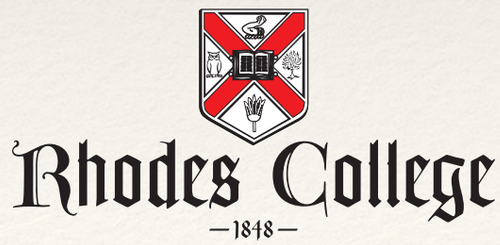
Optimizing Work Stealing Communication With Structured Atomic Operations

Hannah Cartier
University of Utah
u1361980@umail.utah.edu

James Dinan
Intel Corporation
jdinan@nvidia.com

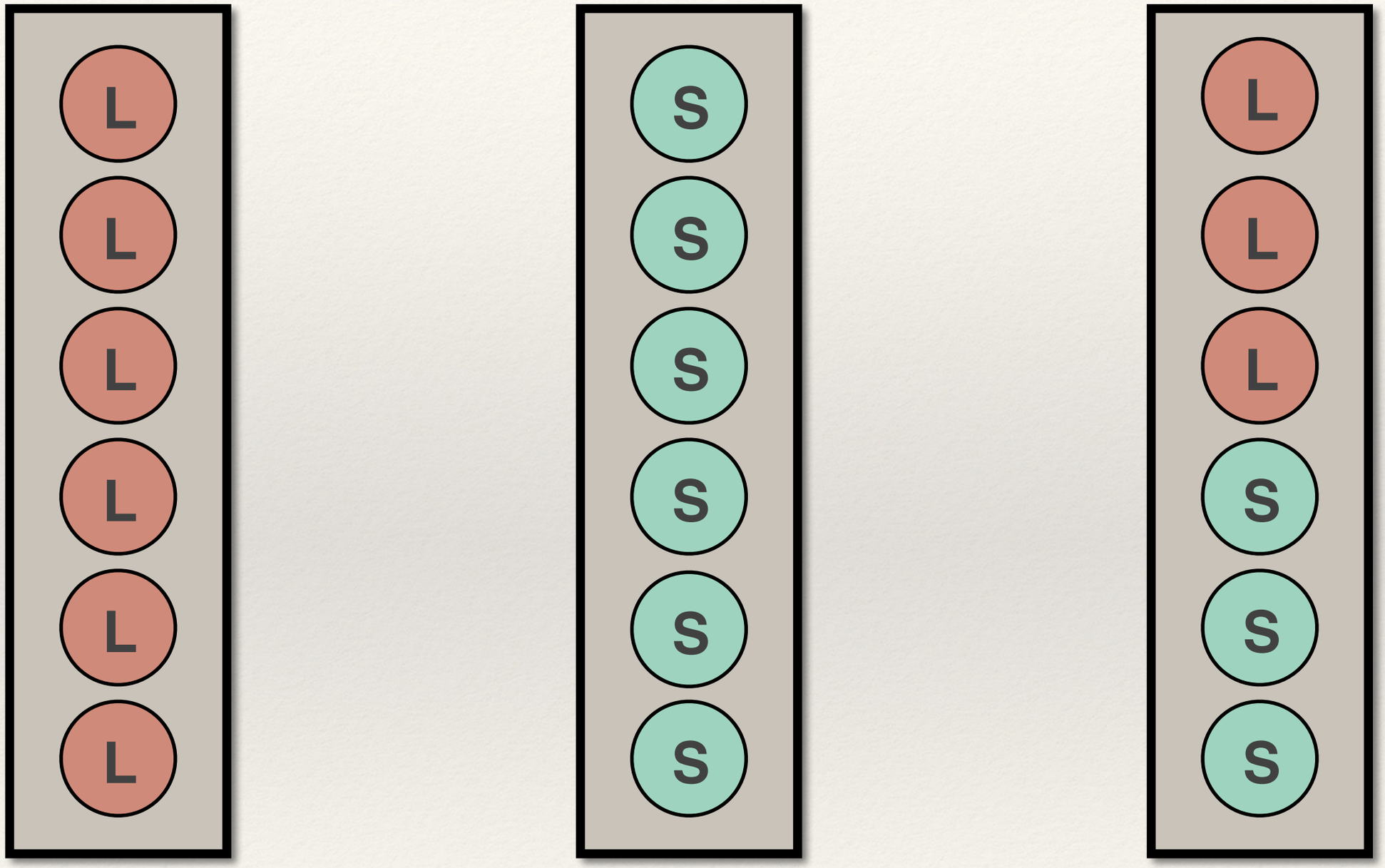
D. Brian Larkins
Rhodes College
larkinsb@rhodes.edu



Dynamic Load Balancing

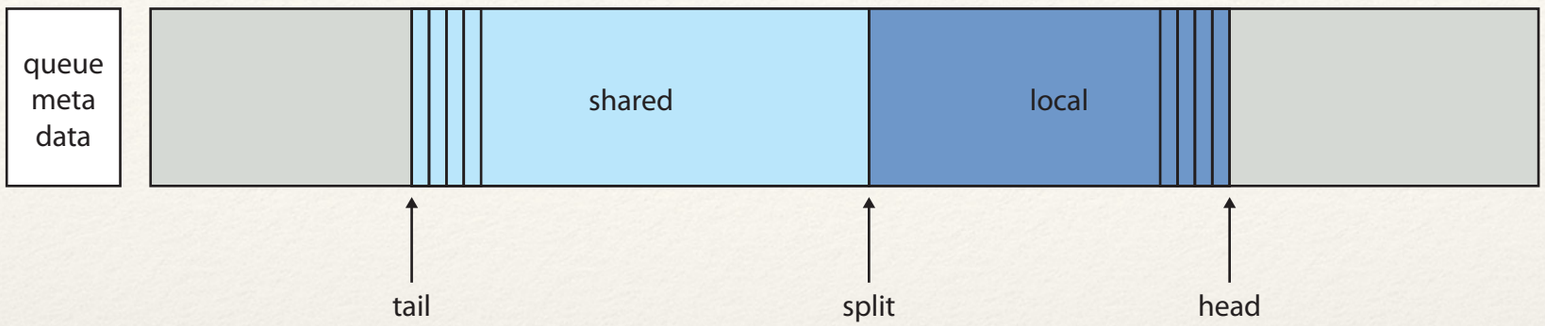
- More efficient than static load-balancers for applications with sparse of irregular data
- Based on prior work-stealing research using the Scioto load balancing framework [SC '08]
- Representation uses a PGAS ring buffer partitioned into local and shared portions.
- Local access to task queue is lock free, but work-stealing operation requires a sequence of one-sided communications

