MULTICORE PROGRAMMING

Harnessing Disorder

Lecture 5

Trevor Brown

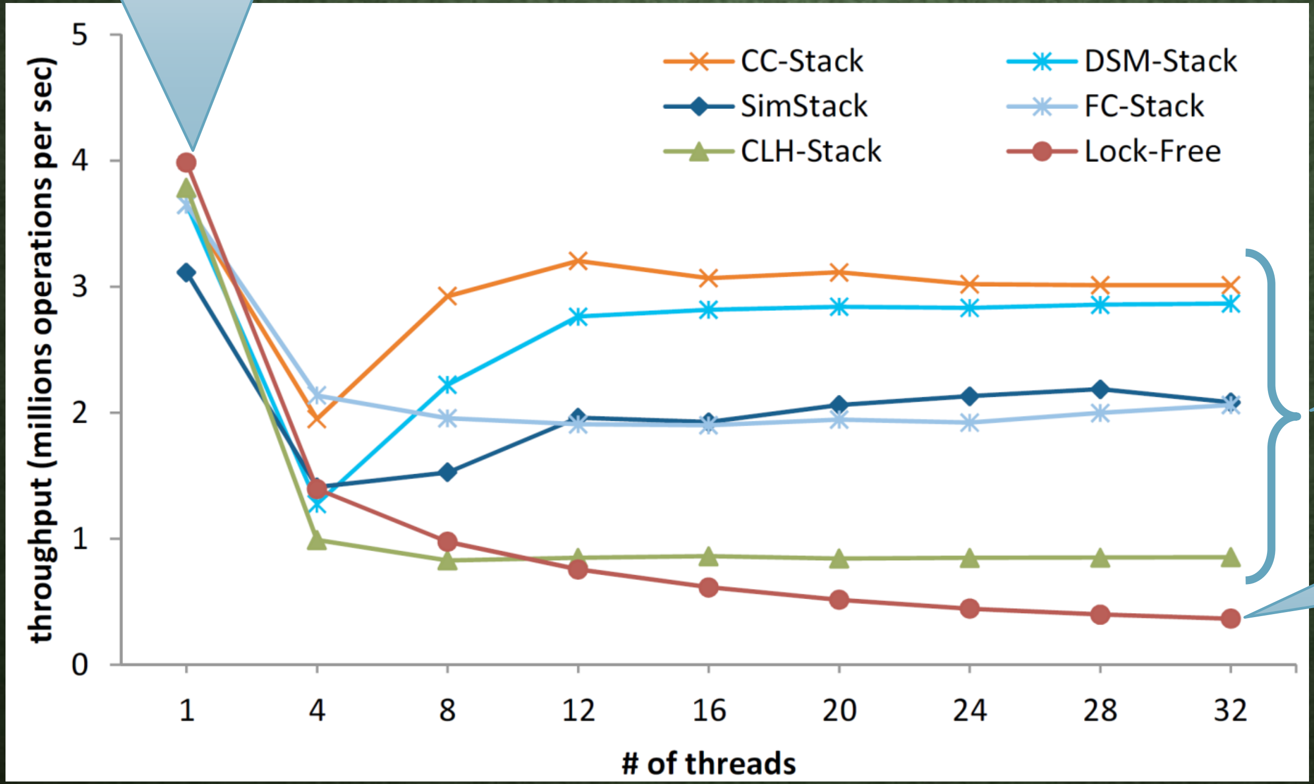
LAST TIME

- We proved that our lock-free stack is correct (linearizable)

- **This time:**
  - Stack performance
  - Difficulties in **using** ordered data structures
  - Harnessing disorder

PERFORMANCE

Are stacks really suitable for multicore programming?
One thread is best…



Other stacks developed up to 2012

Treiber stack we just saw

QUEUES

- Like stacks, but FIFO instead of LIFO
- Logical next step
  - Concurrent modification of **two** pointers (head/tail) rather than just one (stack top)
- Not covering in detail (no implementation / proofs)
  - They don't scale
  - Are they really useful? **Mainly just for handing data from one thread to another…**

