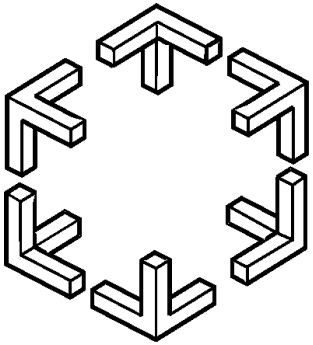


Cache-Aware Lock-Free Queues for Multiple Producers/Consumers and Weak Memory Consistency

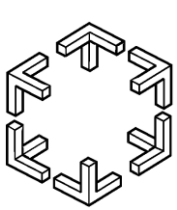
Anders Gidenstam

Håkan Sundell

Philippas Tsigas

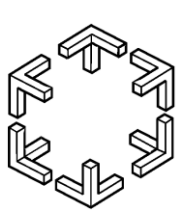
School of business and informatics
University of Borås

Distributed Computing and Systems group,
Department of Computer Science and Engineering,
Chalmers University of Technology

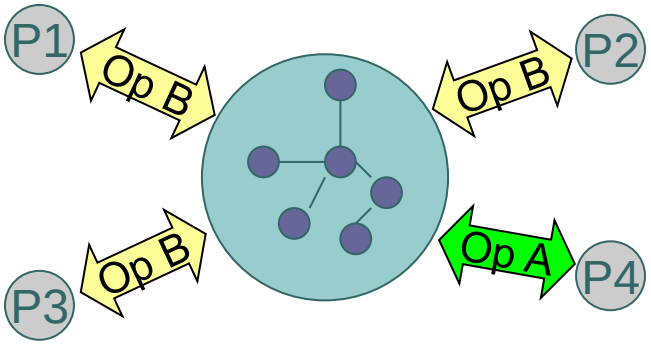


Outline

- Introduction
  - Lock-free synchronization
  - The Problem & Related work
- The new lock-free queue algorithm
- Experiments
- Conclusions



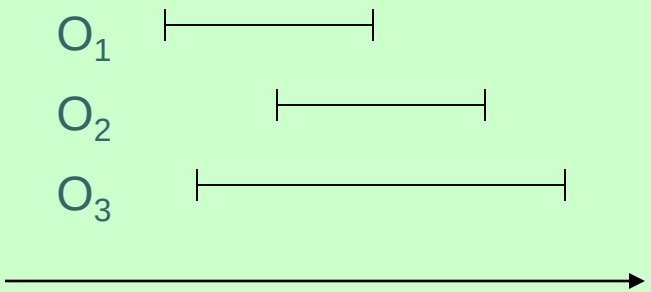
Synchronization on a shared object



- Lock-free synchronization
  - Concurrent operations without enforcing mutual exclusion
  - Avoids:
    - Blocking (or busy waiting), convoy effects, priority inversion and risk of deadlock
  - *Progress Guarantee*
    - At least one operation always makes progress

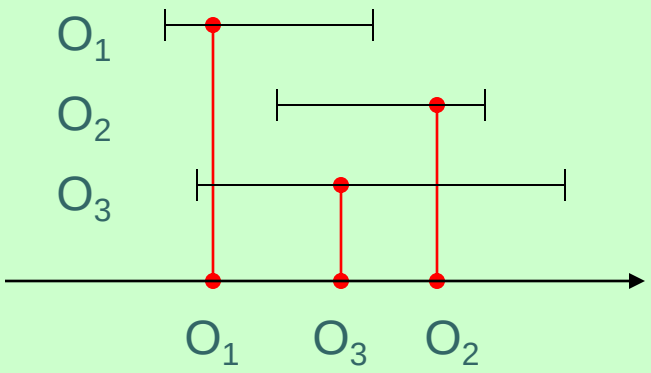
Correctness of a concurrent object

- Desired semantics of a shared data object
  - Linearizability [Herlihy & Wing, 1990]
    - For each operation invocation there must be one single time instant during its duration where the operation appears to take effect.
    - The observed effects should be consistent with a sequential execution of the operations in that order.



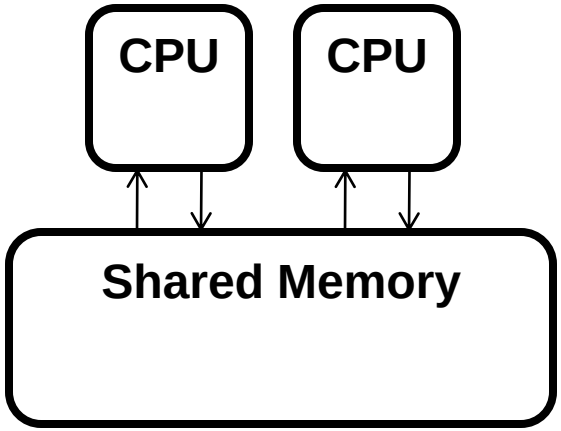
Correctness of a concurrent object

- Desired semantics of a shared data object
  - Linearizability [Herlihy & Wing, 1990]
    - For each operation invocation there must be one single time instant during its duration where the operation appears to take effect.
    - The observed effects should be consistent with a sequential execution of the operations in that order.



System Model

- Processes can read/write single memory words
- Synchronization primitives
  - Built into CPU and memory system
  - Atomic read-modify-write (i.e. a critical section of one instruction)
    - Examples: Compare-and-Swap, Load-Linked / Store-Conditional

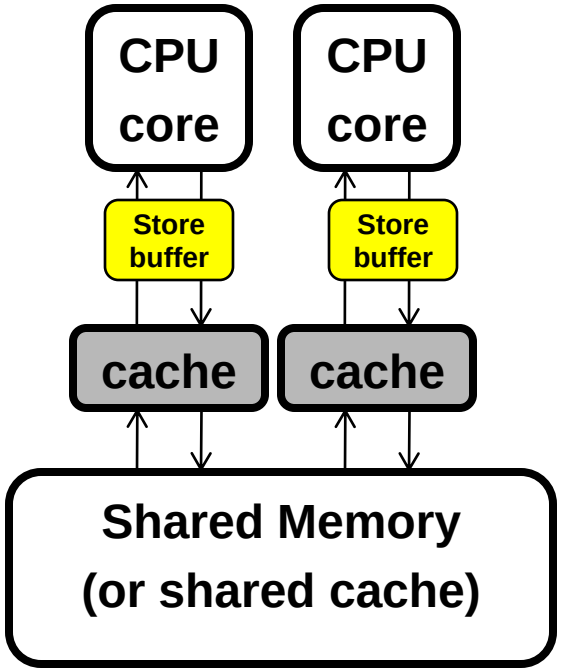


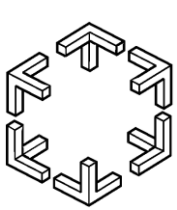
System Model: Memory Consistency

A process'

- Reads/writes may reach memory out of order
- Reads of own writes appear in program order
- Atomic synchronization primitive/instruction
  - Single word Compare-and-Swap
  - Atomic
  - Acts as memory barrier for the process' own reads and writes
    - All own reads/writes before are done before
    - All own reads/writes after are done after

  The affected cache block is held exclusively



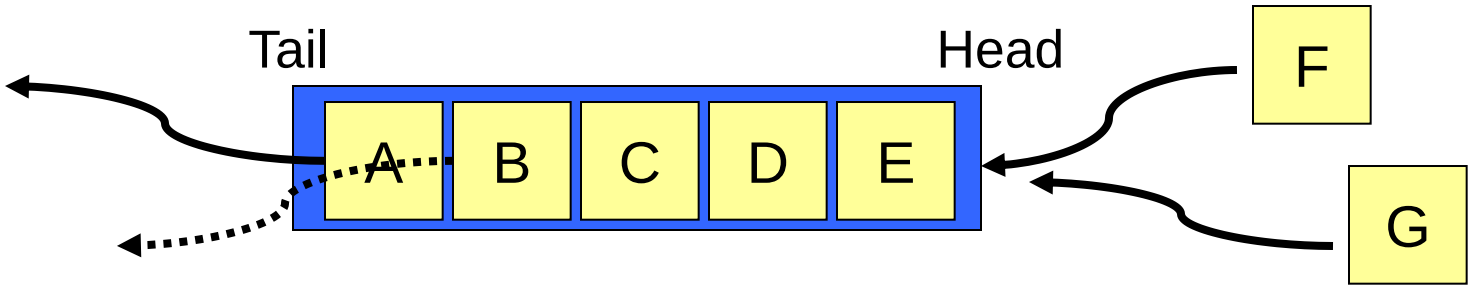


Outline

- Introduction
  - Lock-free synchronization
  - The Problem & Related work
- The new lock-free queue algorithm
- Experiments
- Conclusions