Work Stealing Example

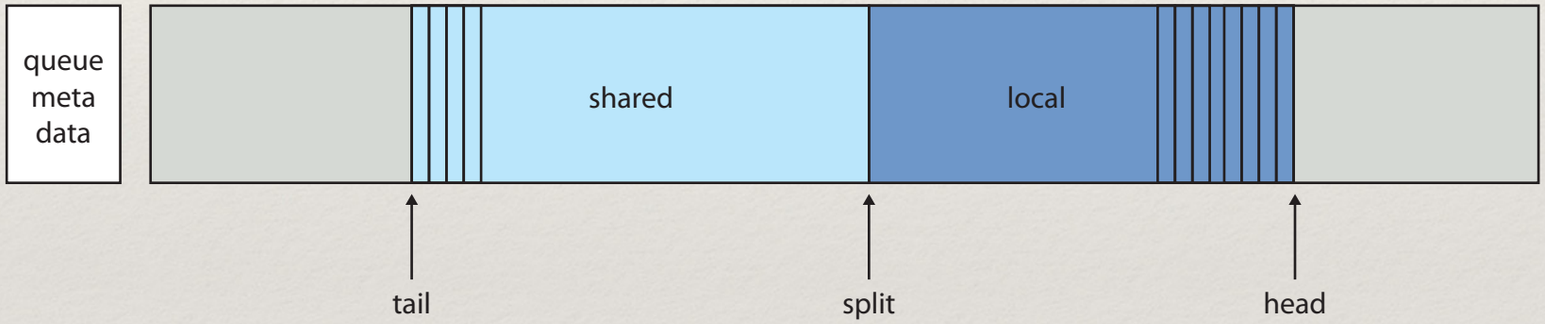


Process 0 Process 1 Process 3

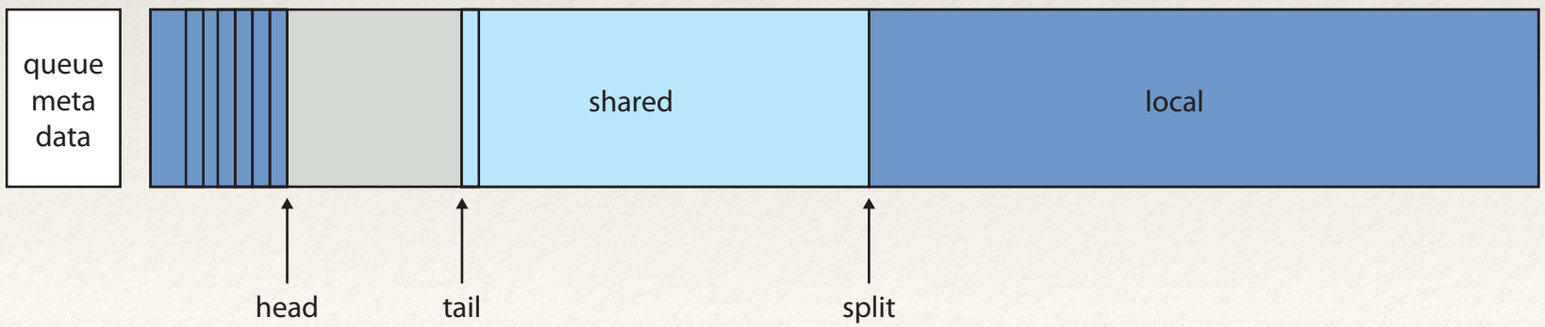
Task Queues - Adding Work



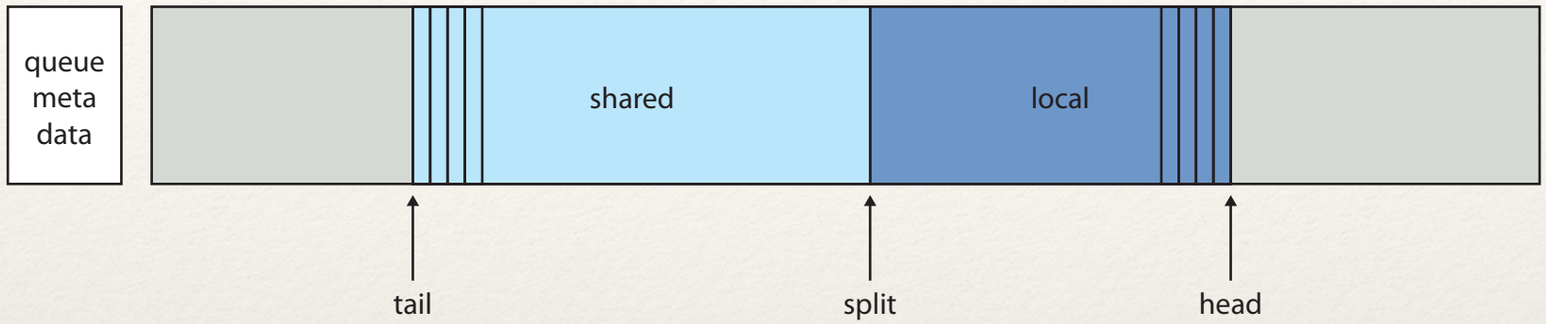
`push`



`push/wrap`

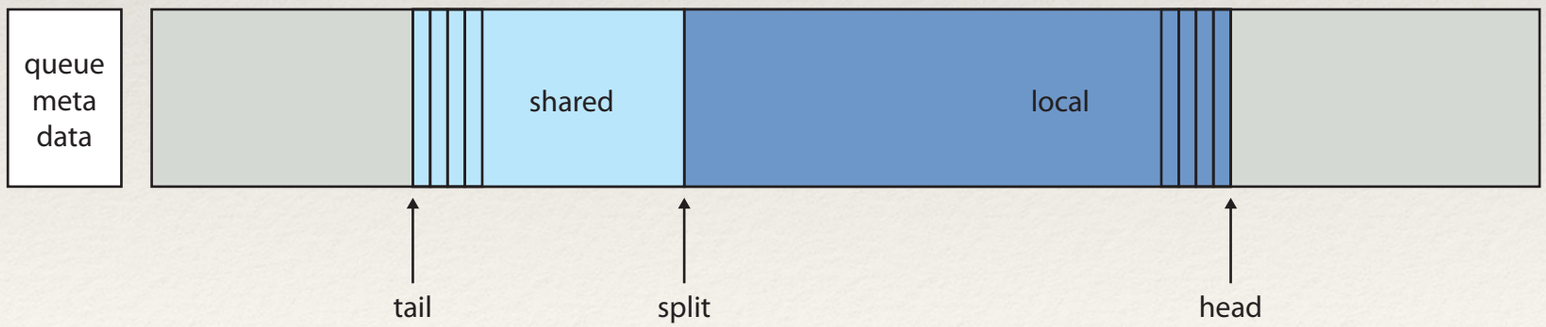


Task Queues - Split Management

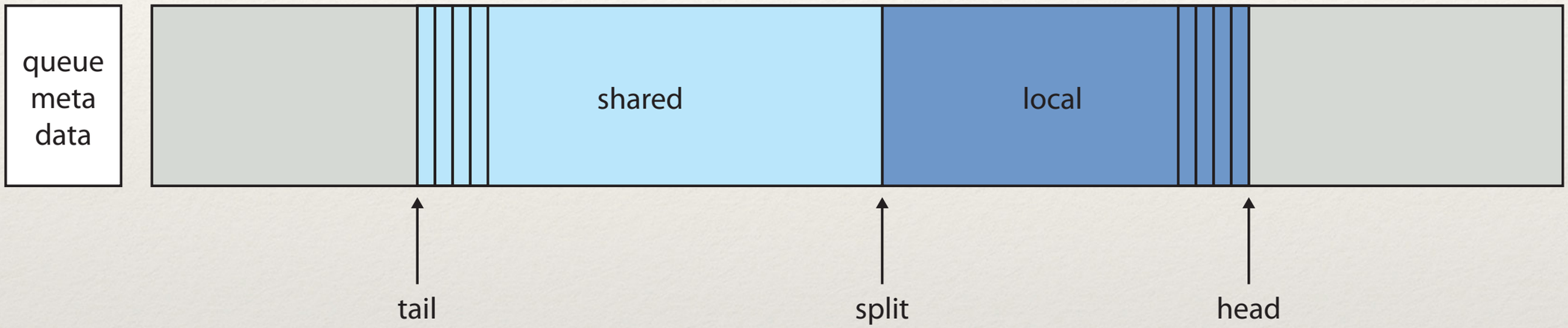


`release`

`acquire`



Baseline (SDC) Steal Operation



Stealing:

6 communications / 5 critical path

- `lock queue (atomic)` **Make sure nobody else can steal**
- `get queue metadata (get)` **How many tasks can I steal?**
- `update queue metadata (put)` **Claim the tasks**
- `unlock queue (atomic)` **Let others steal remaining tasks**