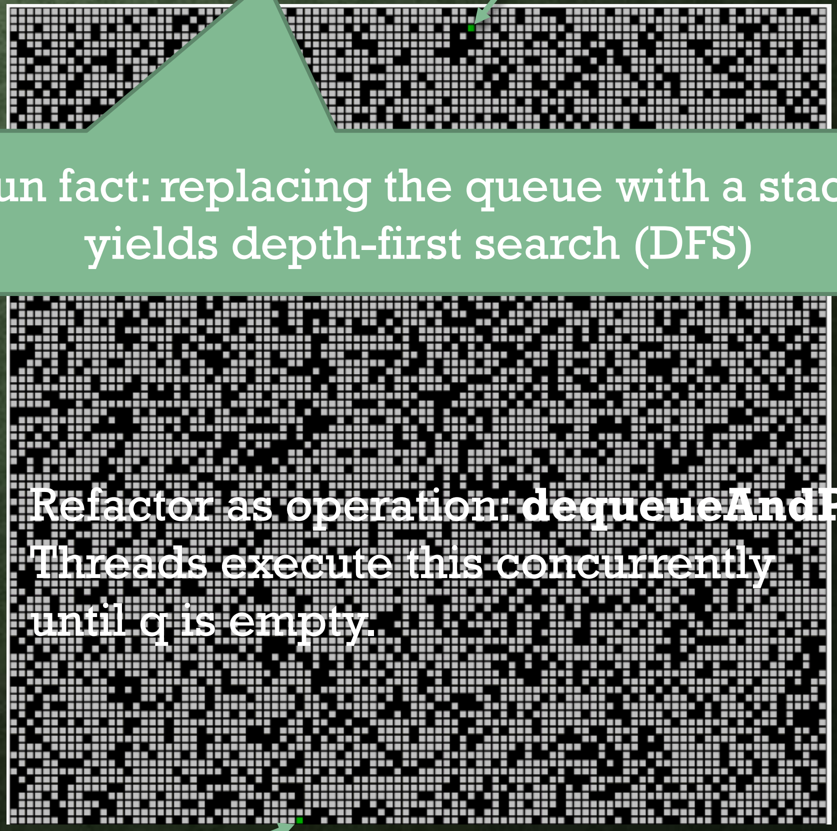
WHY WOULD WE WANT CONCURRENT STACKS OR QUEUES?

- Suppose we have a fast concurrent queue
- Do we care?
- Why use a queue over something with no ordering guarantees?
  - Less ordering would allow more concurrency (and better performance)
  - Must **need** the order!
  - Can we actually use the ordering a concurrent queue provides to do anything useful?

EXAMPLE: BREADTH-FIRST SEARCH (BFS)

