Data Structures in Work-Stealing

Daniel Cederman and Philippas Tsigas
Distributed Computing and Systems
Chalmers University of Technology

This project is part of the portfolio of the
G.3 - Embedded Systems and Control Unit
Information Society and Media Directorate-General
European Commission

Contract Number: 248481
Total Cost [€]: 3.44 million
Starting Date: 2010-01-01
Duration: 36 months

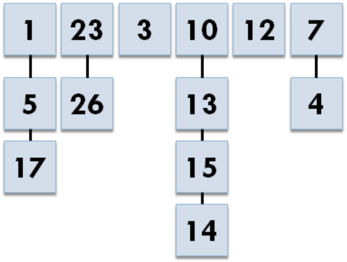
SEVENTH FRAMEWORK PROGRAMME

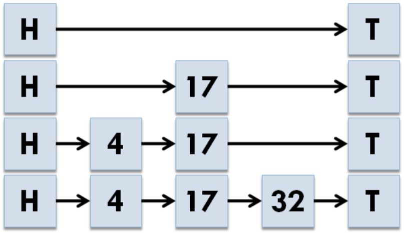
Overview

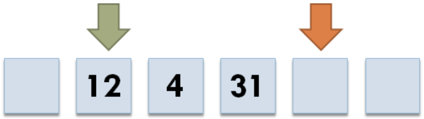
- Our part in PEPPHER
  - Provide library of **lock-free** data structures
- Prior work on load balancing on graphics processors
  - Compared **blocking** global queue synchronization with a **non-blocking**
  - Compared with **non-blocking work-stealing** scheme
  - Auto-tuning of application
- Data structures in work-stealing
  - Why and how?
  - Why is non-blocking important for graphics processor?
  - Queues, stacks and deques – How do they match up?
- Conclusion and further work

PEPPHER

- Generic **lock-free** data structures for component programmers and the PEPPHER run-time
  - Queues
  - Stacks
  - Dictionaries
  - Skip-lists
  - Priority Queues
  - ...
- Adapted to heterogeneous systems where possible
- Optimal implementation selected by run-time system







Why Lock-Free?

- Mutual exclusion
  - Locks limits concurrency
  - Busy waiting – repeated checks to see if lock has been released or not
  - Convoying – processes stack up before locks

- Lock-freedom is a **progress guarantee**
- In practice it means that
  - A fast process doesn't have to wait for a slow or dead process
  - No deadlocks
- Shown to **scale better** than blocking approaches

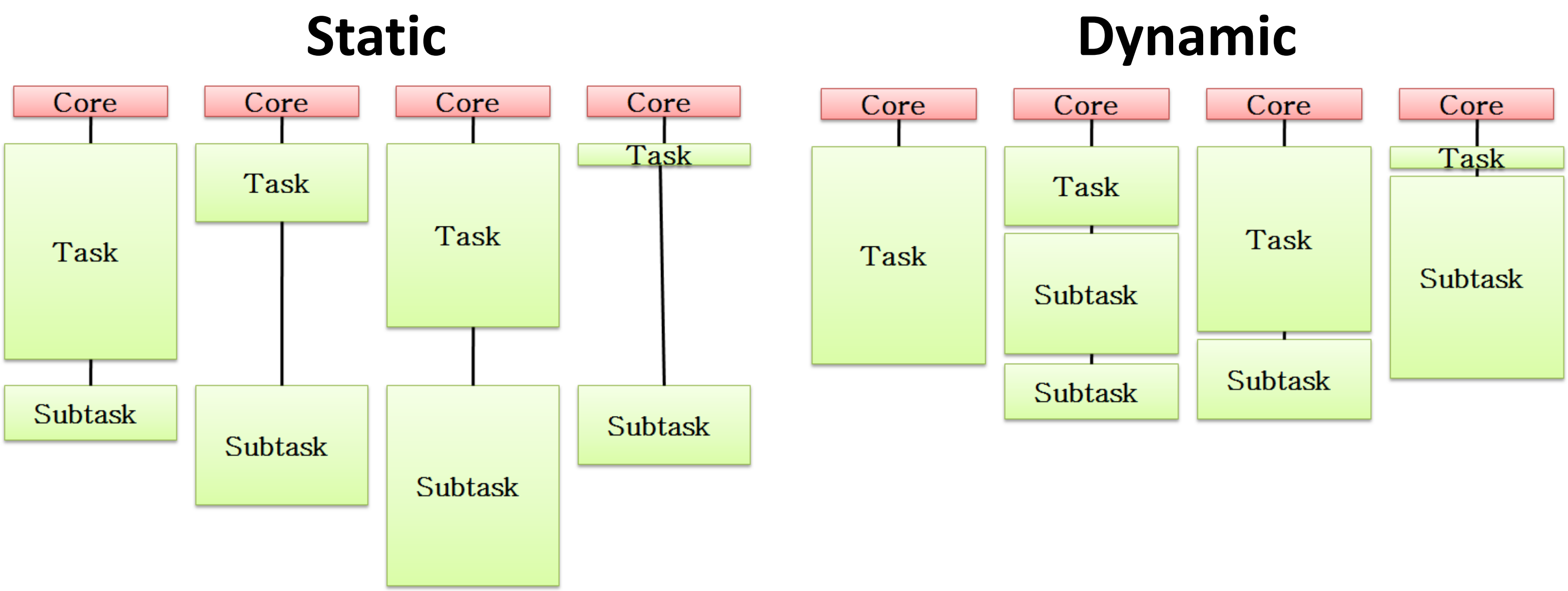
Definition

For all possible executions, **at least one** concurrent operation will **succeed** in a **finite** number of its own steps

Dynamic Load Balancing on GPUs

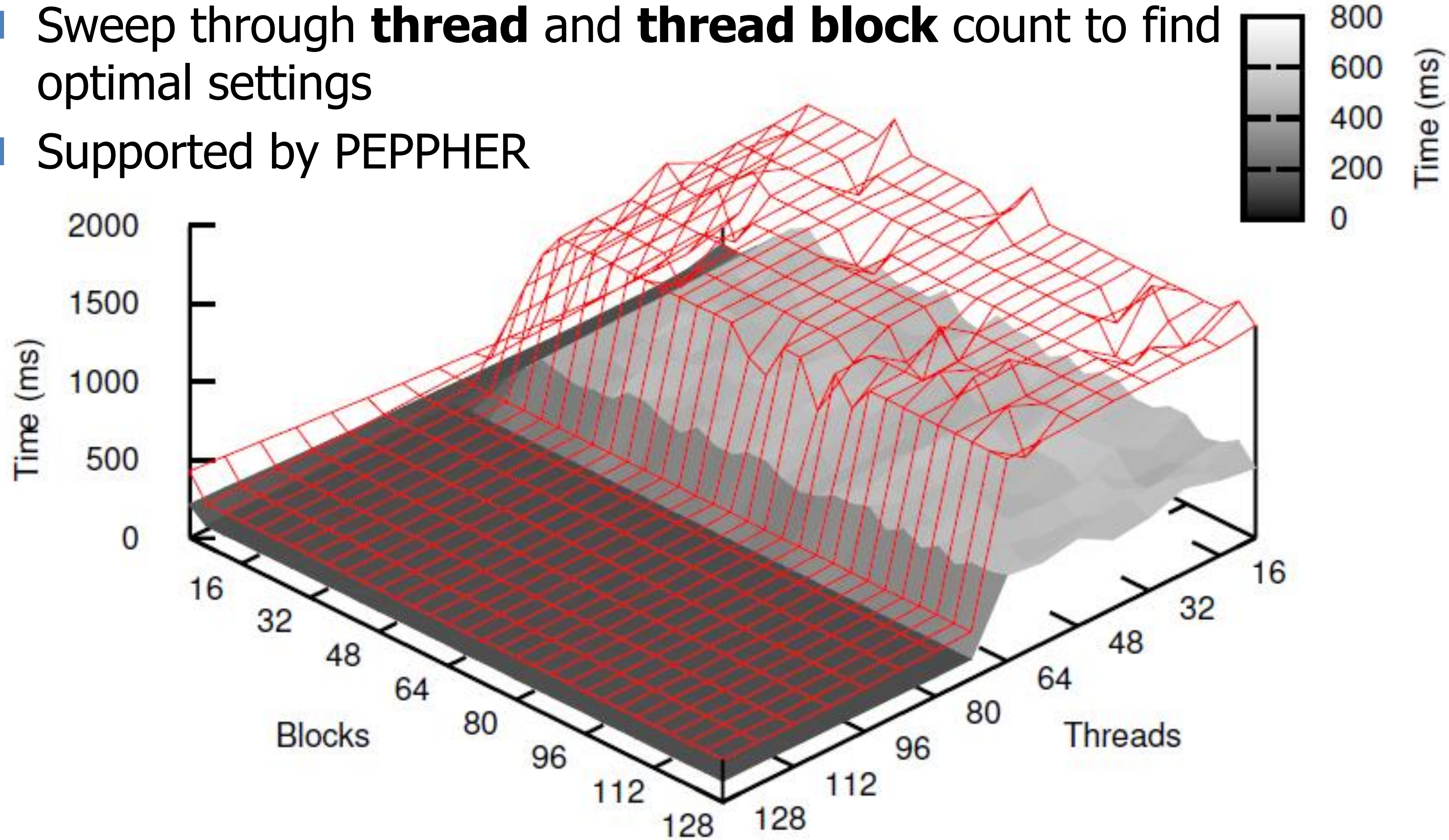
PEPPHER

- In earlier work we have compared different load balancing schemes on graphics processors
- We asked the question: can **dynamic** load balancing using a **single global queue** improve performance over static load balancing
- And: **blocking** or **lock-free?** Does it make any difference



Auto-tuning

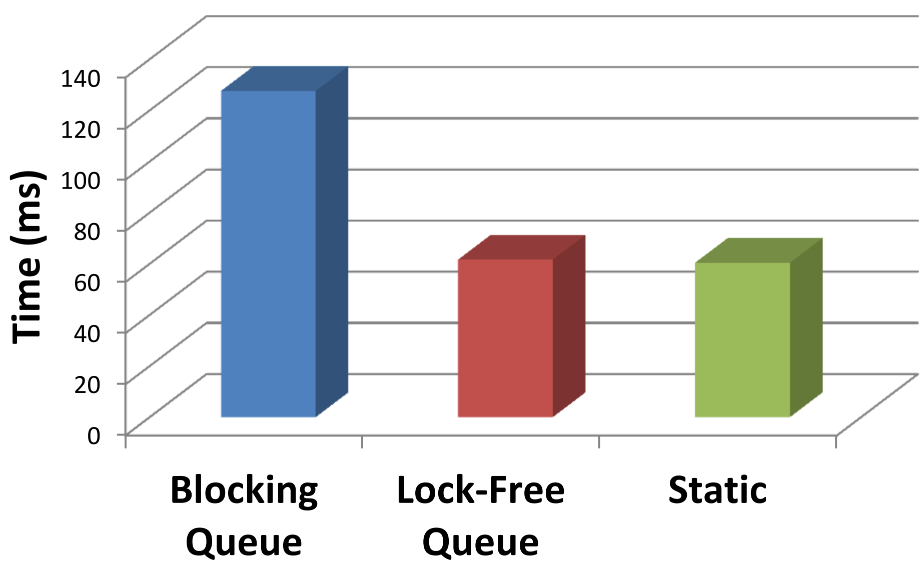
- Sweep through **thread** and **thread block** count to find optimal settings
- Supported by PEPPHER