- `get tasks (get)` **Steal the tasks**
- `update queue steal progress (atomic/deferred)`

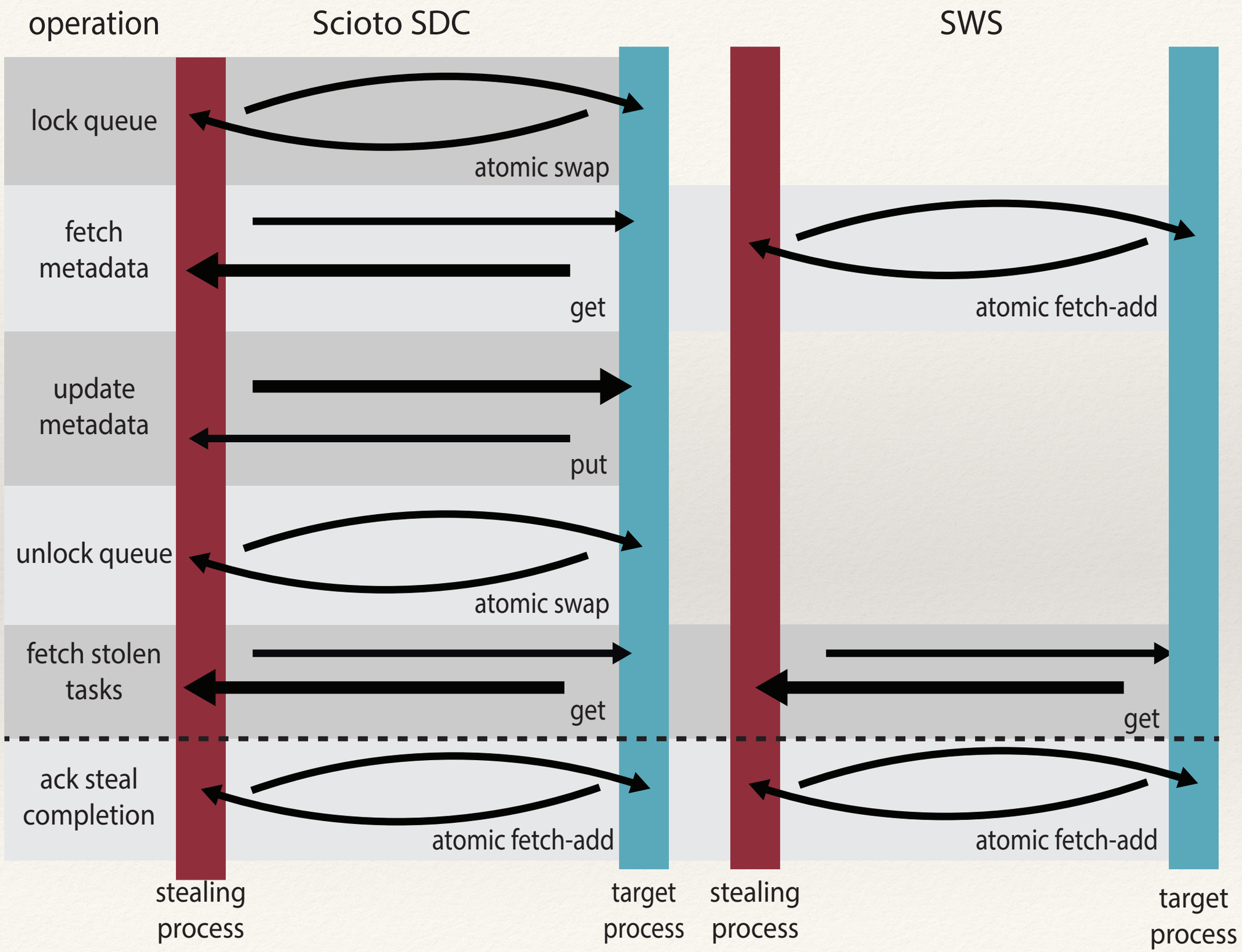
Insights and Contributions

- The information needed by the stealing process to discover and claim work can be represented by a compact data structure
- Work discovery and work stealing can be combined into one step
- A compact representation of key task queue metadata can be operated on with a single atomic communication operation
- the additional complexity of this representation adds minimal processing to queue metadata upkeep and maintenance.
- Greatest improvement with highly irregular workloads (short tasks + large # of spawned tasks)

