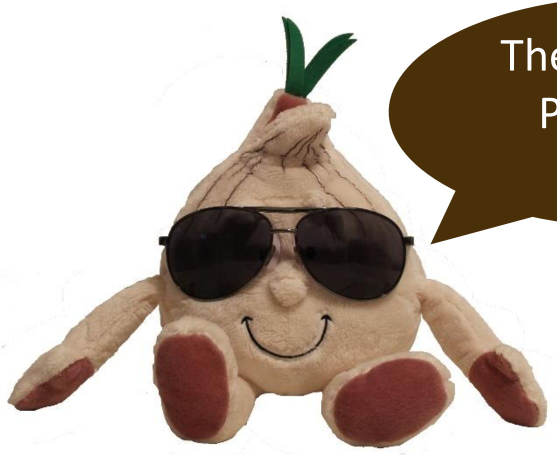
T. HOEFLER, M. PUESCHEL

Lecture 9: Finishing consensus, scalable lock study, and oblivious algorithms

Teaching assistant: Salvatore Di Girolamo

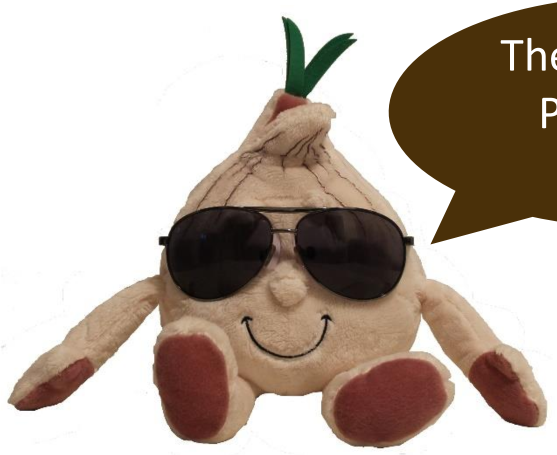
Motivational video: https://www.youtube.com/watch?v=qx2dRIQXnbs

The simplest networking question: ping pong latency!



The latency of
Piz Dora is
1.77us!

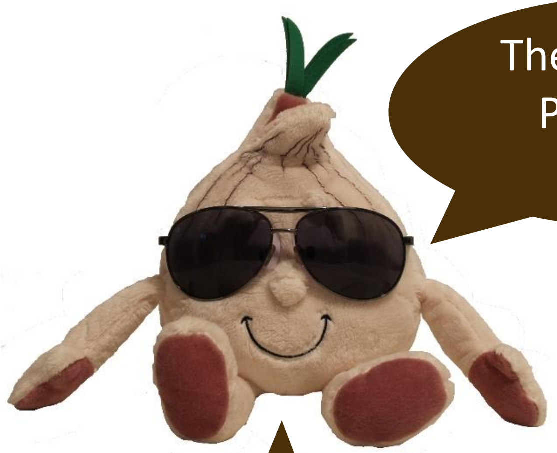
The simplest networking question: ping pong latency!



The latency of
Piz Dora is
1.77us!

How did you
get to this?

The simplest networking question: ping pong latency!

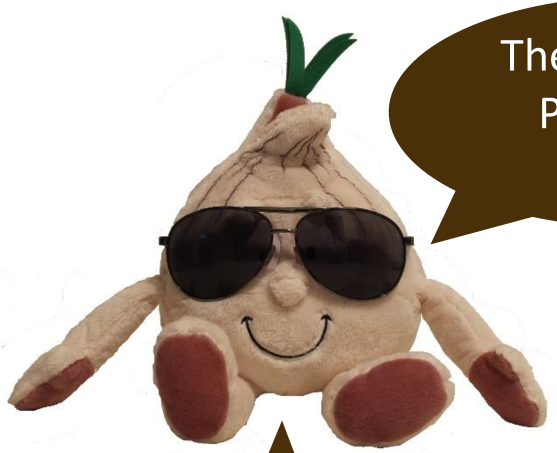


The latency of Piz Dora is 1.77us!

I averaged 10^6 tests, it must be right!

How did you get to this?

The simplest networking question: ping pong latency!



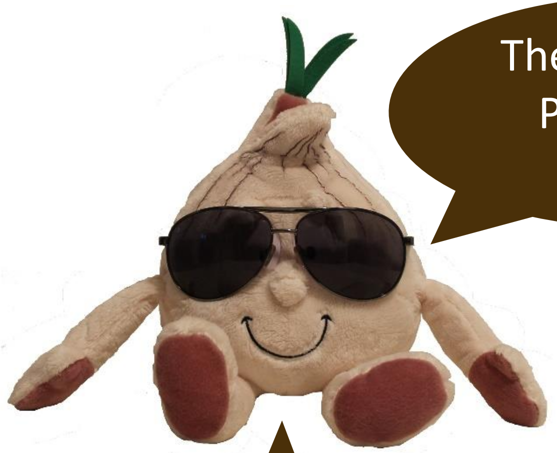
The latency of Piz Dora is 1.77us!

I averaged 10^6 tests, it must be right!

How did you get to this?

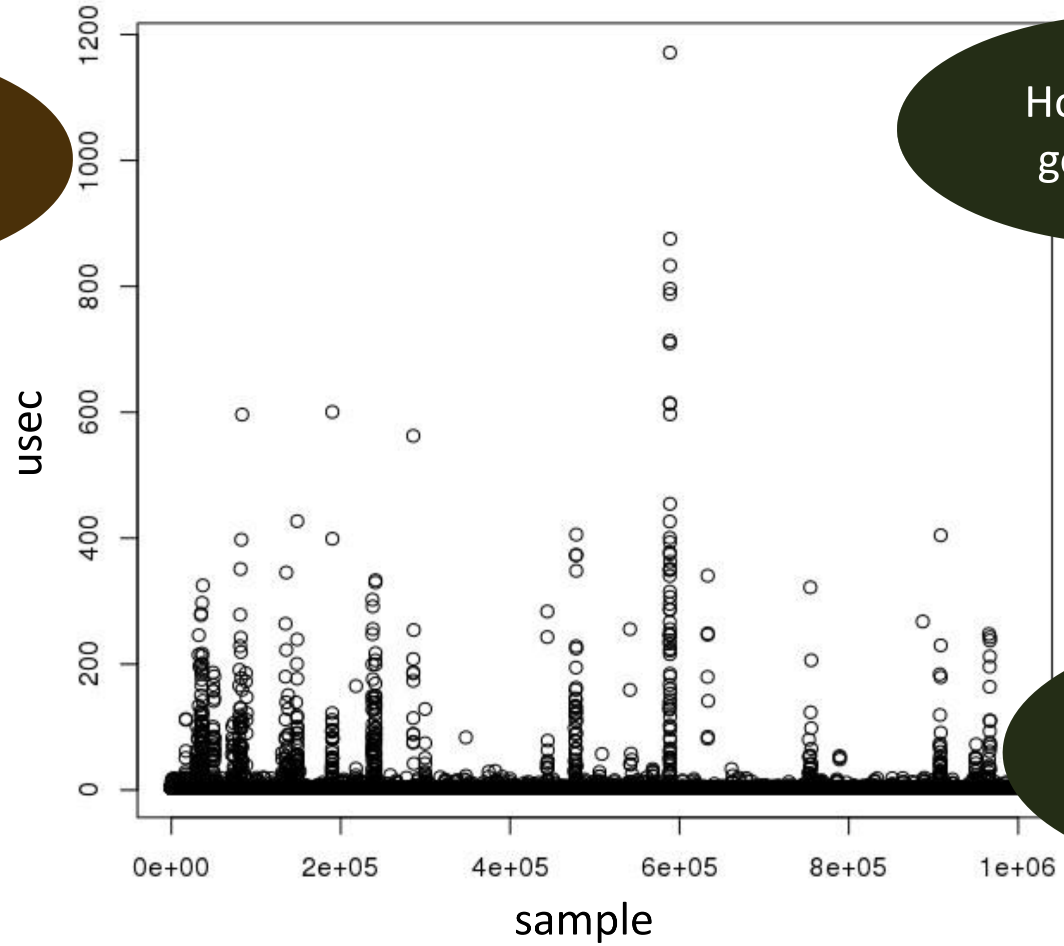
Why do you think so? Can I see the data?

The simplest networking question: ping pong latency!



The latency of Piz Dora is 1.77us!

I averaged 10^6 tests, it must be right!



How did you get to this?

Why do you think so? Can I see the data?

TH, Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15

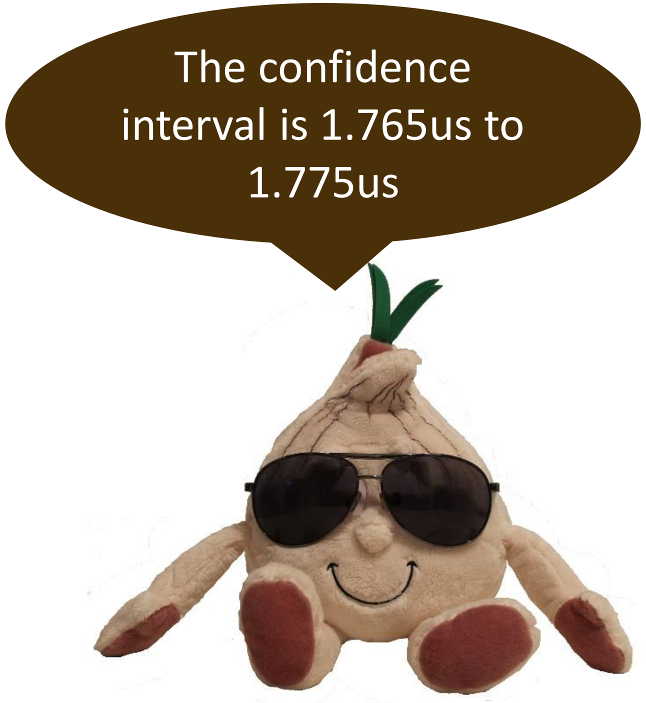
The simplest networking question: ping pong latency!

Rule 5: *Report if the measurement values are deterministic. For nondeterministic data, report confidence intervals of the measurement.*

- **CIs allow us to compute the number of required measurements!**
- **Can be very simple, e.g., single sentence in evaluation:**

  “We collected measurements until the 99% confidence interval was within 5% of our reported means.”

Thou shalt not trust your average textbook!



Thou shalt not trust your average textbook!

The confidence interval is 1.765us to 1.775us

Did you assume normality?

Thou shalt not trust your average textbook!

The confidence interval is 1.765us to 1.775us

Did you assume normality?

Yes, I used the central limit theorem to normalize by summing subsets of size 100!

TH, Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15

Thou shalt not trust your average textbook!

The confidence interval is 1.765us to 1.775us

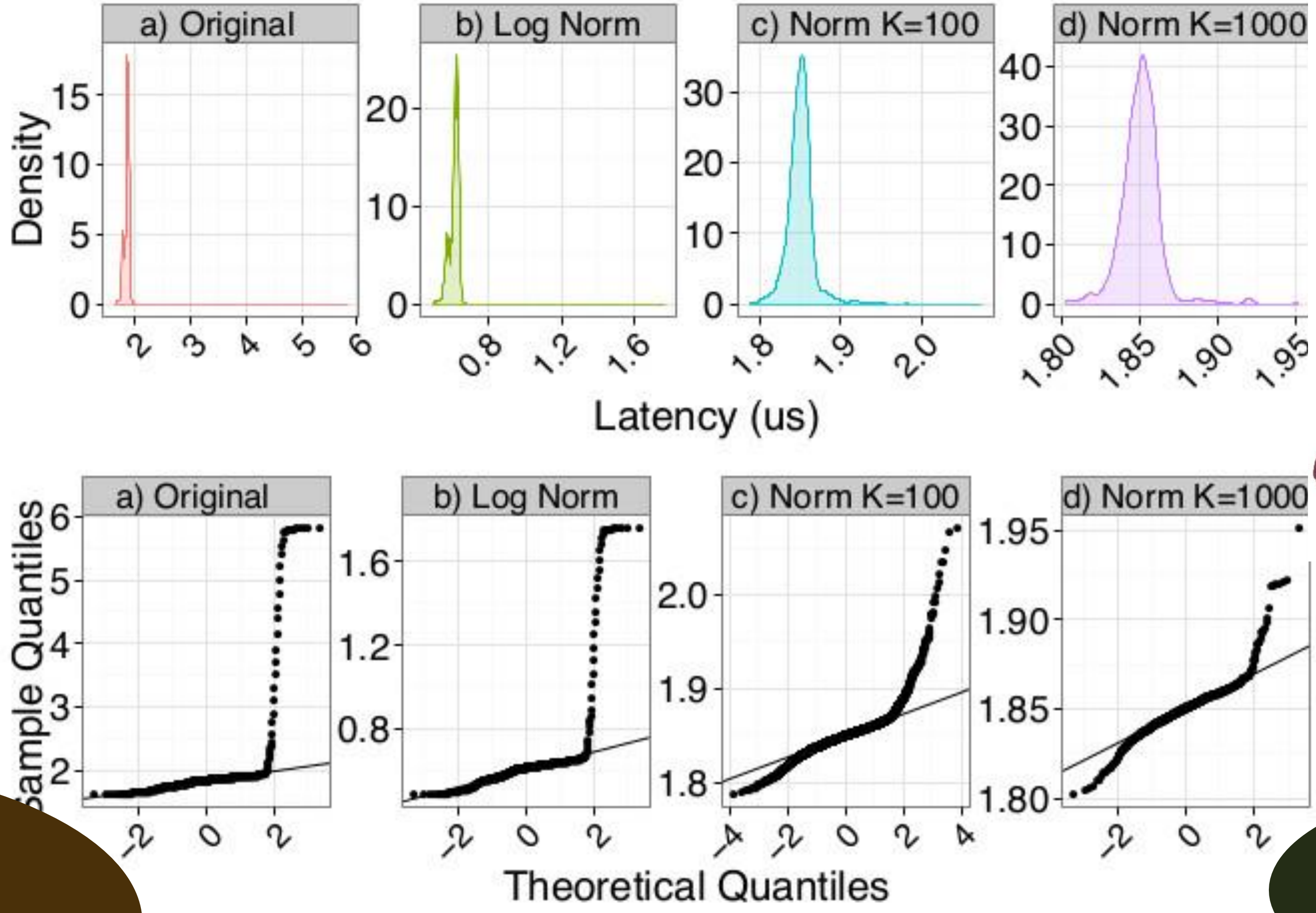
Yes, I used the central limit theorem to normalize by summing subsets of size 100!

Can we test for normality?

Thou shalt not trust your average textbook!

The confidence interval is 1.765us to 1.775us

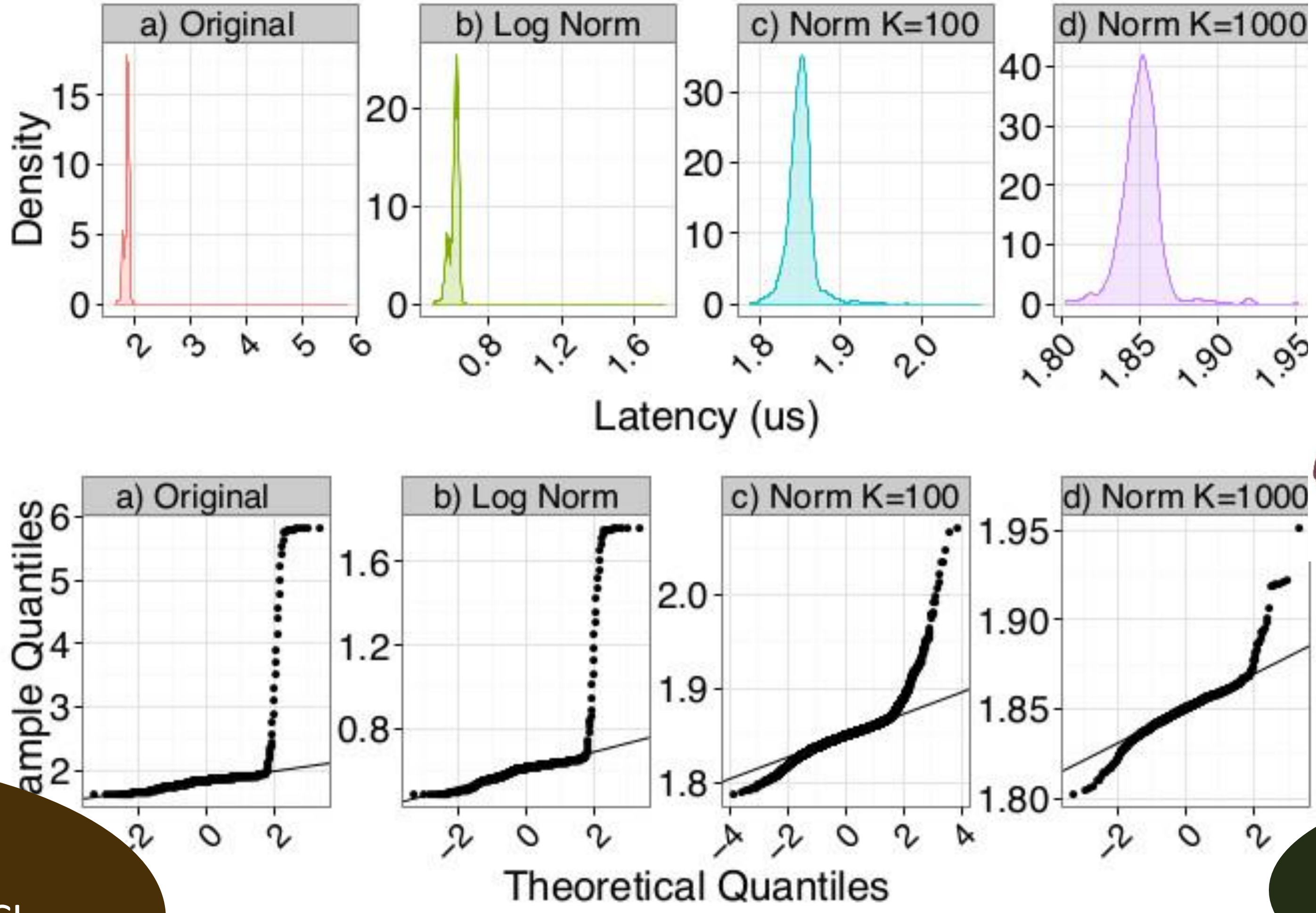
Yes, I used the central limit theorem to normalize by summing subsets of size 100!



Can we test for normality?

Thou shalt not trust your average textbook!

The confidence interval is 1.765us to 1.775us



Ugs, the data is not normal at all! The real CI is actually 1.6us to 1.9us!

Can we test for normality?

TH, Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15

Thou shalt not trust your average textbook!

The confidence interval is 1.765us to 1.775us

Rule 6: *Do not assume normality of collected data (e.g., based on the number of samples) without diagnostic checking.*

- **Most events will slow down performance**
  - **Heavy right-tailed distributions**
- **The Central Limit Theorem only applies asymptotically**
  - **Some papers/textbook mention “30-40 samples”, don’t trust them!**

Ugs, the data is not normal at all! The real CI is actually 1.6us to 1.9us!

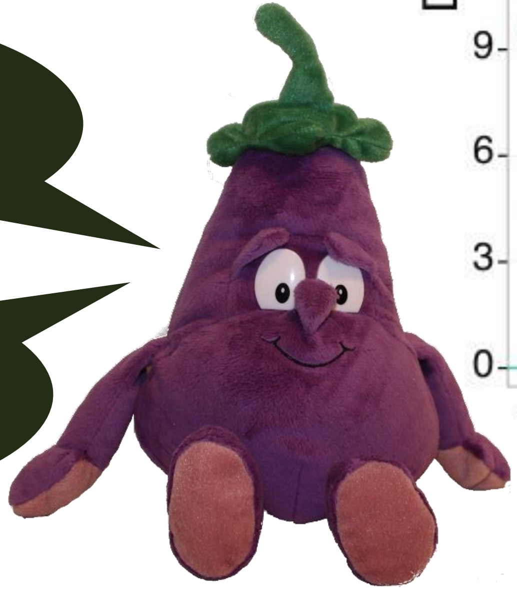
Can we test for normality?

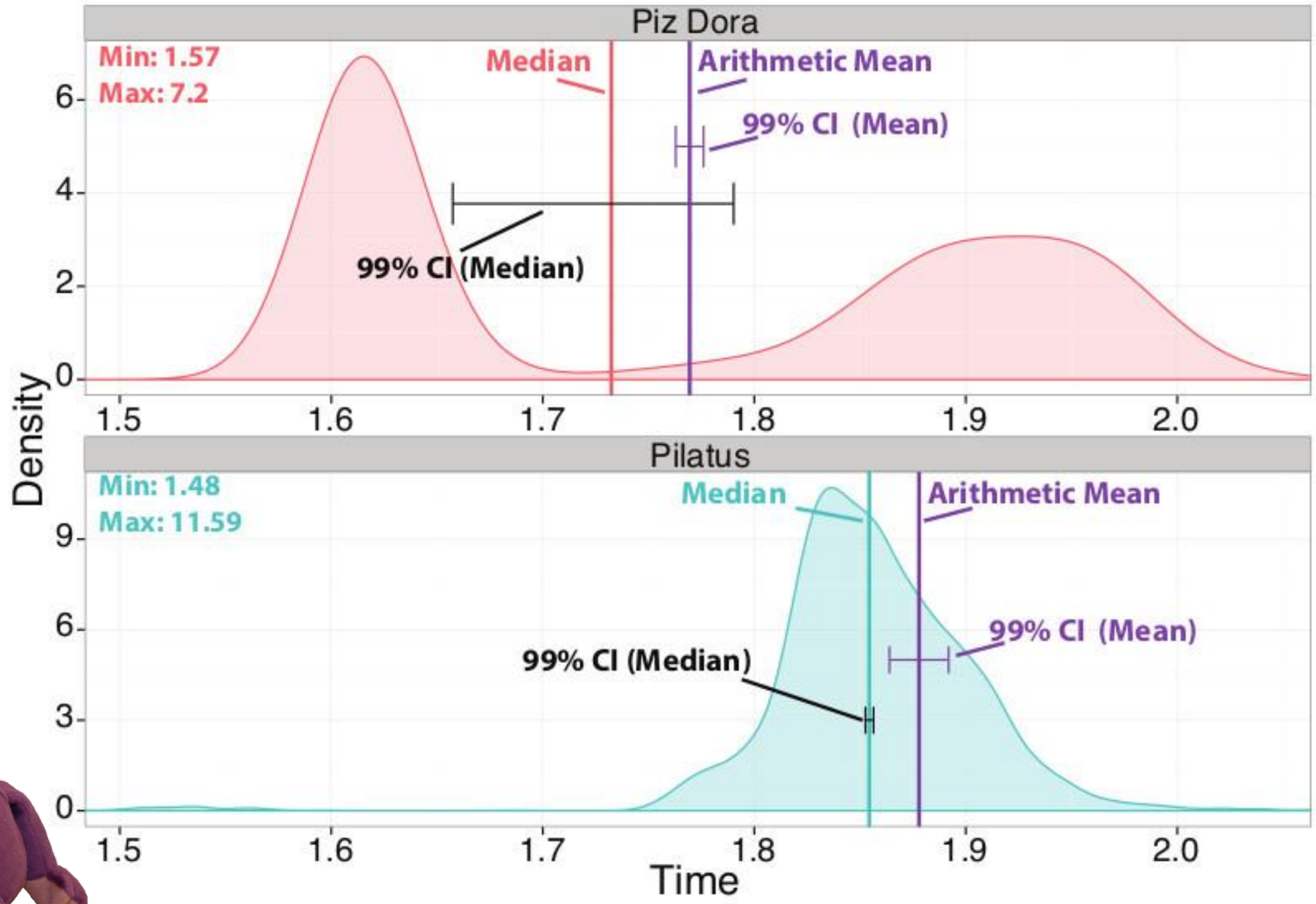
Thou shalt not trust your system!

Look what data I got!

Clearly, the mean/median are not sufficient!

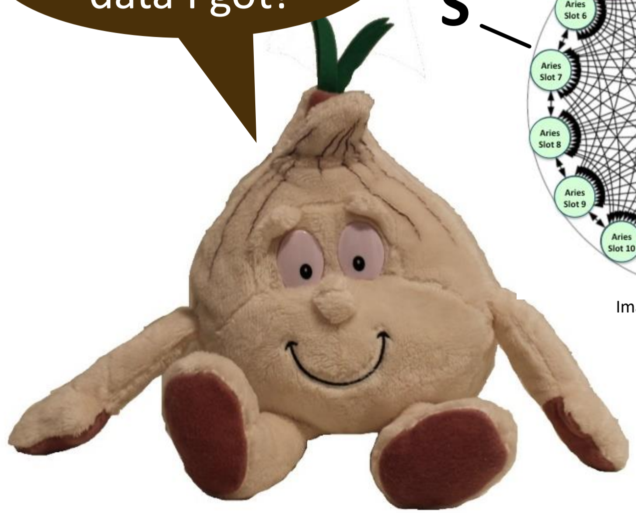
Try quantile regression!





Thou shalt not trust your system!

Look what data I got!



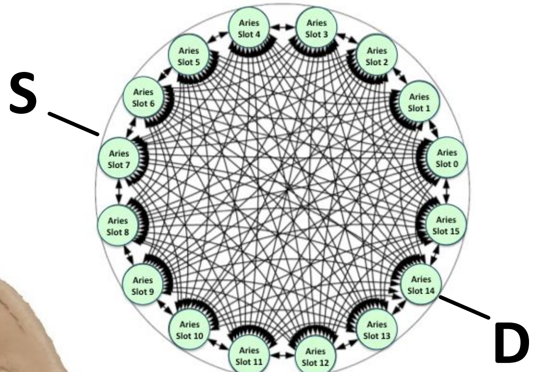
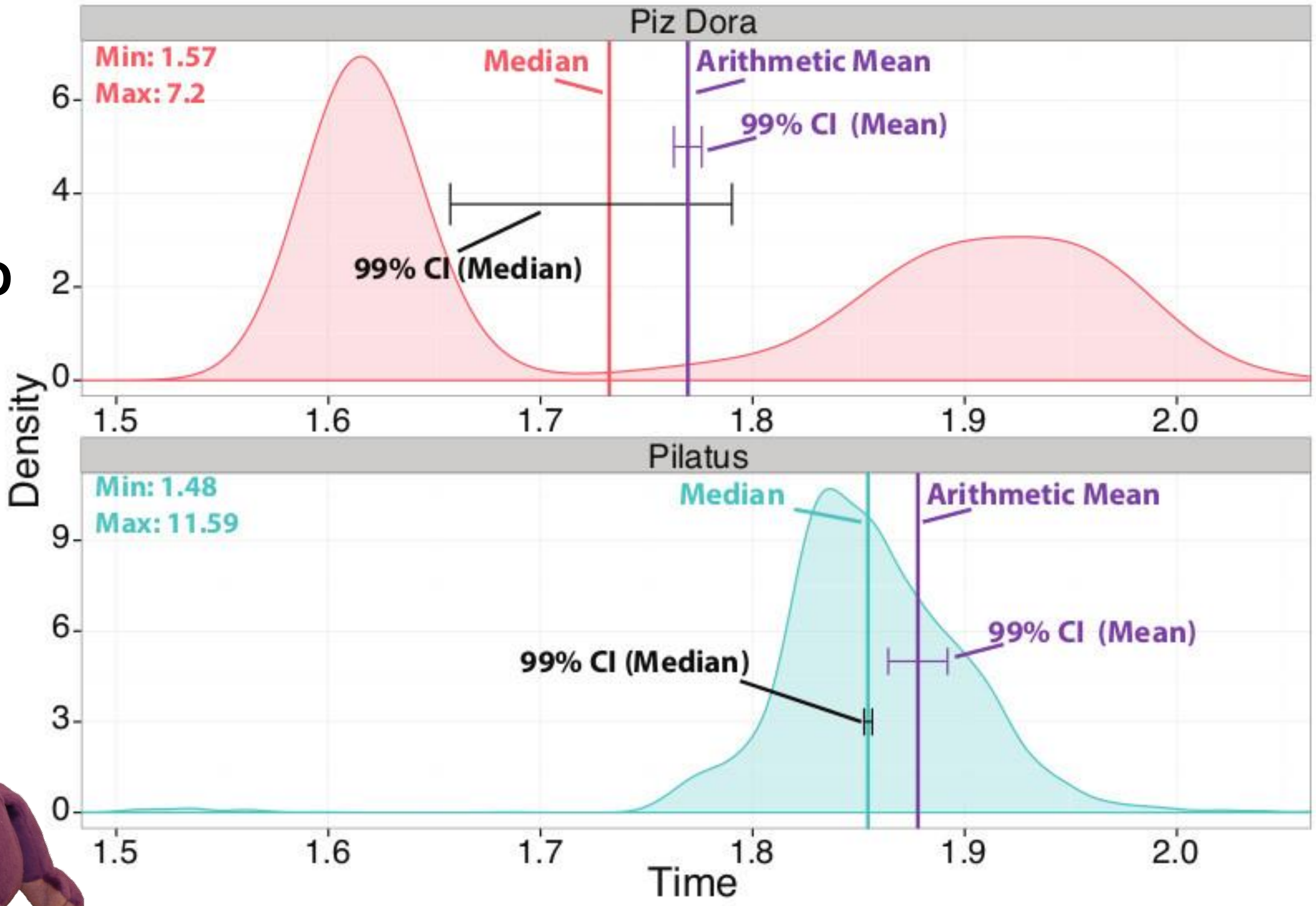
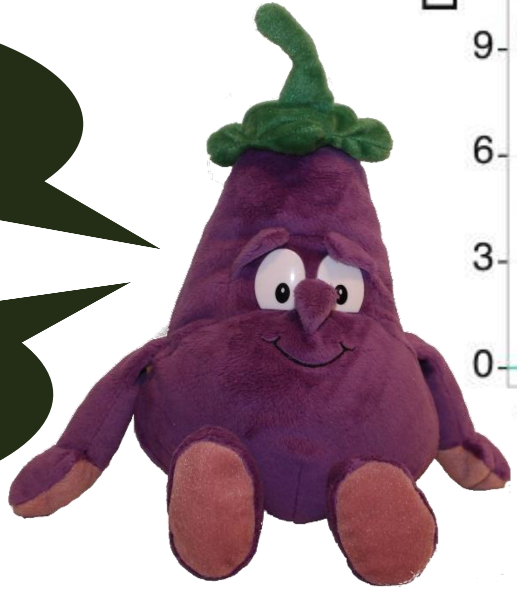


Image credit: nersc.gov



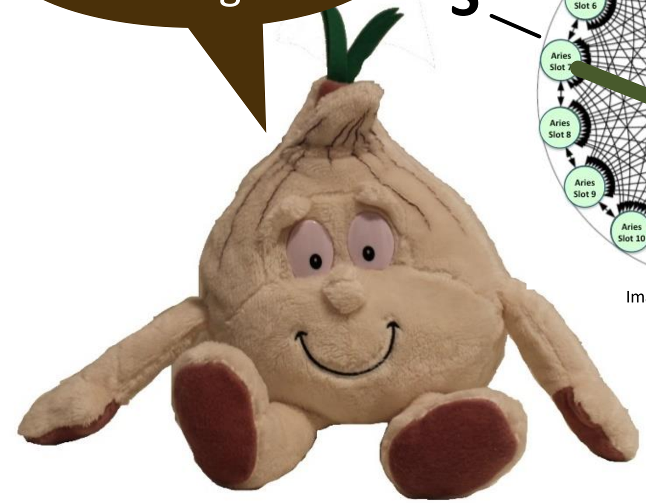
Clearly, the mean/median are not sufficient!

Try quantile regression!



Thou shalt not trust your system!

Look what data I got!



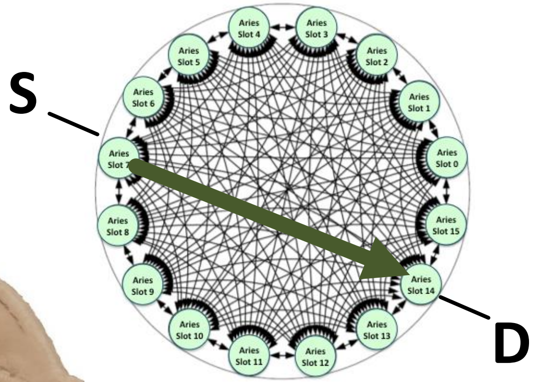
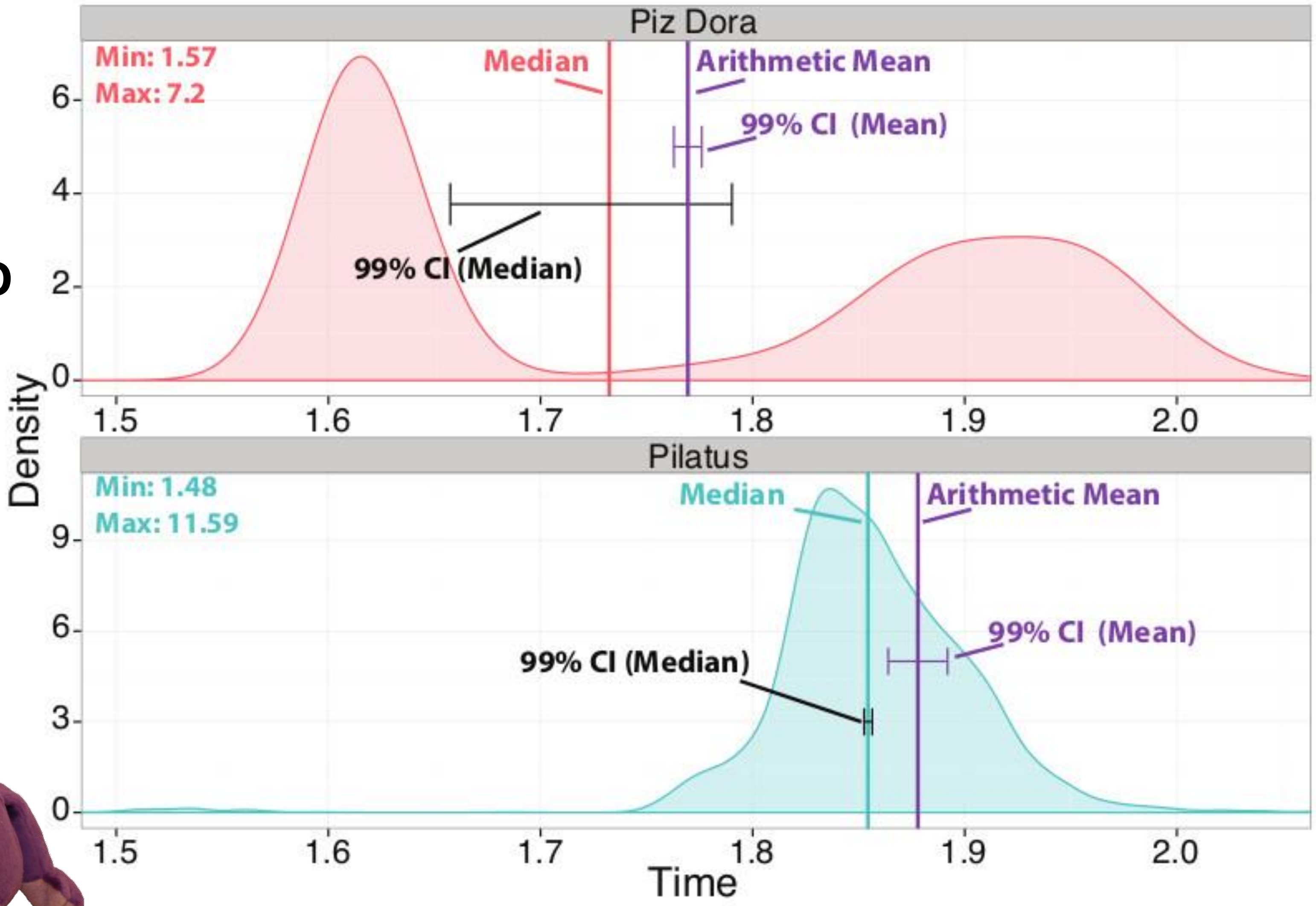
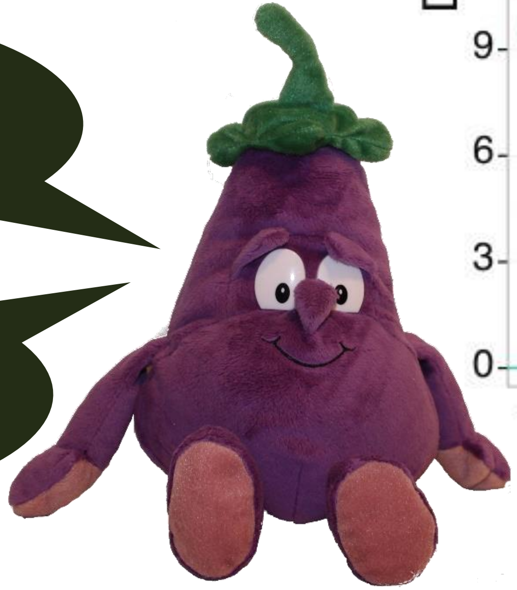


Image credit: nersc.gov



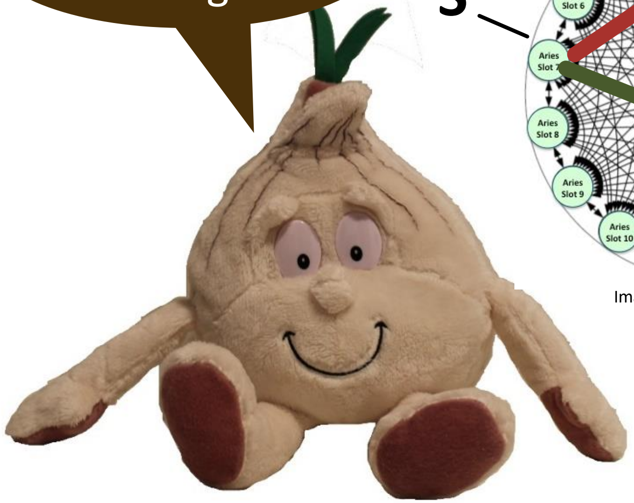
Clearly, the mean/median are not sufficient!

Try quantile regression!



Thou shalt not trust your system!

Look what data I got!



