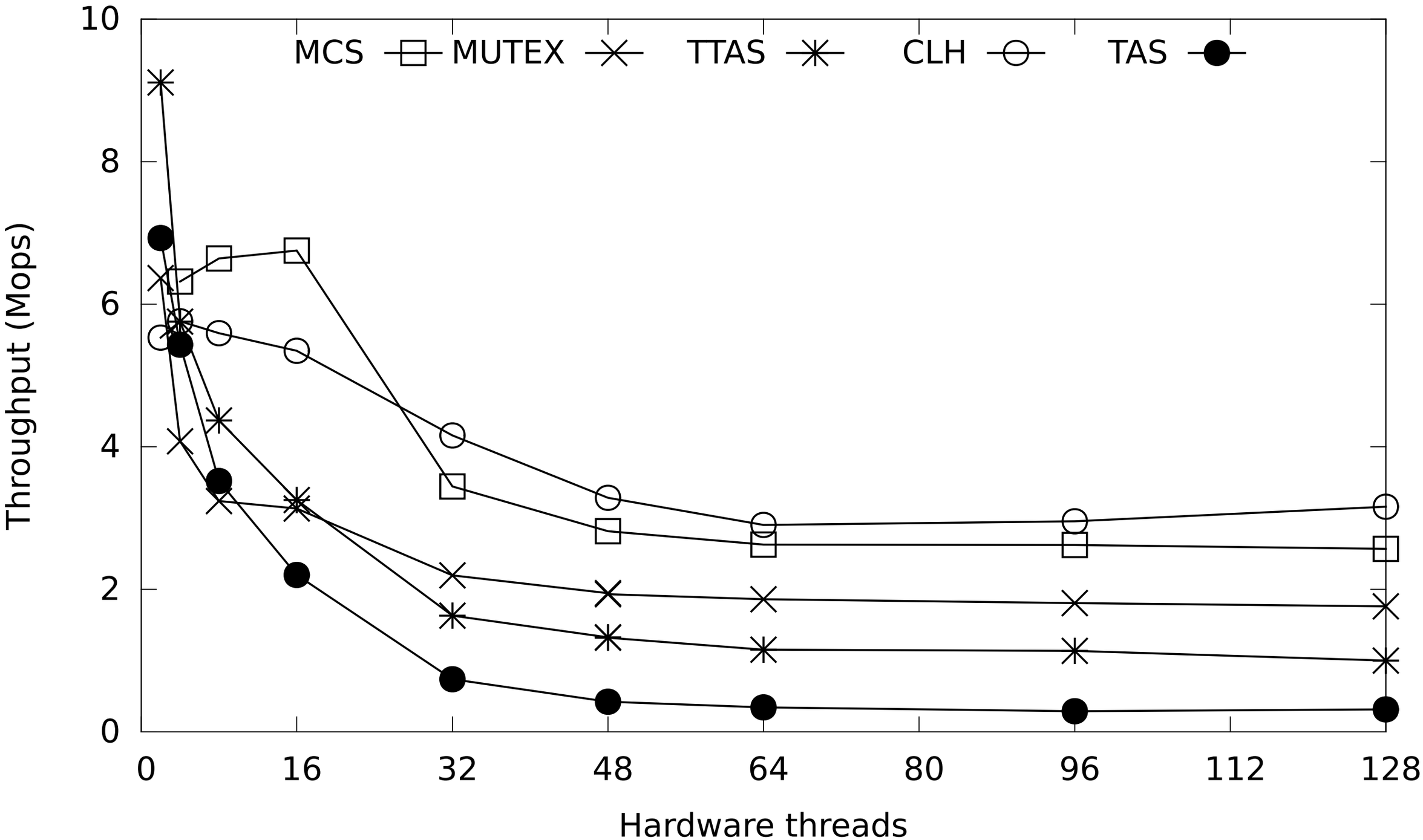
ffwd: delegation is (much) faster than you think

Sepideh Roghanchi, Jakob Eriksson, Nilanjana Basu

```
int get_seqno() {
  return ++seqno;
}
```

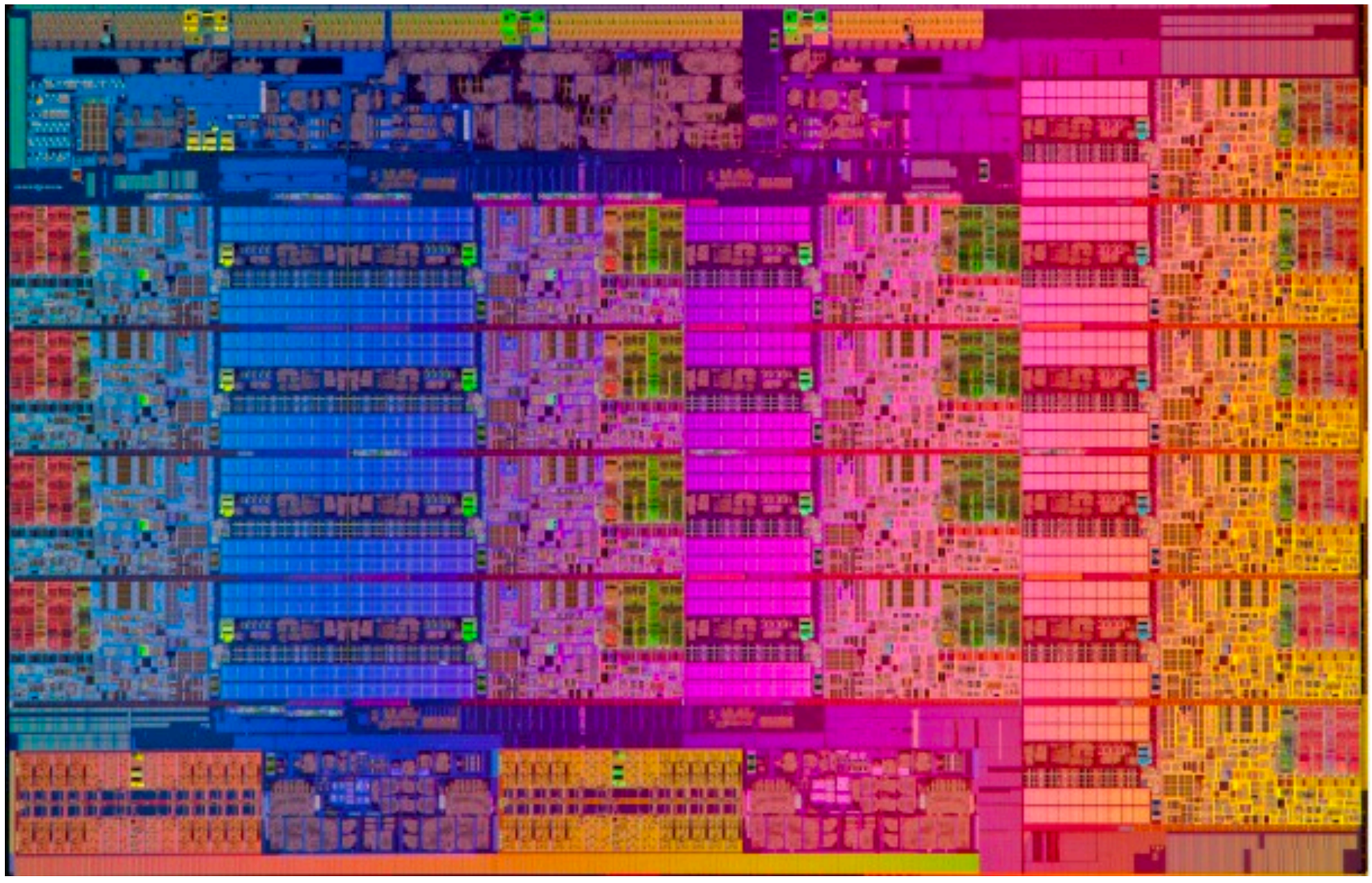
// ~1 Billion ops/s
// single-threaded

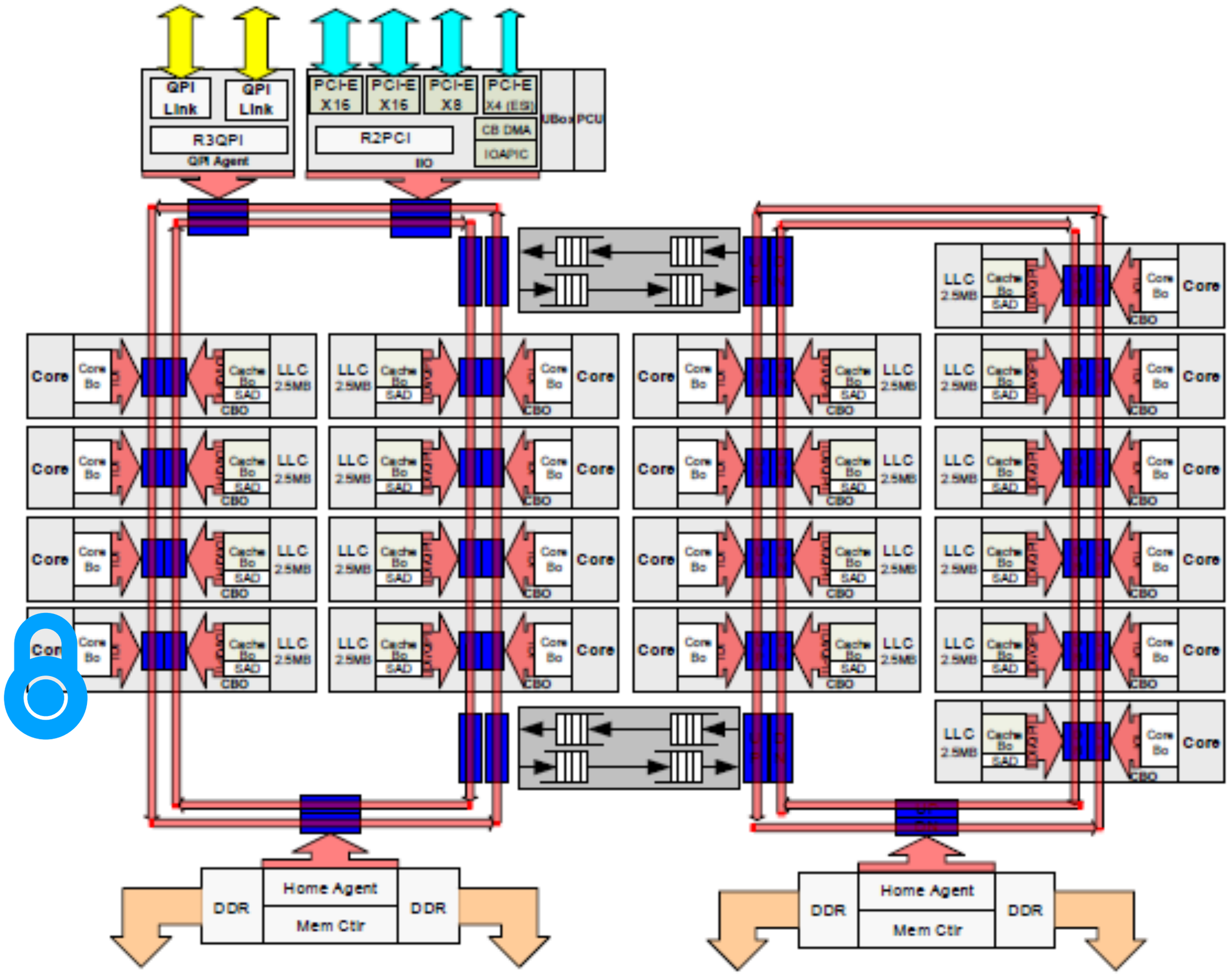
```
int threadsafe_get_seqno() {
  acquire(lock);
  int ret=++seqno;
  release(lock);
  return ret;
}

// < 10 Million ops/s
```


MCS
MUTEX
TTAS
CLH
TAS
Throughput (Mops)
0
2
4
6
8
10
0
16
32
48
64
80
96
112
128
Hardware threads

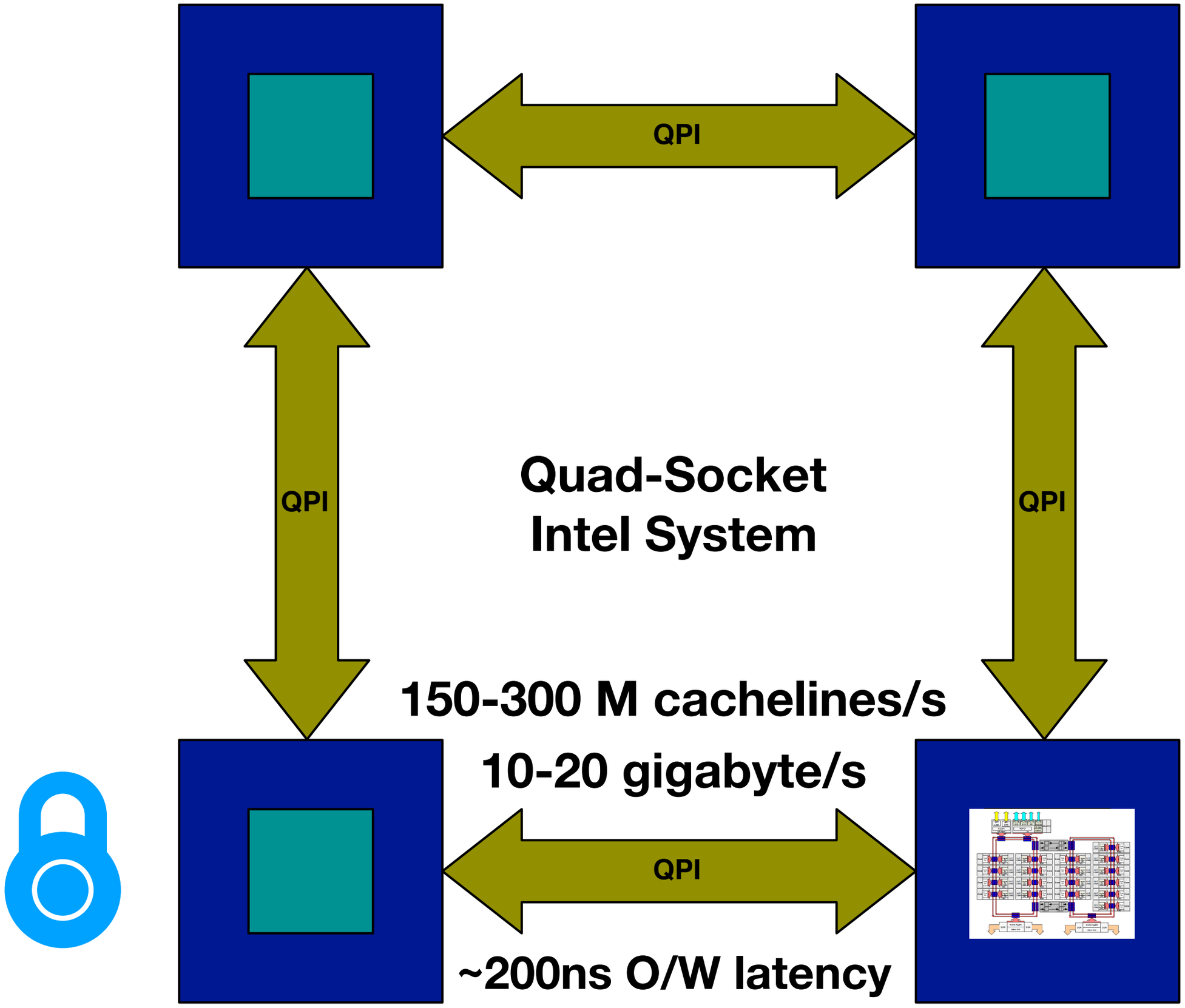
why so slow?


