A Lock-Free Algorithm for Concurrent Bags

Håkan Sundell
Anders Gidenstam
Marina Papatriantafilou
Philippas Tsigas

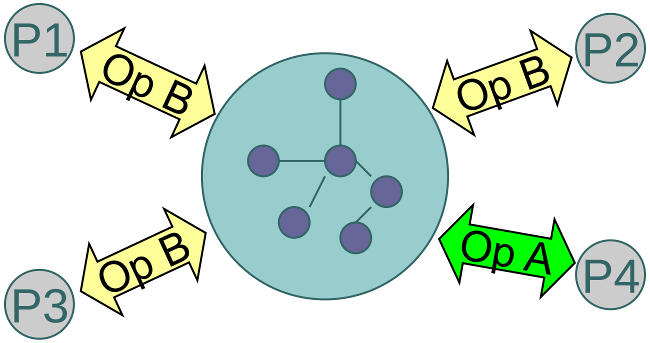
School of business and informatics
University of Borås

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 bag algorithm
- Experiments
- Conclusions

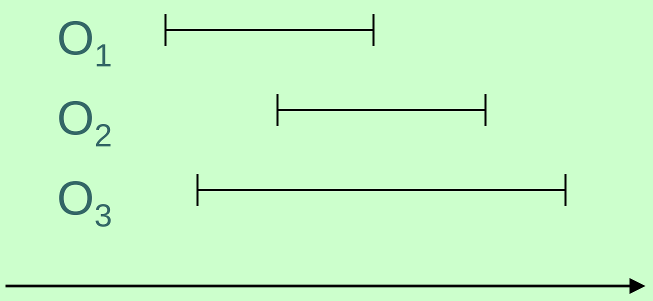
Synchronization on a shared object



- Lock-free synchronization
  - Allows 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

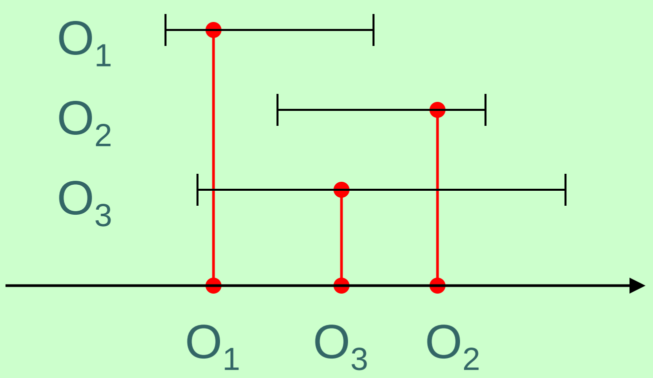
Correctness of a concurrent object

- 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.



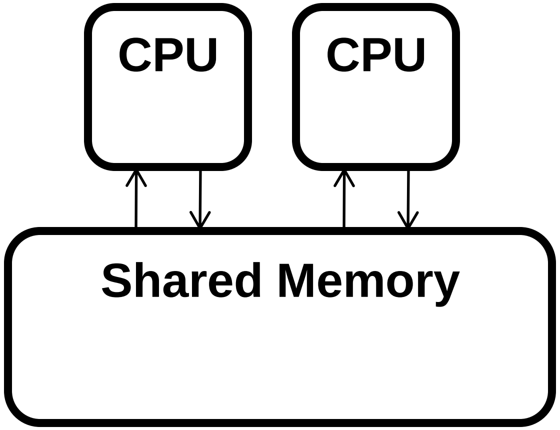
Correctness of a concurrent object

- 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
  - Model: Sequential consistency
- 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

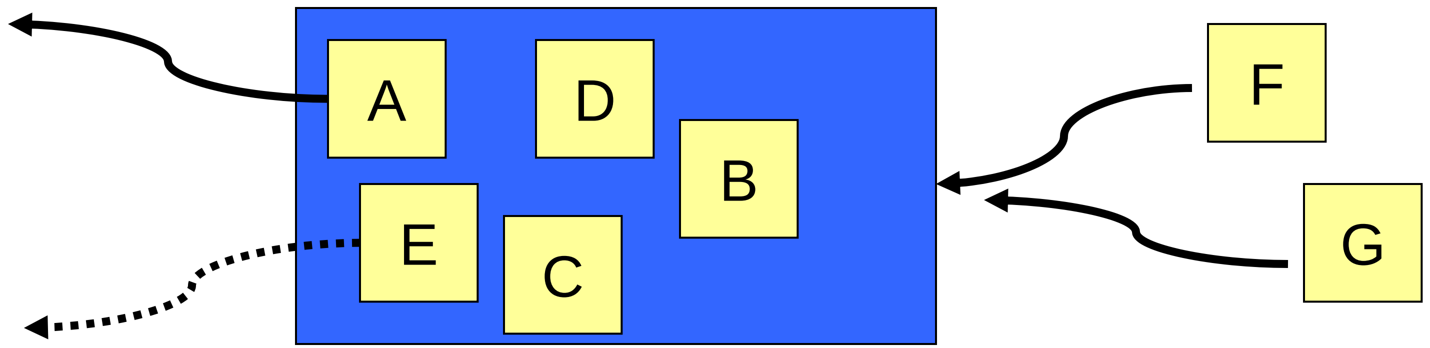


Outline

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

The Problem: Concurrent bag shared data object

- Basic operations: **Add()** and **TryRemoveAny()**
- Elements in the bag are *unordered*



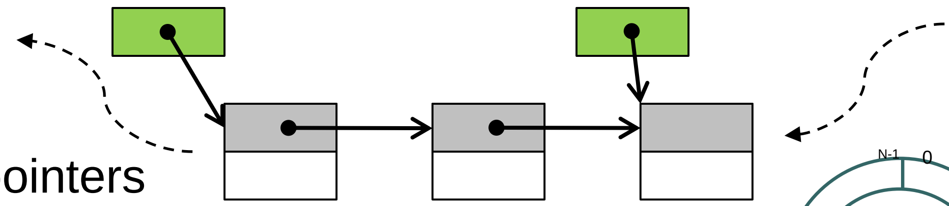
- Desired Properties
  - Linearizable and lock-free
    - Linearizable means: Add(A) -> TryRemoveAny() returns A;
      if TryRemoveAny() returns EMPTY then the bag really was empty
  - Dynamic size (maximum only limited by available memory)
  - Bounded memory usage (in terms of live contents)
  - Fast on current systems

The Problem: Concurrent bag shared data object

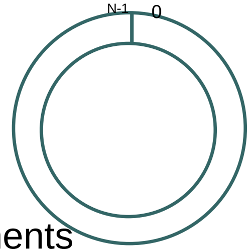
- Motivation
  - Useful for communication/work distribution, e.g.
    - Implementation of Parallel foreach / forall
    - Between (unordered) pipeline stages
  - Abstract data type available in some languages
    - Bag / Producer-Consumer Collection
    - .NET C#
    - …
  - A concurrent bag can be implemented with other data structures, e.g. queue, stack, …
    - But “better” service comes at a price

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

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

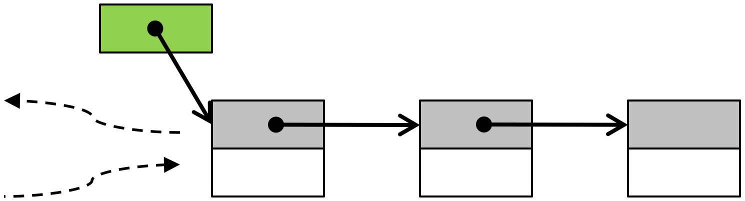


- [Tsigas & Zhang, 2001]
  - Static circular array of elements
    - Two different NULL values for distinguishing initially empty from dequeued elements
  - Global shared head and tail indices, lazily updated