Blocking queue on a 9600GT using two different distributions

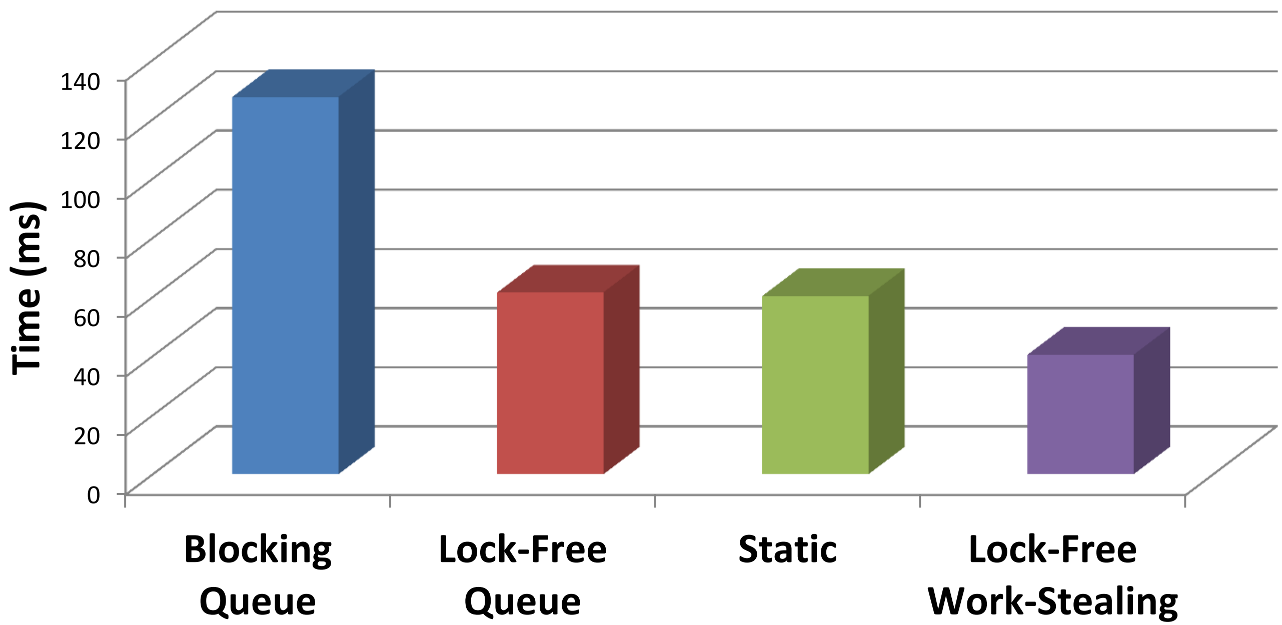
Dynamic Load Balancing on GPUs

- Results showed that the **lock-free synchronization** outperformed the **blocking** one
- But the result was similar to static load balancing
- We then compared the global queue approach with a **lock-free work-stealing** scheme



Dynamic Load Balancing on GPUs

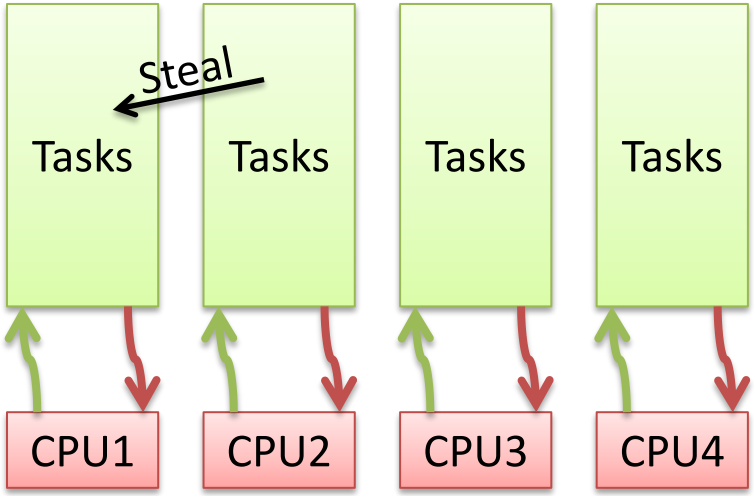
- We found that work-stealing could **perform much better** than static load balancing
- But how much does the **type** of data structure used within the work-stealing scheme affect the result?



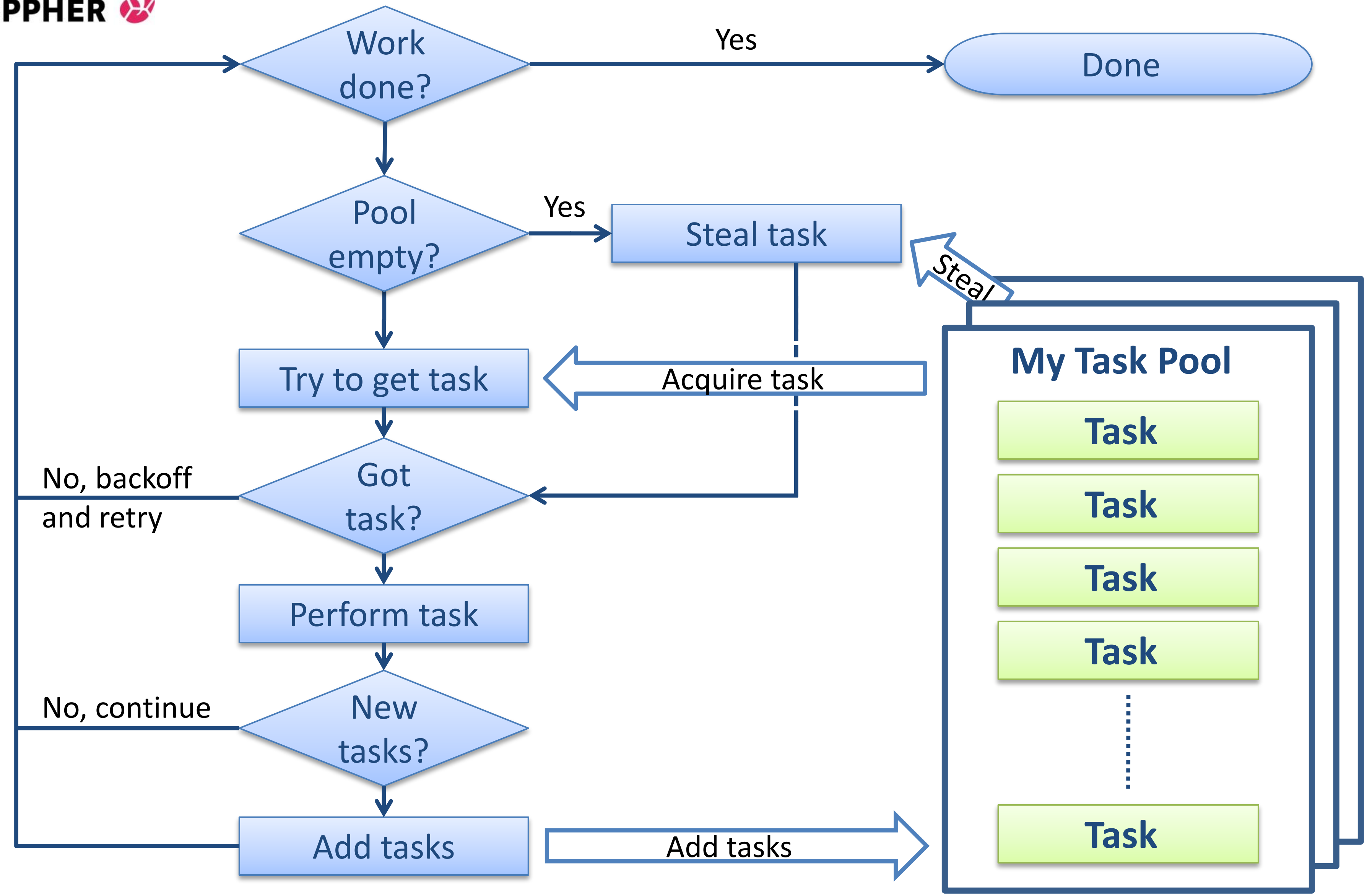
Work-Stealing

Work-stealing – Why and how?

- Main idea
  - Each processing unit has a local task pool
  - When the local task pool is empty, try stealing from another pool
- Lower communication and synchronization cost
  - Steals are rare
  - Single enqueuer
- Task locality
  - Better cache use
  - Don't need to move or generate data as often