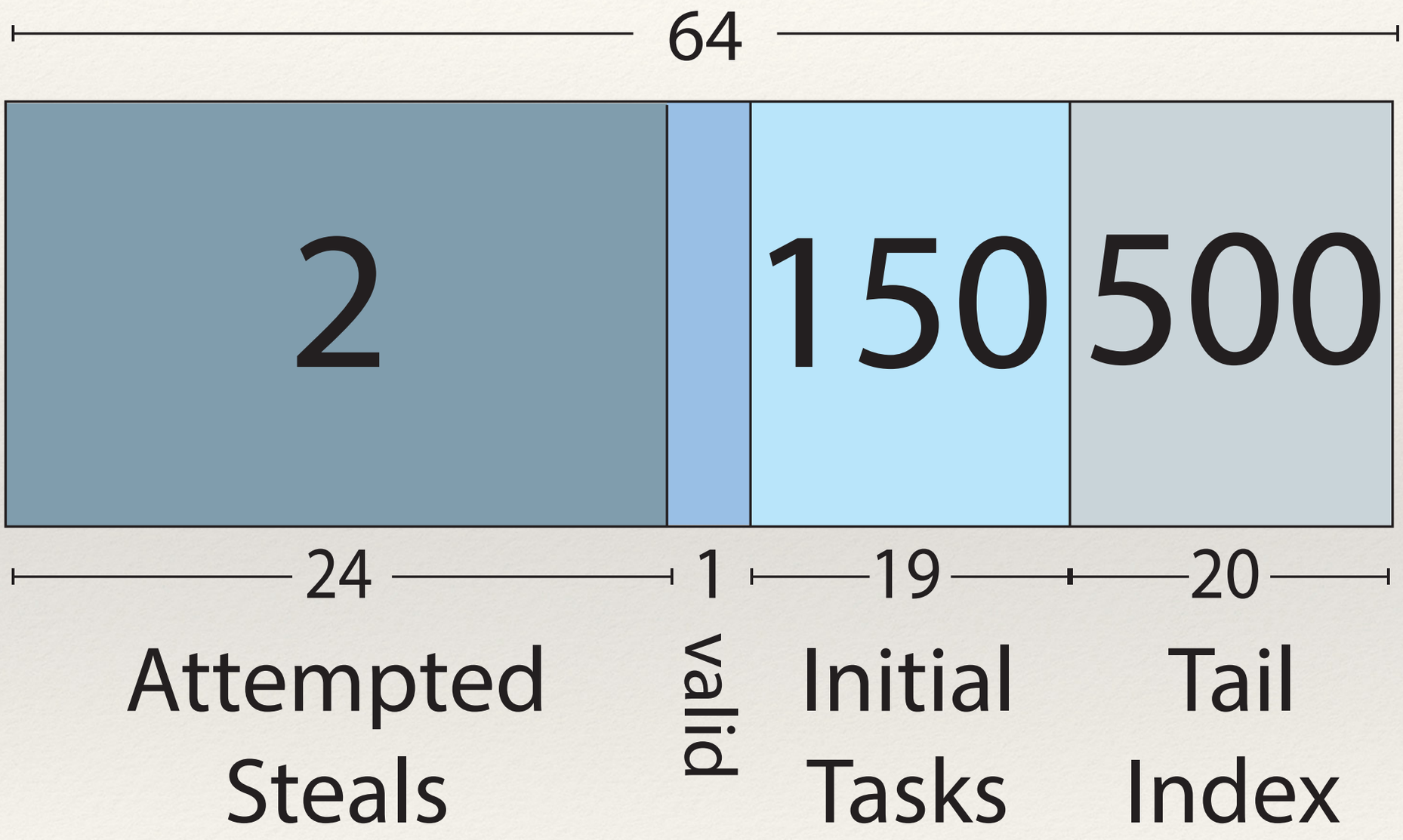
Structured Atomic Work Stealing (SWS)

- Use *steal-half* heuristic to set amount of tasks to be transferred per-steal. Established technique suited for our workloads
- Metadata for stealing tasks stored in a 64-bit integer (largest value that can be handled by atomic operations)
- Reduces number of remote communications from 6 to 3.
- Task states used to keep track of progress and determine safety of local queue operations

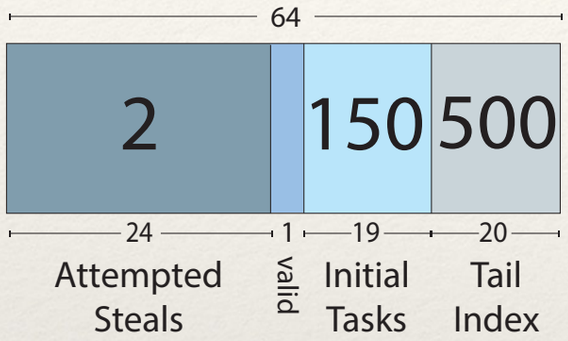
Steal Communication Operations

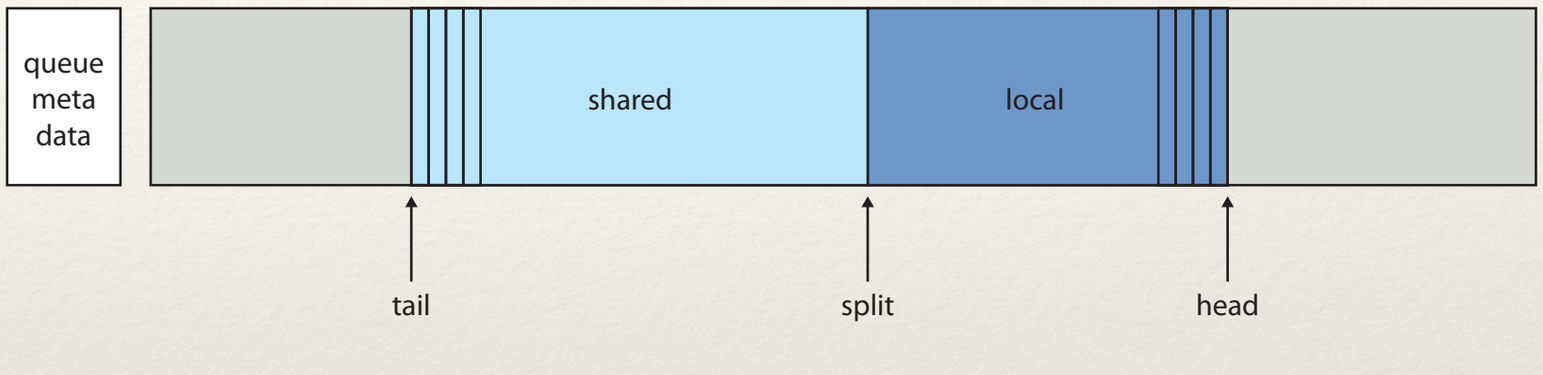


SWS Queue Data



SWS Steal Operation Example





Example:

1) 150 initial tasks gives us the following sequence of 9 steals: {75,37,19,9,5,2,1,1,1}