The Problem

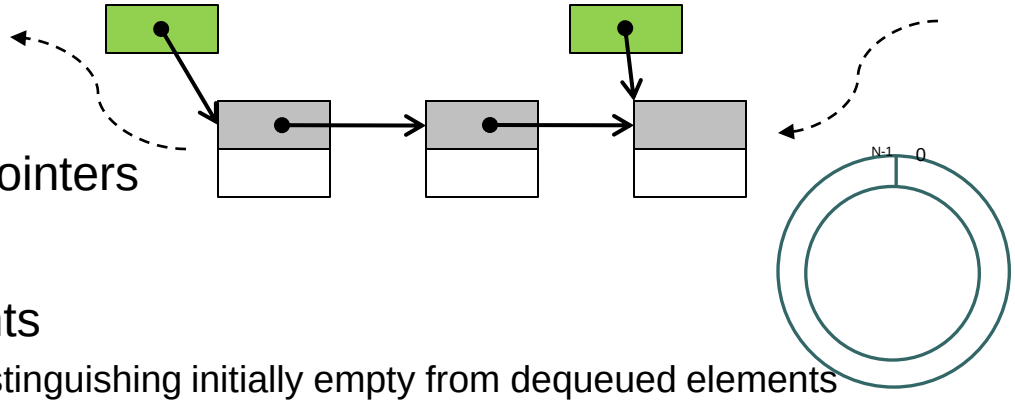
- Concurrent FIFO queue shared data object
  - Basic operations: enqueue and dequeue



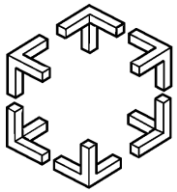
  - Desired Properties
    - Linearizable and Lock-free
    - Dynamic size (maximum only limited by available memory)
    - Bounded memory usage (in terms of live contents)
    - Fast on real systems

Related Work: Lock-free Multi-P/C Queues

- [Michael & Scott, 1996]
  - Linked-list, one element/node
  - Global shared head and tail pointers



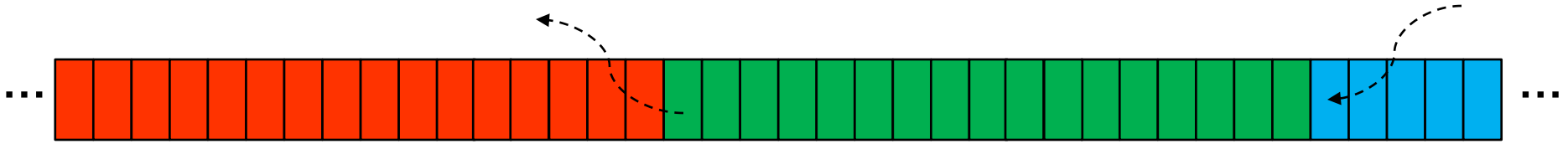
- [Tsigas & Zhang, 2001]
  - Static circular array of elements
    - Two different NULL values for distinguishing initially empty from dequeued elements
  - Global shared head and tail indices, lazily updated
- [Michael & Scott, 1996] + Elimination [Moir, Nussbaum, Shalev & Shavit, 2005]
  - Same as the above + elimination of concurrent pairs of enqueue and dequeue when the queue is near empty
- [Hoffman, Shalev & Shavit, 2007] Baskets queue
  - Linked-list, one element/node
  - Reduces contention between concurrent enqueues after conflict
  - Needs stronger memory management than M&S (SLFRC or Beware&Cleanup)



Outline

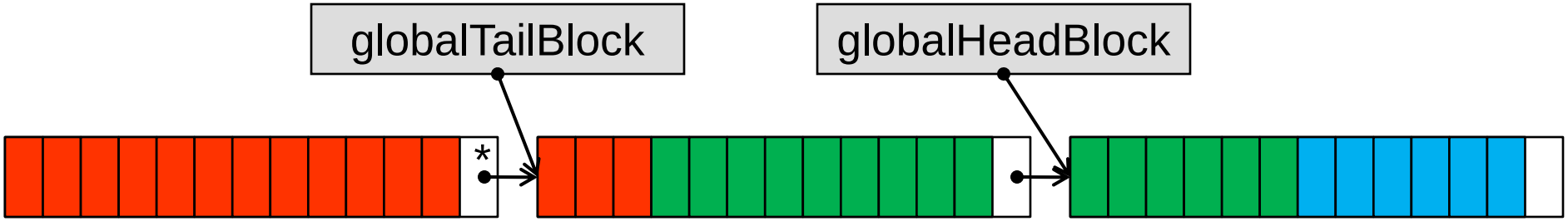
- Introduction
  - Lock-free synchronization
  - The Problem & Related work
- The new lock-free queue algorithm
- Experiments
- Conclusions

The Algorithm



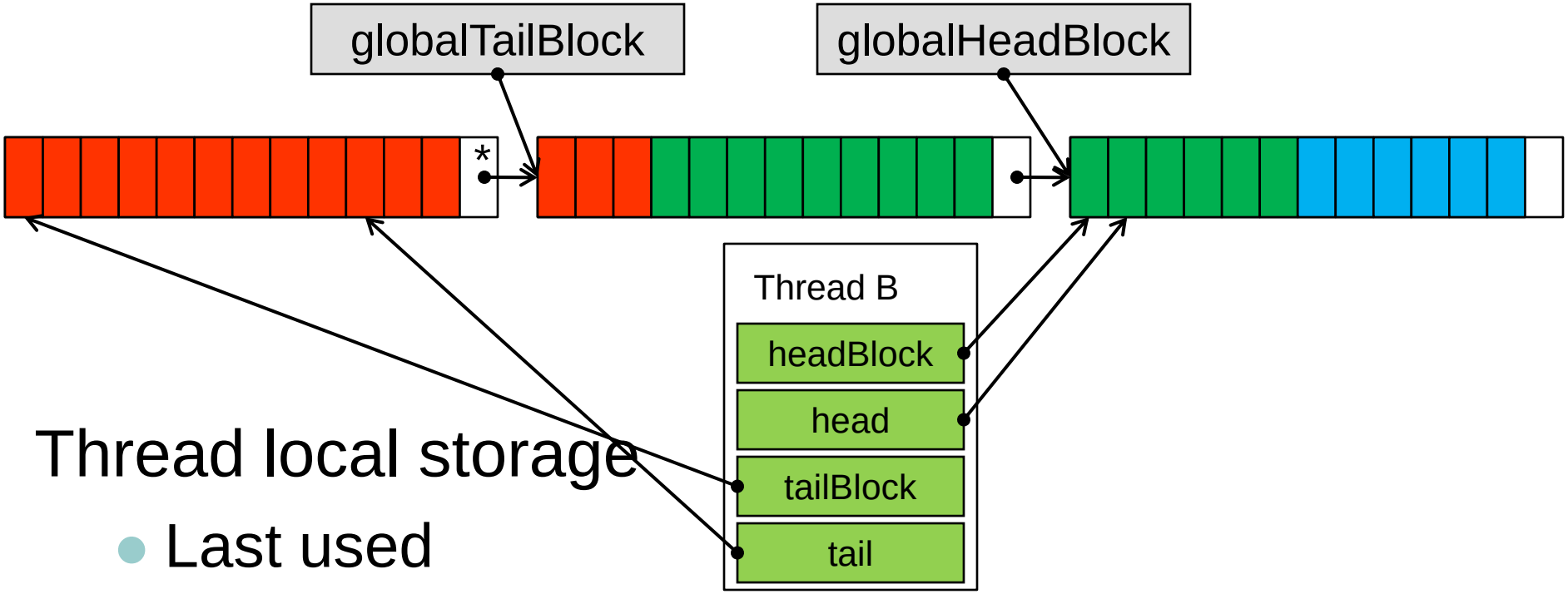
- Basic idea:
  - Cut and unroll the circular array queue
  - Primary synchronization on the elements
    - Compare-And-Swap
      (NULL1 -> Value -> NULL2 avoids the ABA problem)
  - Head and tail both move to the right
    - Need an “infinite” array of elements

The Algorithm



- Basic idea:
  - Creating an “infinite” array of elements.
  - Divide into blocks of elements, and link them together
    - New empty blocks added as needed
    - Emptied blocks are marked deleted and eventually reclaimed
    - Block fields: Elements, next, (filled, emptied flags), deleted flag.
  - Linked chain of dynamically allocated blocks
    - Lock-free memory management needed for safe reclamation!
    - Beware&Cleanup [Gidenstam, Papatriantafilou, Sundell & Tsigas, 2009]

