SNUG: Architectural Support for Relaxed Concurrent Priority Queueing in Chip Multiprocessors

Azin Heidarshenas*, Tanmay Gangwani*, Serif Yesil, Adam Morrison, and Josep Torrellas

* Co-first authorship

International Conference on Supercomputing 2020
Priority-based Task Scheduling

Tasks are executed based on their *priority order*

If $T_1$ has higher priority than $T_2$ it gets executed before $T_2$
Priority-based Task Scheduling

Tasks are executed based on their *priority order*
If $T_1$ has higher priority than $T_2$ it gets executed before $T_2$

Example: Dijkstra’s Single-Source Shortest Path (SSSP)
• Source vertex A

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

A, 0  C, 1  B, 2  B, 3
Concurrent Priority Queue (PQ)

Parallel threads \textit{dequeue} (pop) tasks and \textit{enqueue} (push) new ones.

\begin{itemize}
  \item Head ptr $\rightarrow$ highest priority
  \item 1, 2, 4, 7
\end{itemize}
Concurrent Priority Queue (PQ)

Parallel threads **dequeue** (pop) tasks and **enqueue** (push) new ones

Head ptr → highest priority

Core 1

Core 2

Core 3

Enqueue

5

1 2 4 7
Concurrent Priority Queue (PQ)

Parallel threads **dequeue** (pop) tasks and **enqueue** (push) new ones
Concurrent Priority Queue (PQ)

Parallel threads *enqueue* (push) new ones and *dequeue* (pop) tasks.

Head ptr $\rightarrow$ highest priority

Core 1 → Enqueue → 1 2 4 5 7
Concurrent Priority Queue (PQ)

Parallel threads **dequeue** (pop) tasks and **enqueue** (push) new ones

Head ptr $\rightarrow$ highest priority

Core 1

Core 2

Core 3
Concurrent Priority Queue (PQ)

Parallel threads **dequeue** (pop) tasks and **enqueue** (push) new ones

Head ptr $\rightarrow$ highest priority

Core 1
Core 2
Core 3

Enqueue
Dequeue
Dequeue

1 2 4 5 7

Compare\&Swap(&head_ptr, ...)

Core 1
Core 2
Core 3
Concurrent Priority Queue (PQ)

Parallel threads **dequeue** (pop) tasks and **enqueue** (push) new ones.

Head ptr → highest priority

Compare&Swap(&head_ptr, ...)

Not scalable due to synchronization bottleneck
Relaxed PQ

Relaxed Priority $\rightarrow$ alleviate synchronization at the dequeues

Core 1

Core 2

Core 3

1 2 4 7

Head ptr $\rightarrow$ highest priority

In many applications, relaxed priorities don’t harm correctness

Graph applications, discrete event simulation, ...
Relaxed PQ

Relaxed Priority $\rightarrow$ alleviate synchronization at the dequeues

In many applications, relaxed priorities don’t harm correctness

Graph applications, discrete event simulation, ...
Relaxed PQ

Relaxed Priority $\rightarrow$ alleviate synchronization at the deques

In many applications, relaxed priorities don’t harm correctness

Graph applications, discrete event simulation, ...
Relaxed PQ: Synch. vs Wasted Work

Relaxed PQ doesn’t enforce ordered execution of tasks
Lower priority task can be overwritten by a higher priority one

Example: Single-Source Shortest Path (SSSP)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
</tr>
</tbody>
</table>

![Graph](image)
Relaxed PQ: Synch. vs Wasted Work

Relaxed PQ doesn’t enforce ordered execution of tasks
Lower priority task can be overwritten by a higher priority one

Example: Single-Source Shortest Path (SSSP)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
</tr>
</tbody>
</table>

A, 0
Relaxed PQ: Synch. vs Wasted Work

Relaxed PQ doesn’t enforce ordered execution of tasks
Lower priority task can be overwritten by a higher priority one

Example: Single-Source Shortest Path (SSSP)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>$\infty$</td>
</tr>
<tr>
<td>C</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

A, 0  C, 1  B, 3
Relaxed PQ: Synch. vs Wasted Work

Relaxed PQ doesn’t enforce ordered execution of tasks
Lower priority task can be overwritten by a higher priority one

Example: Single-Source Shortest Path (SSSP)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

Vertex | Distance
--- | ---
A | 0
B | ∞
C | 1
Relaxed PQ: Synch. vs Wasted Work

Relaxed PQ doesn’t enforce ordered execution of tasks
Lower priority task can be overwritten by a higher priority one

Example: Single-Source Shortest Path (SSSP)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

Vertex Distance

A, 0
C, 1
B, 2
B, 3
Relaxed PQ: Synch. vs Wasted Work

Relaxed PQ doesn’t enforce ordered execution of tasks
Lower priority task can be overwritten by a higher priority one

Example: Single-Source Shortest Path (SSSP)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>3, 2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

A, 0  C, 1  B, 2  B, 3
Overview of SNUG

Wasted work vs. synchronization trade-off
Local enqueues, relaxed global dequeues

Core 1 → Enqueue → 1 3 4 7
Core 2 → Enqueue → 4 5 8 10
Core 3 → Enqueue → 2 3 4 6

Work Registers
Pointer
Priority
0x...

Software
Hardware
SNUG Programming Model

Programmer

Application

Enqueue

Dequeue

SNUG Library

New Instructions

AllocHeads

UpdateHead

FetchHead

PickHead

SNUG Architecture

New Instructions

Work Registers

One logical queue

Multiple physical queues, distributed

0x...

0x...