- [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
  - Uses stronger memory management than M&S (SLFRC or Beware&Cleanup)

Related Work:
Lock-free Multi-P/C Stacks and Pools



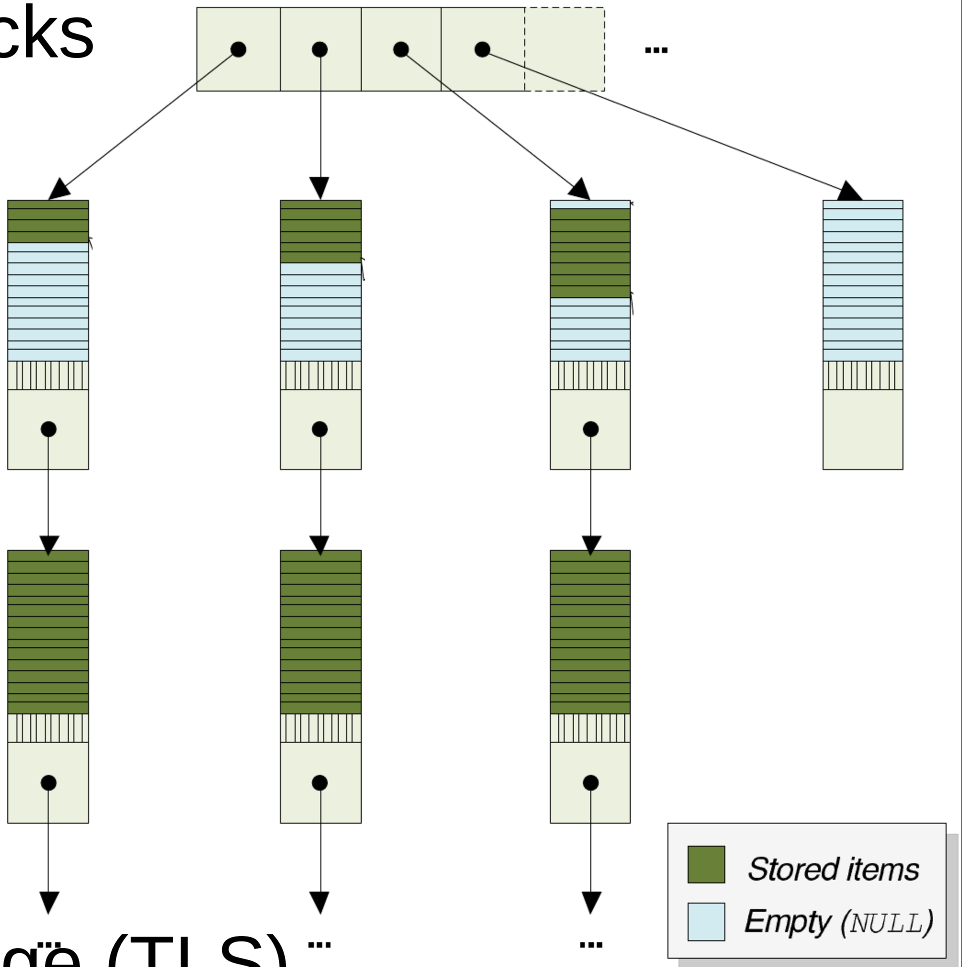
- L-F stacks
  - [Michael, 2004]
    - Linked-list, one element/node
    - Global shared head pointer
  - [Michael, 2004] +
    Elimination [Hendler, Shavit & Yerushalami, 2004]
    - Same as the above + elimination of concurrent pairs of push and pop.
- L-F pool
  - [Afek, Korland, Natanzon & Shavit, 2010]
    - Tree of balancers with elimination + queues.

Outline

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

The Algorithm
Basic Idea

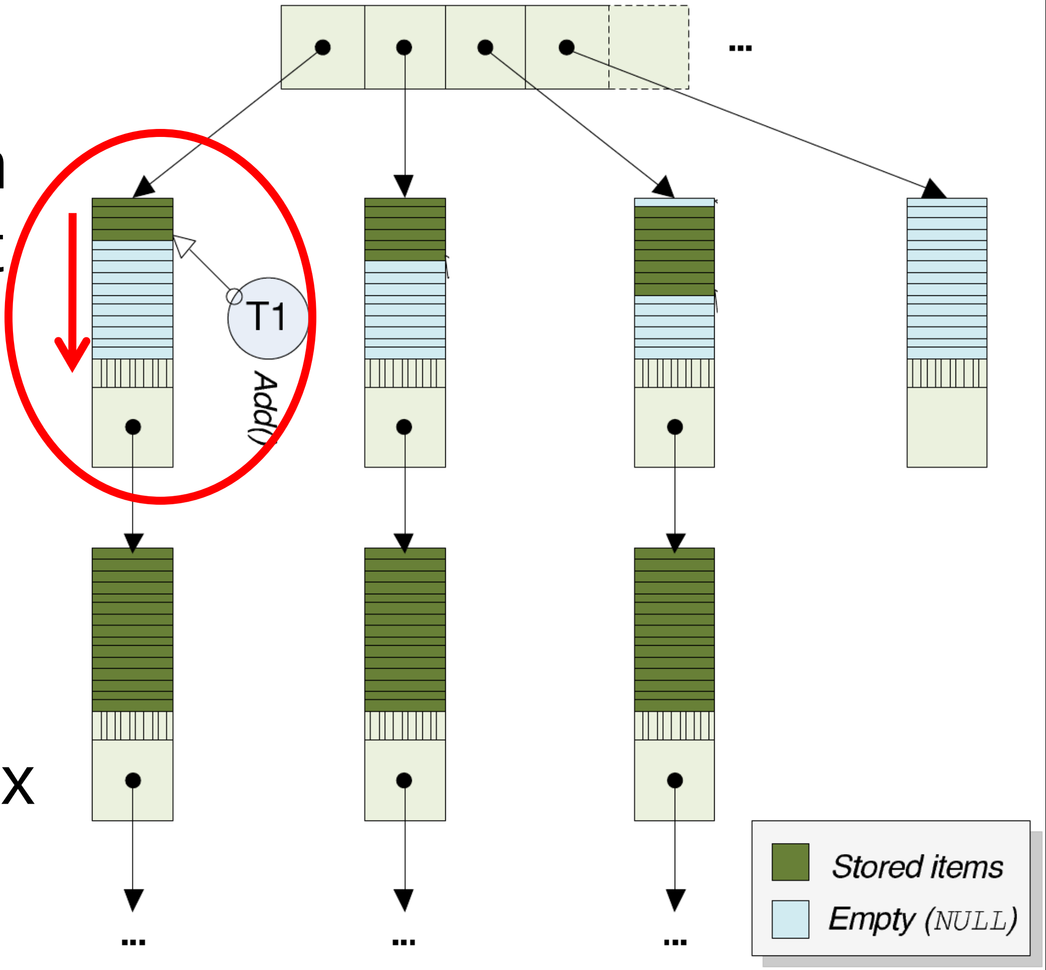
- Linked-lists of array blocks
  - One list per thread
    - Always used by Add()
    - TryRemoveAny() looks there first
  - Add()s by different threads do not contend
  - A TryRemoveAny() has a large number of blocks to choose from
    - Low risk for contention



- Static thread-local storage (TLS)
  - Used to avoid reading/writing shared state