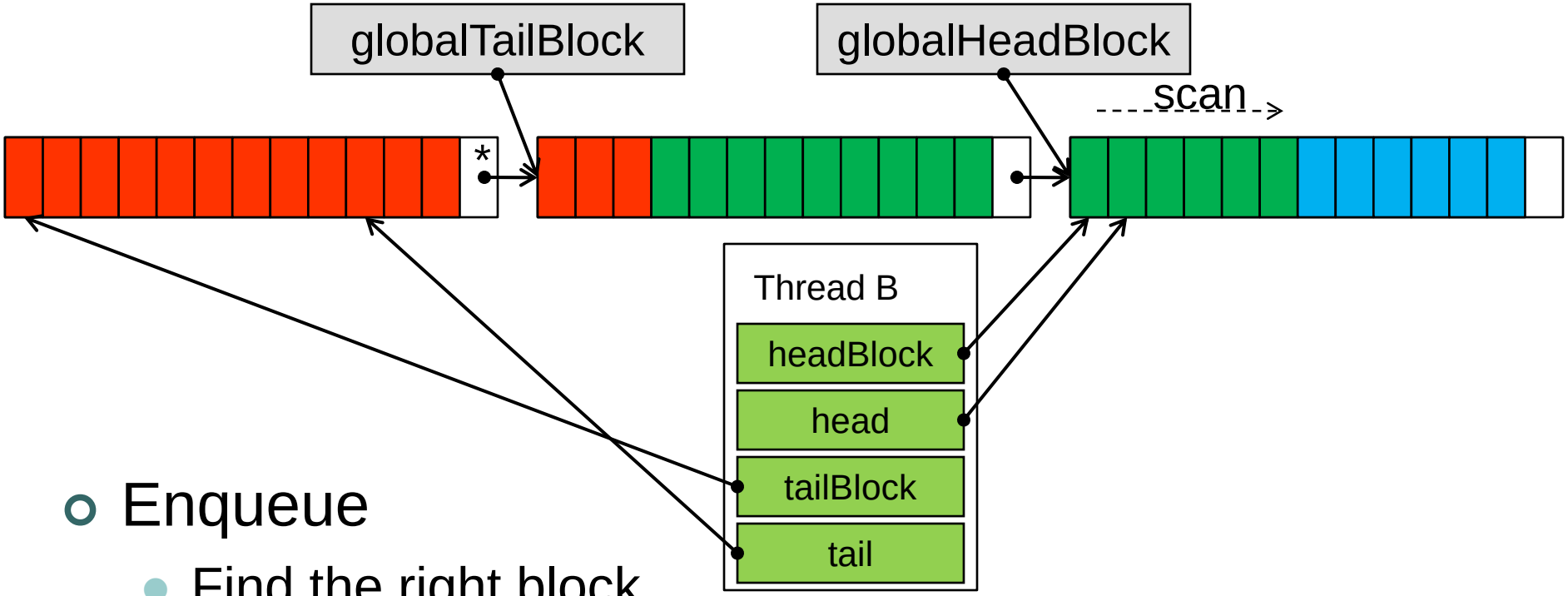
The Algorithm



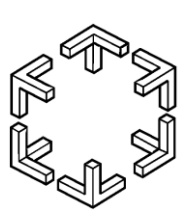
Thread local storage

- Last used
  - Head block/index for Enqueue
  - Tail block/index for Dequeue
- Reduces need to read/update global shared variables

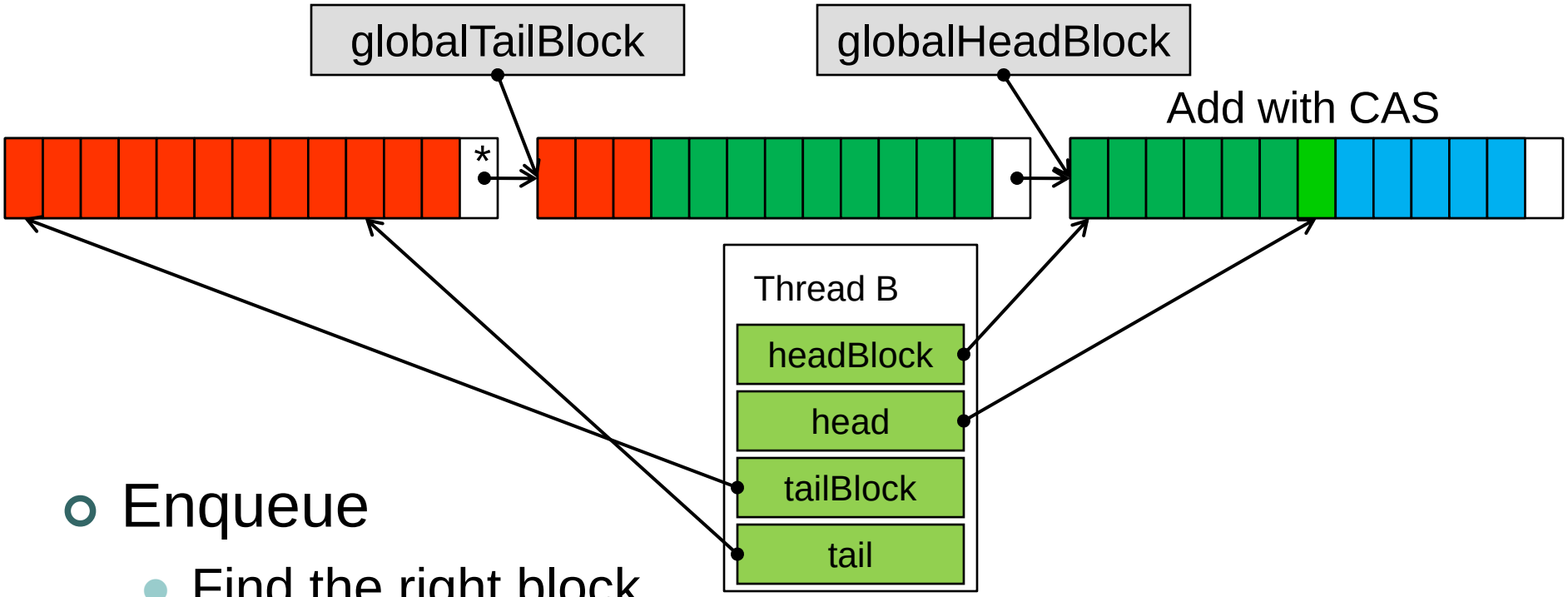
The Algorithm



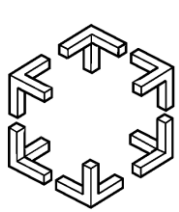
- Enqueue
  - Find the right block
    (first via TLS, then TLS->next or globalHeadBlock)
  - Search the block for the first empty element



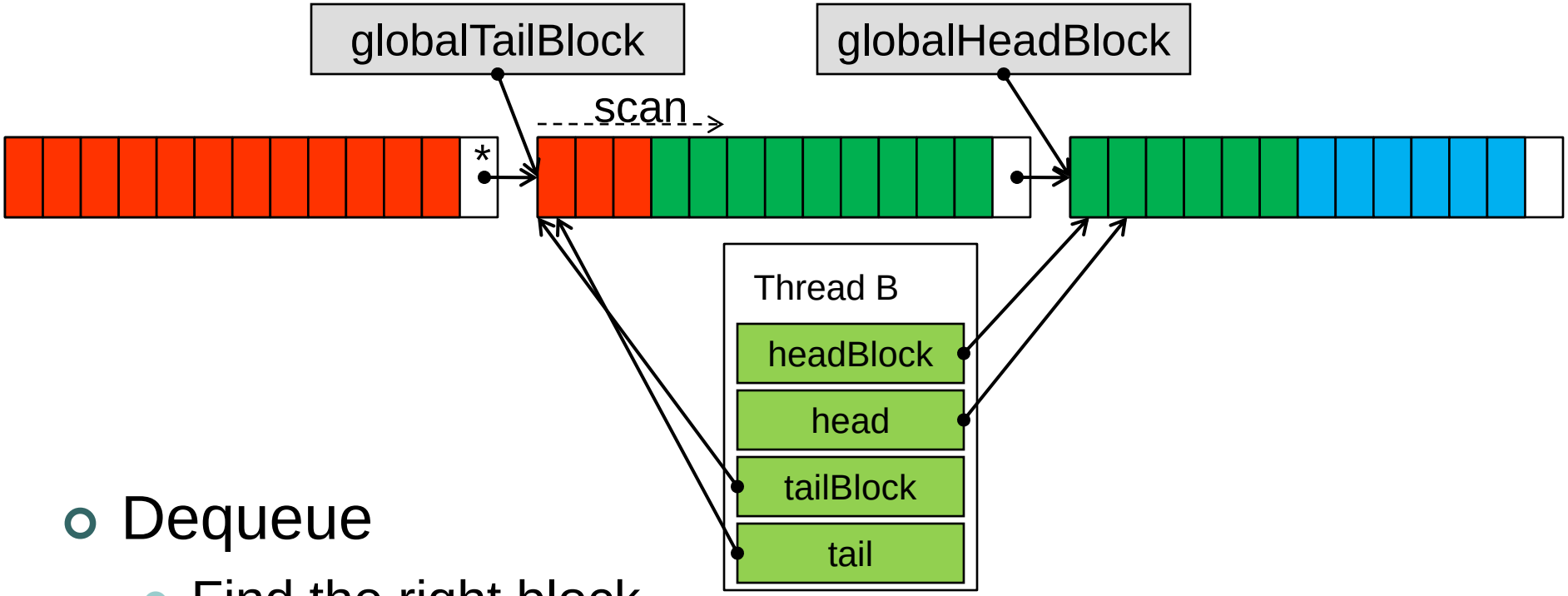
The Algorithm



- Enqueue
  - Find the right block
    (first via TLS, then TLS->next or globalHeadBlock)
  - Search the block for the first empty element
  - Update element with CAS (Also, the linearization point)