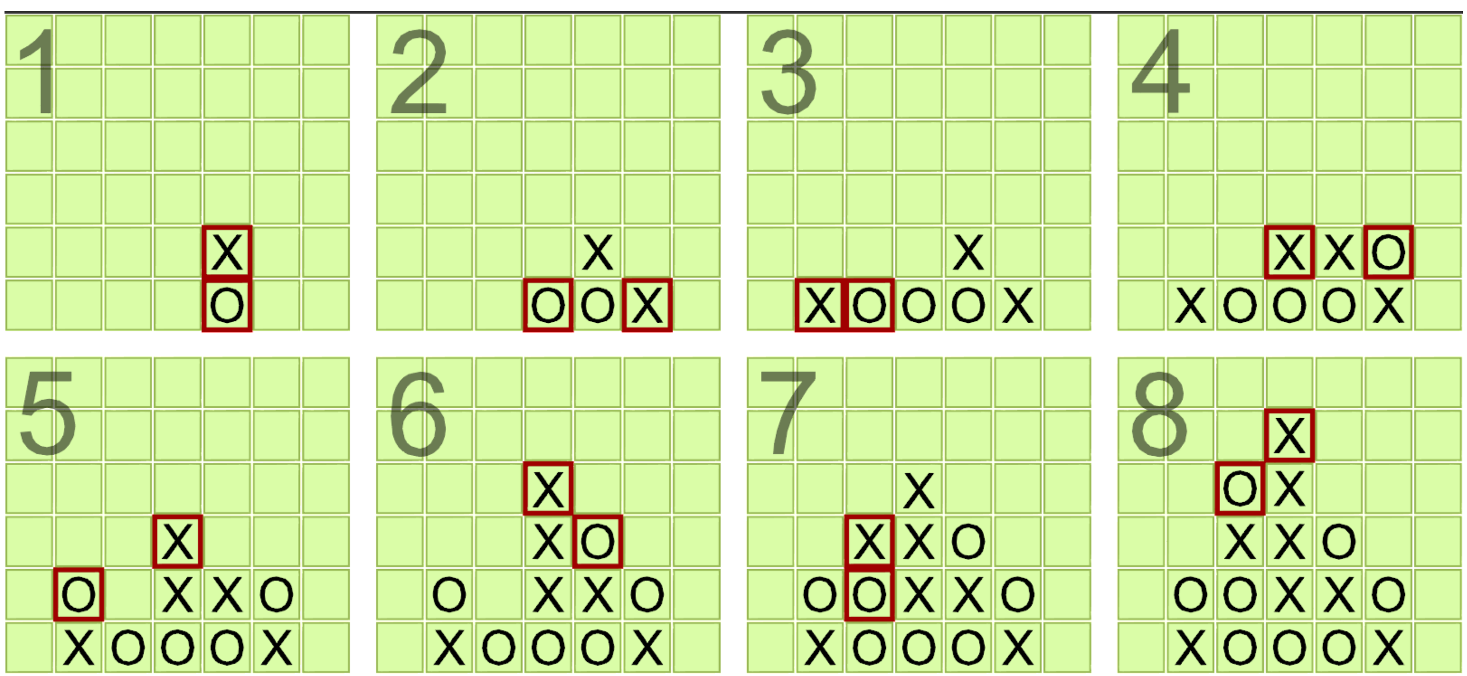
Work-stealing Scheme



Applications

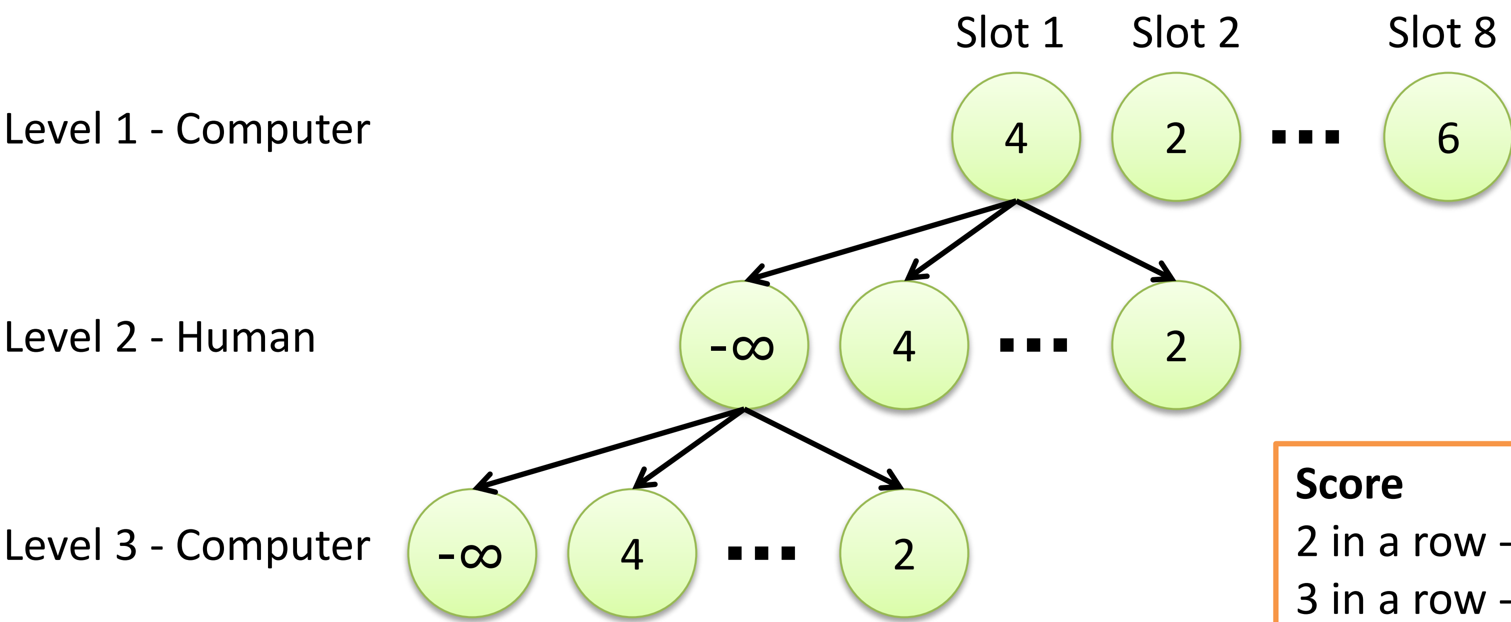
Four-in-a-row

- Computer opponent
- Move decided by looking *n* steps ahead using a minimax algorithm



Four-in-a-row - Details

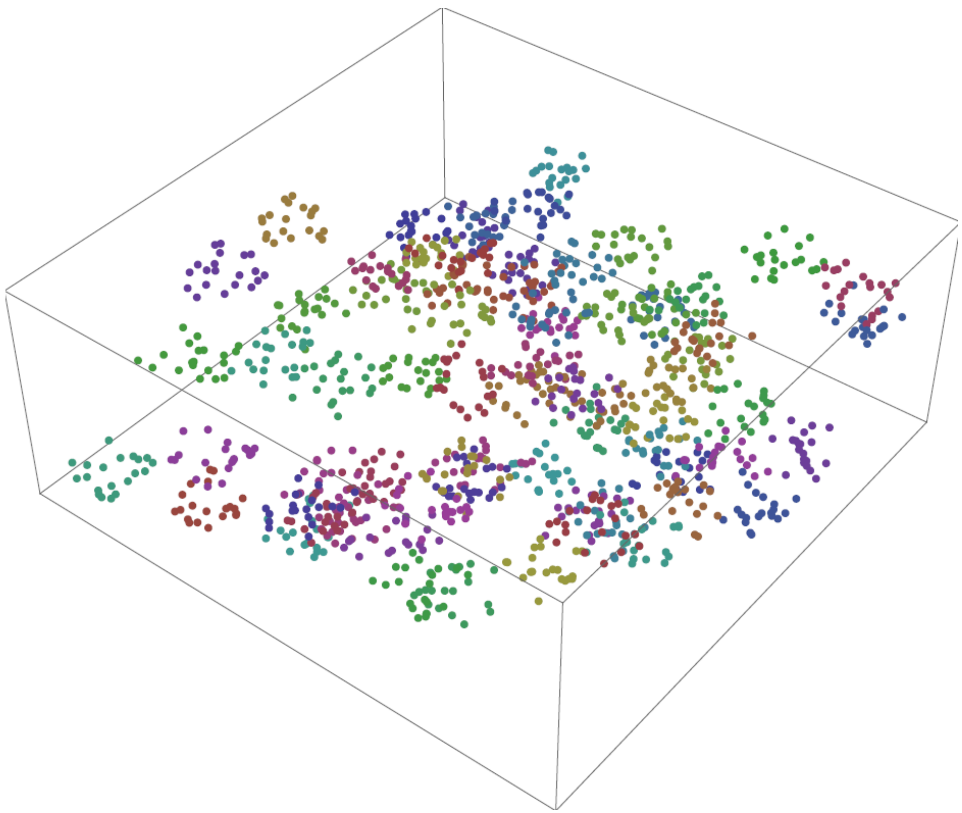
- Every child represents a move by either the computer or a human player
- When no move is possible or the cut-off depth has been reached, use a heuristic to calculate a score
- Propagate results upward assuming both players play optimal



Score
2 in a row -> 2p or -2p
3 in a row -> 4p or -4p
4 in a row -> ∞ or -∞

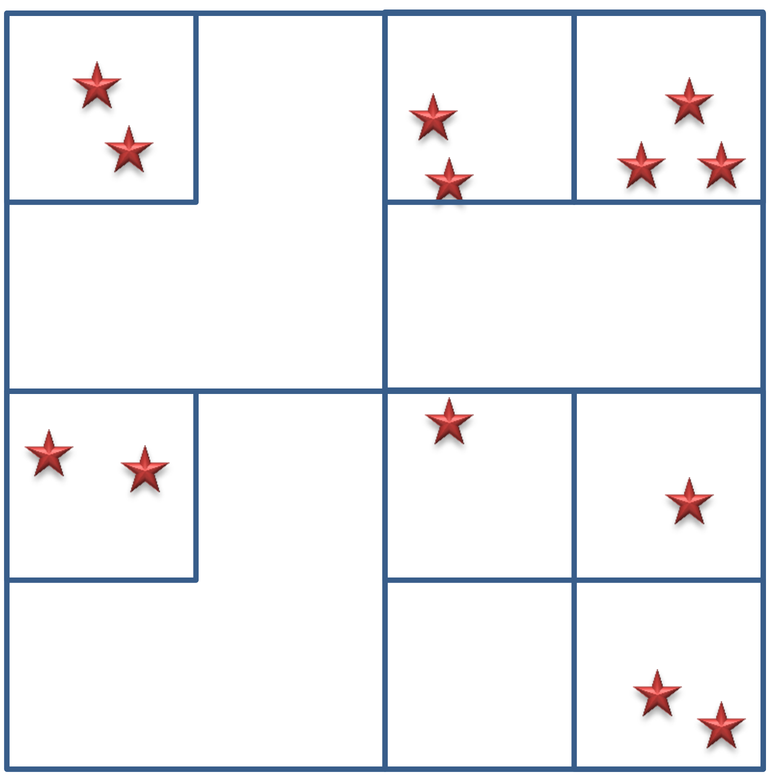
Octree Partitioning

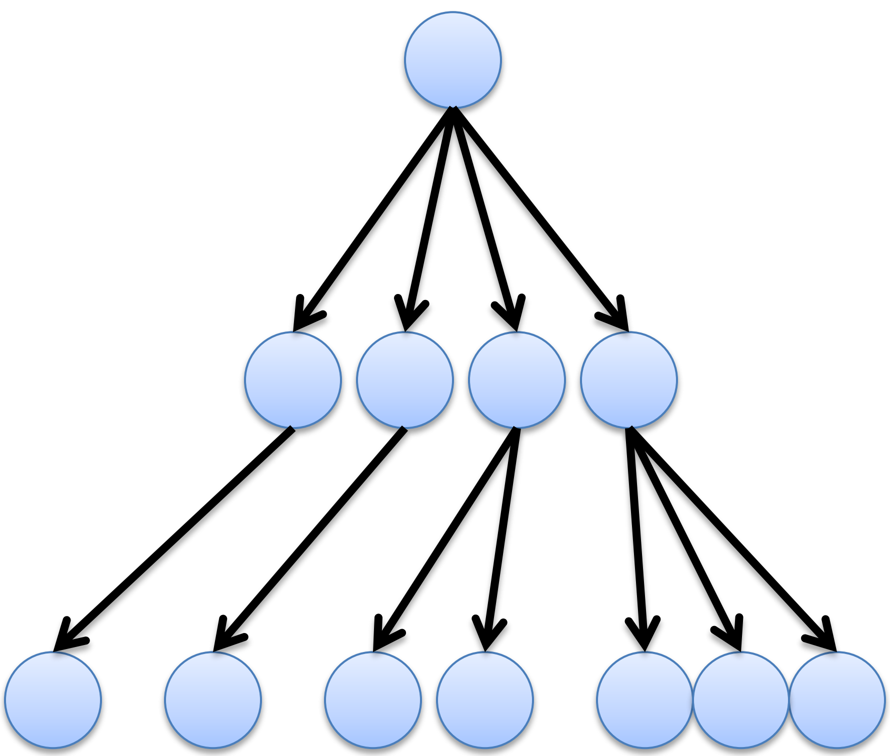
- Recursively divide a set of particles in each dimension to create octants
- Stop when less than *n* elements in the octant



