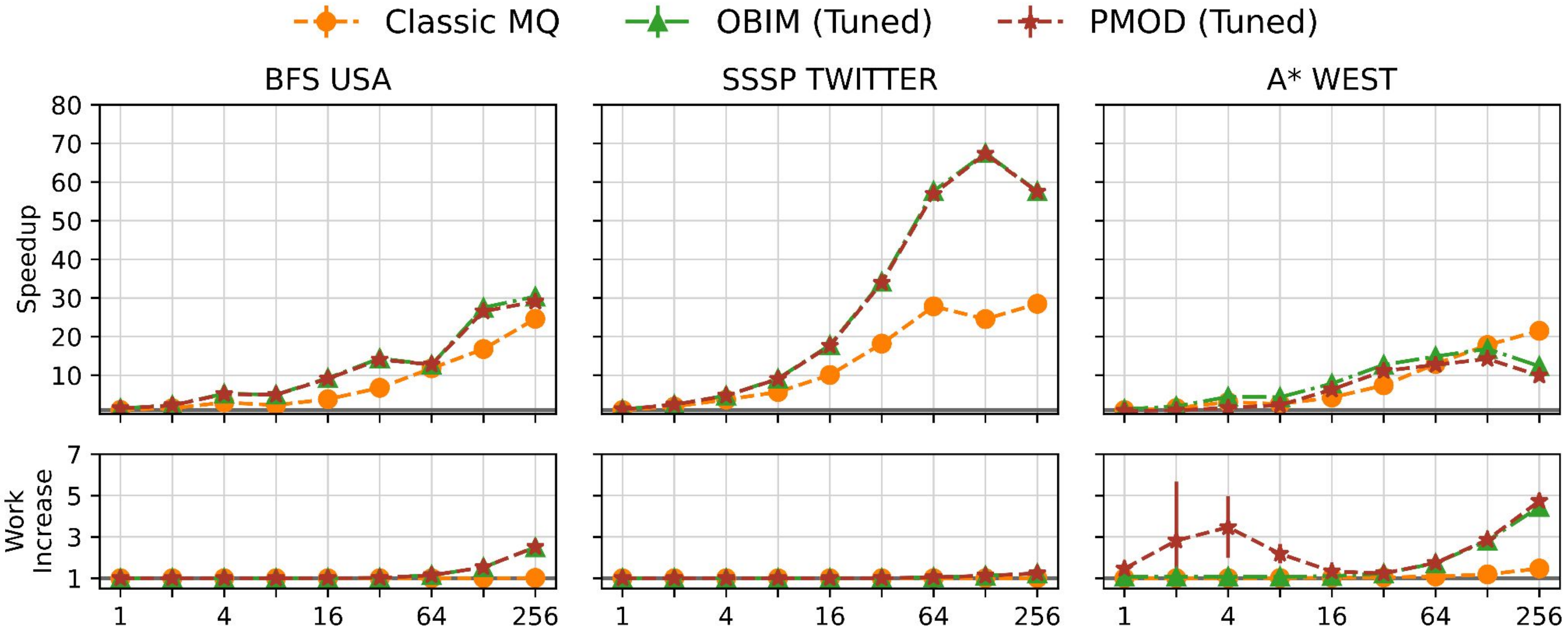
Multi-Queues provide fairness guarantees!

“The power of choice in priority scheduling” by Dan Alistarh et al. (PODC’17)

OBIM vs PMOD vs MQ

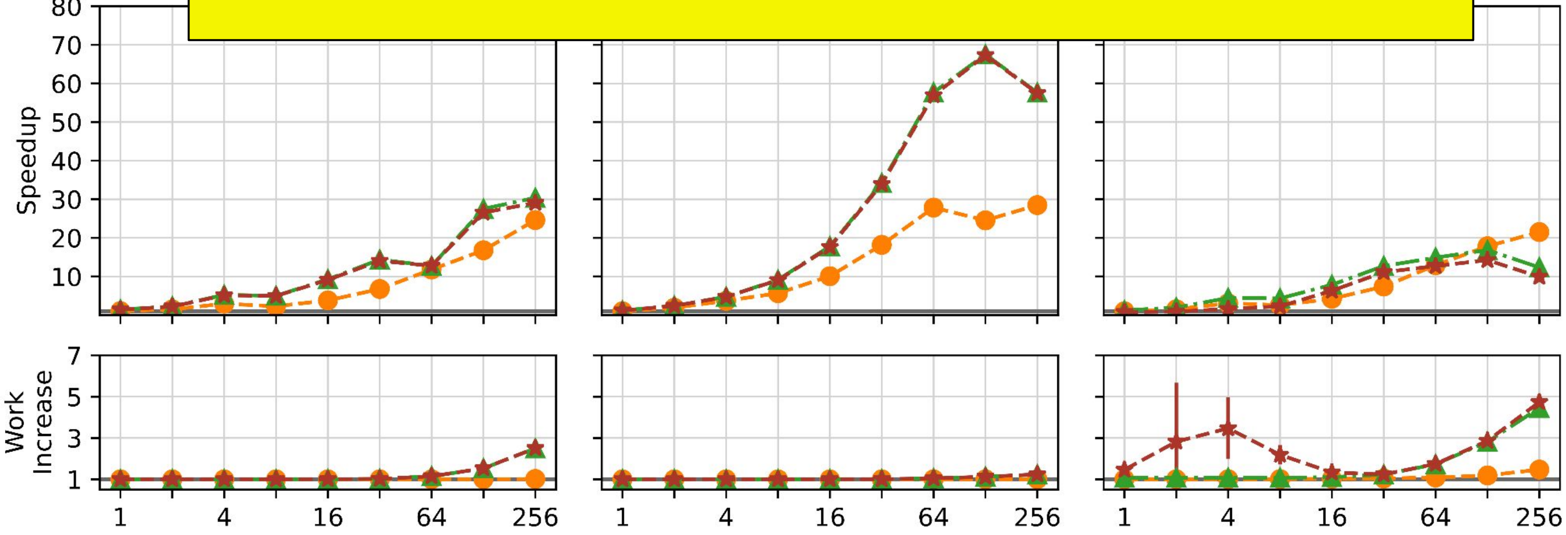


OBIM vs PMOD vs MQ



Much lower work increase with MQ

OBIM vs PMOD vs MQ

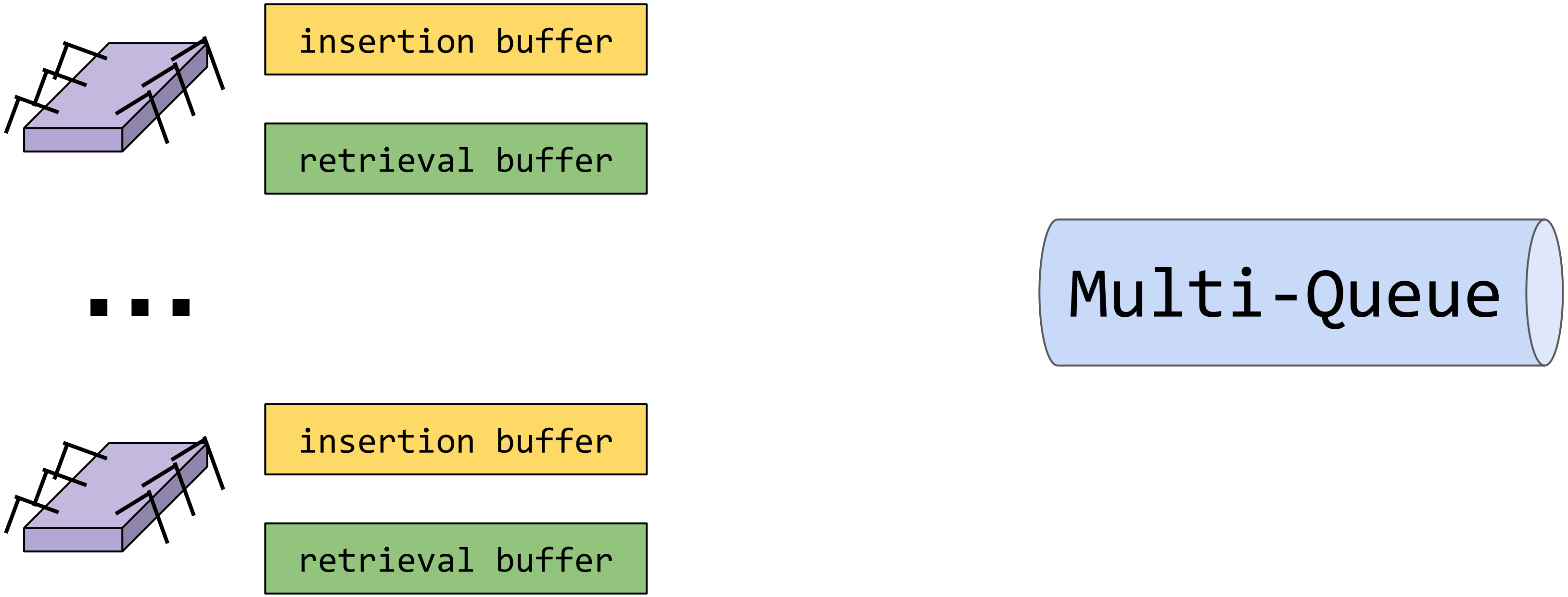


Faster iterations with OBIM / PMOD

Much lower work increase with MQ

Can we achieve better results with the Multi-Queue design?