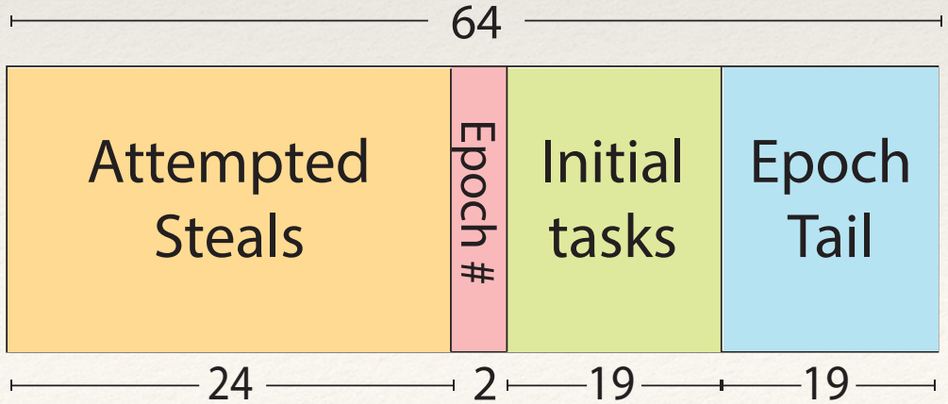
2) Initiator steals third block of tasks (19), beginning at the index at tail + previously claimed tasks

(500 + 75 + 37 = 612)

3) Tasks are copied and target's completion array is updated

Further Improvements

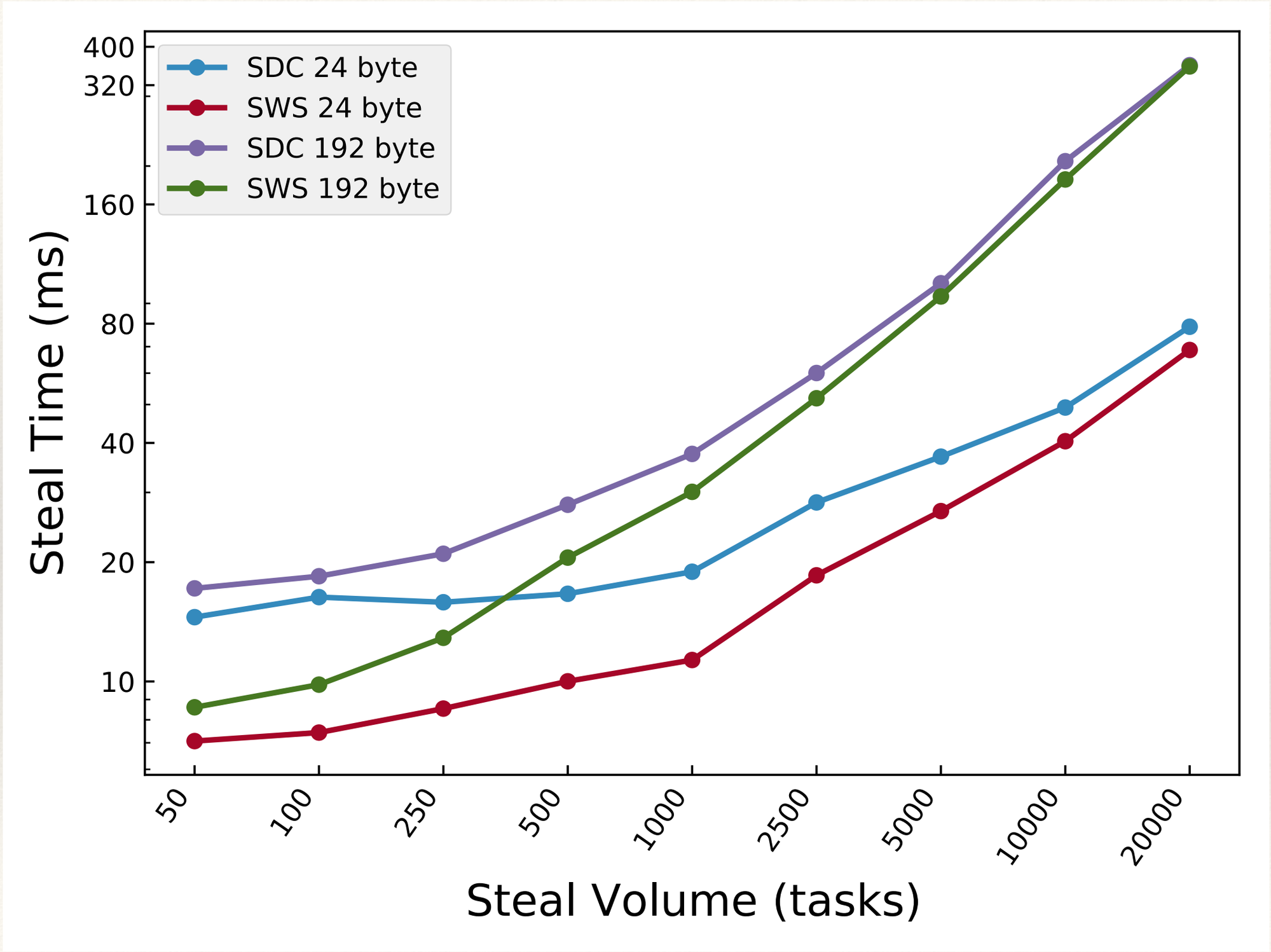
- Steal Dampening
  - As work becomes more sparse in the system steal attempts increase
  - When there are many steal attempts on a specific process the attempted-steals value can get very large
  - Steal-dampening ensures we avoid integer overflow which would mess up the queue metadata
- Completion Epochs
  - Steals can complete asynchronously which can hinder concurrency
  - Steal Epochs remedy this by introducing multiple completion arrays