Octree Partitioning - Details

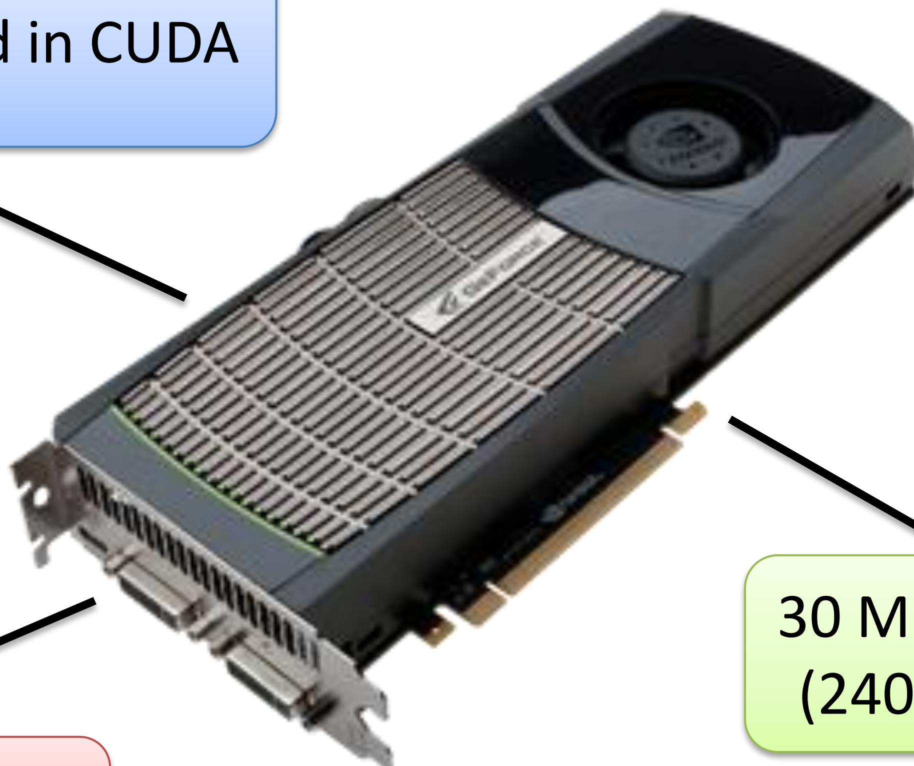
- Count the number of elements that go to each octant
- Use prefix sum to find their correct destination
- Move elements and create up to eight new sub-tasks if necessary





Hardware

Implemented in CUDA



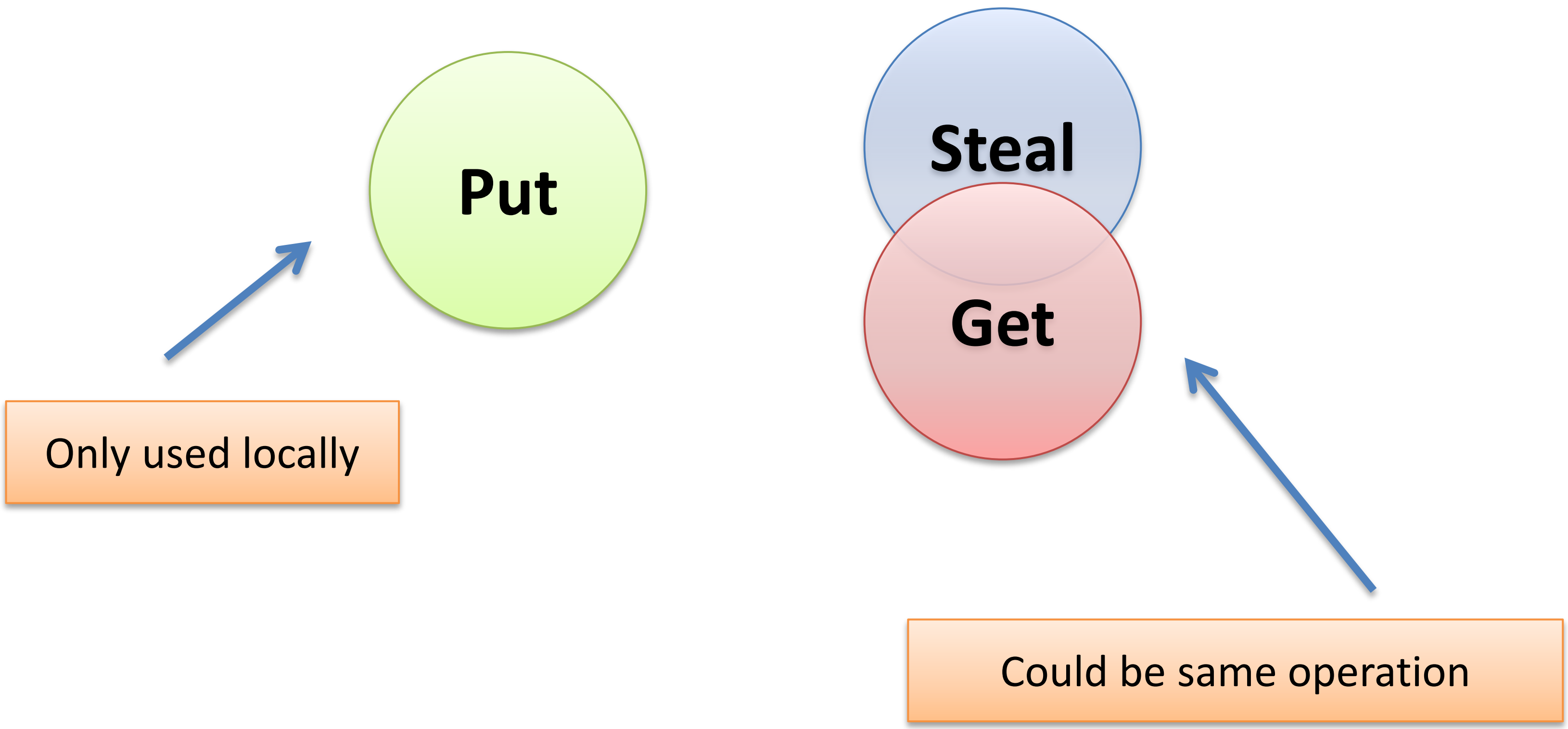
30 Multiprocessors (240 CUDA cores)

Nvidia GTX 480

Data Structures in Work-Stealing

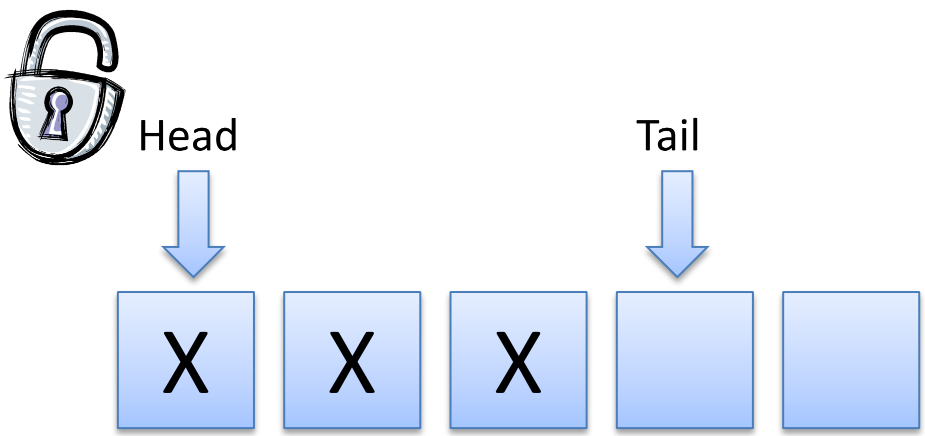
Task Pool Operations

- Two (or three) basic operations



Lock-based Queue

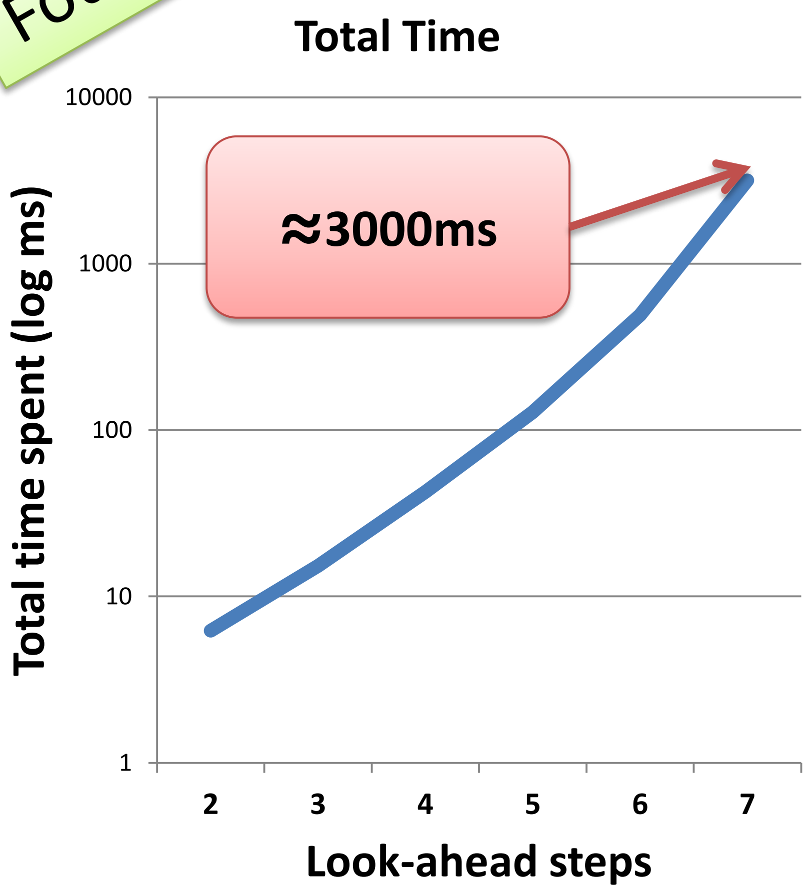
- Circular array
- **Get** operation protected by lock
- Single enqueuer
- Thief tries to acquire lock **once**