MQ Optimizations: Task Batching



insertion buffer

retrieval buffer

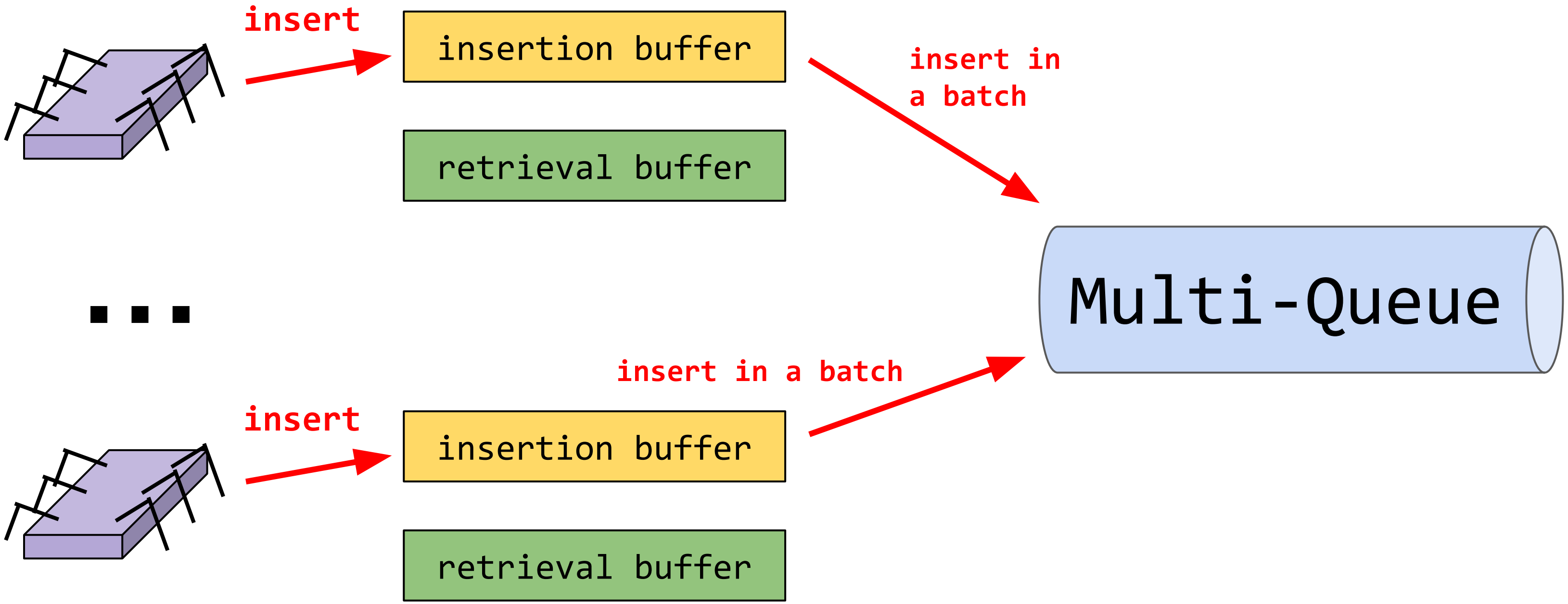
...

insertion buffer

retrieval buffer

Multi-Queue

MQ Optimizations: Task Batching



insert

insertion buffer

retrieval buffer

insert in
a batch

...

Multi-Queue

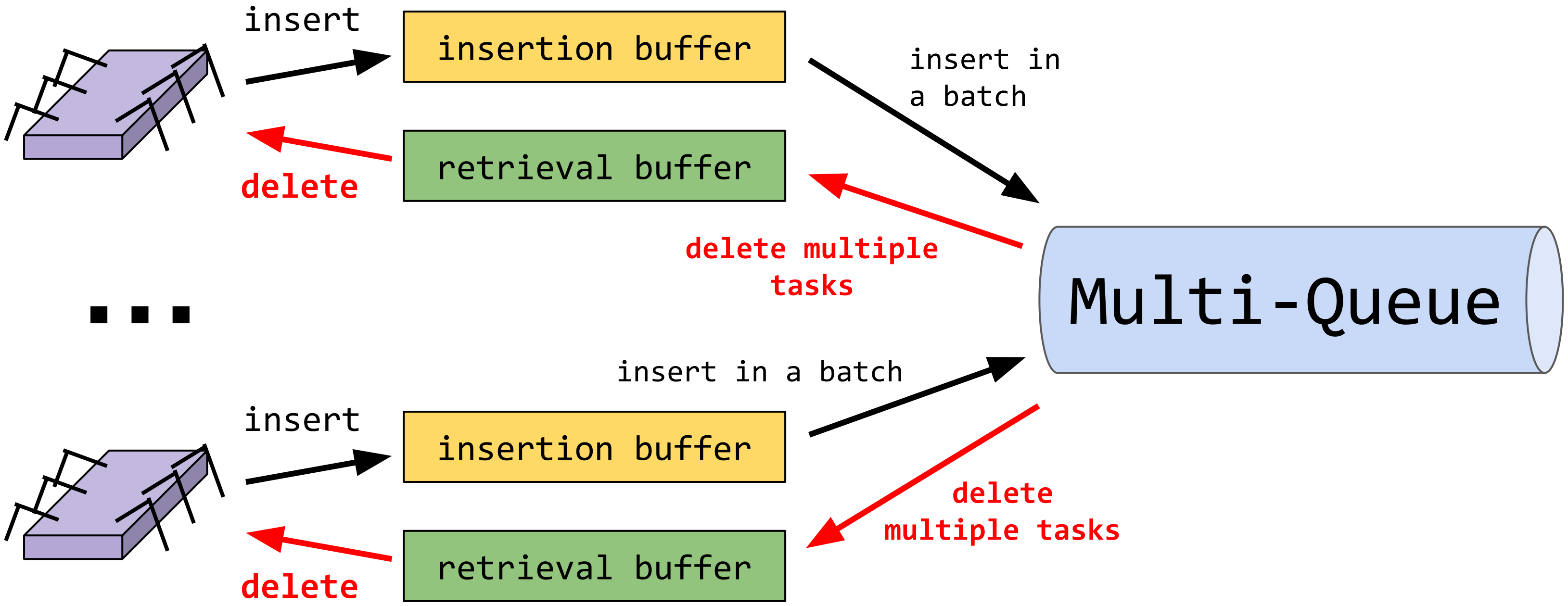
insert in a batch

insert

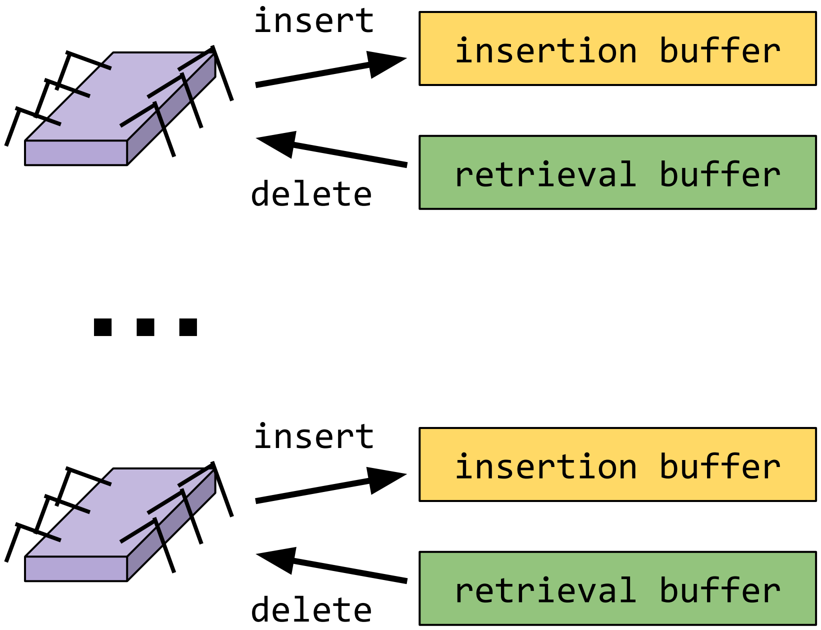
insertion buffer

retrieval buffer

MQ Optimizations: Task Batching



MQ Optimizations: Task Batching



insert

insertion buffer

retrieval buffer

delete

...