intel
Xeon® E7

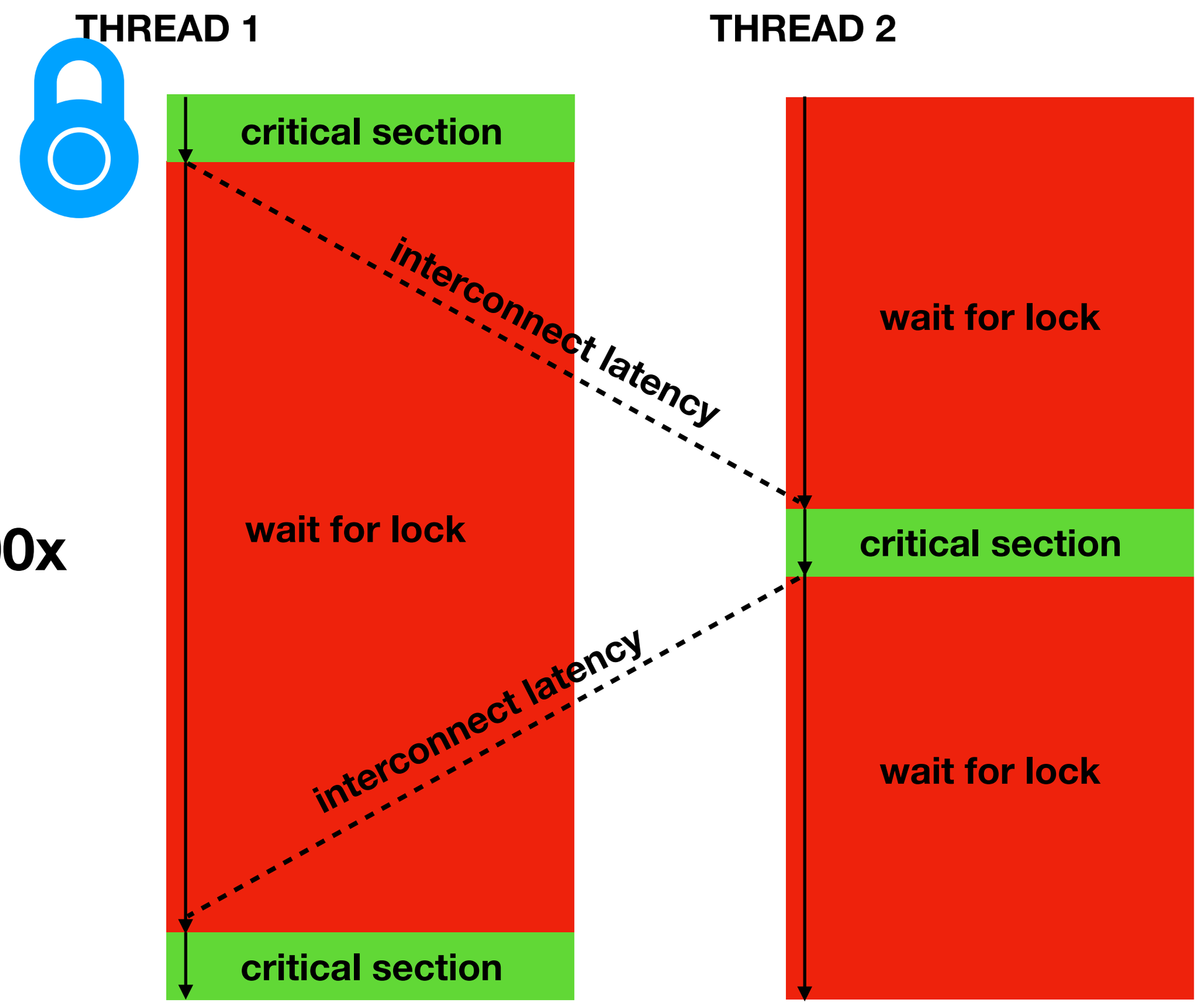


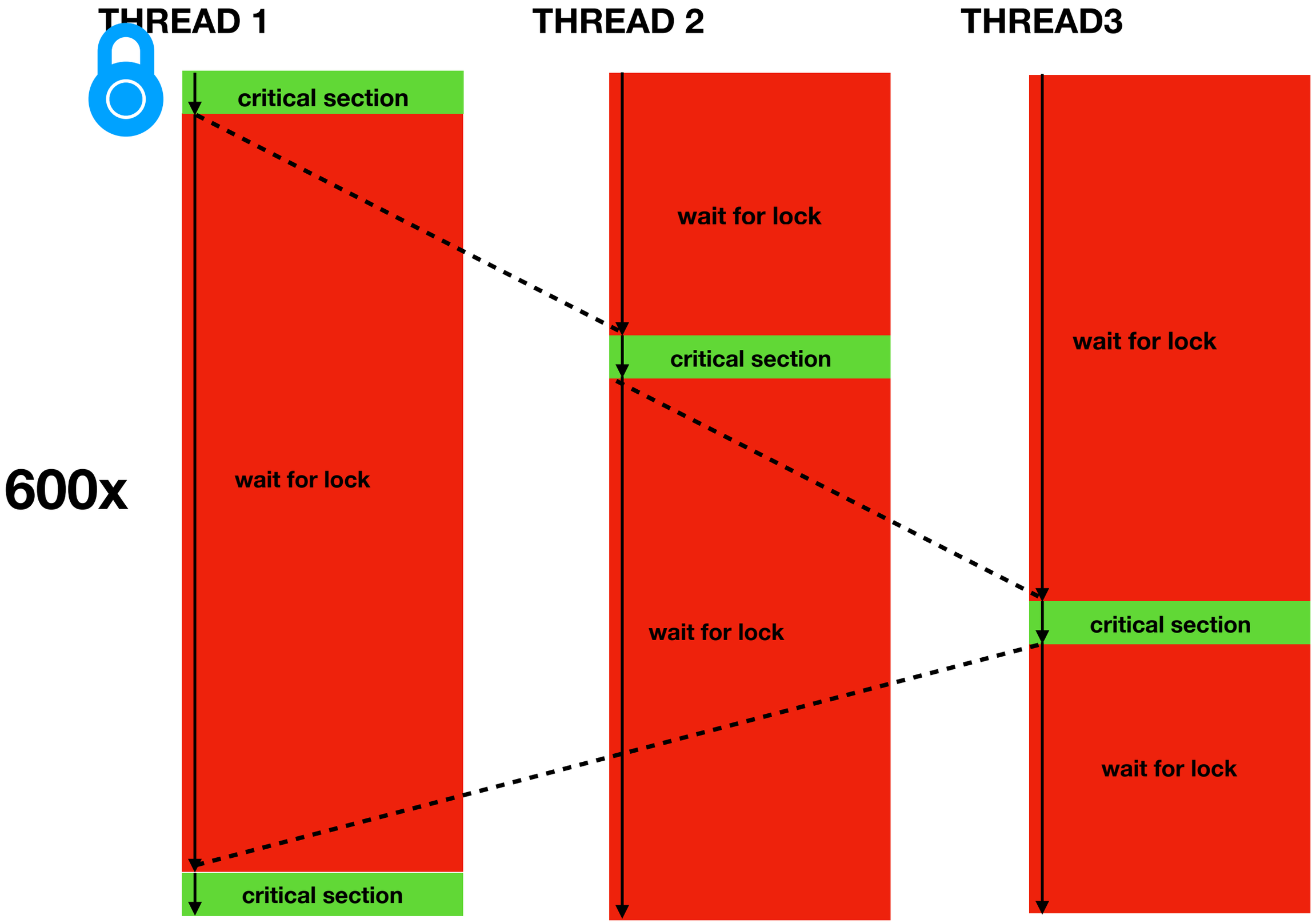


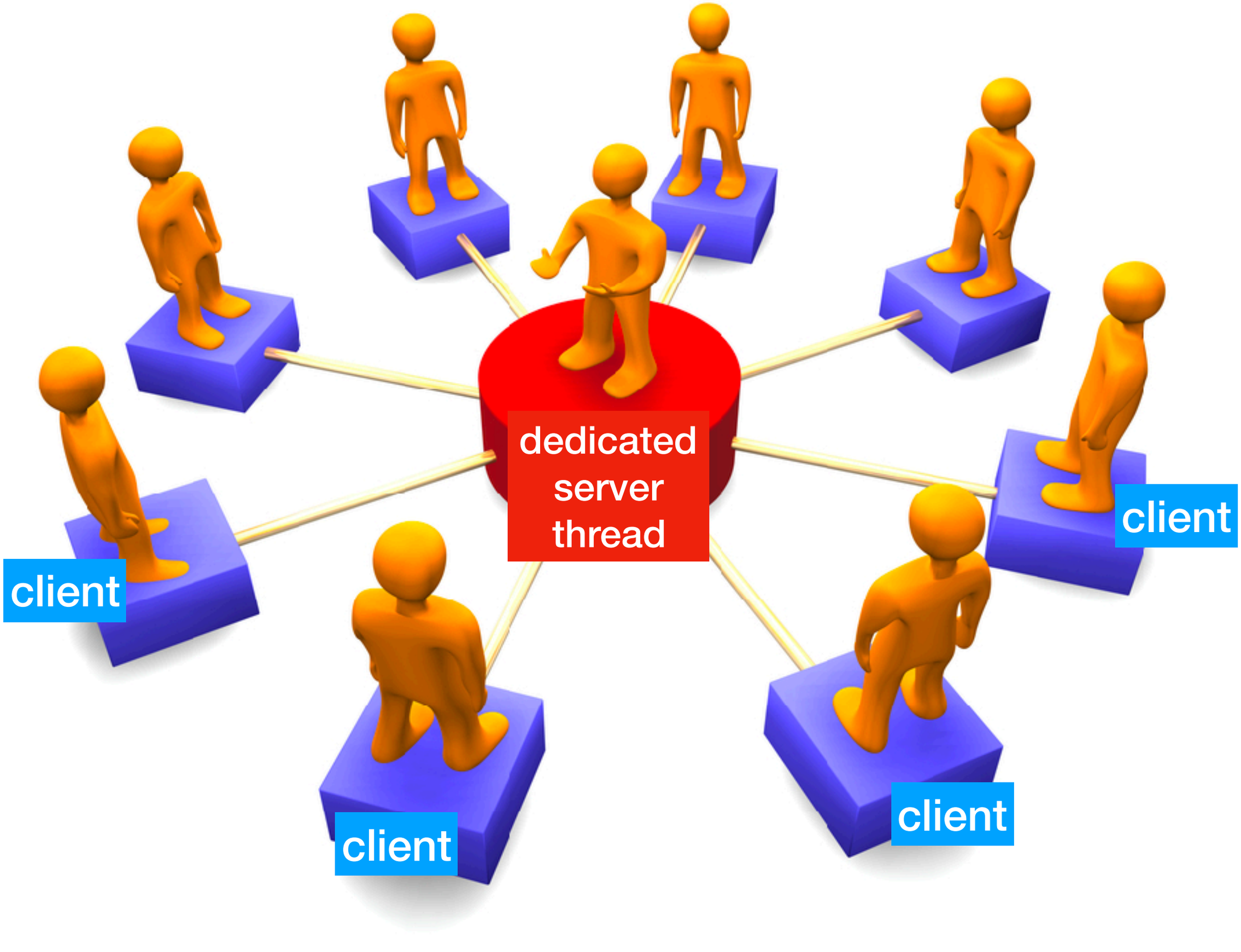
~70 ns intra-socket latency ~14 Mops


QPI
QPI
QPI
Quad-Socket
Intel System
150-300 M cachelines/s
10-20 gigabyte/s
QPI
~200ns O/W latency
= 5 million ops per second

400x


THREAD 1
THREAD 2
critical section
wait for lock
interconnect latency
wait for lock
critical section
interconnect latency
wait for lock
critical section


THREAD 1
THREAD 2
THREAD3
critical section
wait for lock
wait for lock
critical section
wait for lock
critical section
wait for lock
critical section
600x


dedicated
server
thread
client
client
client
client
client

CLIENT1

DEDICATED SERVER THREAD

400x

request

wait for response

wait for request

response

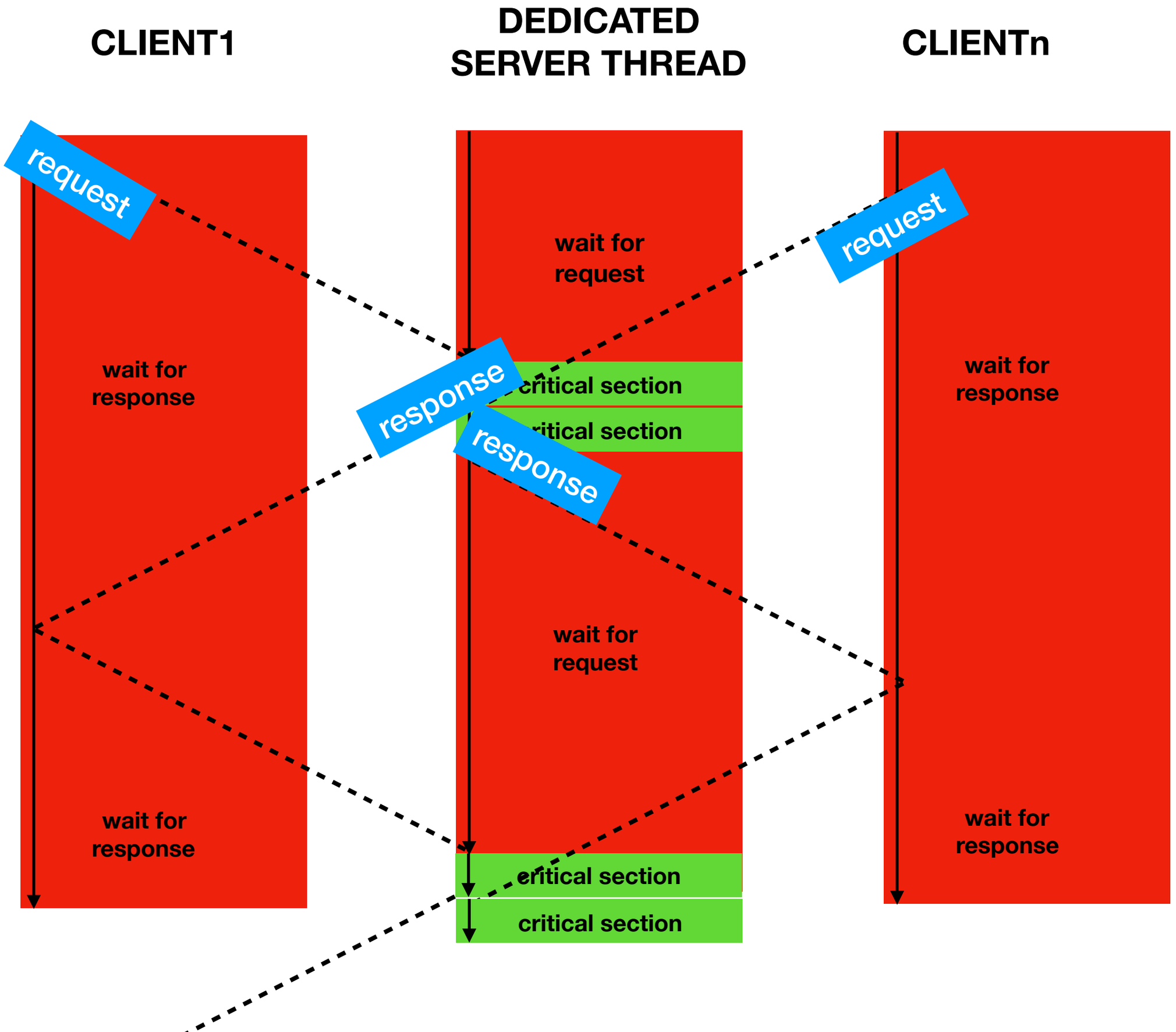
critical section

wait for request

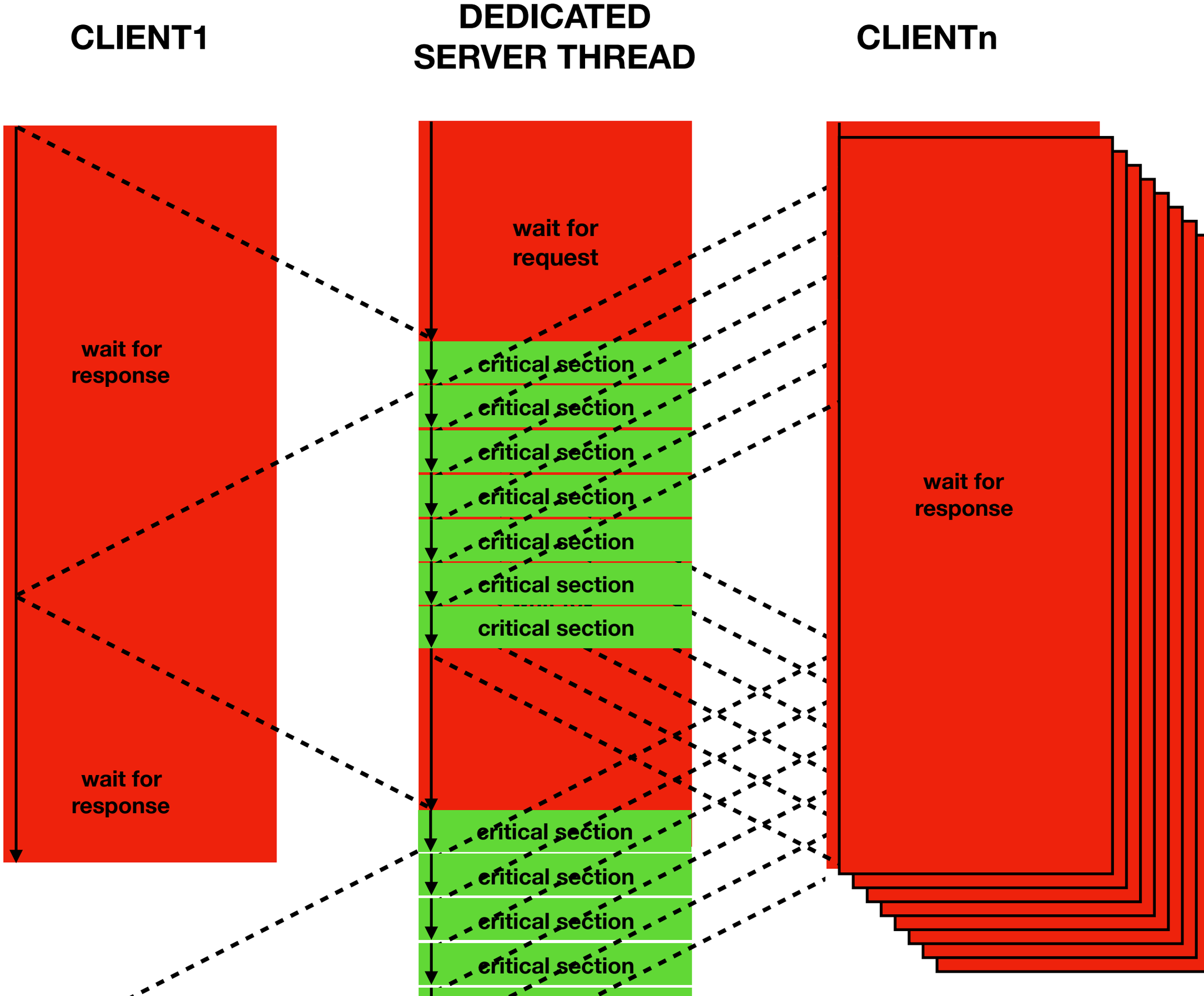
wait for response

critical section

400x still!