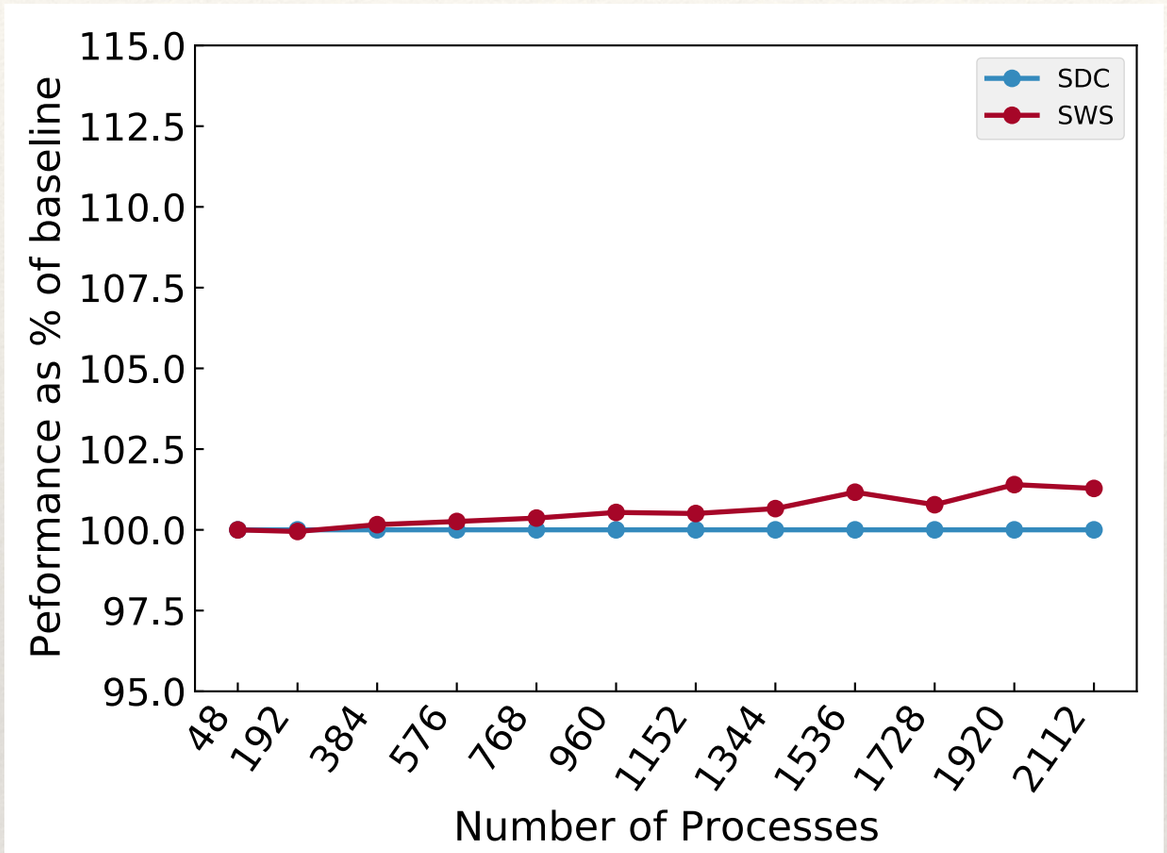
Experimental Evaluation

- Tested against previous Scioto implementation.
- Results show reduced latency for steal operations, lower overhead from the load-balancer due to less time spent searching for work, and improved responsiveness (throughput) at the target
- Experiments run on LOTUS compute cluster at Rhodes College
  - 44 compute nodes of 48 cores each. Each node is configured with two AMD EPYC 7352 24 core CPUs operating at 2.3GHz and 256 gigabytes of memory.
  - Mellanox EDR 100Gb/s Infini- Band fabric, using ConnectX-6 InfiniBand host channel adapters.
- Benchmarks:
  - BPC : Bouncing producer consumer
  - UTS : Unbalanced tree search.