insert

insertion buffer

retrieval buffer

delete

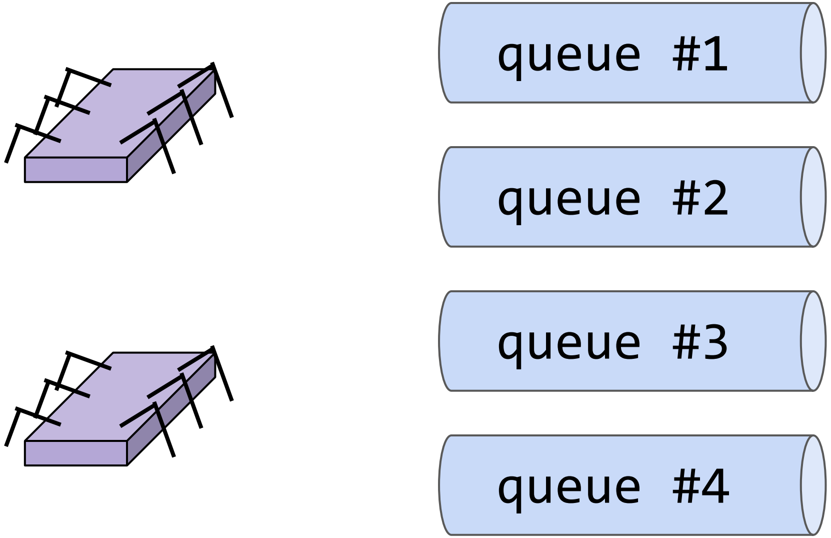
Win:

- less lock acquisitions
- less cache misses
- lower contention

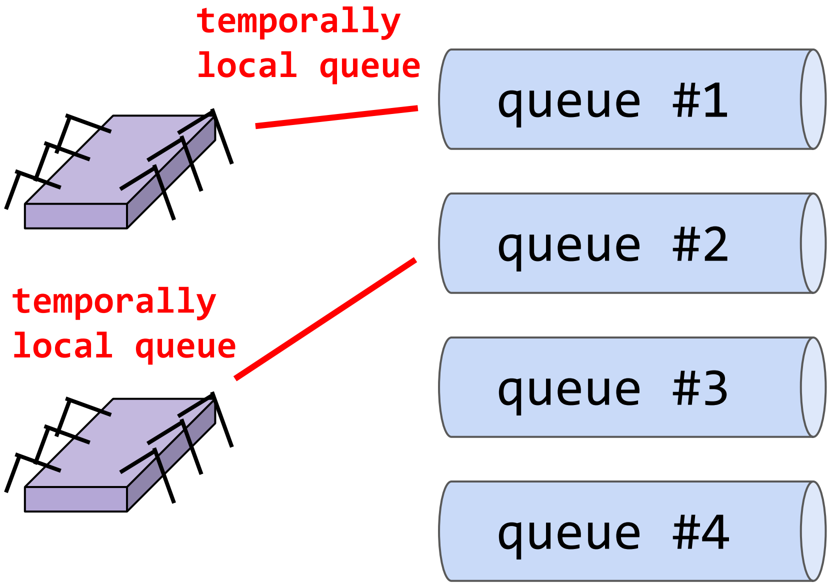
Loss:

- lower fairness

MQ Optimizations: Temporal Locality

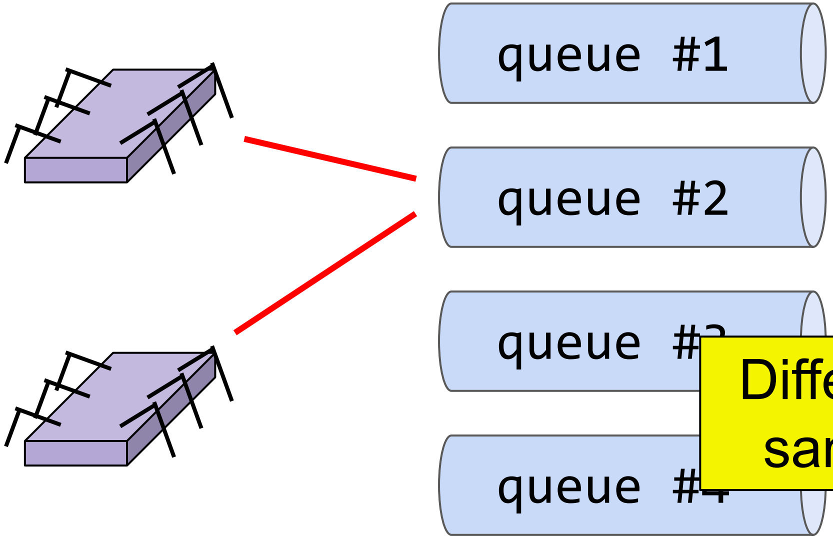


MQ Optimizations: Temporal Locality



Work with the temporally local queue, changing it with some probability

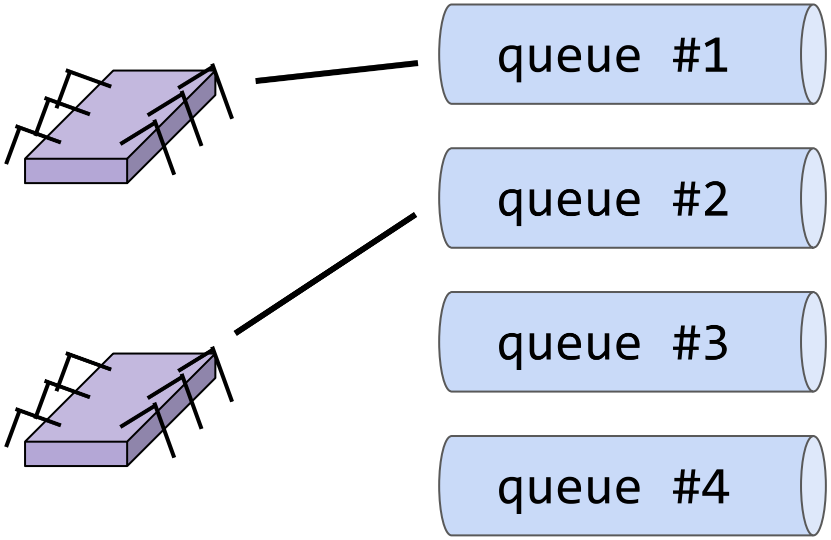
MQ Optimizations: Temporal Locality



Different threads may use the same queue (rare situation)

Work with the temporally local queue, changing it with some probability

MQ Optimizations: Temporal Locality



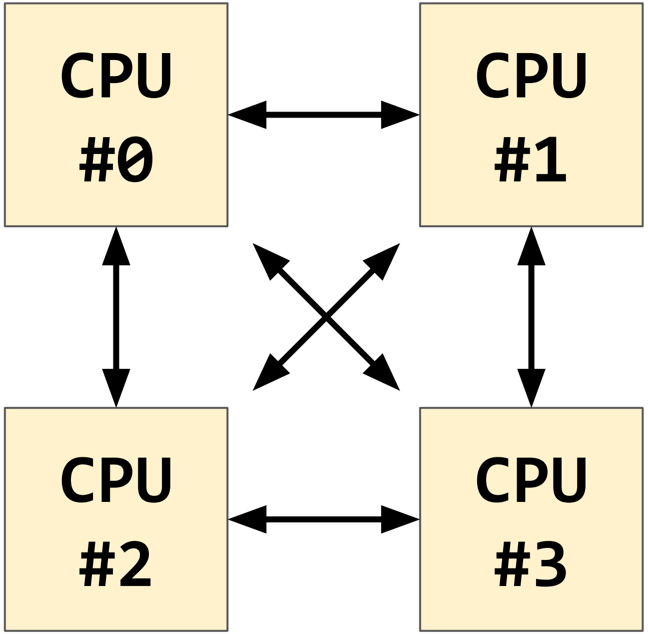
Win:

- better fairness compared to batching

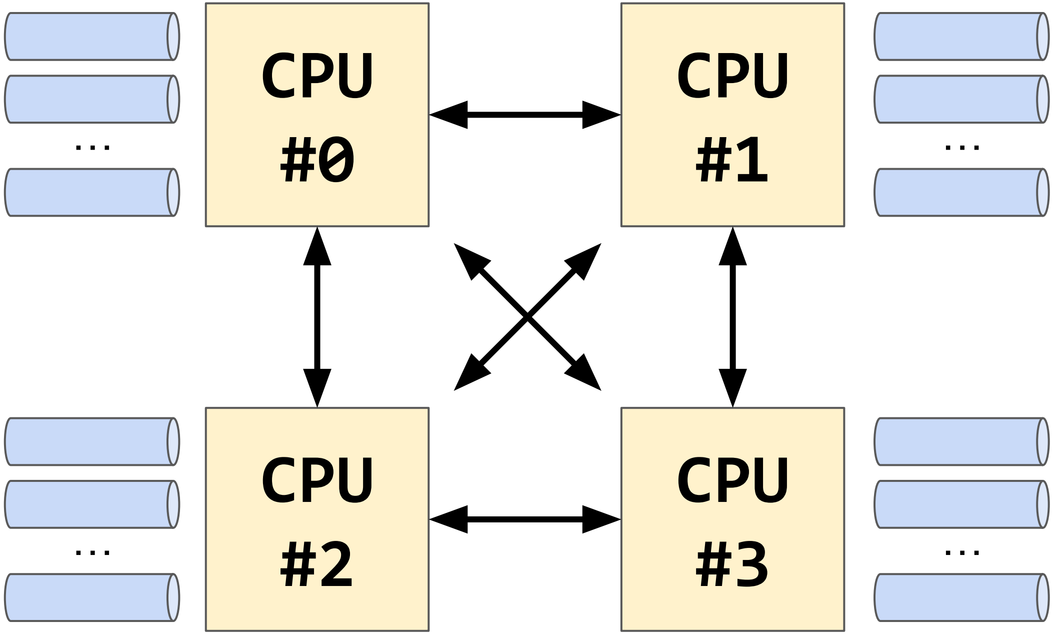
Loss:

- acquires/releases locks on every operation

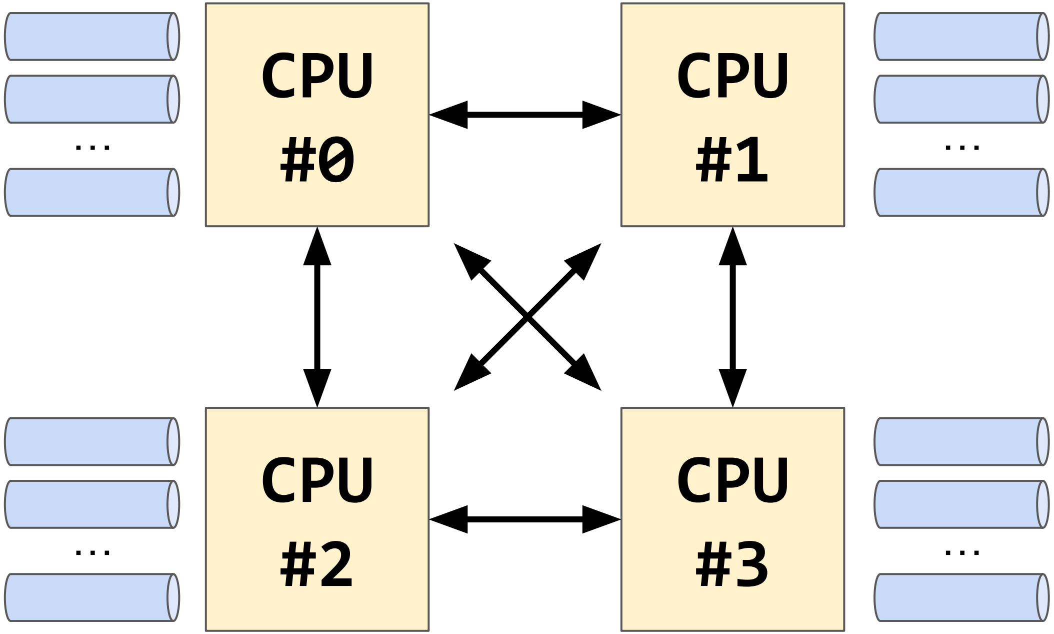
MQ Optimizations: NUMA-Awareness



MQ Optimizations: NUMA-Awareness

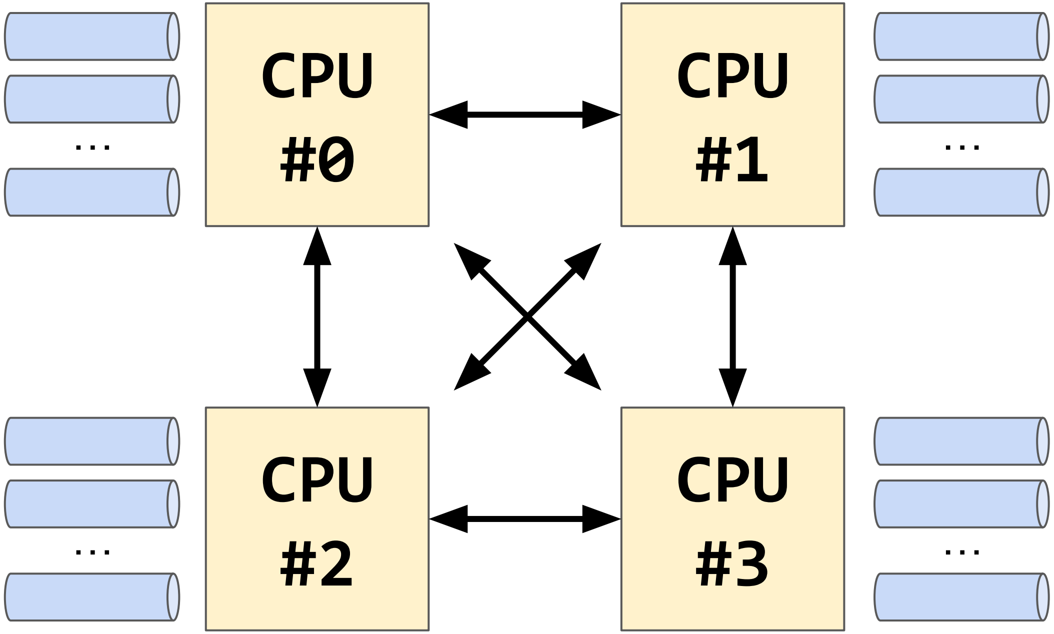


MQ Optimizations: NUMA-Awareness



- Choose a queue in the same socket with higher probability
- Never use out-of-the-socket queues as local ones with *temporal locality*