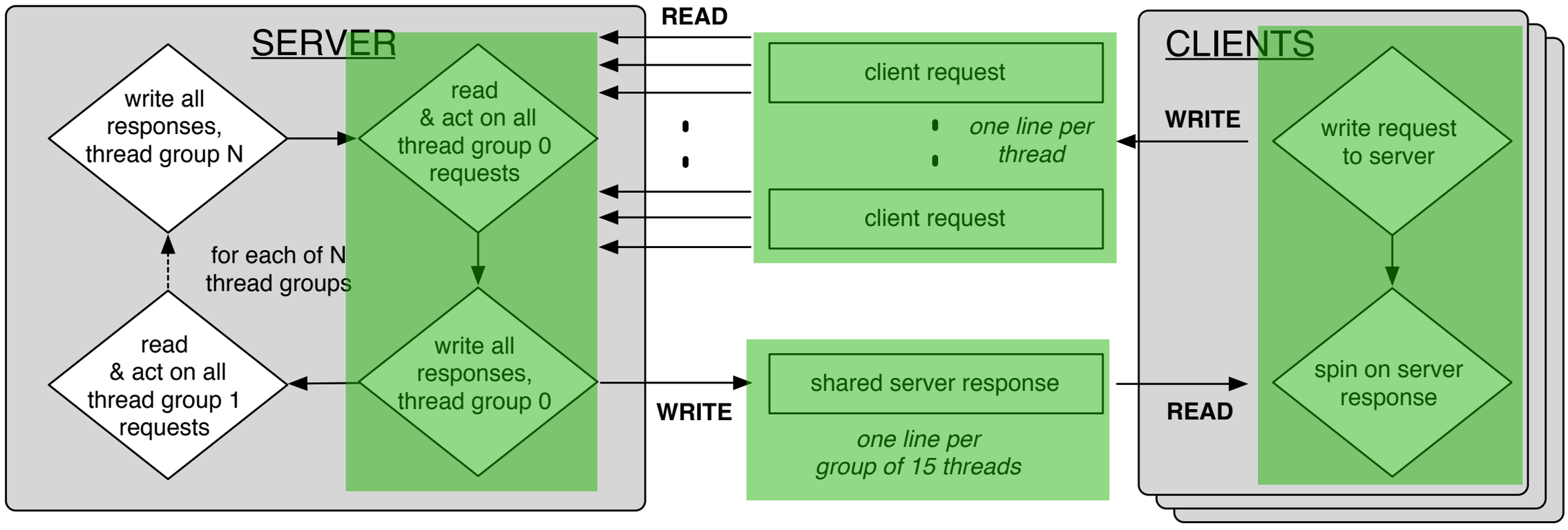

CLIENT1
DEDICATED SERVER THREAD
CLIENTn
request
wait for response
wait for request
critical section
critical section
response
response
request
wait for response
wait for request
wait for response
wait for response
critical section
critical section

400x still!


CLIENT1
DEDICATED SERVER THREAD
CLIENTn
wait for response
wait for request
critical section
critical section
critical section
critical section
critical section
critical section
wait for response
wait for response
critical section
critical section
critical section
critical section

(**f**ast, **f**ly-**w**eight **d**elegation)

ffwd design



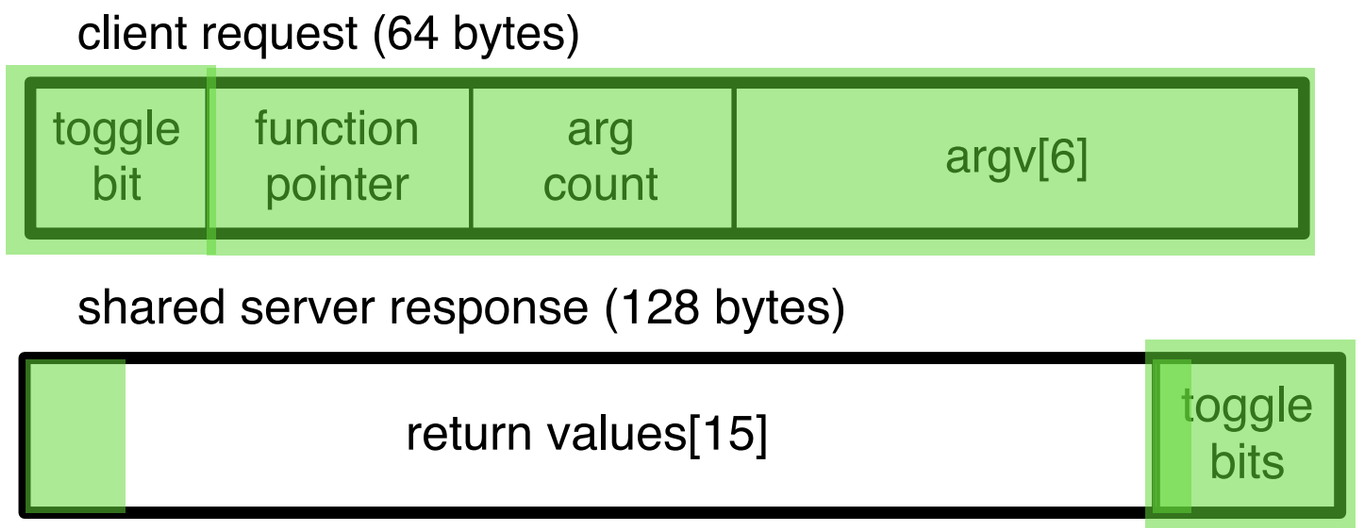
Server acts upon pending requests in batches 15 clients.

Each group of 15 clients shares one 128-byte response line pair.

One dedicated 64-byte request line, per client-server pair

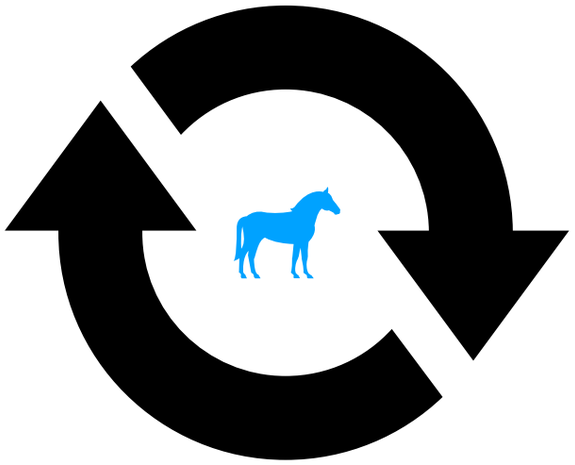
Requests are sent synchronously

A request in more detail

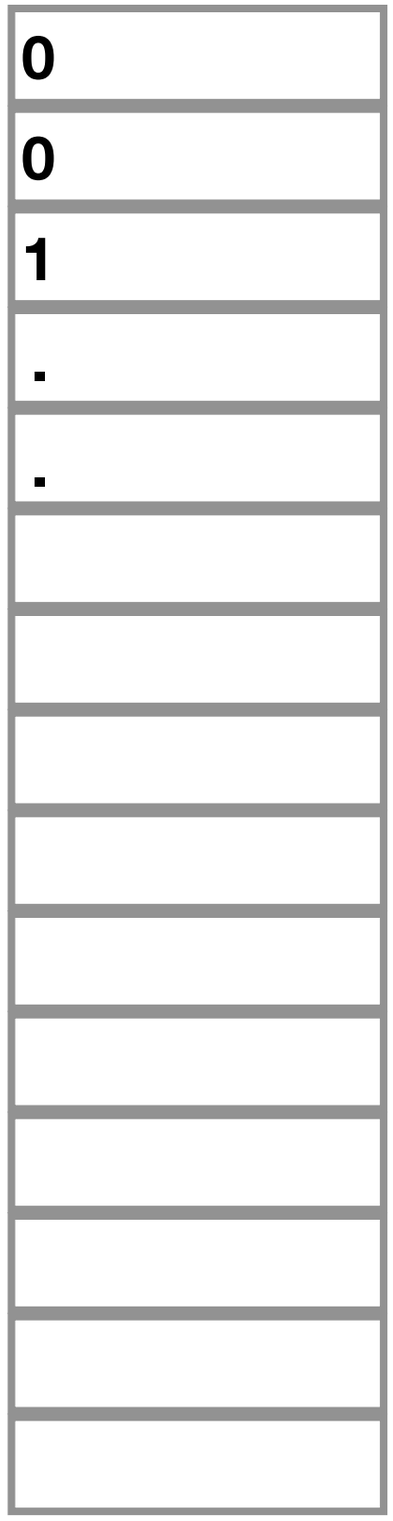


- *request is new if:* request toggle bit != response toggle bit
- server calls function with (64-bit) arguments provided
- client polls response line until toggle bit == response bit