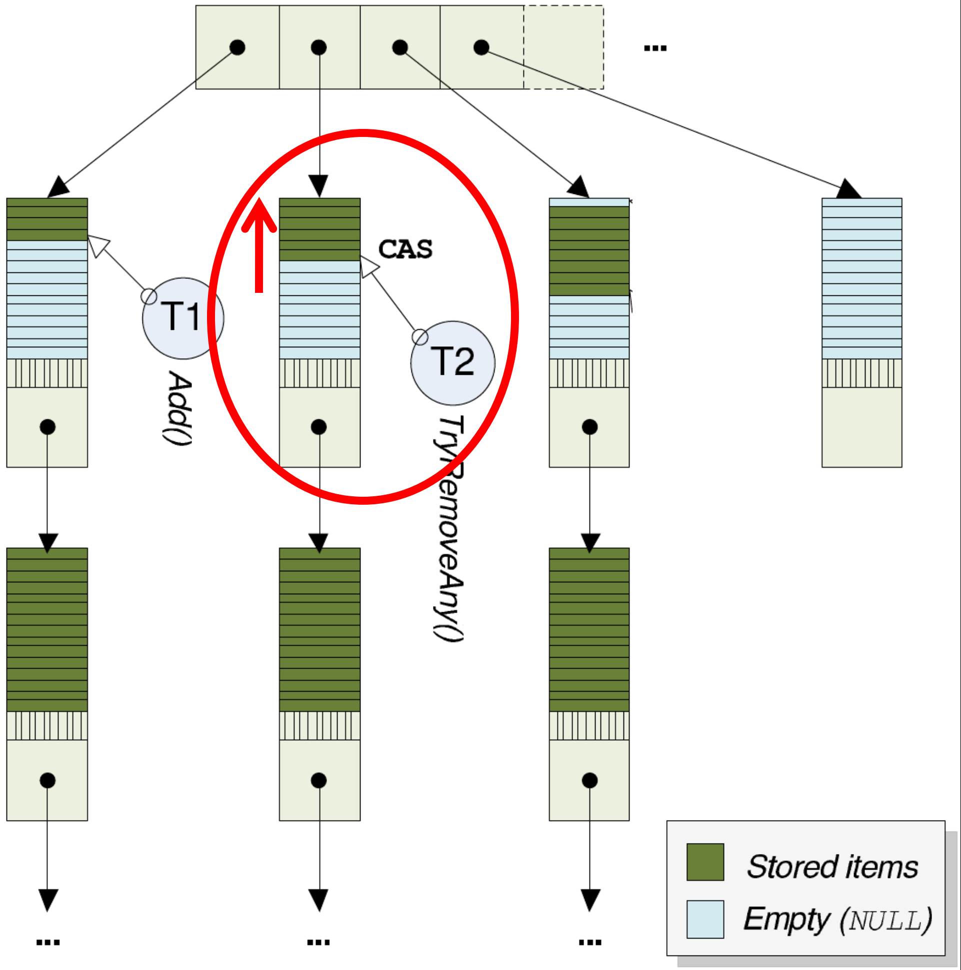
The Algorithm
Basic Idea

- Add()
  - Item is inserted in an empty slot in the first array block in the thread's list
  - A new first block is added when all slots have been used
  - The current slot index is stored in TLS



The Algorithm
Basic Idea

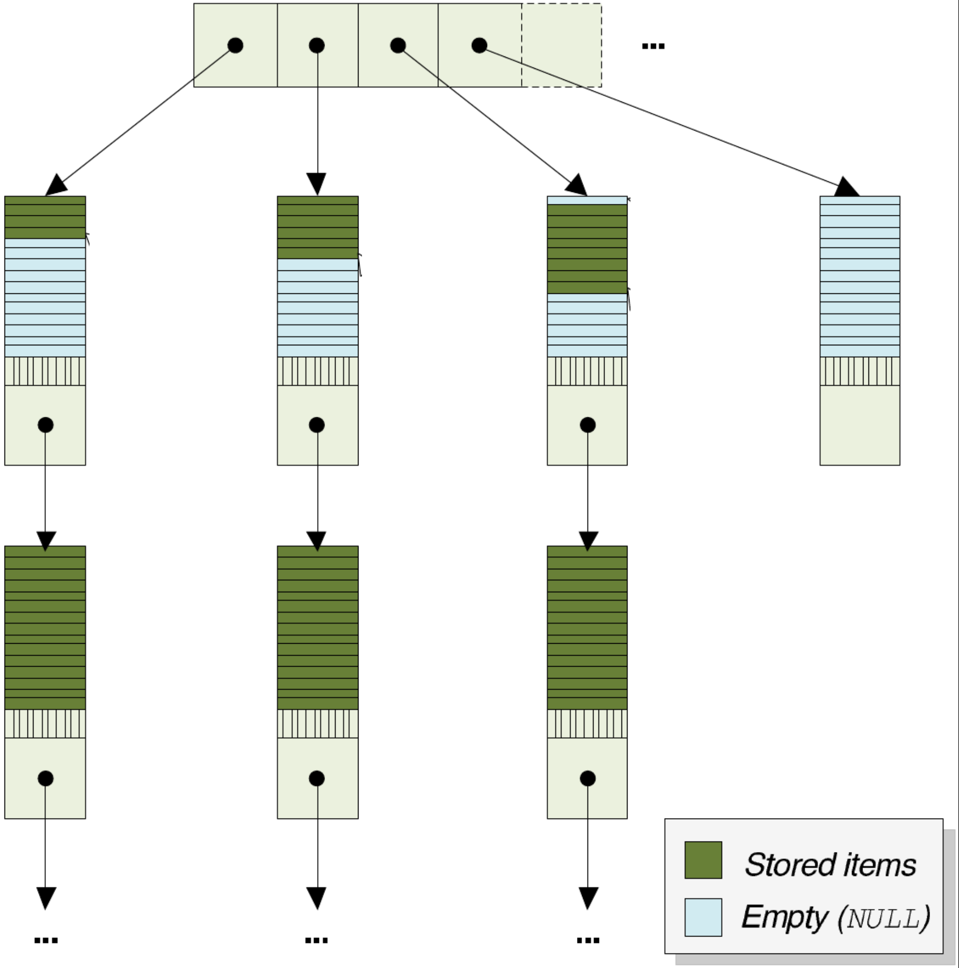
- TryRemoveAny()
  - The thread first scans the first block in its list (from the current index in TLS)
  - When an item is found it is removed via CAS
  - If the block is empty it is removed from the list