MQ Optimizations: NUMA-Awareness



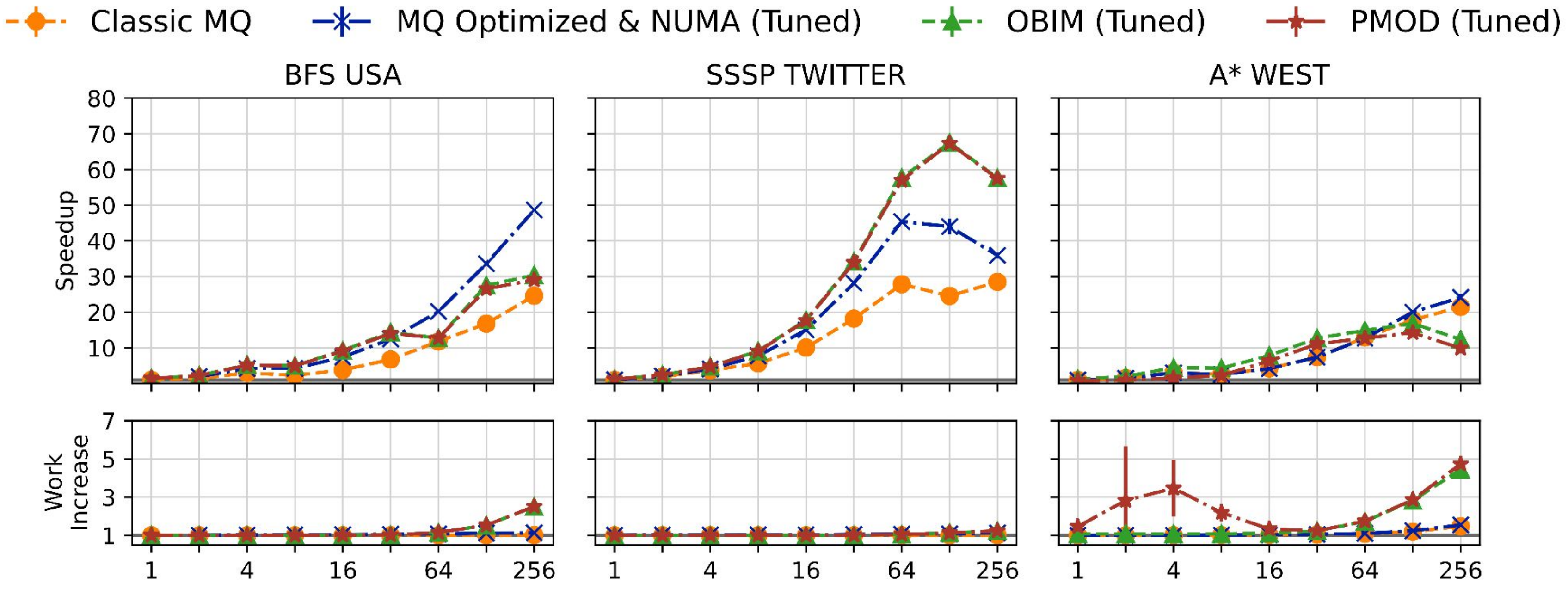
Win:

- less out-of-socket accesses

Loss:

- lower fairness

OBIM vs PMOD vs MQ vs MQ-Optimized



Significant improvement over the classic MQ

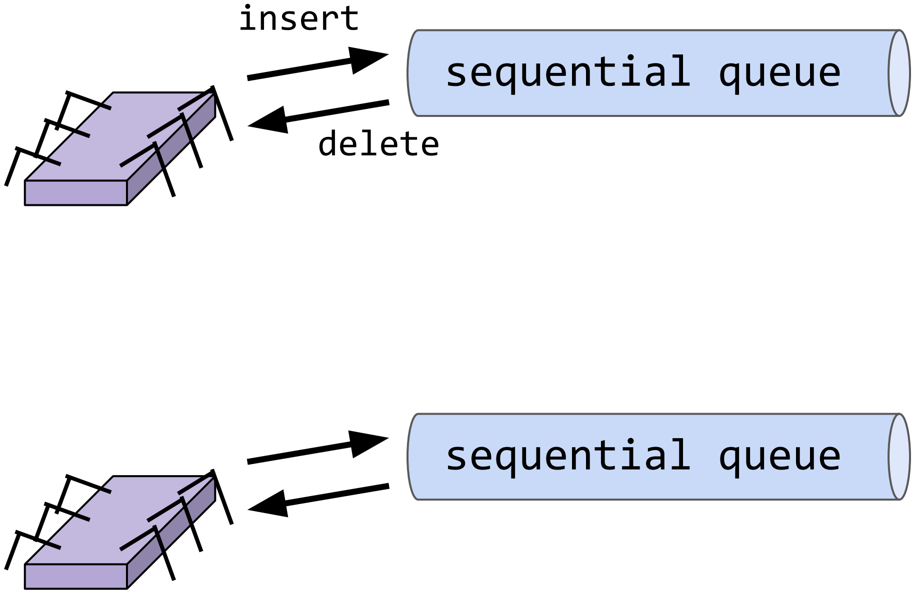
MQ and MQ-Optimized Fairness

	Average rank	Maximum rank (with high probability)
MQ:	$O\left(\frac{n}{\beta^2}\right)$	$O\left(\frac{1}{\alpha}n(\log n + \log C)\right)$
MQ-Optimized:	$O\left(\frac{nB(1+\gamma)}{p_{steal}}\left(\log n + \log\frac{(1+\gamma)}{p_{steal}}\right)\right)$	$O\left(\frac{nB(1+\gamma)}{p_{steal}}\log\frac{(1+\gamma)}{p_{steal}}\right)$

MQ-Optimized provide essentially the same guarantees, with parametrization depending on choice probabilities

Can we do better?