delegation server thread

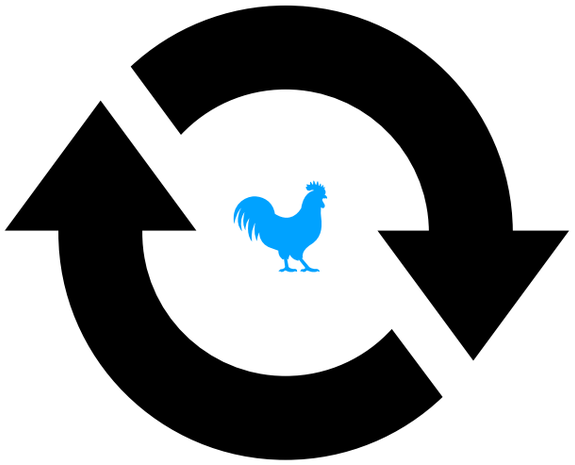


group 0 requests

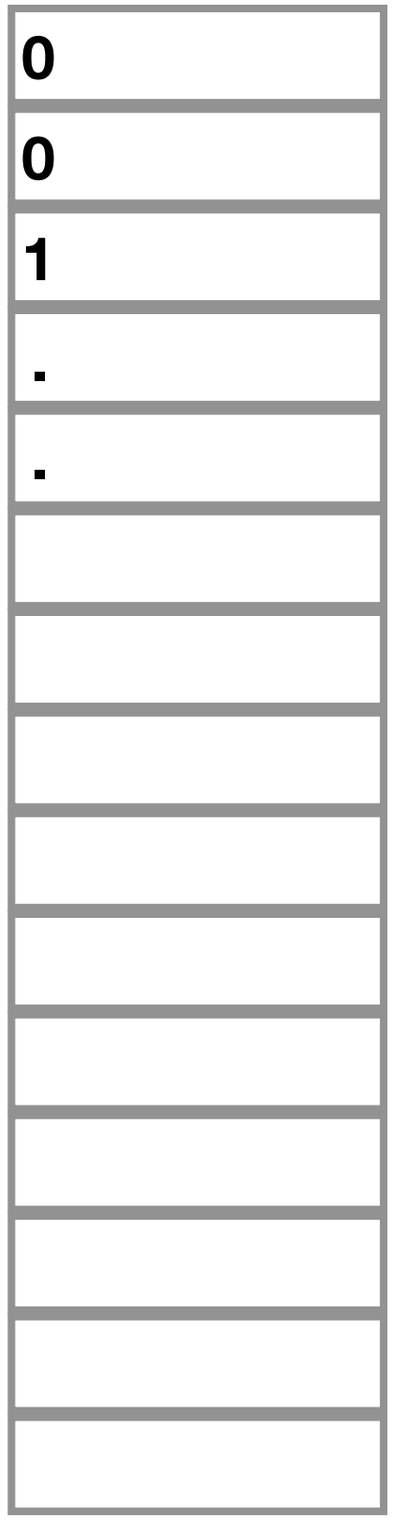


local response buffer

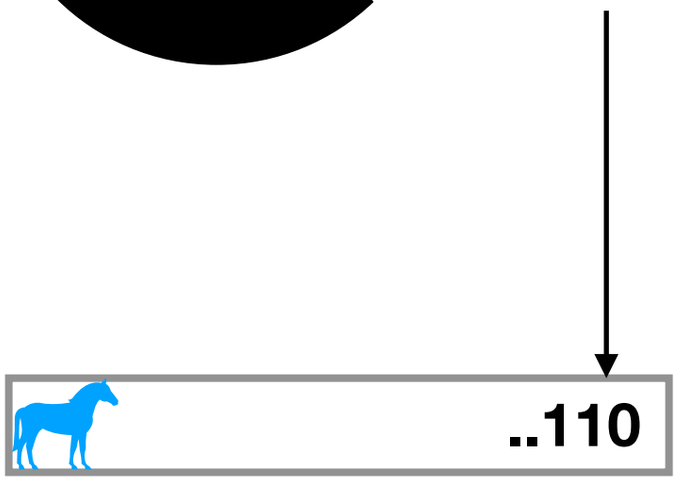
delegation server thread



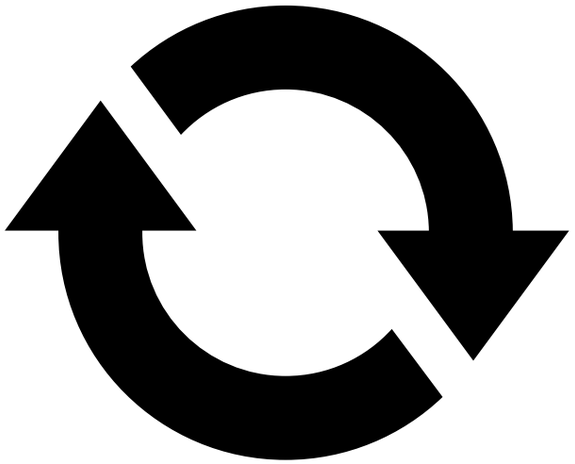
group 0 requests



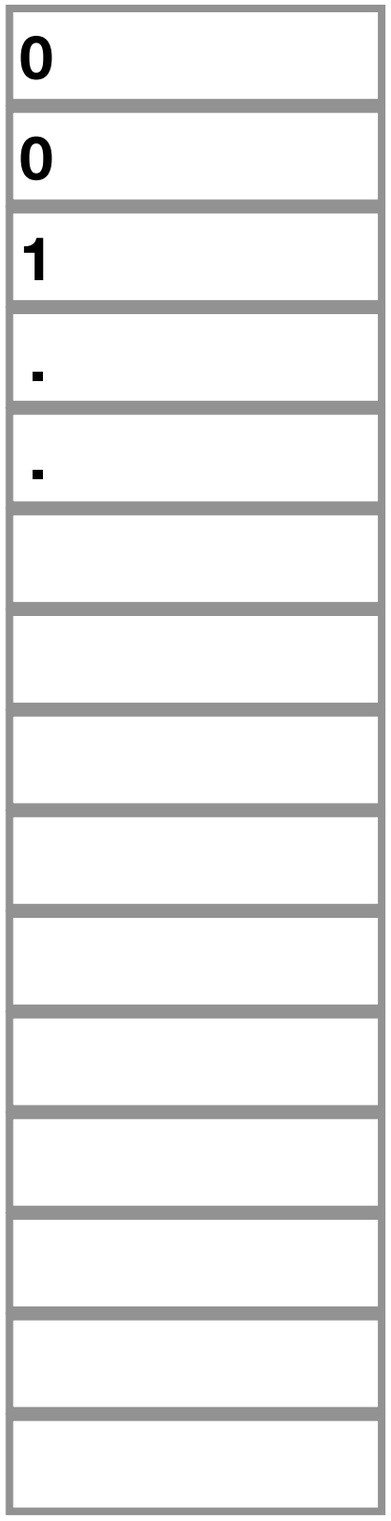
local response buffer



delegation server thread

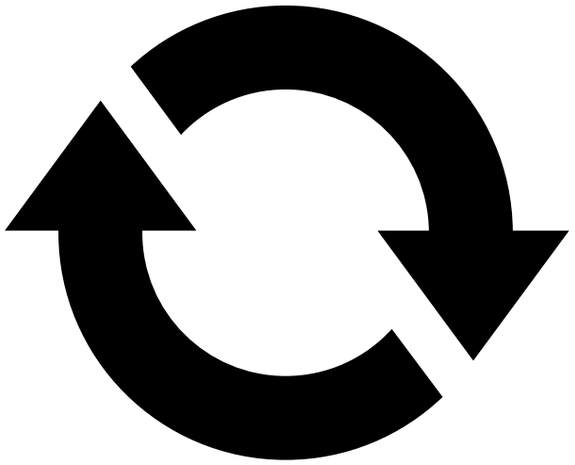


group 0 requests

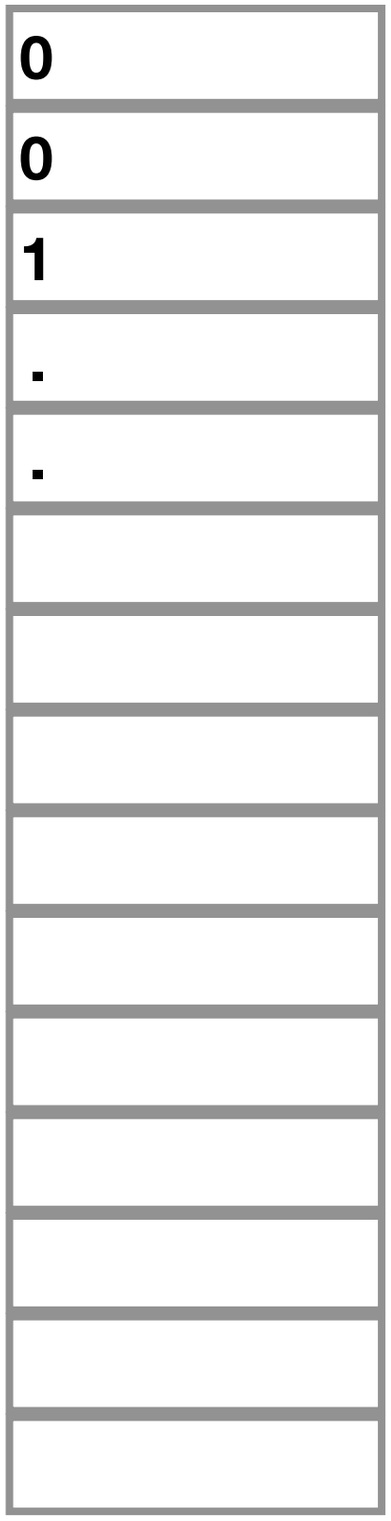


local response buffer

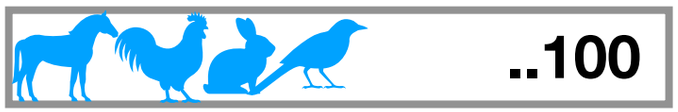
delegation server thread



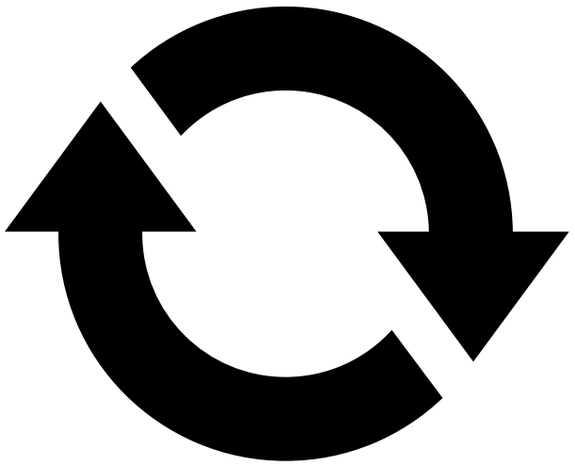
group 0 requests



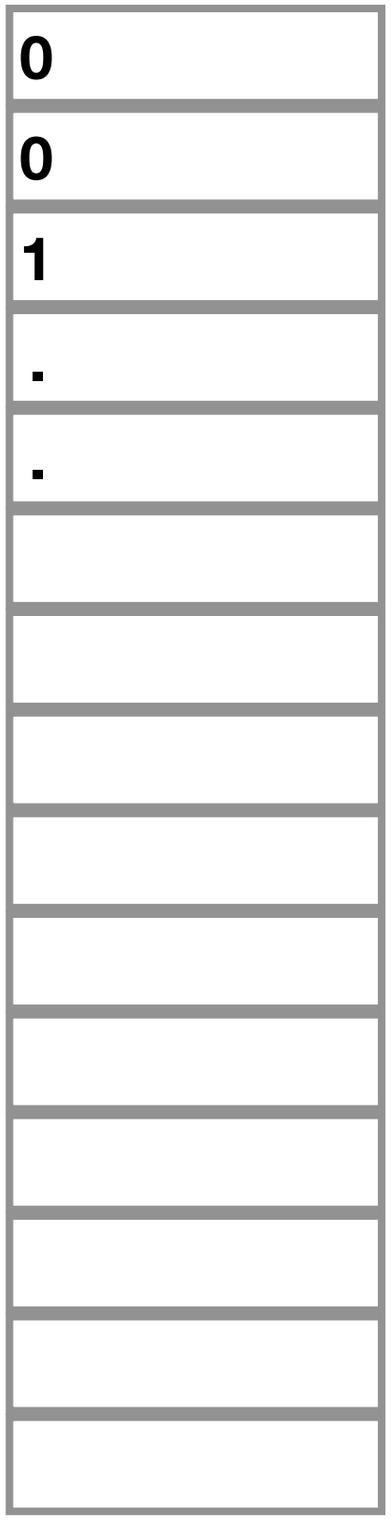
local response buffer



delegation server thread



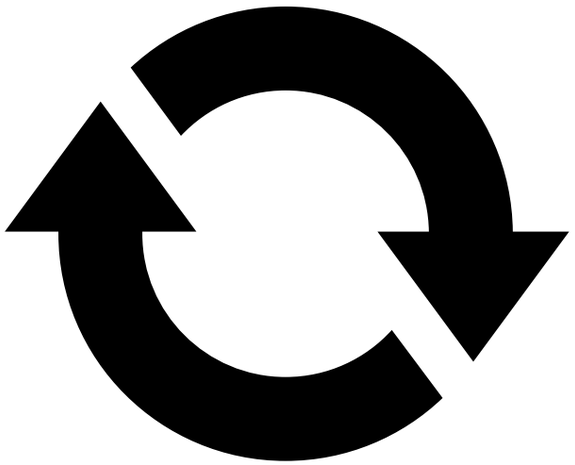
group 0 requests



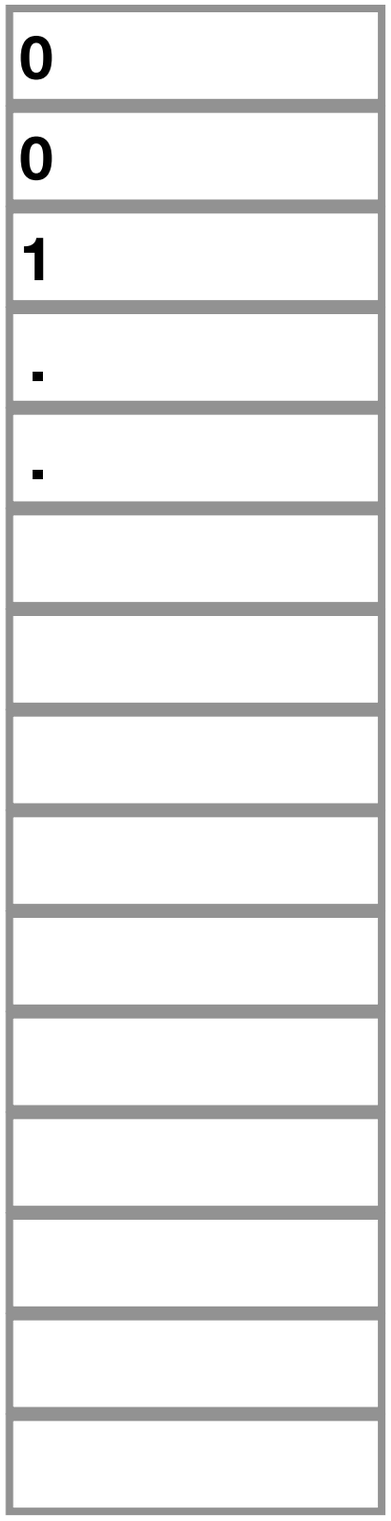
local response buffer

global response buffer

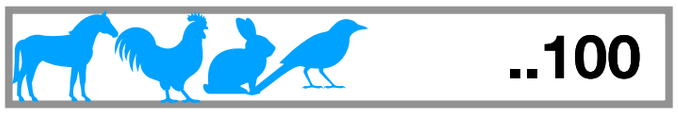
modified <— — — — — — — —response cache lines— — — — — — — —> shared



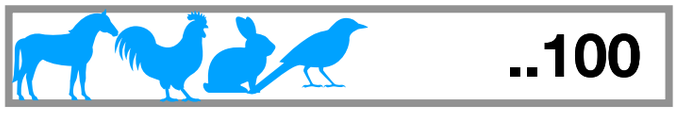
group 0 requests



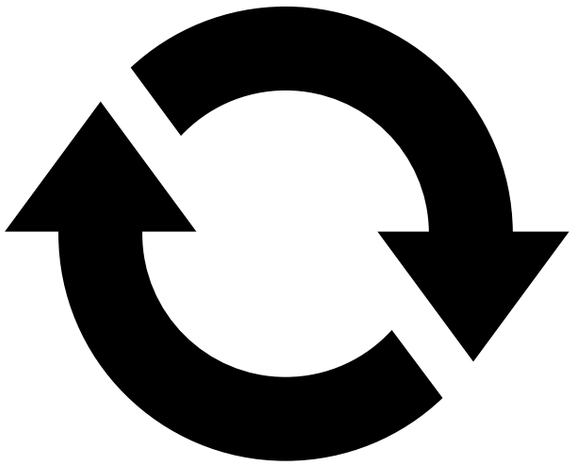
local response buffer



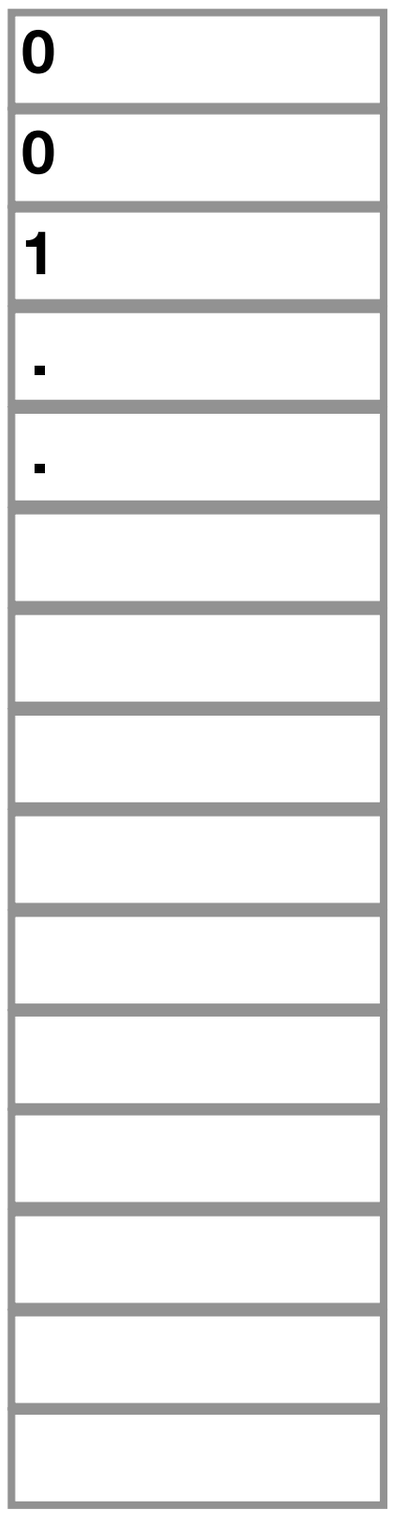
global response buffer



modified <— — — — — — — —response cache lines— — — — — — — —> shared

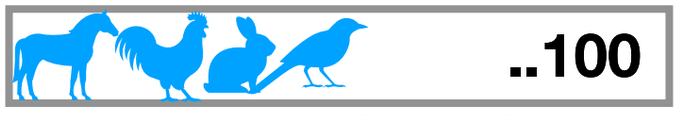


group shared requests--—> modified



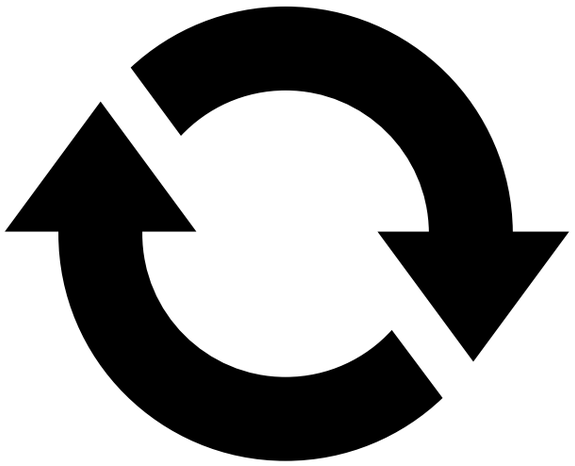
local response buffer

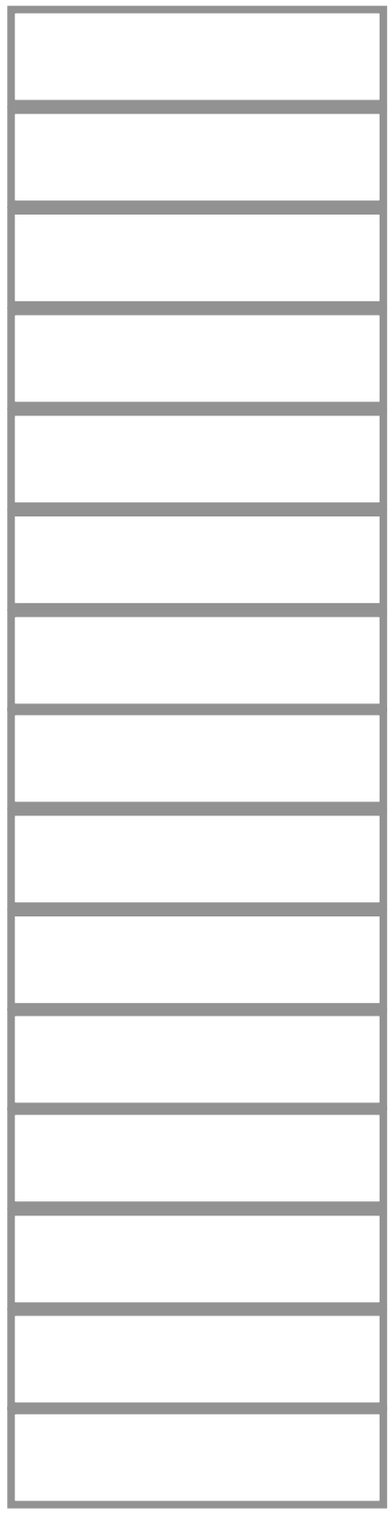
global response buffer



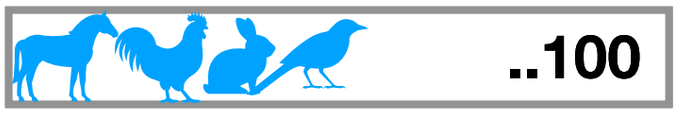
modified <— — — — — — — —response cache lines— — — — — — — —> shared