The Algorithm

Issues

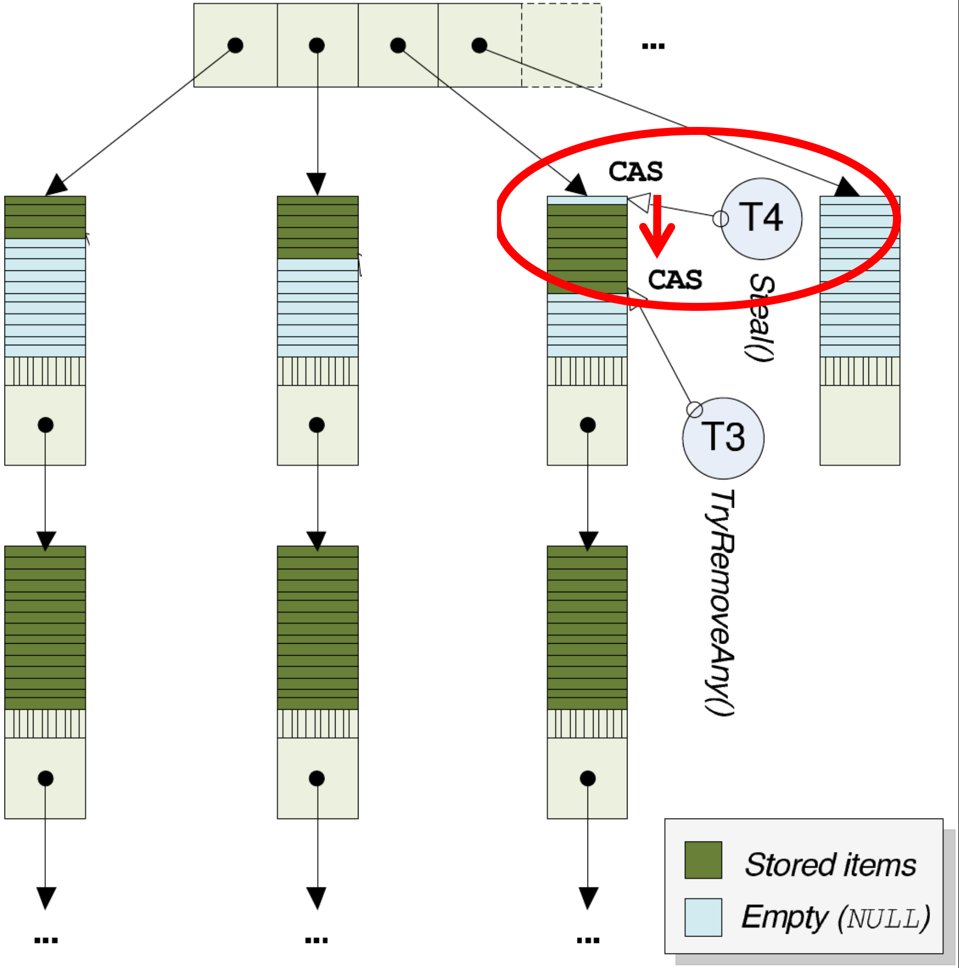
- Finding items when own list is empty
- Detecting that the bag is empty
- Managing the linked lists



The Algorithm

Finding items when the own list is empty

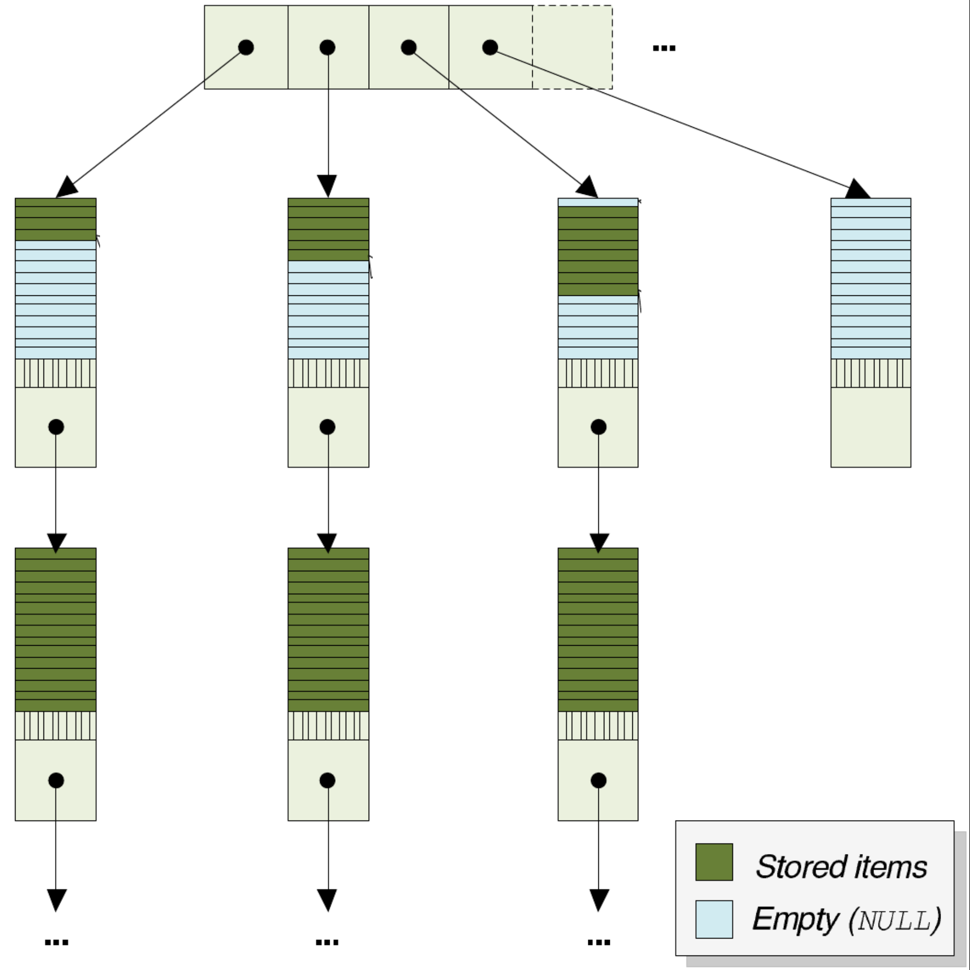
- Steal items from blocks belonging to other threads
  - Hence, CAS needed to remove items
- Never leave a block until it is empty
  - Help removing empty blocks



The Algorithm

Detecting that the bag is empty

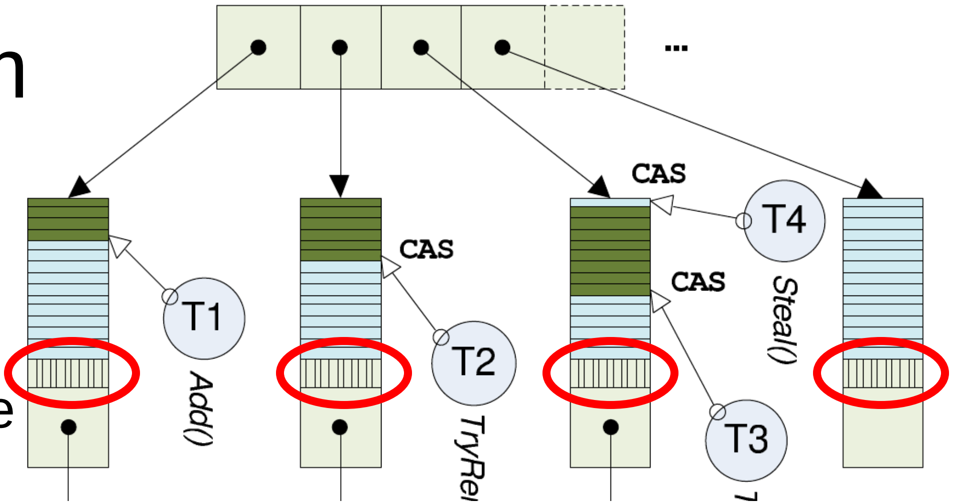
- No single place to look
- Scan all blocks of all threads
  - Items may be added concurrently
  - Items may be removed concurrently