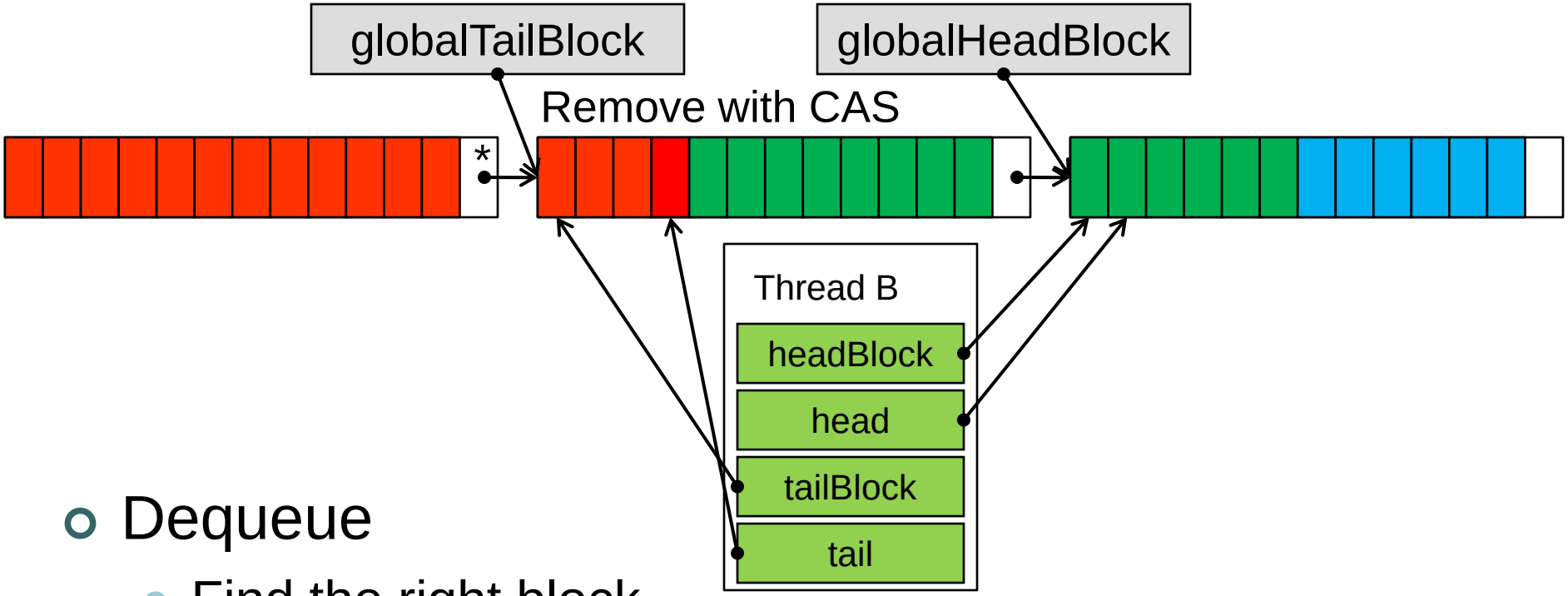


The Algorithm



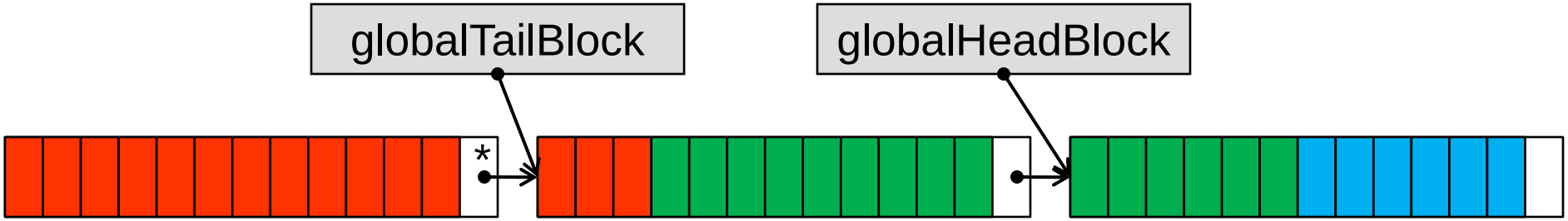
- Dequeue
  - Find the right block
    (first via TLS, then TLS->next or globalTailBlock)
  - Search the block for the first valid element

The Algorithm



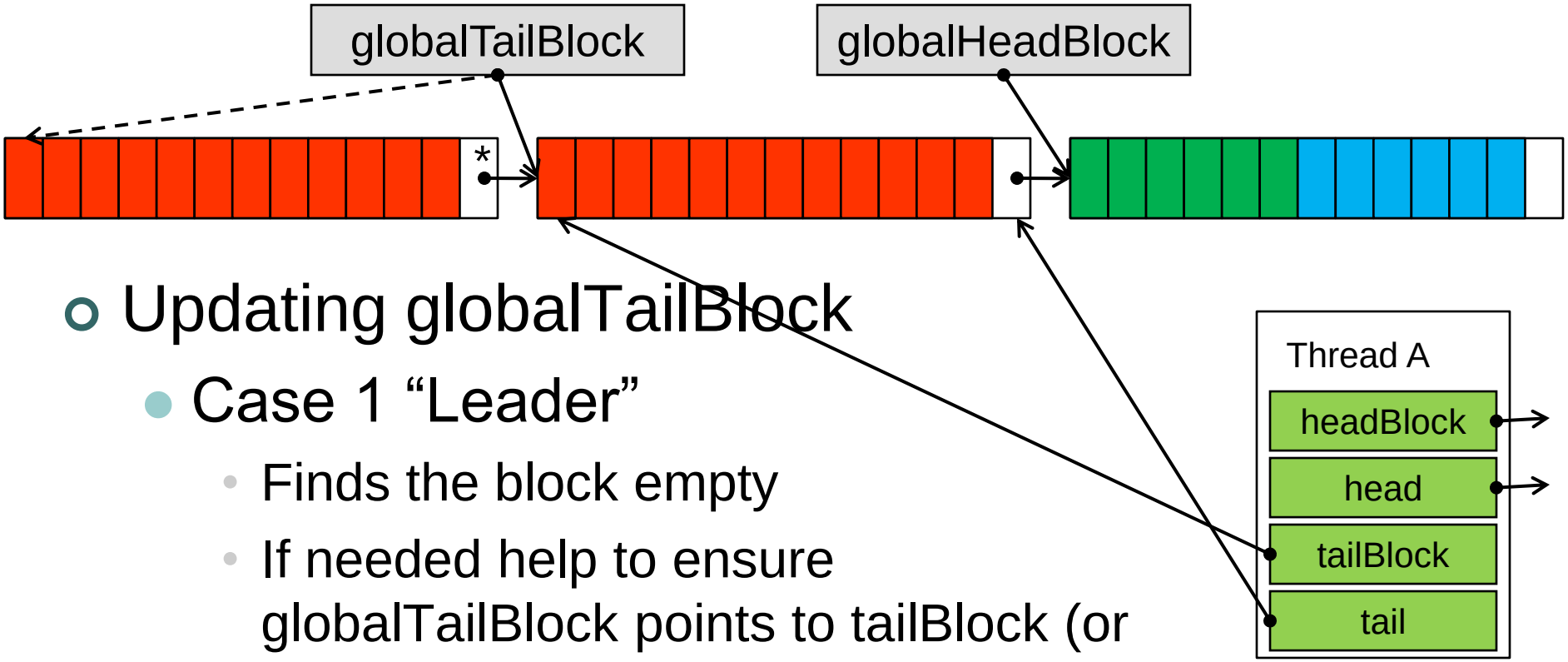
- Dequeue
  - Find the right block
    (first via TLS, then TLS->next or globalTailBlock)
  - Search the block for the first valid element
  - Remove with CAS, replace with NULL2 (linearization point)

The Algorithm

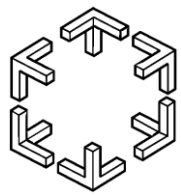


- Maintaining the chain of blocks
  - Helping scheme when moving between blocks
  - Invariants to be maintained
    - globalHeadBlock points to
      - The newest block or the block before it
    - globalTailBlock points to
      - The oldest active block (not deleted) or the block before it