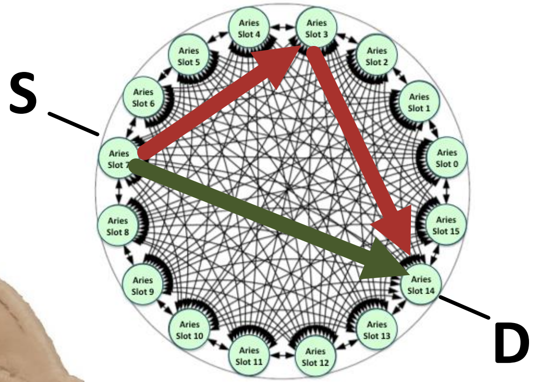
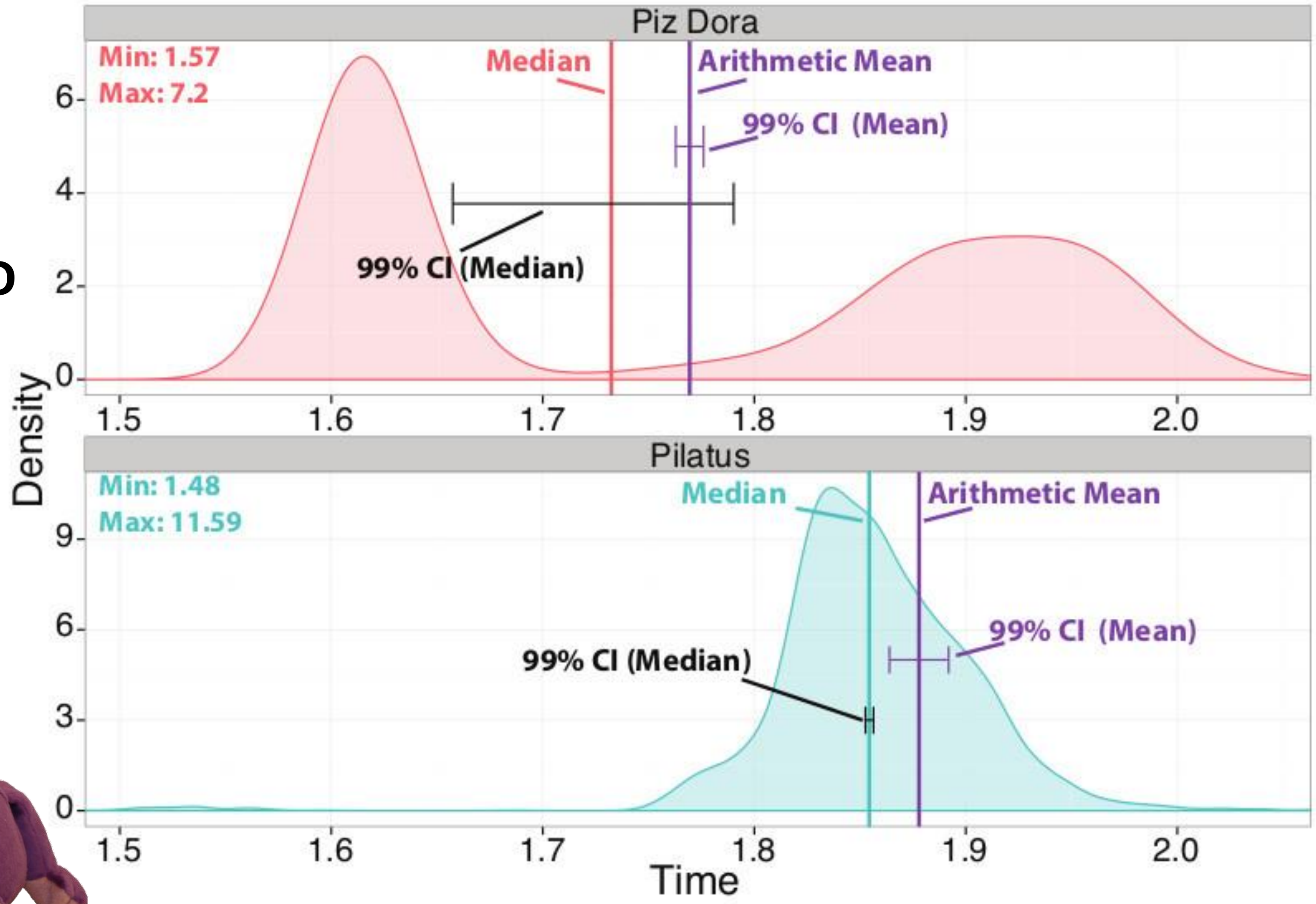
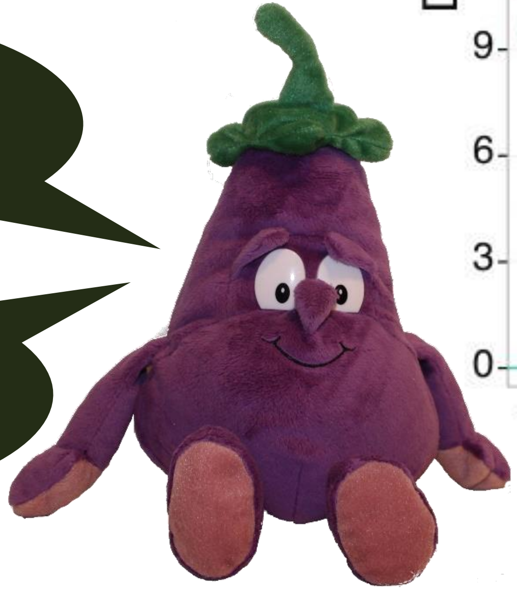


Image credit: nersc.gov



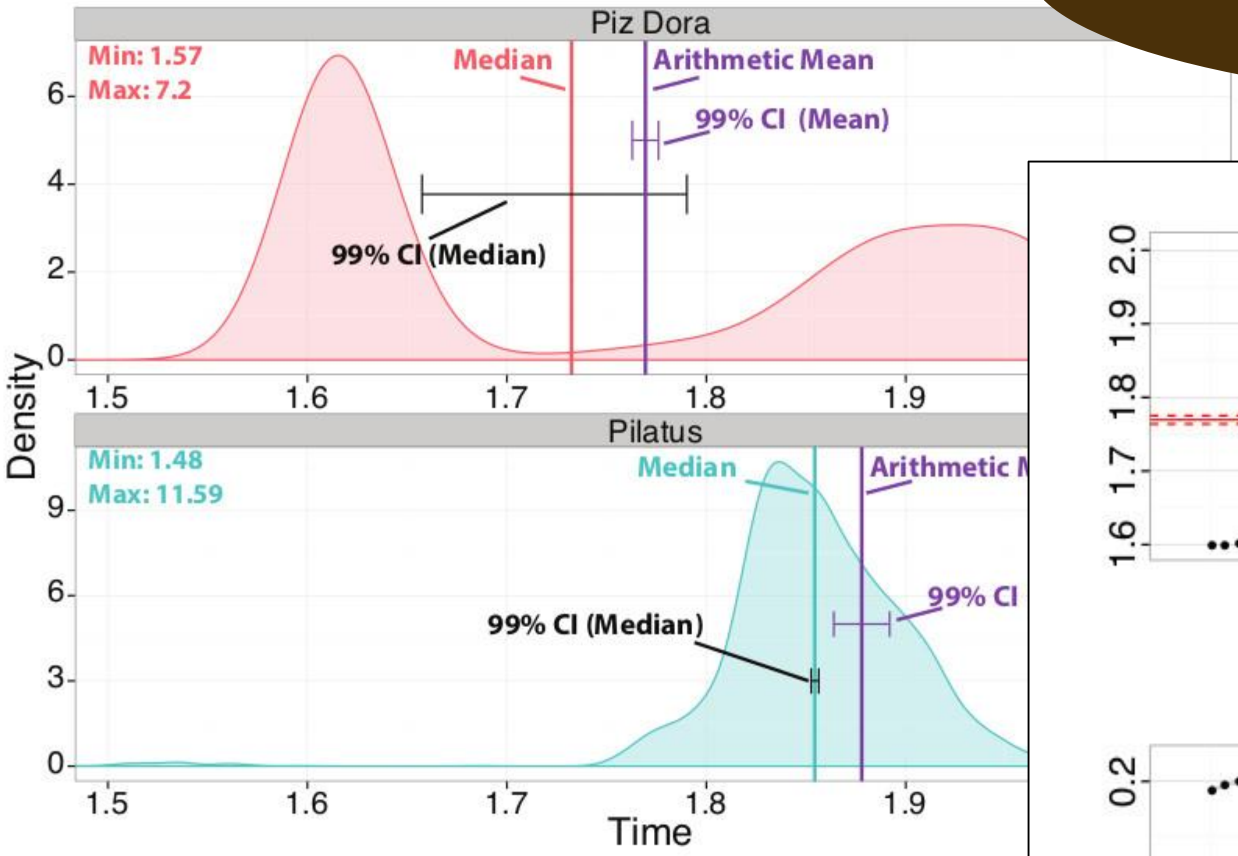
Clearly, the mean/median are not sufficient!

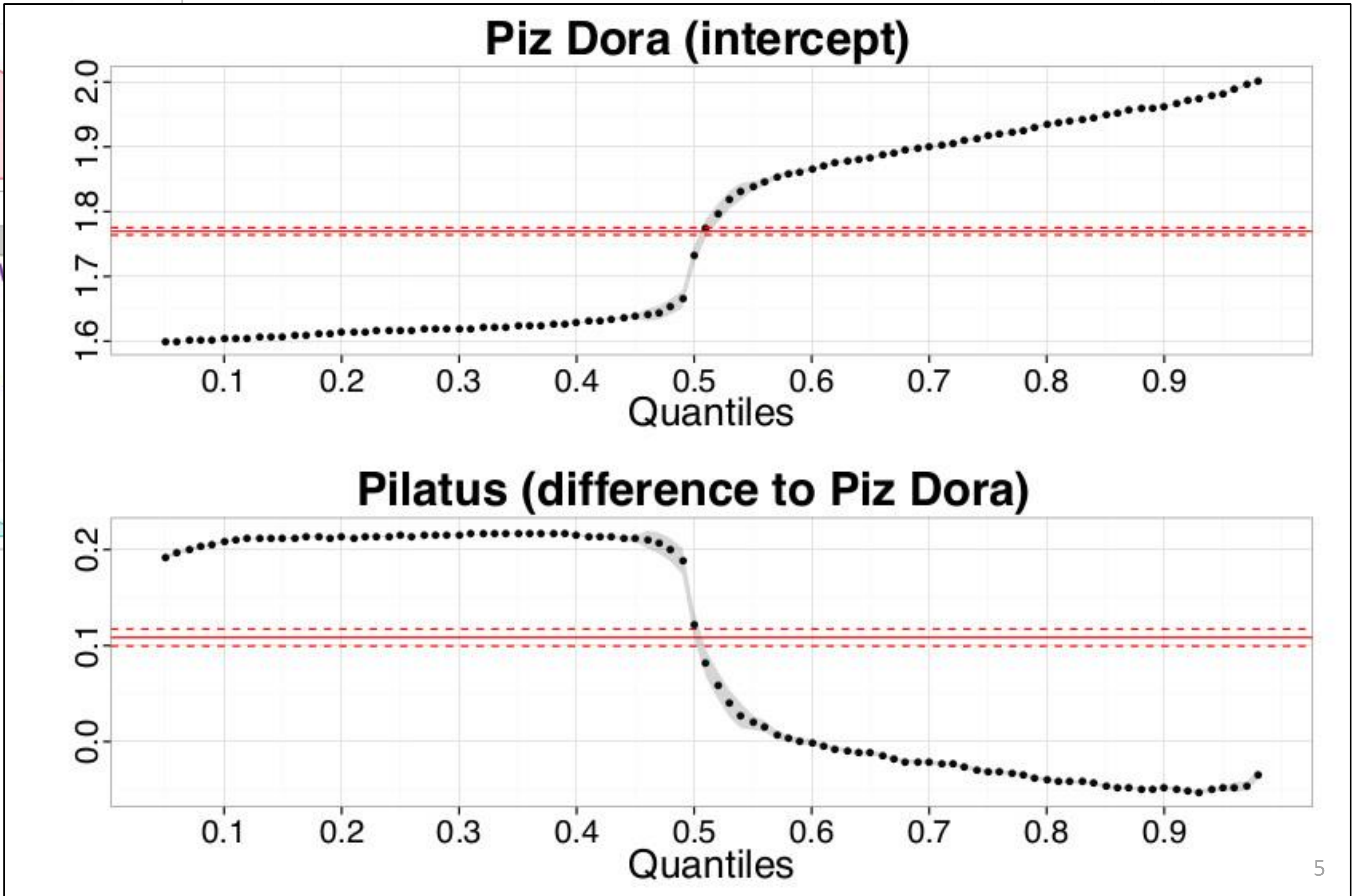
Try quantile regression!



Quantile Regression

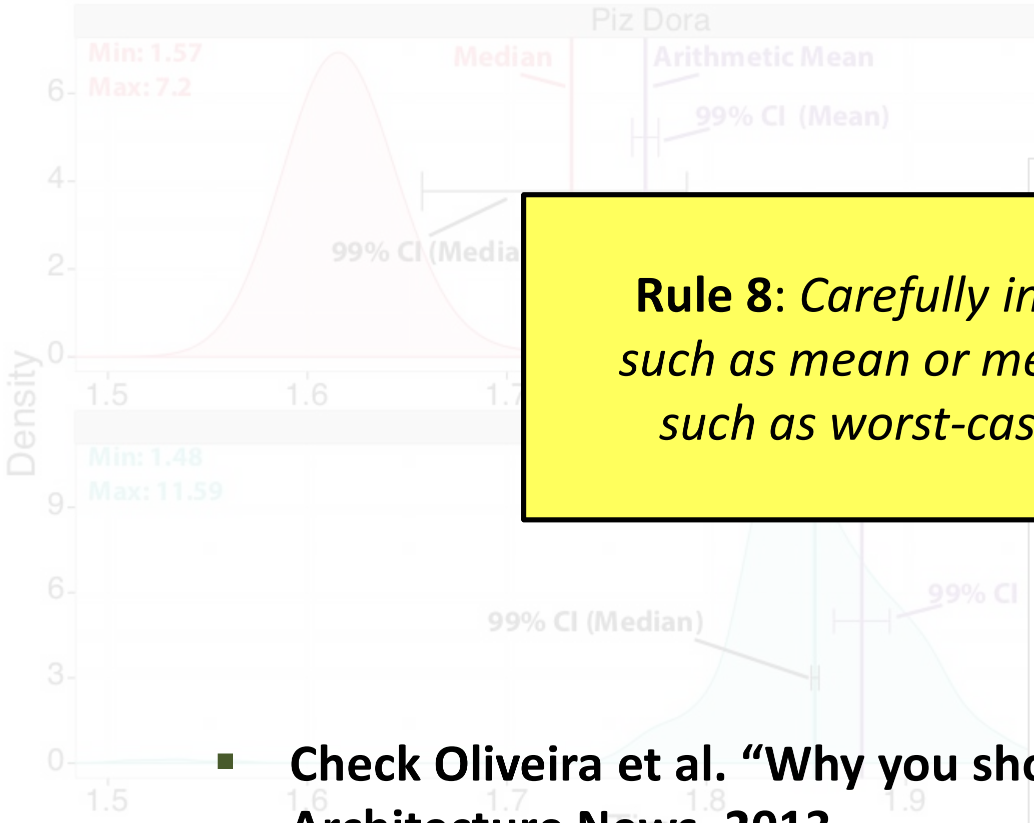
Wow, so Pilatus is better for (worst-case) latency-critical workloads even though Dora is expected to be faster

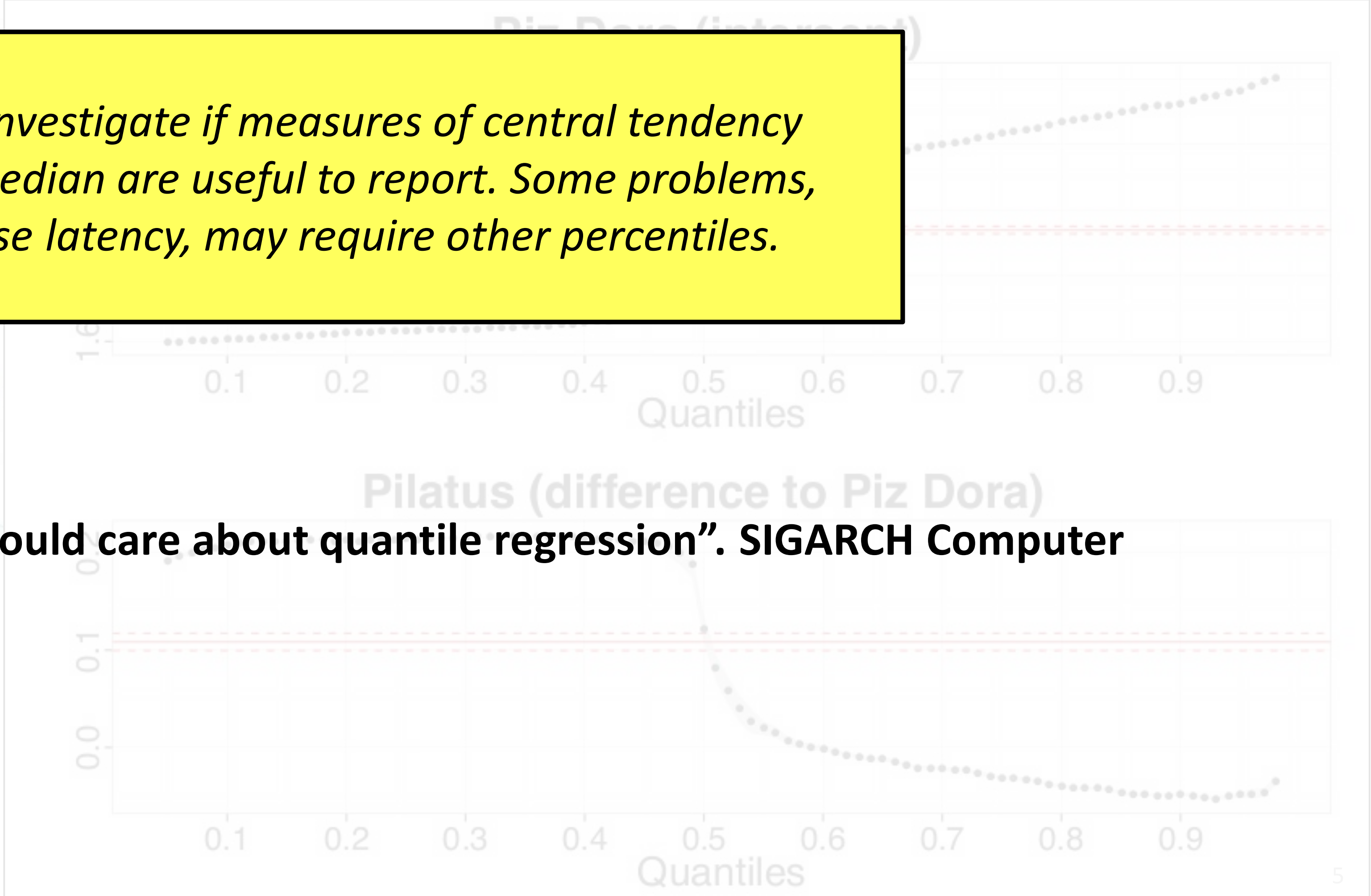




Quantile Regression

Wow, so Pilatus is better for (worst-case) latency-critical workloads even though Dora is expected to be faster





Rule 8: *Carefully investigate if measures of central tendency such as mean or median are useful to report. Some problems, such as worst-case latency, may require other percentiles.*

- **Check Oliveira et al. "Why you should care about quantile regression". SIGARCH Computer Architecture News, 2013.**

Administrivia

- **Final project presentation: last Monday 12/17 during lecture**
  - Report will be due in January!

    Starting to write early is very helpful --- write – rewrite – rewrite (no joke!)

  - Coordinate your talk! You have 10 minutes (8 talk + 2 Q&A)

    What happened since the intermediate report?

    Focus on the key aspects (time is tight)!

    Try to wrap up – only minor things left for final report.

    Engage the audience ☺

  - Send slides by Sunday night (11:59pm Zurich time) to Salvatore!

    We will use a single (windows) laptop to avoid delays when switching

    Expect only Windows (powerpoint) or a PDF viewer

    The order of talks will again be randomized for fairness

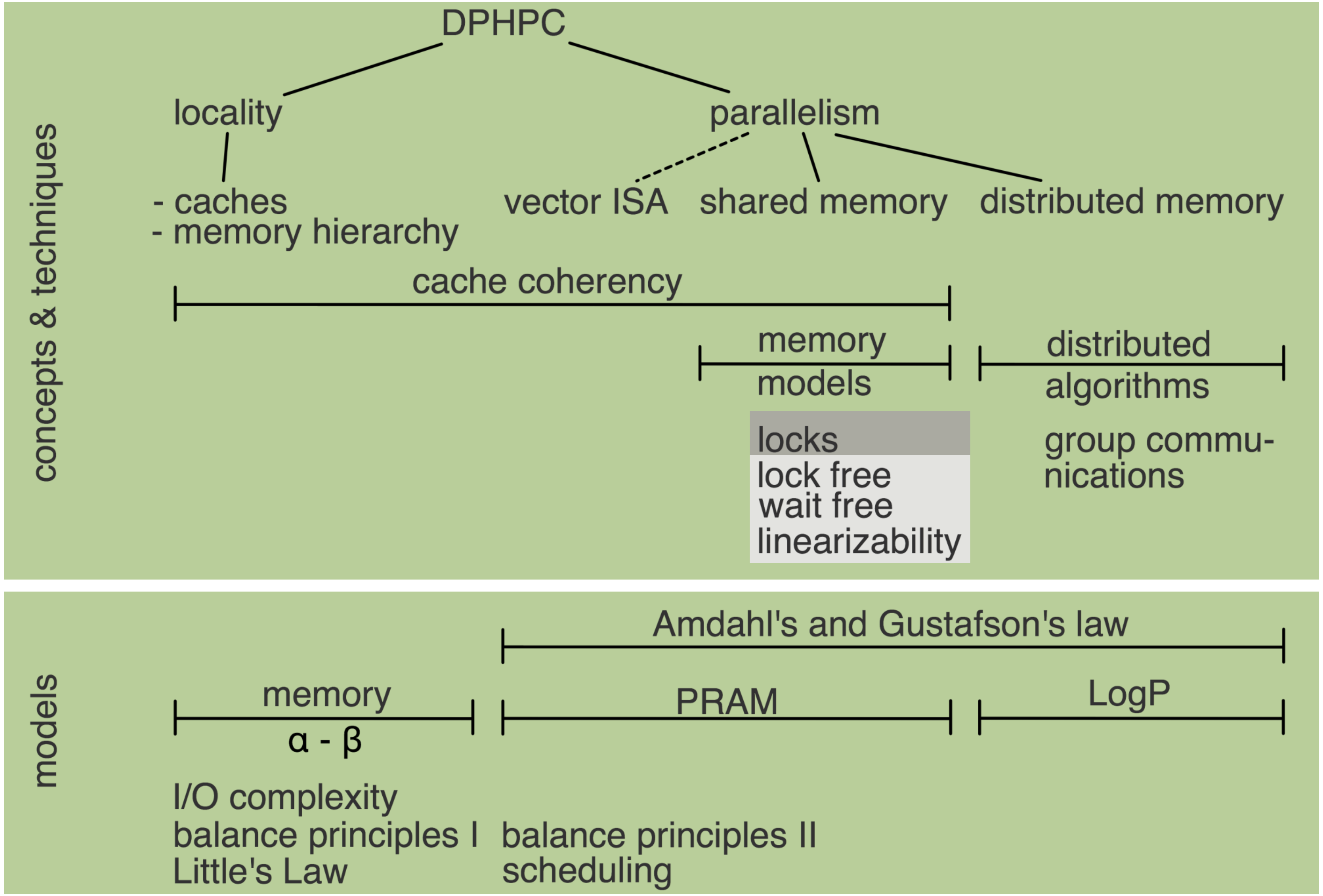
Review of last lecture(s)

- **Lock implementation(s)**
  - Advanced locks (CLH + MCS)
- **Started impossibility of wait-free consensus with atomic registers**
  - "perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)
    Will continue/finish proof today as starter!
- **Theoretical background for performance**
  - Amdahl's law
  - Models: PRAM, Work/Depth, simple alpha-beta (Hockney) model
  - Simple algorithms: reduce, scan, mergesort,
  - Brent's scheduling lemma + Little's law
  - Greedy scheduling + random work stealing
- **Practical performance**
  - Roofline and balance modeling for practical performance optimization
  - Vectorization

Learning goals for today

- **Quickly recap consensus and first part of valence proof**
  - impossibility of atomic registers for wait-free consensus
  - Complete proof together
- **Case study about scalable locking**
  - Complete correctness section!
- **Oblivious algorithms**
  - How do work-depth graphs relate to practice?
- **Strict optimality**
  - Work/depth tradeoffs and bounds
- **Applications of prefix sums**
  - Parallelize seemingly sequential algorithms

DPHPC Overview



Remember: lock-free vs. wait-free

- A locked method

- A lock-free method

- A wait-free method

Remember: lock-free vs. wait-free

- **A locked method**
  - May deadlock (methods may never finish)
- **A lock-free method**
- **A wait-free method**

Remember: lock-free vs. wait-free

- **A locked method**
  - May deadlock (methods may never finish)

- **A lock-free method**
  - Guarantees that infinitely often **some** method call finishes in a finite number of steps

- **A wait-free method**

Remember: lock-free vs. wait-free

- **A locked method**
  - May deadlock (methods may never finish)

- **A lock-free method**
  - Guarantees that infinitely often **some** method call finishes in a finite number of steps

- **A wait-free method**
  - Guarantees that **each** method call finishes in a finite number of steps (implies lock-free)

Remember: lock-free vs. wait-free

- **A locked method**
  - May deadlock (methods may never finish)
- **A lock-free method**
  - Guarantees that infinitely often **some** method call finishes in a finite number of steps
- **A wait-free method**
  - Guarantees that **each** method call finishes in a finite number of steps (implies lock-free)
- **Synchronization instructions are not equally powerful!**
  - Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level z>x.

Concept: Consensus Number

- **Each level of the hierarchy has a "consensus number" assigned.**
  - Is the maximum number of threads for which primitives in level x can solve the consensus problem

- **The consensus problem:**
  - Has single function: decide(v)
  - Each thread calls it at most once, the function returns a value that meets two conditions:

    consistency: all threads get the same value

    validity: the value is some thread's input
  - Simplification: binary consensus (inputs in {0,1})

Understanding Consensus

- **Can a particular class solve n-thread consensus wait-free?**
  - A class C solves n-thread consensus if there exists a consensus protocol using **any number** of objects of class C and **any number** of atomic registers
  - The protocol has to be wait-free (bounded number of steps per thread)
  - The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite)
  - Assume we have a class D whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?

Starting simple …

- **Binary consensus with two threads (A, B)!**
  - Each thread moves until it decides on a value
  - May update shared objects
  - Protocol state = state of threads + state of shared objects
  - Initial state = state before any thread moved
  - Final state = state after all threads finished
  - States form a tree, wait-free property guarantees a finite tree

    Example with two threads and two moves each!

- **Define various states**
  - Bivalent, univalent, critical

- **Two helper lemmata**
  - Lemma 1: the initial state is bivalent
  - Lemma 2: every wait-free consensus protocol has a critical state

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
  - I.e., they cannot be used to solve even two-thread consensus! Really?

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
  - I.e., they cannot be used to solve even two-thread consensus! Really?
- **Proof outline:**
  - Assume arbitrary consensus protocol, thread A, B
  - Run until it reaches critical state where next action determines outcome (show that it must have a critical state first)
  - Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions!
    1) *Any thread reads (other thread runs solo until end)*
    2) *Threads write to different registers (order doesn't matter)*
    3) *Threads write to same register (solo thread can start after each write)*

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
- **Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers**
  - "perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)
  - → We **need** hardware atomics or Transactional Memory!

- **Proof technique borrowed from:**

  Impossibility of distributed consensus with one ... - ACM Digital Library
  https://dl.acm.org/citation.cfm?id=214121
  by MJ Fischer - 1985 - Cited by 4669 - Related articles
  Sep 4, 2012 - Michael J. Fischer , Nancy A. Lynch , Michael S. Paterson, **Impossibility of distributed consensus** with one faulty process, Proceedings of the ...

- **Very influential paper, always worth a read!**
  - Nicely shows proof techniques that are central to parallel and distributed computing!

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
  - Similar proof technique (bivalence argument)

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
  - Similar proof technique (bivalence argument)

- **CAS and TM have consensus number ∞**

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
  - Similar proof technique (bivalence argument)

- **CAS and TM have consensus number ∞**
  - Constructive proof:

```
const int first = -1
volatile int thread = -1;
int proposed[n];

int decide(v) {
  proposed[tid] = v;
  if(CAS(thread, first, tid))
    return v; // I won!
  else
    return proposed[thread]; // thread won
}
```

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
  - Similar proof technique (bivalence argument)

- **CAS and TM have consensus number ∞**
  - Constructive proof:

```
const int first = -1
volatile int thread = -1;
int proposed[n];

int decide(v) {
  proposed[tid] = v;
  if(CAS(thread, first, tid))
    return v; // I won!
  else
    return proposed[thread]; // thread won
}
```

  - Machines providing CAS are **asynchronous** computation equivalents of the Turing Machine
    I.e., any concurrent object can be implemented in a wait-free manner (not necessarily fast!)

Now you know everything about parallel program correctness ☺

- **At least a lot … ;-)**
  - We’ll argue more about **performance** now!

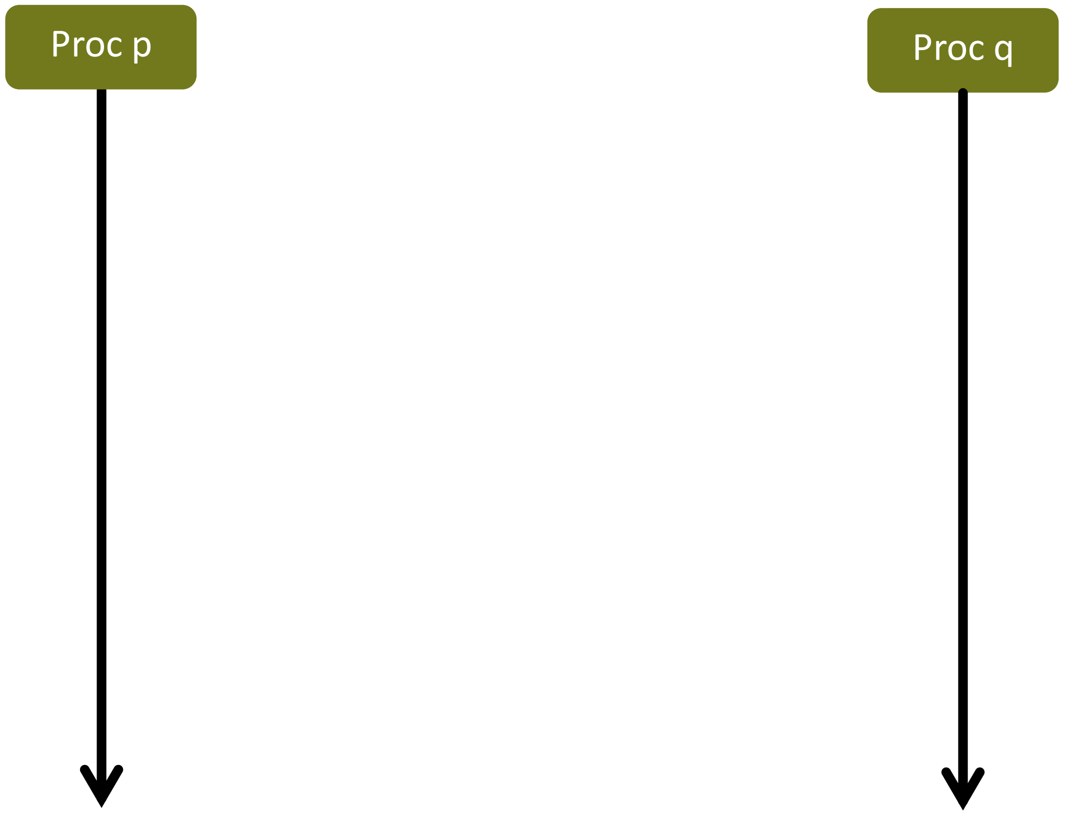
- **You have all the tools for:**
  - Efficient locks
  - Efficient lock-based algorithms
  - Reasoning about parallelism!

- **What now?**
  - Now you understand practice and will appreciate theory
    Wasn’t that all too messy ☺?
  - Focus on (parallel) performance, techniques, and algorithms

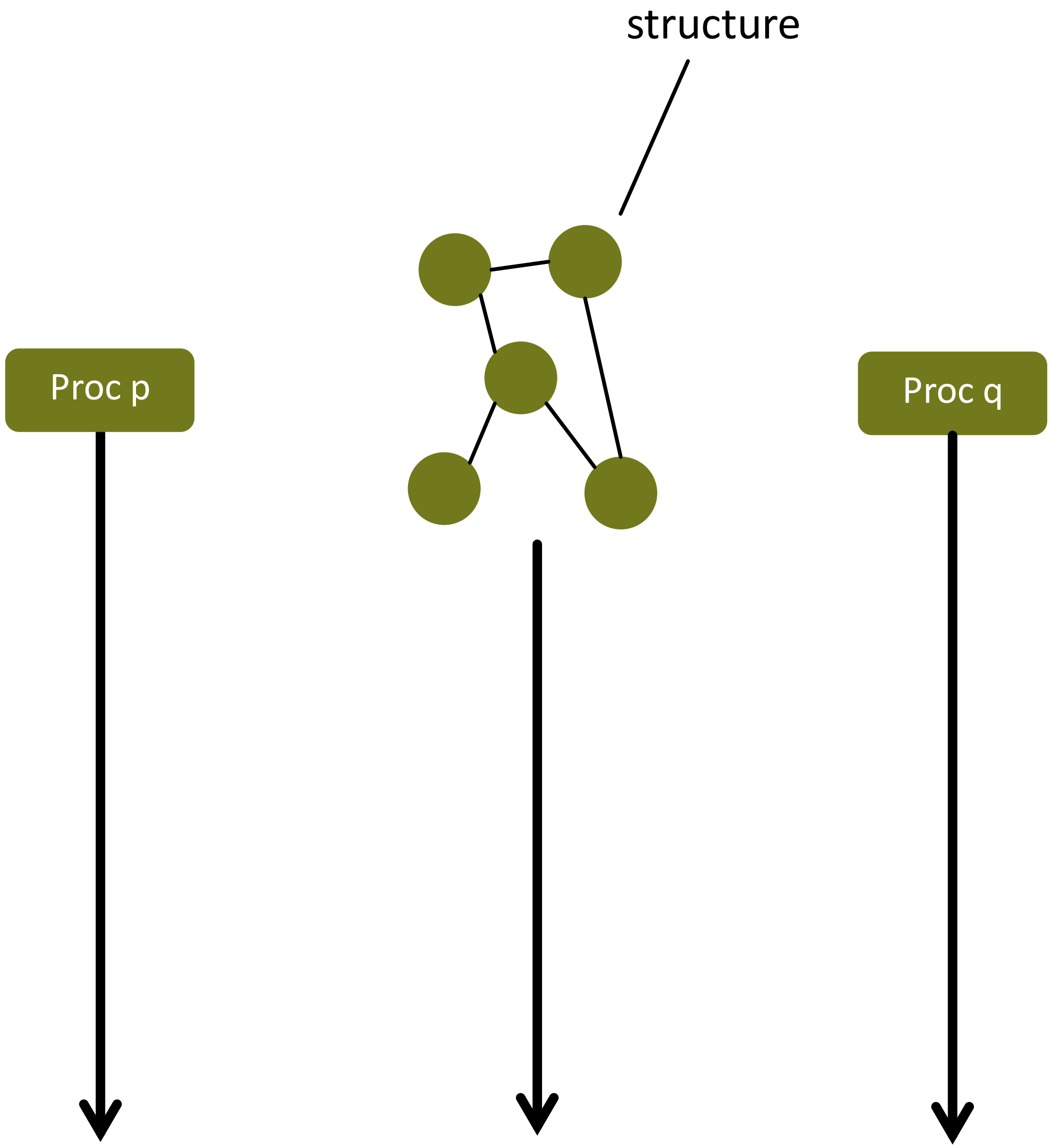
- **But let’s start with another case study about locks**
  - Research (best) paper published at a top-tier conference some years ago
    So you get a feeling of the field – and deepen understanding of MCS locks in practice

Case study: Fast Large-scale Locking in Practice

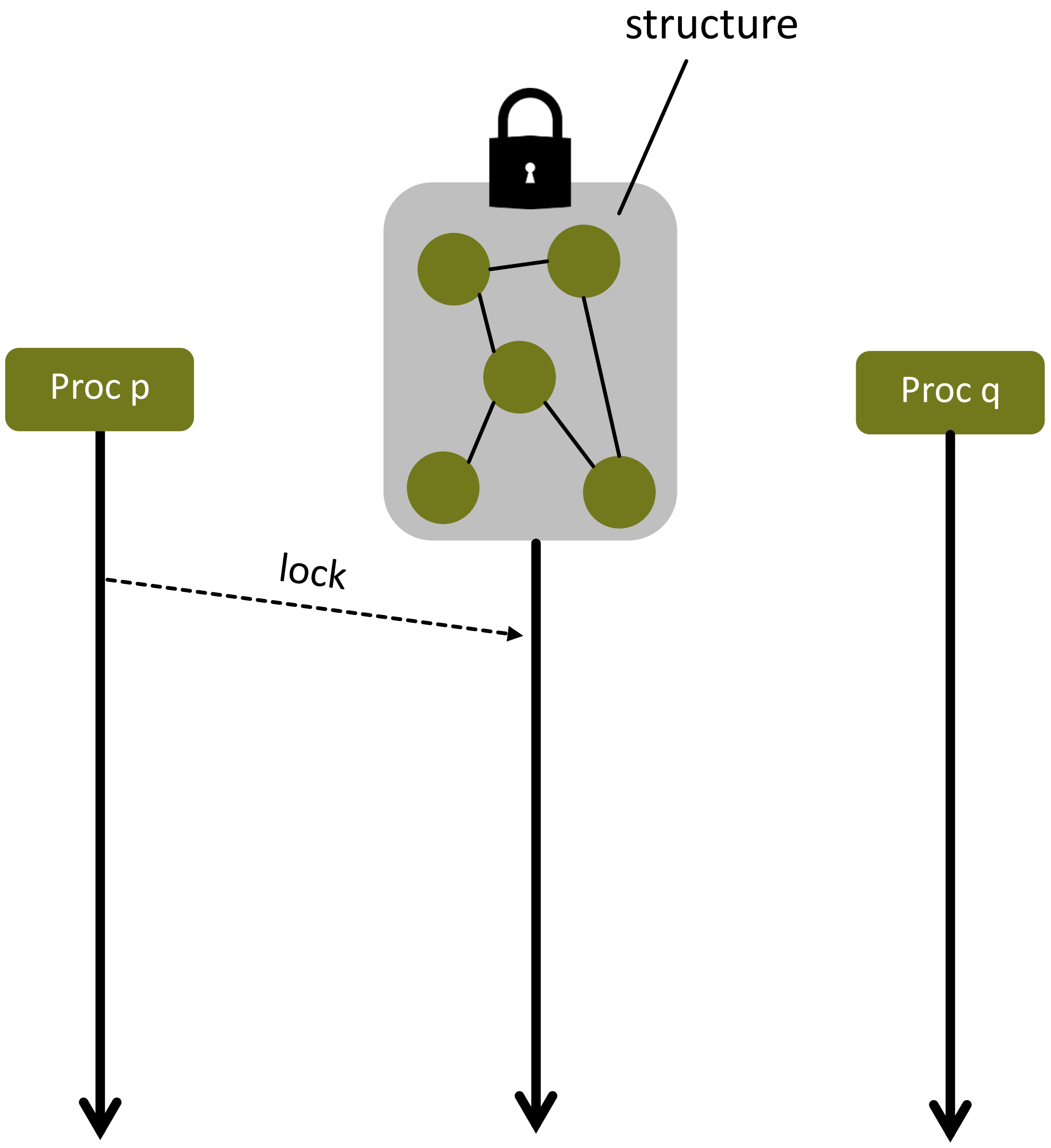
Case study: Fast Large-scale Locking in Practice



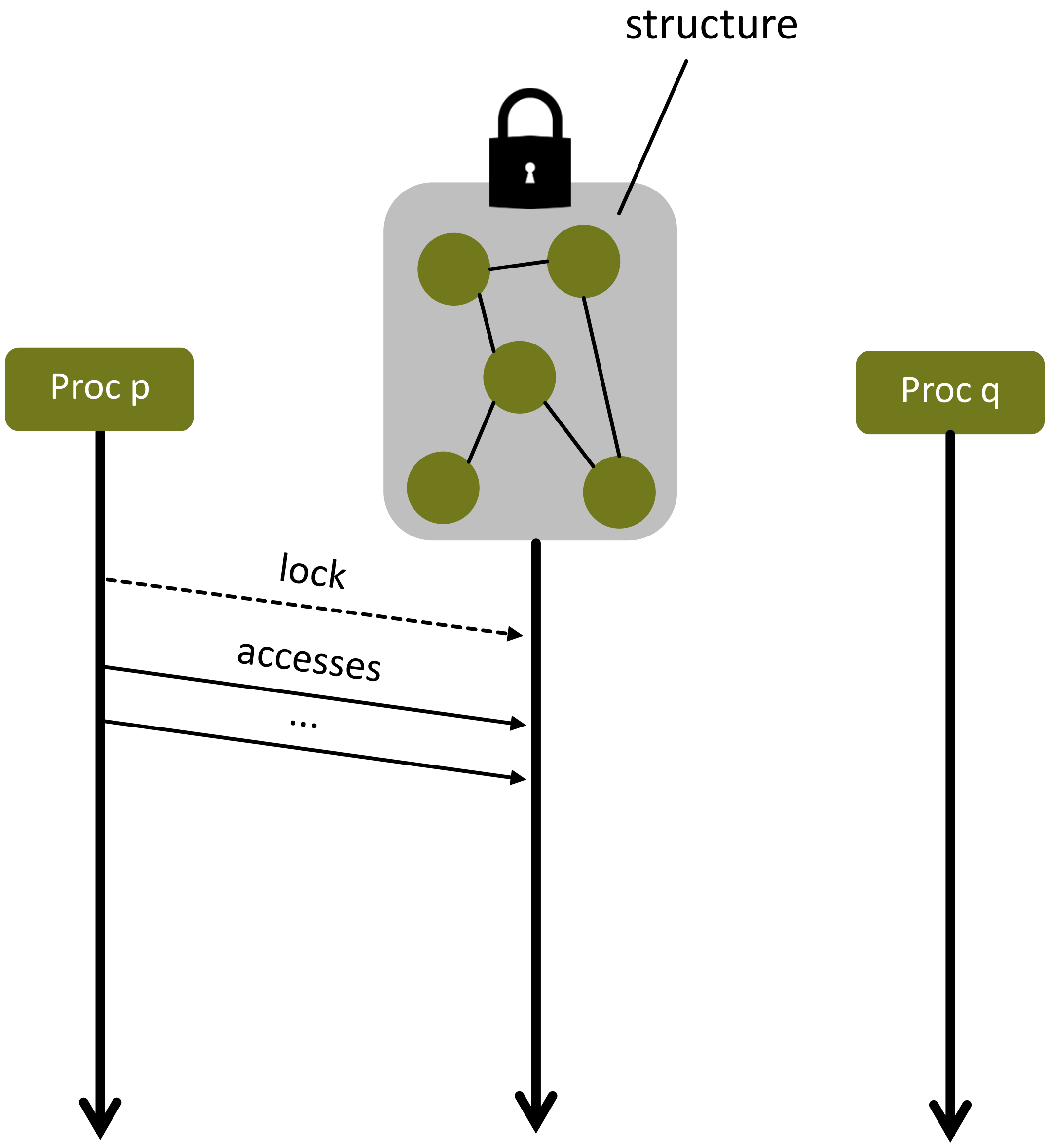
Case study: Fast Large-scale Locking in Practice



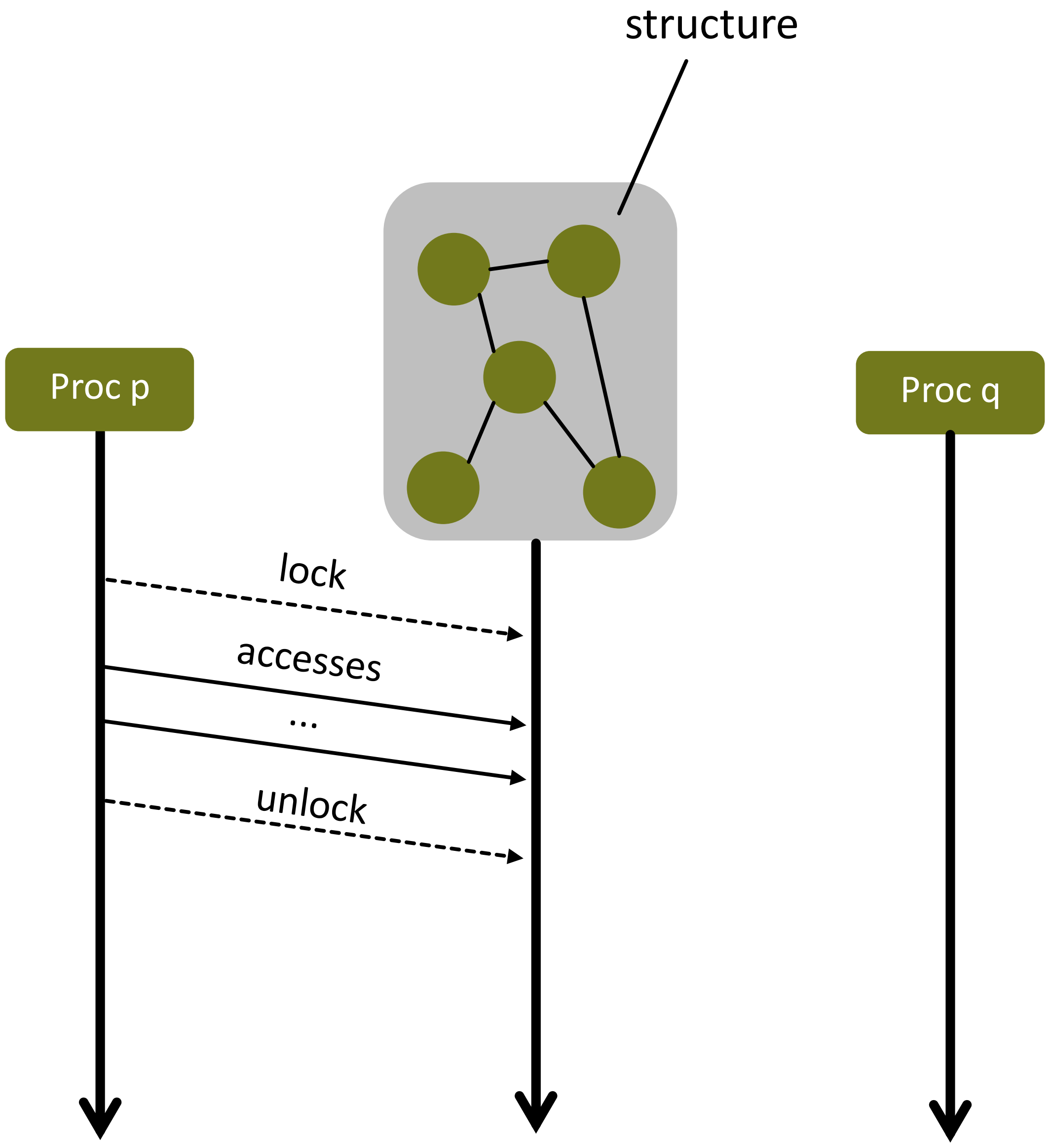
Case study: Fast Large-scale Locking in Practice