0x...
Each core has access to a set of work registers and a PickHead module.
PickHead Instruction
PickHead Instruction

Gather ptrs to queue heads + priorities
Gather ptrs to queue heads + priorities
PickHead Instruction

Gather ptrs to queue heads + priorities
Gather ptrs to queue heads + priorities
PickHead Instruction

Gather ptrs to queue heads + priorities

PickHead Module

Snapshot Memory
ptr, 7
ptr, 10
ptr, 15
PickHead Instruction

Gather ptrs to queue heads + priorities
PickHead Instruction

PickHead Module

Snapshot Memory
ptr, 4
ptr, 7
ptr, 10
ptr, 15

Gather ptrs to queue heads + priorities
PickHead Instruction

PickHead Module

Snapshot Memory
ptr, 1
ptr, 2
ptr, 3
ptr, 4
ptr, 5
...
ptr, 24

Sorting heads of physical queues
PickHead Instruction

PickHead Module

Snapshot Memory

ptr, 1
ptr, 2
ptr, 3
ptr, 4
ptr, 5
...
ptr, 24

Relaxation Count (R)

PM
0x... 10
PM
0x... 3
PM
0x... 5
PM
0x... 15
PM
0x... 24
PM
0x... 12
PM
0x... 1
PM
0x... 8
PM
0x... 20
PM
0x... 17
PM
0x... 9
PM
0x... 2
PM
0x... 7
PM
0x... 6
PM
0x... 21
PM
0x... 4

Sorting heads of physical queues
PickHead Instruction

```
PM 0x10
PM 0x03
PM 0x05
PM 0x15
PM 0x24
PM 0x12
PM 0x01
PM 0x08
PM 0x20
PM 0x17
PM 0x09
PM 0x02
PM 0x07
PM 0x06
PM 0x21
PM 0x04
```

PickHead Module

Snapshot Memory

```
ptr, 1
ptr, 2
ptr, 3
ptr, 4
ptr, 5
...
ptr, 24
```

Relaxation Count (R)

Return one priority at random
PickHead Instruction

Then, SW performs a CAS
Modes of PickHead Instruction

**Global Access**: gathers the heads of all queues

**No Network Access**: reuses the snapshot before it gets stale

**Local Access**: uses the head of the local queue
Modes of PickHead Instruction
Modes of PickHead Instruction

PickHead Module

ptr, 1
ptr, 2
ptr, 3
ptr, 4
ptr, 5
...
ptr, 24

Global Access

Relaxation Count (R)
Modes of PickHead Instruction

PickHead Module

- ptr, 1
- ptr, 2
- ptr, 3
- ptr, 4
- ptr, 5
- ...
- ptr, 24

Global Access

Relaxation Count (R)
Modes of PickHead Instruction

PickHead Module

- ptr, 1
- ptr, 2
- ptr, 3
- ptr, 4
- ptr, 5
- ...
- ptr, 24

Relaxation Count (R)

Global Access
No Network Access
No Network Access
No Network Access

U times
Modes of PickHead Instruction

PickHead Module

- ptr, 1
- ptr, 2
- ptr, 3
- ptr, 4
- ptr, 5
- ... ptr, 24

Relaxation Count (R)

- Global Access
- No Network Access
- Local Access

- No Network Access
- Local Access

- Local Access

U times

- No Network Access

L times

...
Adaptivity of SNUG to CAS Failures

SW performs a CAS on the chosen queue. It can fail.

If failure: retry the same queue

SNUG adapts to contention

➢ If frequent CAS failures \( \rightarrow \) increase \( R \)
➢ If rare CAS failures \( \rightarrow \) decrease \( R \)

Relaxation Count (R)

\[
\begin{array}{lllll}
\text{ptr, 1} & \text{ptr, 2} & \text{ptr, 3} & \text{ptr, 4} & \text{ptr, 5}
\end{array}
\]
PickHead Module

Selected Head

- Snapshot Memory
- Sorter & Selector
- Tag Counter
- Work Registers

Tag

R Reg

DEMA

Decision

PickHead Instruction

Reuse Snapshot

Local/Global Dequeue

Request Generator
Evaluation Setup

64-core simulations in Gem5, U = 4 and L = 4
Applications: SSSP, BFS, SIMUL, A*

PQs evaluated:
  • SW-SK: Concurrent skiplist\(^1\) implementation (baseline)
    • Always dequeues the highest priority task
  • SW-SP: Concurrent spraylist\(^2\)
    • Dequeues are sprayed over a range of high priority tasks
  • SW-D: Distributed concurrent skiplist with work-stealing
    • Local enqueues, local dequeues
  • HW-C: Centralized version of SNU. No PickHead
  • HW-D: SNU
    • Local enqueues, global dequeues

Push and pop contention in SW-SK and SW-SP. Wasted work in SW-D

SNUG achieves 1.4x, 2.4x, and 3.6x speedup over SW-SK, SW-SP and SW-D
SW-D suffers from wasted work due to local dequeues
Adaptation of the Relaxation Count

R is able to adapt to the CAS contentions
More in the Paper

- Analysis of Network Traffic
- SNUG Scalability
- Sensitivity Analysis of SNUG Parameters
- Characterizing R Adaptivity
- Power and Area Analysis
- ...
Key Takeaways

- Relaxed PQ alleviates synchronization overhead but is prone to wasted work.

- SNUG distributes PQ in hardware and minimizes both wasted work and synchronization overhead simultaneously.

- SNUG’s relaxed PQ adapts to the rate of synchronization failures over time.

- For 64 cores: SNUG achieves avg. speedup of 1.4x, 2.4x, and 3.6x speedup over skip PQ, spray PQ, and distributed concurrent skiplist.
SNUG: Architectural Support for Relaxed Concurrent Priority Queueing in Chip Multiprocessors

Azin Heidarshenas*, Tanmay Gangwani*, Serif Yesil, Adam Morrison, and Josep Torrellas

* Co-first authorship