SMQ: Stealing Multi-Queue



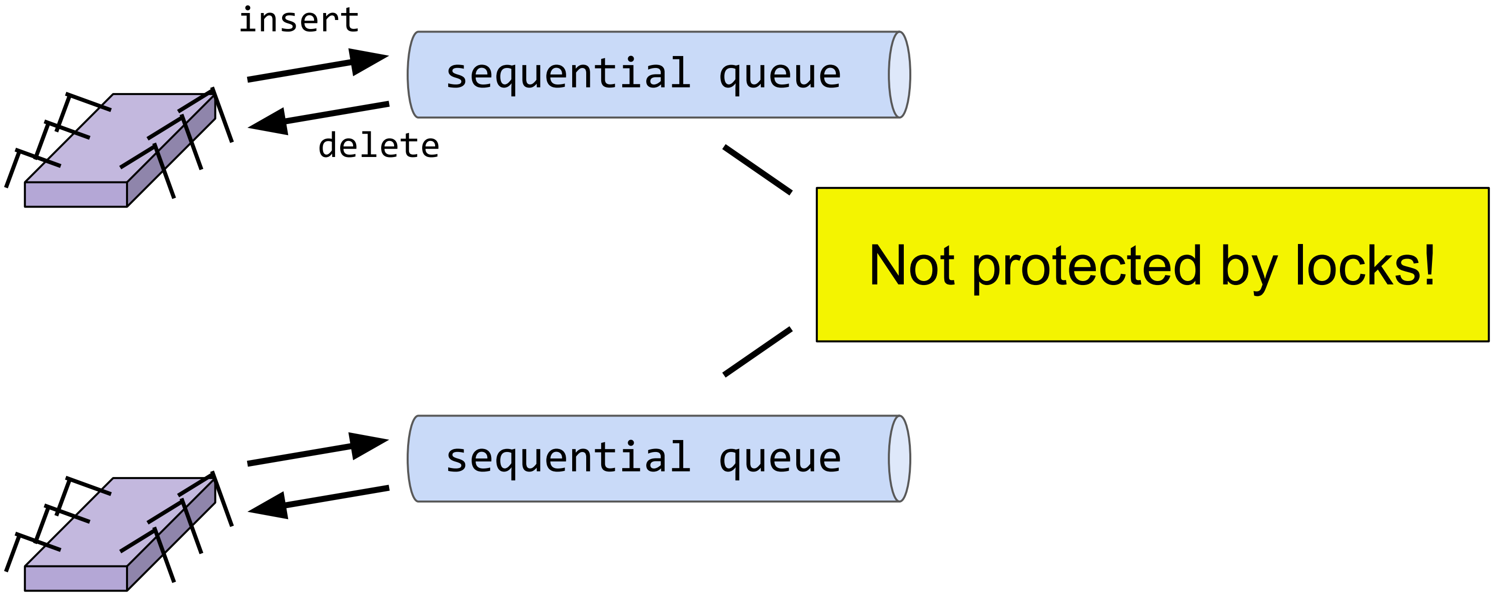
insert

sequential queue

delete

sequential queue

SMQ: Stealing Multi-Queue



insert

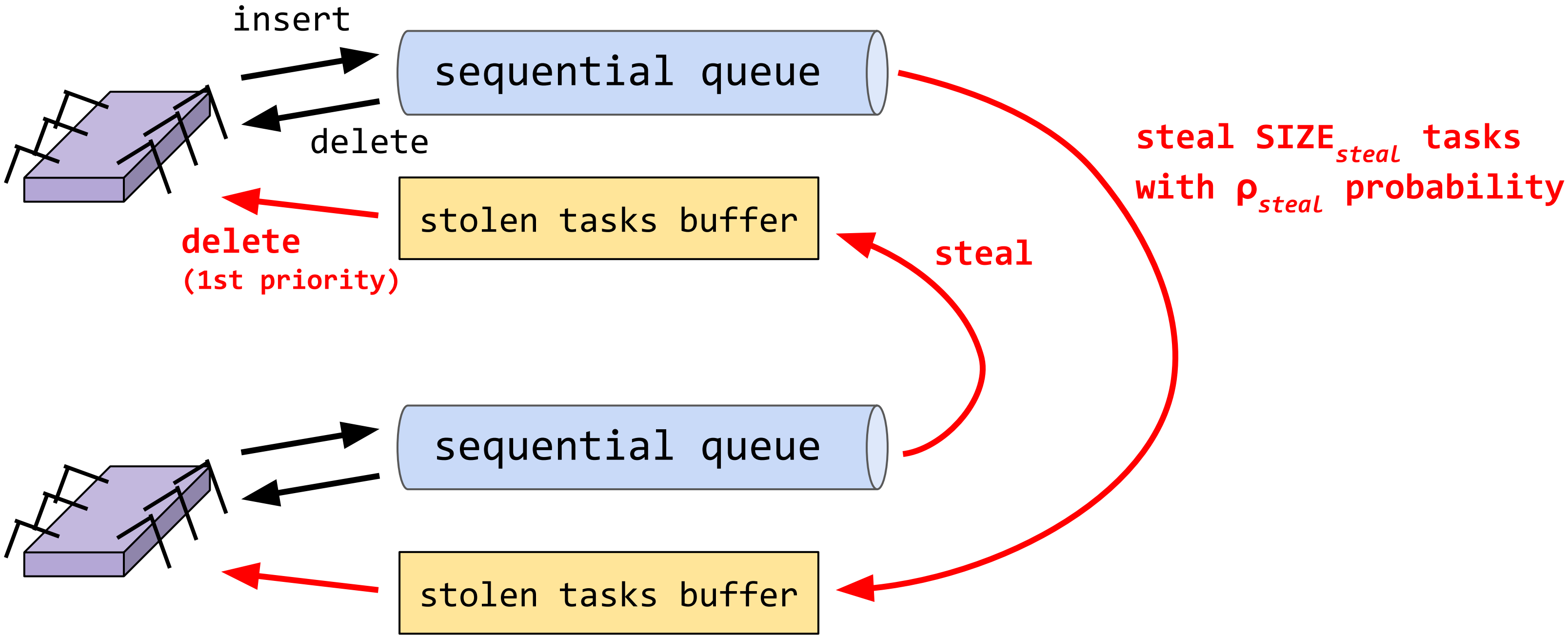
sequential queue

delete

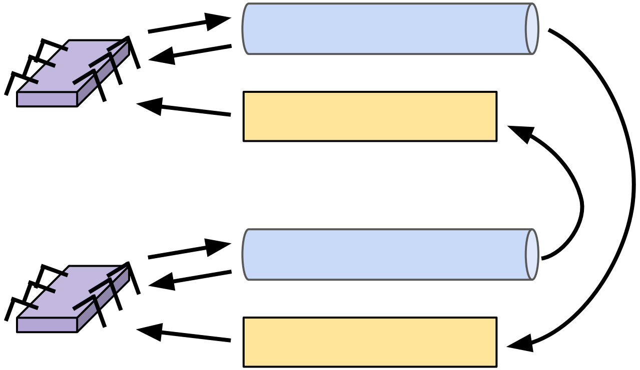
Not protected by locks!

sequential queue

SMQ: Stealing Multi-Queue

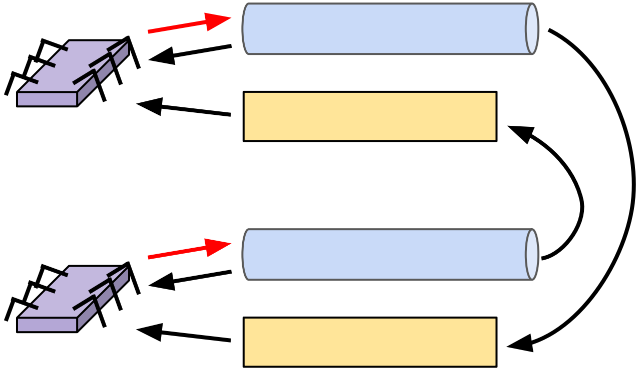


SMQ: Stealing Multi-Queue



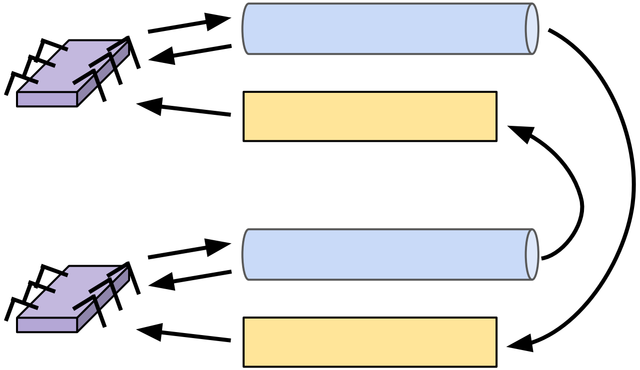
```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE_steal- 1)
```

SMQ: Stealing Multi-Queue



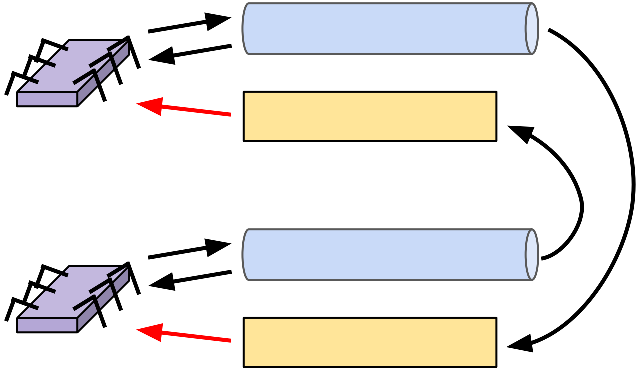
```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE_steal - 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}
```

SMQ: Stealing Multi-Queue



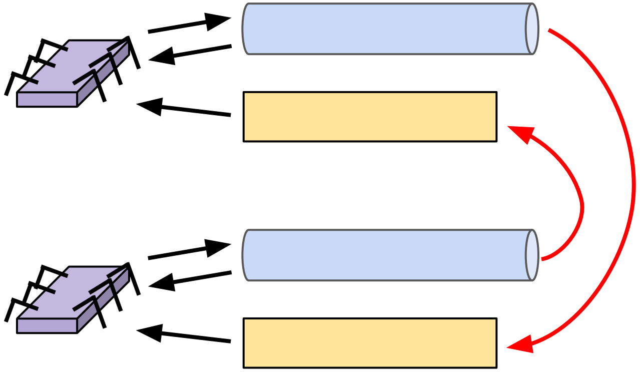
```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE_steal - 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {









}
```

SMQ: Stealing Multi-Queue



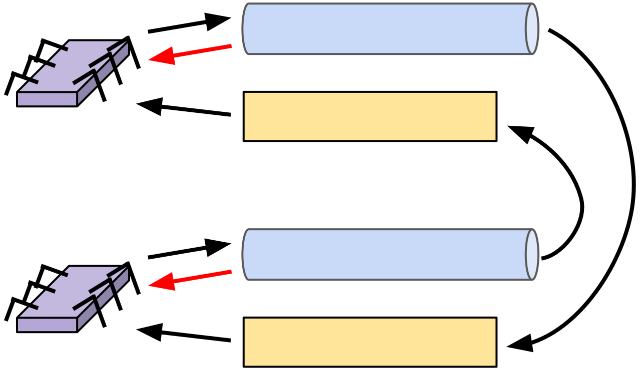
```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE_steal - 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()





}
```

SMQ: Stealing Multi-Queue



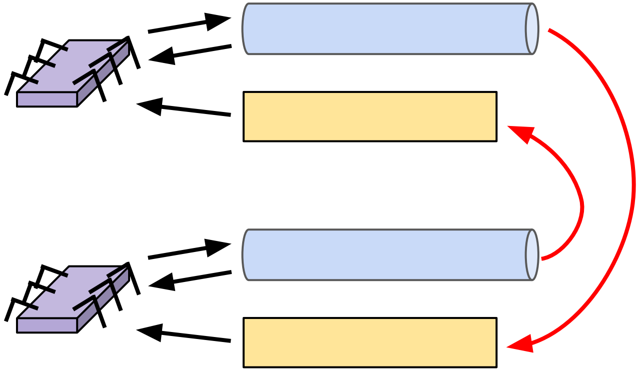
```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE_steal - 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()
  with ρ_steal probability {
    task := trySteal()
    if task != null: return task
  }



}
```