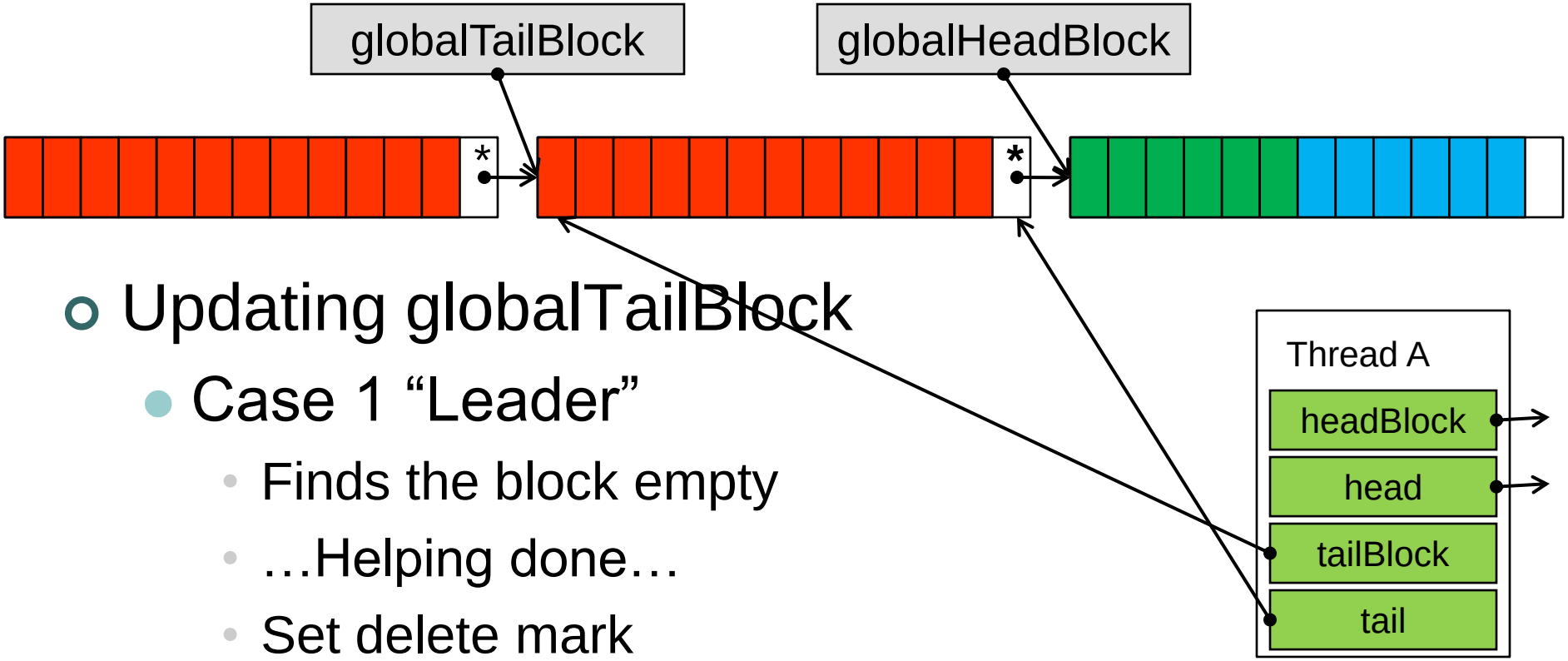
Maintaining the chain of blocks



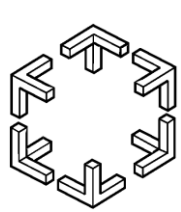
- Updating globalTailBlock
  - Case 1 “Leader”
    - Finds the block empty
    - If needed help to ensure globalTailBlock points to tailBlock (or a newer block)



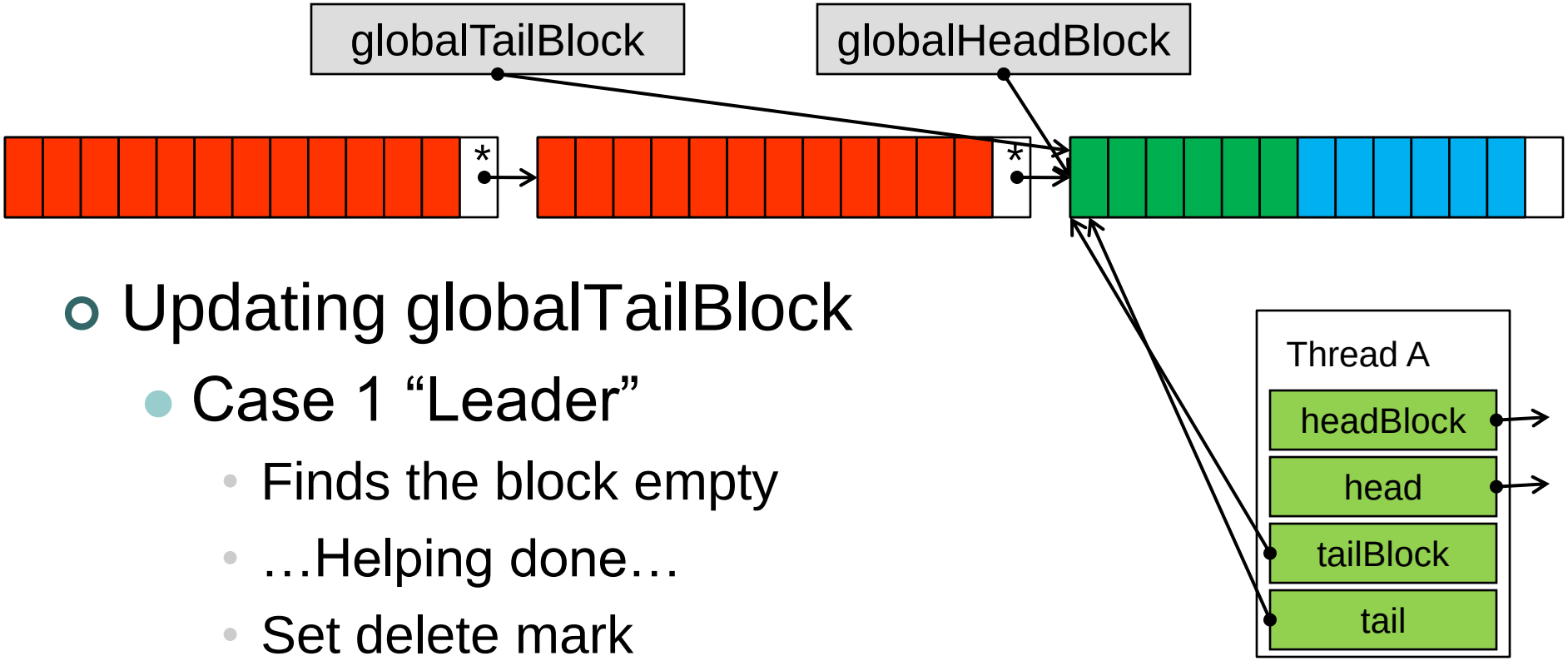
Maintaining the chain of blocks



- Updating globalTailBlock
  - Case 1 “Leader”
    - Finds the block empty
    - …Helping done…
    - Set delete mark