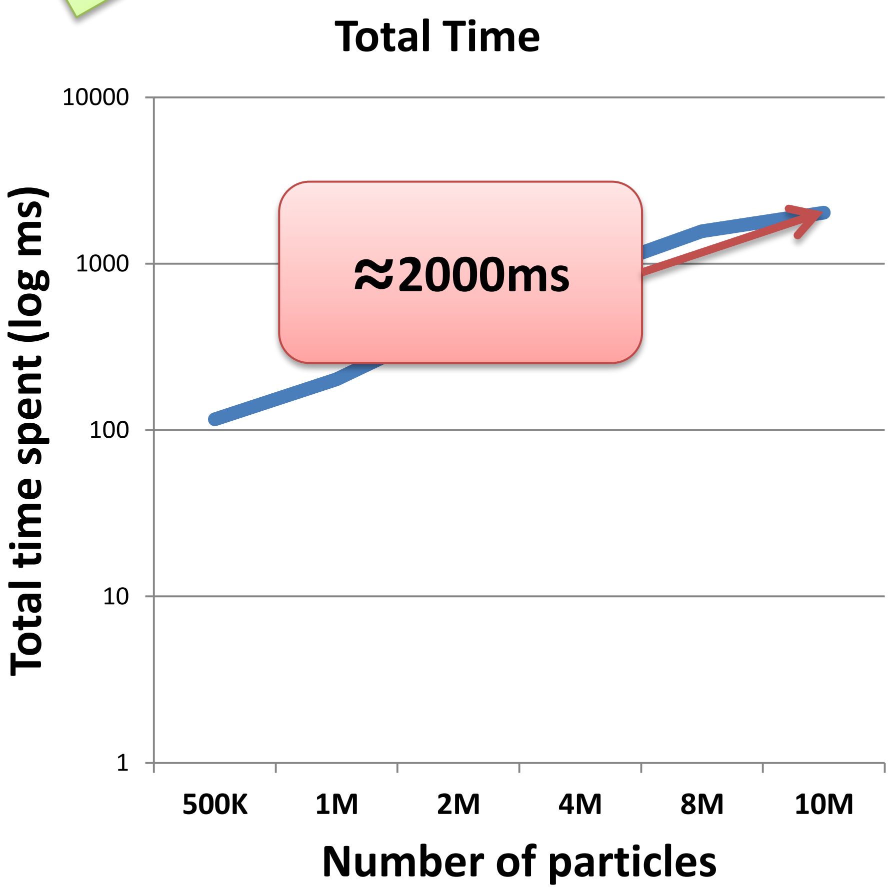


Lock-based Queue - Results

Four-in-a-row

Octree





Locks Not Supported on GPUs

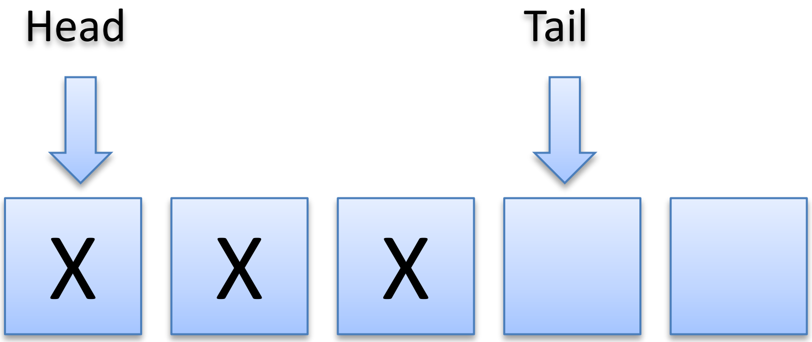
- **Fairness** of hardware scheduler **unknown**
- Thread block holding the lock might be swapped out **indefinitely**
- Locks are discouraged in CUDA and OpenCL

- Locks limit concurrency
- Busy waiting expensive

- Highly disjoint memory access in work-stealing

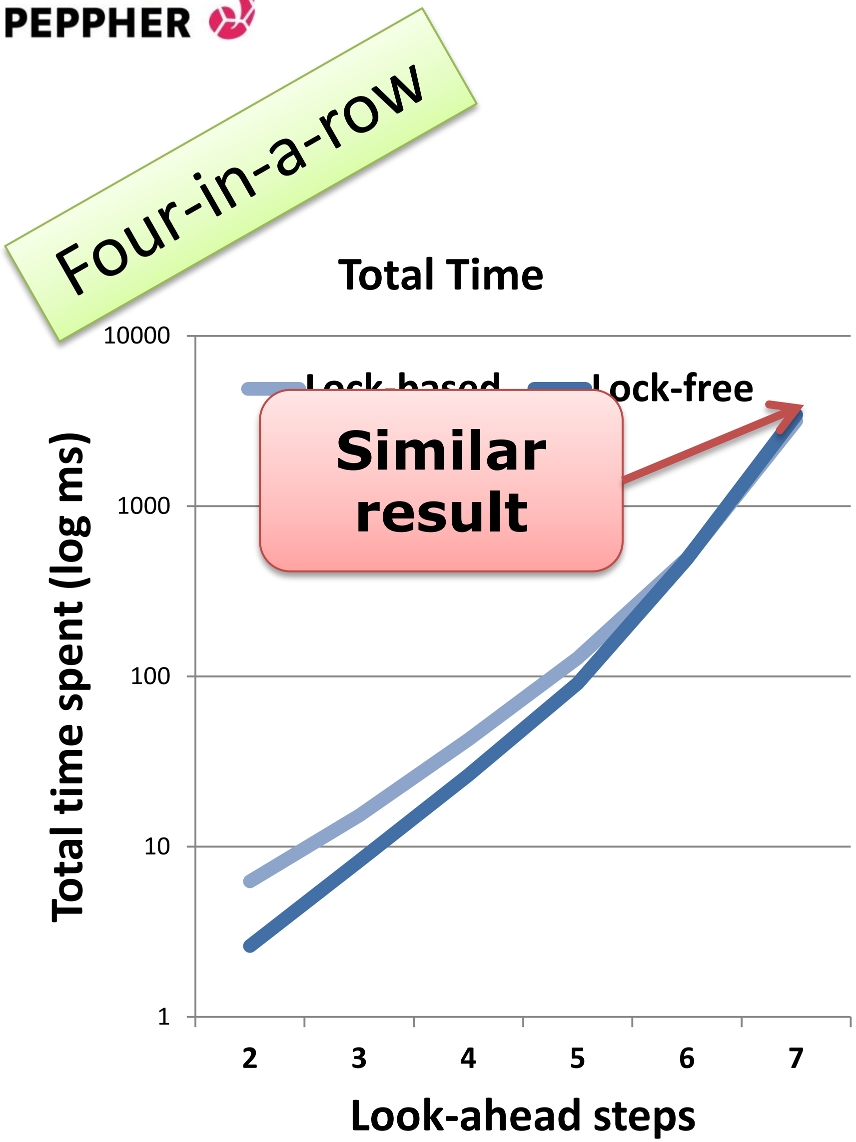
Lock-free Queue

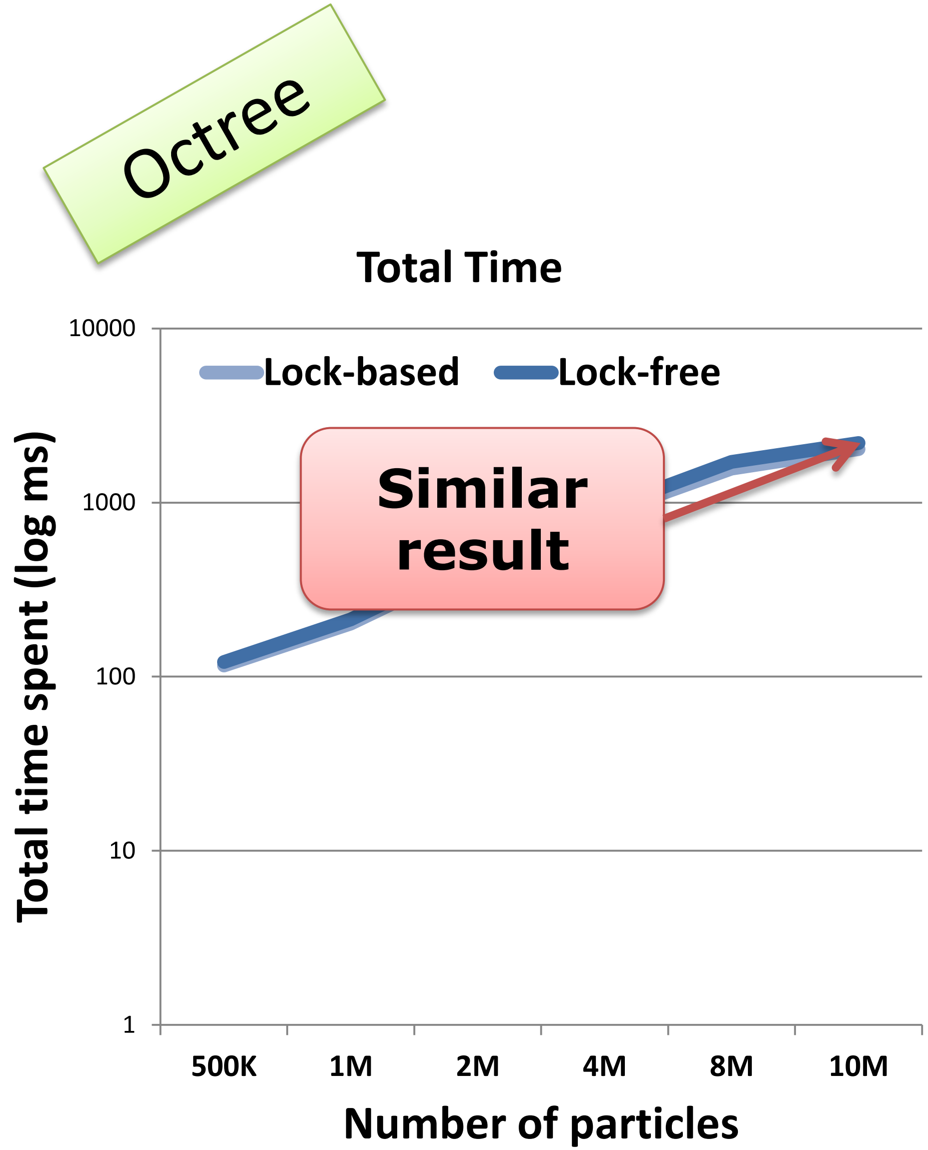
- Algorithm by Yi and Tsigas
- Circular array
- Lazy head and tail update



