

Concurrent data structures

Tom Iagovet

Master's student of Prof. Trevor Brown

University of Waterloo

for CS 848, Fall 2026



UNIVERSITY OF WATERLOO
FACULTY OF MATHEMATICS
David R. Cheriton School
of Computer Science



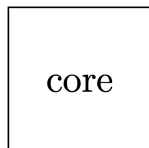
Multicore Lab

HARNESSING MASSIVELY
PARALLEL SYSTEMS AT SCALE

motivation

traditional computer

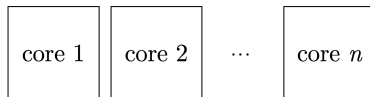
a *core* executing instructions
sequentially



→ LOAD
ADD
STORE
⋮

modern computer

many cores executing
instructions **in parallel**



goal harness parallelism between cores

- ▶ parallelizing can be easy

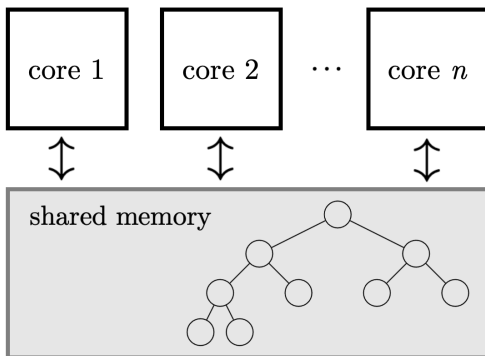
e.g. squaring every number in a list

$$\begin{array}{c} [3 \quad 1 \quad 4 \quad 1 \quad 5 \quad 9 \quad 2 \quad 6 \quad 5 \quad 3] \\ \Downarrow \\ \underbrace{[3^2 \quad 1^2 \quad 4^2 \quad 1^2 \quad 5^2]}_{\text{core } A} \quad \underbrace{[9^2 \quad 2^2 \quad 6^2 \quad 5^2 \quad 3^2]}_{\text{core } B} \end{array}$$

- ▶ we are interested in non-trivial parallelization

e.g. shortest paths in a graph

cores communicate through **data structures** in shared memory



today recent work, design issues + techniques

model

- ▶ n **cores**, each has its instructions + *local* memory
- ▶ **cores compute in local memory**
- ▶ *load* and *store* are all-or-nothing

LOAD x

COMPUTE new value of x

STORE new value of x

core A



core B



LOAD x

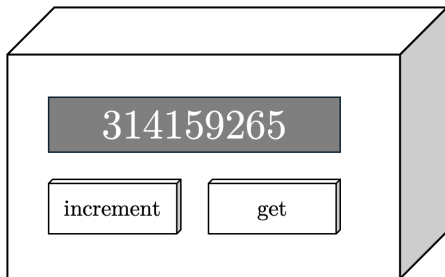
⋮

shared memory

x

counter data structure with

- ▶ *increment*: increase count by one
- ▶ *get*: get the current count



Cores need to be able run *increment* concurrently!

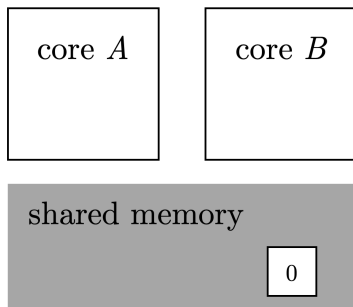
naïve implementation

an integer in memory

► *increment*:

(1) read the integer (2) increment it (3) write it back

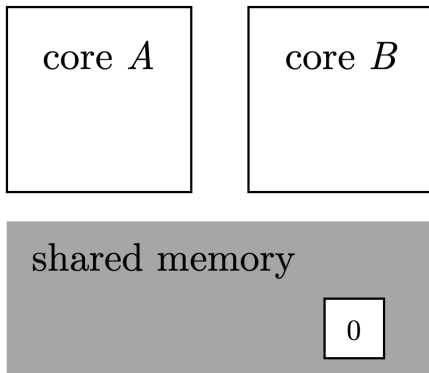
► *get*: (1) read the integer (2) return it



But remember: steps of different cores can interleave arbitrarily!