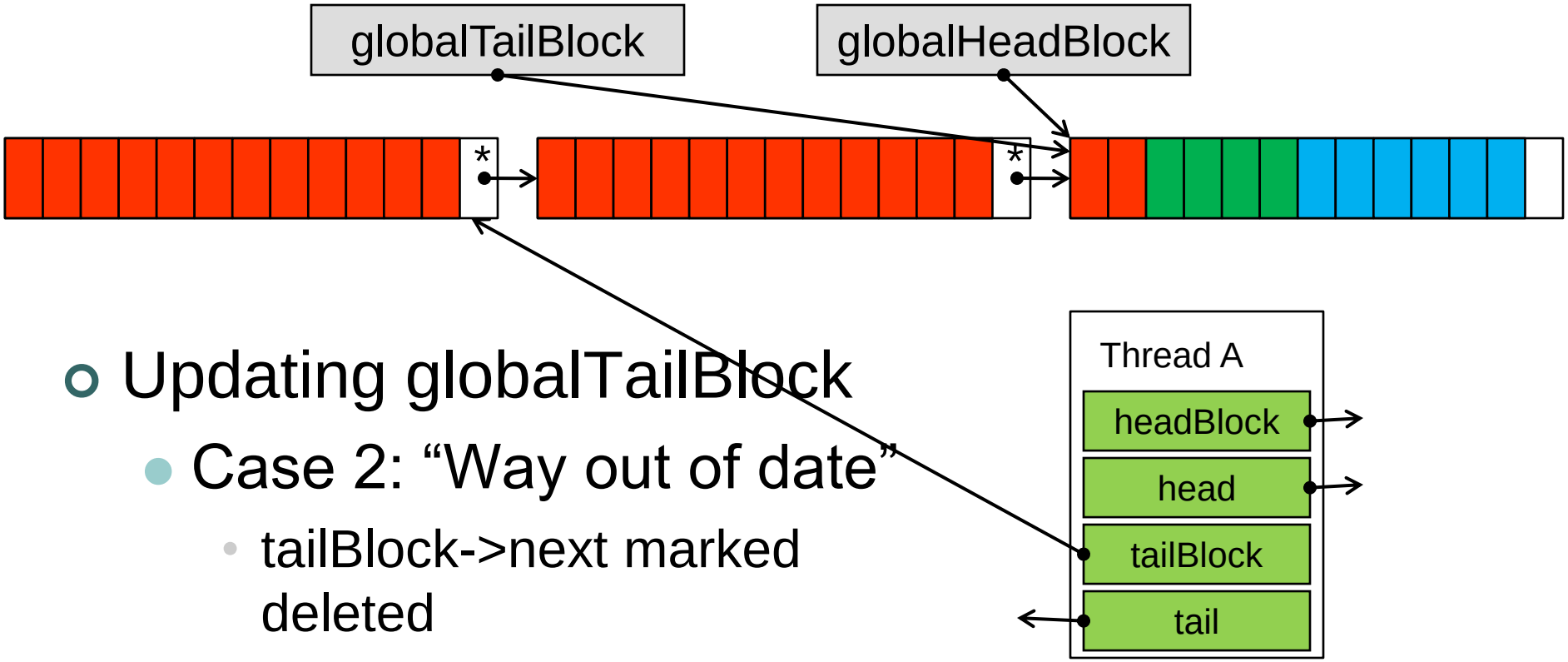


Maintaining the chain of blocks



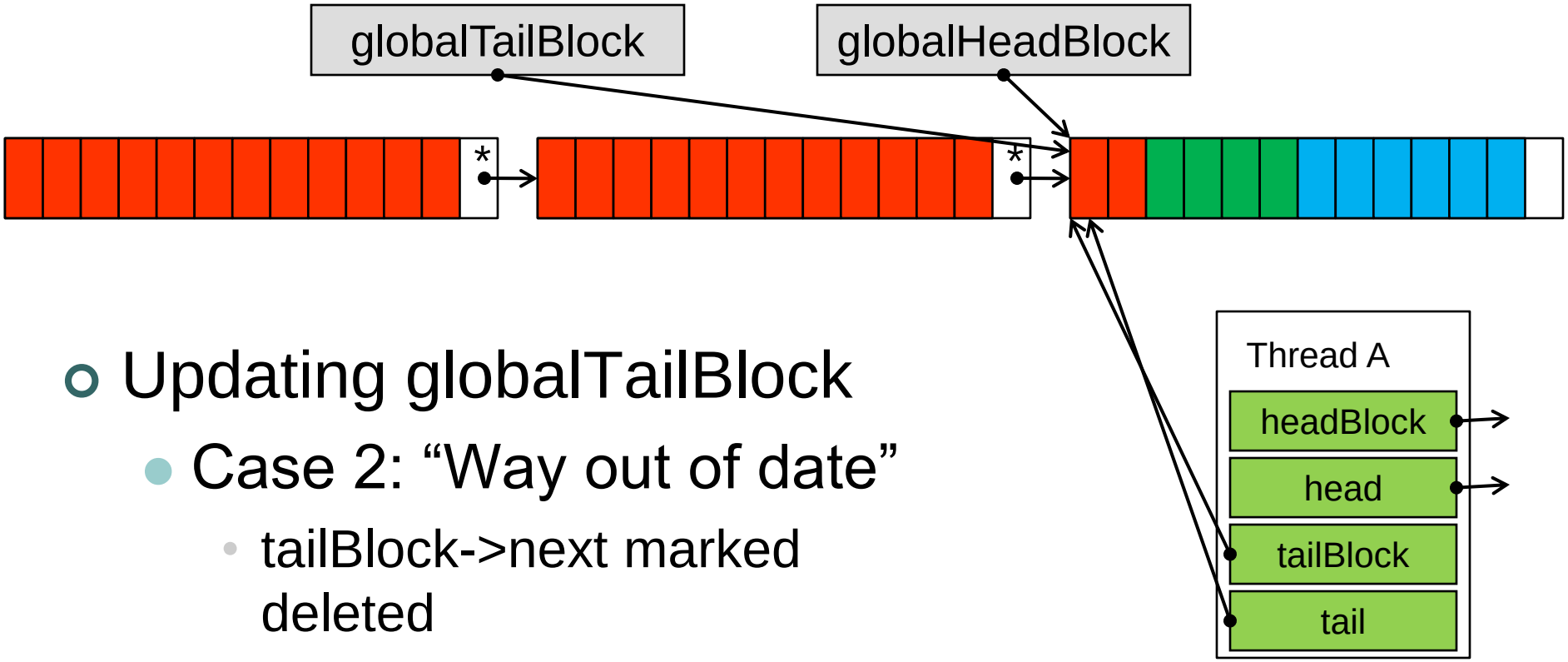
- Updating globalTailBlock
  - Case 1 “Leader”
    - Finds the block empty
    - …Helping done…
    - Set delete mark
    - Update globalTailBlock pointer
    - Move own tailBlock pointer

Maintaining the chain of blocks



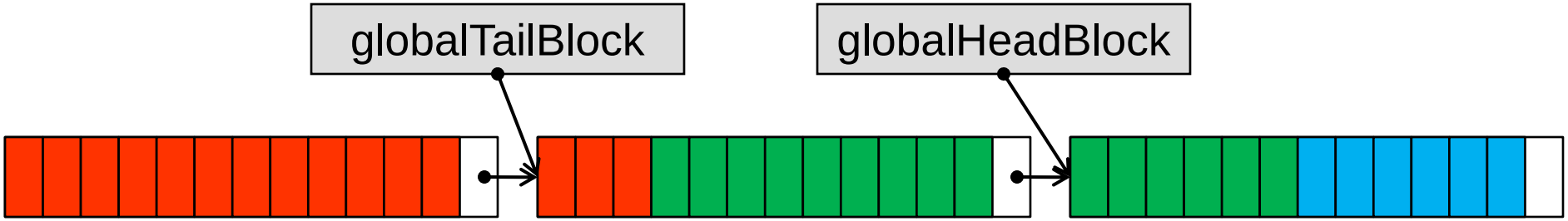
- Updating globalTailBlock
  - Case 2: “Way out of date”
    - tailBlock->next marked deleted
    - Restart with globalTailBlock

Maintaining the chain of blocks

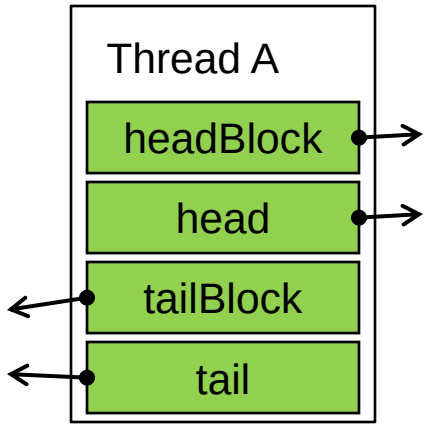


- Updating globalTailBlock
  - Case 2: “Way out of date”
    - tailBlock->next marked deleted
    - Restart with globalTailBlock

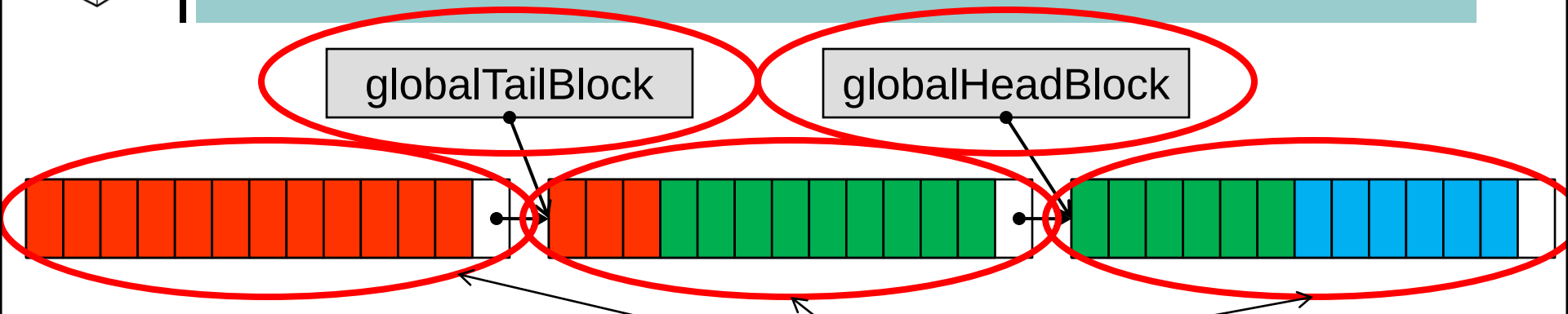
Minding the Cache



- Blocks occupy one cache-line
- Cache-lines for enqueue v.s. dequeue are disjoint (except when near empty)
- Enqueue/dequeue will cause coherence traffic for the affected block
- Scanning for the head/tail involves one cache-line

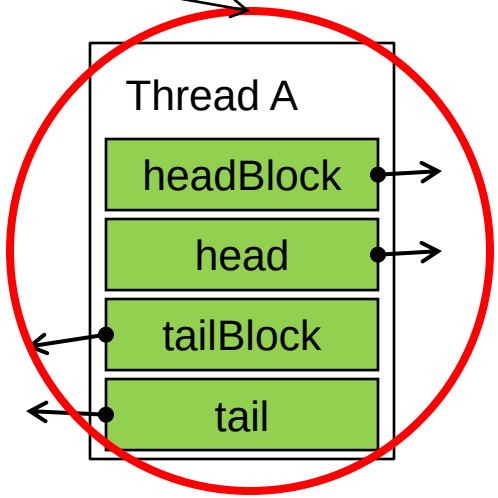


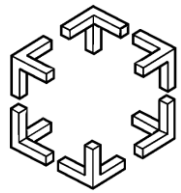
Minding the Cache



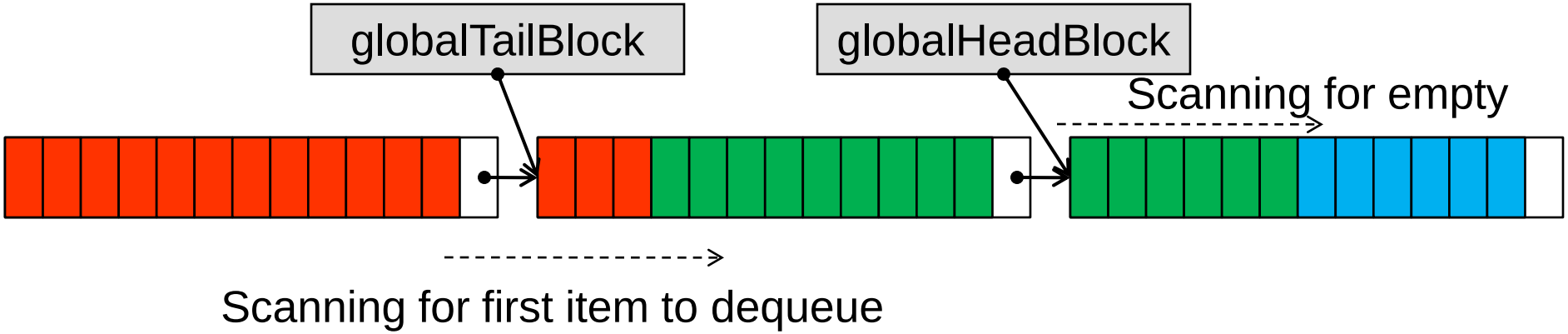
- Blocks occupy one cache-line
- Cache-lines for enqueue v.s. dequeue are disjoint (except when near empty)
- Enqueue/dequeue will cause coherence traffic for the affected block
- Scanning for the head/tail involves one cache-line

Cache-lines

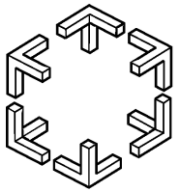




Scanning in a weak memory model?