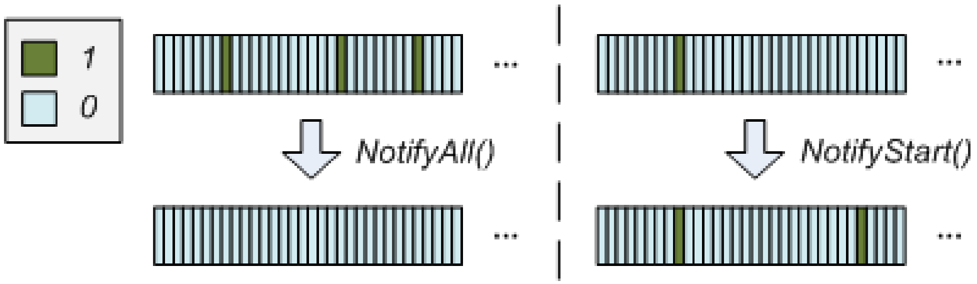
The Algorithm

Notification mechanism

- Per-block bit field
  - One bit for each thread
  - All bits cleared by Add()
  - Thieves set their bit before scanning the block



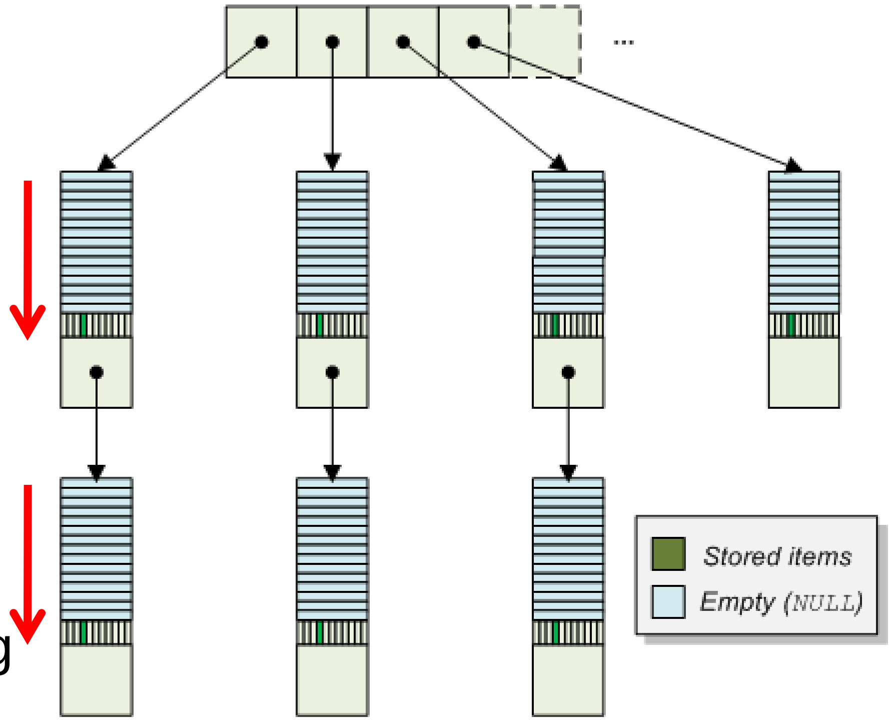
If the bit is still set for all blocks when the thief rescans the bag is empty?



The Algorithm

Notification mechanism

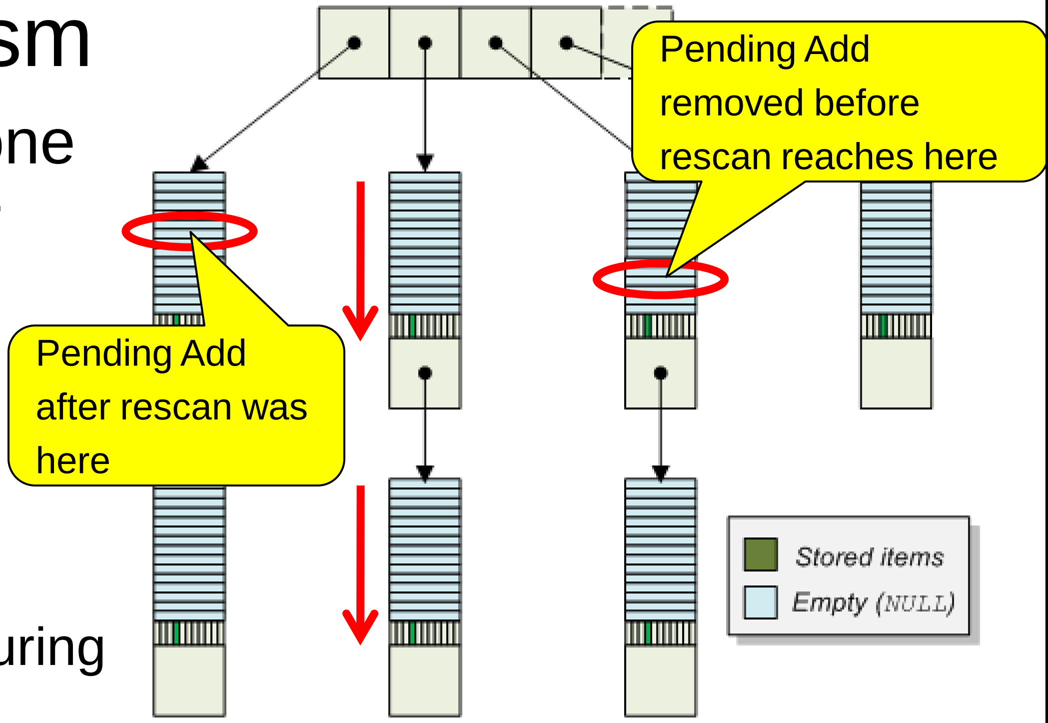
- No, there can still be one pending Add per other thread
  - Cleared the notify bits before the thief started scanning
  - Items can show up and disappear (removed) during the scan



The Algorithm

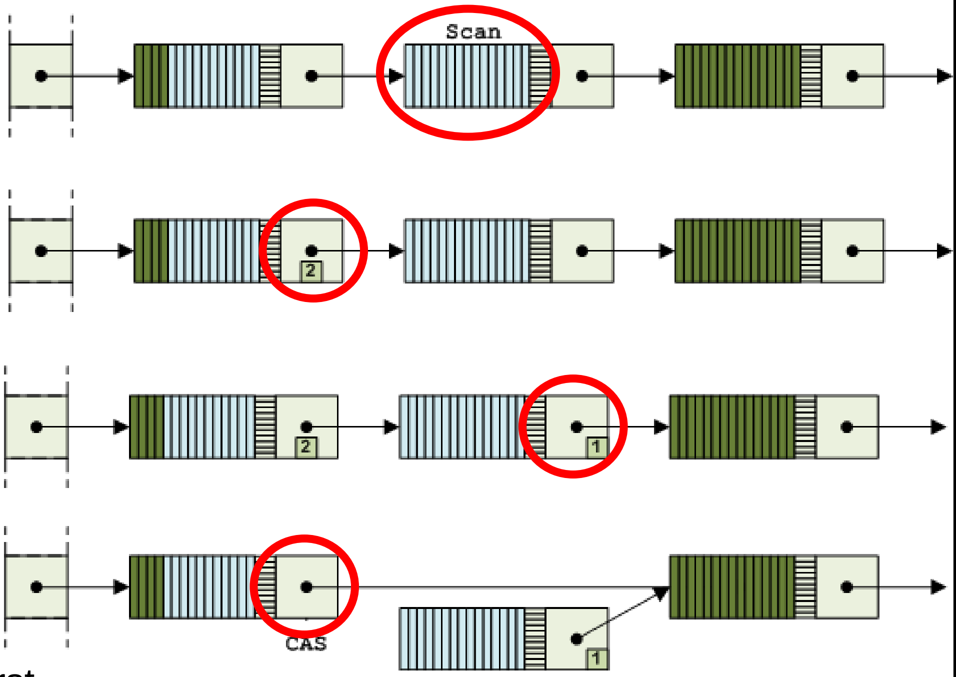
Notification mechanism

- No, there can still be one pending Add per other thread
  - Cleared the notify bits before the thief started scanning
  - Items can show up and disappear (removed) during the scan
- Rescan everything *#threads+1* times
  - if found empty in all scans it truly was empty