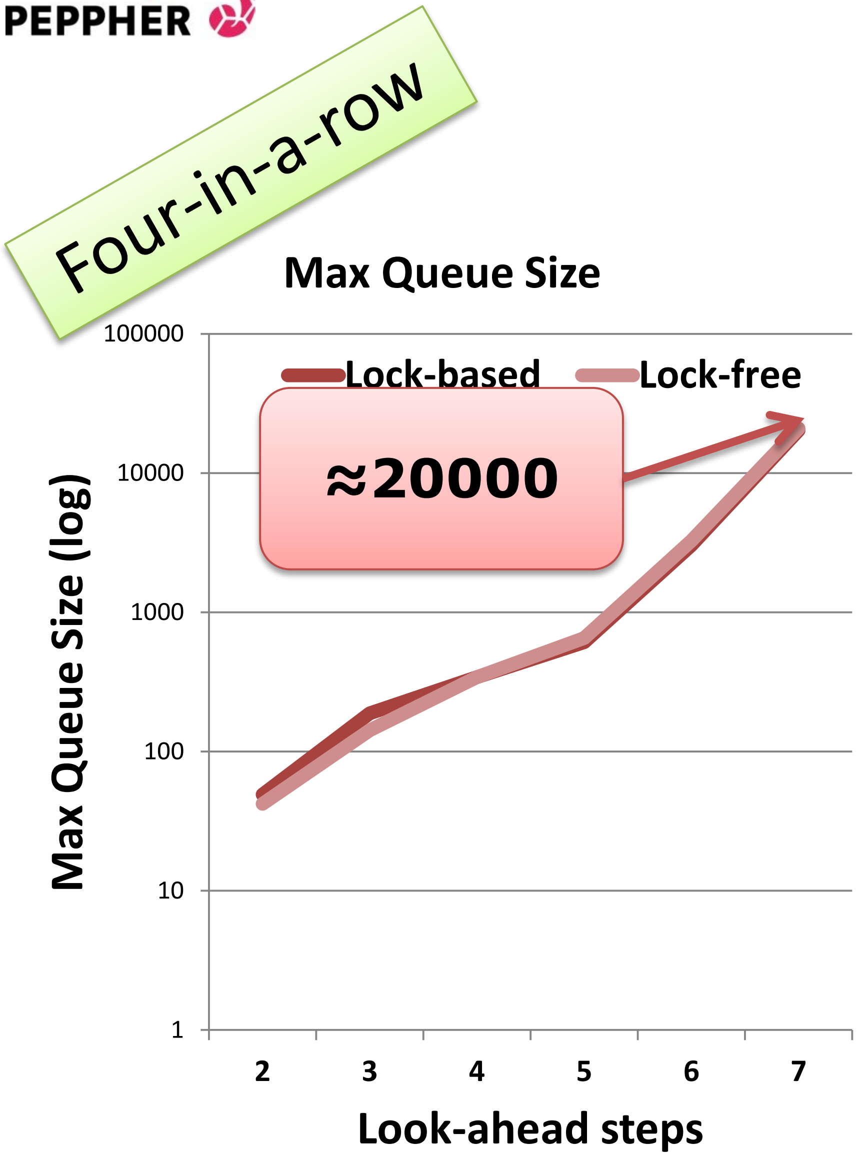
Lock-free Queue - Results

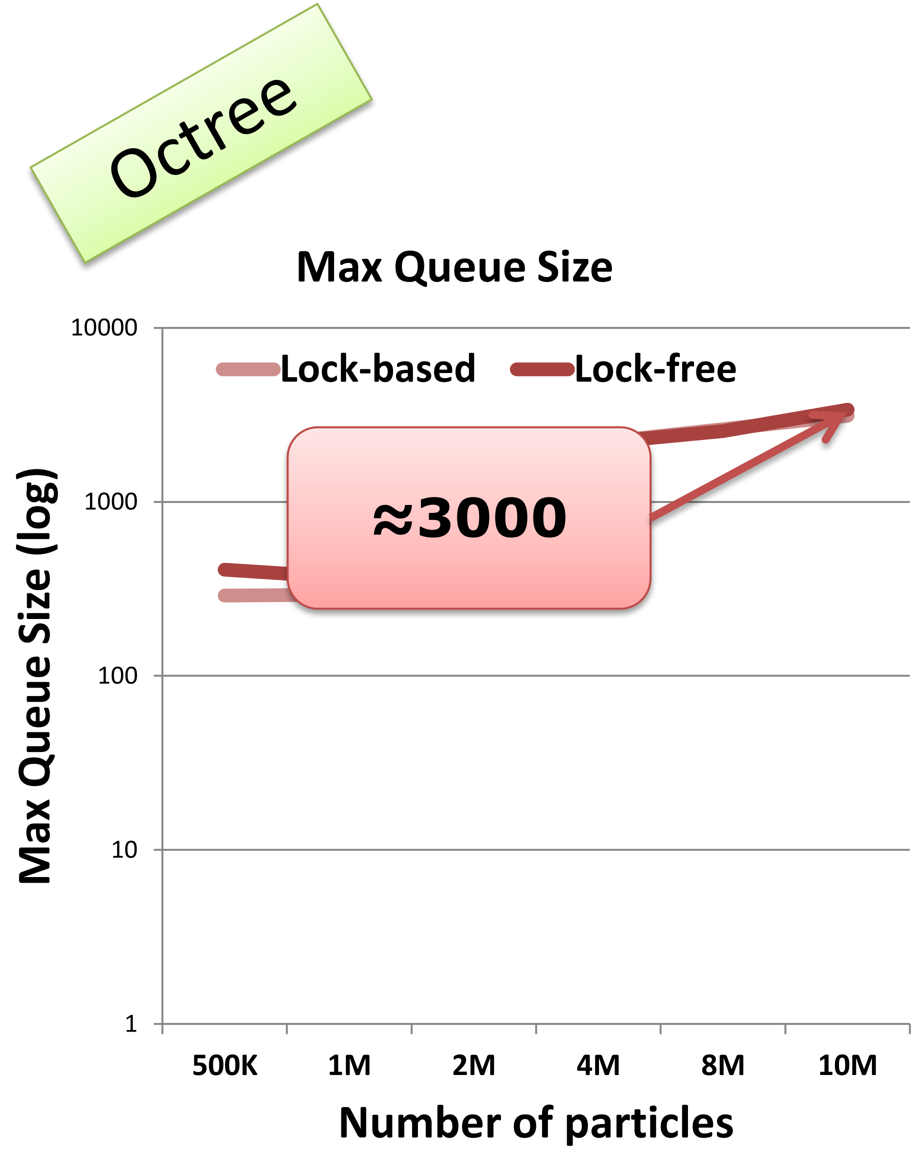
PEPPHER



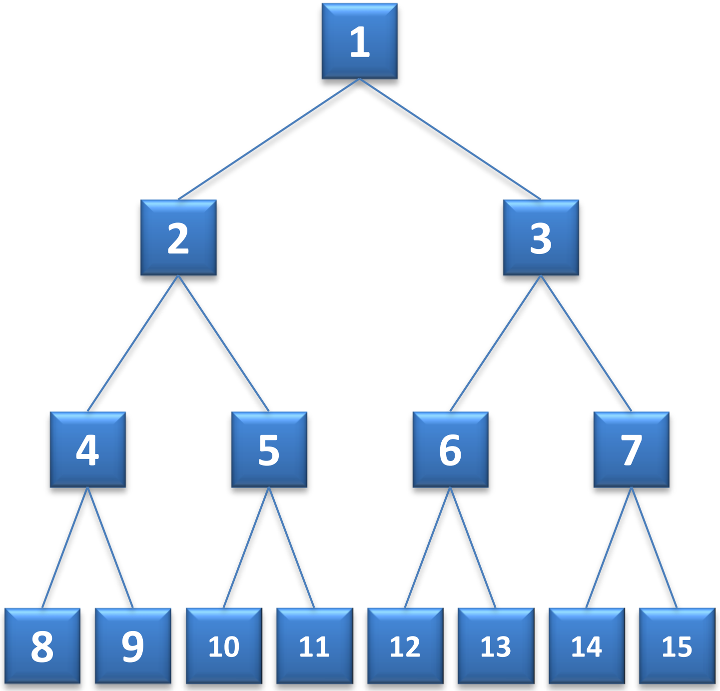


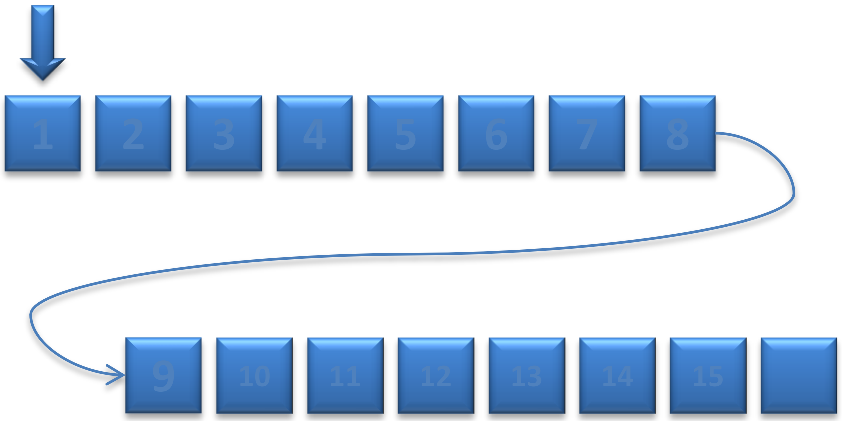
Lock-free Queue - Results





Why so many tasks?



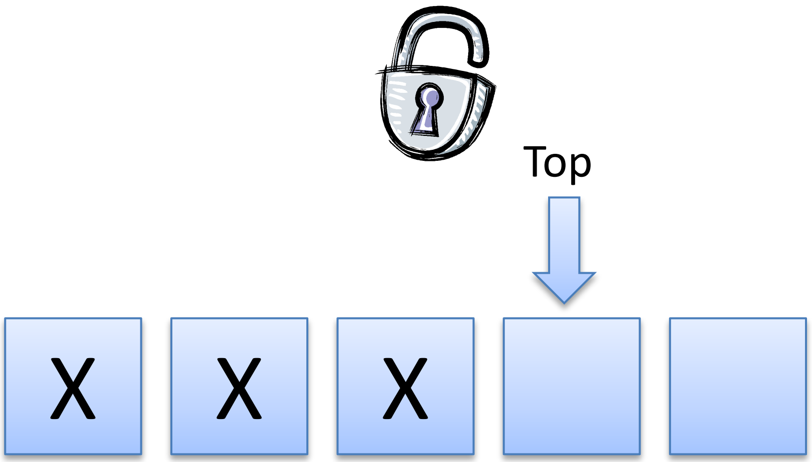


Breadth-First

Do It Depth-First Instead

Lock-based Stack

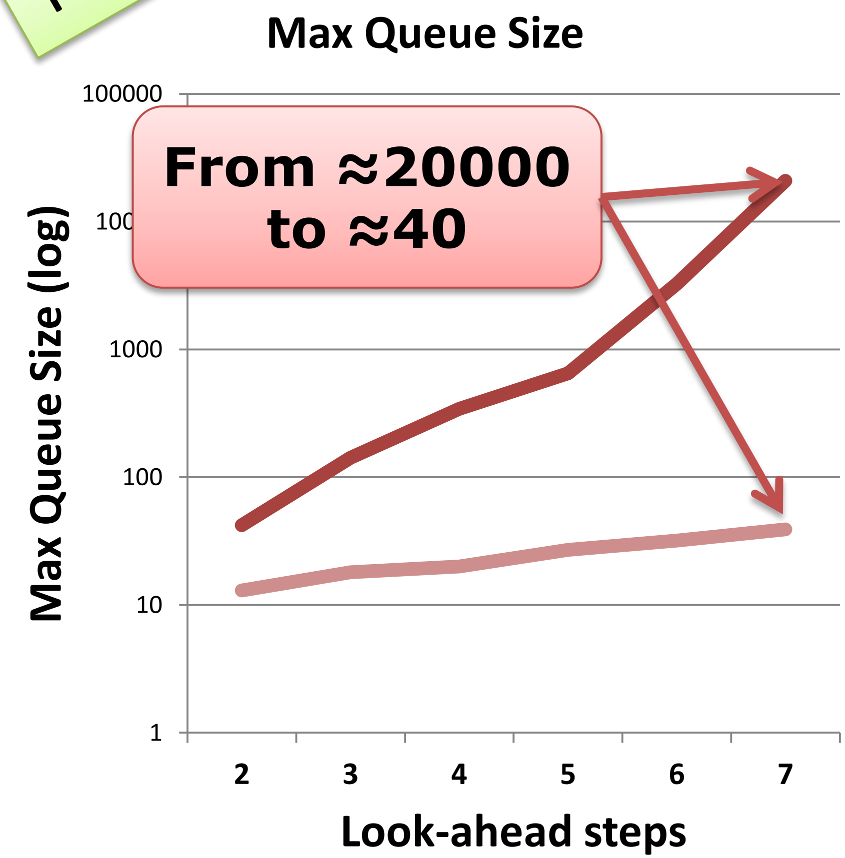
- **Get/Put** operation protected by lock
- Single enqueuer gives no benefit
- Thief tries to acquire lock **once**