shared <—-requests-—> modified





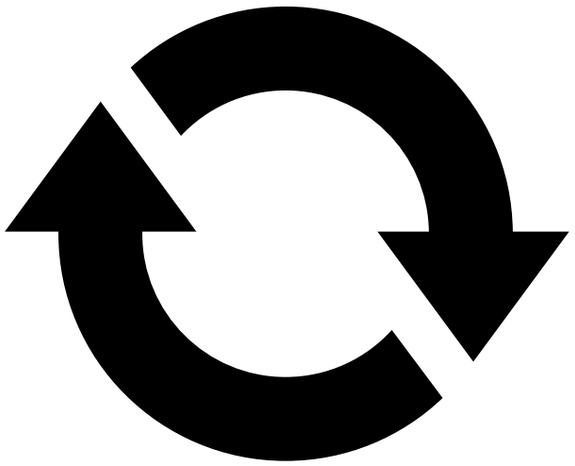
local response buffer

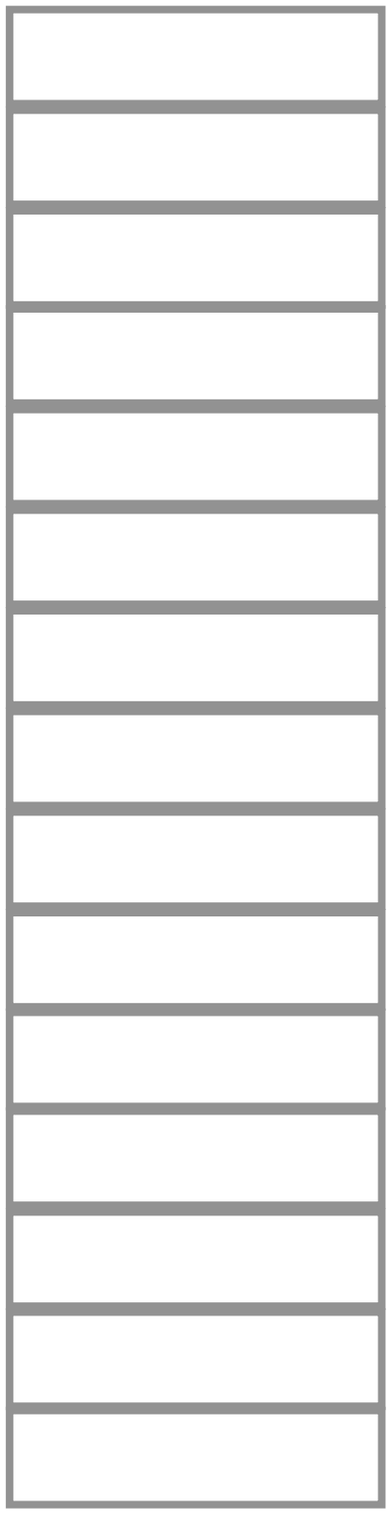


global response buffer

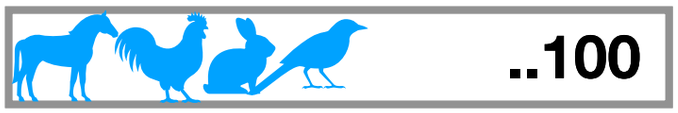
modified <— — — — — — — —response cache lines— — — — — — — —> shared

shared <—-requests-—> modified

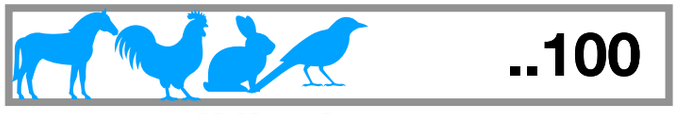




local response buffer

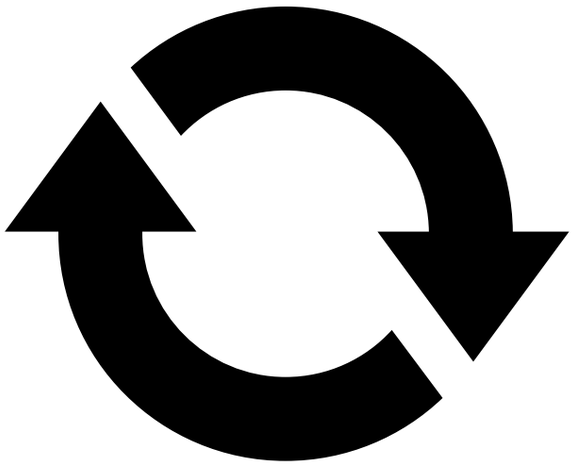


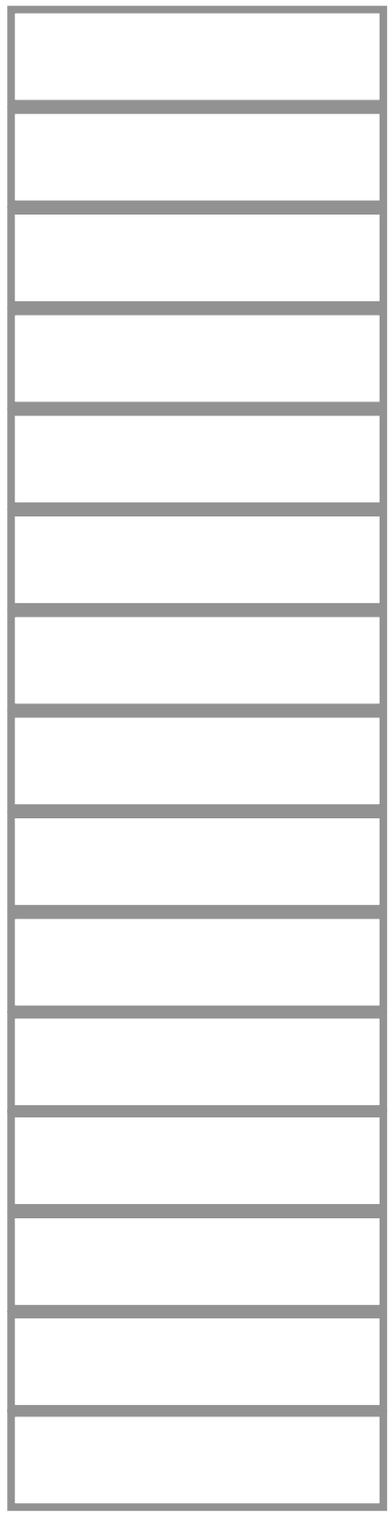
global response buffer



modified <— — — — — — — —response cache lines— — — — — — — —> shared

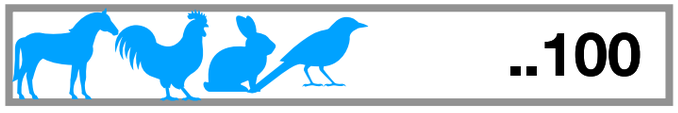
shared <—-requests-—> modified





local response buffer

global response buffer



modified <— — — — — — — —response cache lines— — — — — — — —> shared

performance evaluation

evaluation systems

4×16-core Xeon E5-4660, Broadwell, 2.2 GHz

4×8-core Xeon E5-4620, Sandy Bridge-EP, 2.2 GHz

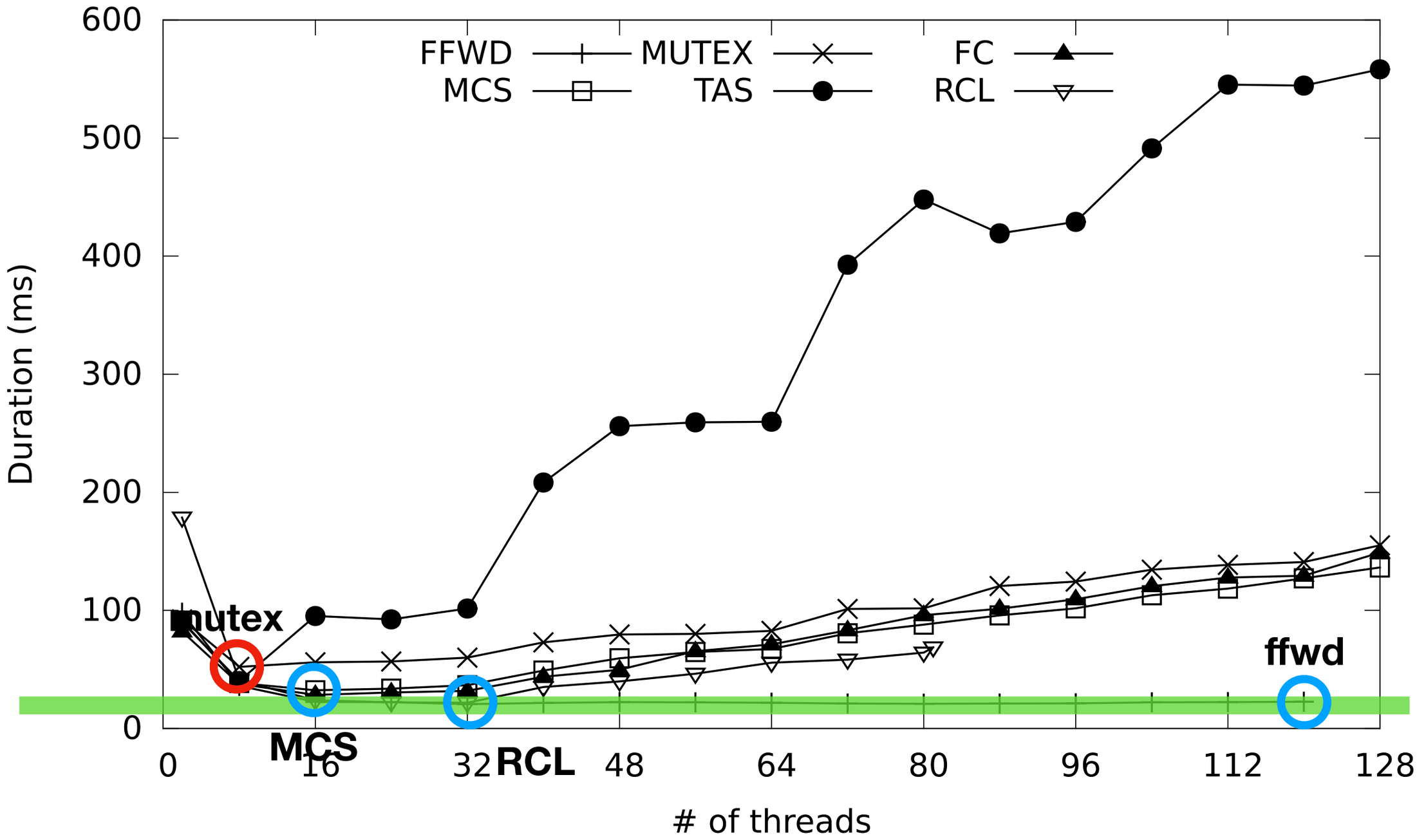
4×8-core Xeon E7-4820, Westmere-EX, 2.0 GHz

4×8-core AMD Opteron 6378, Abu Dhabi, 2.4 GHz

application benchmarks

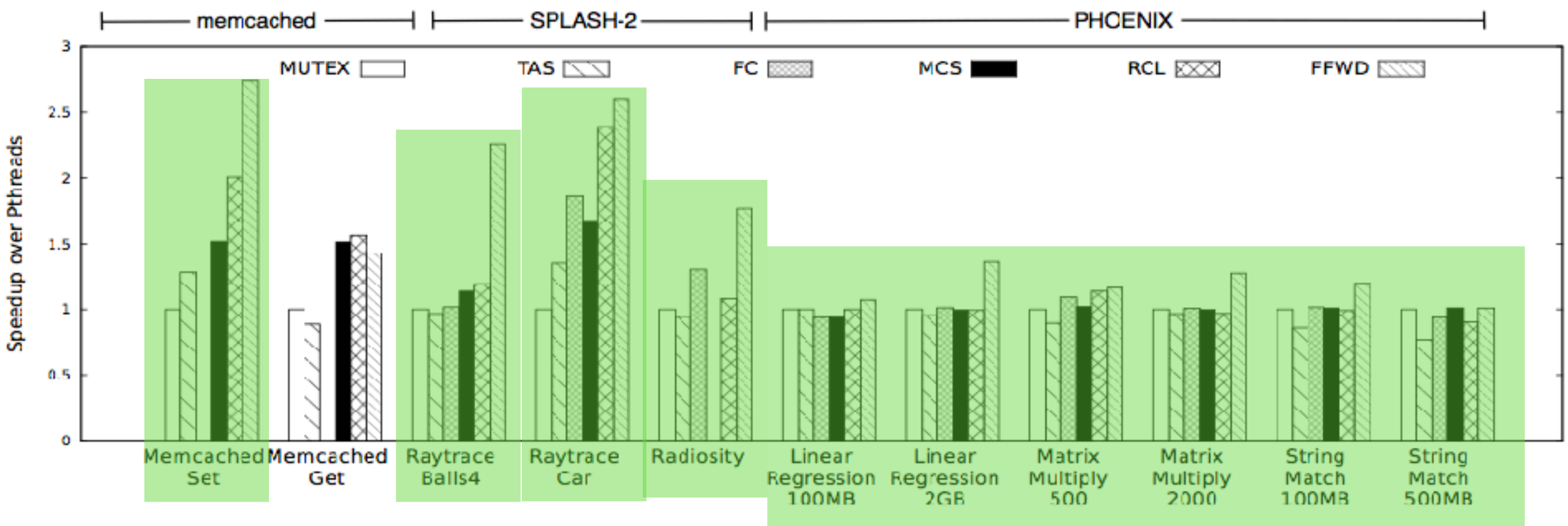
- Same benchmarks as in Lozi et al. (RCL) [USENIX ATC'12]
  - programs that spend large % of time in critical sections
- Except BerkeleyDB - ran out of time

raytrace-car (SPLASH-2)



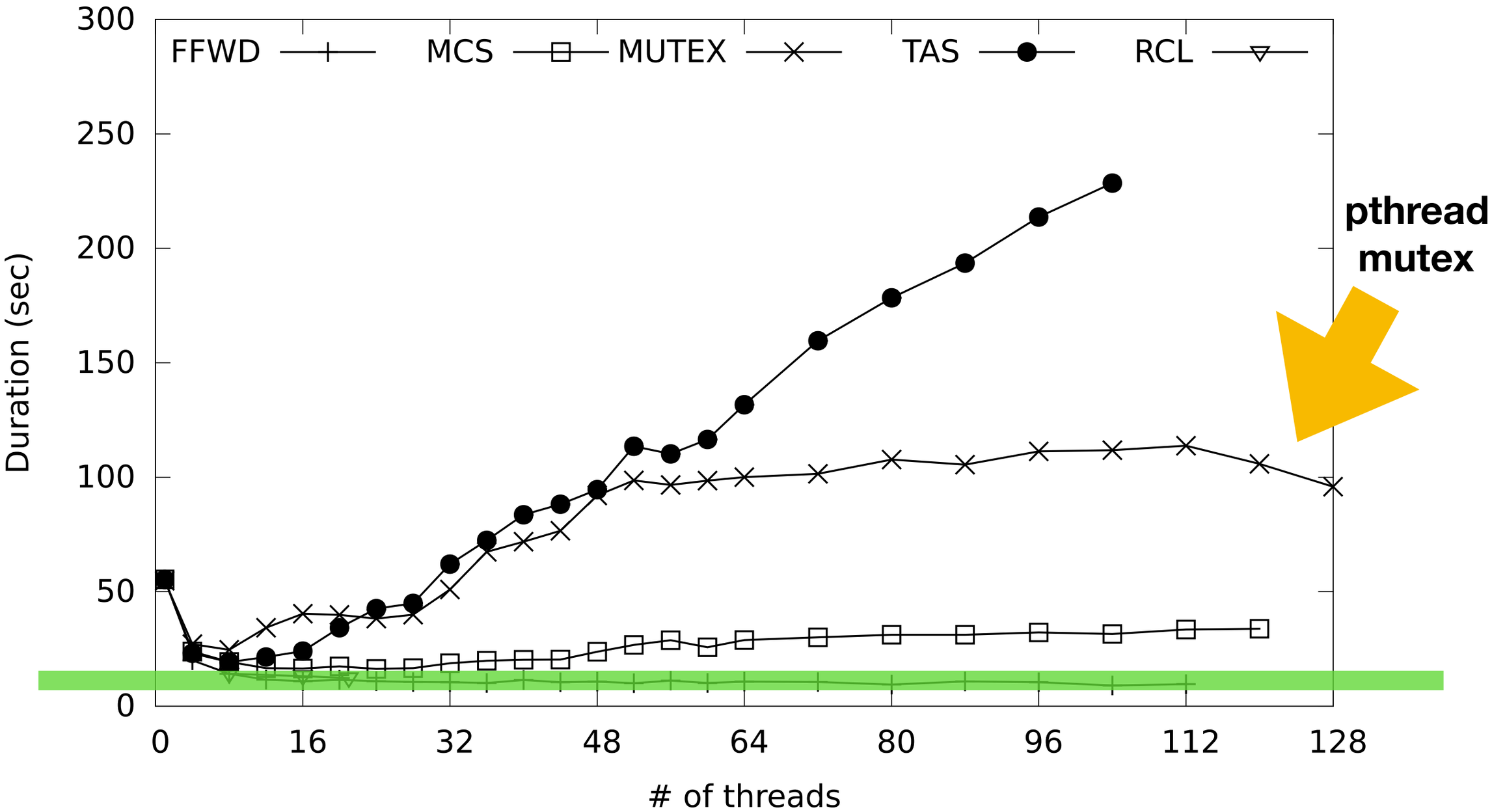
RCL experienced correctness issues above 82 threads.