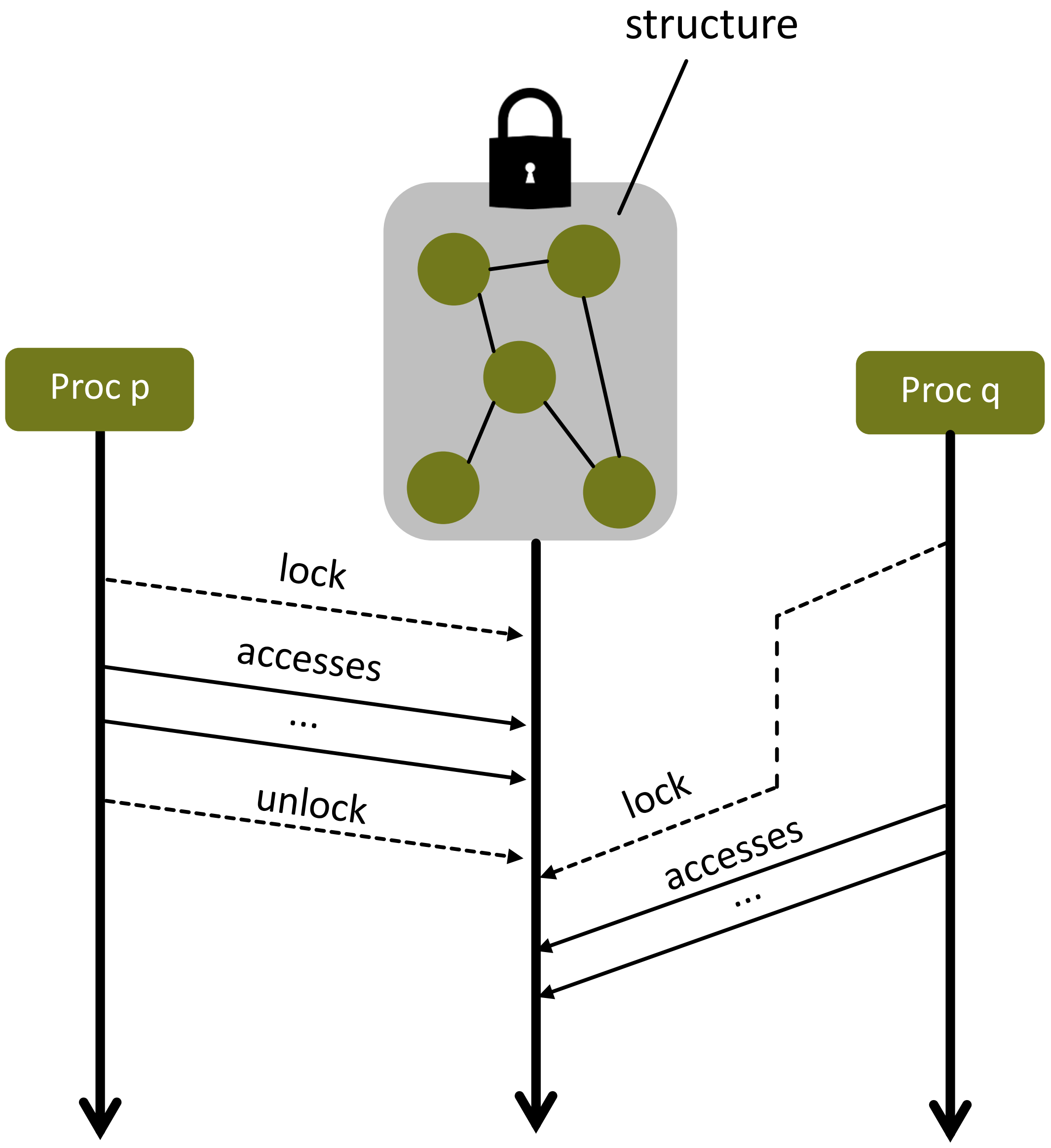
Case study: Fast Large-scale Locking in Practice



Case study: Fast Large-scale Locking in Practice

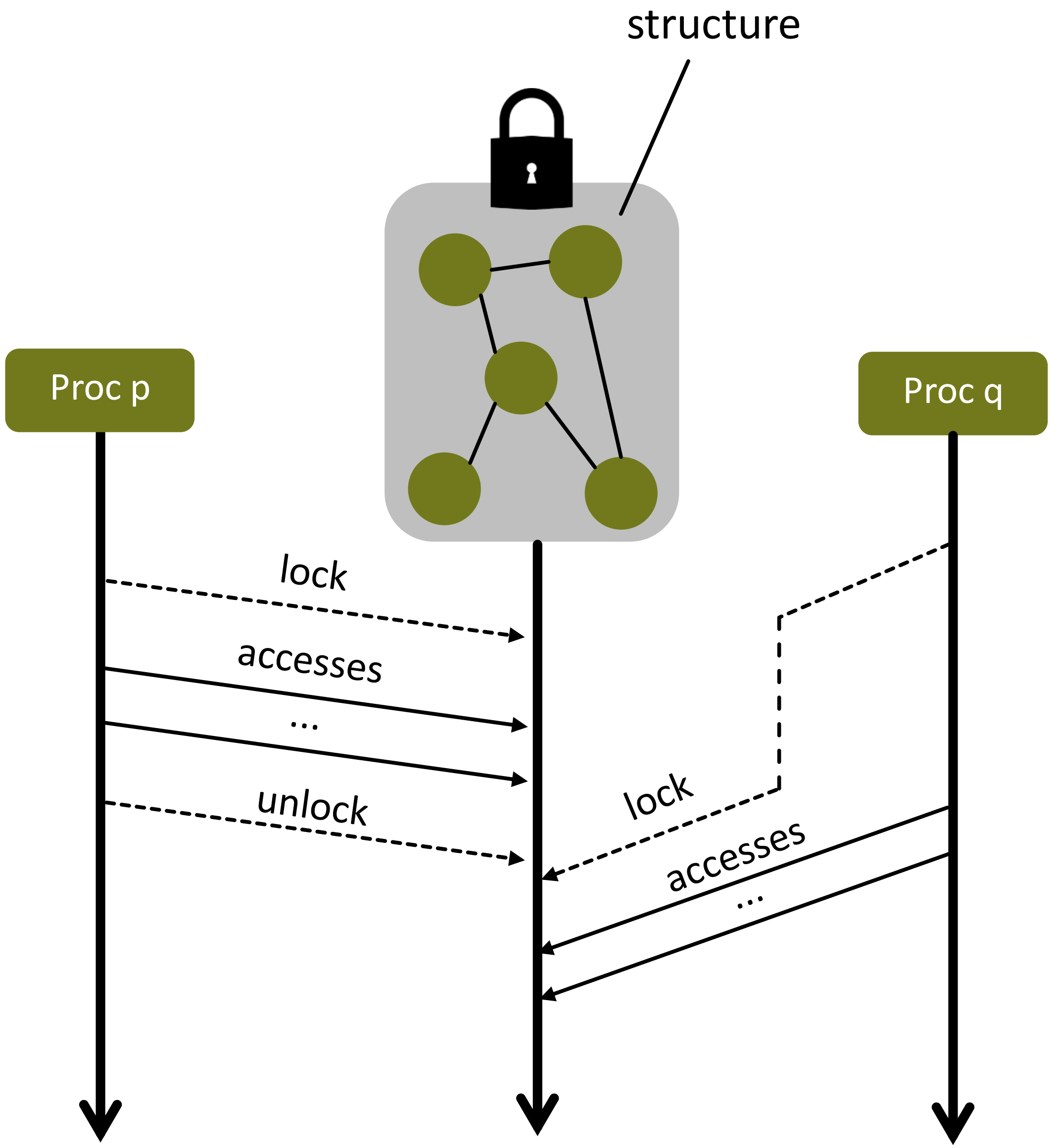


Case study: Fast Large-scale Locking in Practice



Case study: Fast Large-scale Locking in Practice

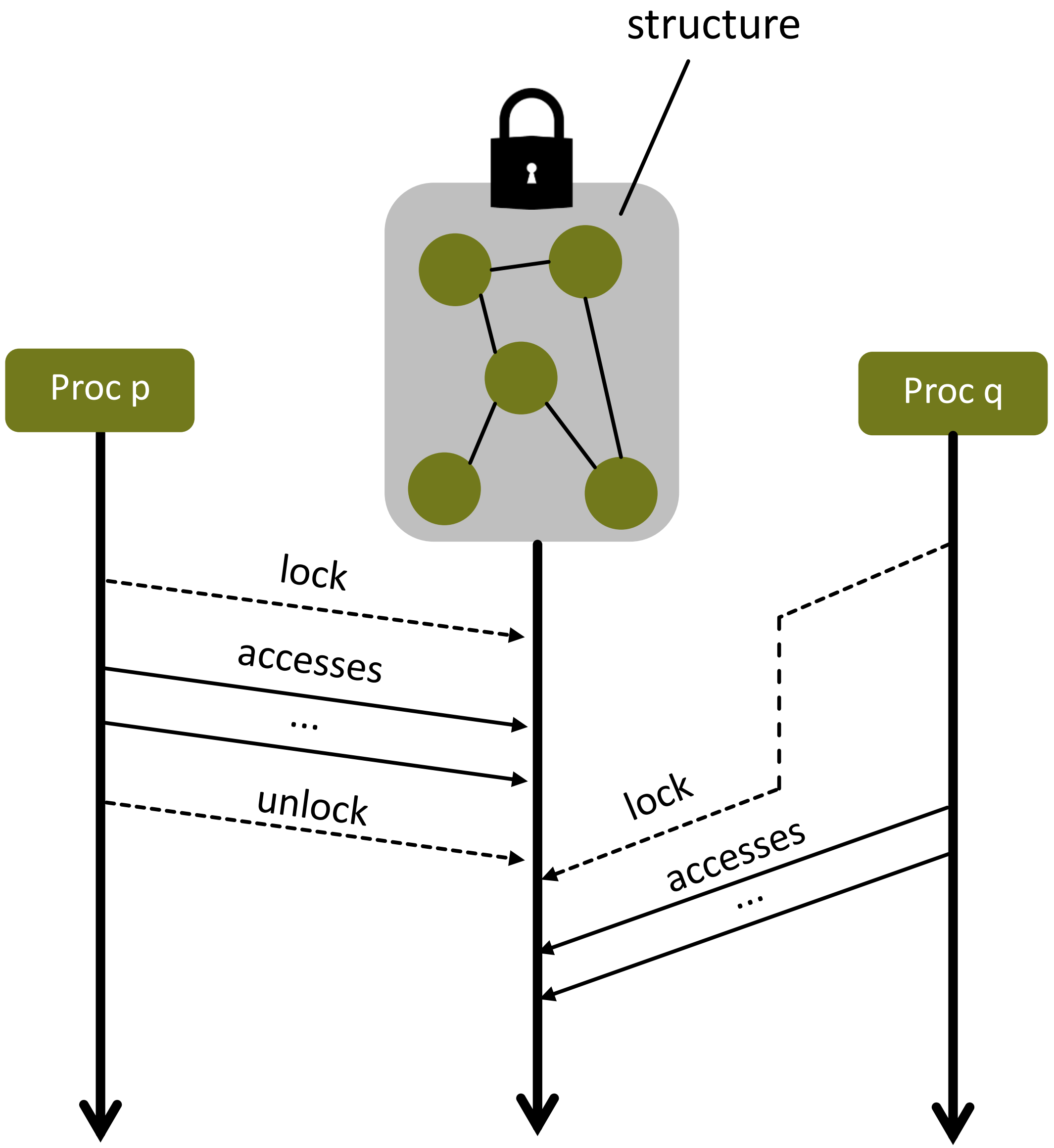
Inuitive semantics



Case study: Fast Large-scale Locking in Practice

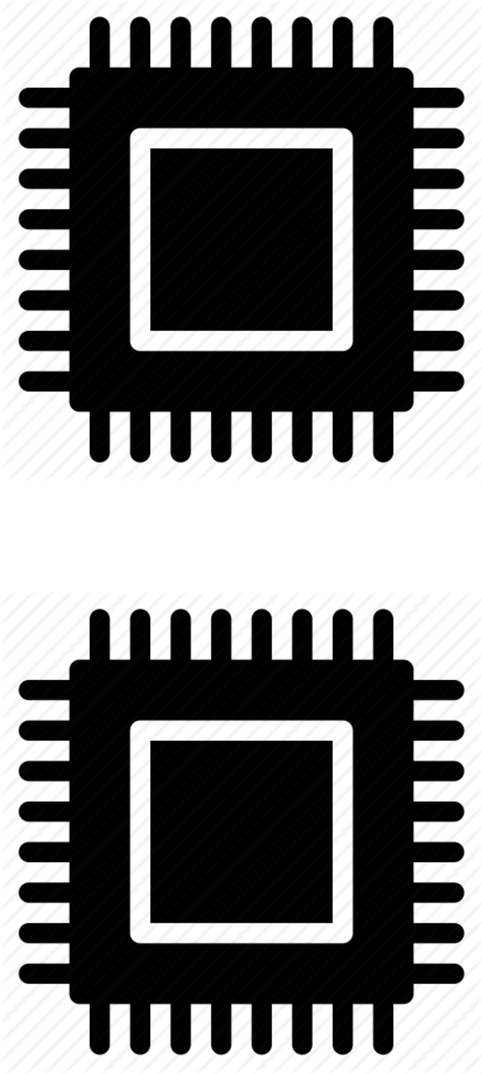
Inuitive semantics

Various performance penalties



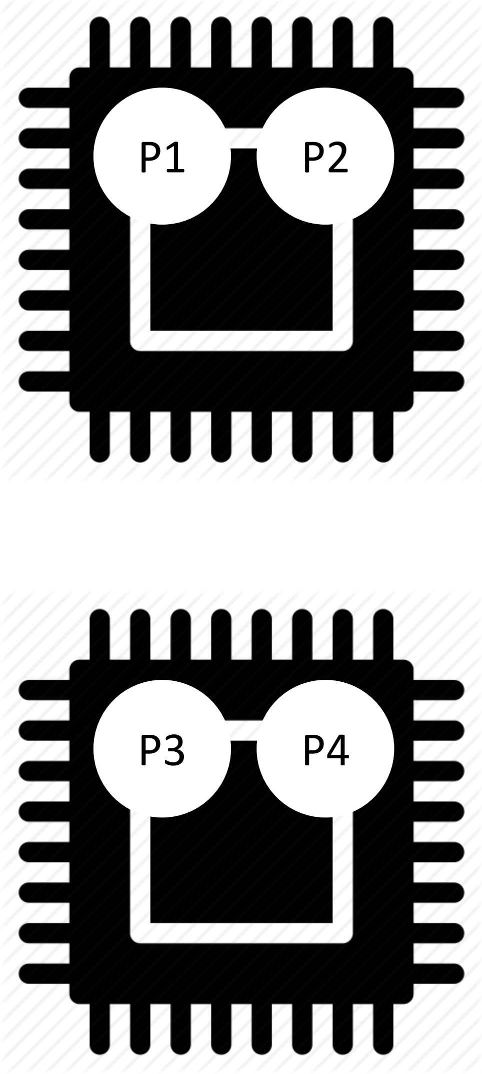
Locks: Challenges

Locks: Challenges



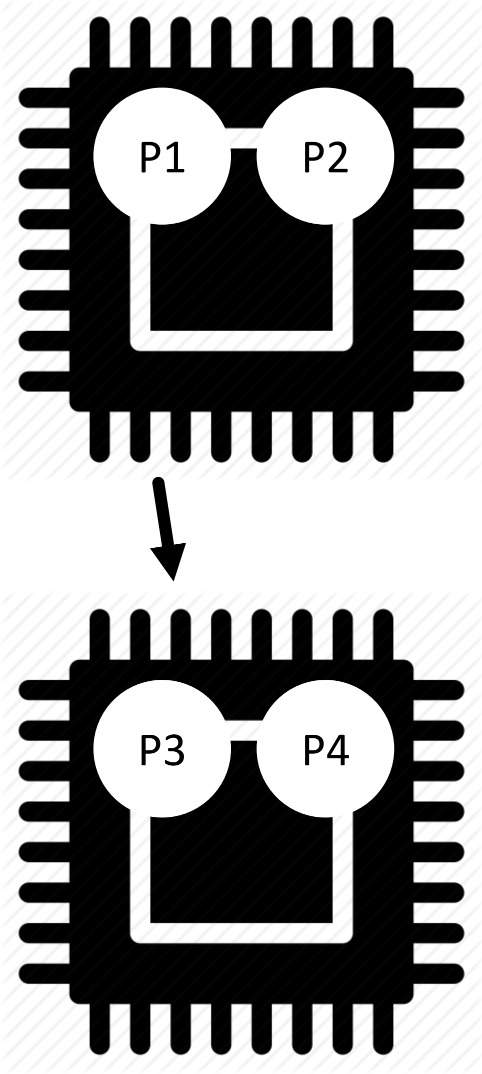
Calciu et al.: NUMA-aware reader-writer locks, PPoPP'13

Locks: Challenges



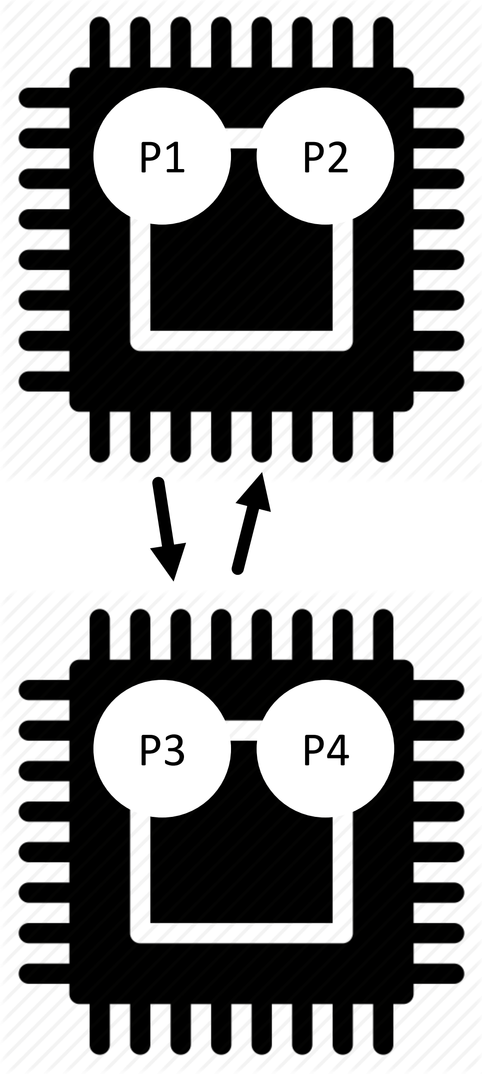
Calciu et al.: NUMA-aware reader-writer locks, PPoPP'13

Locks: Challenges



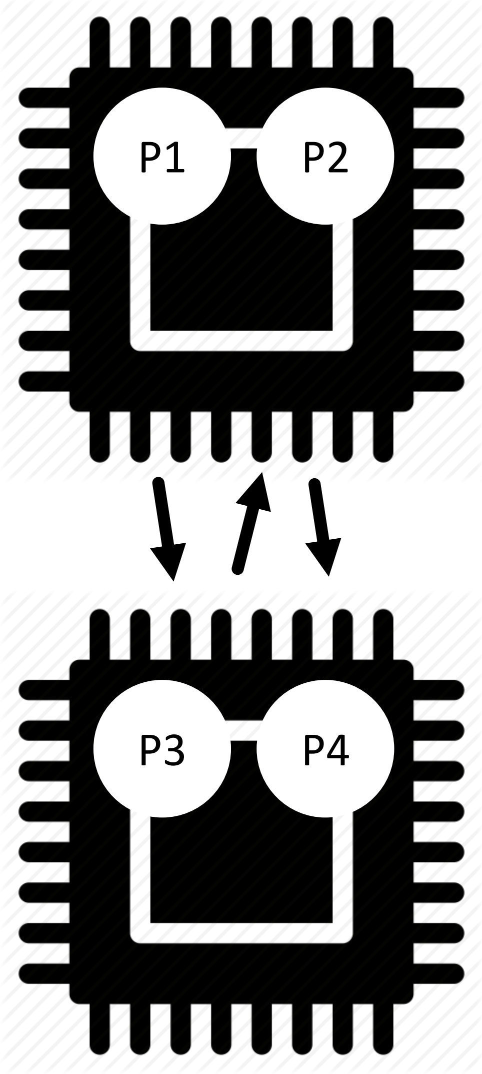
Calciu et al.: NUMA-aware reader-writer locks, PPoPP'13

Locks: Challenges



Calciu et al.: NUMA-aware reader-writer locks, PPoPP'13

Locks: Challenges



Calciu et al.: NUMA-aware reader-writer locks, PPoPP'13

Locks: Challenges

Locks: Challenges

Locks: Challenges

We need intra- and inter-node topology-awareness

We need to cover arbitrary topologies

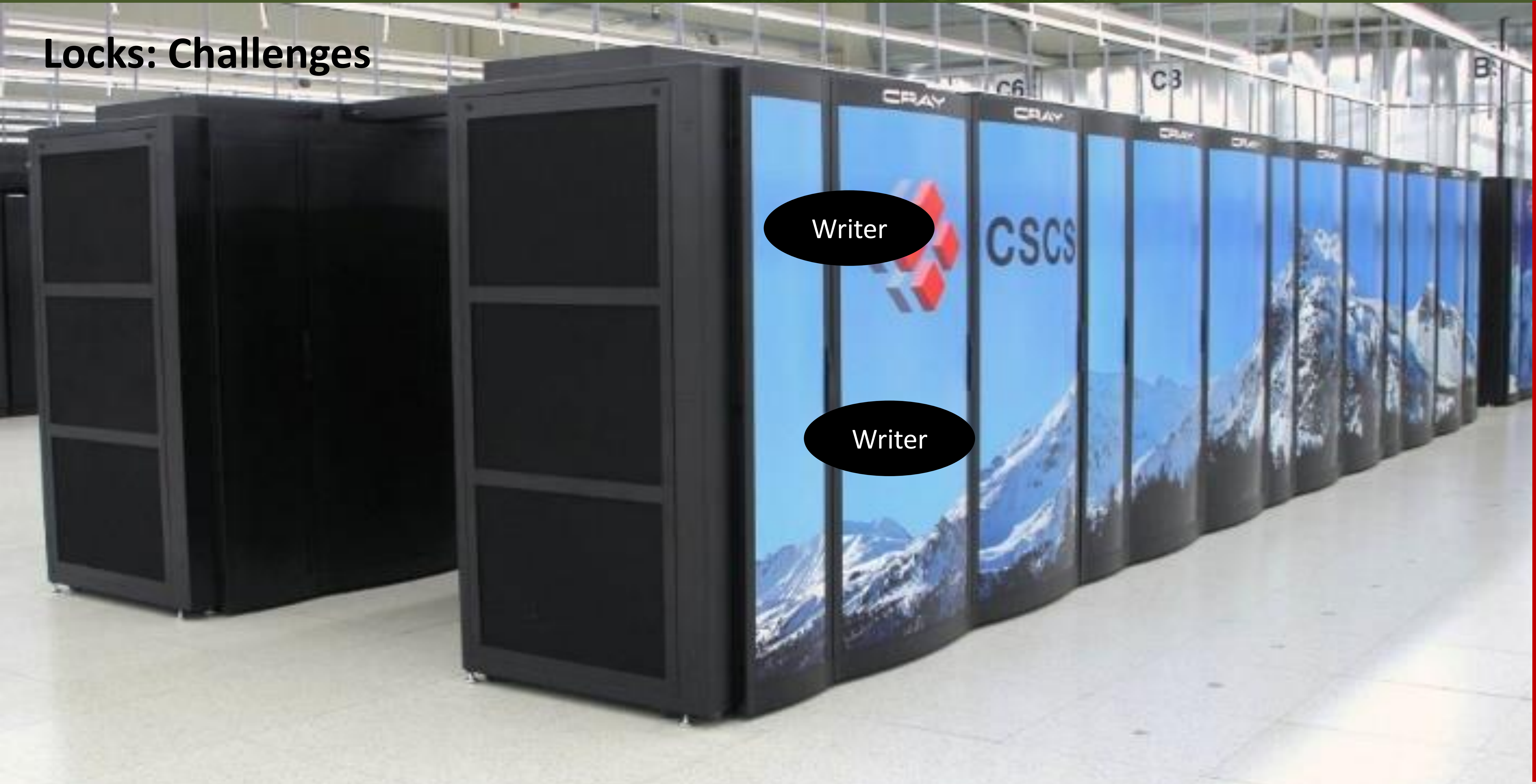
Locks: Challenges

Locks: Challenges

Locks: Challenges

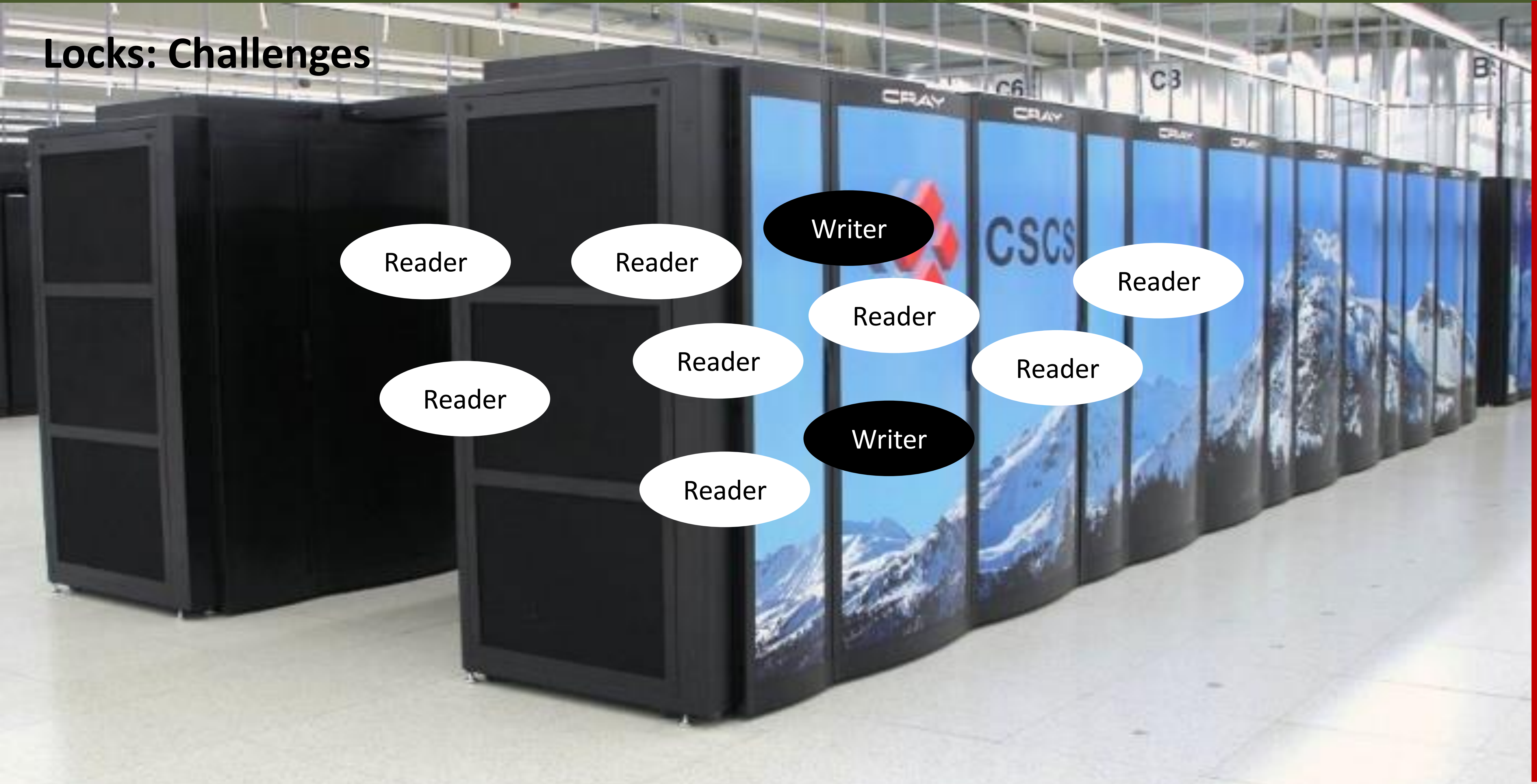
[1] V. Venkataramani et al. Tao: How facebook serves the social graph. SIGMOD’12.

Locks: Challenges



[1] V. Venkataramani et al. Tao: How facebook serves the social graph. SIGMOD'12.

Locks: Challenges



[1] V. Venkataramani et al. Tao: How facebook serves the social graph. SIGMOD'12.

Locks: Challenges

We need to distinguish between readers and writers

[1] V. Venkataramani et al. Tao: How facebook serves the social graph. SIGMOD'12.

Locks: Challenges

We need to distinguish between readers and writers

Reader

Reader

Reader

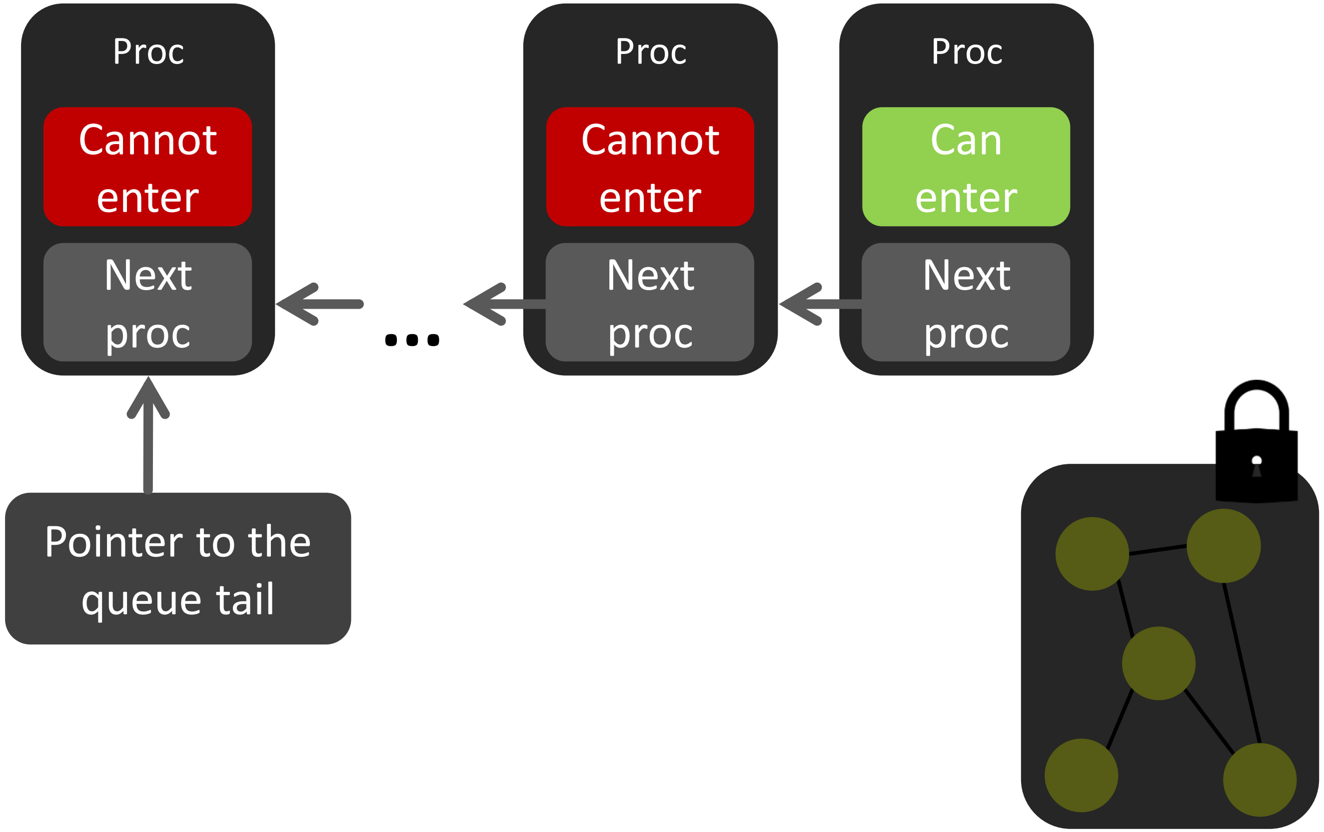
Writer

We need flexible performance for both types of processes

[1] V. Venkataramani et al. Tao: How facebook serves the social graph. SIGMOD'12.

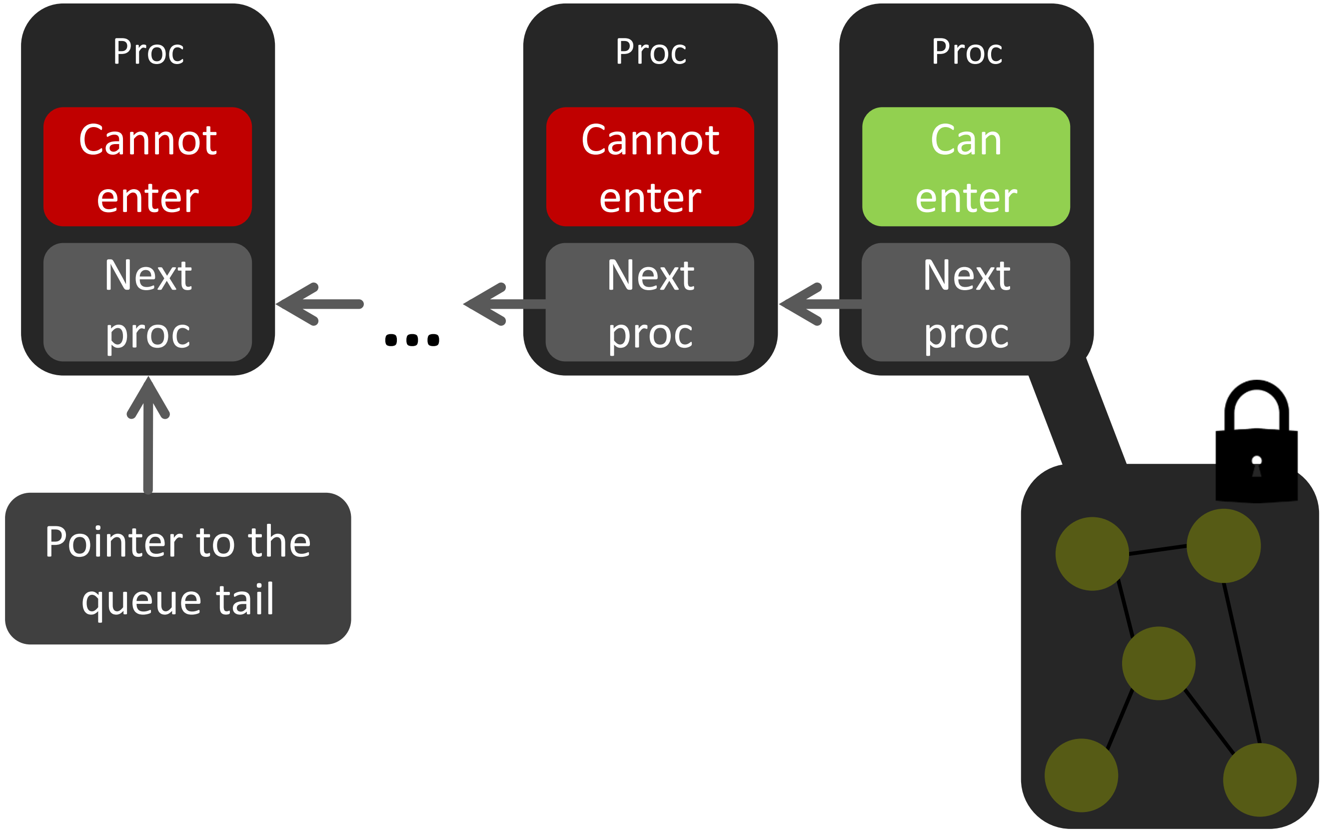
What will we use in the design?