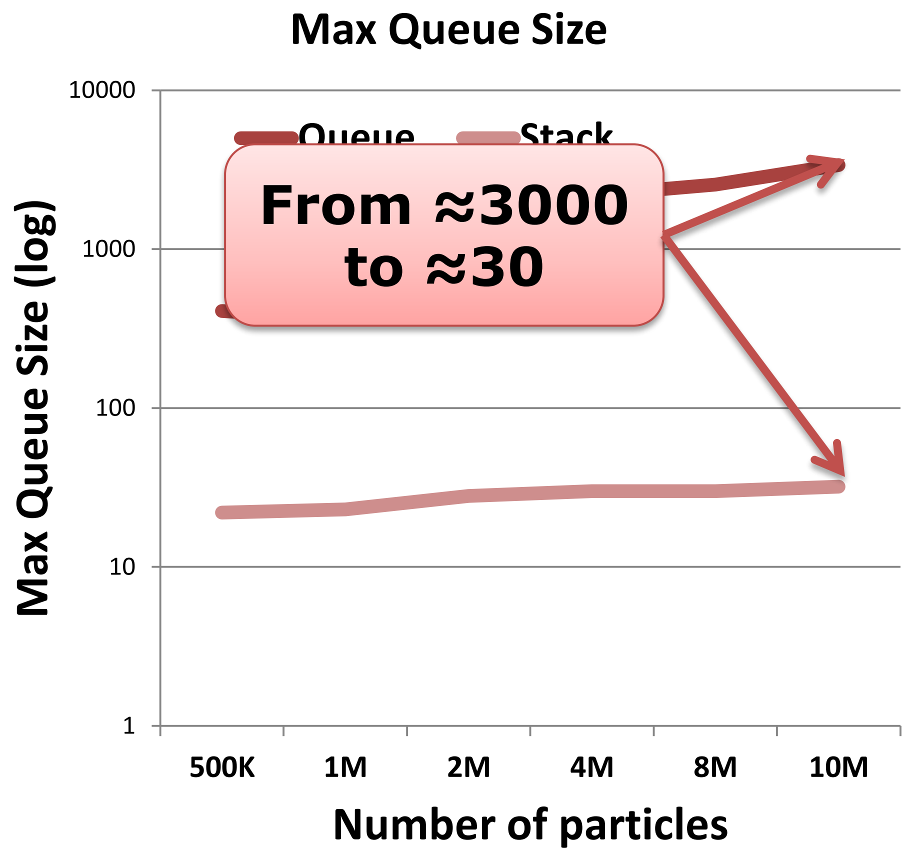


Lock-based Stack - Results

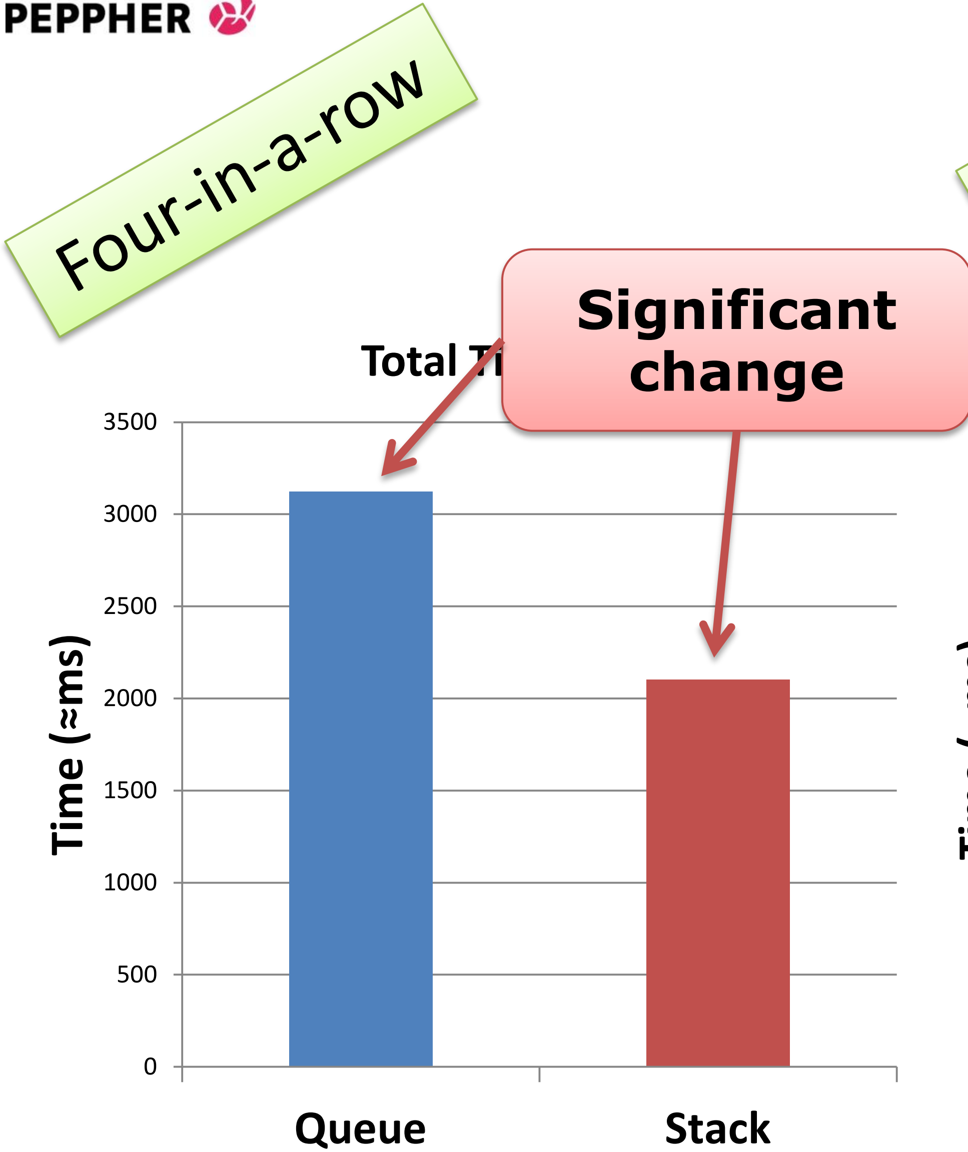
Four-in-a-row

Octree



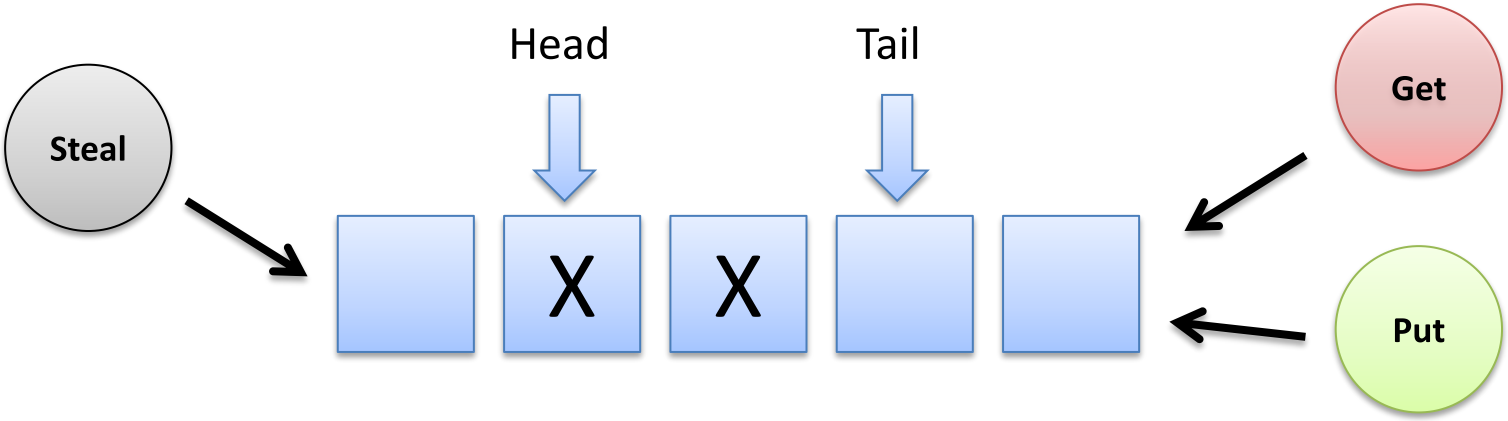


Lock-based Stack - Results



Lock-free Deque

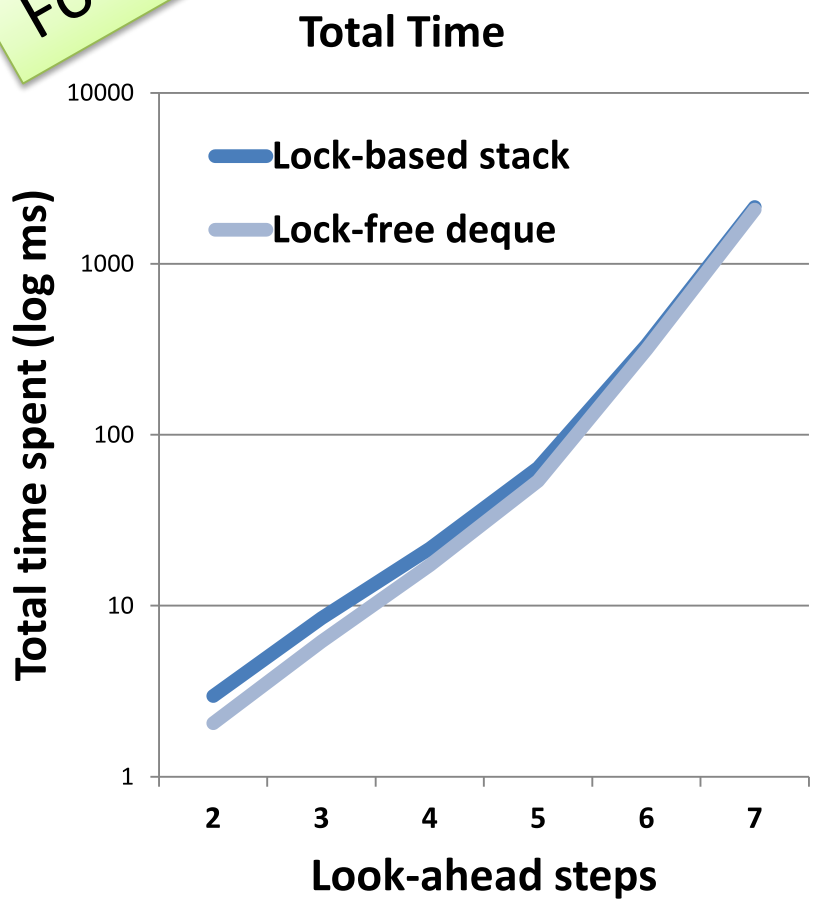
- Algorithm by Arora et al.
- Local *get* is FILO (short queue), *steal* is FIFO (many children)
- *Steal* always uses CAS, *get* only when on last element