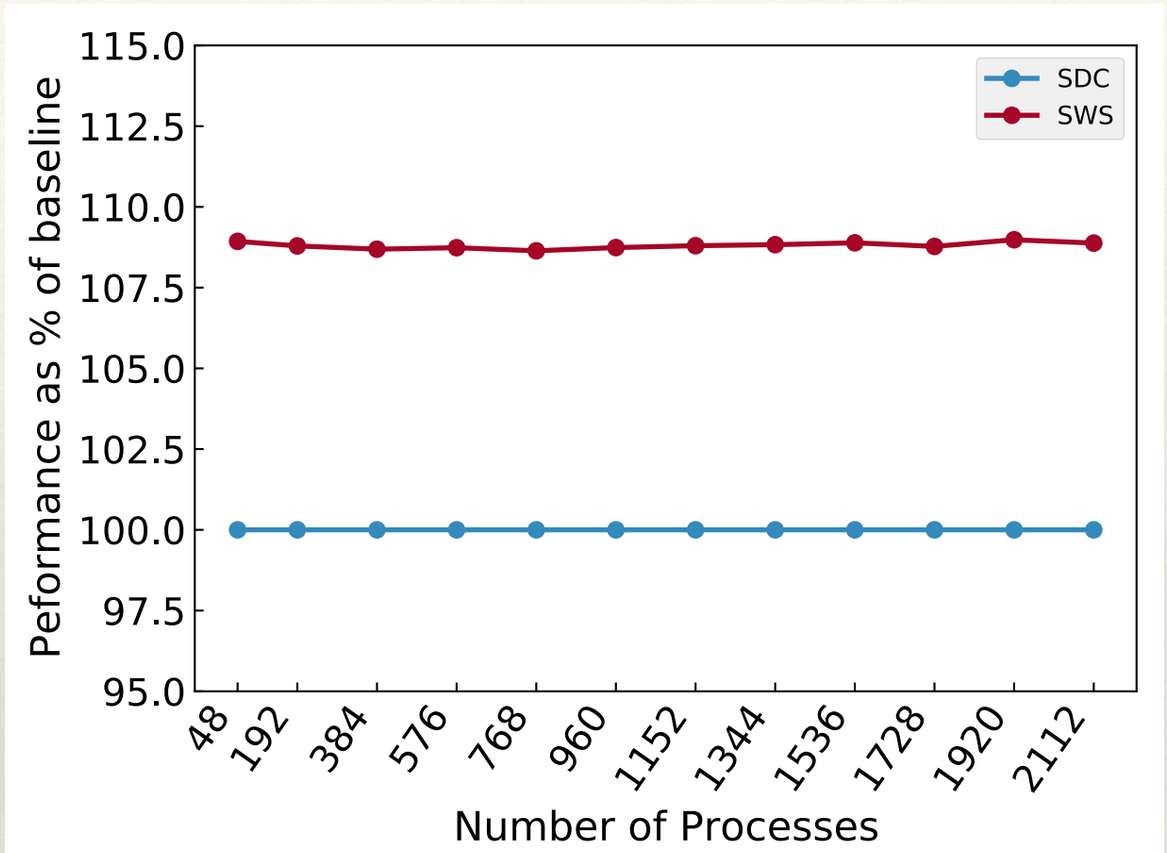
Remote Steal Performance



Relative Performance

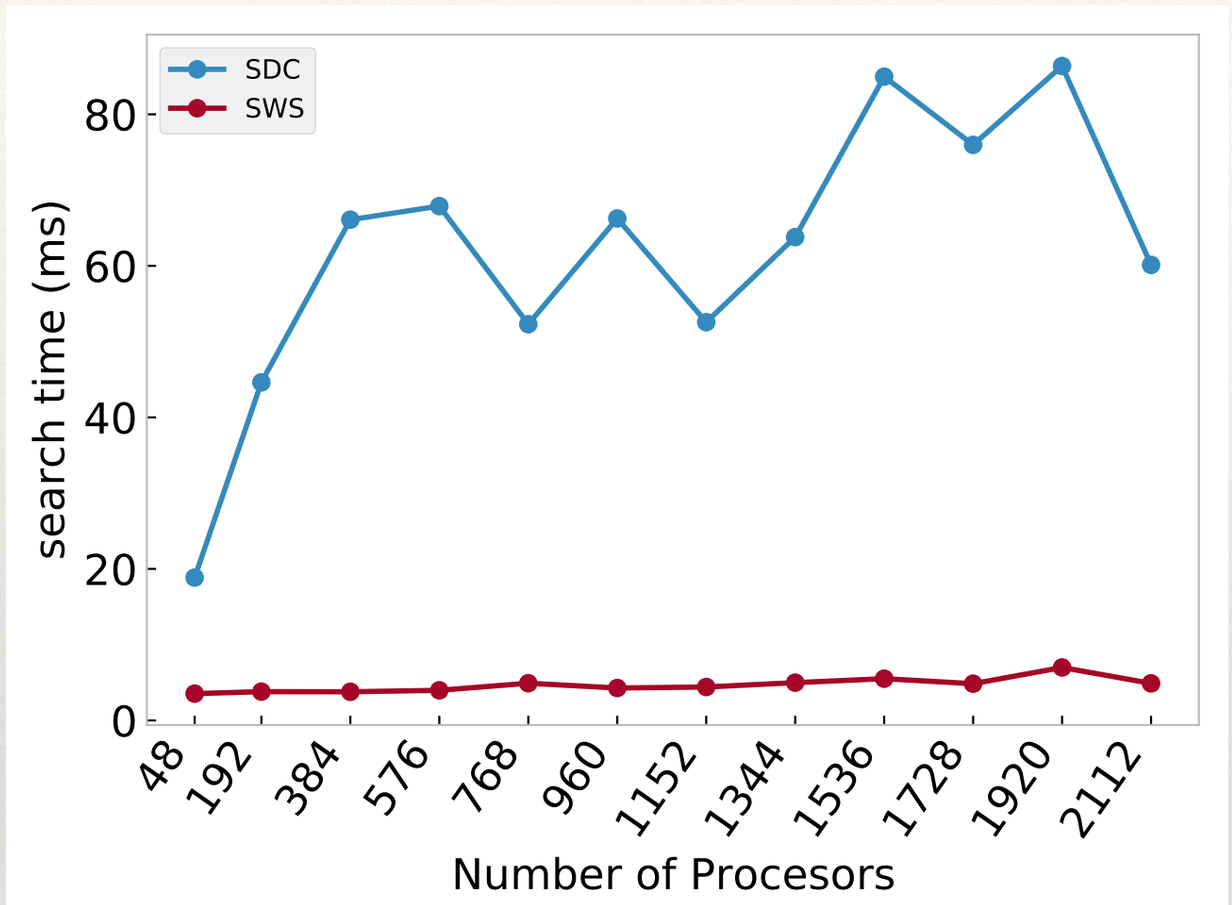


Relative BPC run-time performance improvement with SWS over baseline.

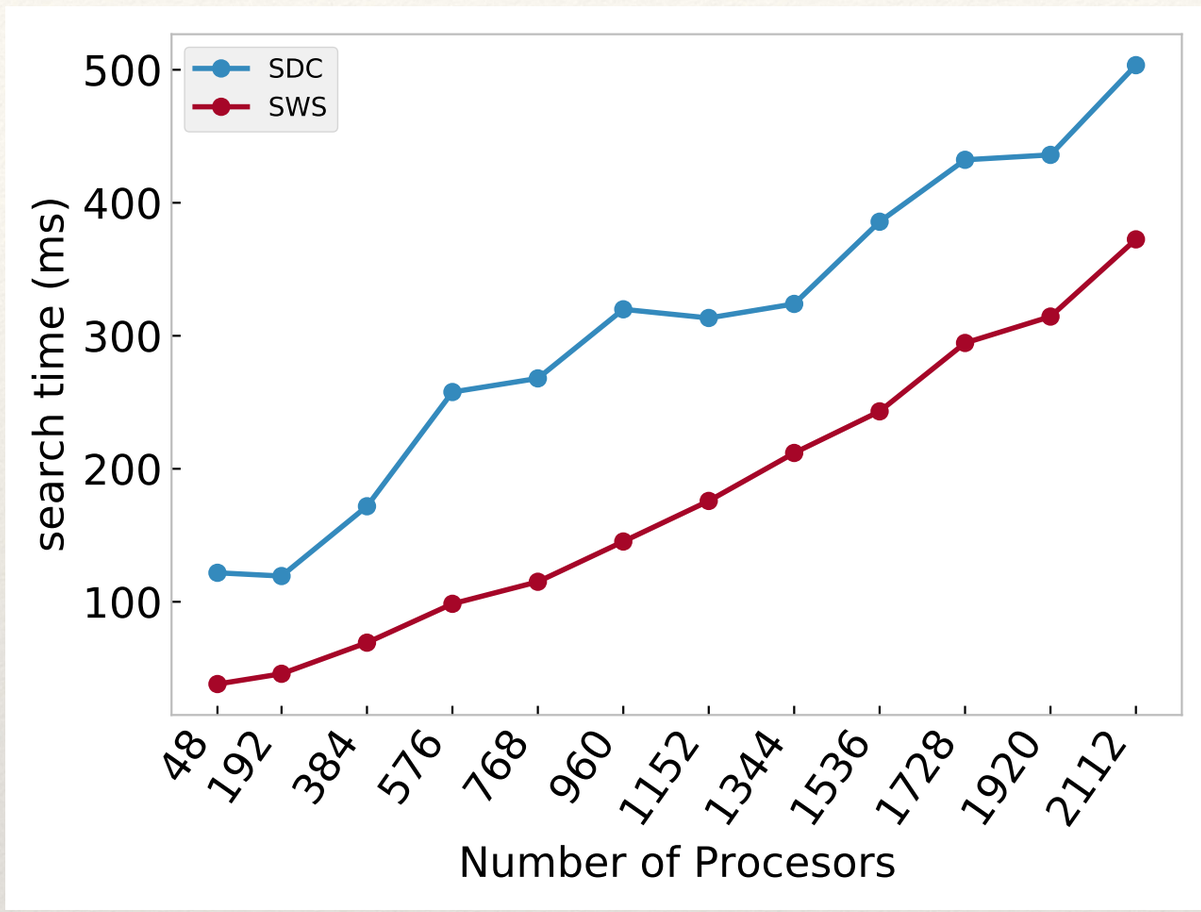


Relative UTS run-time performance improvement with SWS over baseline.