SMQ: Stealing Multi-Queue



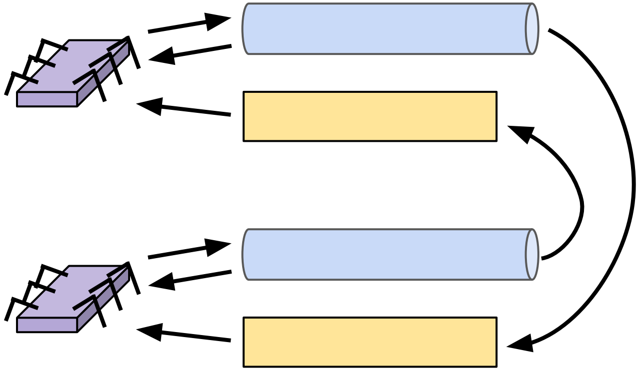
```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE$_{steal}$- 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()
  with ρ$_{steal}$ probability {
    task := trySteal()
    if task != null: return task
  }
  task := queues[curThread()].extractTopLocal()
  if task != null: return task

}
```

SMQ: Stealing Multi-Queue



```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE_steal - 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()
  with ρ_steal probability {
    task := trySteal()
    if task != null: return task
  }
  task := queues[curThread()].extractTopLocal()
  if task != null: return task
  return trySteal()
}
```

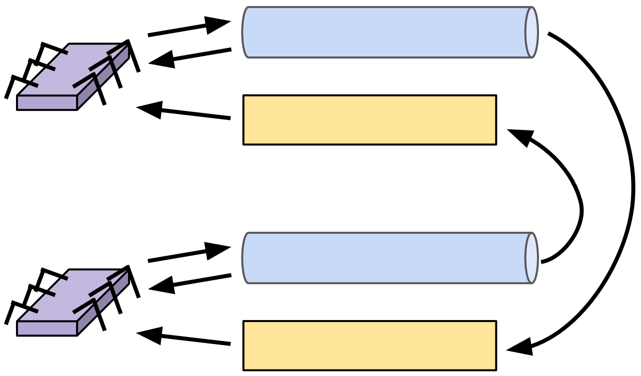
SMQ: Stealing Multi-Queue



```
fun trySteal(): E? {
  i := random(0, T)
  t := curThread()
  if queues[i].top() < queues[t].top() {
    stolen := queues[i].steal($SIZE_{steal}$)
    if stolen.isEmpty(): return null
    stolenTasks.add(stolen[1:])
    return stolen[0]
  }
}
```

```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>($SIZE_{steal}$- 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()
  with $\rho_{steal}$ probability {
    task := trySteal()
    if task != null: return task
  }
  task := queues[curThread()].extractTopLocal()
  if task != null: return task
  return trySteal()
}
```

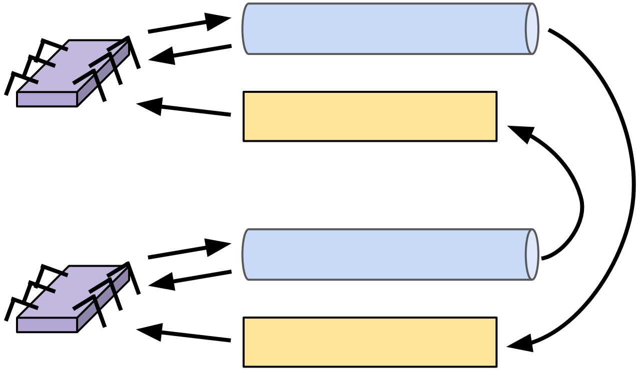
SMQ: Stealing Multi-Queue



```
fun trySteal(): E? {
  i := random(0, T)
  t := curThread()
  if queues[i].top() < queues[t].top() {
    stolen := queues[i].steal($SIZE_{steal}$)
    if stolen.isEmpty(): return null
    stolenTasks.add(stolen[1:])
    return stolen[0]
  }
}
```

```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>($SIZE_{steal}$- 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()
  with $\rho_{steal}$ probability {
    task := trySteal()
    if task != null: return task
  }
  task := queues[curThread()].extractTopLocal()
  if task != null: return task
  return trySteal()
}
```

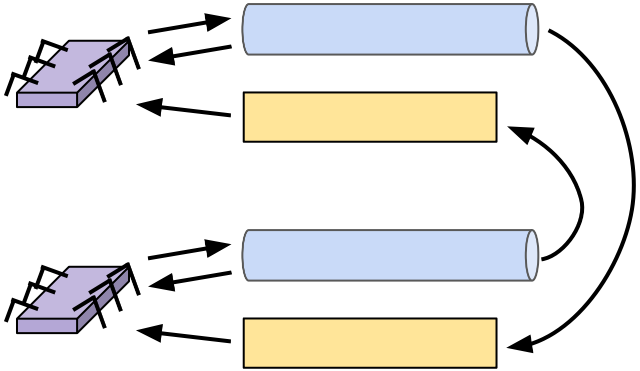
SMQ: Stealing Multi-Queue



```
fun trySteal(): E? {
  i := random(0, T)
  t := curThread()
  if queues[i].top() < queues[t].top() {
    stolen := queues[i].steal($SIZE_{steal}$)
    if stolen.isEmpty(): return null
    stolenTasks.add(stolen[1:])
    return stolen[0]
  }
}
```

```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>($SIZE_{steal}$ - 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()
  with $\rho_{steal}$ probability {
    task := trySteal()
    if task != null: return task
  }
  task := queues[curThread()].extractTopLocal()
  if task != null: return task
  return trySteal()
}
```

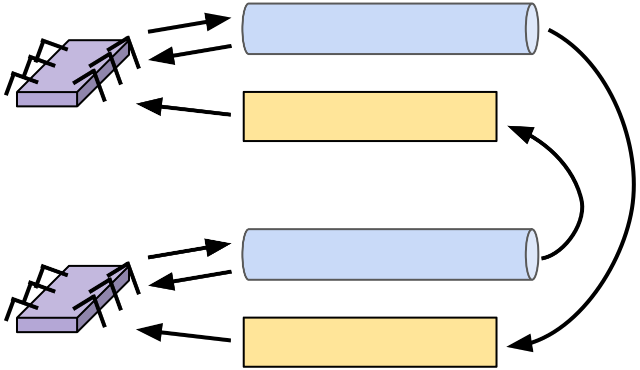
SMQ: Stealing Multi-Queue



```
fun trySteal(): E? {
  i := random(0, T)
  t := curThread()
  if queues[i].top() < queues[t].top() {
    stolen := queues[i].steal(SIZE_steal)
    if stolen.isEmpty(): return null
    stolenTasks.add(stolen[1:])
    return stolen[0]
  }
}
```

```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE_steal - 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()
  with ρ_steal probability {
    task := trySteal()
    if task != null: return task
  }
  task := queues[curThread()].extractTopLocal()
  if task != null: return task
  return trySteal()
}
```

SMQ: Stealing Multi-Queue



```
fun trySteal(): E? {
  i := random(0, T)
  t := curThread()
  if queues[i].top() < queues[t].top() {
    stolen := queues[i].steal(SIZE_steal)
    if stolen.isEmpty(): return null
    stolenTasks.add(stolen[1:])
    return stolen[0]
  }
}
```

```
val queues := SequentialPriorityQueue<E>[T]
val threadlocal stolenTasks := Buffer<E>(SIZE_steal- 1)

fun insert(task: E) {
  queues[curThread()].addLocal(task)
}

fun delete(): E? {
  if stolenTasks.isNotEmpty():
    return stolenTasks.removeFirst()
  with ρ_steal probability {
    task := trySteal()
    if task != null: return task
  }
  task := queues[curThread()].extractTopLocal()
  if task != null: return task
  return trySteal()
}
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E)

fun extractTopLocal(): E?

fun top(): E?

fun steal(size: Int): List<E>
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E?

fun top(): E?

fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E?

fun top(): E?

fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E?

fun top(): E?

fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E?

fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E>

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE$_{steal}$)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer()
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer() {
  stealingBuffer.clear()
  fillStealingBufferUnsafe()
  (epoch, stolen) = (epoch + 1, false)
}
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer() {
  stealingBuffer.clear()
  fillStealingBufferUnsafe()
  (epoch, stolen) = (epoch + 1, false)
}
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer() {
  stealingBuffer.clear()
  fillStealingBufferUnsafe()
  (epoch, stolen) = (epoch + 1, false)
}
```

SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer() {
  stealingBuffer.clear()
  fillStealingBufferUnsafe()
  (epoch, stolen) = (epoch + 1, false)
}
```

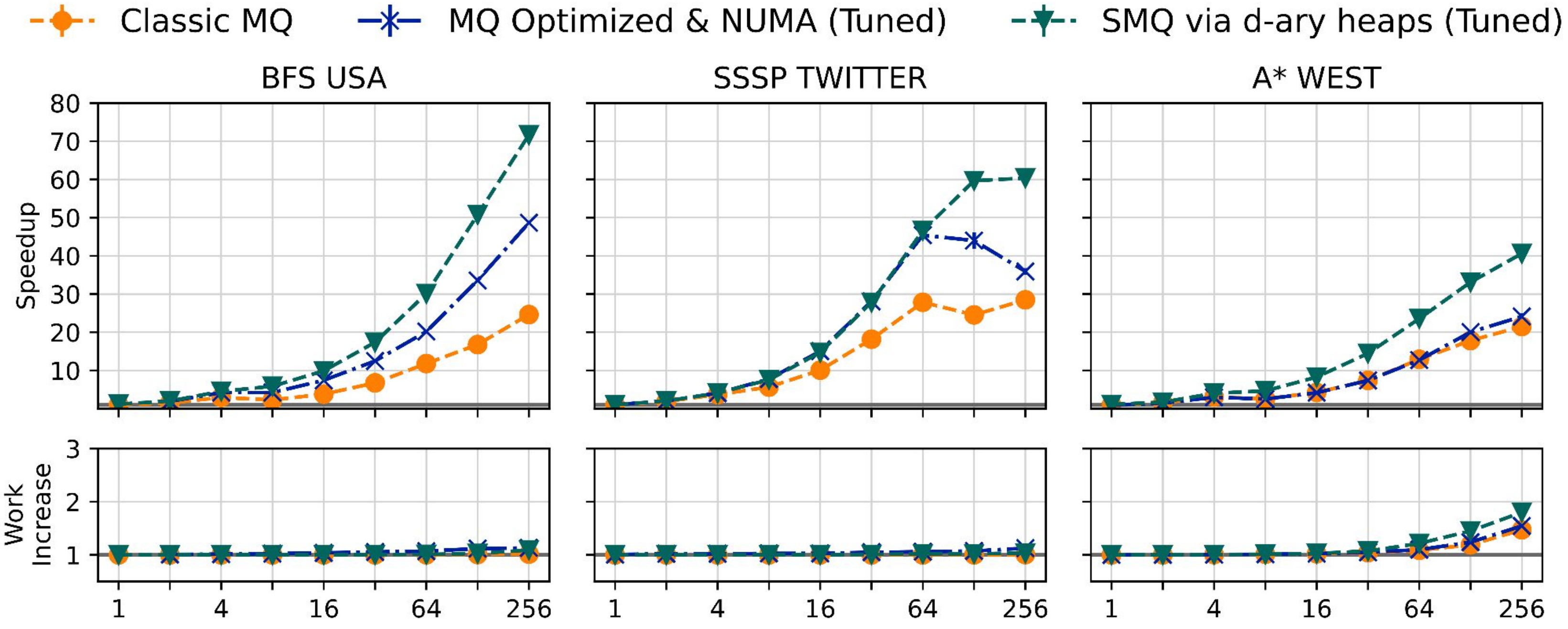
SMQ via d-Ary Heaps with Stealing Buffers

```
val q := d-AryHeap<E>()
val stealingBuffer := SequentialBuffer<E>(SIZE_steal)
val (epoch, stolen): (Int, Bool) = (0, false)
```

```
fun addLocal(task: E) {
  q.add(task)
  if stolen: fillBuffer()
}

fun extractTopLocal(): E? {
  return q.extractTop()
}

fun top(): E? = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return null
  top := stealingBuffer.first() // UNSAFE
  if curEpoch != epoch: continue
  return top
}
```

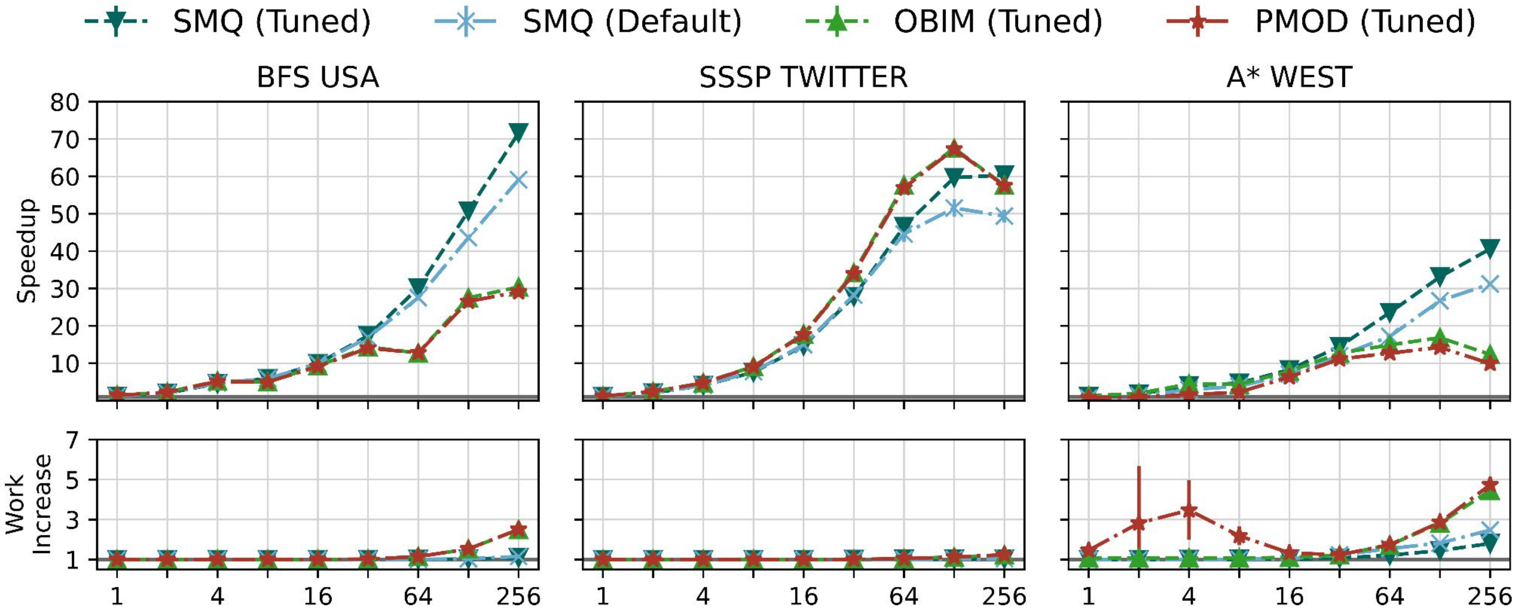
```
fun steal(size: Int): List<E> = while(true) {
  (curEpoch, curStolen) := (epoch, stolen)
  if stolen: return emptyList()
  tasks := stealingBuffer.read() // UNSAFE
  if (epoch, stolen).CAS({curEpoch, false},
                         {curEpoch, true}) {
    return tasks
  }
}

fun fillBuffer() {
  stealingBuffer.clear()
  fillStealingBufferUnsafe()
  (epoch, stolen) = (epoch + 1, false)
}
```

MQ vs MQ-Optimized vs SMQ



SMQ is faster and more scalable!

SMQ vs OBIM vs PMOD



SMQ either outperforms the state-of-the-art or shows competitive performance

SMQ Fairness Guarantees

The same guarantees as for the MQ-Optimized

Therefore, the same in principle as for the classic MQ

Conclusions

- Multi-Queues can be practical
  - *task batching* + *temporal locality* + *NUMA* optimizations

- We suggested a novel ***Stealing Multi-Queue*** algorithm, which outperforms the state-of-the-art in many real-world scenarios

- Both the MQ-Optimized and the SMQ algorithms provide theoretical fairness guarantees

Thank you!