The Algorithm
Managing the linked lists

- Removing blocks
  - When the block is scanned and found empty it is marked logically deleted, with mark1
    - By owner
      - No problem
    - By thief
      - Must not be the first block in the linked-list since owner may add items there
      - Mark the preceding block with mark2 first
        - The block cannot be the first
        - Prevents the block from becoming the first block



  - Seeing mark1 or mark2 invokes helping
- Memory management
  - (Modified) Hazard pointers scheme [Michael, 2002]

The Algorithm
Managing the linked lists

- Properties
  - Only the owner can remove the first block
  - The last block of each linked-list cannot be removed
  - Thieves can remove any other block found empty
- So
  - After an linked list has been scanned by TryRemoveAny() there can be at most 2 empty blocks in it
  - Hence, a thread finding the bag empty will have no more than *2*#threads* blocks to traverse once it has helped any pending removals (at most 1 per thread)

Outline

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

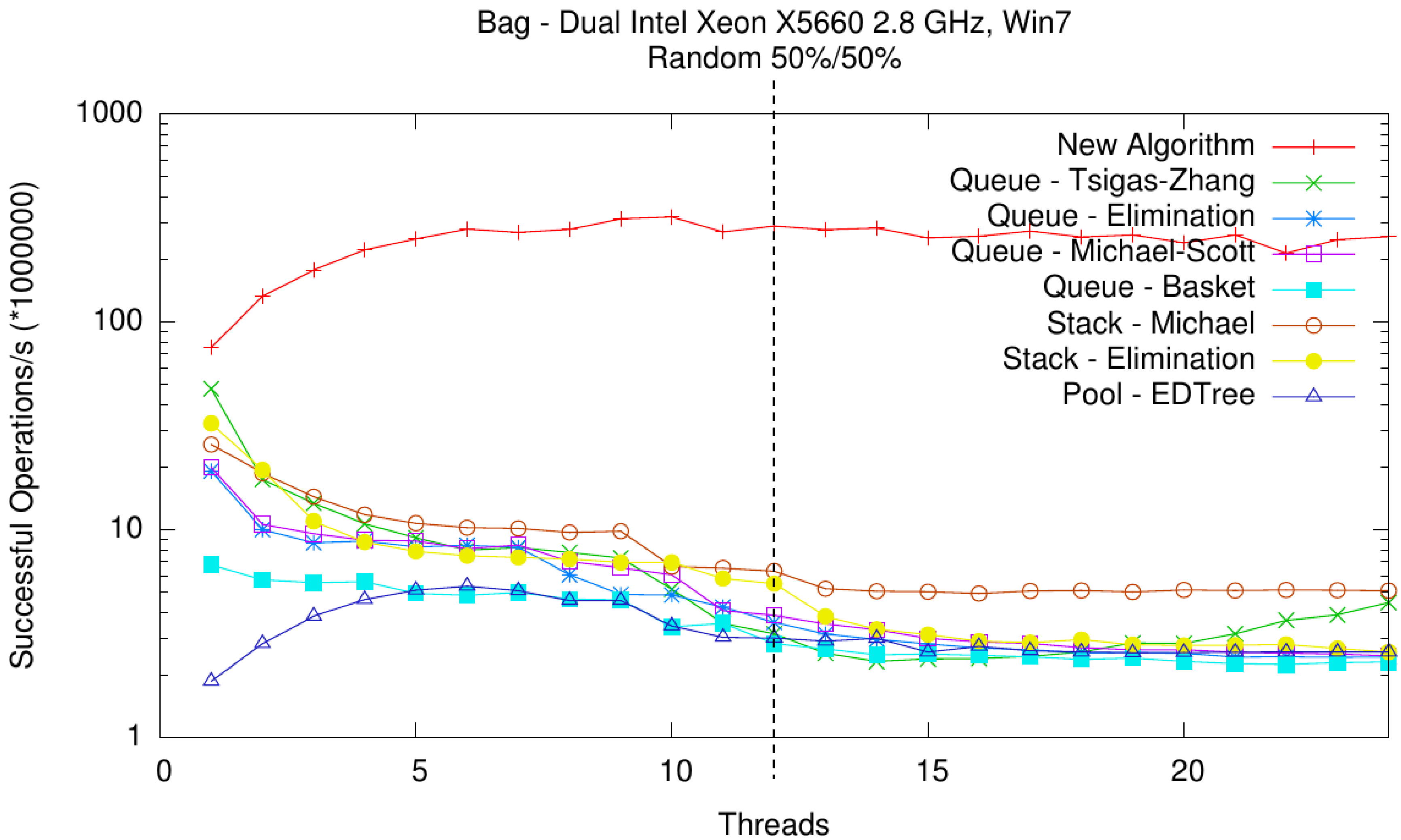
Experimental evaluation

- Micro benchmark
  - Threads execute Add and TryRemoveAny operations on a shared bag
    - High contention
  - Test Configurations
    1. Random *50% / 50%*, initial size *0*
    2. *1* Producer / *N-1* Consumers, initial size *0*
    3. *N-1* Producers / *1* Consumer, initial size *0*
    4. *N/2* Producers / *N/2* Consumers, initial size *0*
  - Measured throughput in items/sec
    - #TryRemoveAny not returning EMPTY
- Application
  - Parallel computation of Mandelbrot set
  - Producer/Consumer pattern