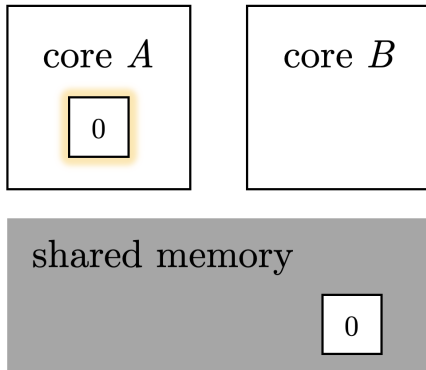
naïve's *increment*:

(1) read the integer (2) increment it (3) write it



naïve's *increment*:

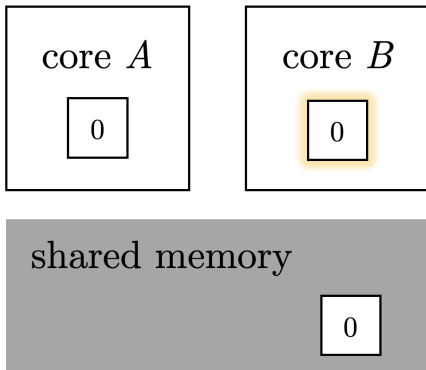
(1) read the integer (2) increment it (3) write it



core A reads

naïve's *increment*:

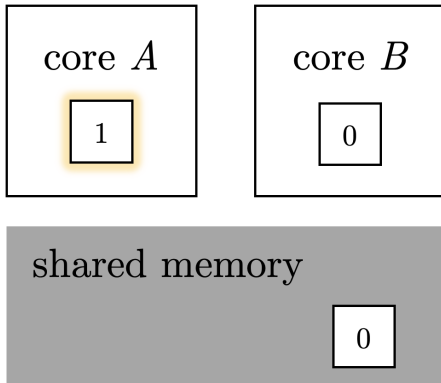
(1) read the integer (2) increment it (3) write it



core *B* reads

naïve's *increment*:

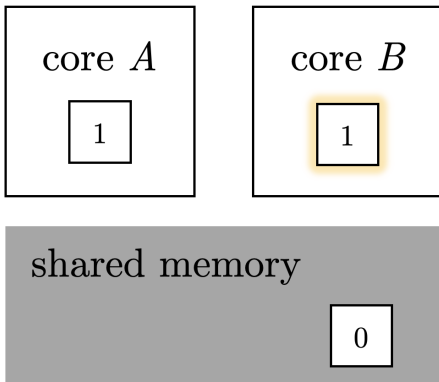
(1) read the integer (2) increment it (3) write it



core A increments

naïve's *increment*:

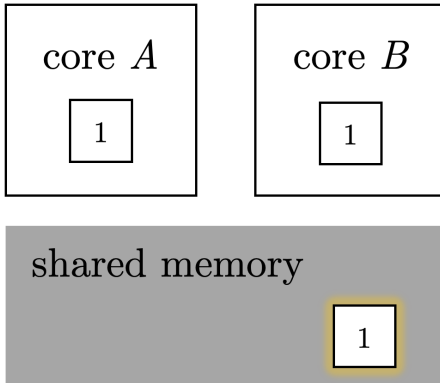
(1) read the integer (2) increment it (3) write it



core *B* increments

naïve's *increment*:

(1) read the integer (2) increment it (3) write it



core *A* writes; core *B* writes

Two increments succeeded, but the counter only shows **one**!

problem *bad* interleaving of steps possible

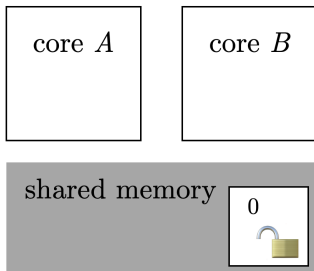
solution prevent *all* interleavings using *locks*

lock: allows only one core to access a memory location at a time

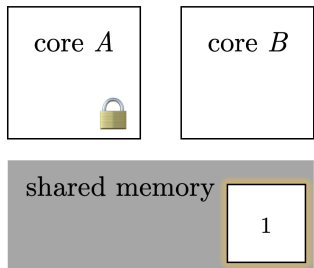


[image source: Canadian occupational safety]

lock-based counter

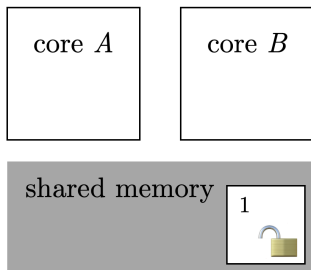


the counter now has a **lock**



when *A* wants to increment the counter, it **acquires the lock**

lock-based counter



once *A* is done incrementing, it **releases the lock**

B can now acquire the lock, and so forth...