application benchmarks



- comparing best performance (any thread count) for all methods
- up to 2.5x improvement over pthreads, any thread count
  - 10+ times speedup at max thread count

memcached-set

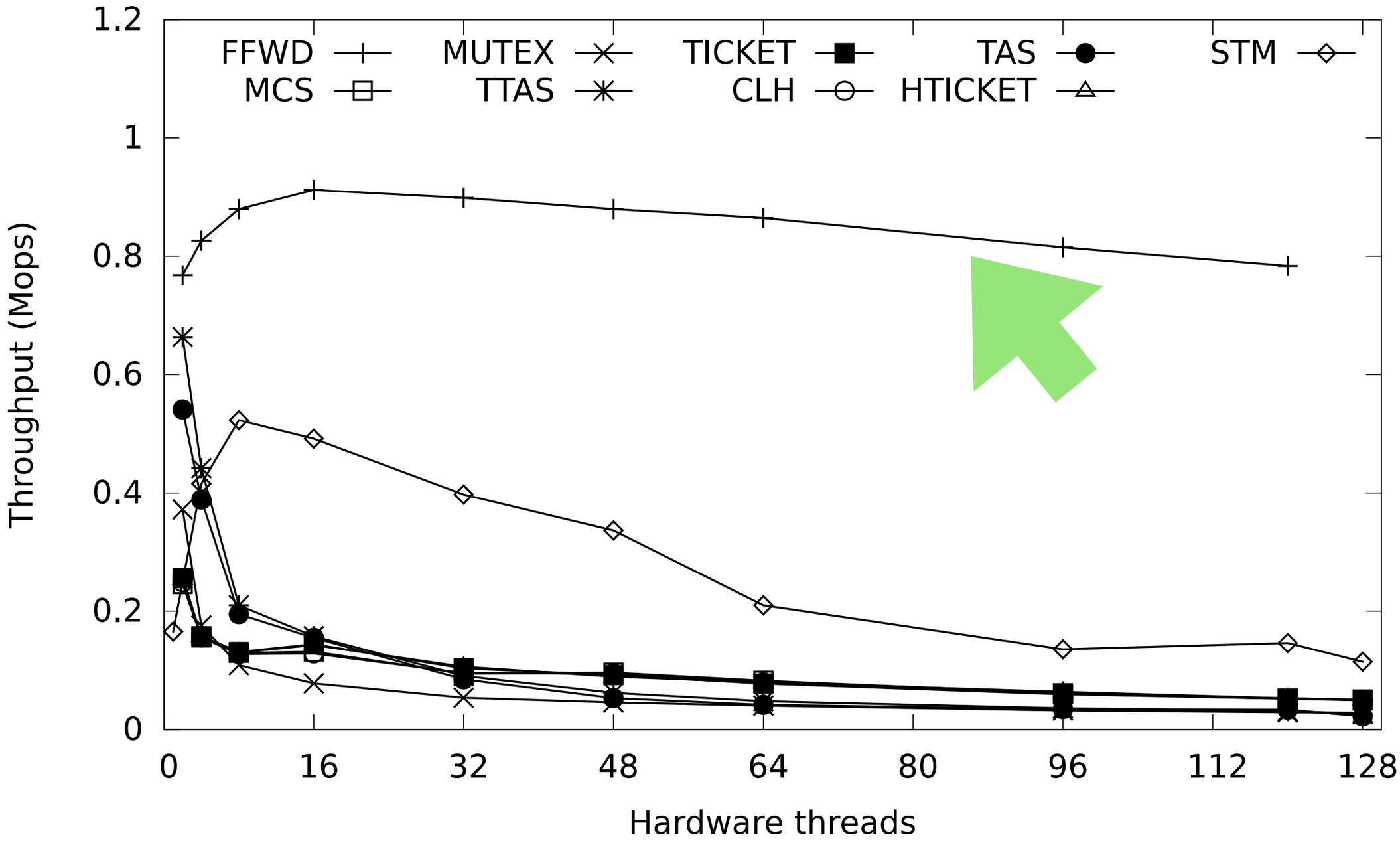


RCL experienced correctness issues above 24 threads.
We did not get Flat Combining to work.

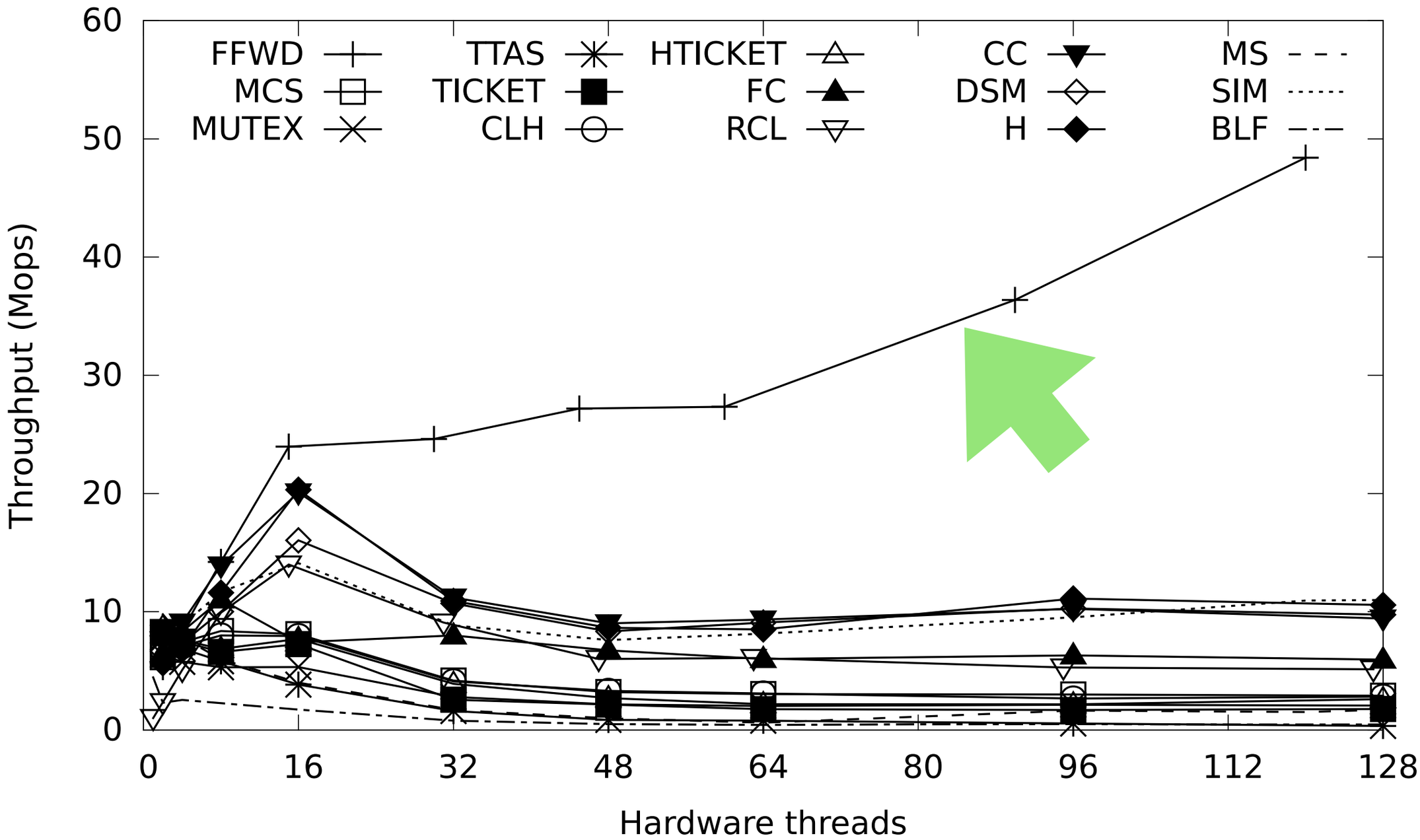
microbenchmarks

- **ffwd is much faster on largely sequential data structures**
  - linked list (coarse locking), stack, queue
  - fetch and add, for few shared variables
- **for highly concurrent data structures, ffwd falls behind when the lock contention is low**
  - fetch and add, with many shared variables
  - hashtable
- **for concurrent data structures with long query times, ffwd keeps up, but is not a clear leader**
  - lazy linked list
  - binary search tree

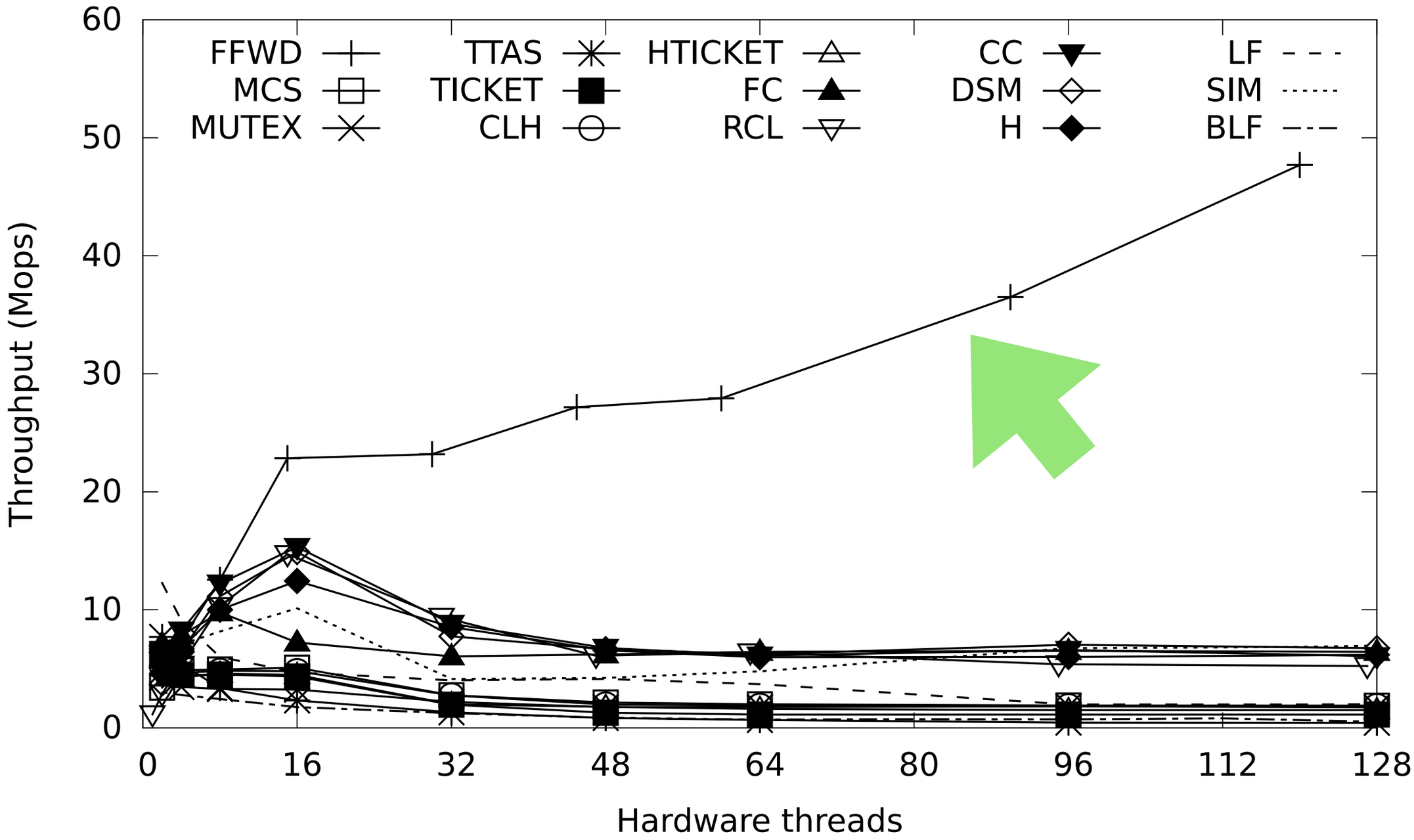
naïve 1024-node linked-list, coarse-grained locking