Ingredient 1 - MCS Locks



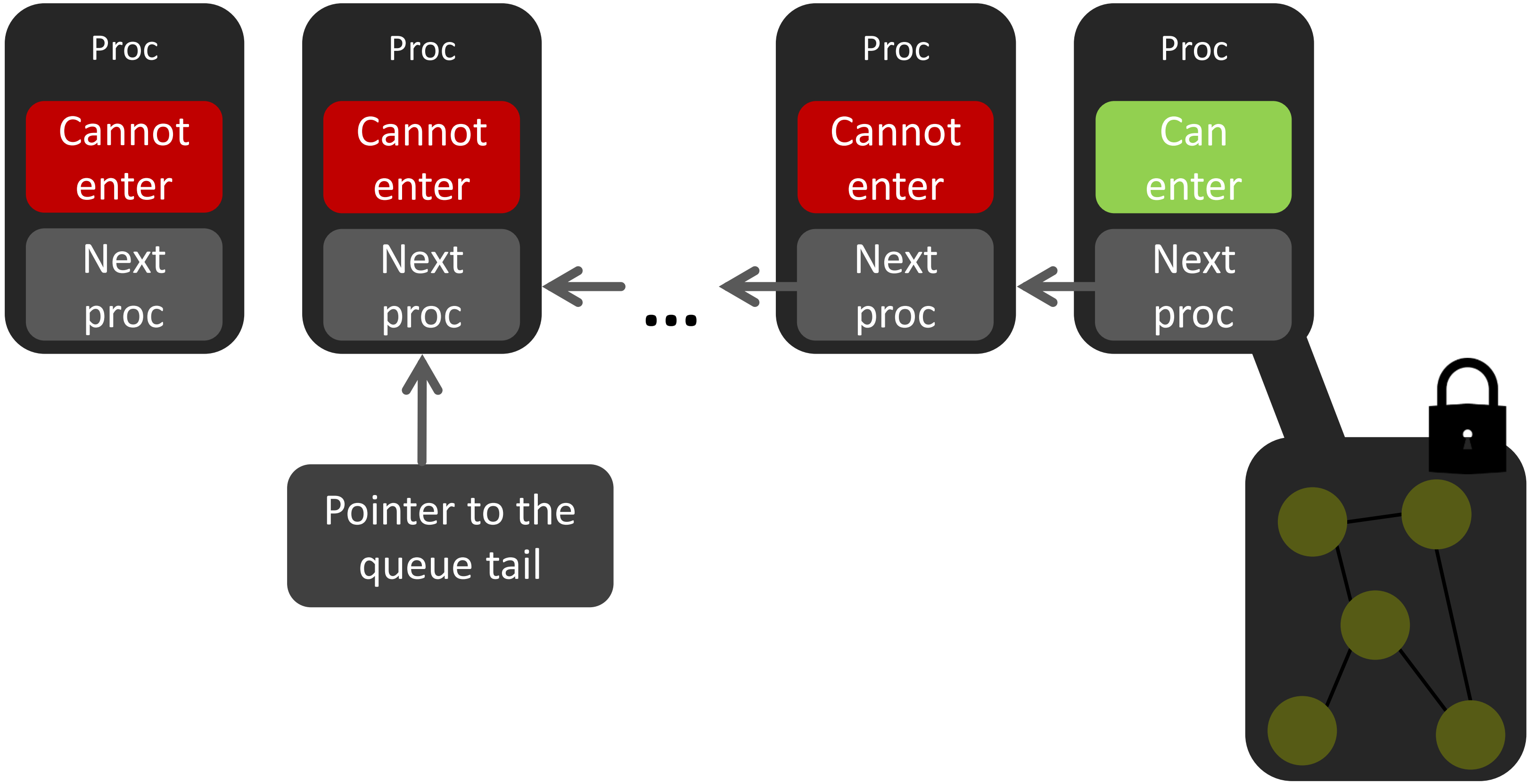
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



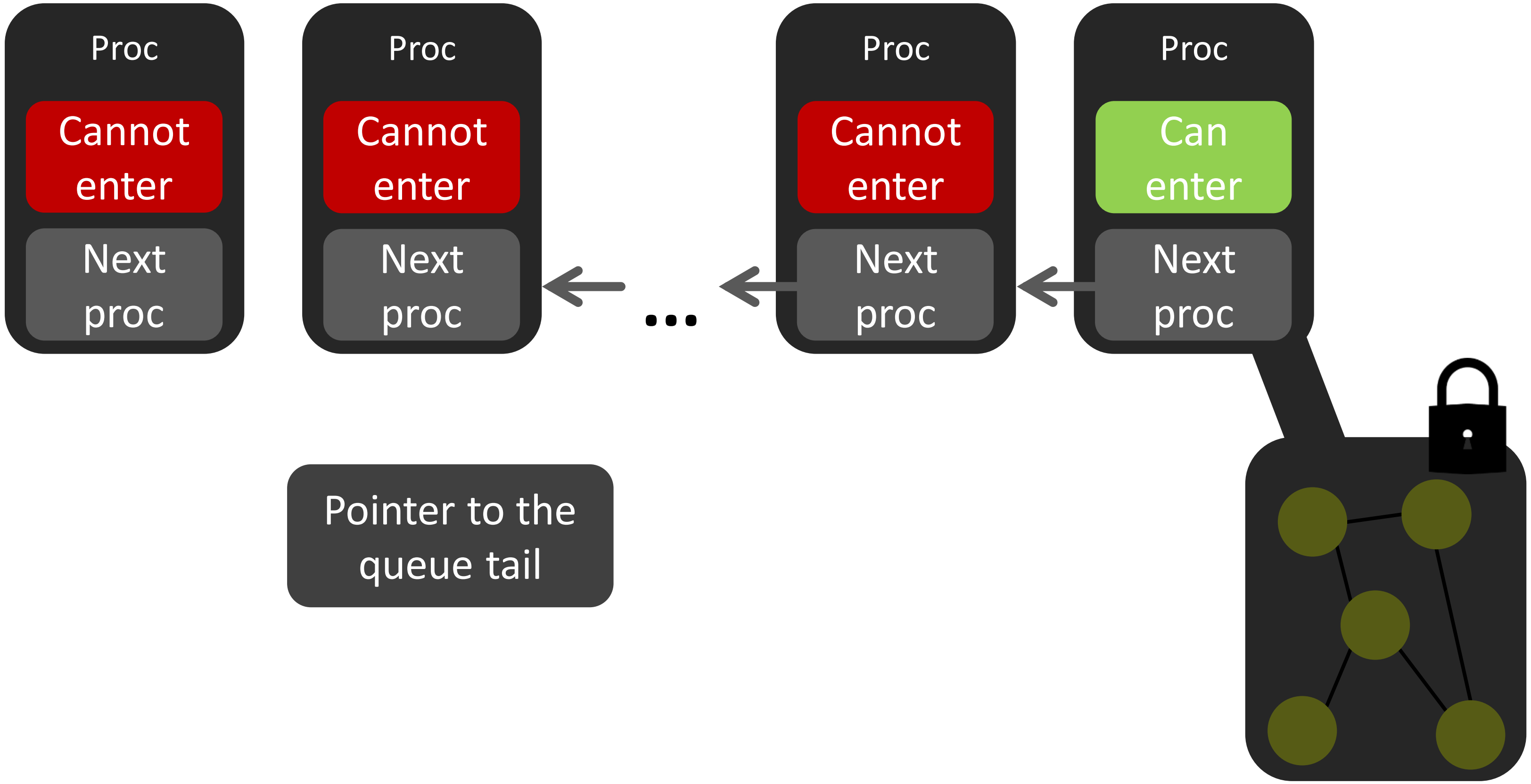
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



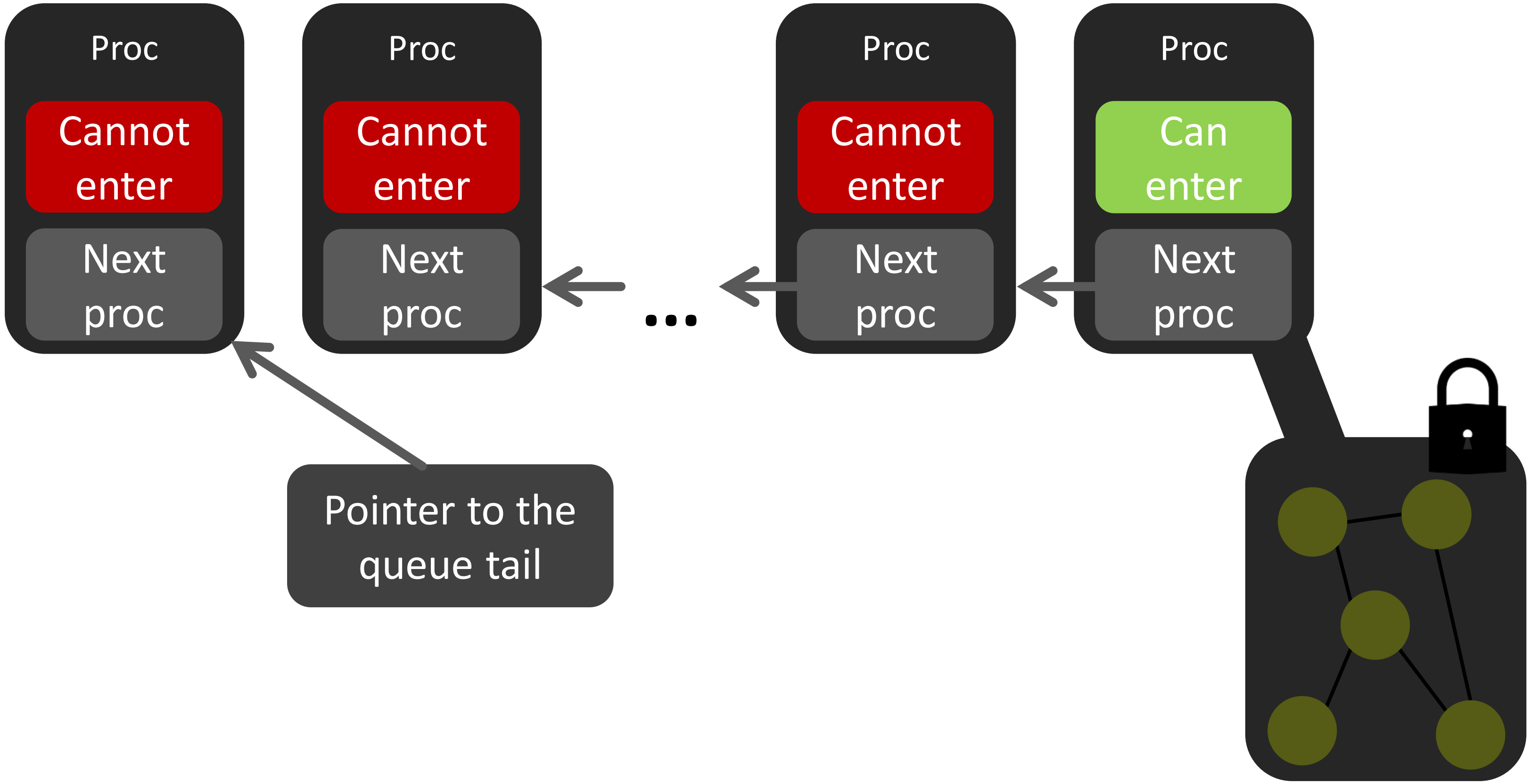
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



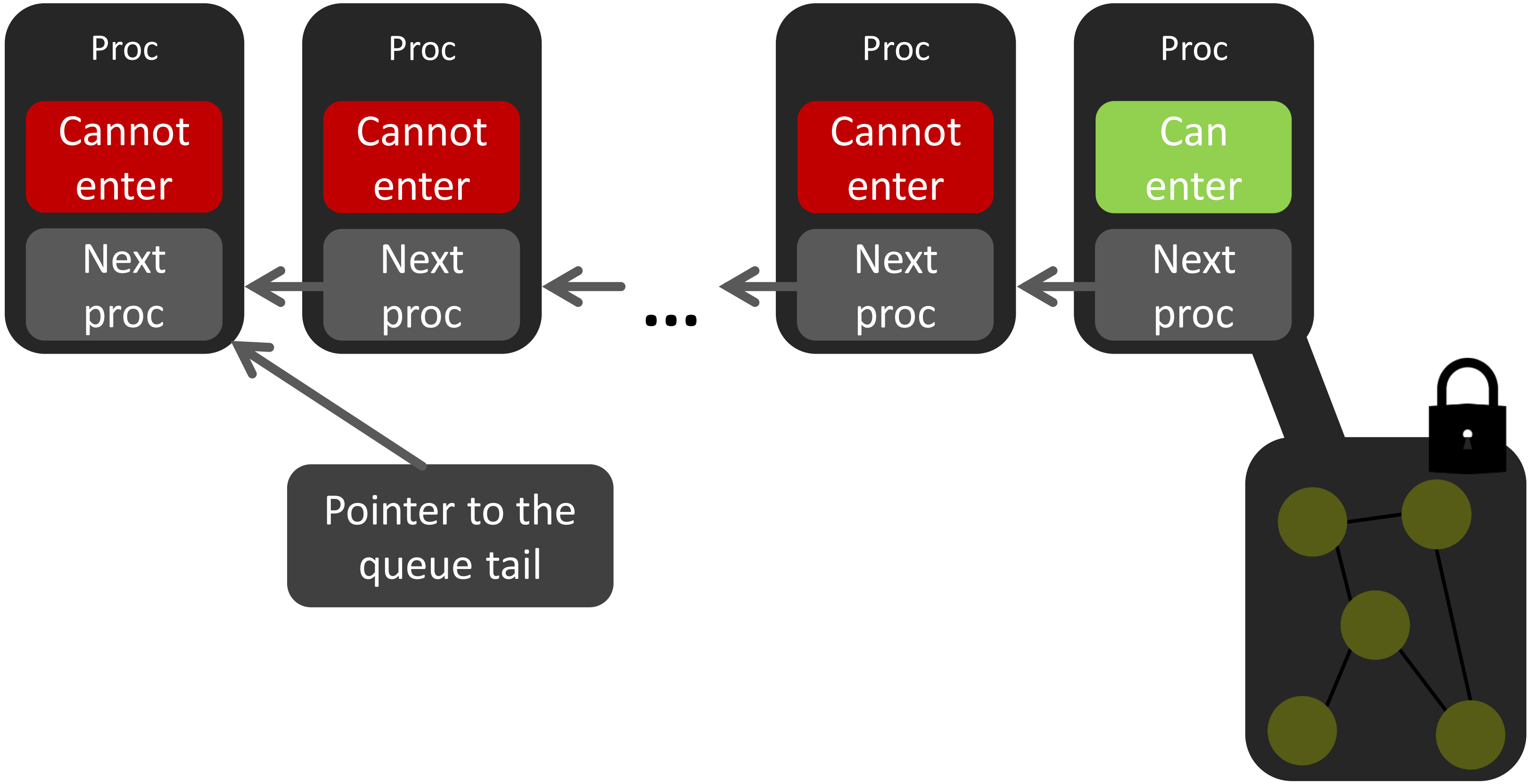
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



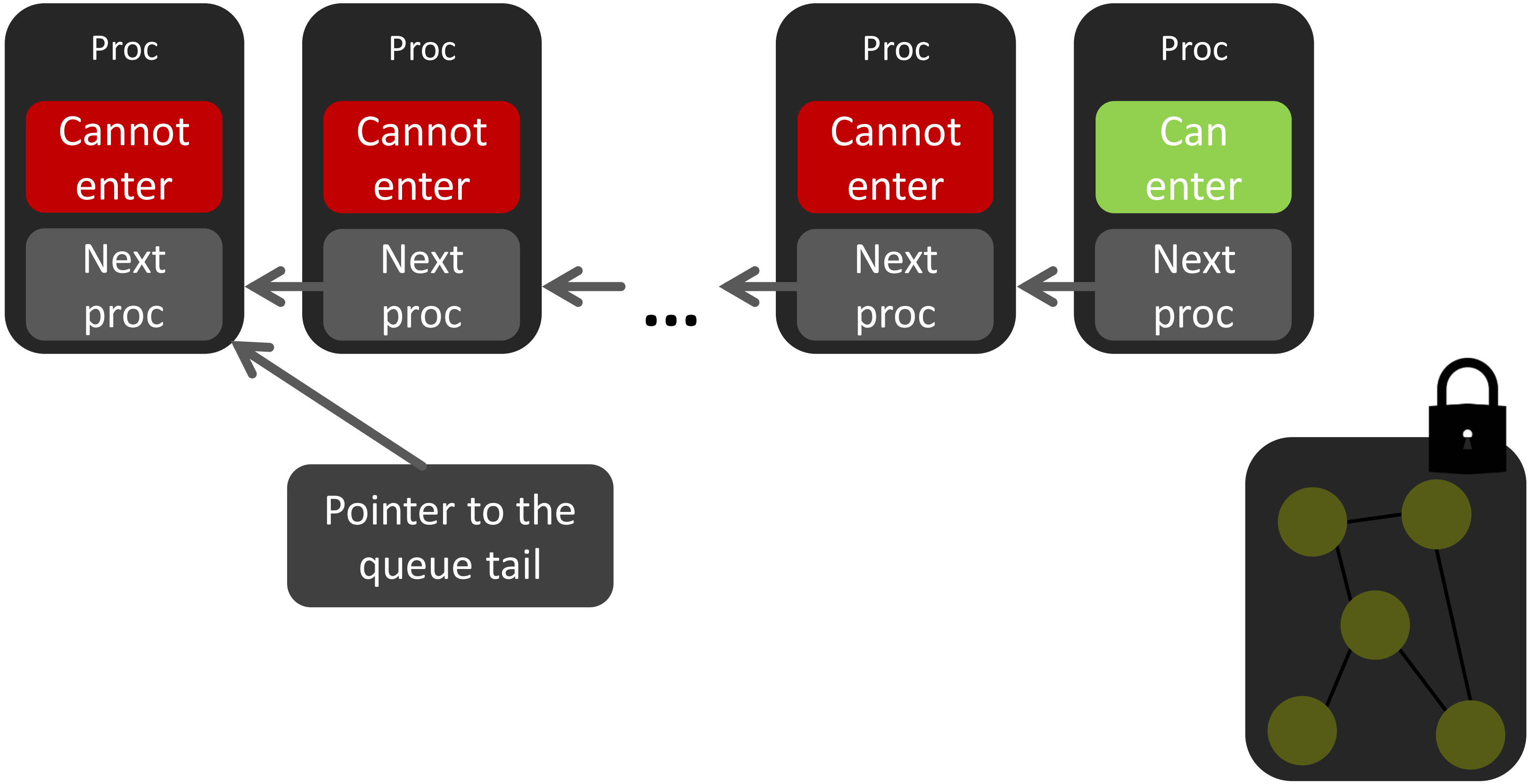
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



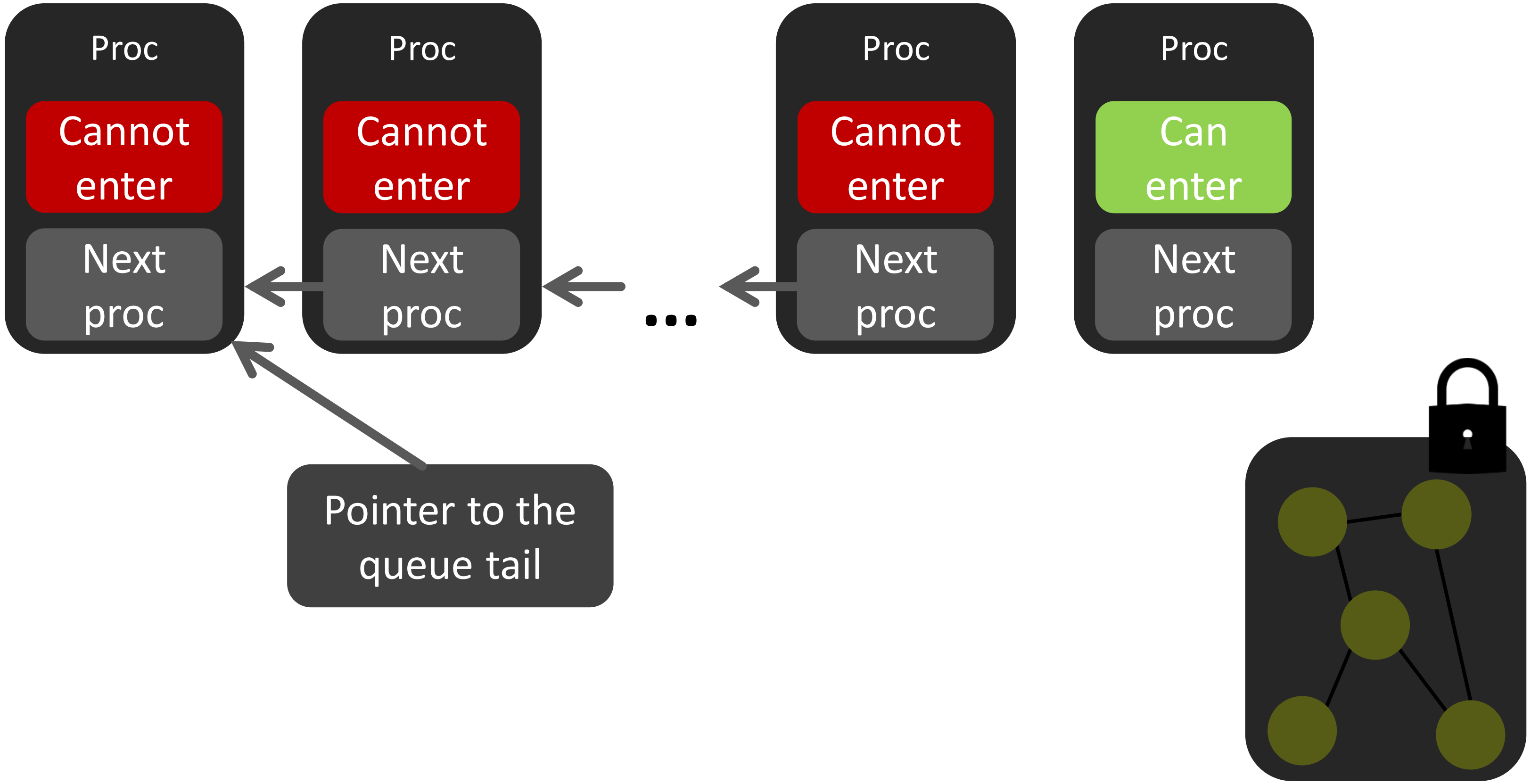
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



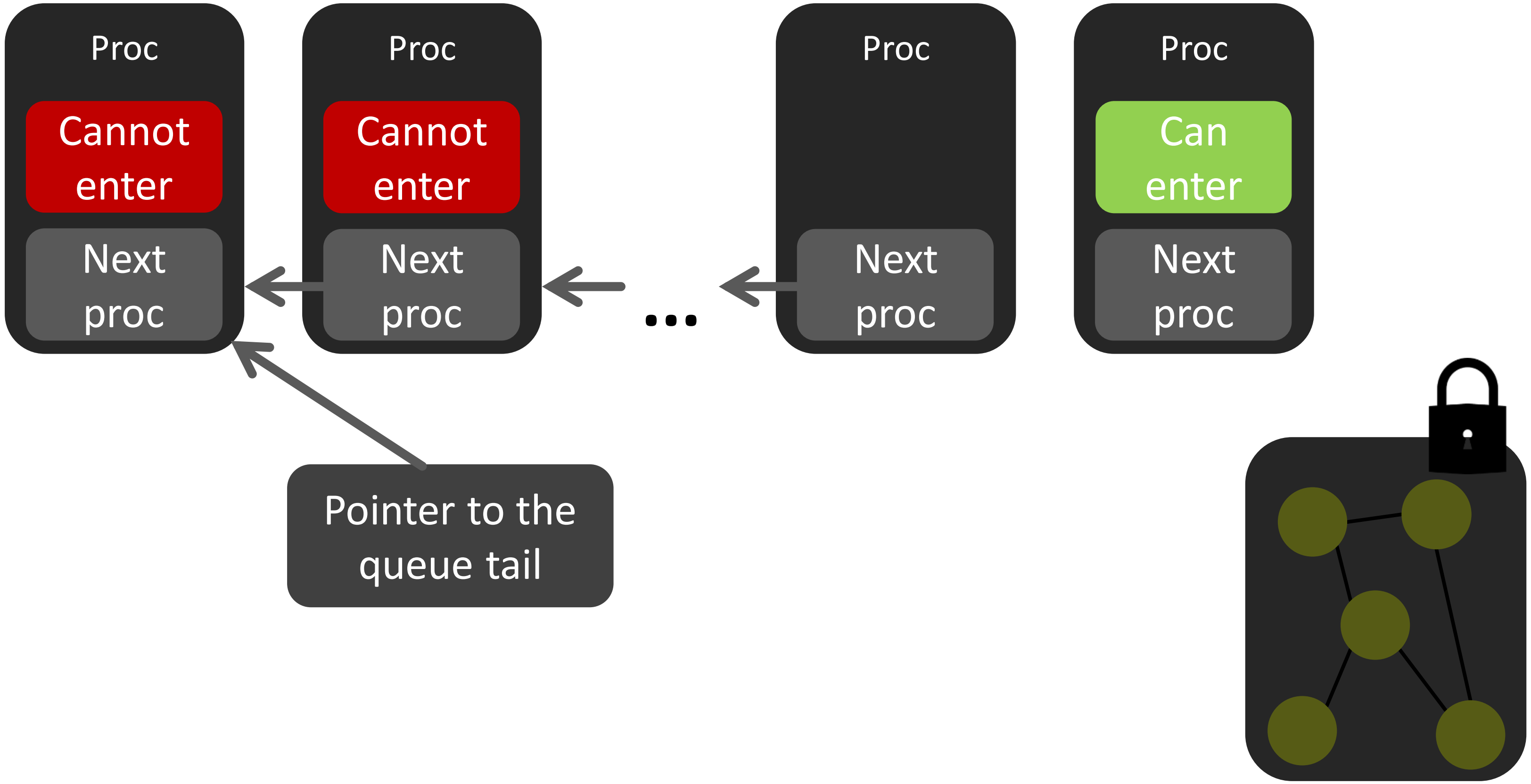
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



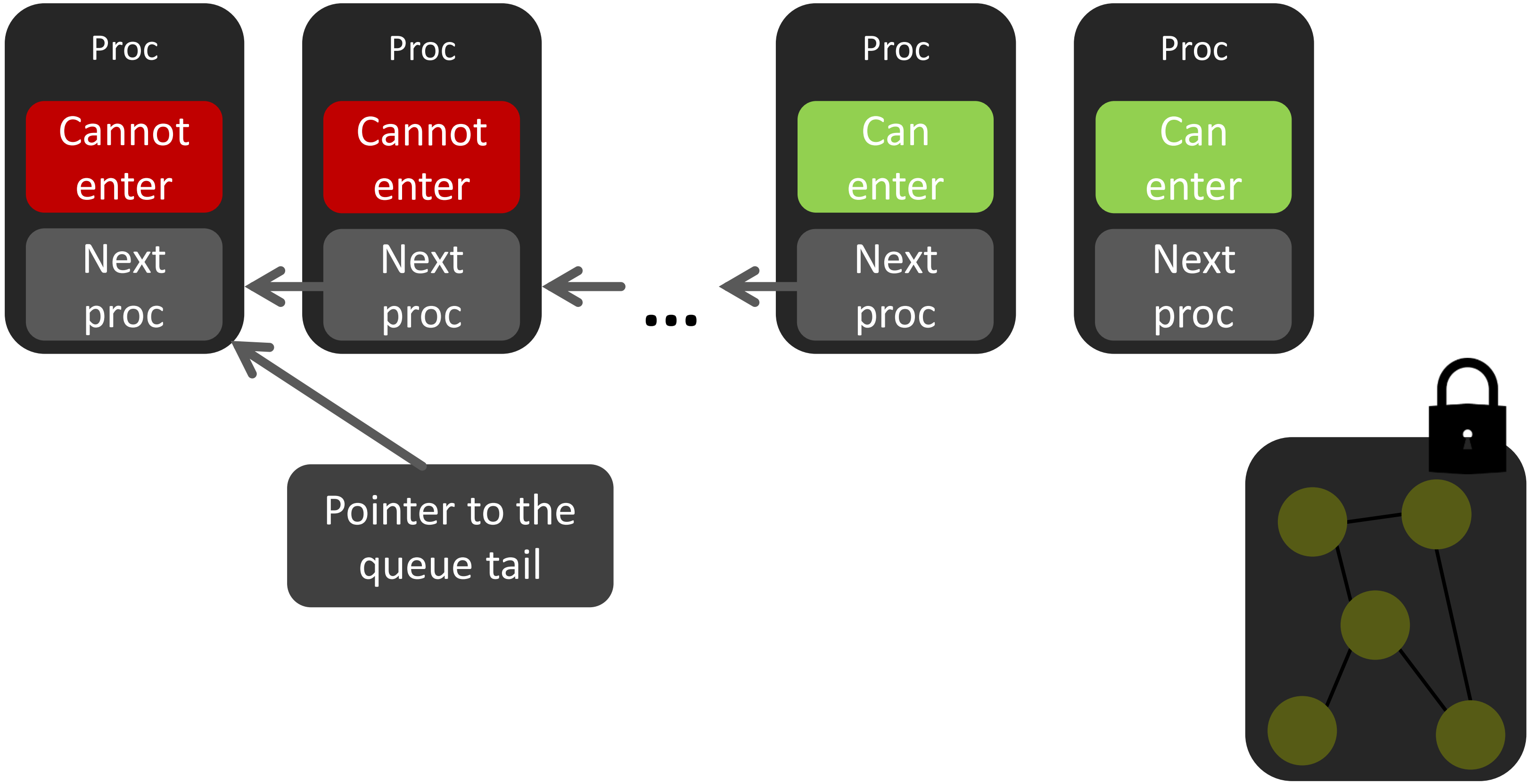
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



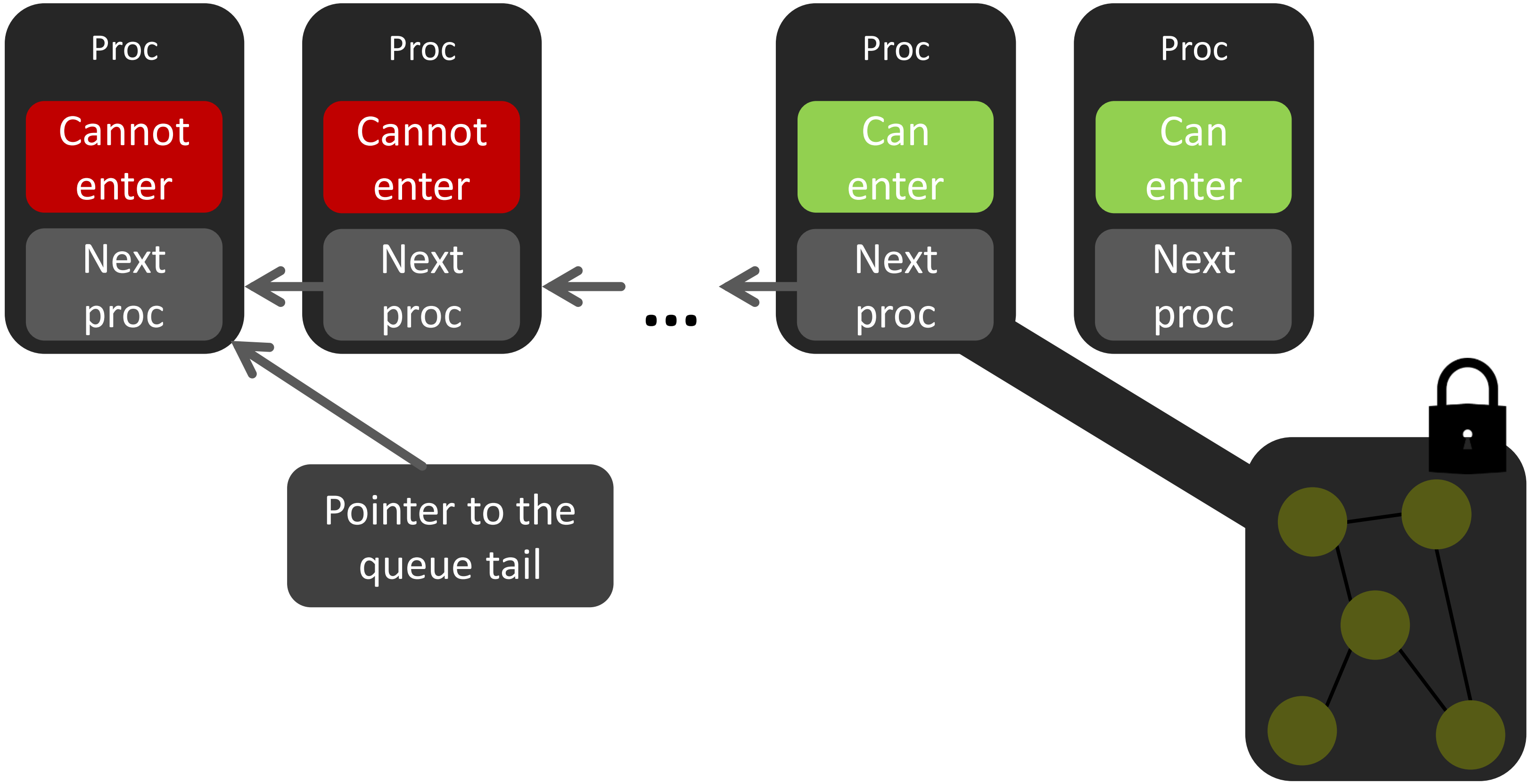
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



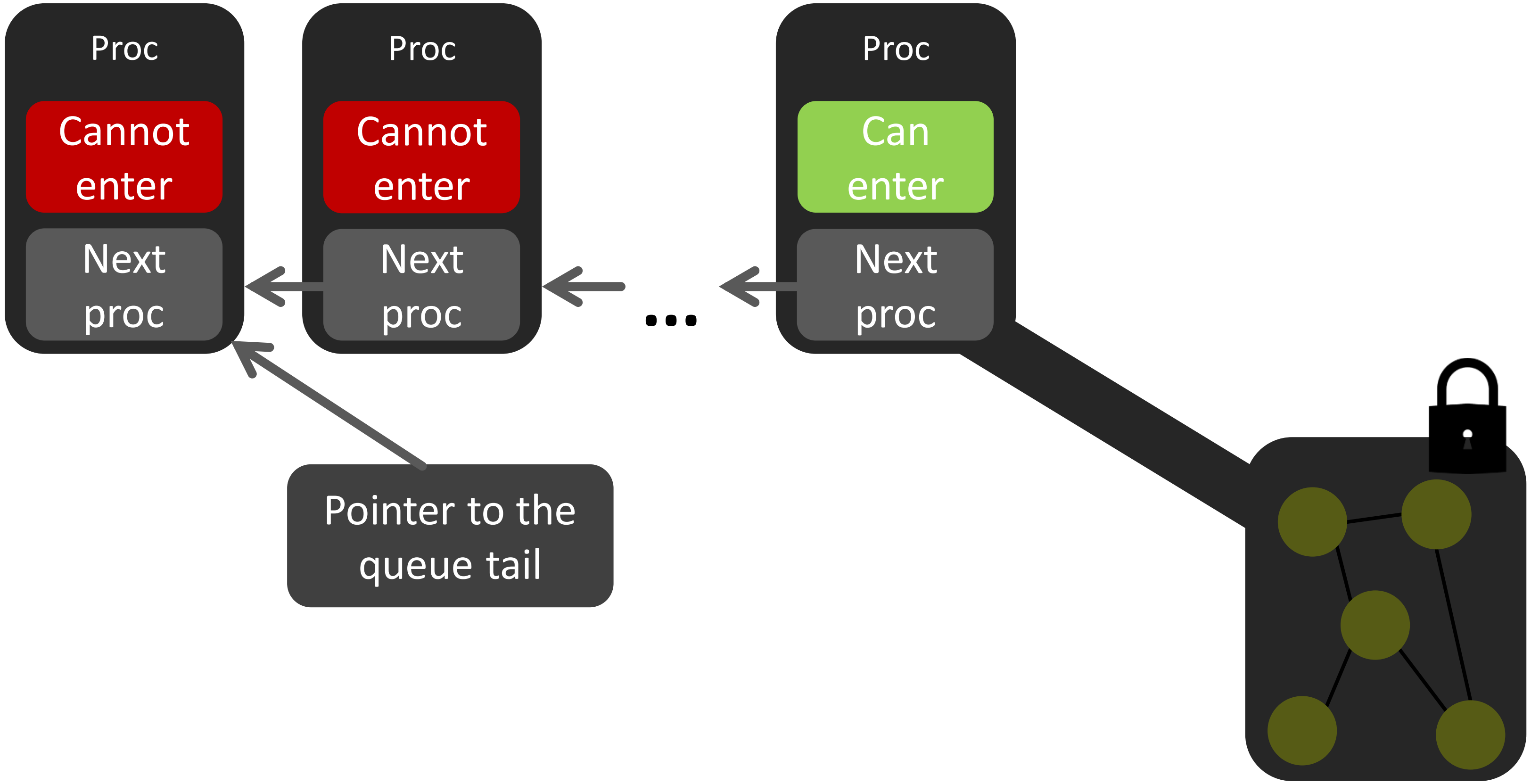
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks



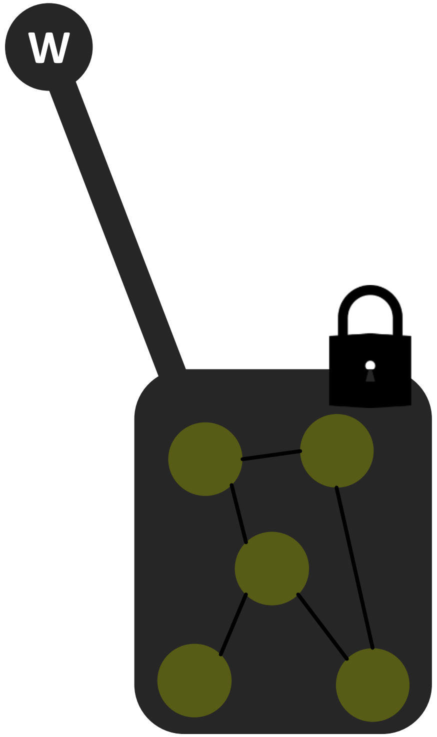
Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

Ingredient 1 - MCS Locks

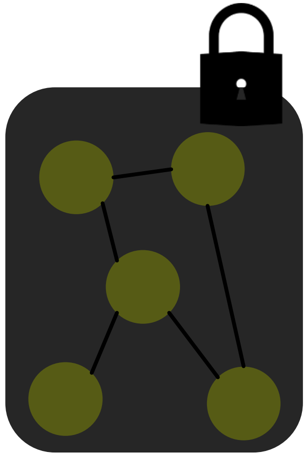


Mellor-Crummey and Scott: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM TOCS'91

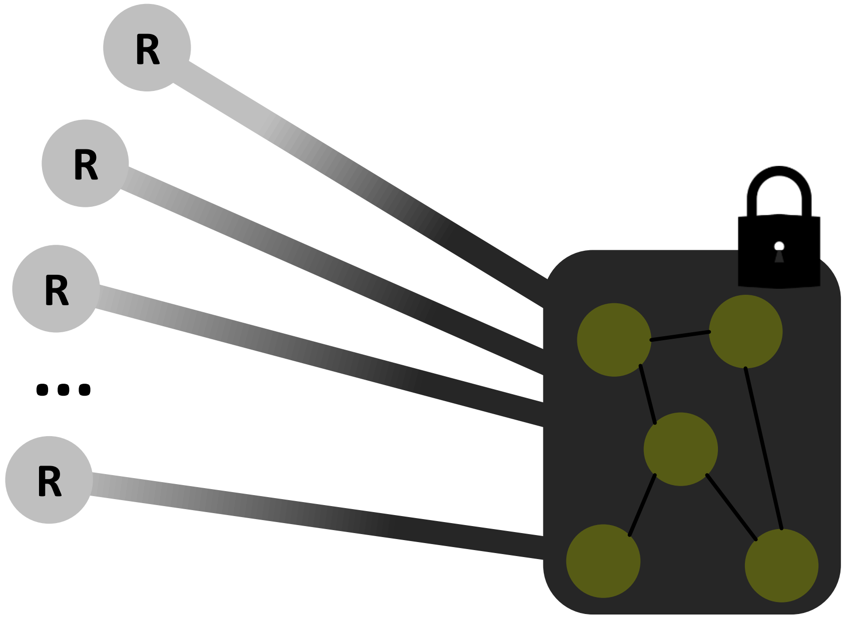
Ingredient 2 - Reader-Writer Locks



Ingredient 2 - Reader-Writer Locks



Ingredient 2 - Reader-Writer Locks



How to manage the design complexity?

How to manage the design complexity?

What mechanism to use for efficient implementation?

How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?

How to manage the design complexity?

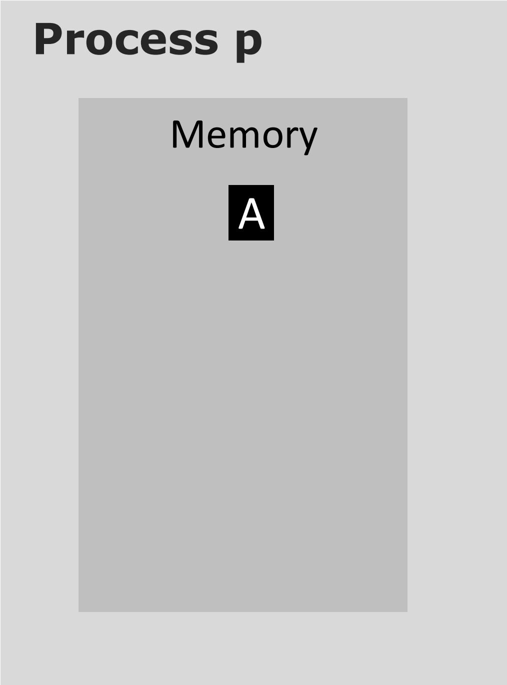
How to ensure tunable performance?

What mechanism to use for efficient implementation?