This counter is **correct** but **slow**!

- ▶ sequential counter with one core: 400M increments / sec
- ▶ lock-based counter with two cores: 6M increments / sec
- ▶ lock-based counter with 64 cores: 4M increments / sec

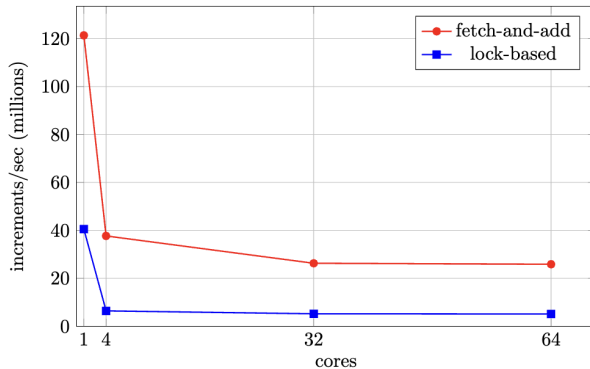
What is the problem?

problem *only one* core doing work at any time
+ locking and unlocking *takes time*

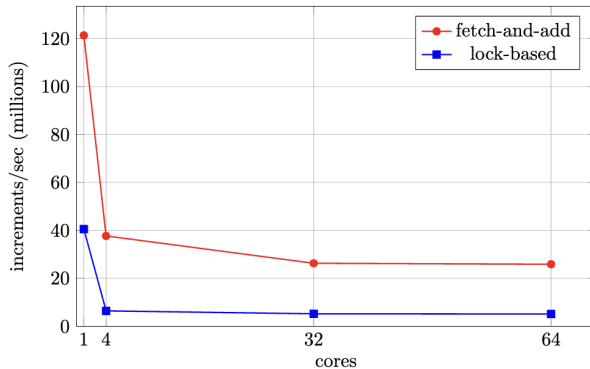
try new instruction: *fetch-and-add*

fetch-and-add: atomically read and increment a memory location

The fetch-and-add counter is *faster*, but it still *does not scale*.



The fetch-and-add counter is *faster*, but it still *does not scale*.



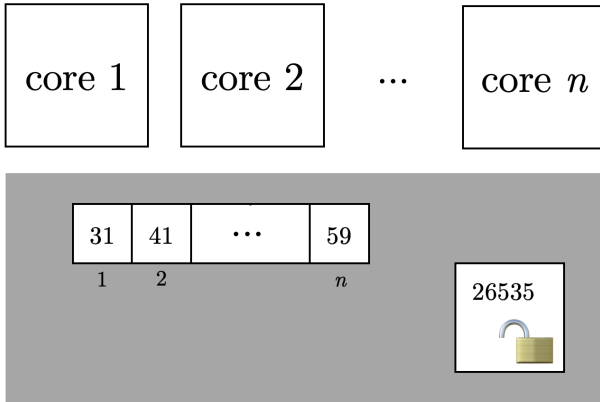
problem *memory contention*

solution partition the data

sharded counter partition the counter into subcounters

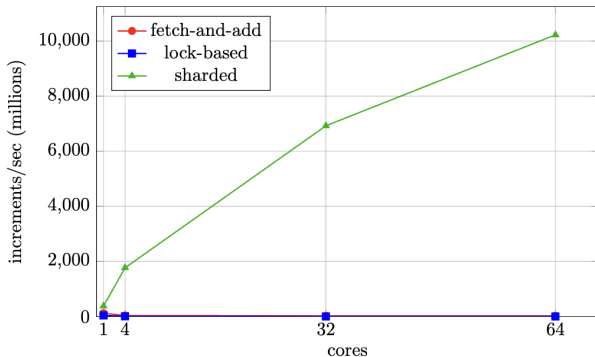
- ▶ every core gets its own subcounter
- ▶ one global lock-based counter

once every 1000 sub-increments, update global counter



sharded counter

- ▶ at least $1000\times$ less contention
- ▶ off by at most $1000 \times (\# \text{ threads})$
constant w.r.t. count