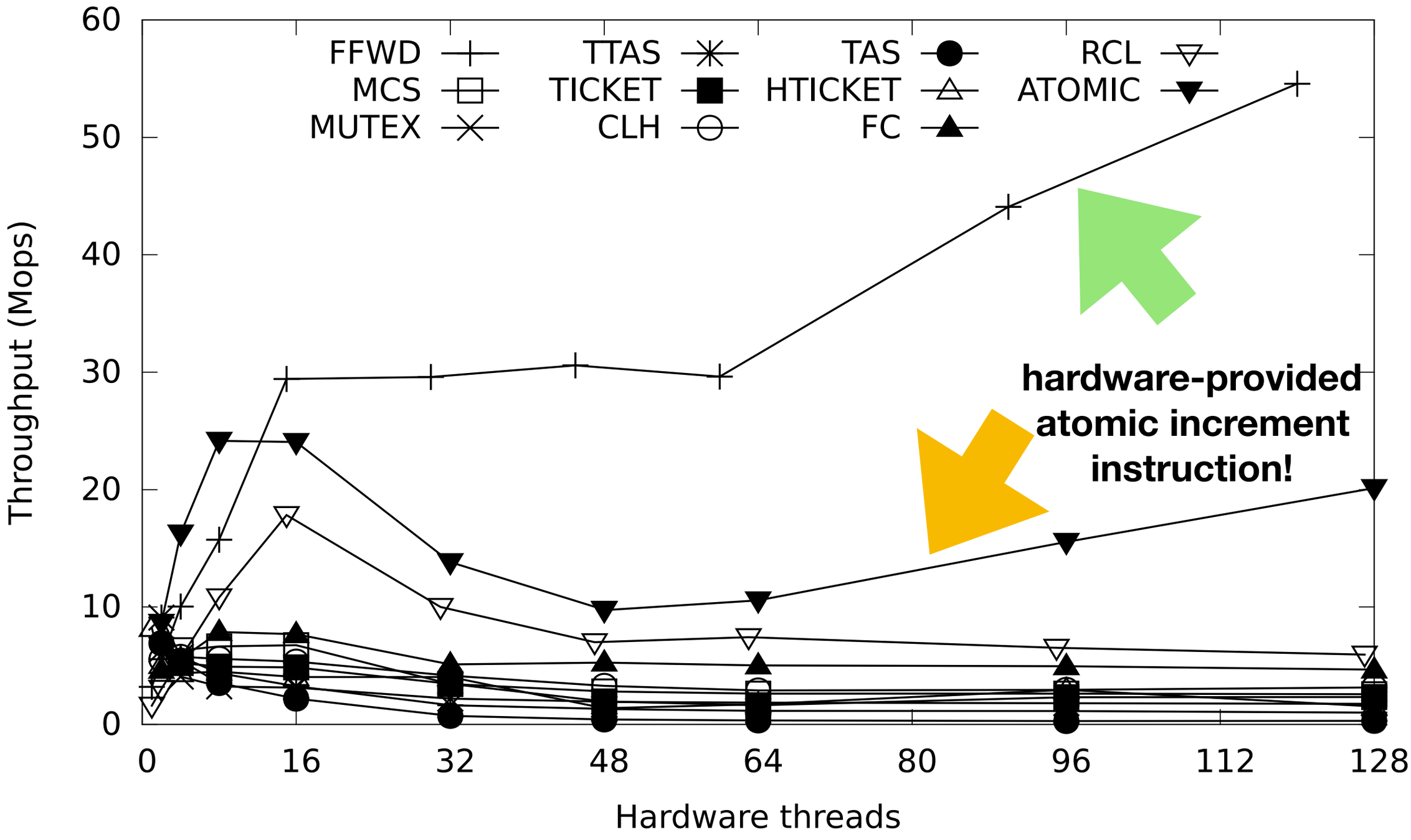
two-lock queue



stack

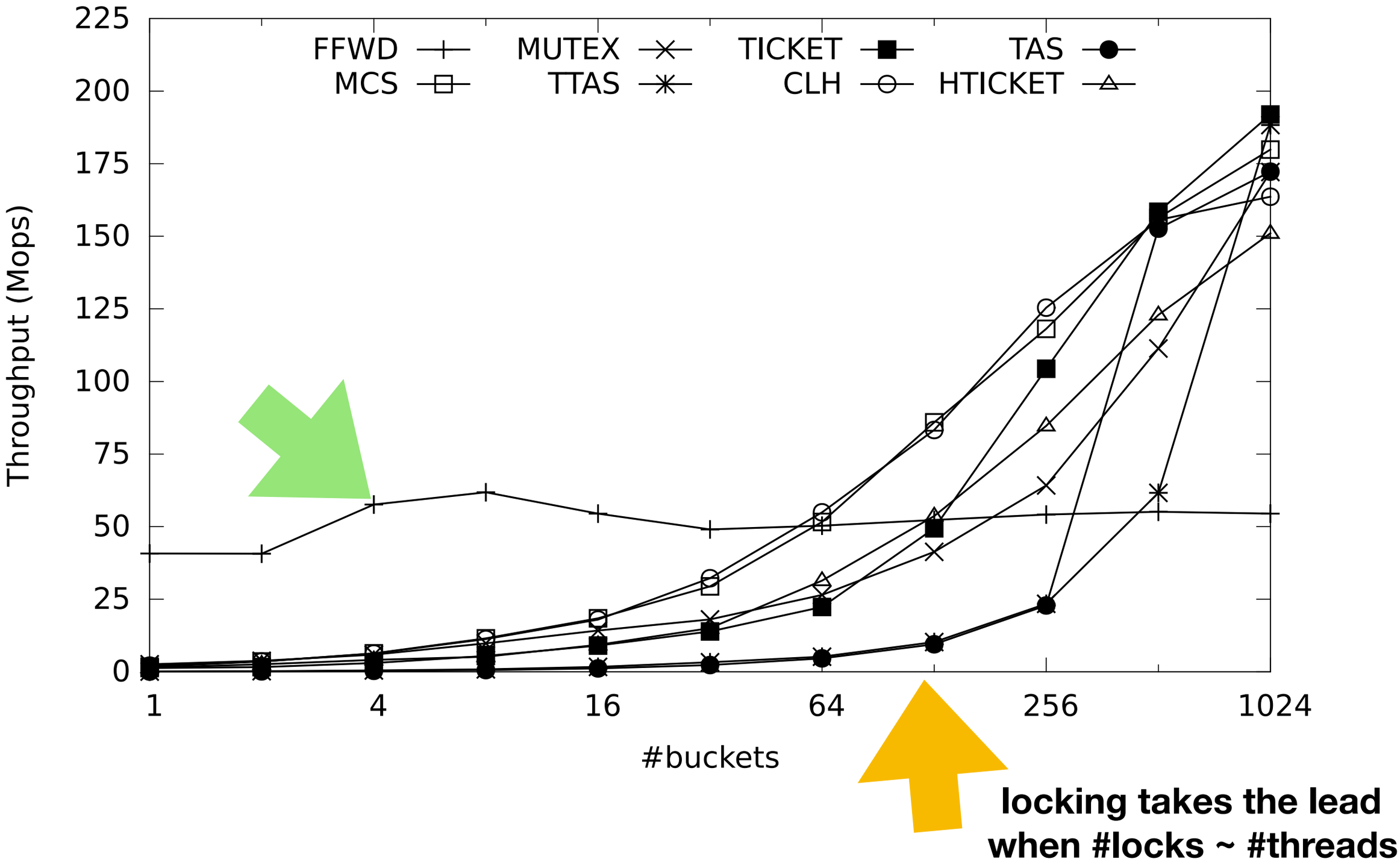

FFWD
MCS
MUTEX
TTAS
TICKET
CLH
HTICKET
FC
RCL
CC
DSM
H
LF
SIM
BLF
Throughput (Mops)
0
10
20
30
40
50
60
0
16
32
48
64
80
96
112
128
Hardware threads

fetch-and-add, 1 variable

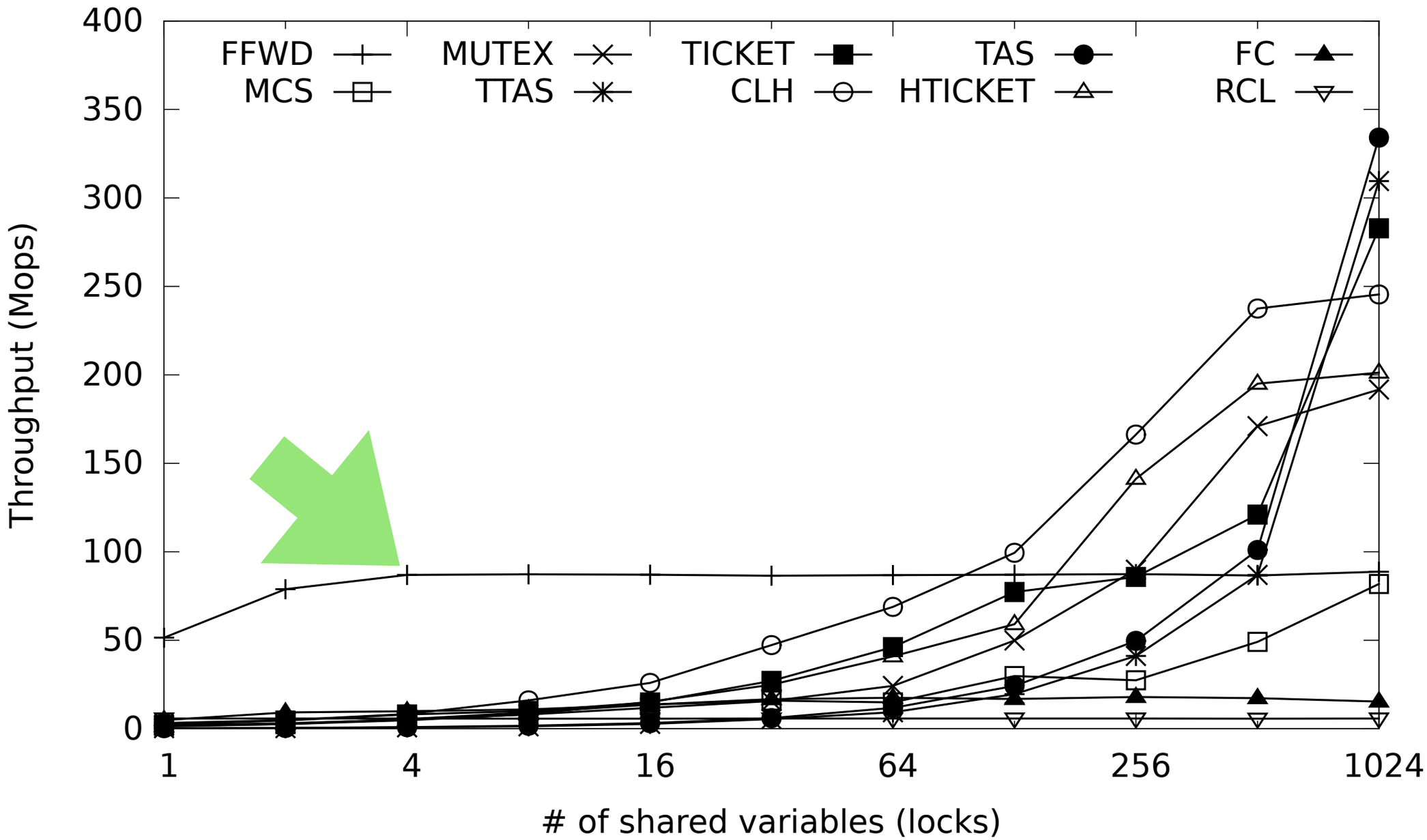


- **ffwd is much faster on largely sequential data structures**
  - naïve linked list, stack, queue
  - fetch and add, for few shared variables
- **for highly concurrent data structures, fwd falls behind when there are many locks**
  - fetch and add, with many shared variables
  - hashtable
- **for concurrent data structures with long query times, ffwd keeps up, but is not a clear leader**
  - lazy linked list
  - binary search tree