Remote Memory Access (RMA) Programming

TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

REMOTE MEMORY ACCESS (RMA) PROGRAMMING

TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

Remote Memory Access (RMA) Programming

Process p

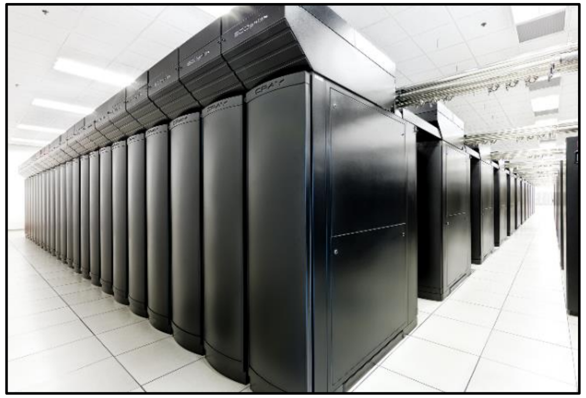
Memory

A

Process q

Memory

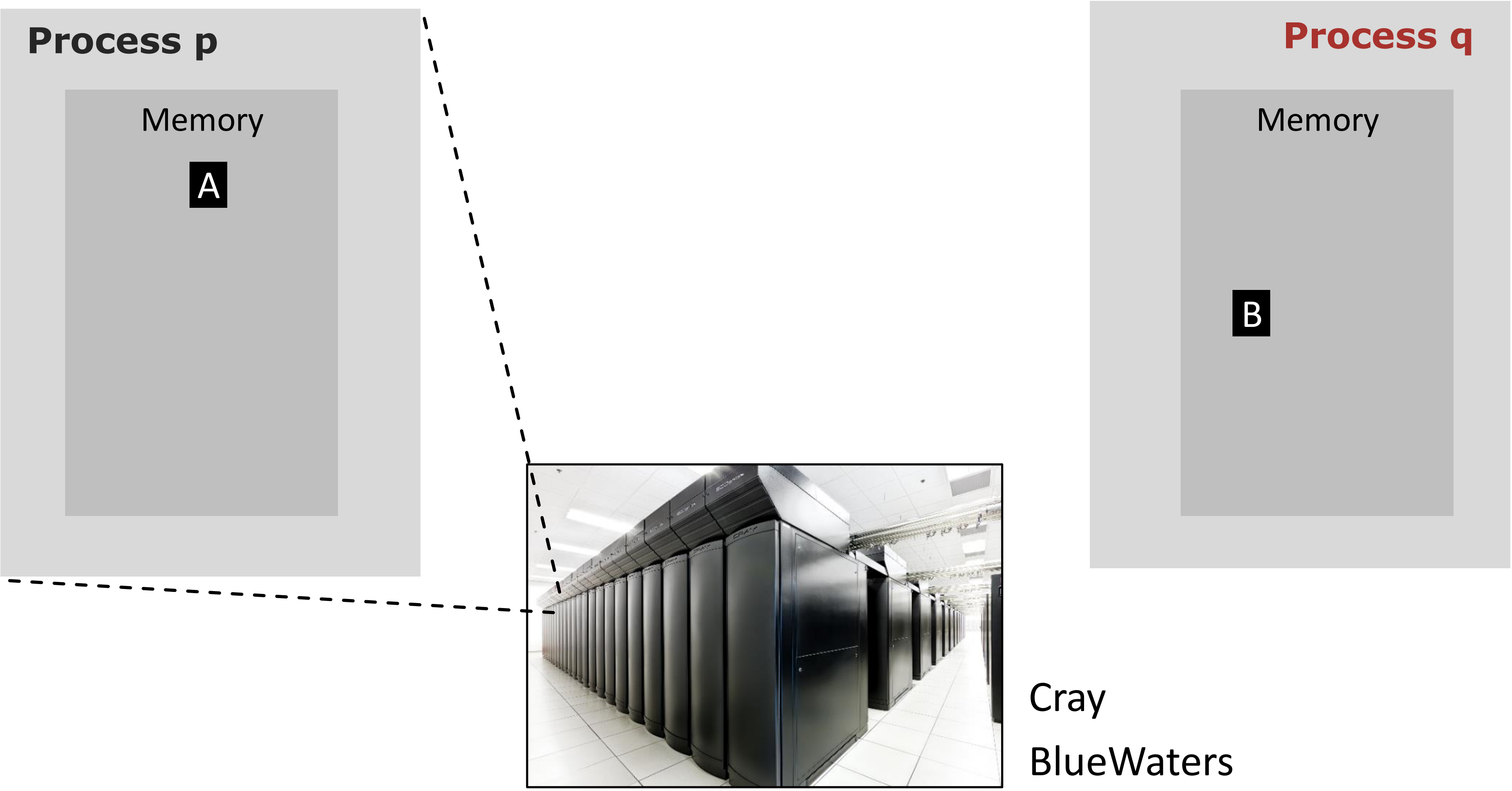
B



Cray BlueWaters

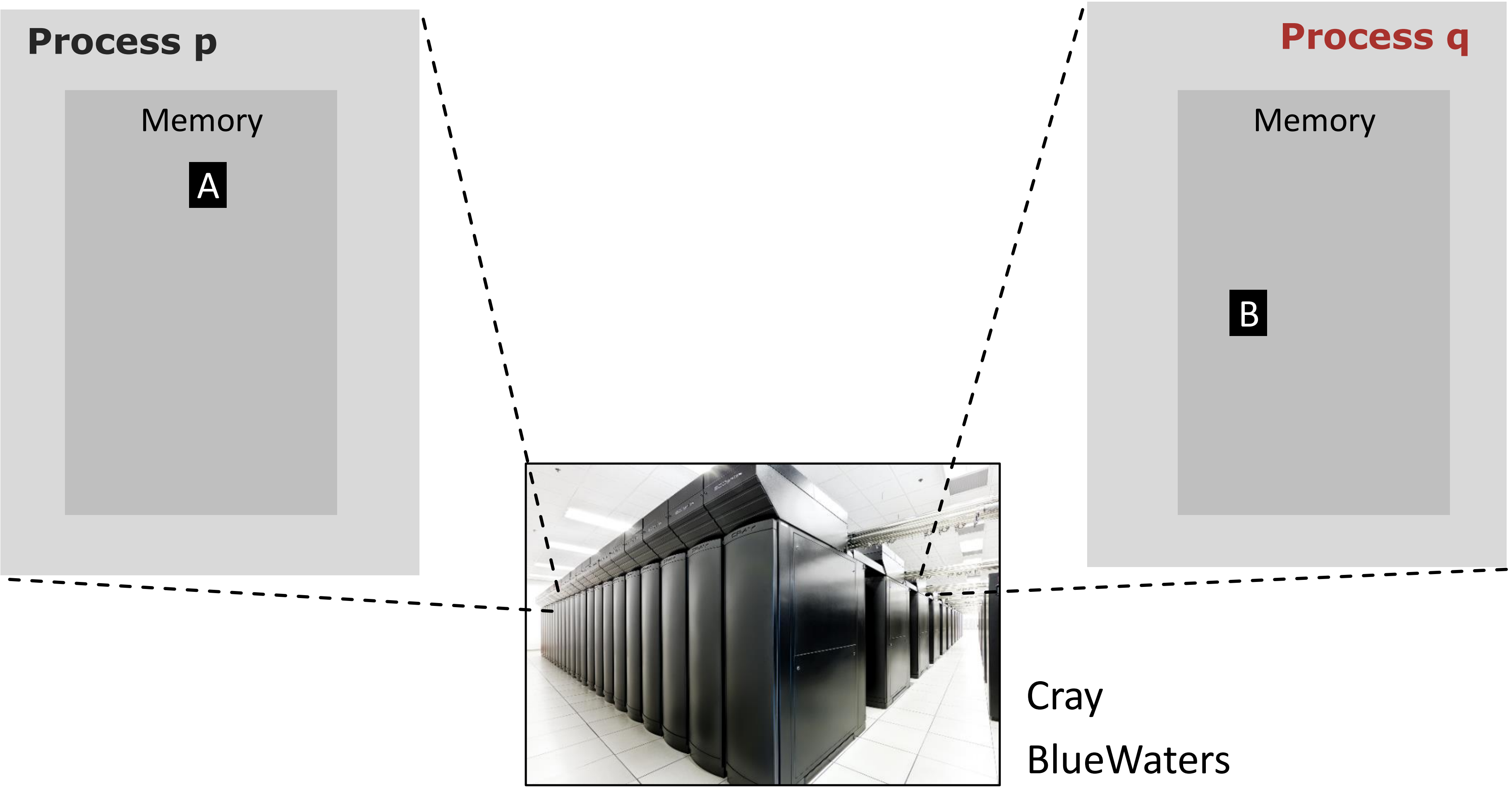
TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

Remote Memory Access (RMA) Programming



TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

Remote Memory Access (RMA) Programming



TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

Remote Memory Access (RMA) Programming

Process p

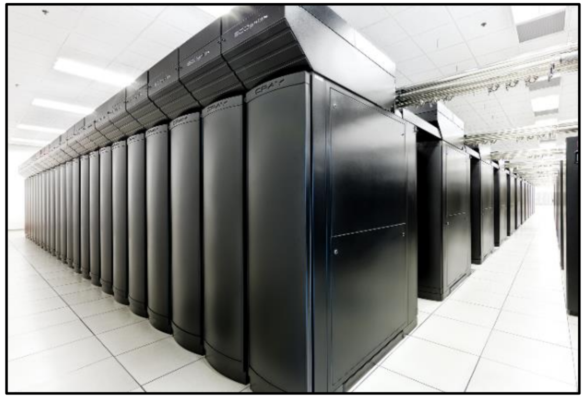
Memory

A

Process q

Memory

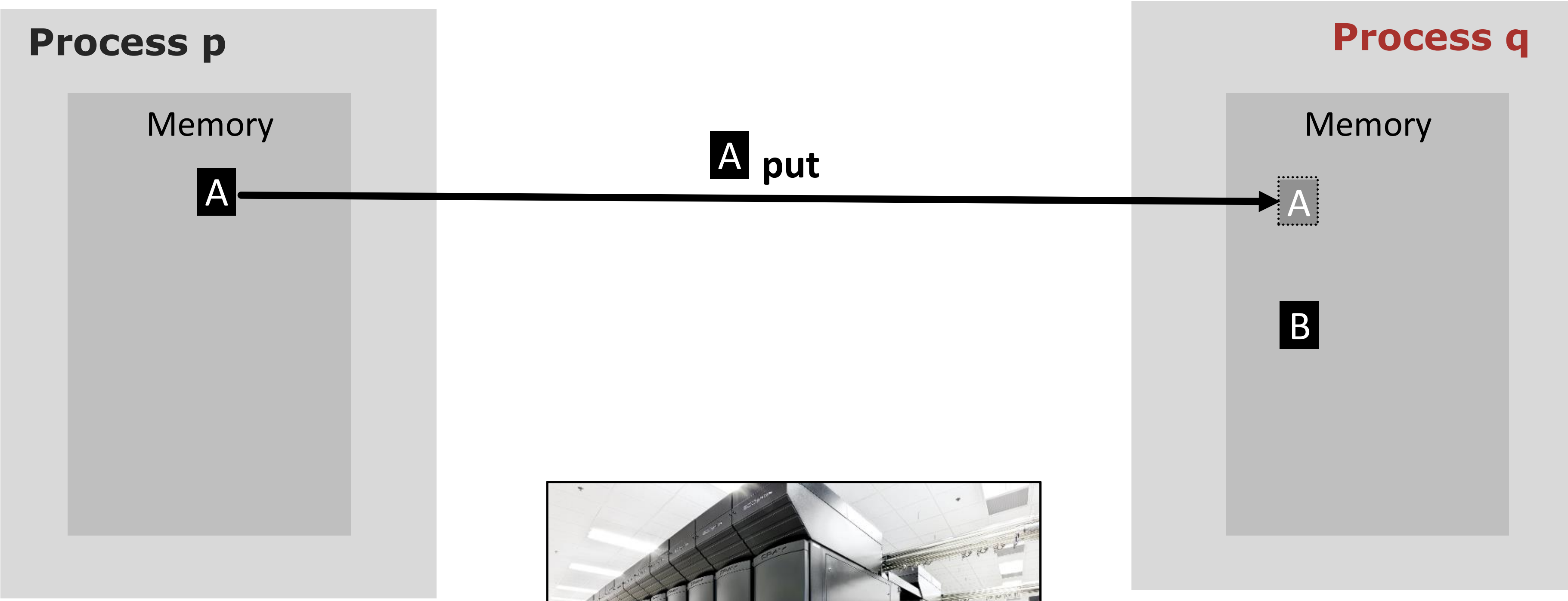
B

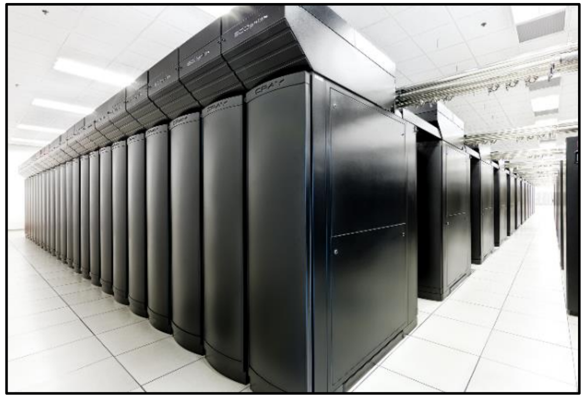


Cray
BlueWaters

TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

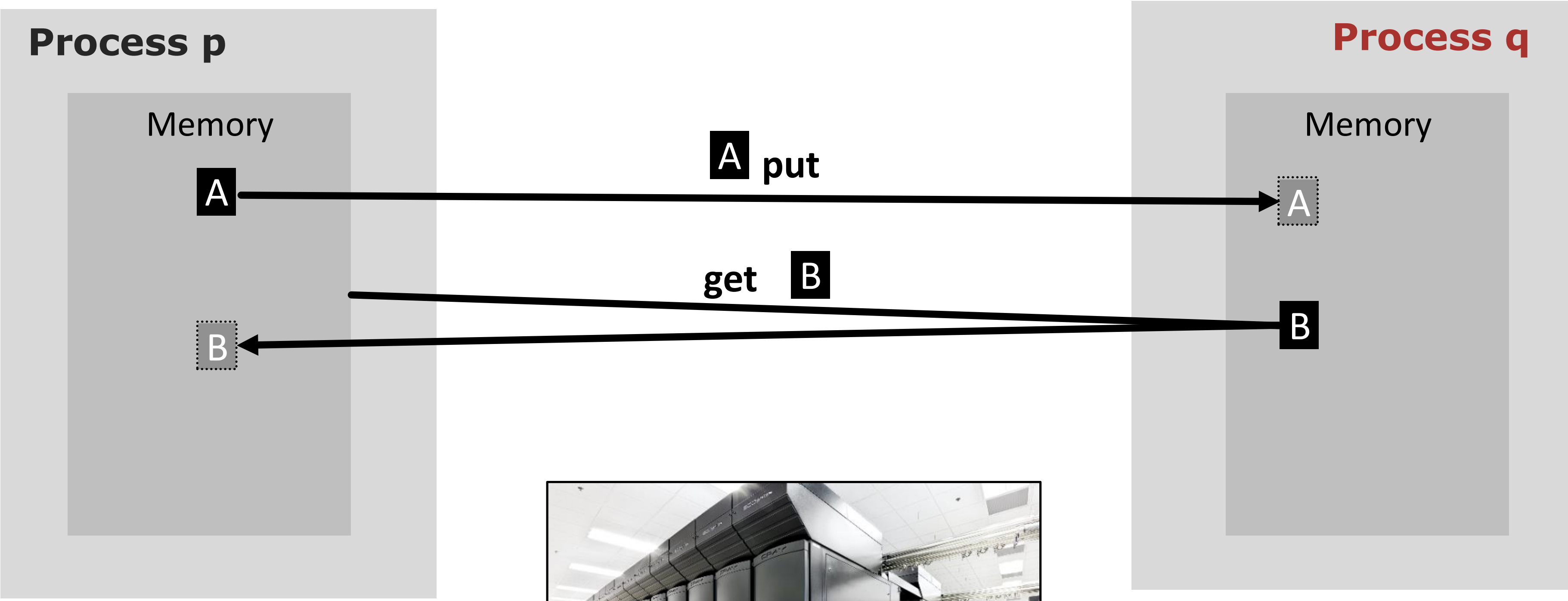
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

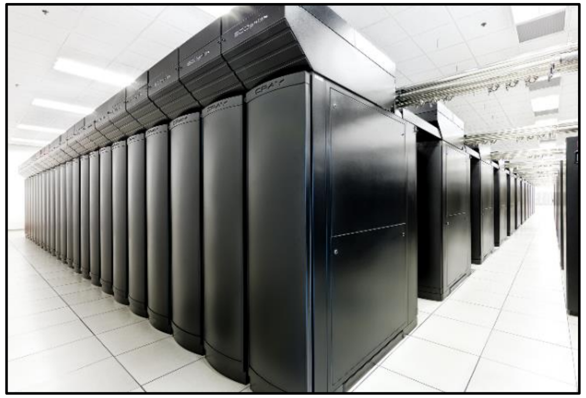




Cray
BlueWaters

REMOTE MEMORY ACCESS (RMA) PROGRAMMING

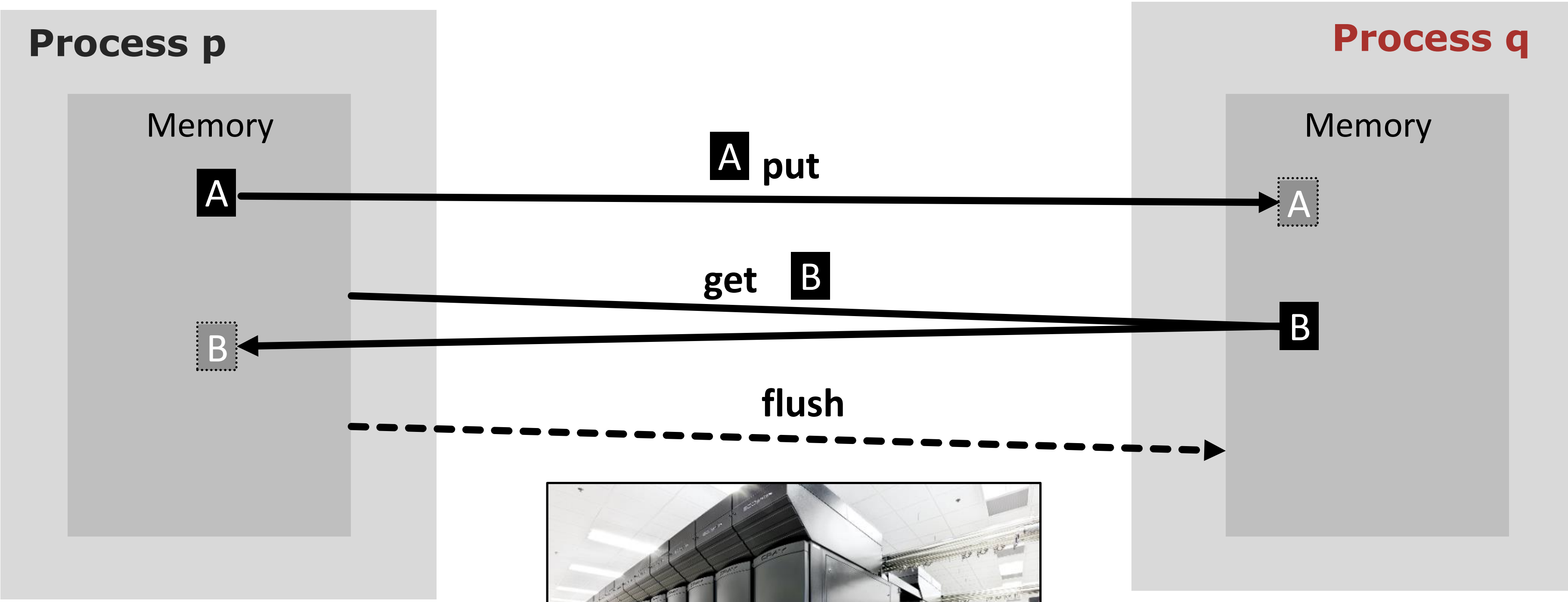


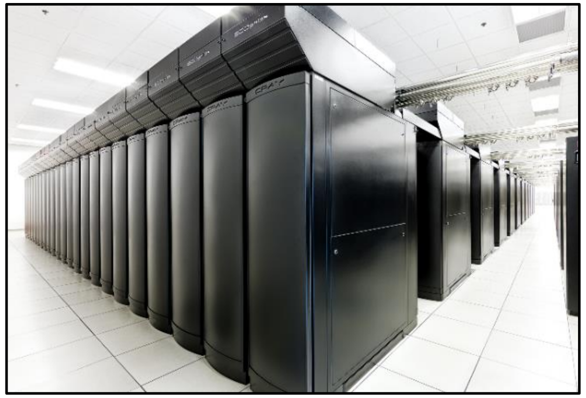


Cray
BlueWaters

TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

Remote Memory Access (RMA) Programming

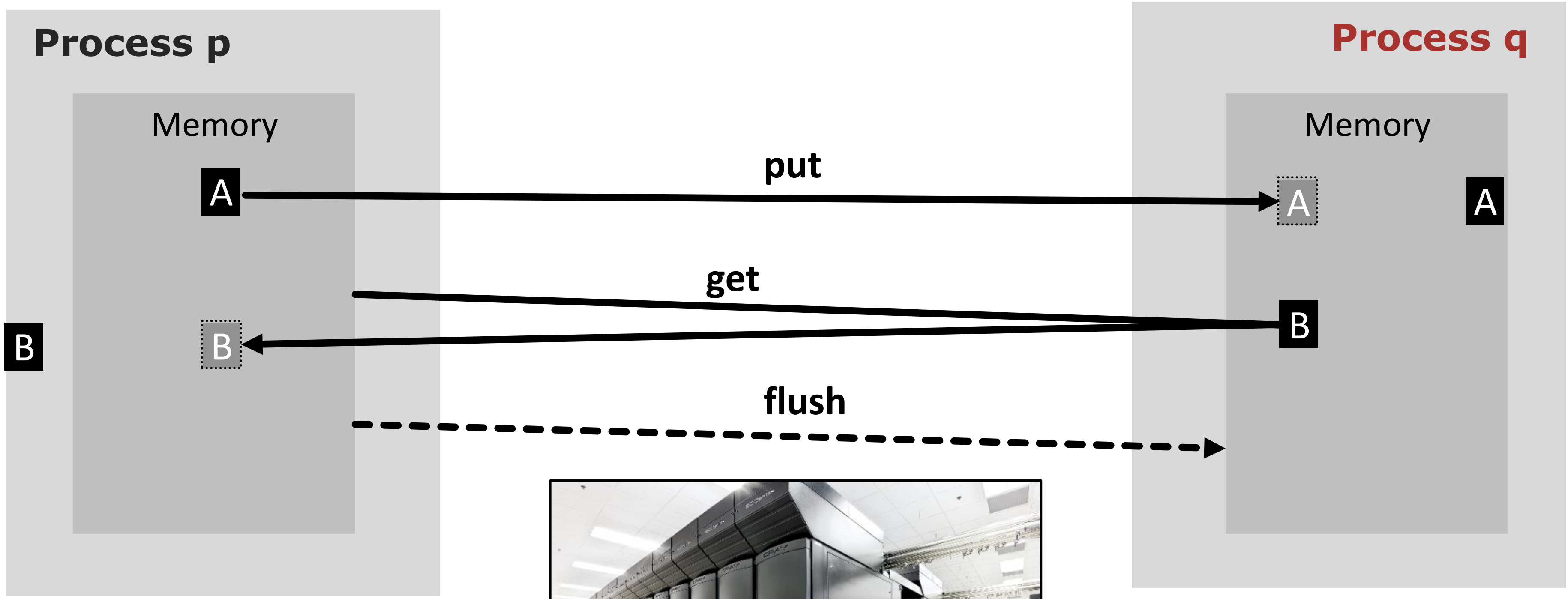


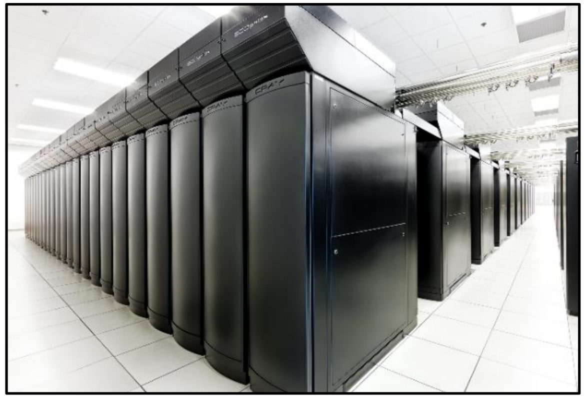


Cray
BlueWaters

TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

REMOTE MEMORY ACCESS (RMA) PROGRAMMING



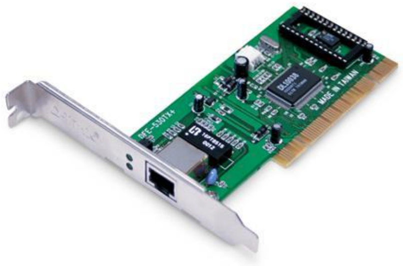


Cray
BlueWaters

TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

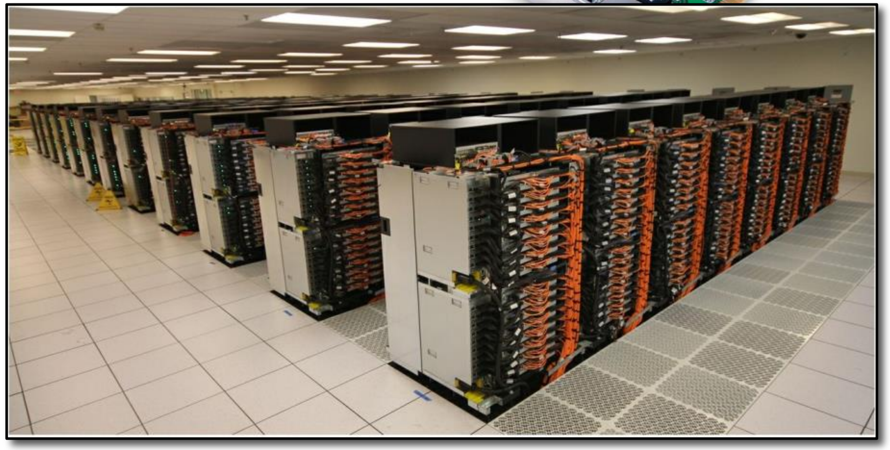
REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).



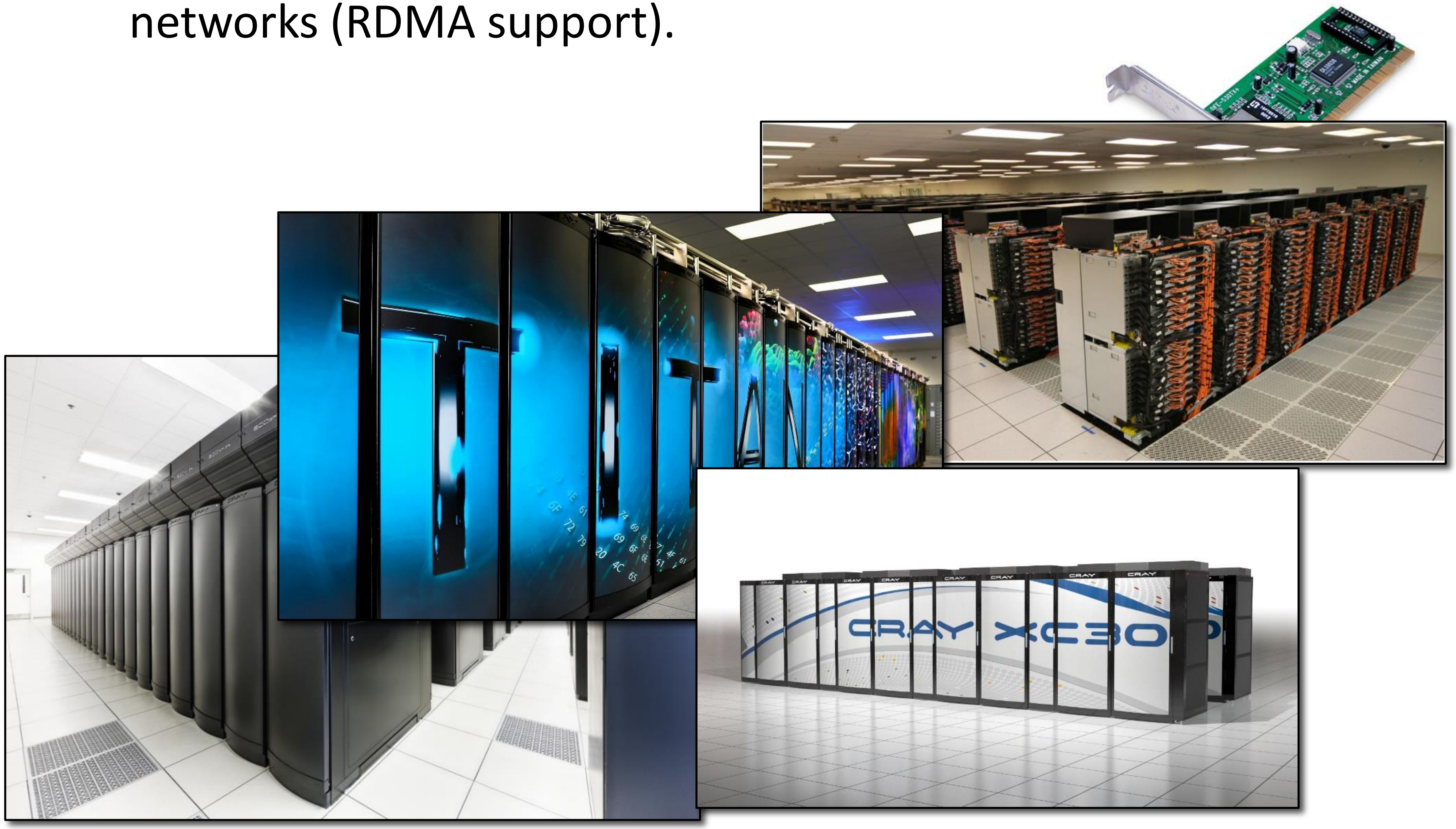
Remote Memory Access Programming

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).



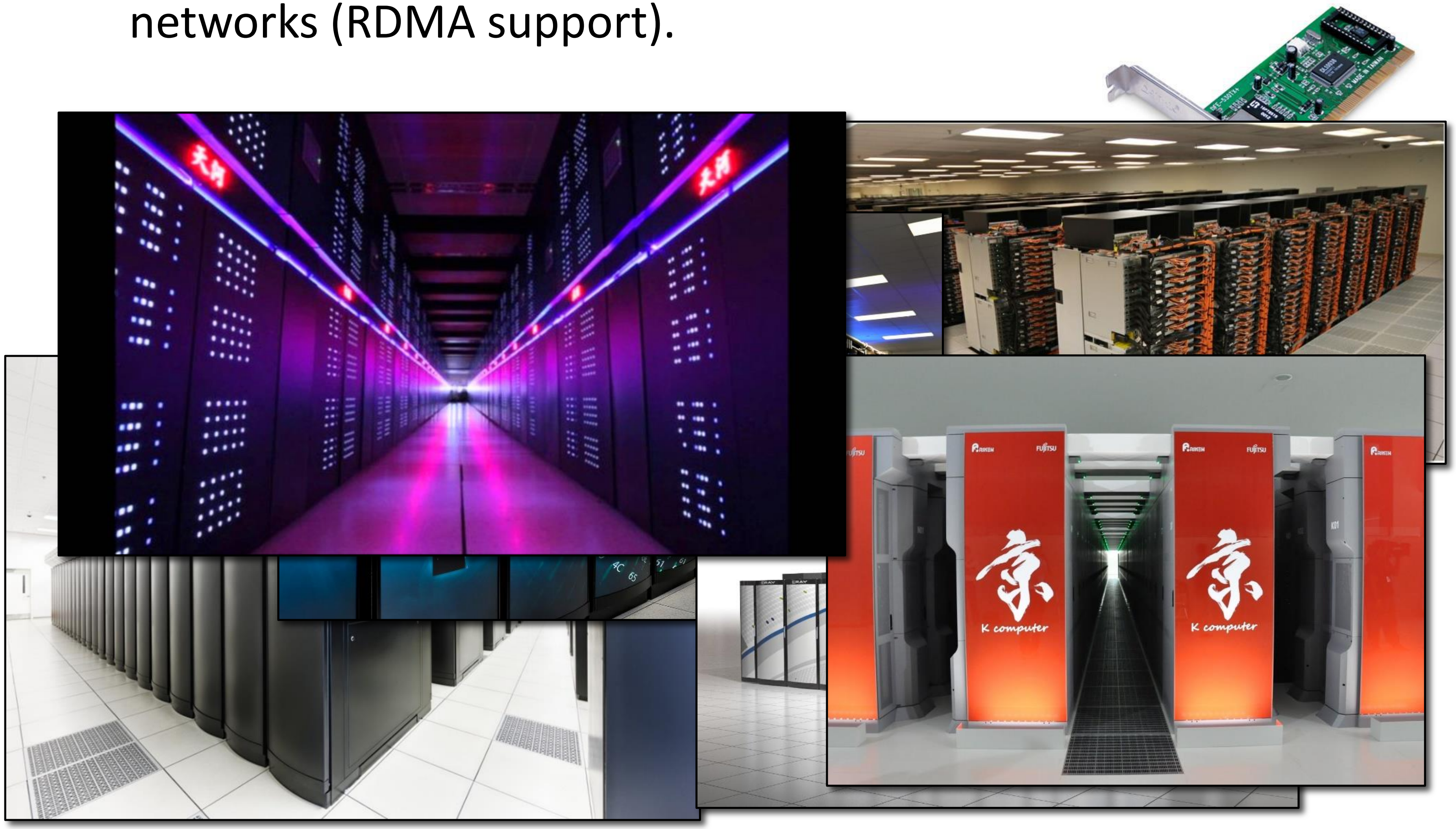
Remote Memory Access Programming

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).



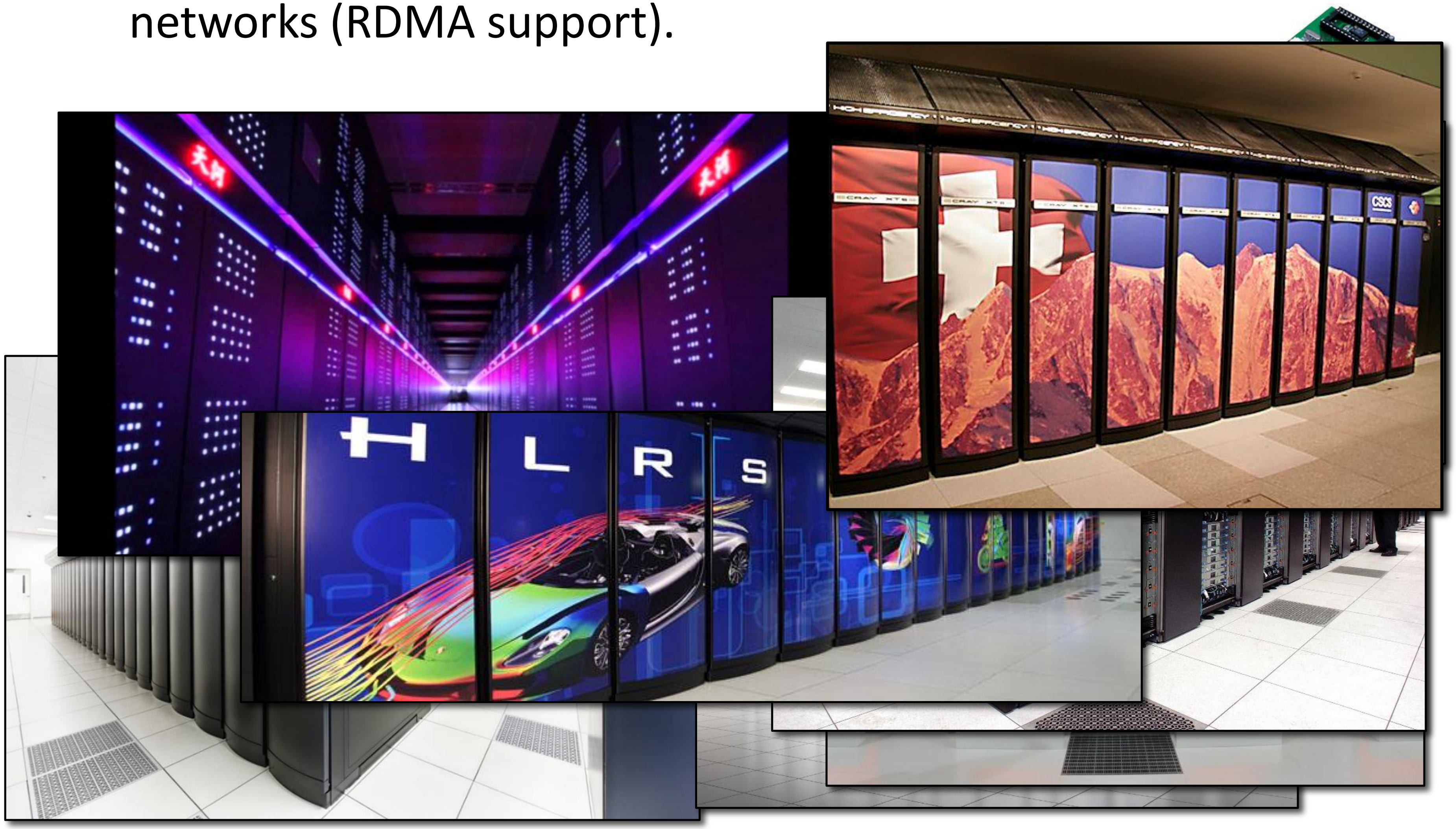
Remote Memory Access Programming

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).

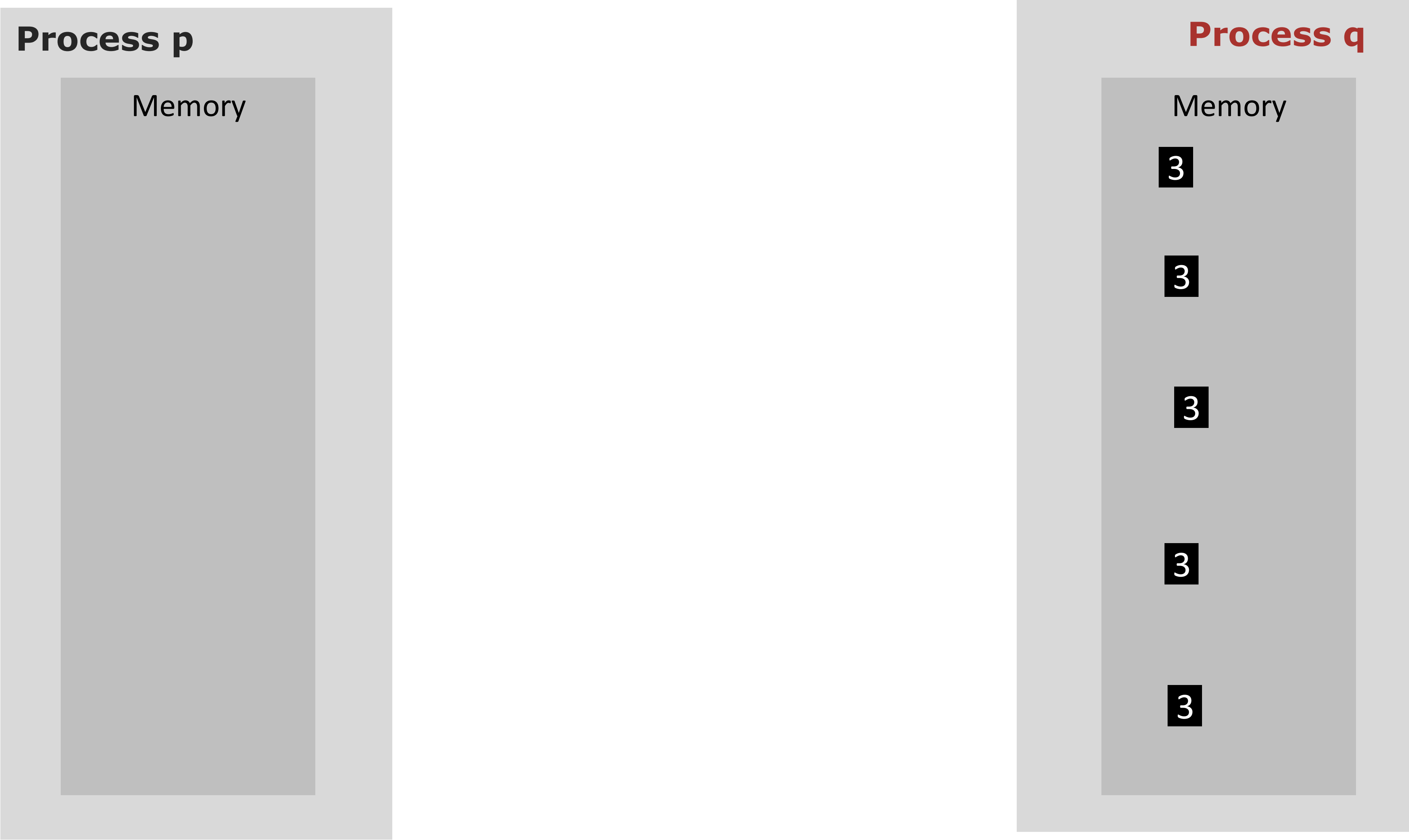


Remote Memory Access Programming

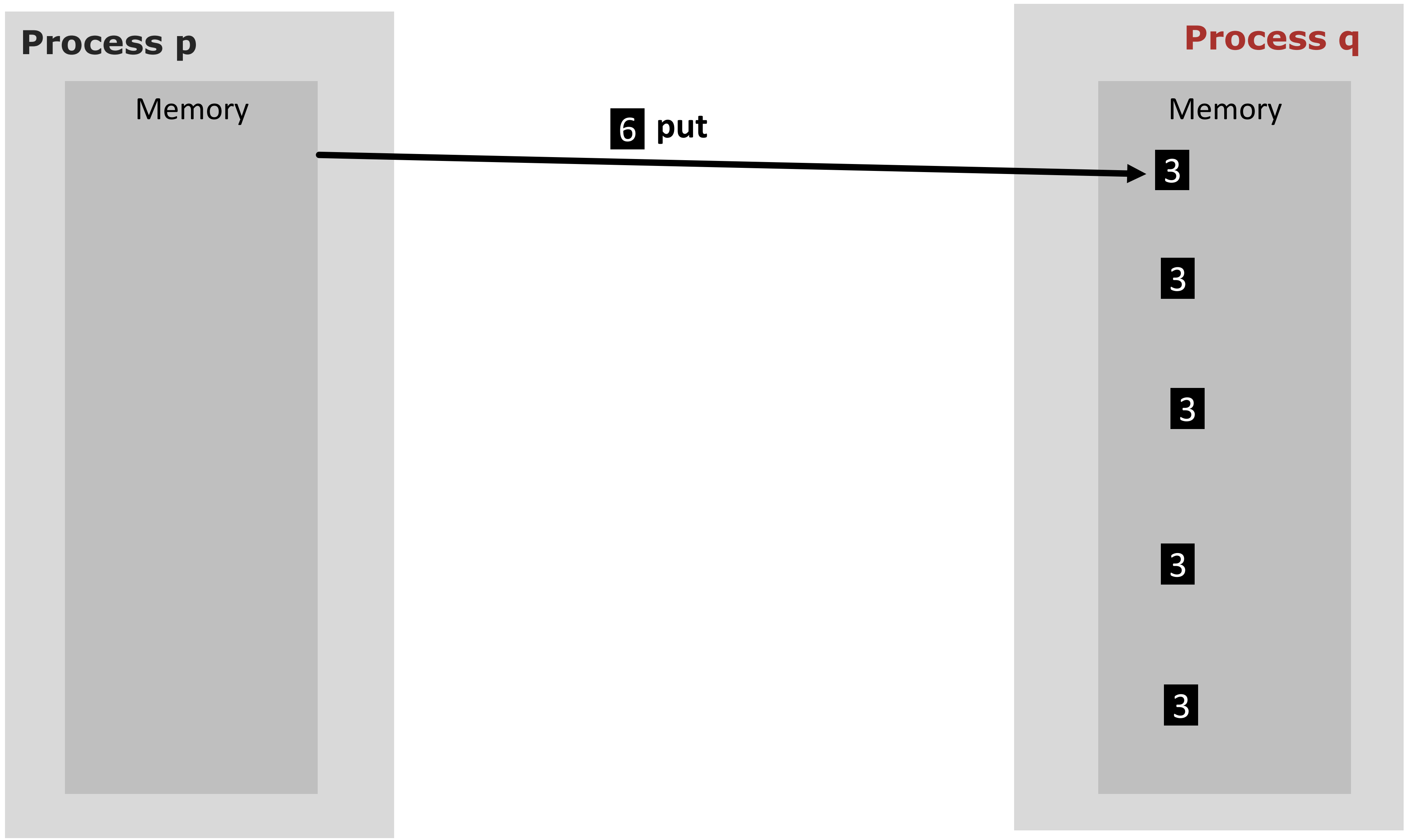
- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).