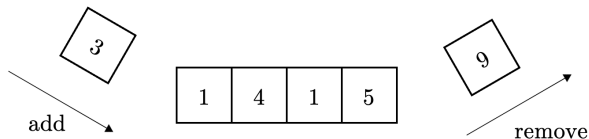


If you are implementing this counter...

false sharing

queue

first in, first out *enqueue* at the back, *dequeue* from the front



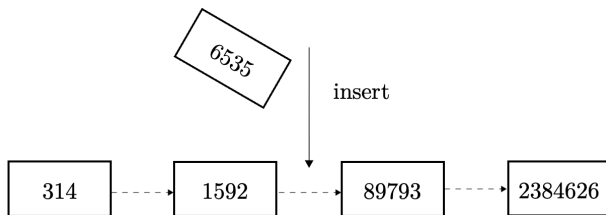
Multiqueue [ABKLN18] one sequential queue per thread

- ▶ **relaxed queue** very close to true FIFO for large queues
- ▶ **performs and scales** solves massive problem instances
e.g. MAX INDEPENDENT SET on 10^9 -vx graph in 1/2 sec

ordered set

maintain an ordering of the items:

- ▶ *insert*(item) add an item to the set
- ▶ *successor*(item) get the next item in order
- ▶ also *delete*, *predecessor*, range-queries, ...



binary search trees:

- ▶ first *linearizable* external BST [EFRvB10]
- ▶ a faster external BST [NM14]

B-trees:

- ▶ first linearizable B^+ -tree [BP12]
- ▶ Bw-tree: Microsoft's B^+ -tree [LLS13]
 - (*) open-source implementation [WPLLZKA18]
- ▶ Elimination *ab*-tree for skewed workloads [SB22]

radix trees:

- ▶ Masstree: 8B slices and B^+ -tree at each node [MKM12]
- ▶ (sequential) Adaptive Radix Tree [LKN13]
 - (*) Arctic: practical concurrent ART [NGSGFW26]

transactional memory

- ▶ **transactional memory (TM)**: treat code as **transactions**
 - ▶ transactions either *abort* or *commit*
 - ▶ can abort for many reasons
- ▶ **hardware** TM implements this abstraction as an extension to the cache coherence protocol

HTM-vEB tree: all ops in $\Theta(\log \log |\text{universe}|)$ [KBDA24]

→ **improved vEB tree** [2026, hopefully]

Concurrent data structures

- [ABKLN18] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z. Li, and Giorgi Nadiradze. 2018. Distributionally Linearizable Data Structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 133–142. <https://doi.org/10.1145/3210377.3210411>
- [KBDA24] Mohammad Khalaji, Trevor Brown, Khuzaima Daudjee, and Vitaly Aksenov. 2024. Practical Hardware Transactional vEB Trees. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 215–228. <https://doi.org/10.1145/3627535.3638504>

- [EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking Binary Search Trees. *PODC '10*. 131–140. <https://doi.org/10.1145/1835698.1835736>
- [NM14] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. *PPoPP '14*. 317–328. <https://doi.org/10.1145/2555243.2555256>

- [BP12] Anastasia Braginsky and Erez Petrank. 2012. A Lock-Free B^+ tree. *SPAA '12*. 58–67. <https://doi.org/10.1145/2312005.2312016>
- [SB22] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a, b) -trees with Fast, Durable Updates. *PPoPP '22*. 416–430. <https://doi.org/10.1145/3503221.3508441>

[LLS13] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. *ICDE '13*. 302–313. <https://doi.org/10.1109/ICDE.2013.6544834>

[WPLLZKA18] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. *SIGMOD '18*. 473–488. <https://doi.org/10.1145/3183713.3196895>

- [MKM12] Yandong Mao, Eddie Kohler, and Robert Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. *EuroSys '12*. 183–196. <https://doi.org/10.1145/2168836.2168855>
- [LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. *ICDE '13*. 38–49. <https://doi.org/10.1109/ICDE.2013.6544812>
- [NGSGFW26] Newton Ni, Nicolas Garza, Jenny Stinehour, Michael Goppert, Michal Friedman, and Emmett Witchel. 2026. Arctic: A Practical Lock-Free Adaptive Radix Tree. *OSDI '26*. 739–754. <https://www.usenix.org/conference/osdi26/presentation/ni>