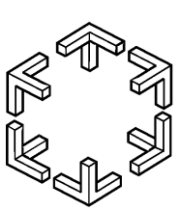


- Key observations
  - Life cycle for element values (NULL1 -> value -> NULL2)
  - Elements are updated with CAS thus requiring the old value to be the expected one.
  - Scanning only skips values later in the life cycle
    - Reading an old value is safe (will try CAS and fail)



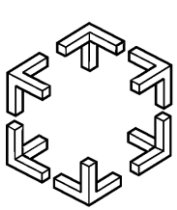
Outline

- Introduction
  - Lock-free synchronization
  - The Problem & Related work
- The lock-free queue algorithm
- Experiments
- Conclusions



Experimental evaluation

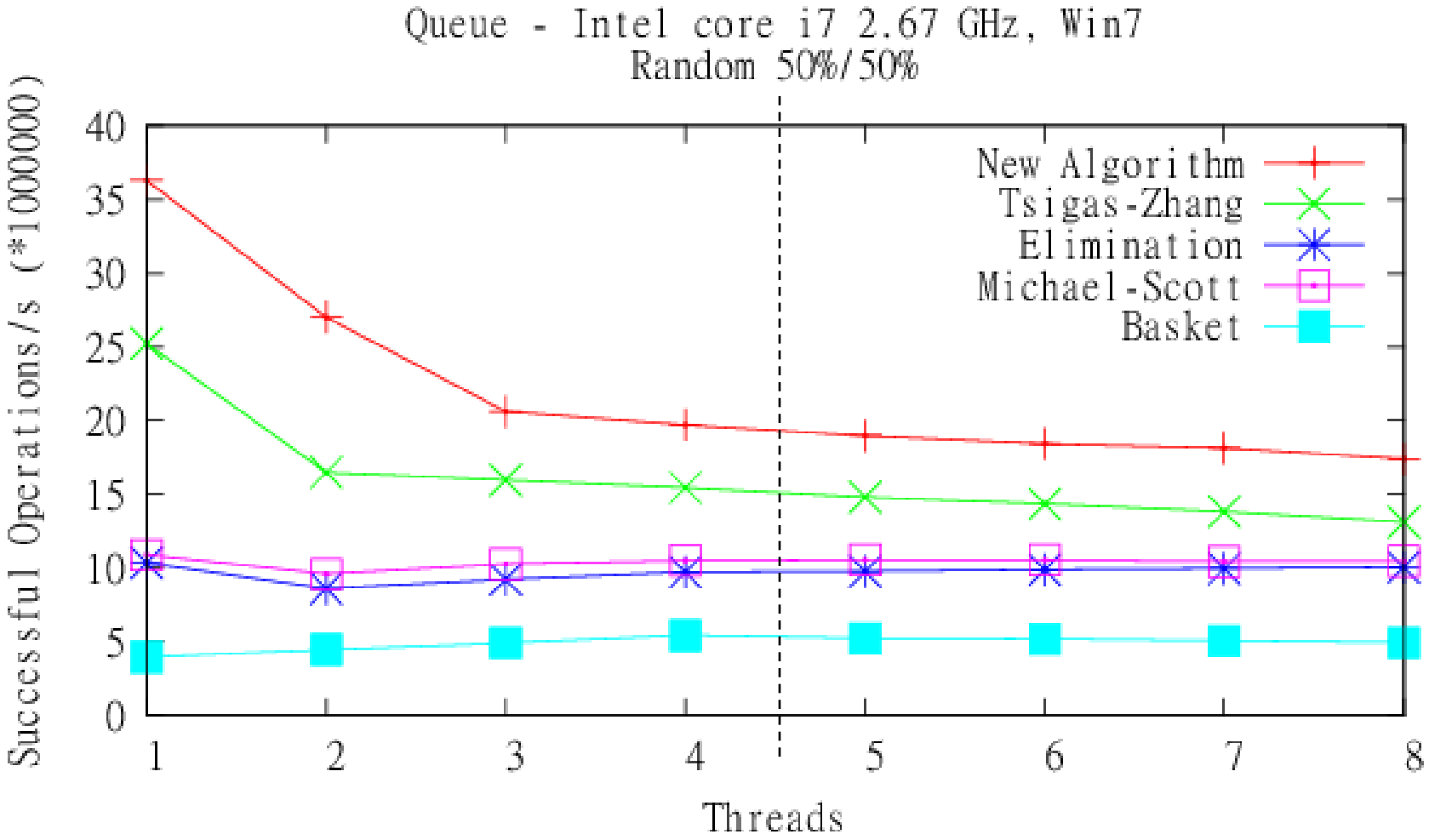
- Micro benchmark
  - Threads execute enqueue and dequeue operations on a shared queue
    - High contention.
  - Test Configurations
    1. Random *50%* / *50%*, initial size *0*
    2. Random *50%* / *50%*, initial size *1000*
    3. *1* Producer / *N-1* Consumers
    4. *N-1* Producers / *1* Consumer
  - Measured throughput in items/sec
    - #dequeues not returning EMPTY



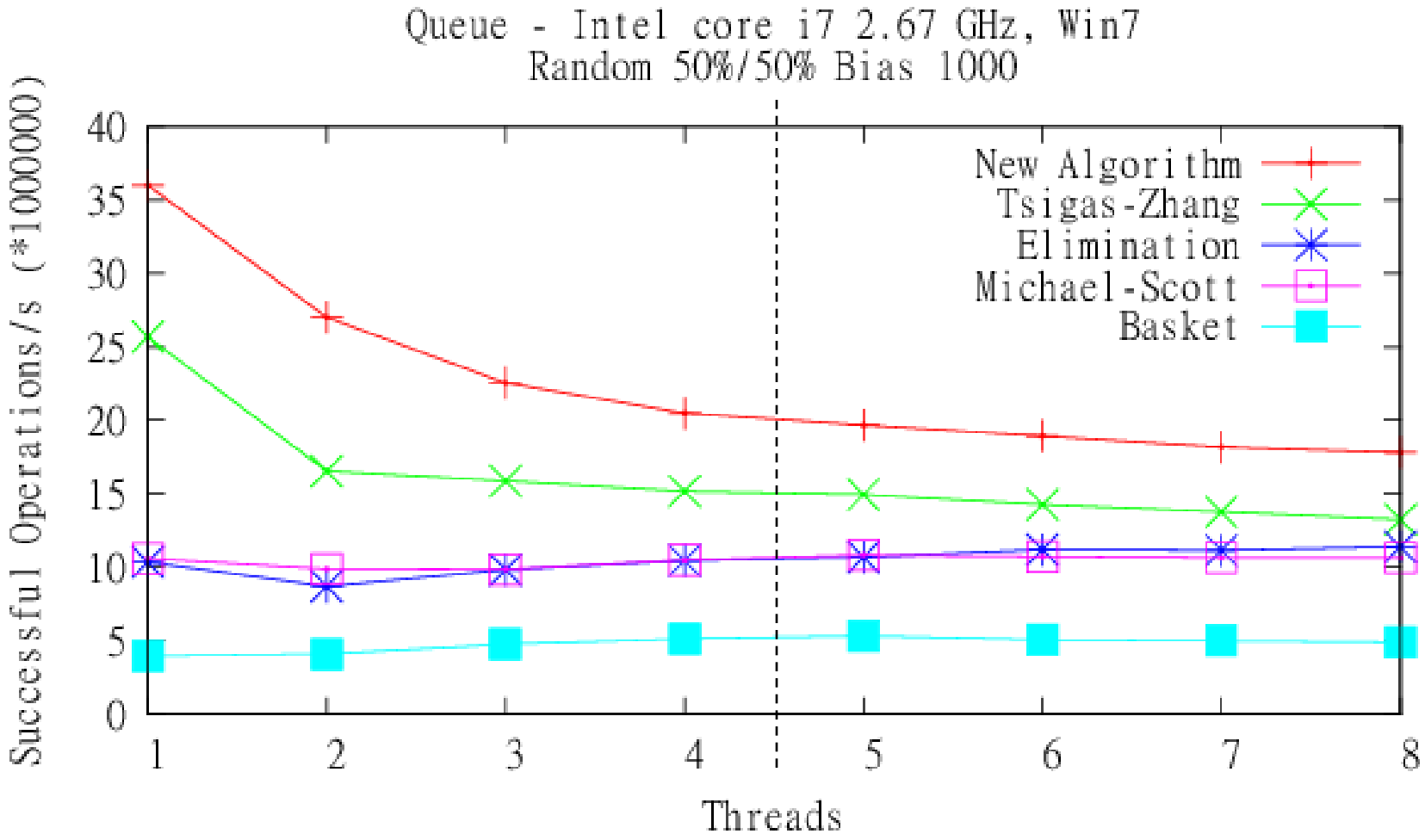
Experimental evaluation

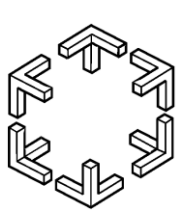
- Micro benchmark
  - Algorithms
    - [Michael & Scott, 1996]
    - [Michael & Scott, 1996] +
      Elimination [Moir, Nussbaum, Shalev & Shavit, 2005]
    - [Hoffman, Shalev & Shavit, 2007]
    - [Tsigas & Zhang, 2001]
    - The new Cache-Aware Queue [Gidenstam, Sundell & Tsigas, 2010]
  - PC Platform
    - CPU: Intel Core i7 920 @ 2.67 GHz
    - 4 cores with 2 hardware threads each
    - RAM: 6 GB DDR3 @ 1333 MHz
    - Windows 7 64-bit