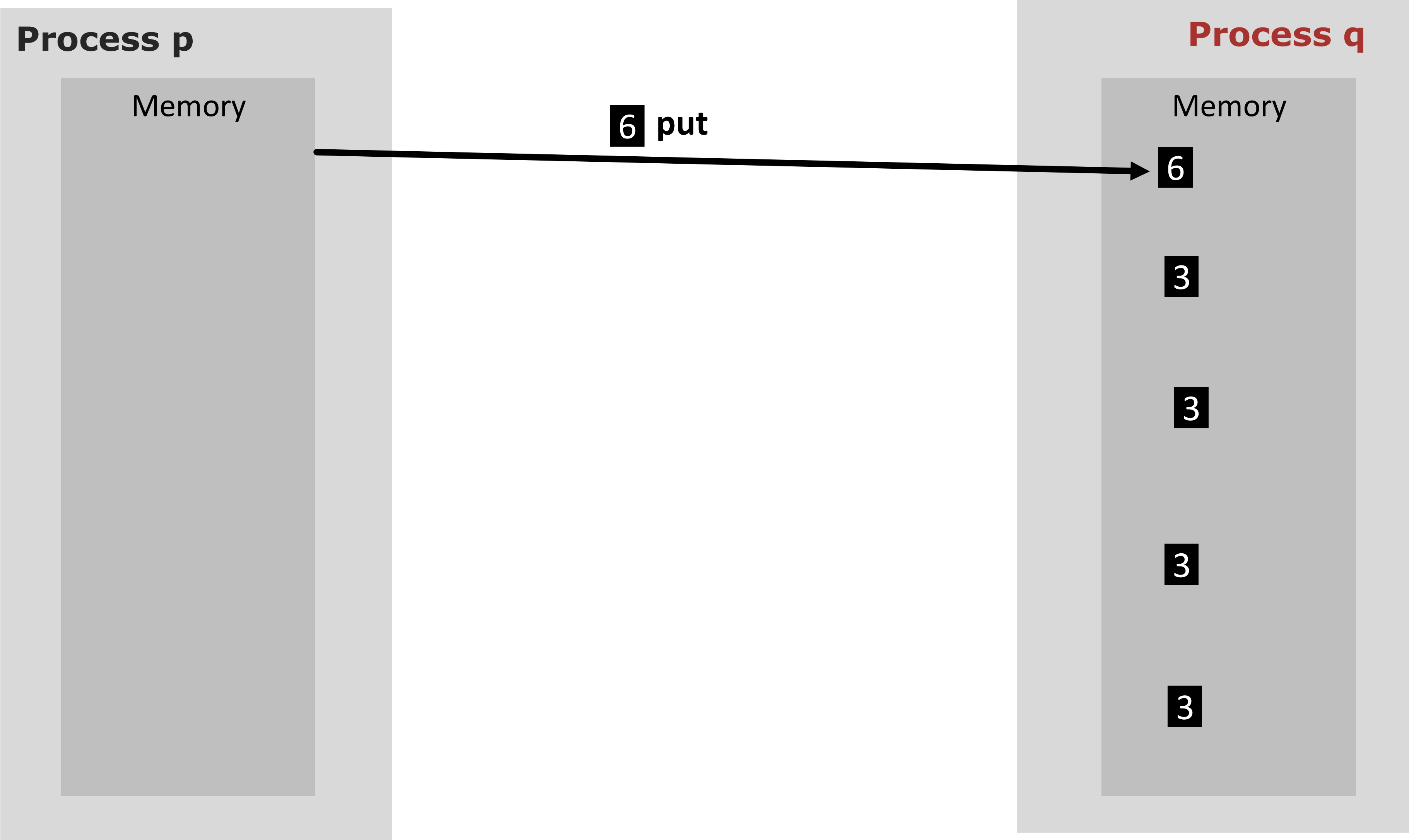
RMA-RW - Required Operations



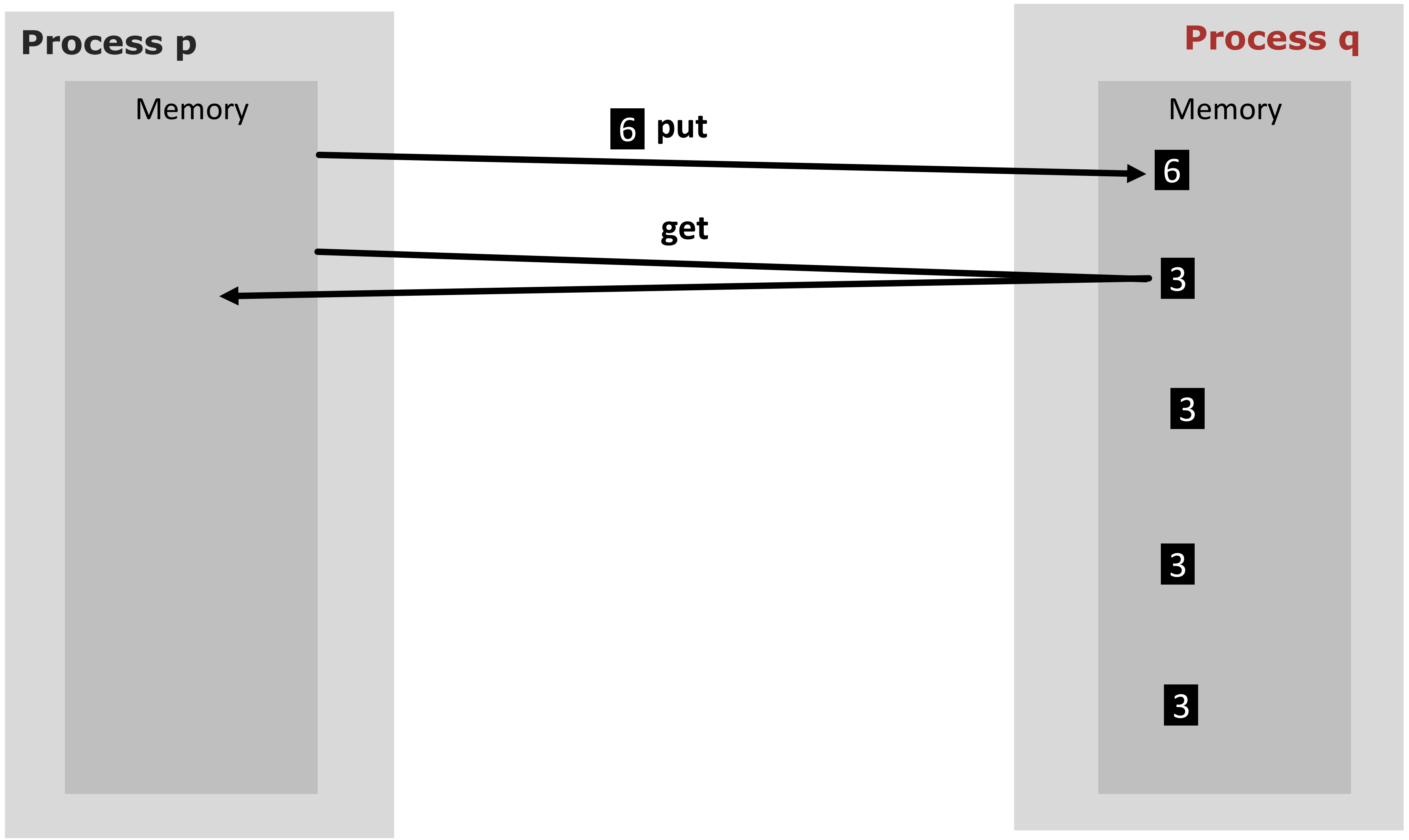
RMA-RW - Required Operations



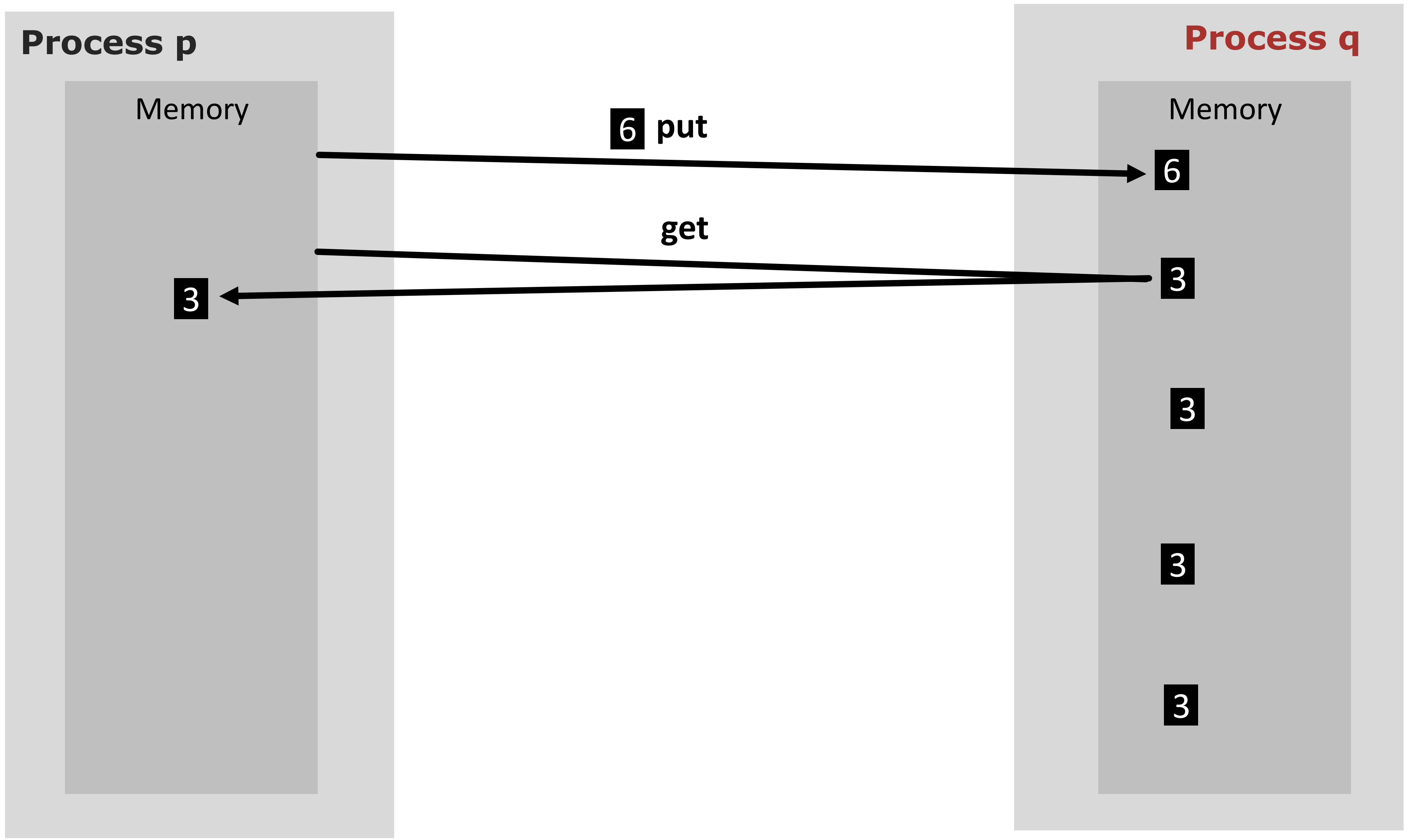
RMA-RW - Required Operations



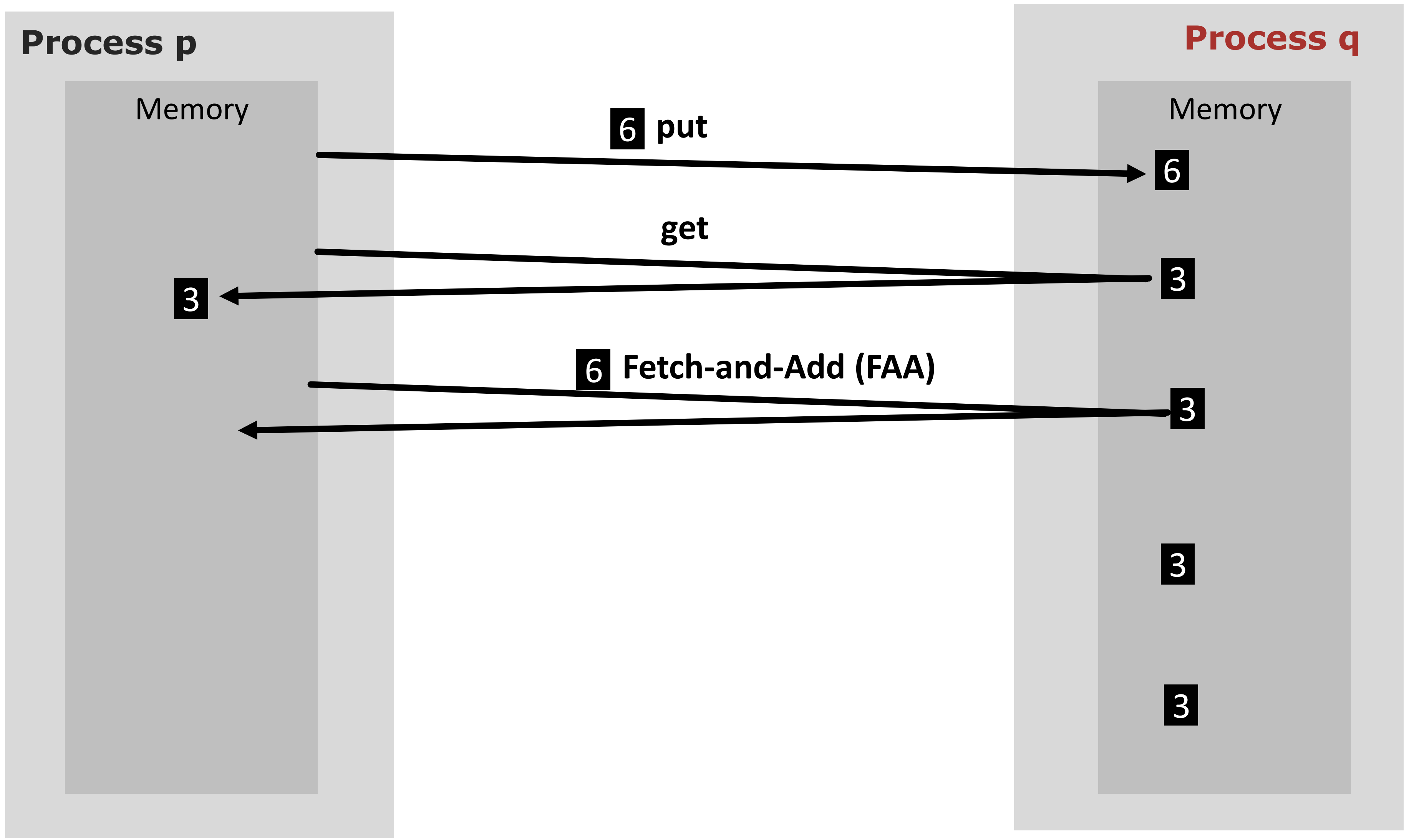
RMA-RW - Required Operations



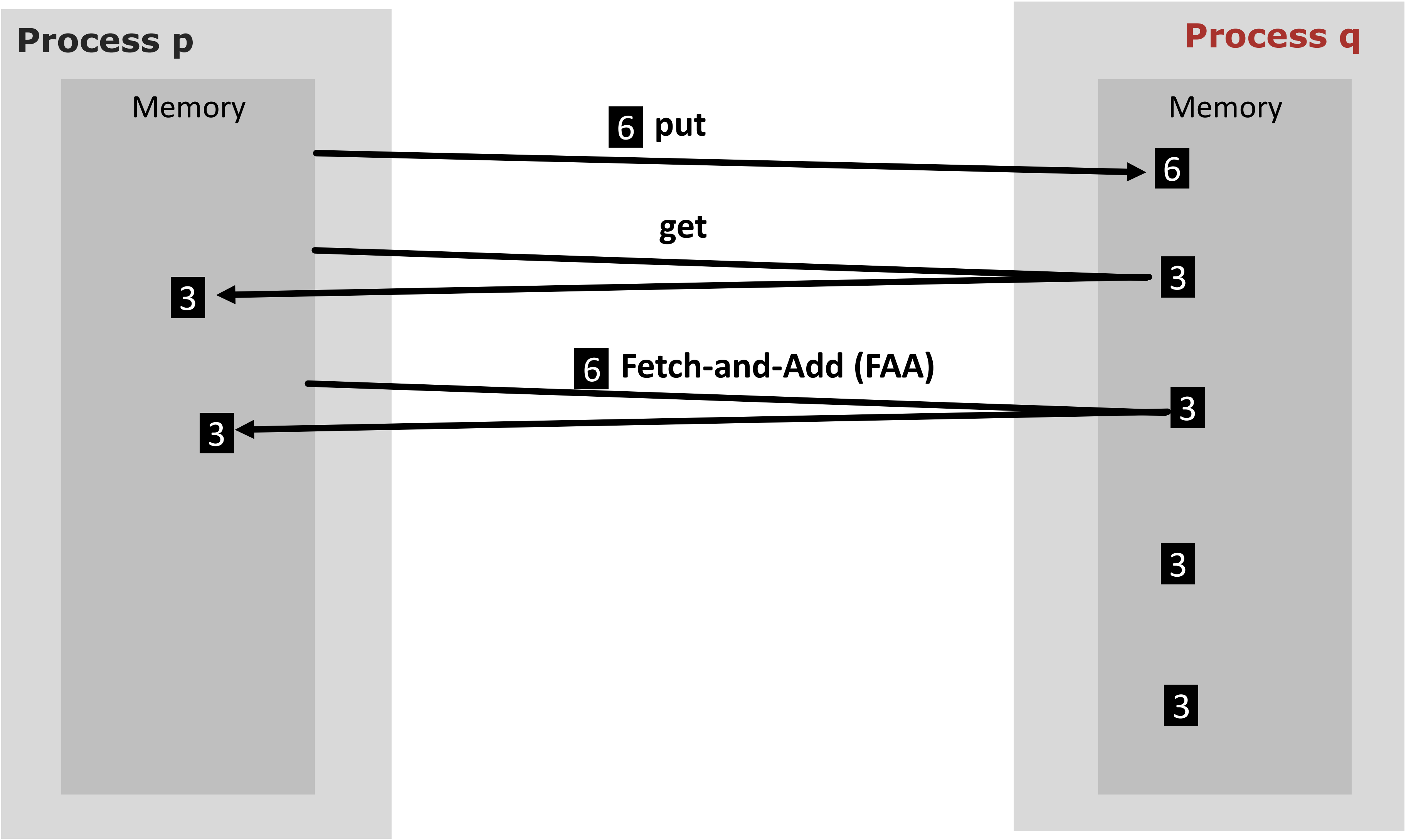
RMA-RW - Required Operations



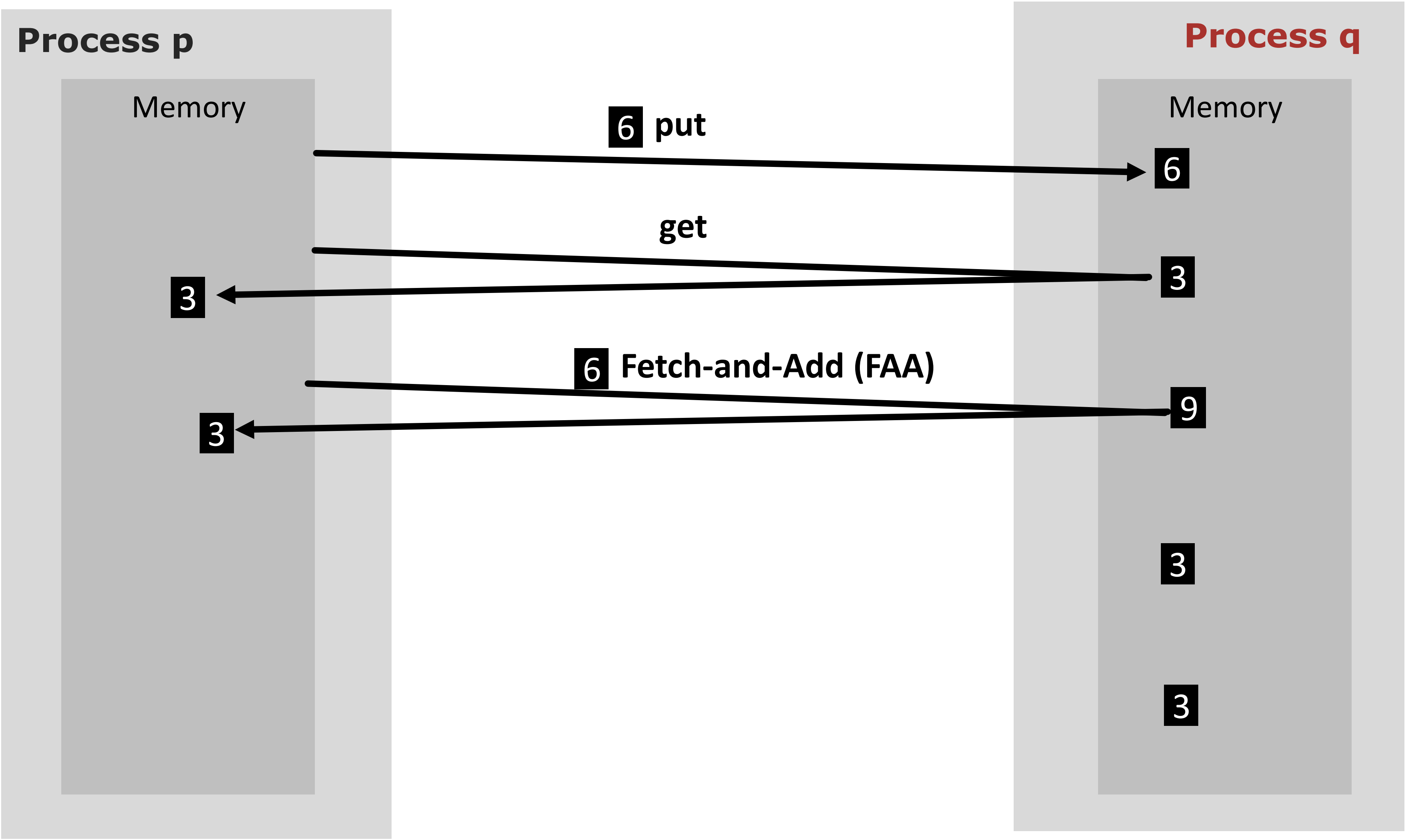
RMA-RW - Required Operations



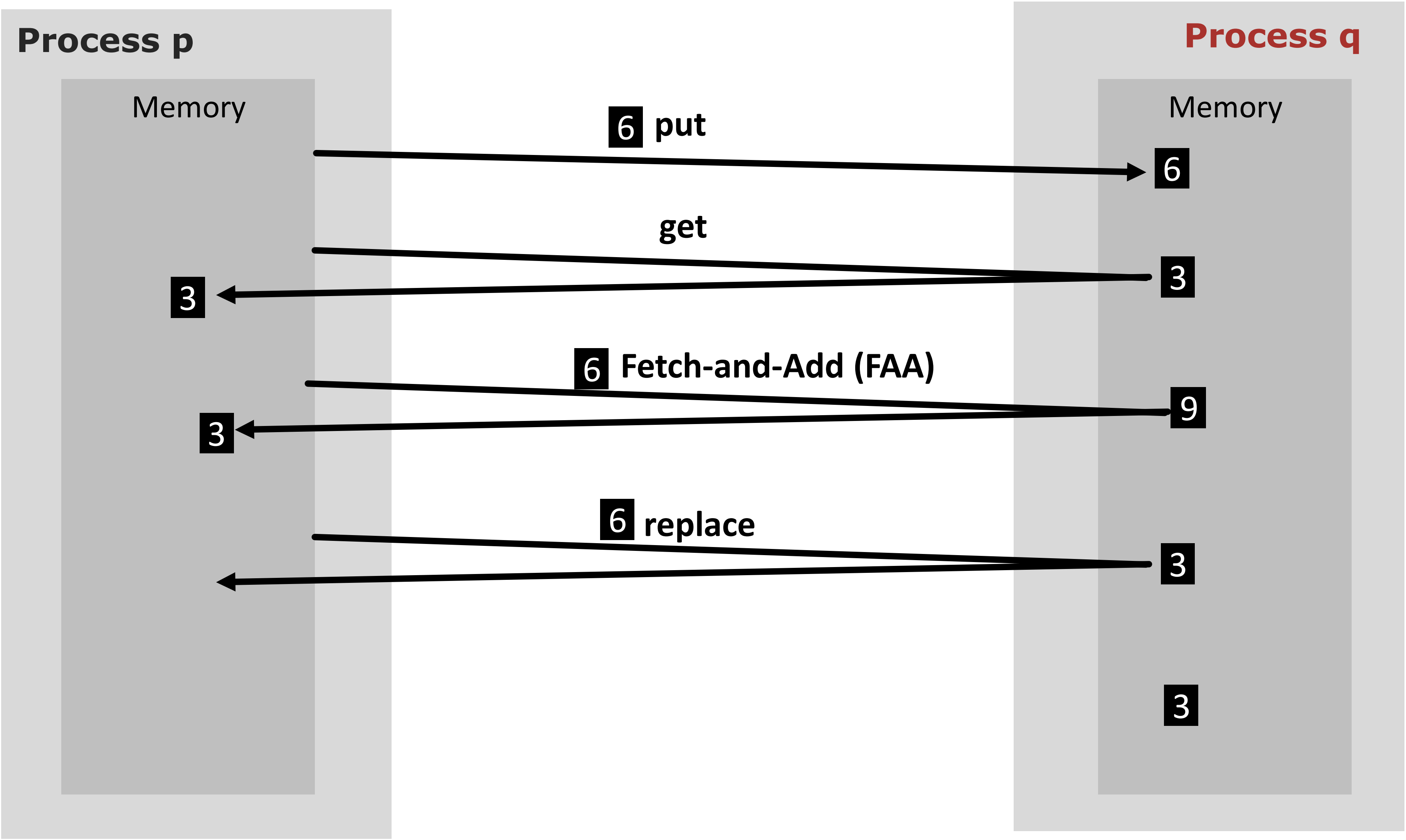
RMA-RW - Required Operations



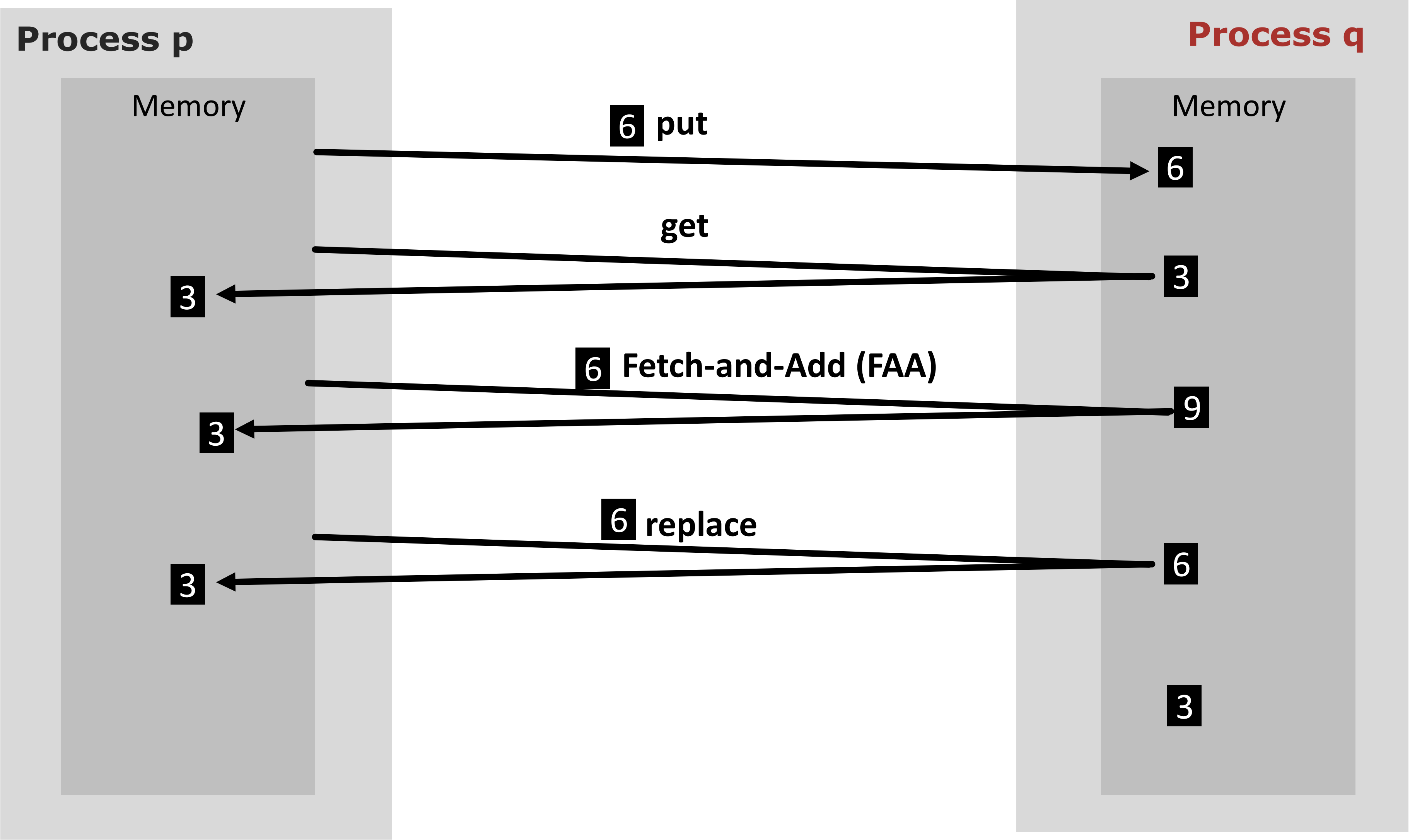
RMA-RW - Required Operations



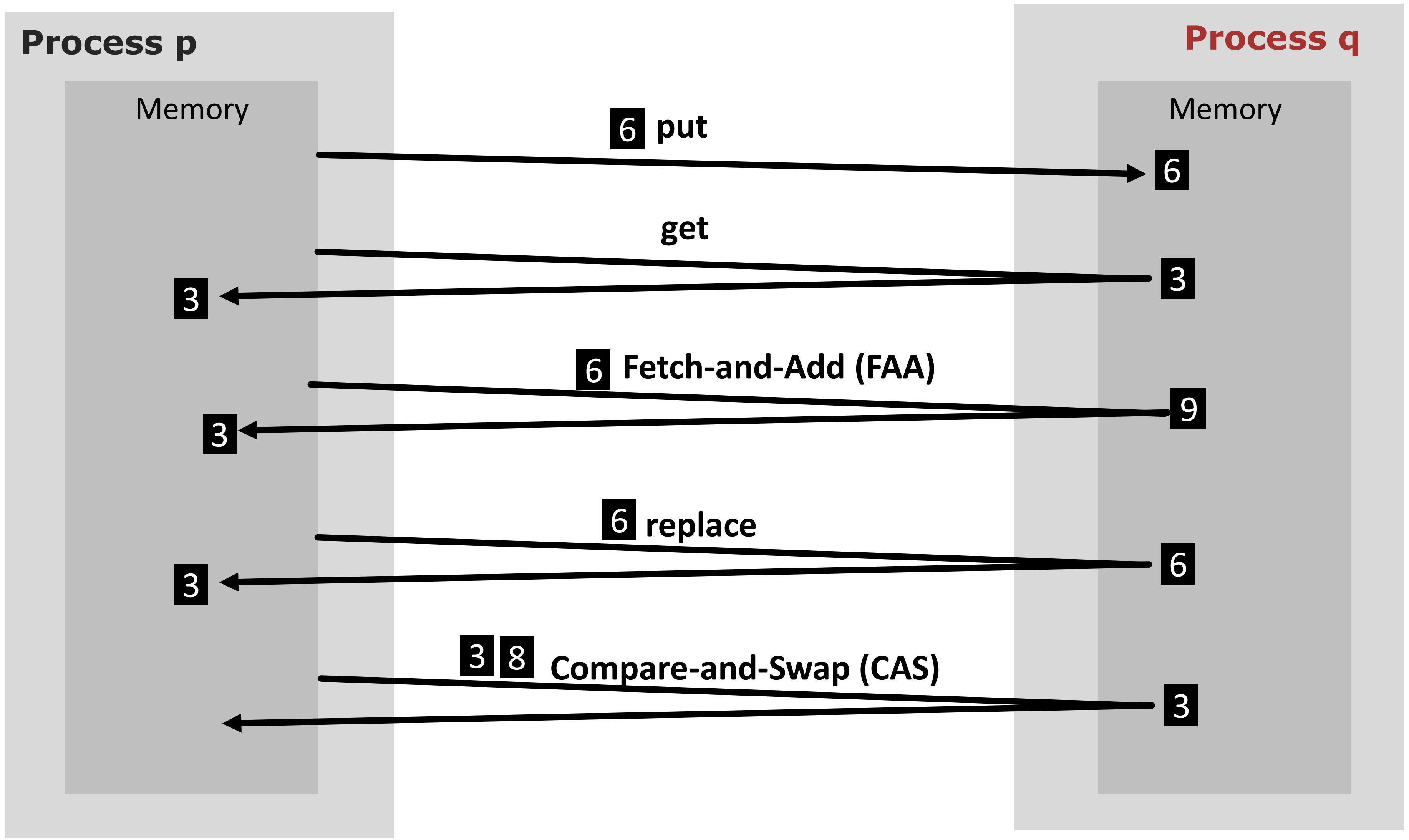
RMA-RW - Required Operations



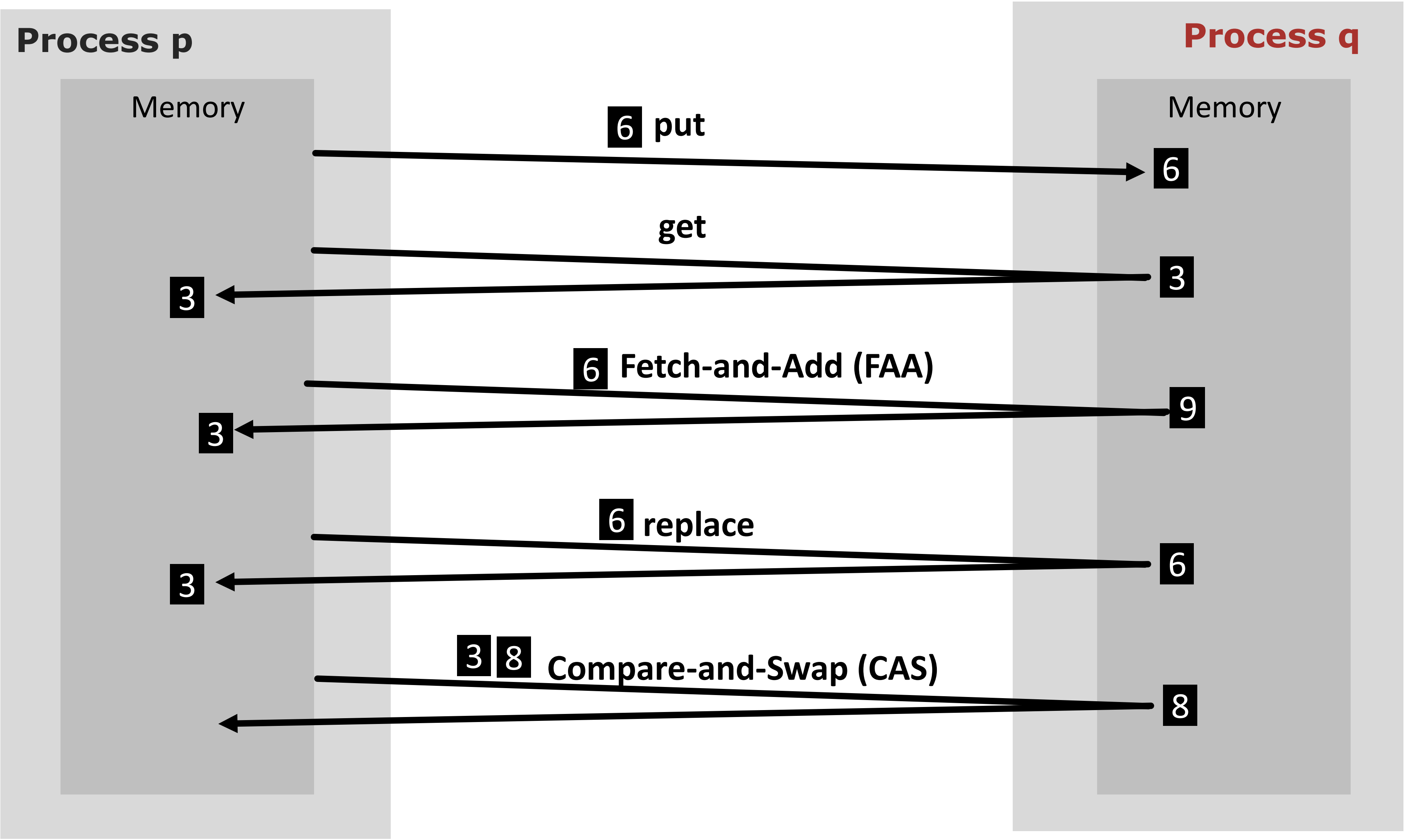
RMA-RW - Required Operations



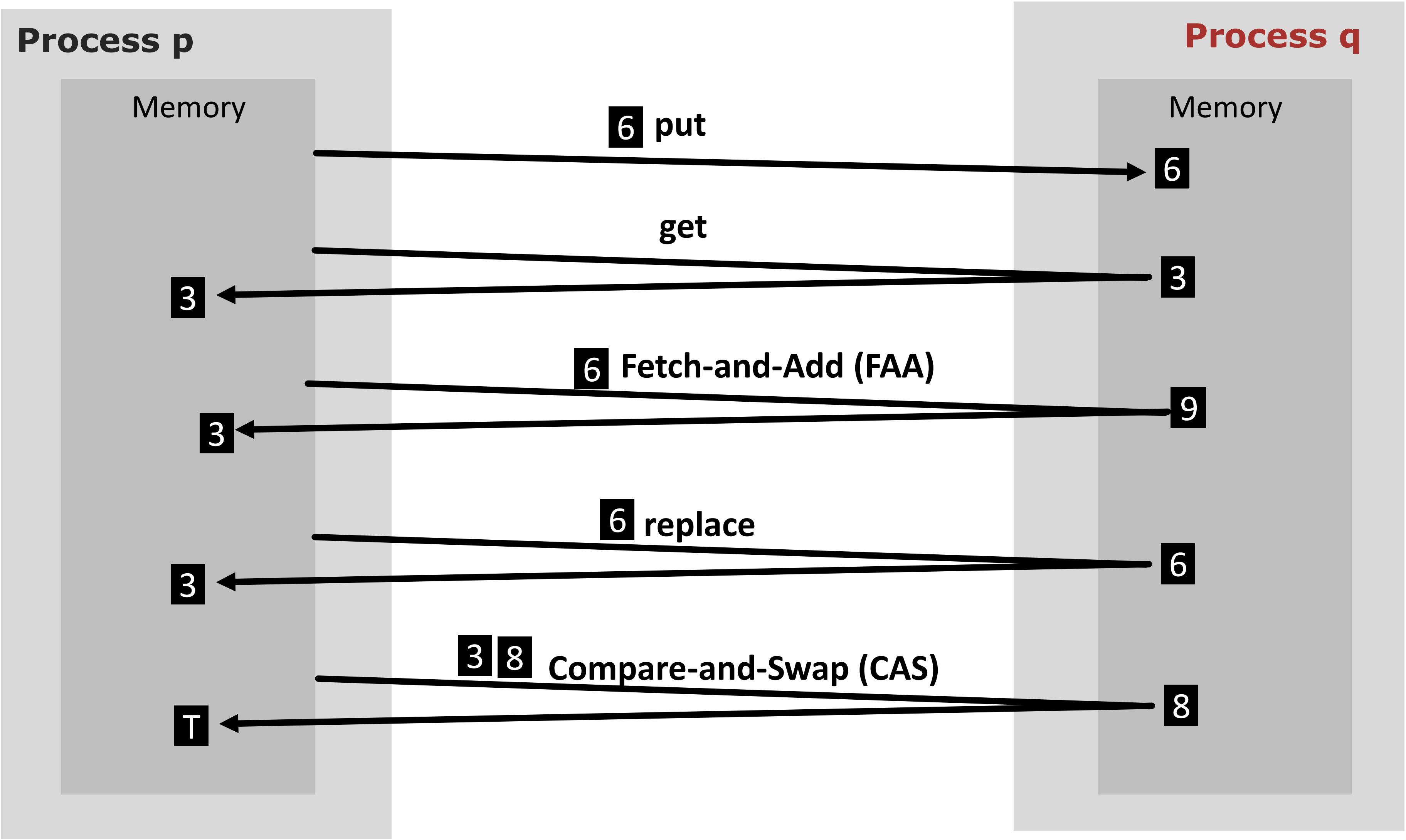
RMA-RW - Required Operations



RMA-RW - Required Operations



RMA-RW - Required Operations



MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**
  - Created explicitly
- **Remote accesses**
  - Put, get
  - Atomics

    Accumulate (also atomic Put)

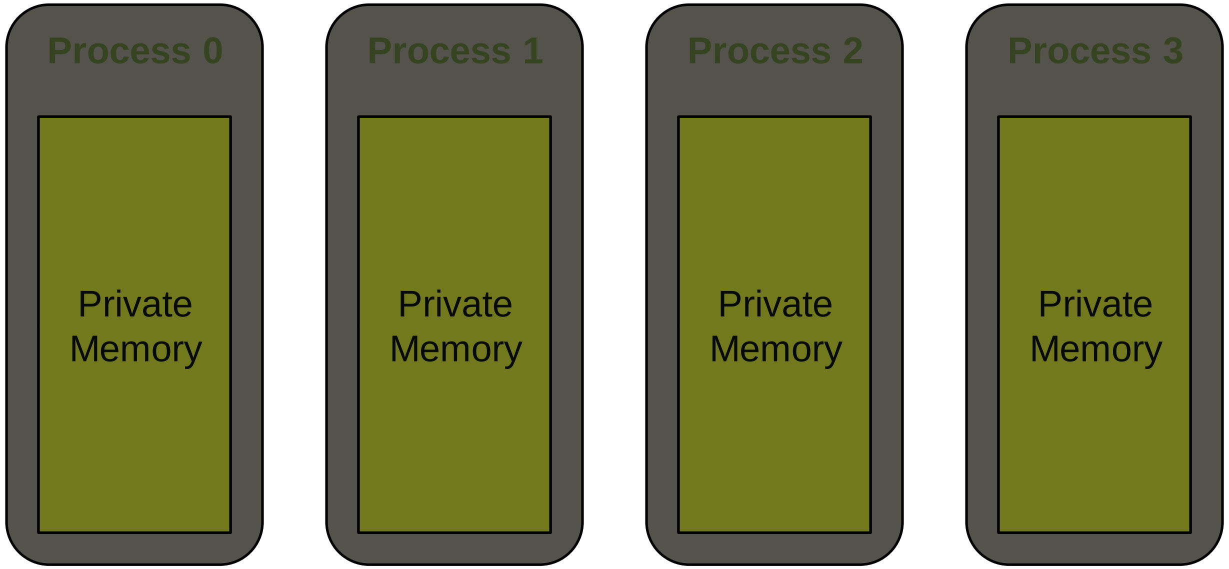
    Get_accumulate (also atomic Get)

    Fetch and op (faster single-word get_accumulate)

    Compare and swap
- **Synchronization**
  - Two modes: passive and active target

    We use passive target today, similar to shared memory!