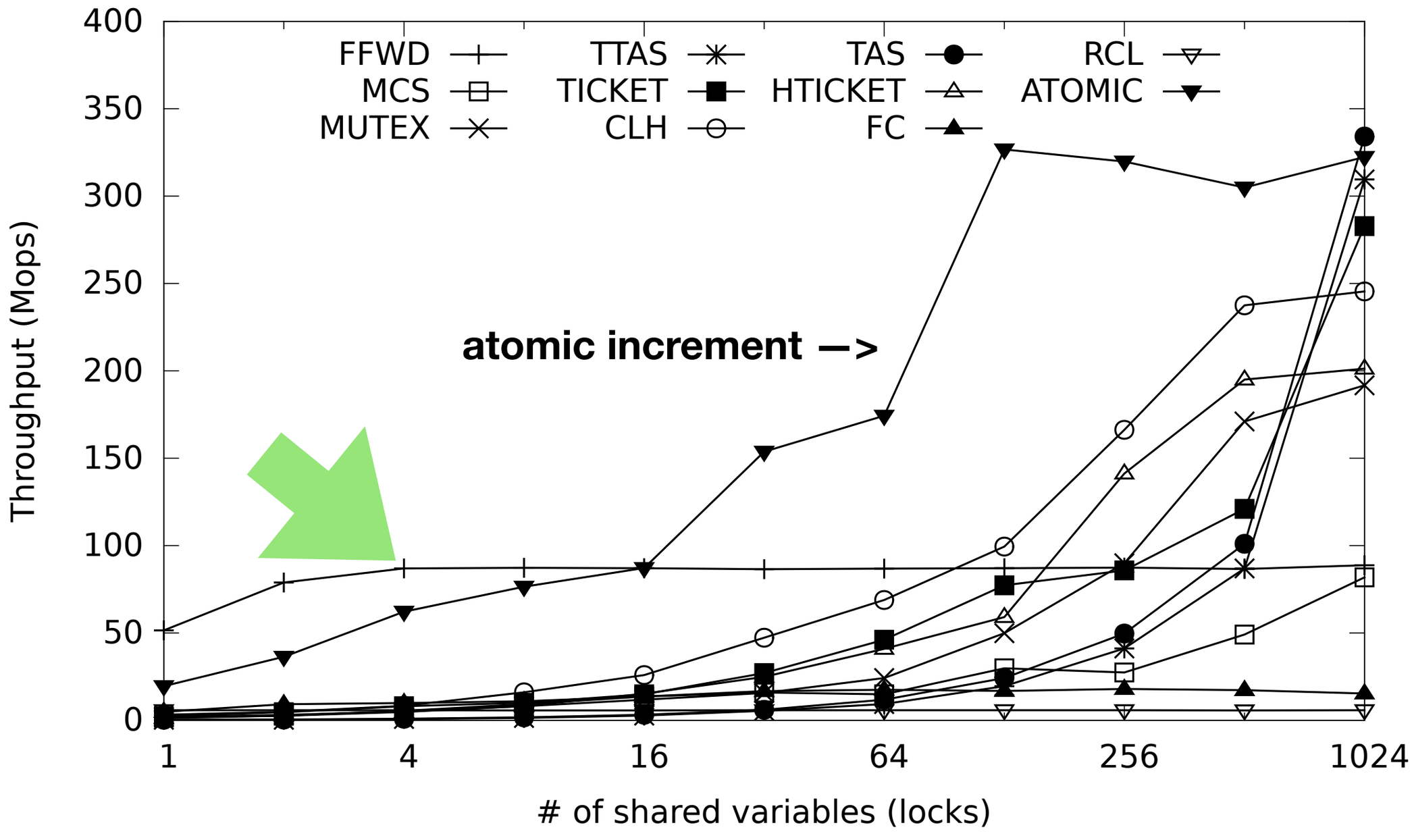
128-thread hash table



fetch-and-add, 128 threads

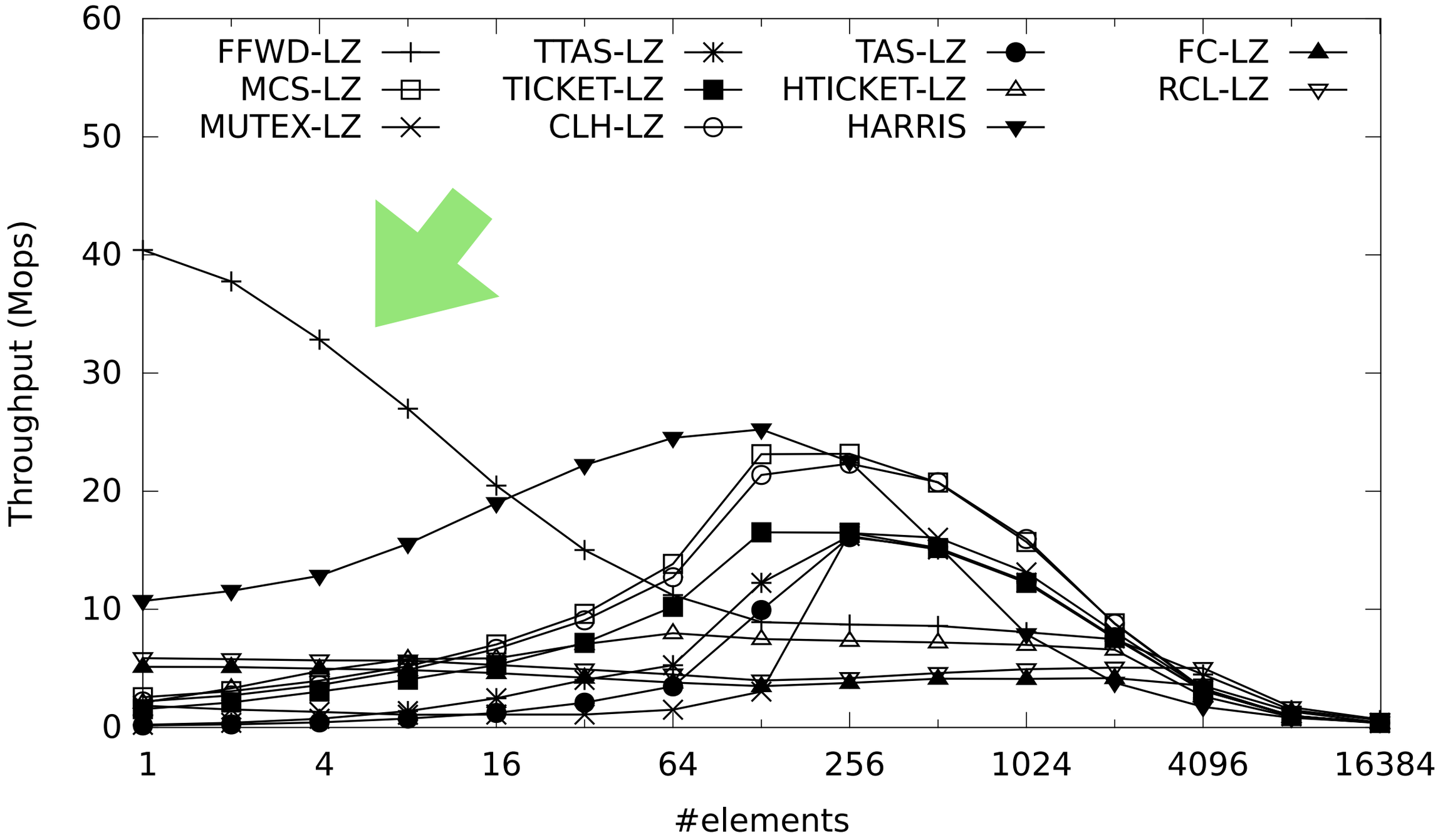


fetch-and-add, 128 threads

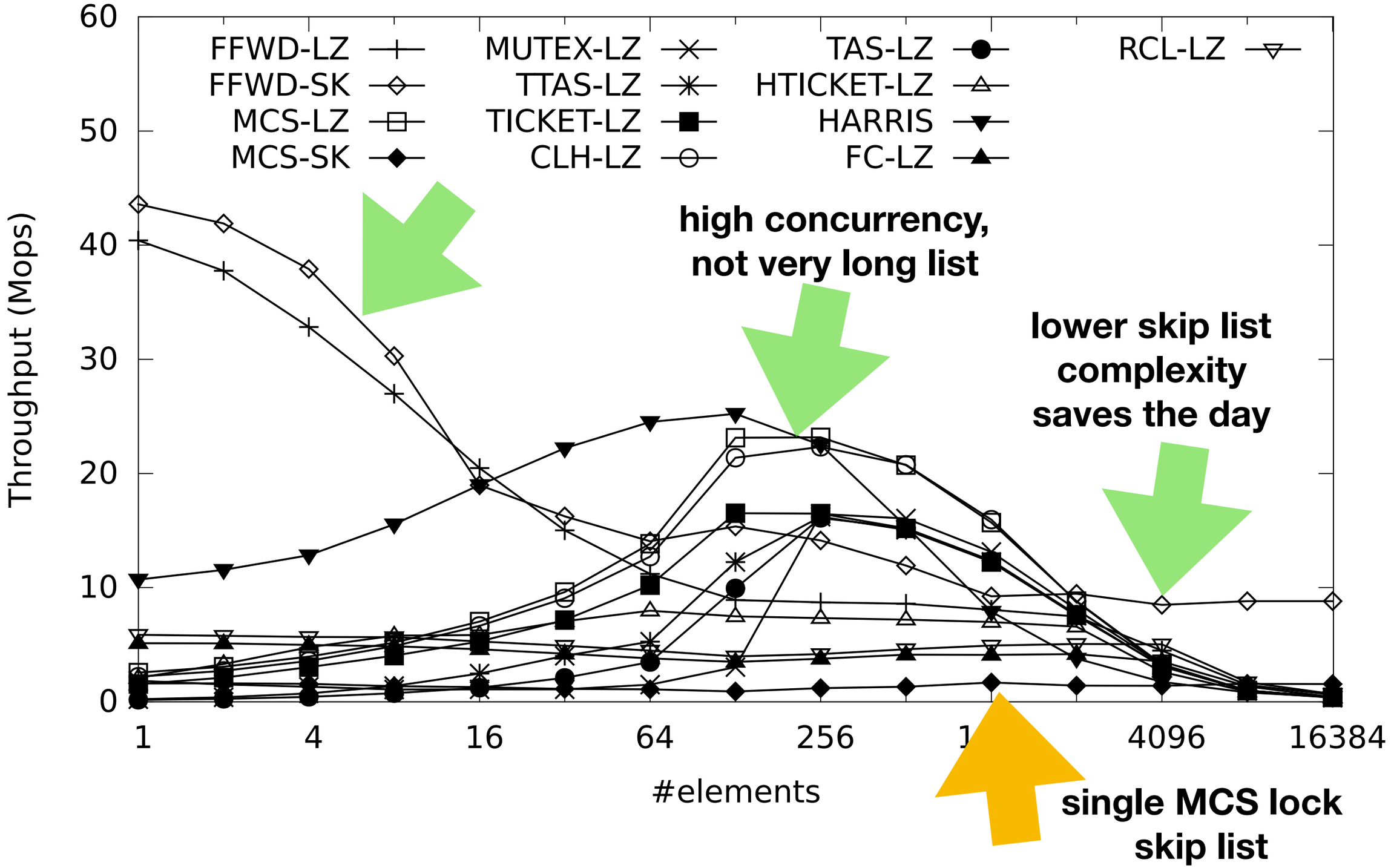


- **ffwd is much faster on largely sequential data structures**
  - naïve linked list, stack, queue
  - fetch and add, for few shared variables
- **for highly concurrent data structures, once lock# is similar to thread#, fwd falls behind**
  - fetch and add, with many shared variables
  - hashtable
- **for concurrent data structures with long query times, ffwd keeps up, but is not a clear leader**
  - lazy linked list
  - binary search tree

128-thread
lazy concurrent lists



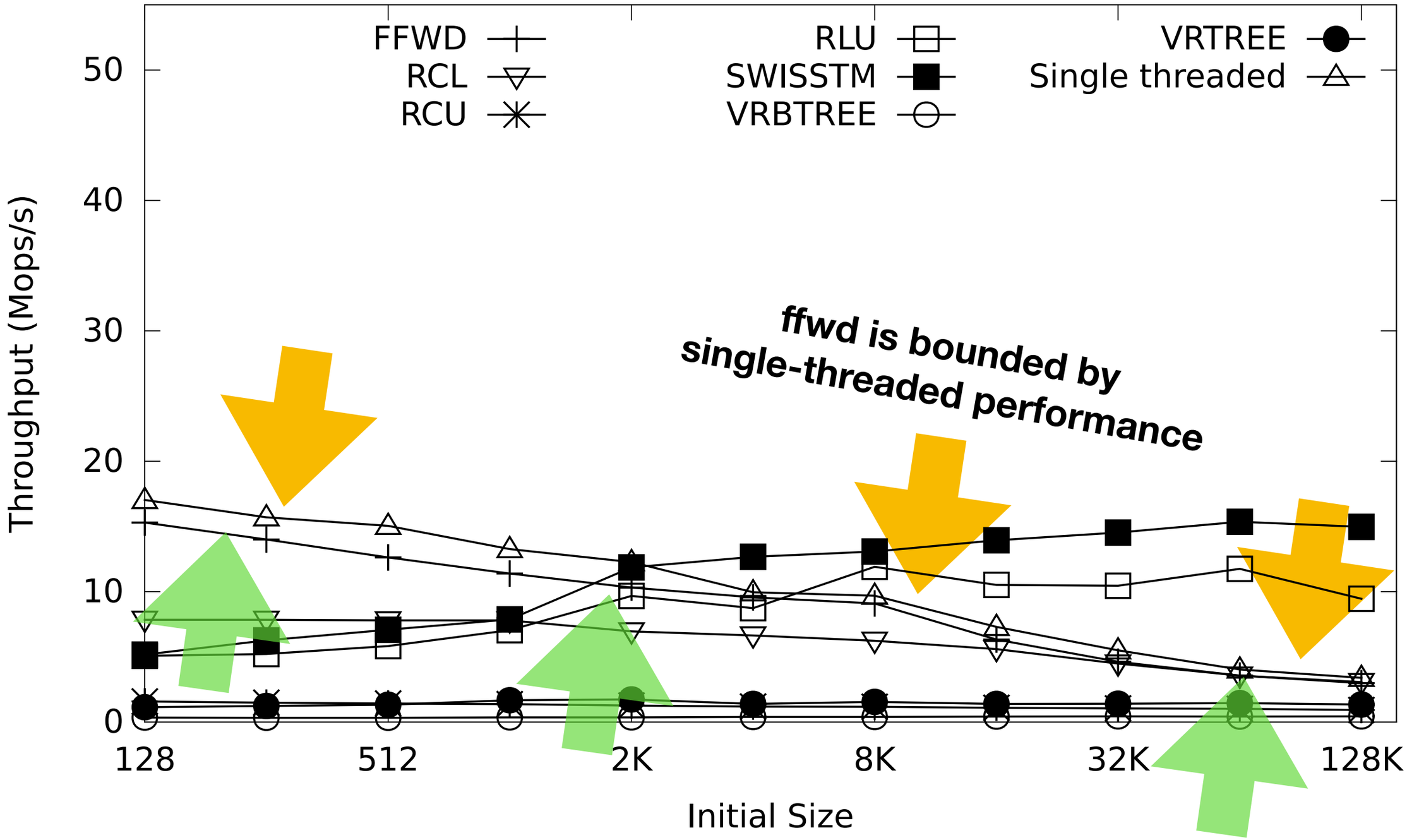
128-thread
lazy (LZ) + skip (SK) lists



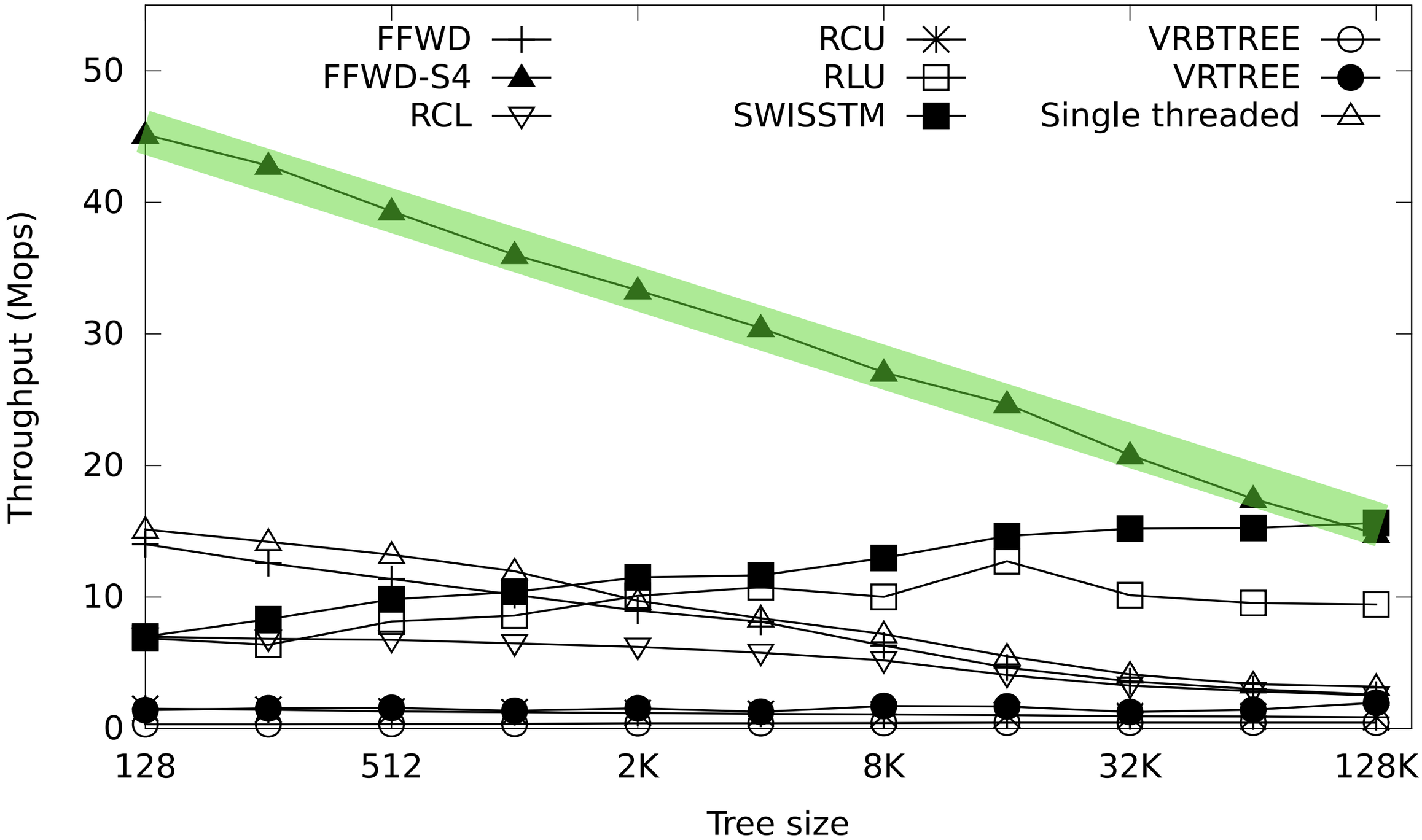
binary search tree

- simple, unbalanced tree
- 50% queries, 50% updates
- all tree operations delegated for ffwd/RCL

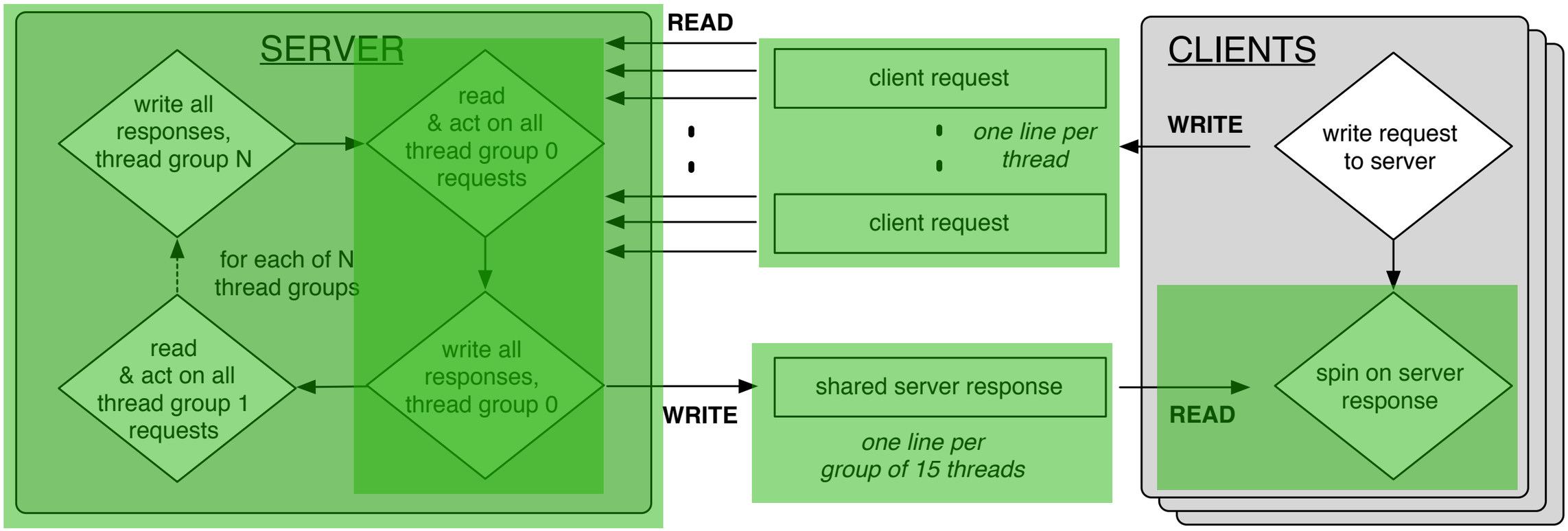
128-thread binary search tree



128-thread tree
+ 4-way sharding ffwd



what makes ffwd so fast?



- requests are virtually un-contended, contiguous in memory
  - = happy server hardware pre-fetcher
- buffered responses on server
  - 15 responses in one contiguous copy
- 2 modified cache-lines instead of 15
- responses are read-only on the client
  - response line never leaves the server L1
- very light-weight processing on the server
  - plenty of hand-tuning

Why isn’t it even faster?

- Link bandwidth is 300 cache lines per link —> **300+ Mops**
- Latency suggests 2.5 Mops/client. 120 clients —> **300 Mops**
- Why are we only seeing 55 Mops?
- Processing limit? 55 Mops = 40 cycles per operation
- Insufficient concurrency: round-trip bandwidth-delay product is 120 cache lines
  - server store / load buffers, reorder window size?

using ffwd

- Free C library available now (Rust is on the way)
- Some current limitations:
  - delegated functions cannot, in turn, delegate functions
  - delegated functions typically should not block (nor acquire locks)
  - up to 6, 64-bit parameters
  - currently assume one client per hardware thread

related work

- Remote Core Locking [Lozi, USENIX ATC’12]
- Barrelfish - delegation-based OS [Baumann, SOSP’09]
- Flat Combining [Hendler, SPAA’10]
- Log-based node replication [Calciu, ASPLOS’17]

in conclusion

- delegation is (much) faster than you thought
- it is easy to use, and has many attractive applications

- similar results on
  Intel Broadwell
  Intel Sandy Bridge
  Intel Westmere-EX, and
  AMD Abu Dhabi

questions?

- UIC has **many open CS faculty positions** this year, all areas

THE UNIVERSITY OF ILLINOIS AT CHICAGO

UIC

- *libffwd*, extended paper and more http://github.com/bitslab/ffwd

comparing parallelism

	locking	delegation
critical section	**#locks**	**#servers**
communication latency	**#locks**	**#clients**