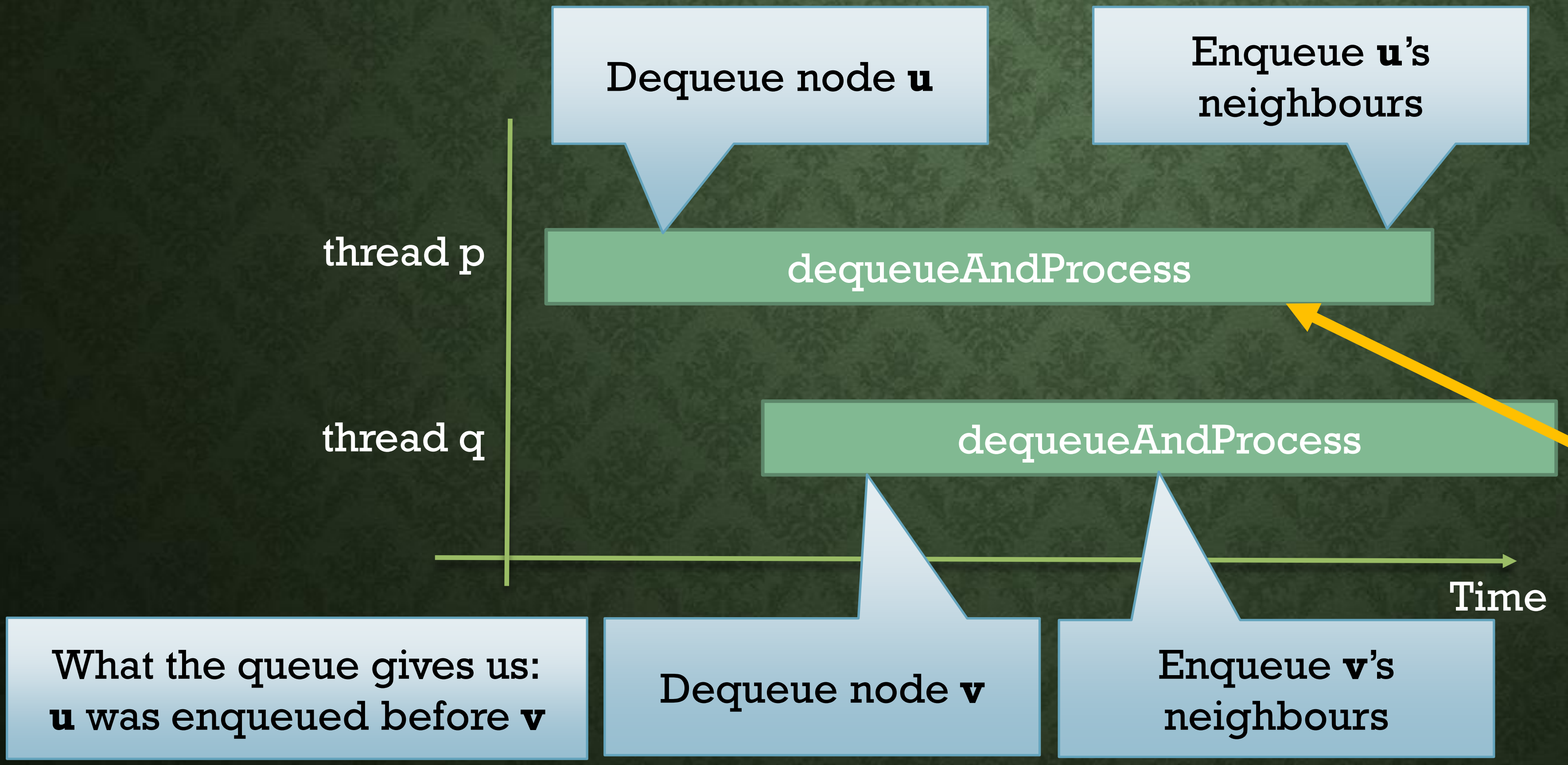
- Graph traversal algorithm that depends on FIFO ordered queue
- BFS(startingNode, visitFunction)
  - q = new ~~Queue~~ ConcurrentQueue
  - q.enqueue(startingNode)
  - while q is not empty
    - curr = q.dequeue()
    - visitFunction(curr)
    - for each neighbor n of curr
      - if n has not been visited and is not in q
        - q.enqueue(n)

Fun fact: replacing the queue with a stack yields depth-first search (DFS)



Refactor as operation: **dequeueAndProcess.** Threads execute this concurrently until q is empty.

DOES QUEUE ORDERING TRANSLATE INTO TRAVERSAL ORDERING?



In the sequential algorithm, **u**'s neighbours are enqueued first!

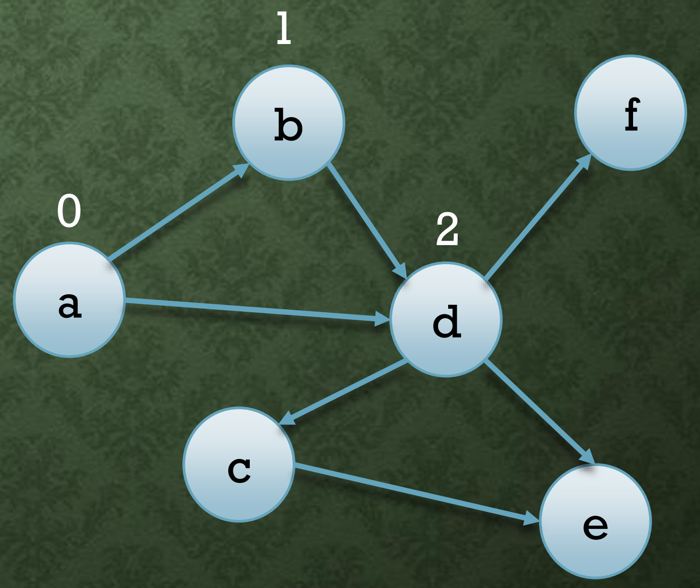
Can get inversions in the enqueue order! **Not** a BFS!