    Synchronization: flush, flush_local
- **Memory model**
  - Unified (coherent) and separate (not coherent) view - it's complicated but versatile



MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**
  - Created explicitly
- **Remote accesses**
  - Put, get
  - Atomics

    Accumulate (also atomic Put)

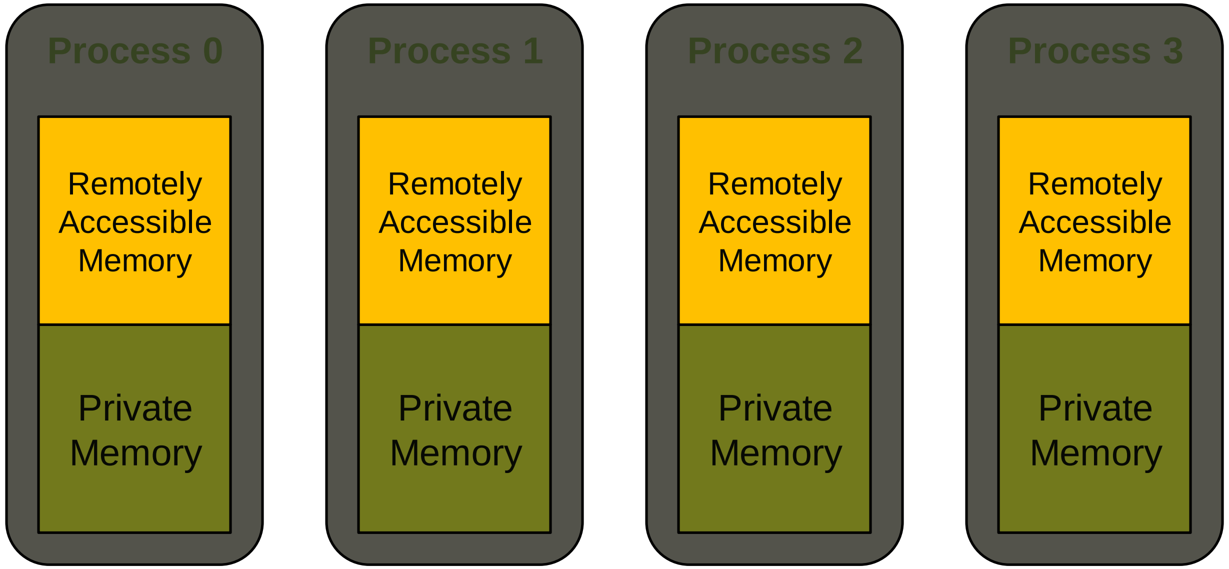
    Get_accumulate (also atomic Get)

    Fetch and op (faster single-word get_accumulate)

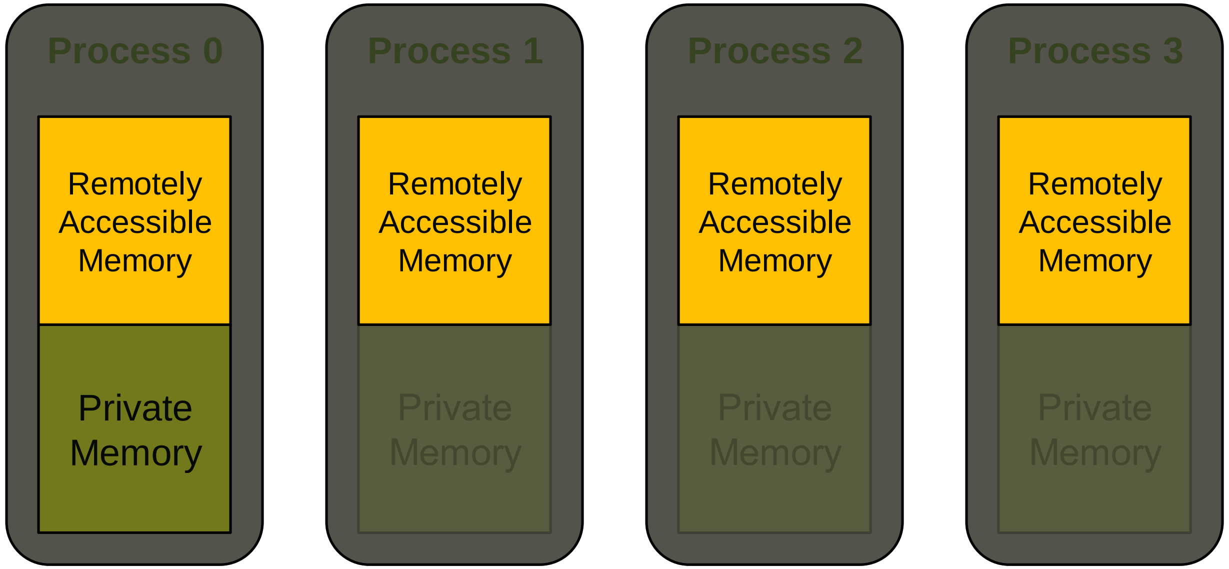
    Compare and swap
- **Synchronization**
  - Two modes: passive and active target

    We use passive target today, similar to shared memory!

    Synchronization: flush, flush_local
- **Memory model**
  - Unified (coherent) and separate (not coherent) view - it's complicated but versatile



MPI RMA primer ([much] more in the recitation sessions)



- **Windows expose memory**
  - Created explicitly
- **Remote accesses**
  - Put, get
  - Atomics

    Accumulate (also atomic Put)

    Get_accumulate (also atomic Get)

    Fetch and op (faster single-word get_accumulate)

    Compare and swap
- **Synchronization**
  - Two modes: passive and active target

    We use passive target today, similar to shared memory!

    Synchronization: flush, flush_local
- **Memory model**
  - Unified (coherent) and separate (not coherent) view - it’s complicated but versatile

MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**
  - Created explicitly
- **Remote accesses**
  - Put, get
  - Atomics

    Accumulate (also atomic Put)

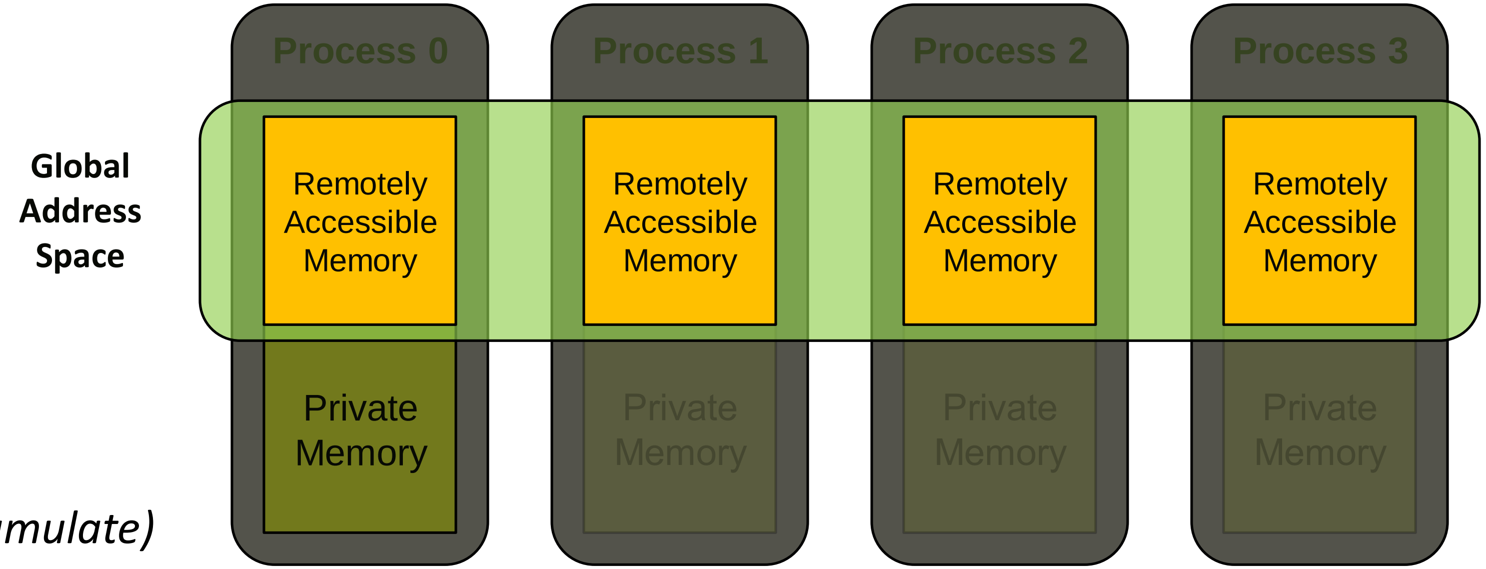
    Get_accumulate (also atomic Get)

    Fetch and op (faster single-word get_accumulate)

    Compare and swap
- **Synchronization**
  - Two modes: passive and active target

    We use passive target today, similar to shared memory!

    Synchronization: flush, flush_local
- **Memory model**
  - Unified (coherent) and separate (not coherent) view - it's complicated but versatile



MPI RMA primer ([much] more in the recitation sessions)

- **Windows expose memory**
  - Created explicitly
- **Remote accesses**
  - Put, get
  - Atomics

    Accumulate (also atomic Put)

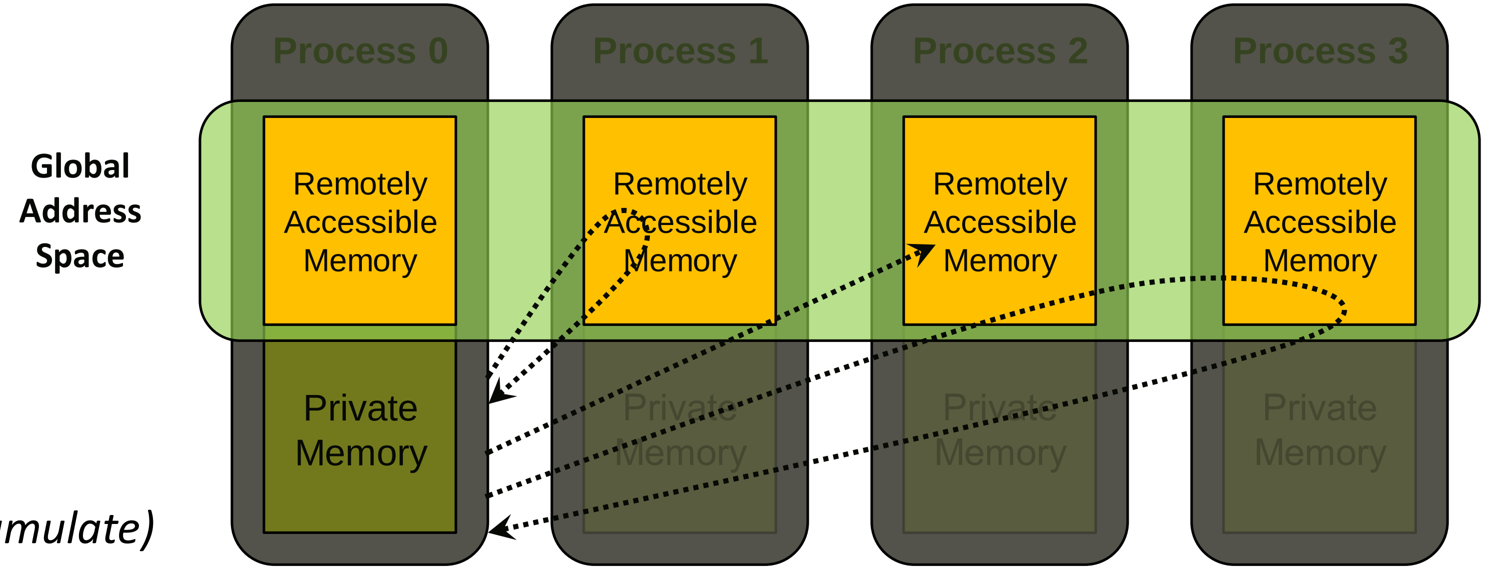
    Get_accumulate (also atomic Get)

    Fetch and op (faster single-word get_accumulate)

    Compare and swap
- **Synchronization**
  - Two modes: passive and active target

    We use passive target today, similar to shared memory!

    Synchronization: flush, flush_local
- **Memory model**
  - Unified (coherent) and separate (not coherent) view - it's complicated but versatile



How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?

How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?

How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?

How to manage the design complexity?

How to manage the design complexity?

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Modular design

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper

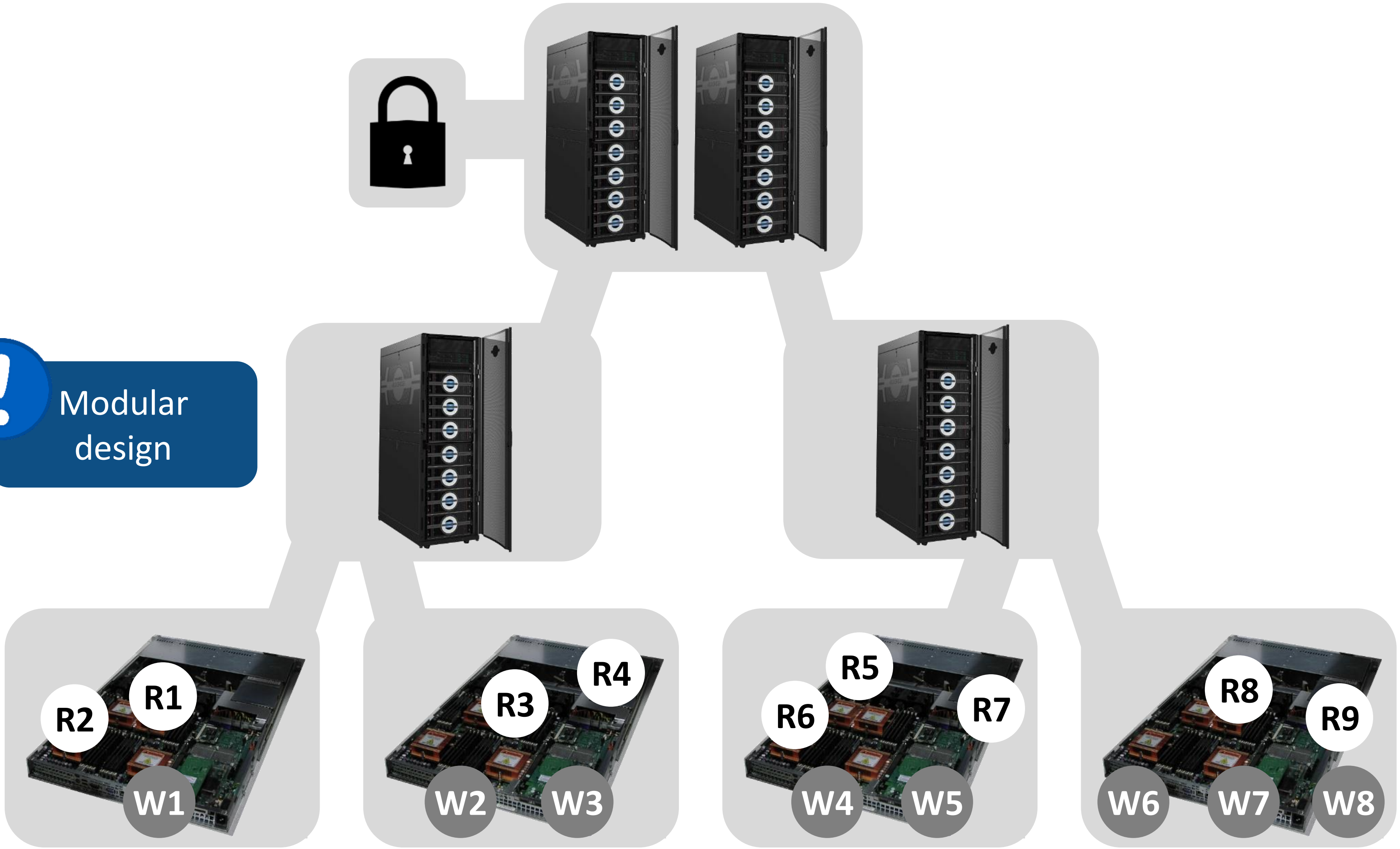
How to manage the design complexity?

Modular design

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Modular design

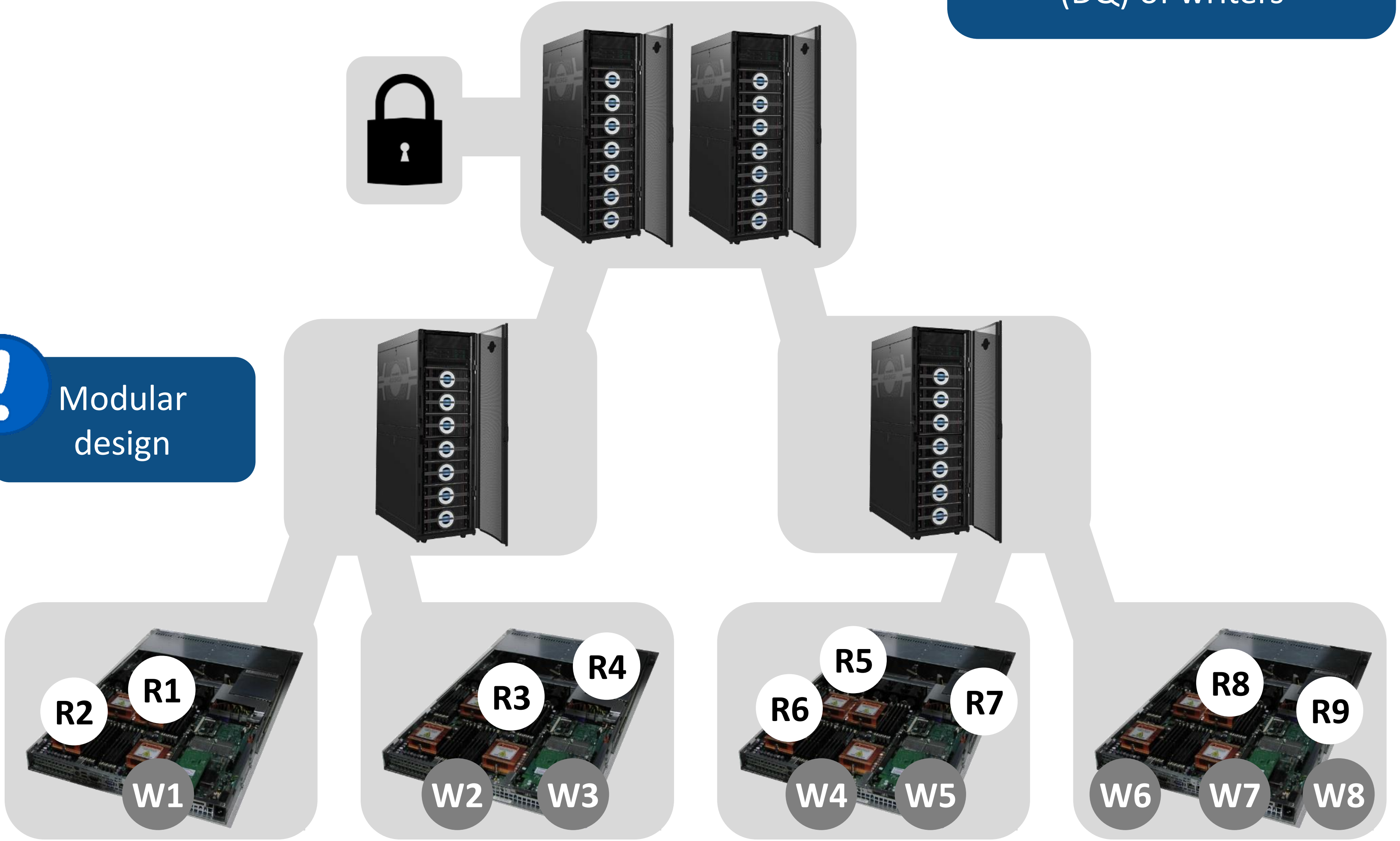


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Modular design

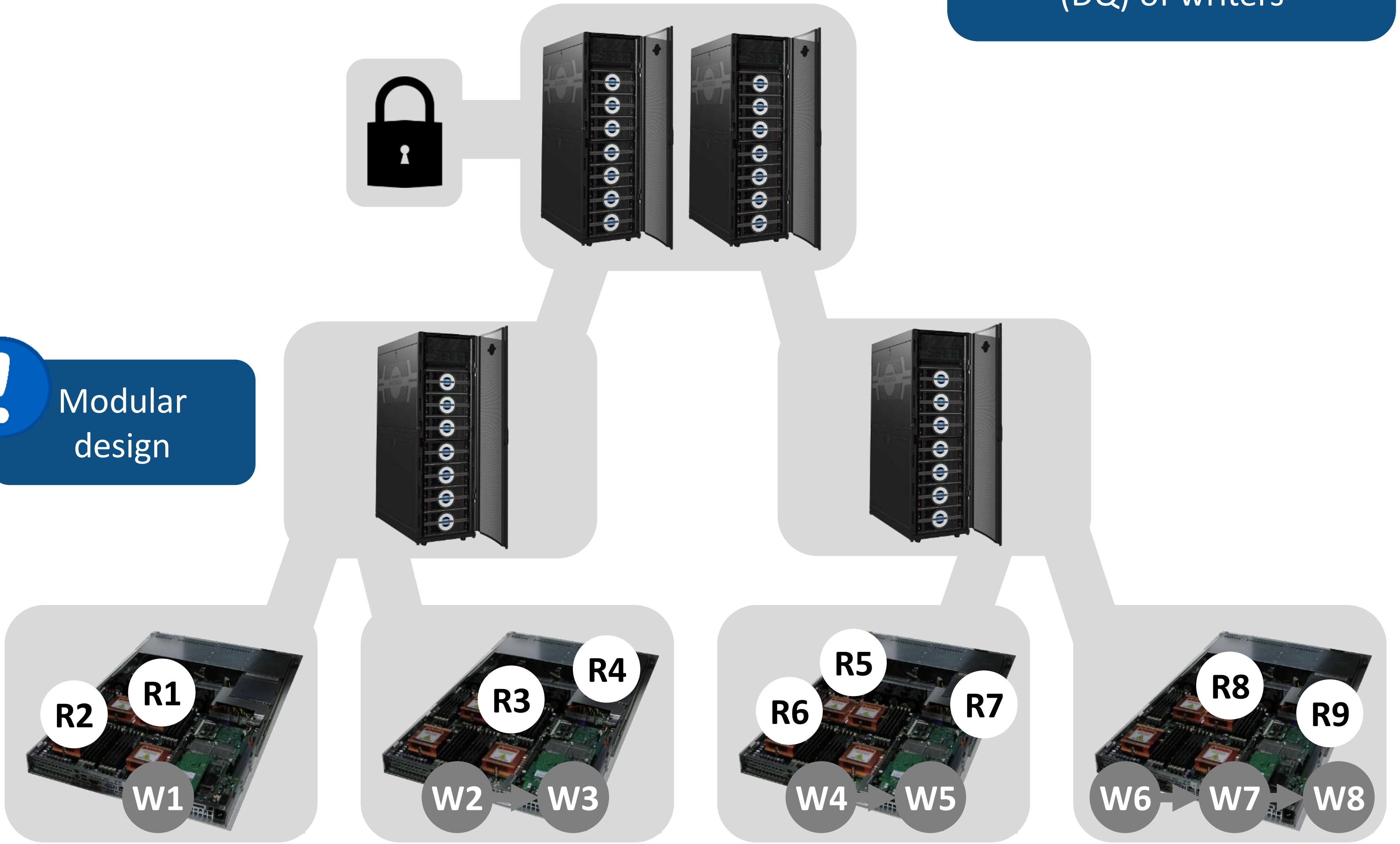


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Modular design

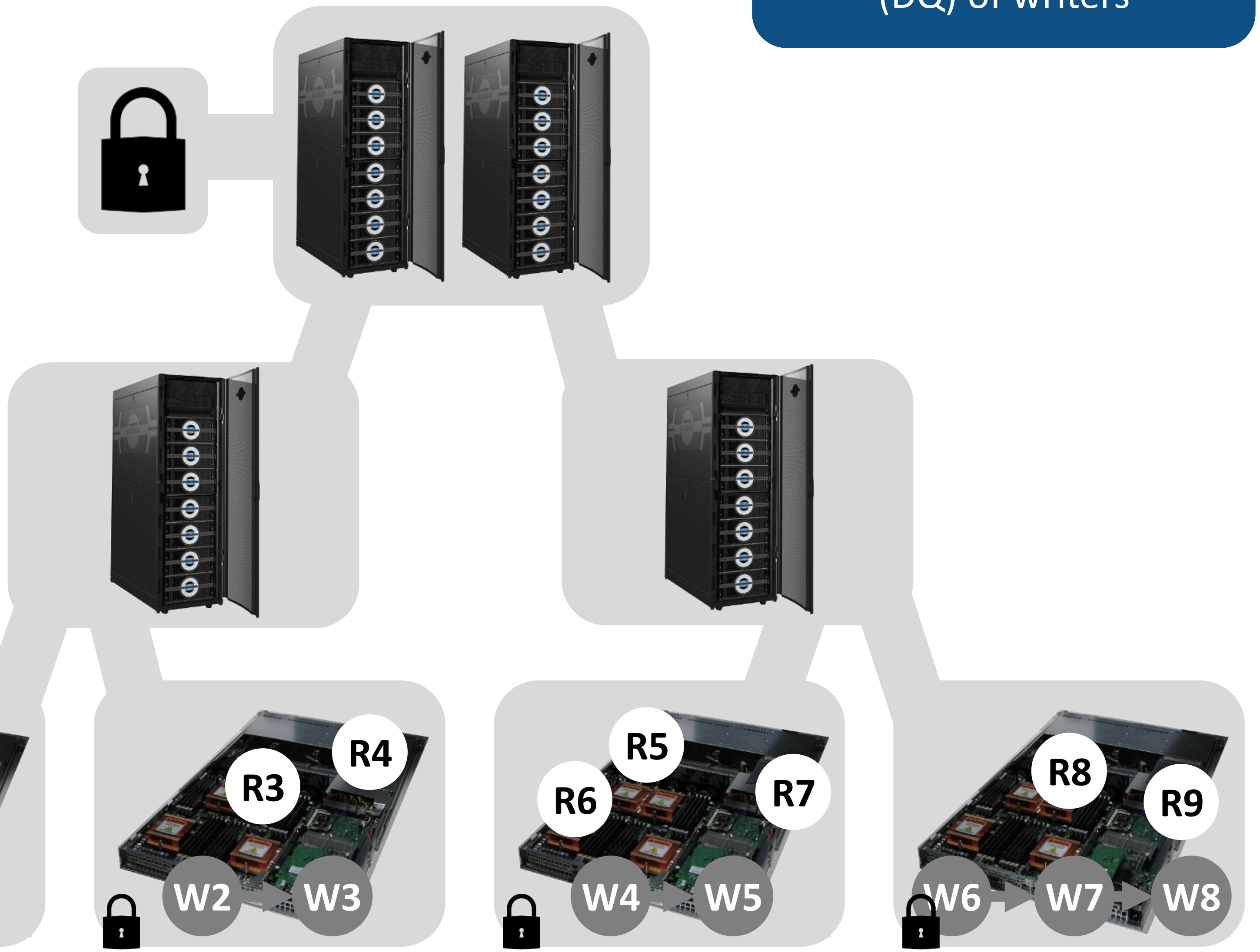


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Modular design

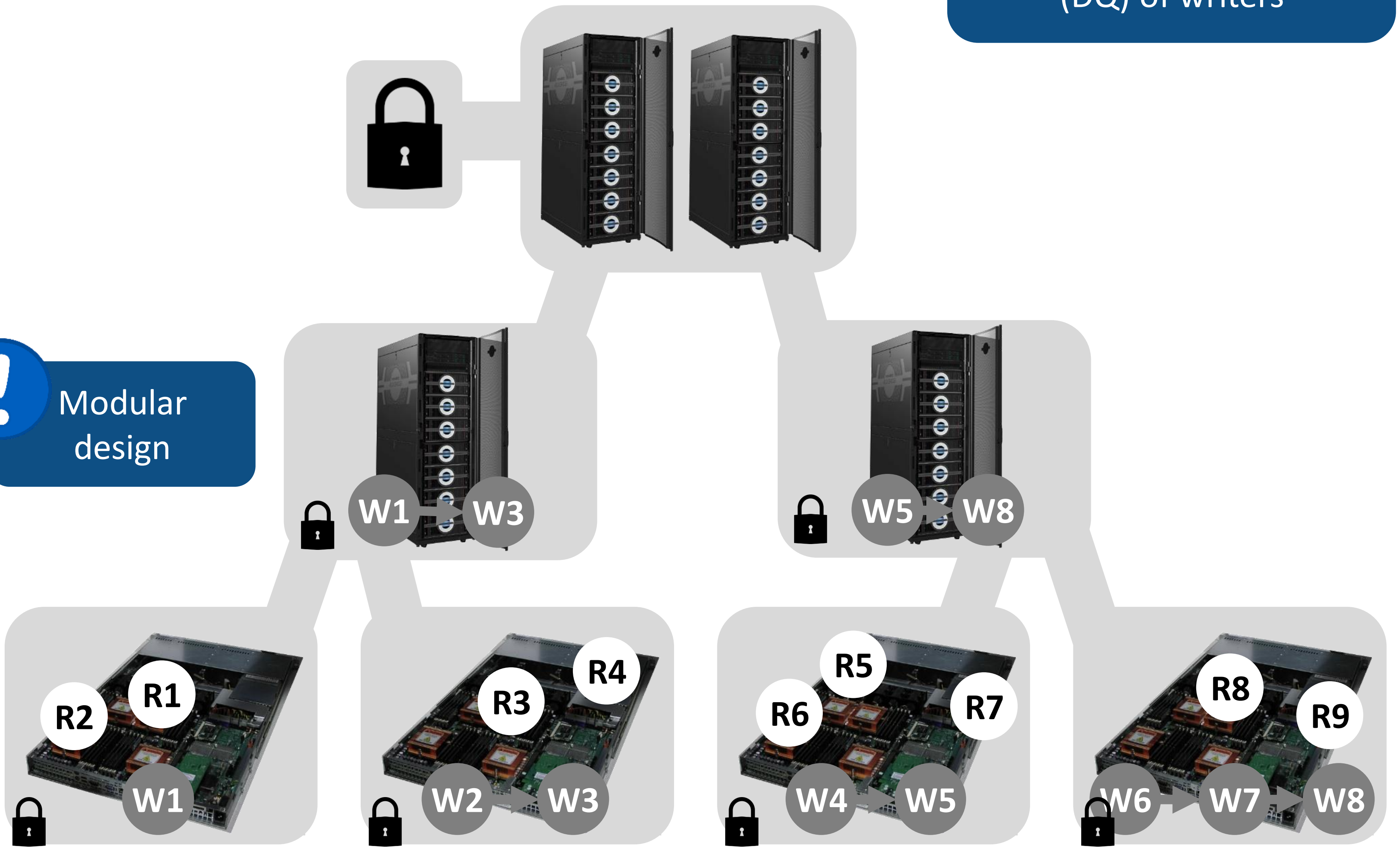


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Modular design

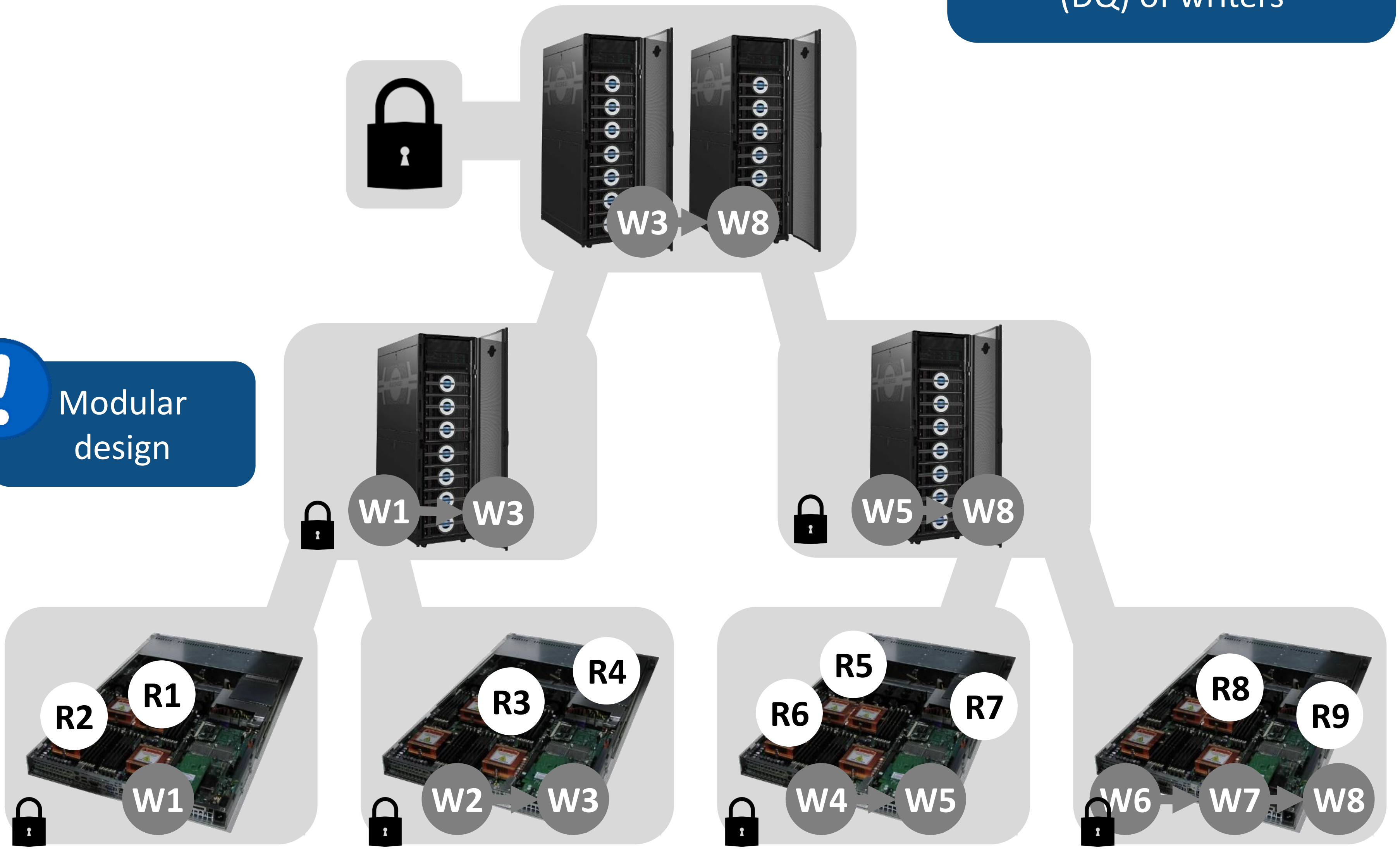


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Modular design



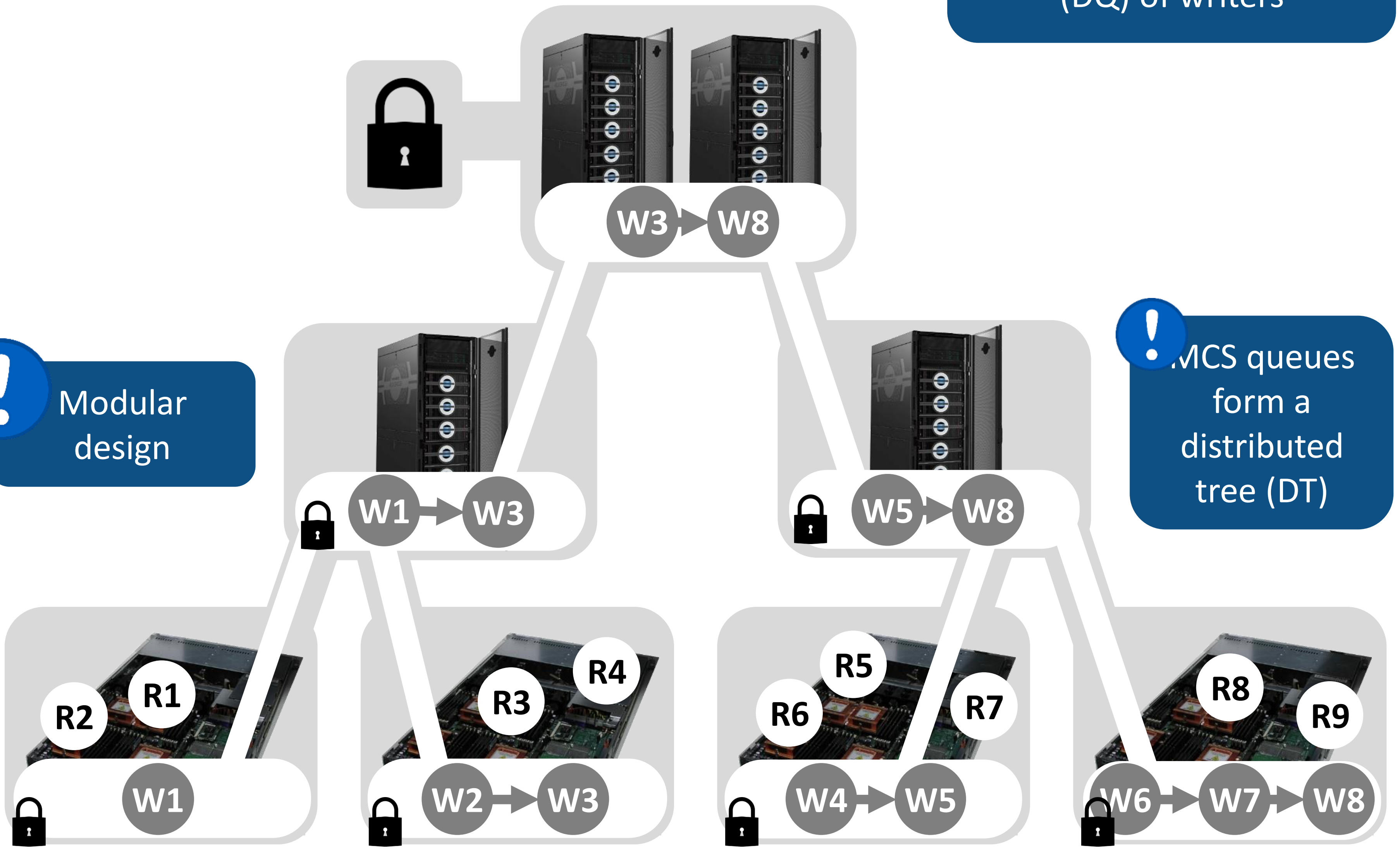
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Modular design

MCS queues form a distributed tree (DT)



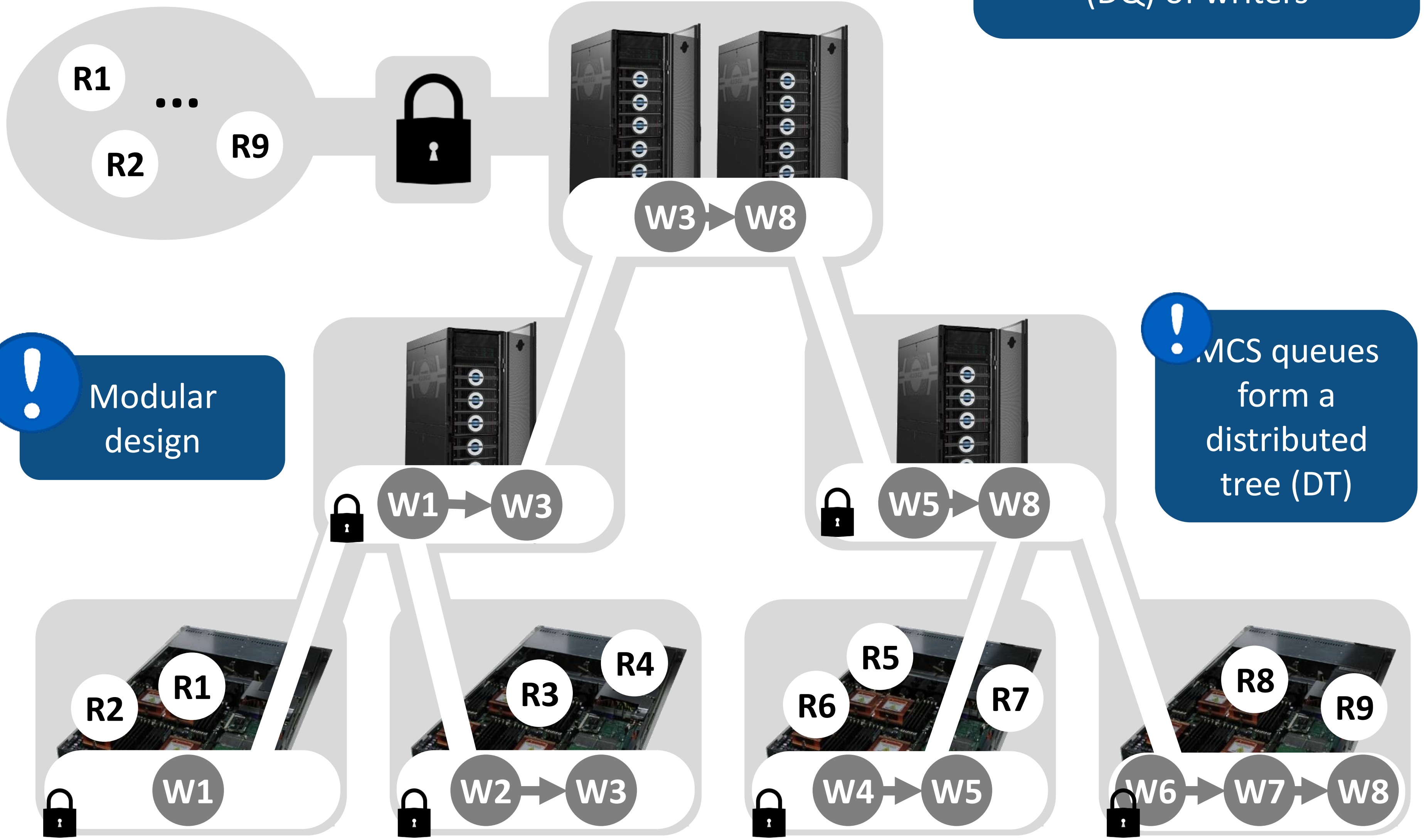
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Modular design

MCS queues form a distributed tree (DT)