Search Time

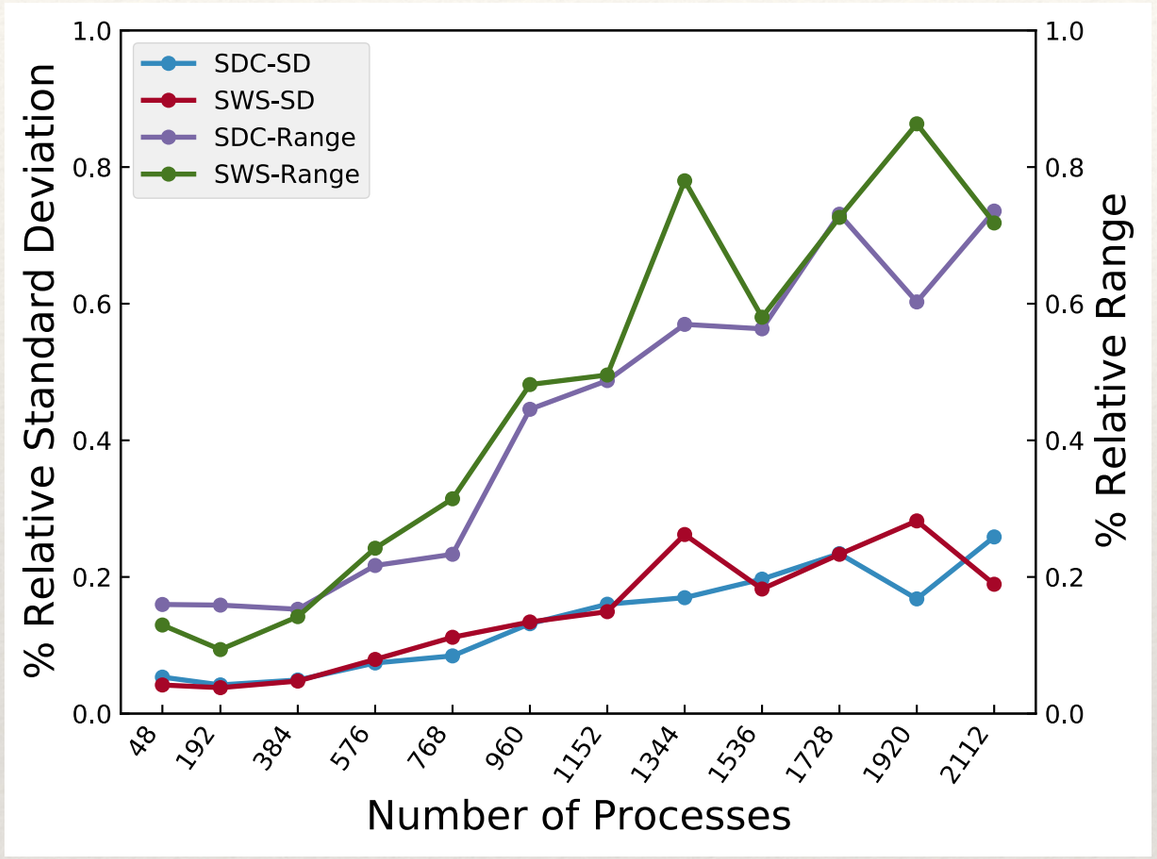


Average time spent searching for work
Per process (BPC)

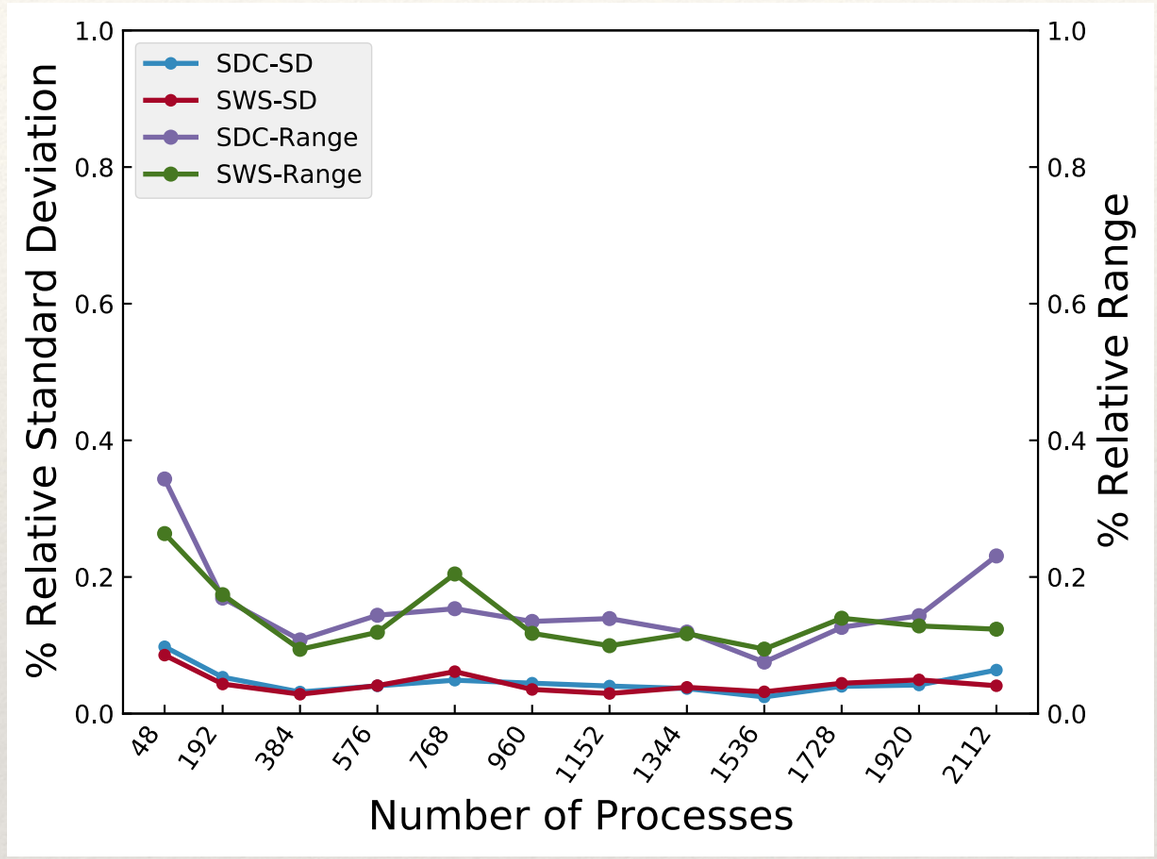


Average time spent searching for work
per process (UTS)

Variation Between Runs



Variation in 10 BPC runs



Variation in 10 UTS runs

Acknowledgements / Q&A

- Thanks to NSF/XSEDE and NVIDIA
- Funded by NSF grant number ACI-1053575 and NSF award 2018758.

- Contact: u1361980@umail.utah.edu
- LinkedIn: https://www.linkedin.com/in/hannah-cartier-ba97031a5/