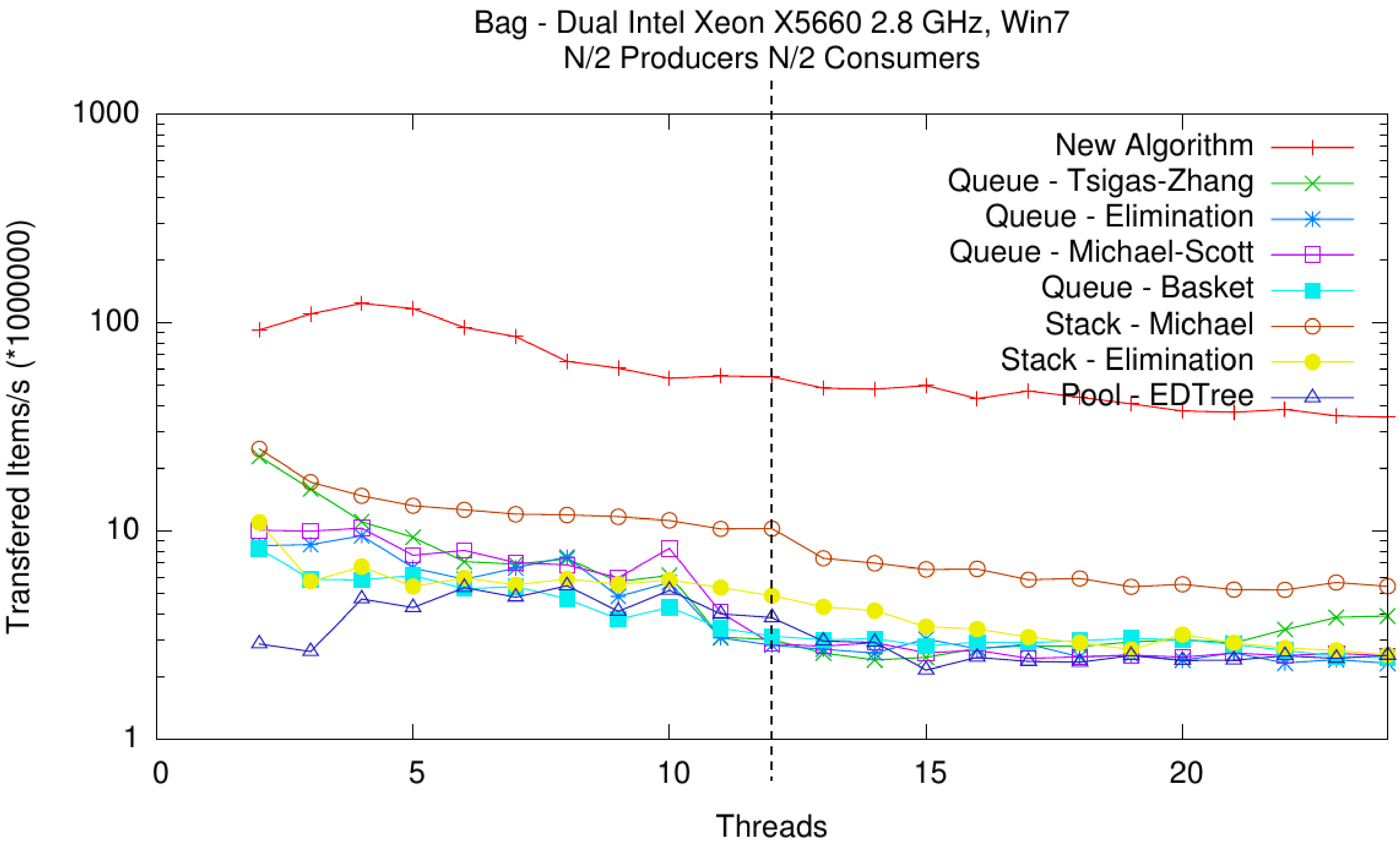
Experimental evaluation

- Algorithms
  - L-F queue [Michael & Scott, 1996]
  - L-F queue [Michael & Scott, 1996] + Elimination [Moir, Nussbaum, Shalev & Shavit, 2005]
  - L-F queue [Tsigas & Zhang, 2001]
  - L-F queue [Hoffman, Shalev & Shavit, 2007]
  - L-F stack [Michael, 2004]
  - L-F stack [Michael, 2004] + Elimination [Hendler, Shavit & Yerushalami, 2010]
  - L-F pool [Afek, Korland, Natanzon & Shavit, 2010]
  - The new L-F bag [Gidenstam, Sundell, Papatriantafilou & Tsigas, 2011]
- PC Platform
  - CPU: 2x Intel Xeon X5660 @ 2.8 GHz
  - 6 cores per CPU with 2 hardware threads each => **12 cores, 24 hw threads**
  - RAM: 12 GB DDR3 @ 1333 MHz
  - Windows 7 64-bit

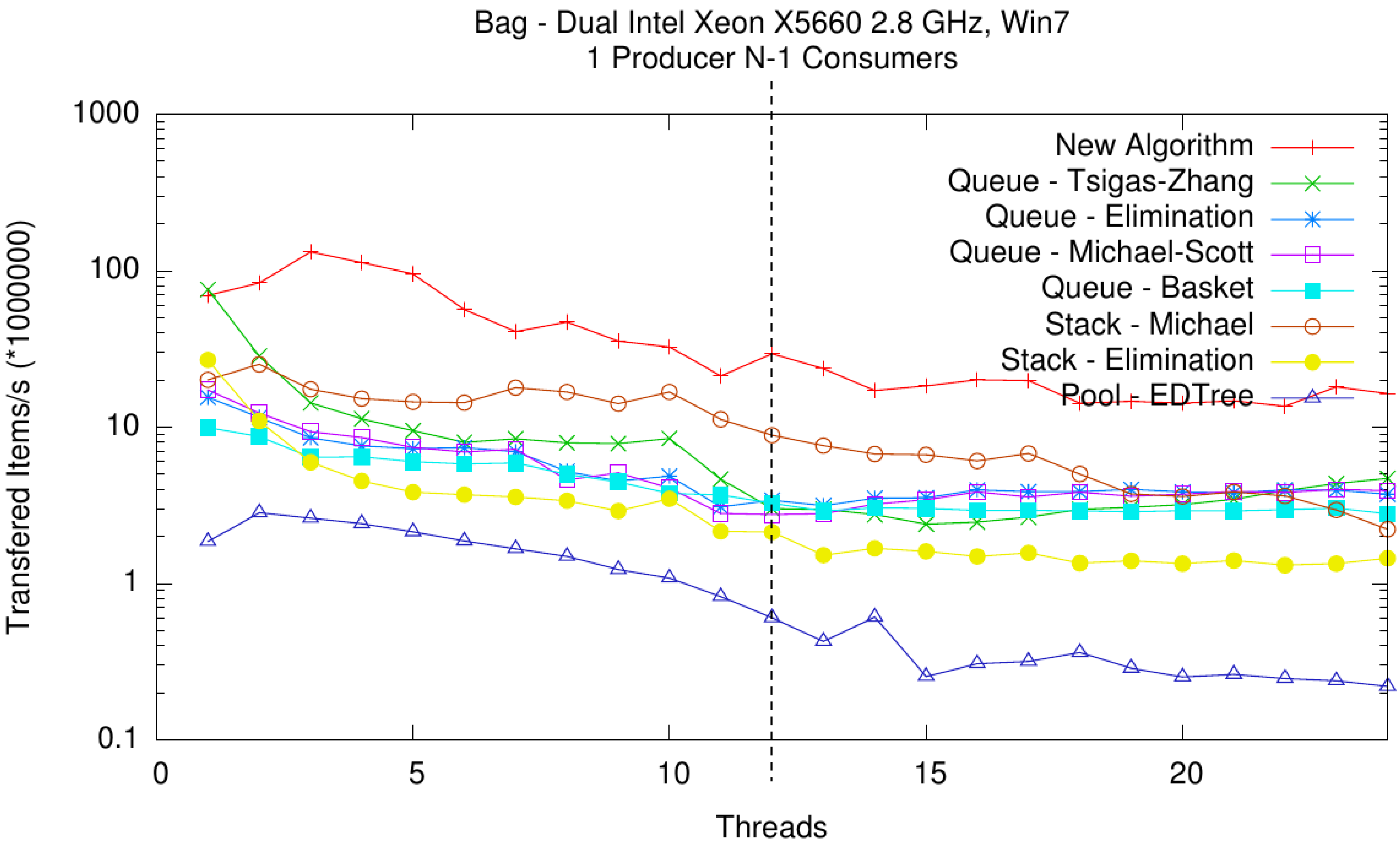
Experimental evaluation (i)