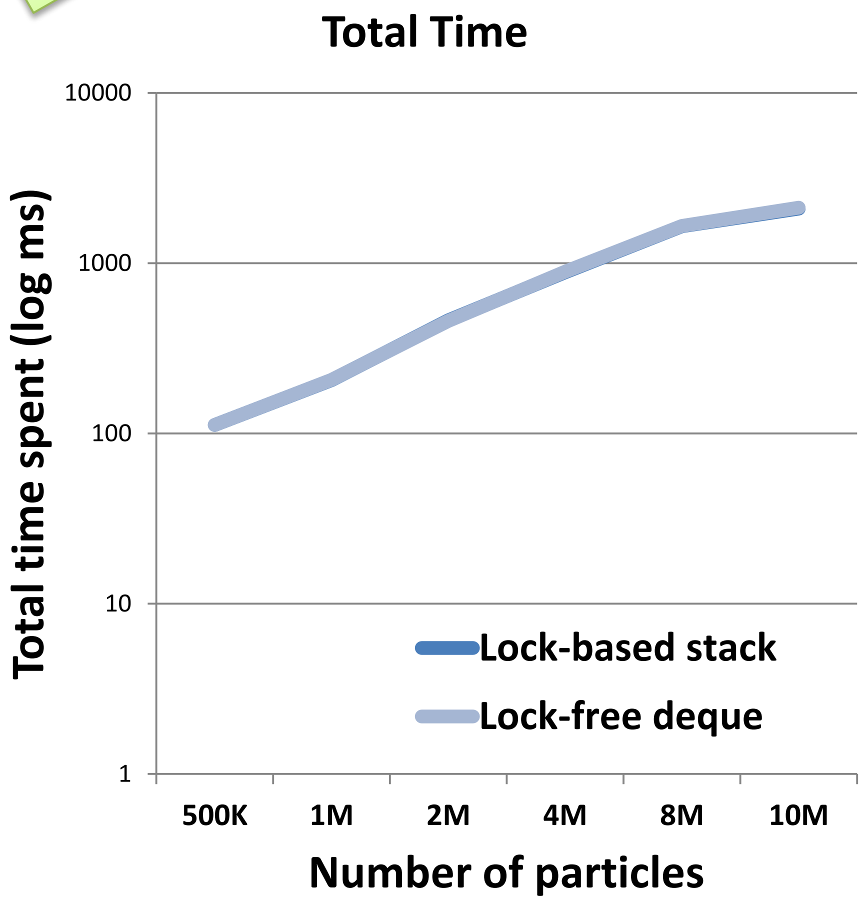


Lock-free Deque - Results

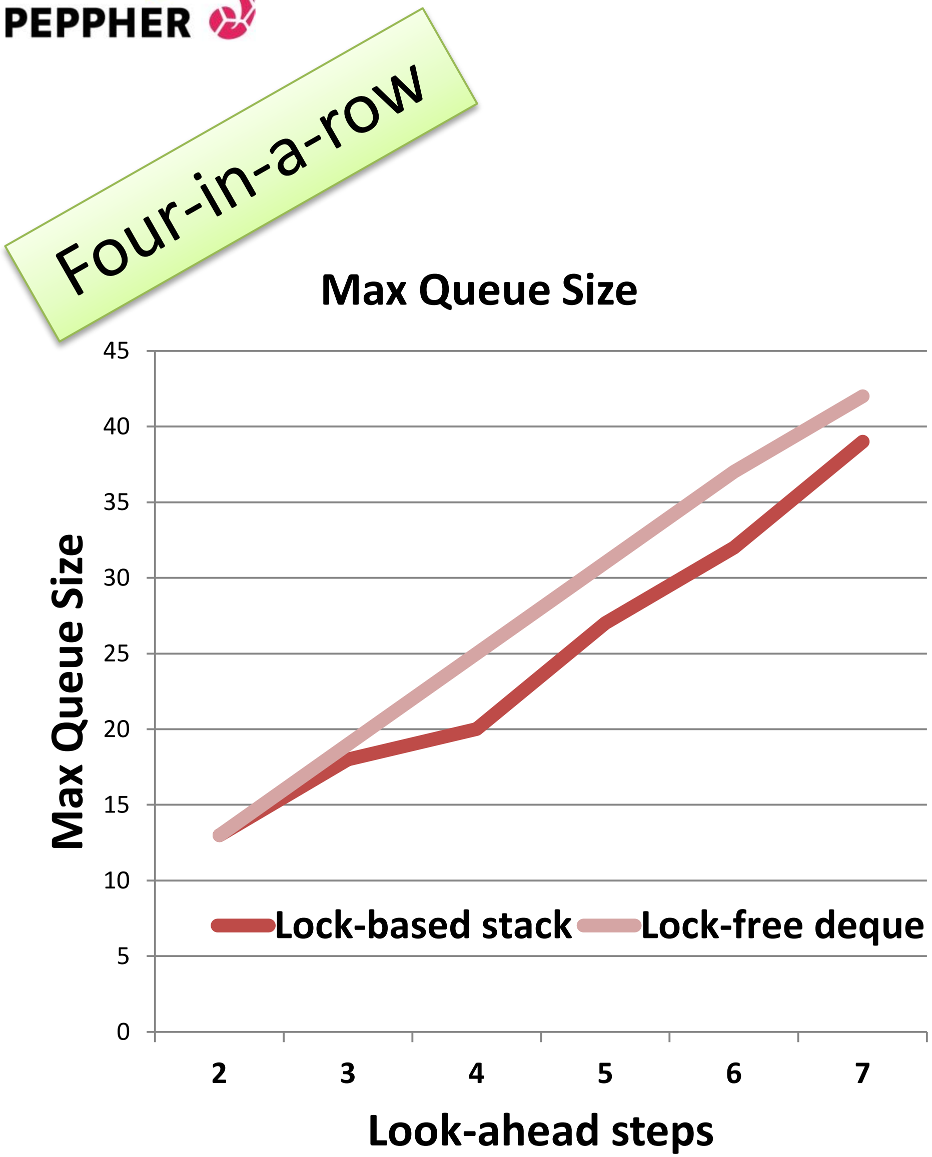
Four-in-a-row

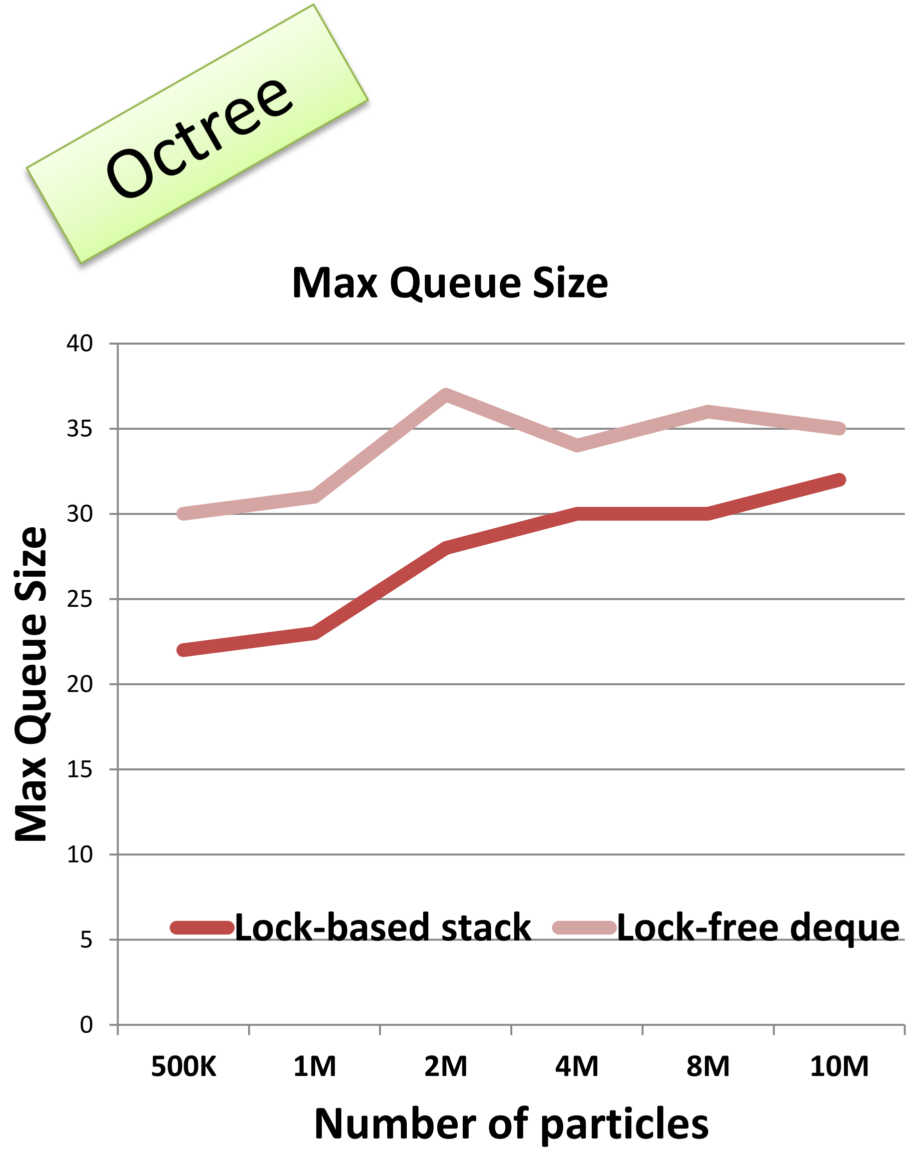
Octree





Lock-free Deque - Results





Conclusions and further work

PEPPHER

- Lock-free data structures are **needed** on GPUs
  - No performance penalty
  - Often significant performance improvements – depending on contention
- No One Type Data Structure Fits All Applications
  - One application improved performance when tasks were performed in FILO order instead of FIFO
  - Different applications benefit from different behavior of the data structure, which in turn requires different lock-free data structures
- Further work
  - Dependencies/grouping - Memory management - PEPPHER benchmarks

- Our part in PEPPHER is to provide **generic** lock-free data structures
  - Can be used for work-stealing, but it is *not* the main intent
  - Providing a library for the component programmer *is*
  - Should be *high performance, portable, scalable* and *easy to use*

Thank you for listening!