Sleeping here can lead to **huge** inversions!

Even with strictly ordered data structures, **the thread scheduler creates disorder**!

CAN WE FIX THE BFS ALGORITHM?

- Consider a BFS starting from <u>**a**</u> used to compute distances from <u>**a**</u>
- Thread p:
  - Dequeue a
  - Enqueue neighbor b @ dist 1
  - **Sleep** <u>before</u> enqueuing d @ dist 1
- Thread q:
  - Dequeue b
  - Enqueue d @ dist 2
- Must somehow <u>**fix**</u> d's distance to get a correct result!

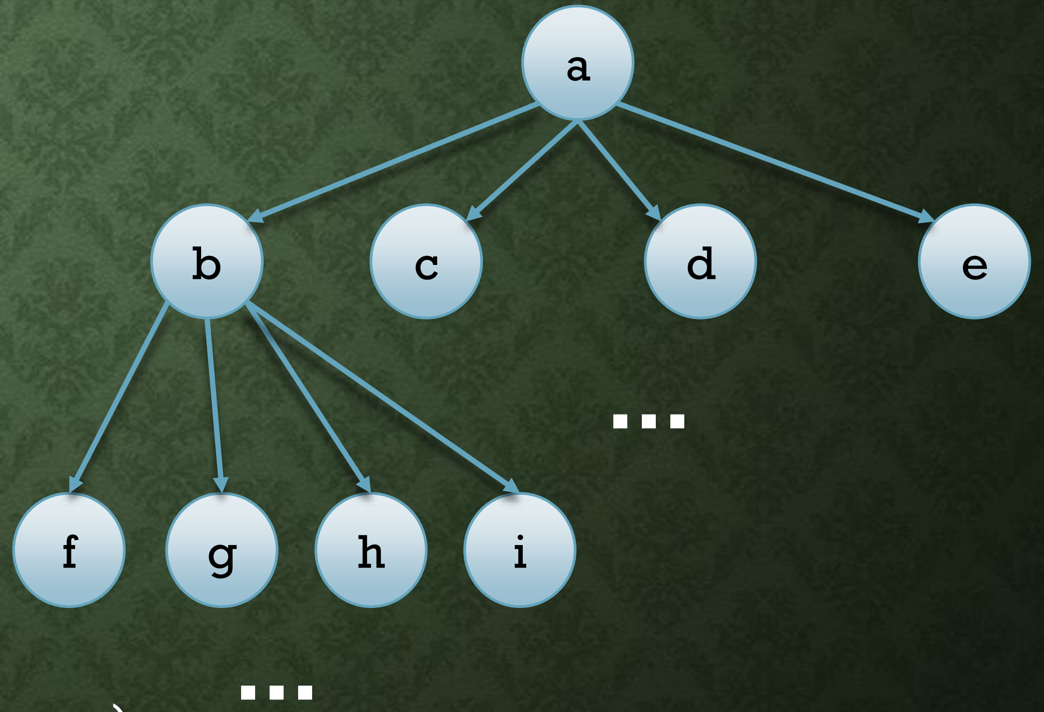


ALGORITHMIC IDEA

- Allow out of order processing of queue elements
- Instead of visiting each node once, visit repeatedly
- On each visit, iteratively improve distance
    - Starting to sound sort of like Dijkstra's algorithm…
- If the distance to a node is **not** improved, don't enqueue the node
    - (No need to update its neighbours,
      because it won't change the distance to them)
- With these changes, we can tolerate the inversions created by the thread scheduler that interfere with the FIFO processing of nodes

A TRADEOFF ARISES

- Original BFS only visits each node once
- Now, we may visit a node many times
- However, we may also gain parallelism
- The question: how much do we win vs lose?
  - Win: parallel node processing
  - Lose: wasted work revisiting nodes
- For example: big win in trees
  - (1 path to each leaf = no need to fix bad distances)