Experimental evaluation (i)

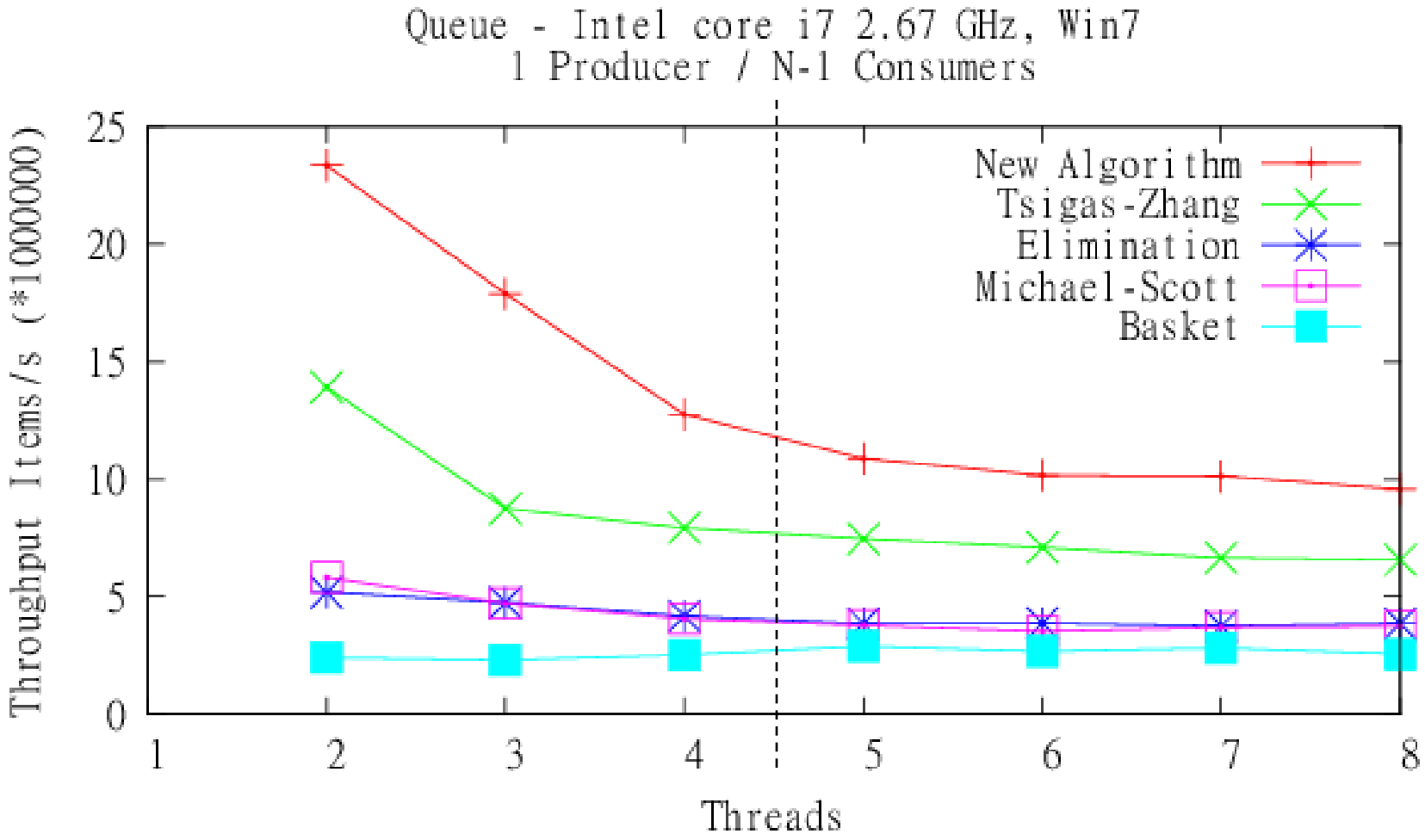


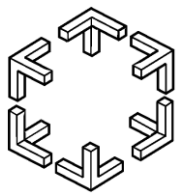
Experimental evaluation (ii)



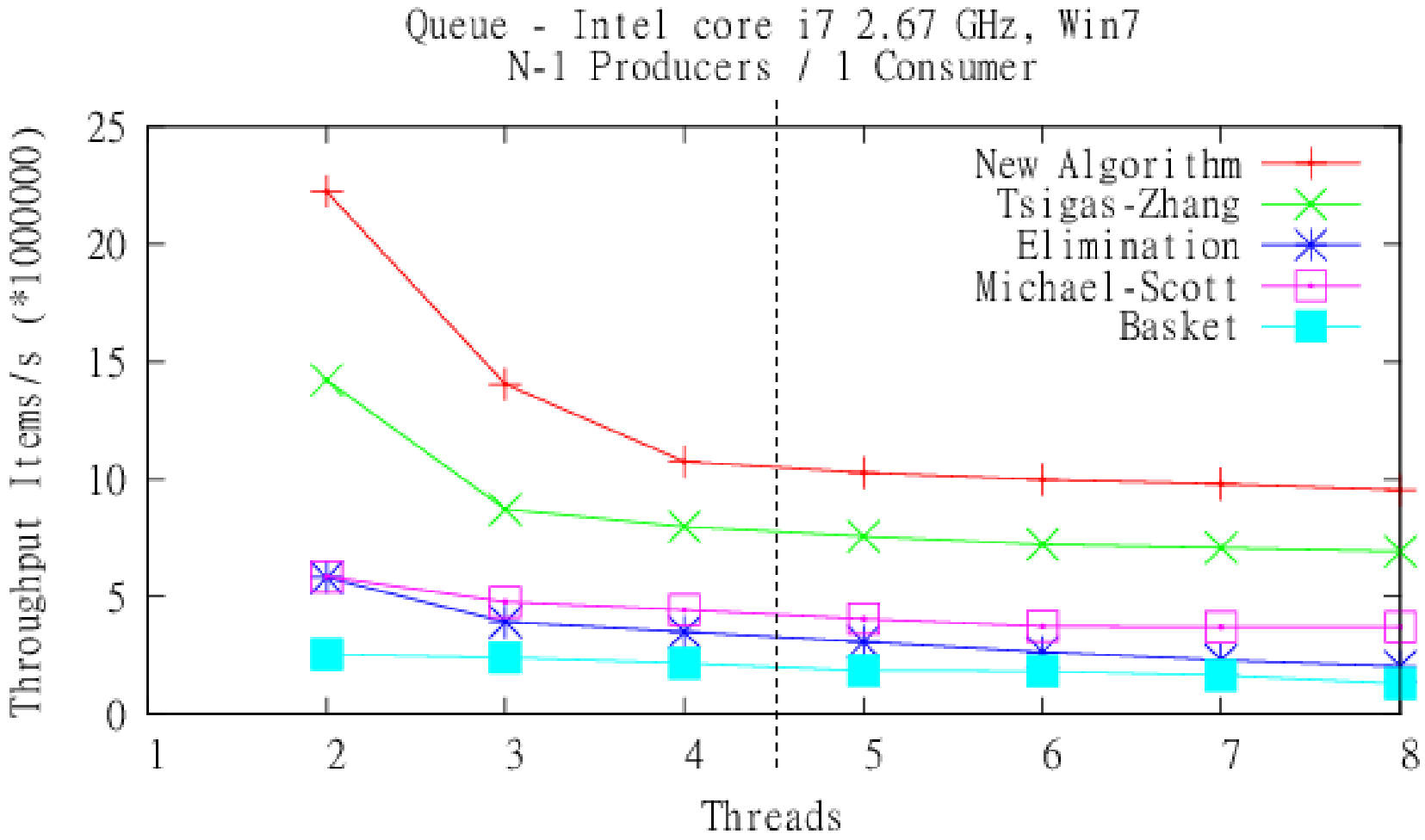


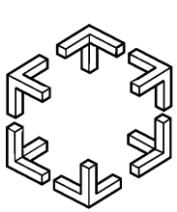
Experimental evaluation (iii)





Experimental evaluation (iv)

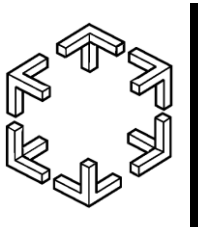




Conclusions

The Cache-Aware Lock-free Queue

- The first lock-free queue algorithm for multiple producers/consumers with all of the properties below
  - Designed to be cache-friendly
  - Designed for the weak memory consistency provided by contemporary hardware
  - Is disjoint-access parallel (except when near empty)
  - Use thread-local storage for reduced communication
  - Use a linked-list of array blocks for efficient dynamic size support



Thank you for listening!

Questions?