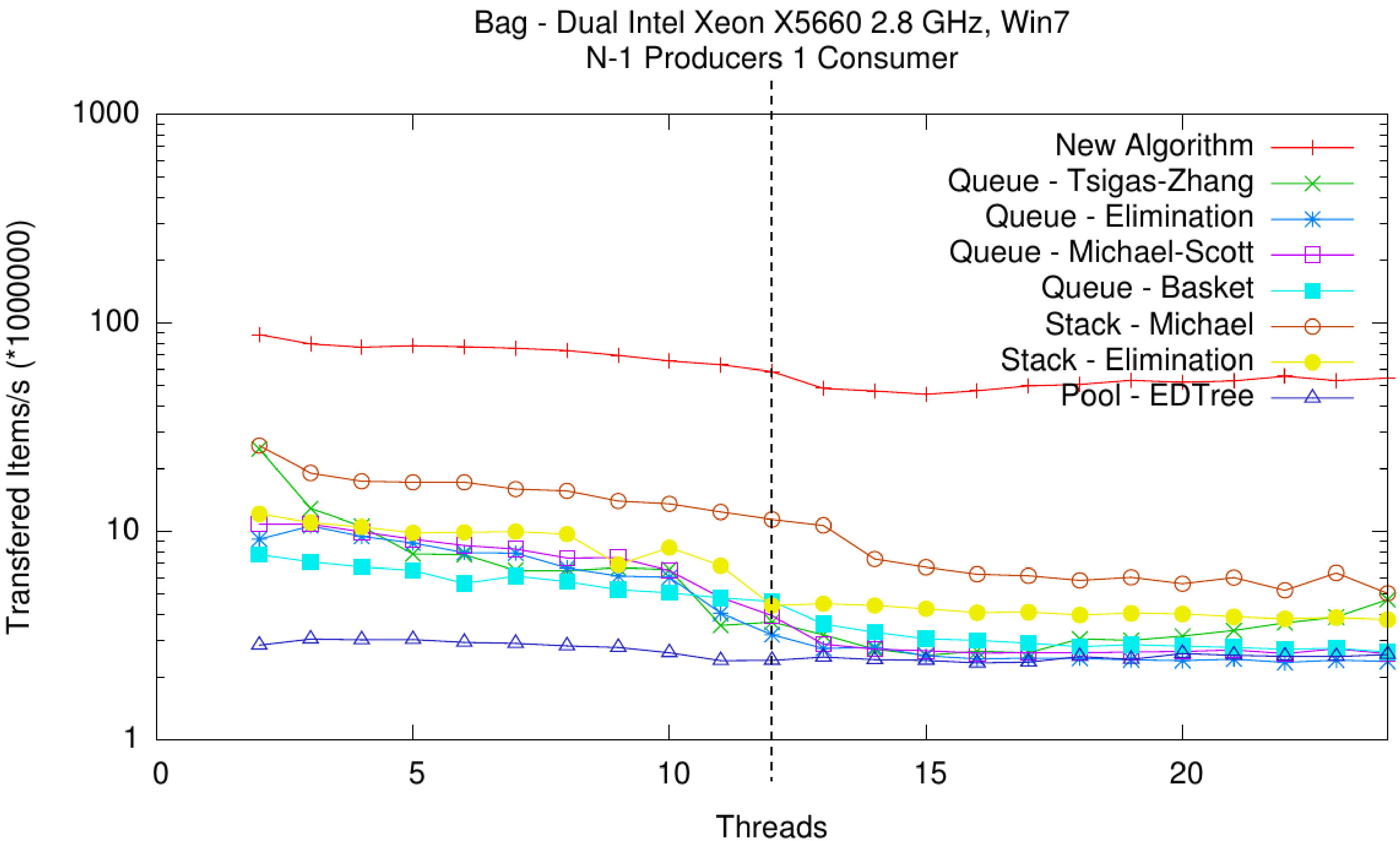
Experimental evaluation (ii)



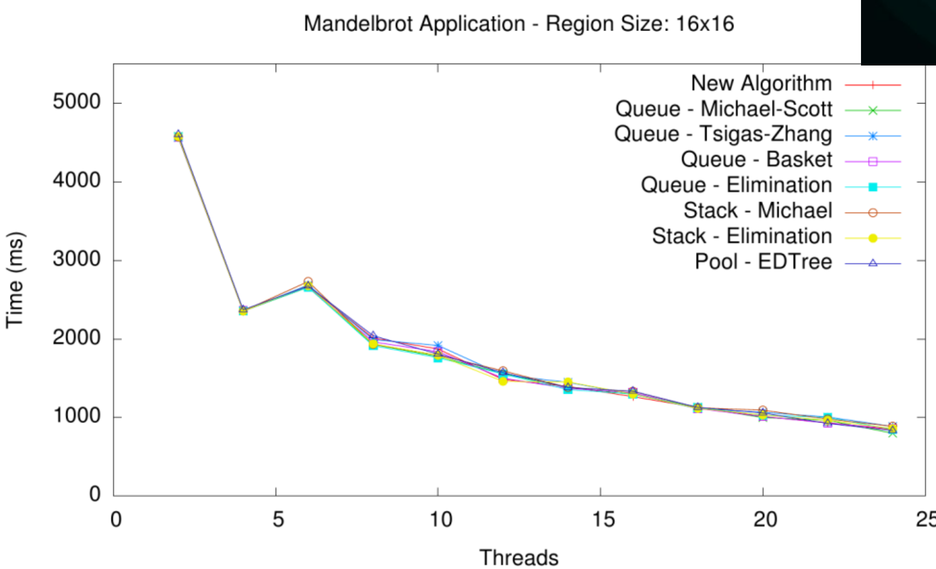
Experimental evaluation (iii)

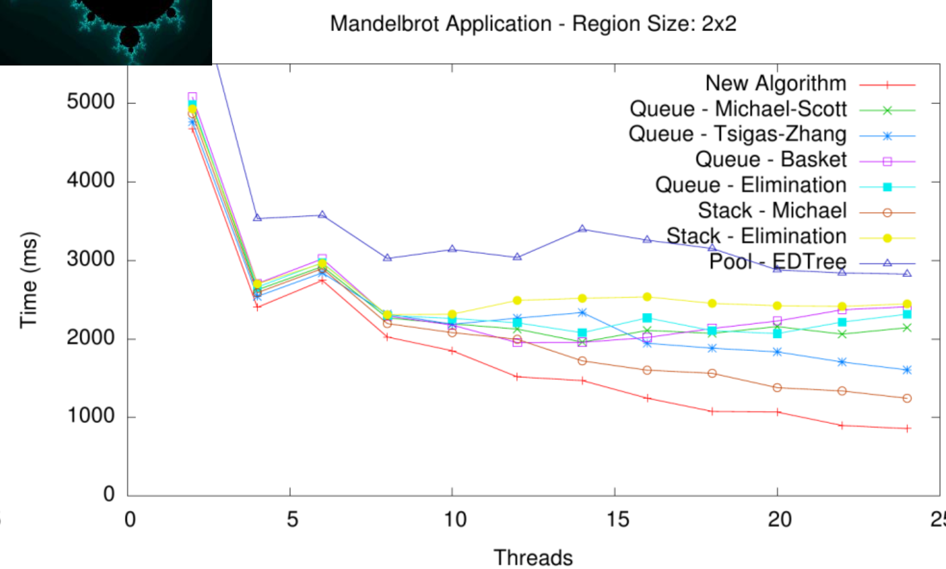


Experimental evaluation (iv)



Experimental evaluation (v)
Parallel application for the Mandelbrot set





- 16x16 chunks: Large work units
  => Low contention on the shared data structure (bag)
- 2x2 chunks: Small work units
  => High contention. The bag implementation matters

Conclusions

- Lock free and linearizable algorithm for a concurrent bag producer/consumer collection data structure
  - Distributed design, promoting access-parallelism.
  - Exploiting thread-local static storage.
  - Dynamic in size via lock-free memory management.
  - Only requires atomic primitives available in contemporary systems.

Thank you for listening!

Questions?