DIJKSTRA'S ALGORITHM IS SIMILAR

- Dijkstra's algorithm already incrementally improves distances
- Like BFS, but with a **priority queue** that sorts by distance
- Instead of dequeue, it uses dequeueMin
- Each node is only visited once
  - Because of the strict priority queue ordering
- Without the strict priority ordering, nodes may need to be visited multiple times
- Similar tradeoff → can win by **relaxing** the ordering

ROLE OF ORDERING

- Strict FIFO queues **do not** make it easy to implement concurrent BFS
- Concurrent BFS does not need to rely on FIFO (Dijkstra's similar)
- How much should we order our data?
  - Strict orders kill concurrency
  - Random orders *may* perform poorly
- Data structures with **relaxed ordering**
  - Relaxed stacks, relaxed queues, relaxed priority queues
  - Typically provide bounds on how out-of-order things can get

Meta-point: concurrency is diametrically opposed to ordering. Ordering → synchronization → waiting.

HARNESSING DISORDER

Concurrent **relaxed** **q**ueues

<u>RELAXED</u> QUEUE OBJECT

- Operations:
  - Enqueue(e)
    - Adds element **e** to the back of the queue
  - Dequeue()
    - Removes **<u>some element</u>** from the queue and returns it
- Meaningless without a **quality guarantee**
  - For example: “dequeue returns one of the k oldest keys in the queue”
  - (Otherwise it offers **no** ordering guarantees)

MULTI-QUEUE [ABKLN2018]: A CONCURRENT RELAXED QUEUE

- Pick your favourite sequential or concurrent **priority** queue implementation X
- We will use X as an algorithmic **building block**
  - If X is sequential, we protect it with a lock
- Idea:
  - Let N be the number of threads in the system
  - Assume threads have access to a consistent clock (wall time)
  - Create N separate **priority** queues of type X (called subqueues)
  - Threads will randomly pick subqueues to work on (in a particular way)
  - Prove dequeue operations return something "close" to the oldest key

PRIORITY QUEUE OBJECT

- Stores keys and associated priorities
- Operations:
  - Enqueue(e, pr)
    - Adds **e** to the priority queue with priority **pr**
  - DequeueMin()
    - Removes the **highest priority** element and returns it

MULTI-QUEUE

- Enqueue(e)
  - Pick a uniform random subqueue **q**
  - t = Read(current wall time)
  - Enqueue **e** in **q** with priority **t**

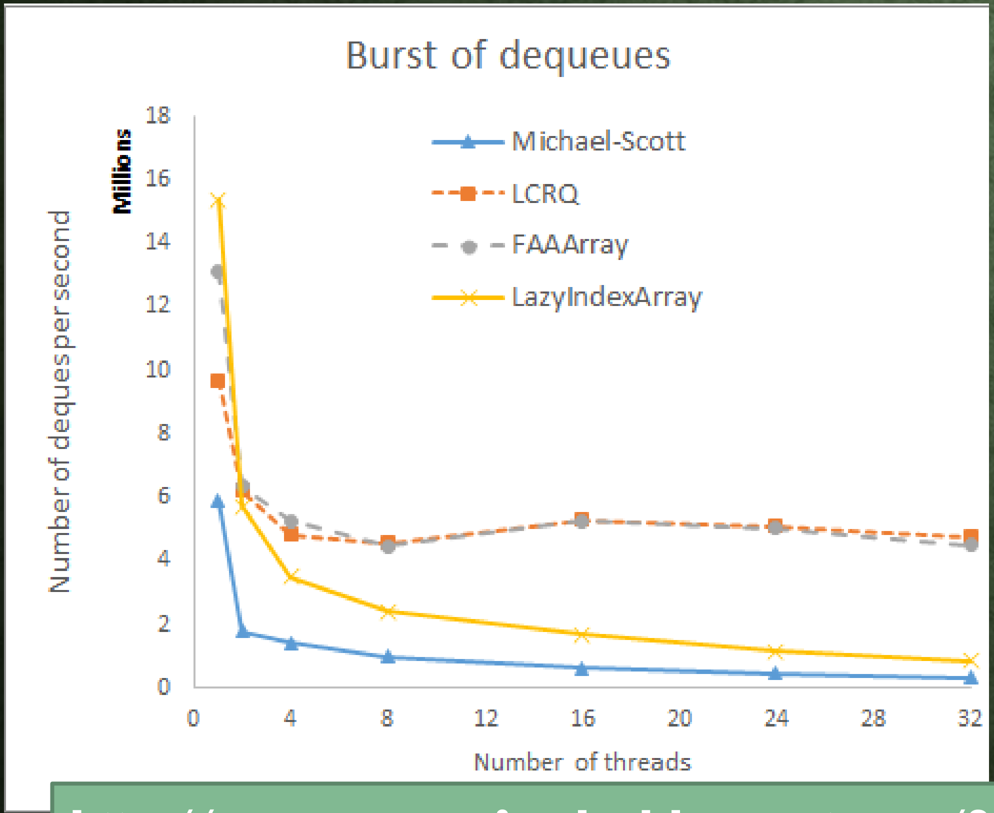
- DequeueMin()
  - Pick **two** uniform random subqueues **qi and qj**
  - Dequeue from whichever of **qi** and **qj** has the <u>older</u> **top** element

WHAT DOES THIS GUARANTEE?

- Consider a multi-queue containing S elements
- We say the oldest element has rank 1 (most desirable),
  and the newest element has rank S (least desirable)
- Dequeue returns an element:
  - with rank O(N log N) with high probability, where N = #threads
- Rank is tied to **number of threads --- independent** of queue size!
  - Very "close" to FIFO for large queues
  - More accurate as queue gets larger

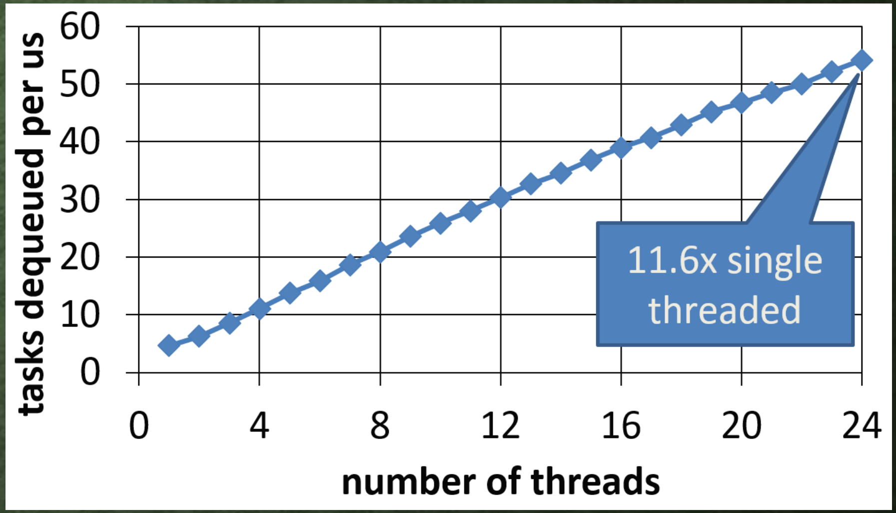
HOW DOES IT PERFORM?

- Leading Strict FIFO queues (up to 2016)
- No real scaling



http://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html

- **MultiQueue**



RECAP

- Challenges of actually using stacks/queues and other **ordered** data structures
- Strictly ordered data structures such as queues
  - limited concurrency
  - algorithms such as BFS cannot easily *harness* this strict ordering
- Relaxed data structures
  - *somewhat* ordered --- allow some *inversions* in the strict ordering (better scalability)
  - applications that can handle the inversions can benefit from this scalability
    - example of relaxed BFS / Dijkstra's
    - tradeoff between greater scalability and repeated work (node adjustments)
- Discussion of NUMA effects (L3 invalidations/misses when running on different sockets)
  - `lscpu` to see CPU topology ; `taskset -c 0-7,16-23` and `numactl -N 0` to pin threads