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

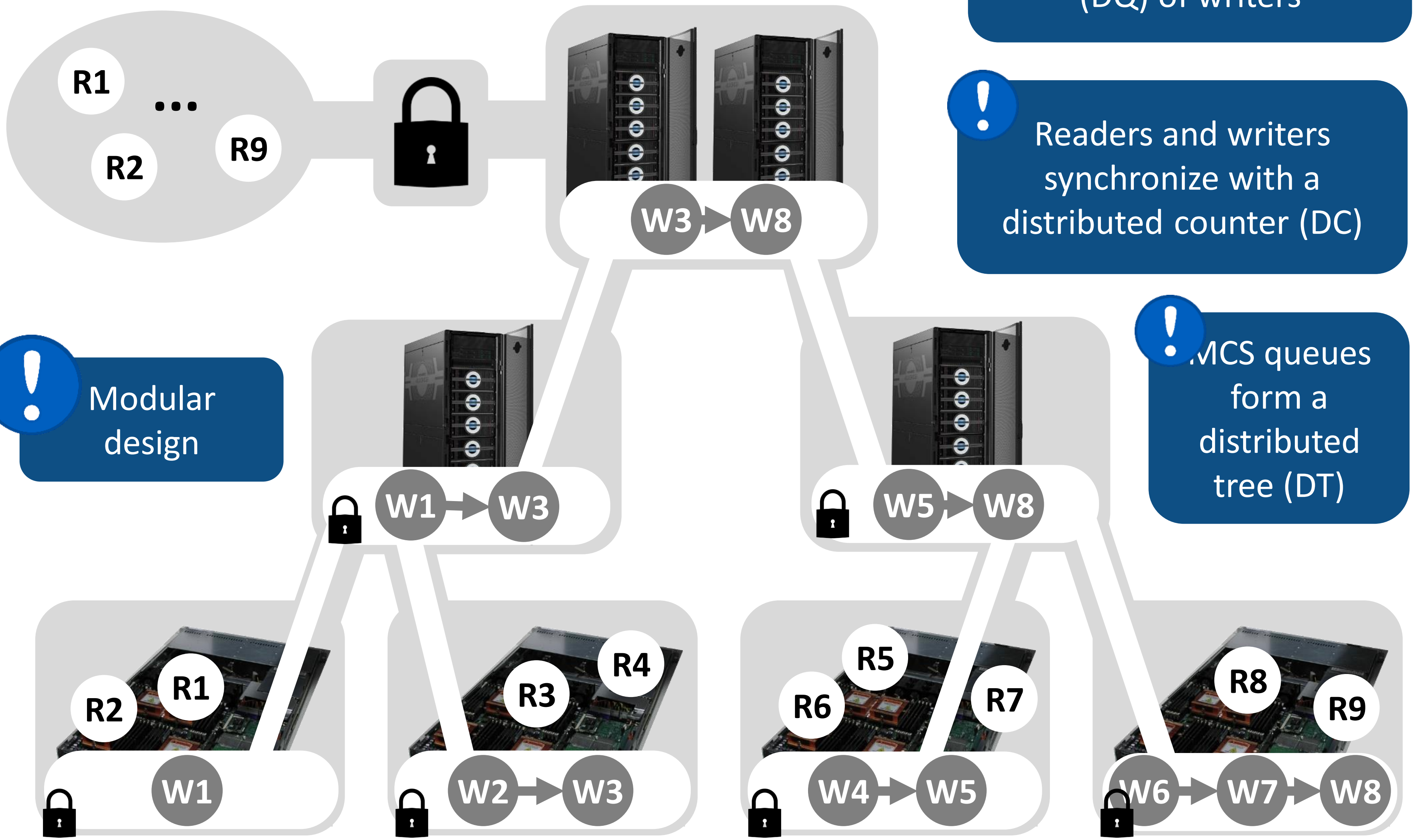
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Readers and writers synchronize with a distributed counter (DC)

MCS queues form a distributed tree (DT)

Modular design



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

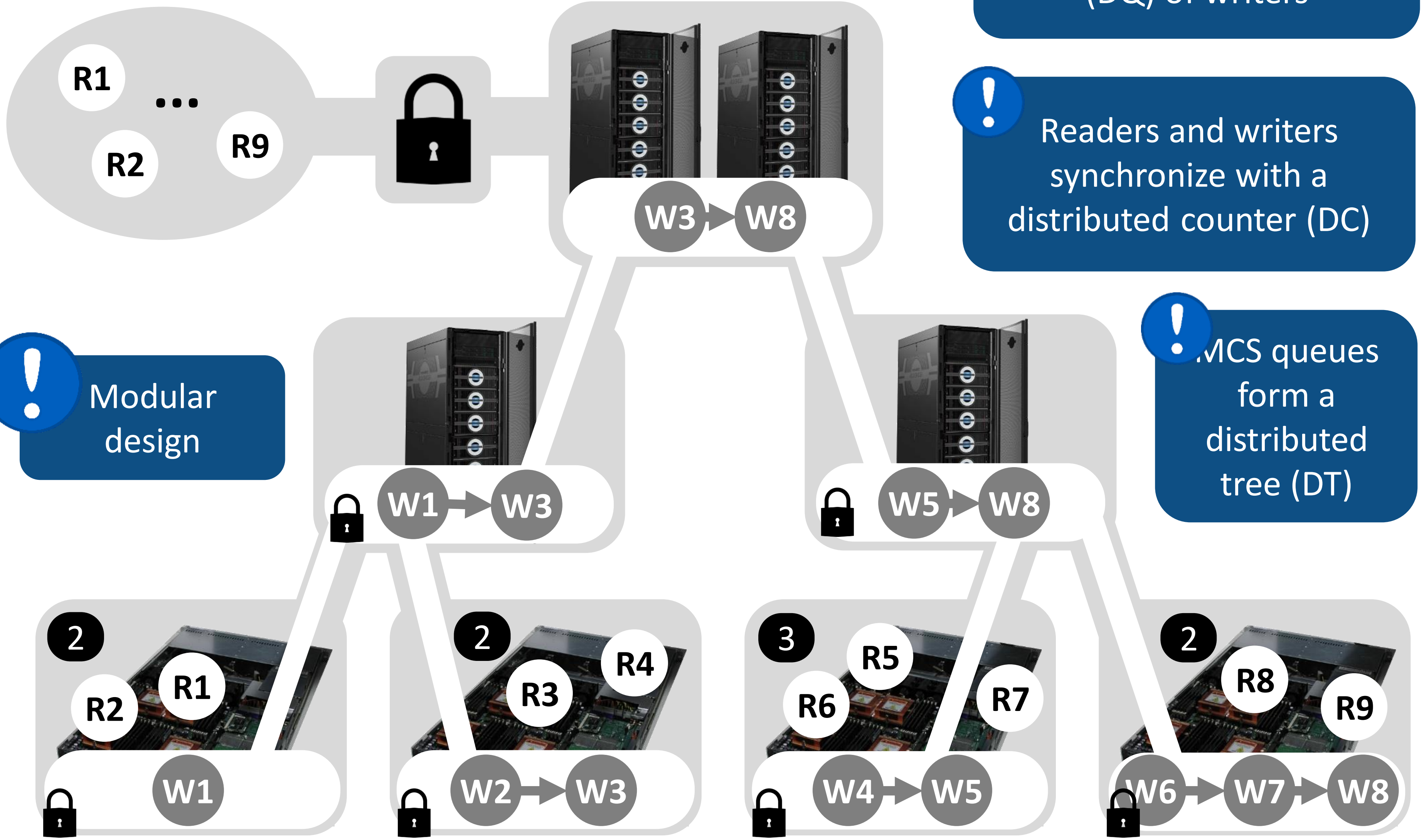
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Readers and writers synchronize with a distributed counter (DC)

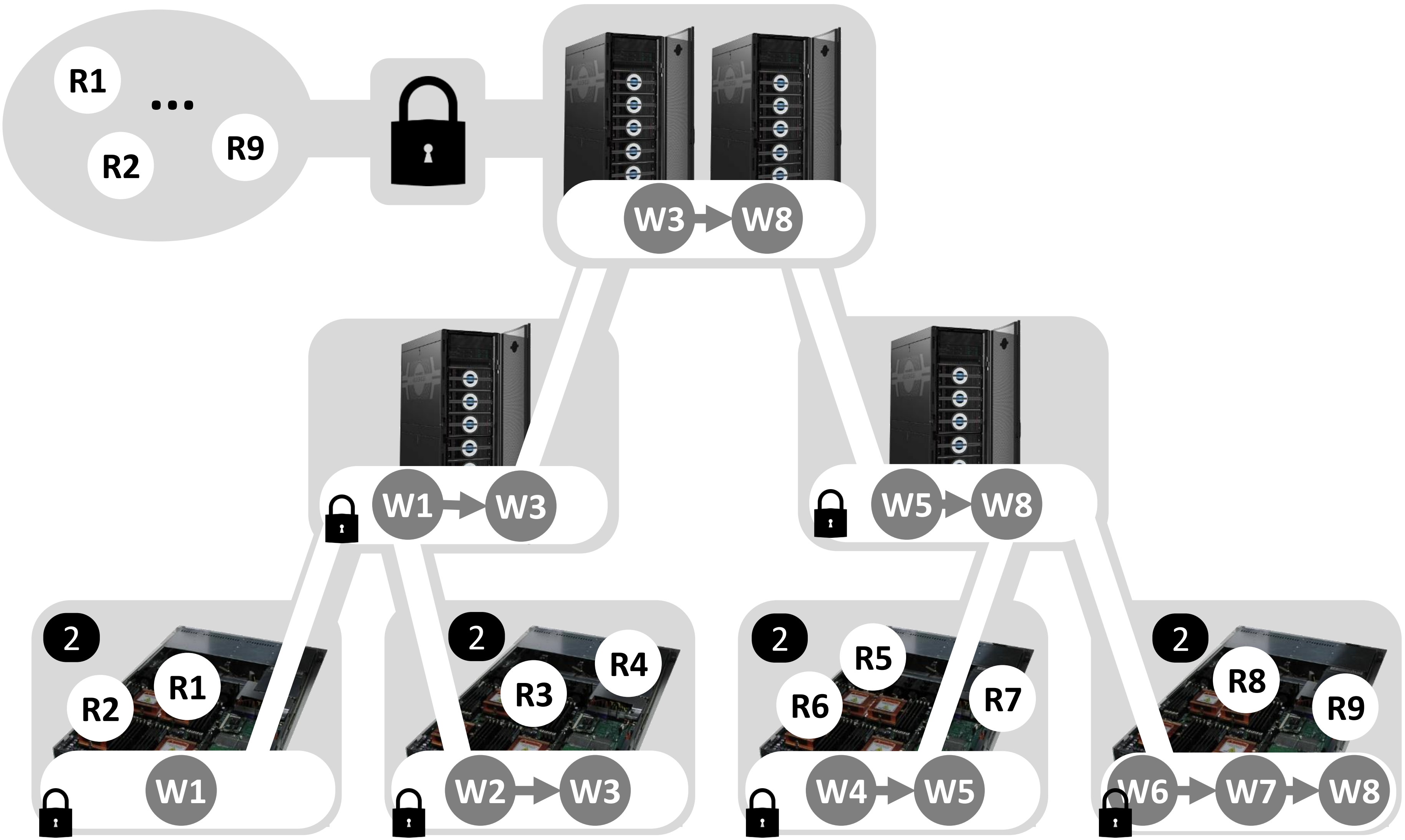
MCS queues form a distributed tree (DT)

Modular design



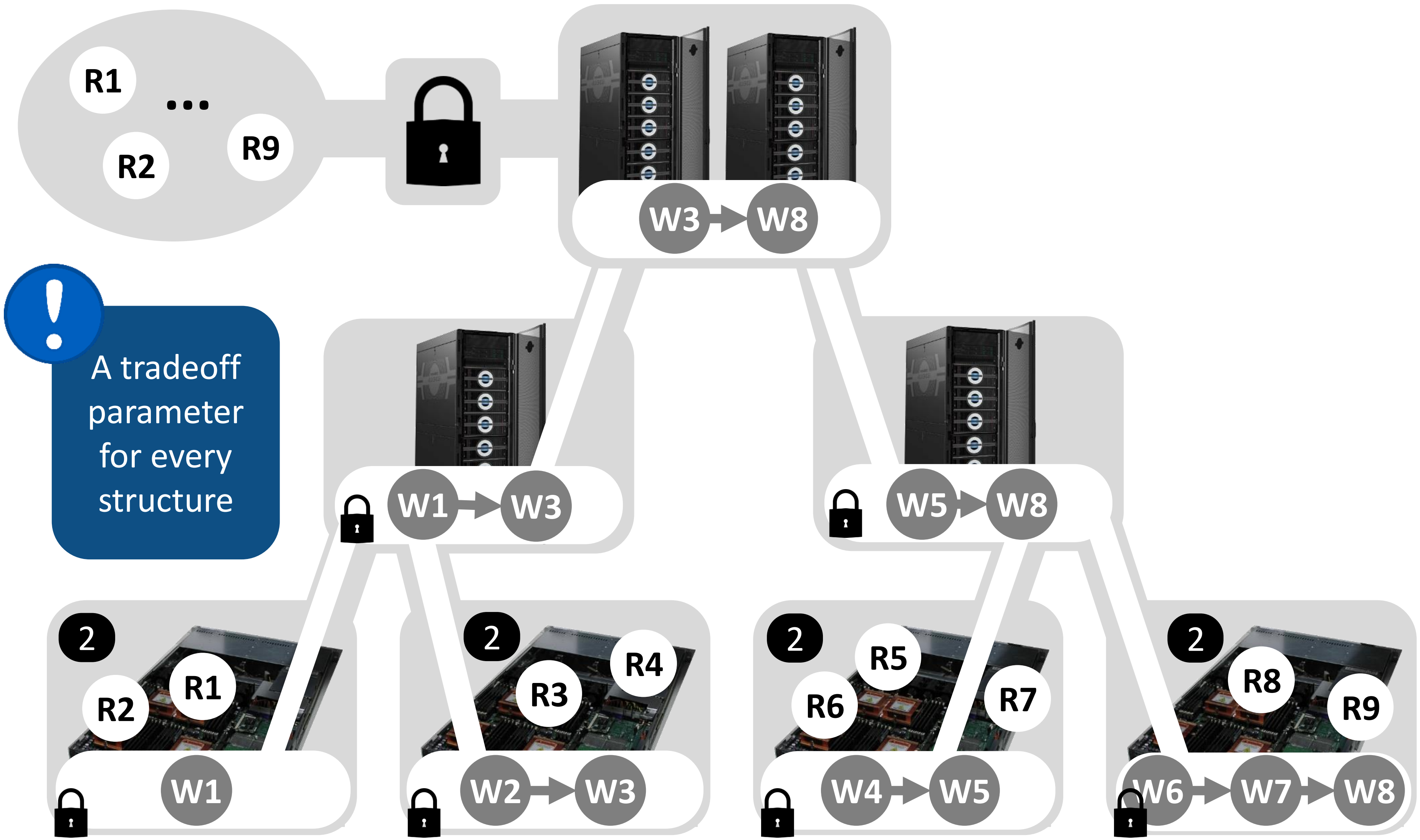
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to ensure tunable performance?



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to ensure tunable performance?



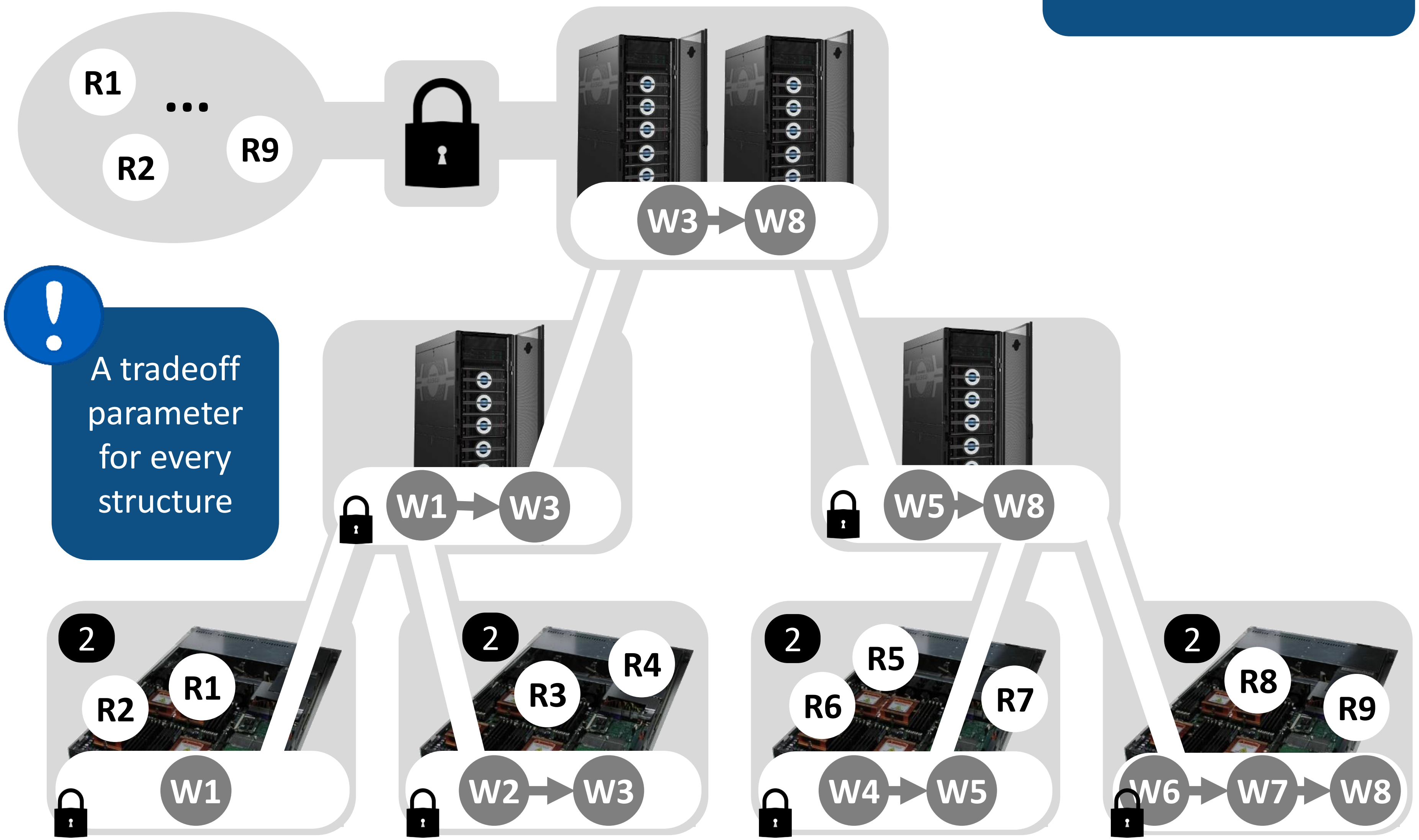
A tradeoff parameter for every structure

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to ensure tunable performance?

Each DQ: fairness vs throughput of writers

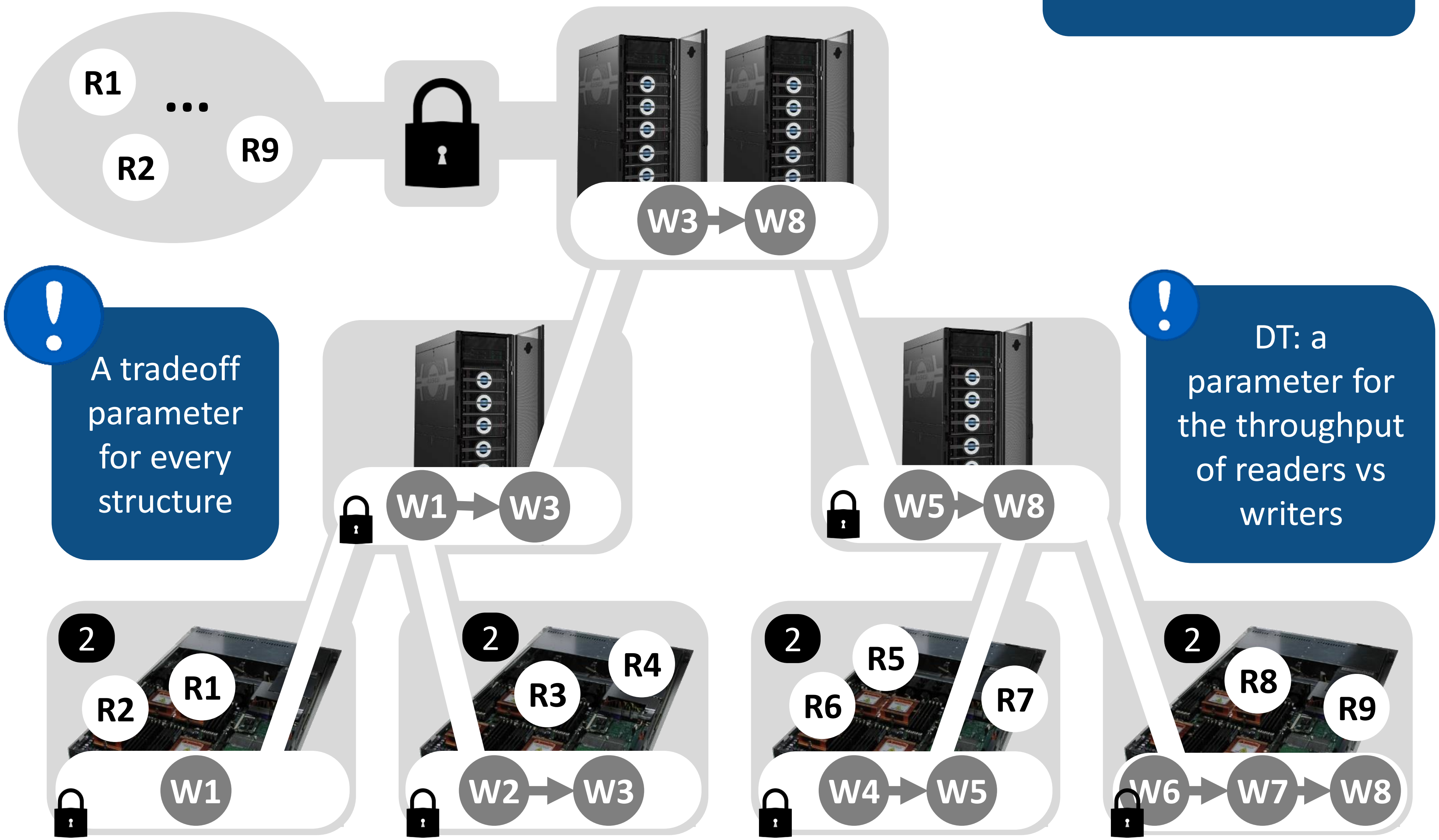
A tradeoff parameter for every structure



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

How to ensure tunable performance?

Each DQ: fairness vs throughput of writers



A tradeoff parameter for every structure

DT: a parameter for the throughput of readers vs writers

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

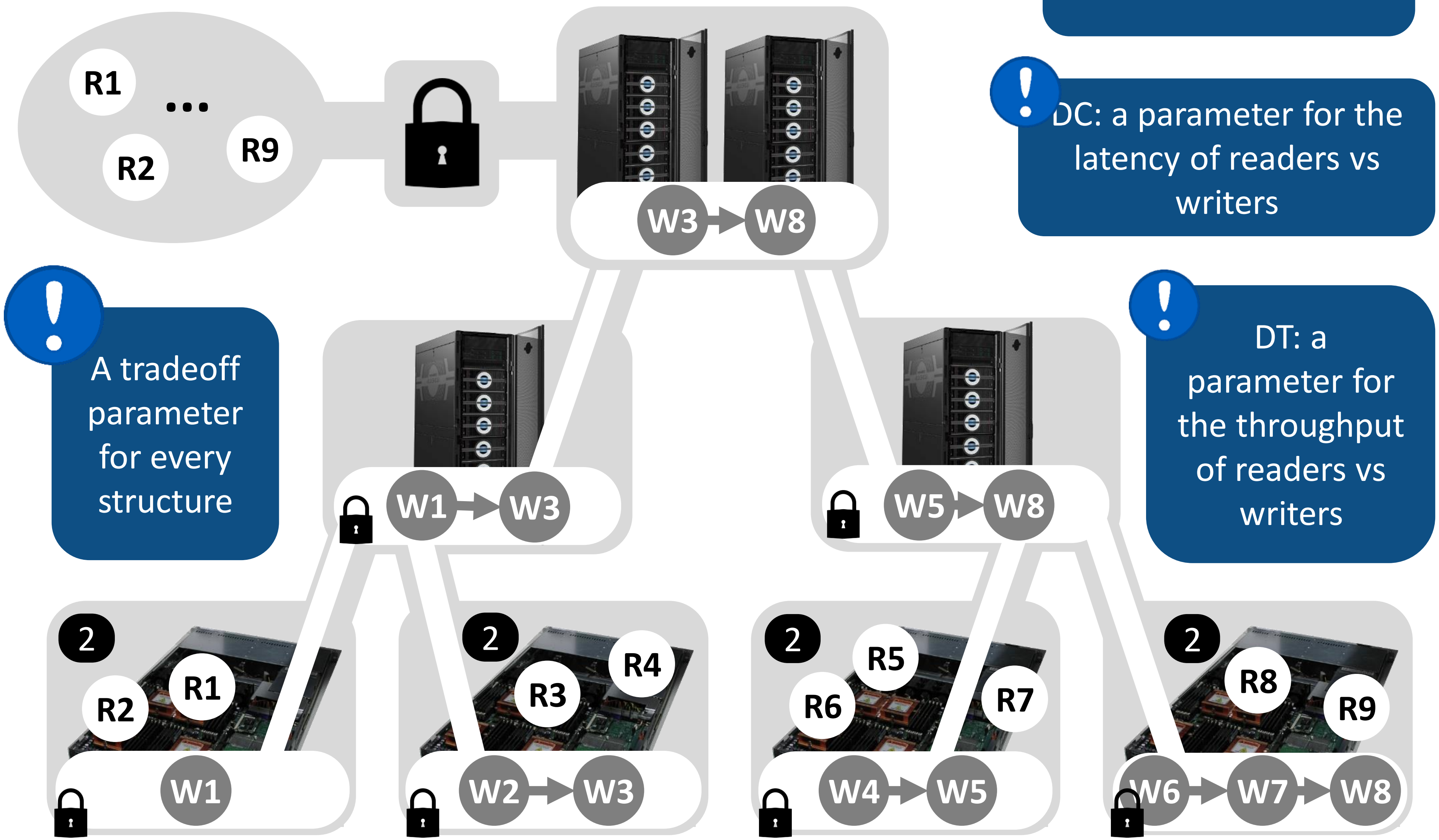
How to ensure tunable performance?

Each DQ: fairness vs throughput of writers

DC: a parameter for the latency of readers vs writers

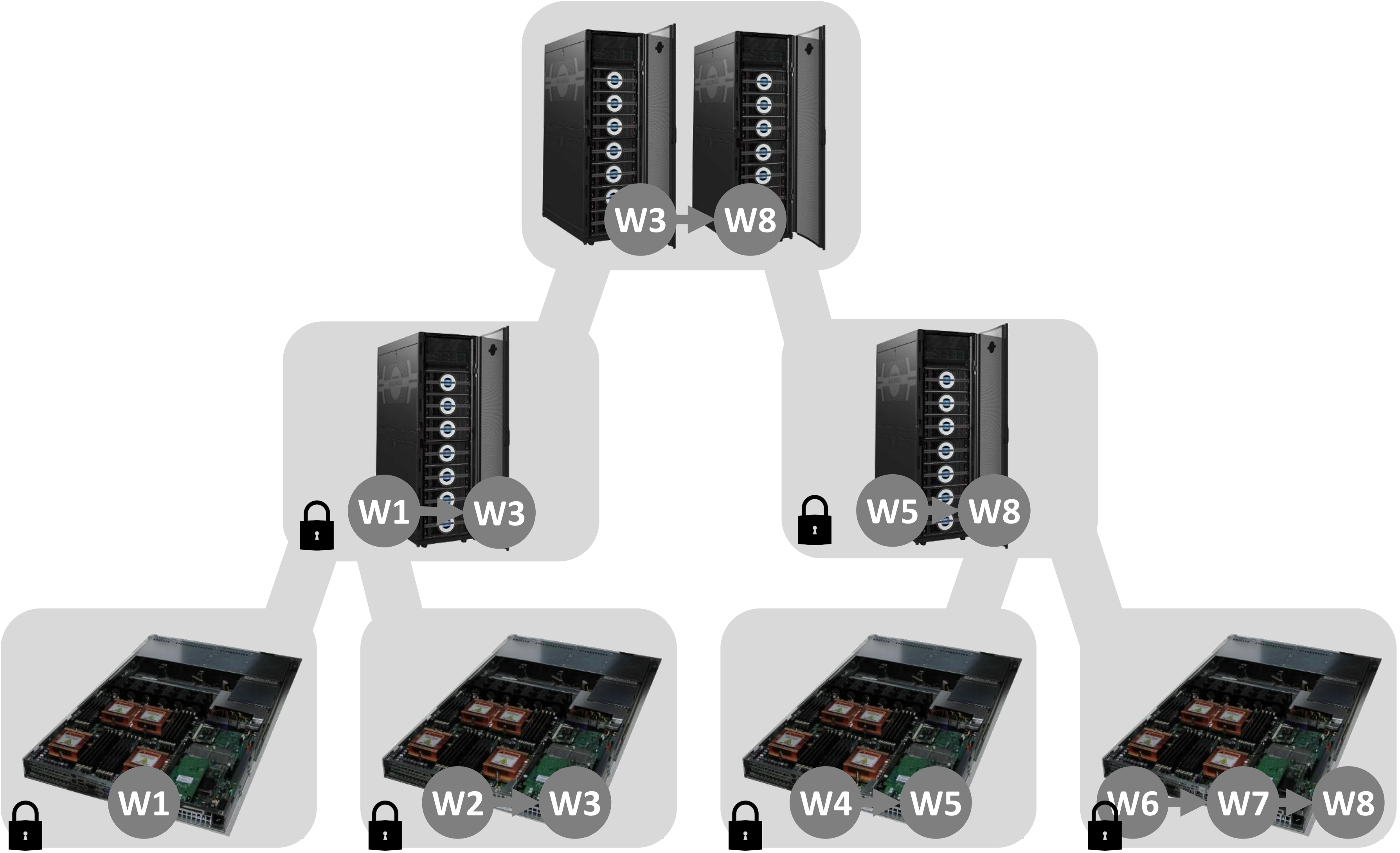
A tradeoff parameter for every structure

DT: a parameter for the throughput of readers vs writers



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed MCS Queues (DQs) - Throughput vs Fairness

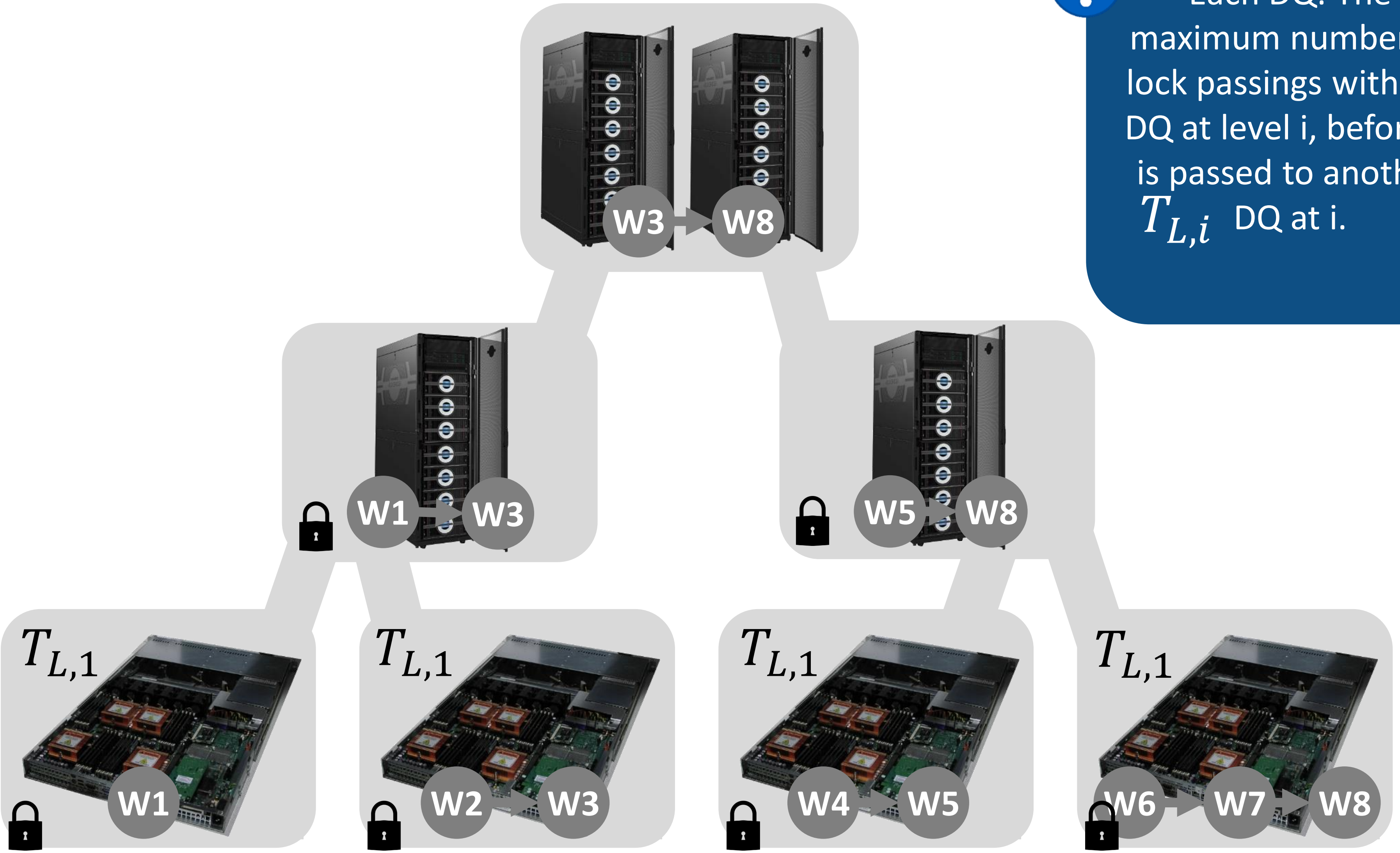


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed MCS Queues (DQs) - Throughput vs Fairness

Each DQ: The maximum number of lock passings within a DQ at level i, before it is passed to another DQ at i. $T_{L,i}$

Distributed MCS Queues (DQs) - Throughput vs Fairness

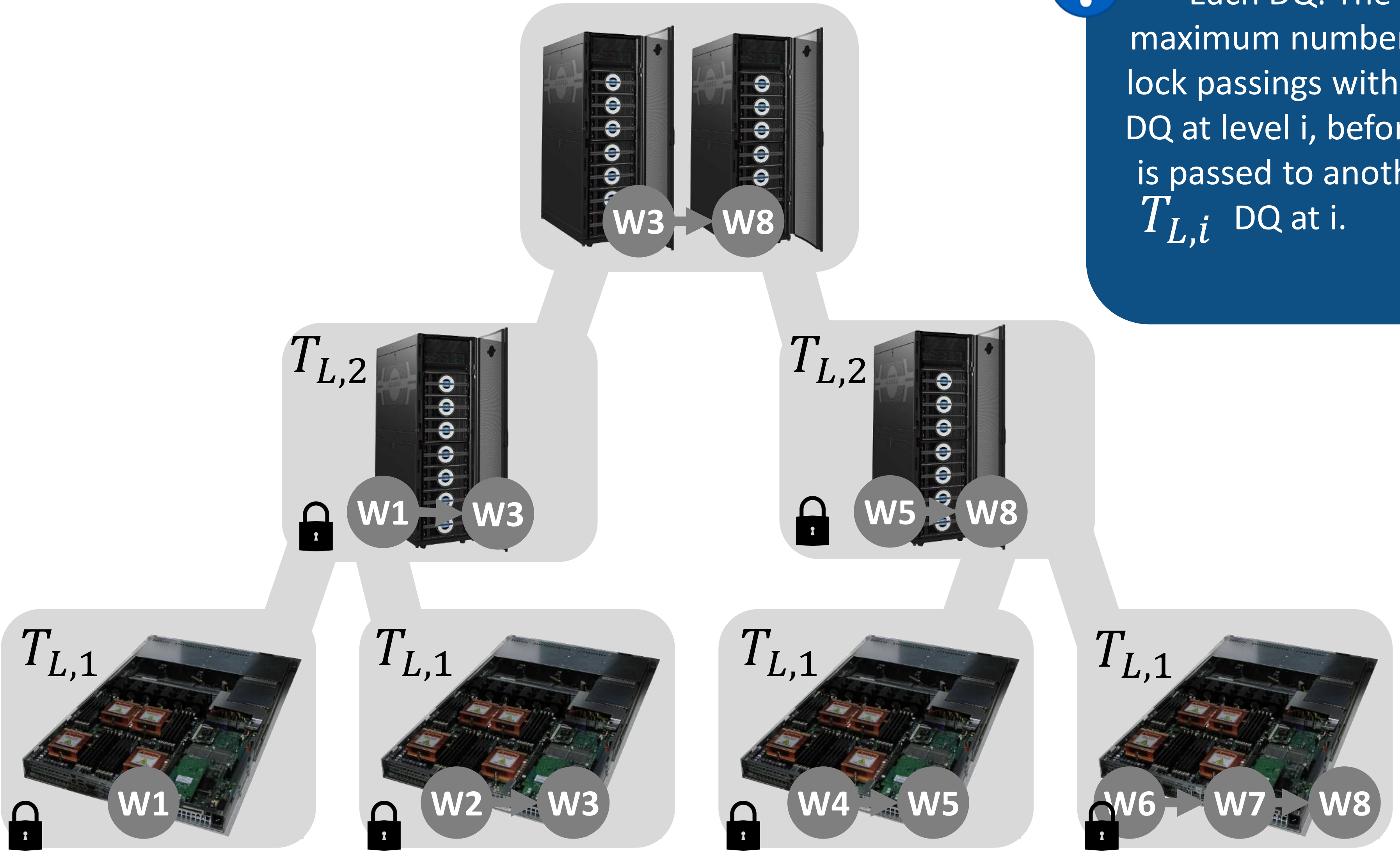


Each DQ: The maximum number of lock passings within a DQ at level i, before it is passed to another DQ at i.

$T_{L,i}$

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed MCS Queues (DQs) - Throughput vs Fairness

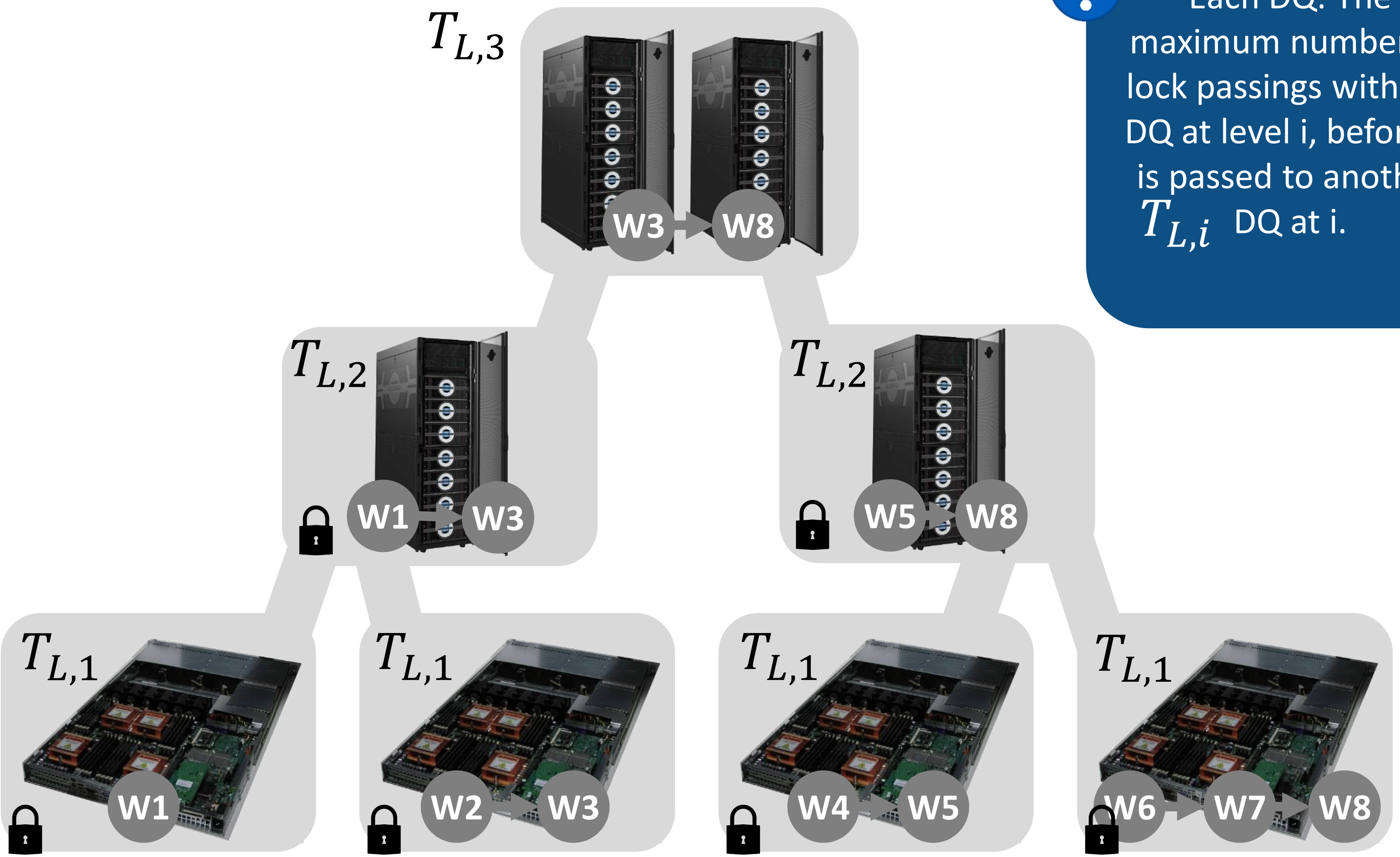


Each DQ: The maximum number of lock passings within a DQ at level i, before it is passed to another DQ at i.

$T_{L,i}$

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed MCS Queues (DQs) - Throughput vs Fairness



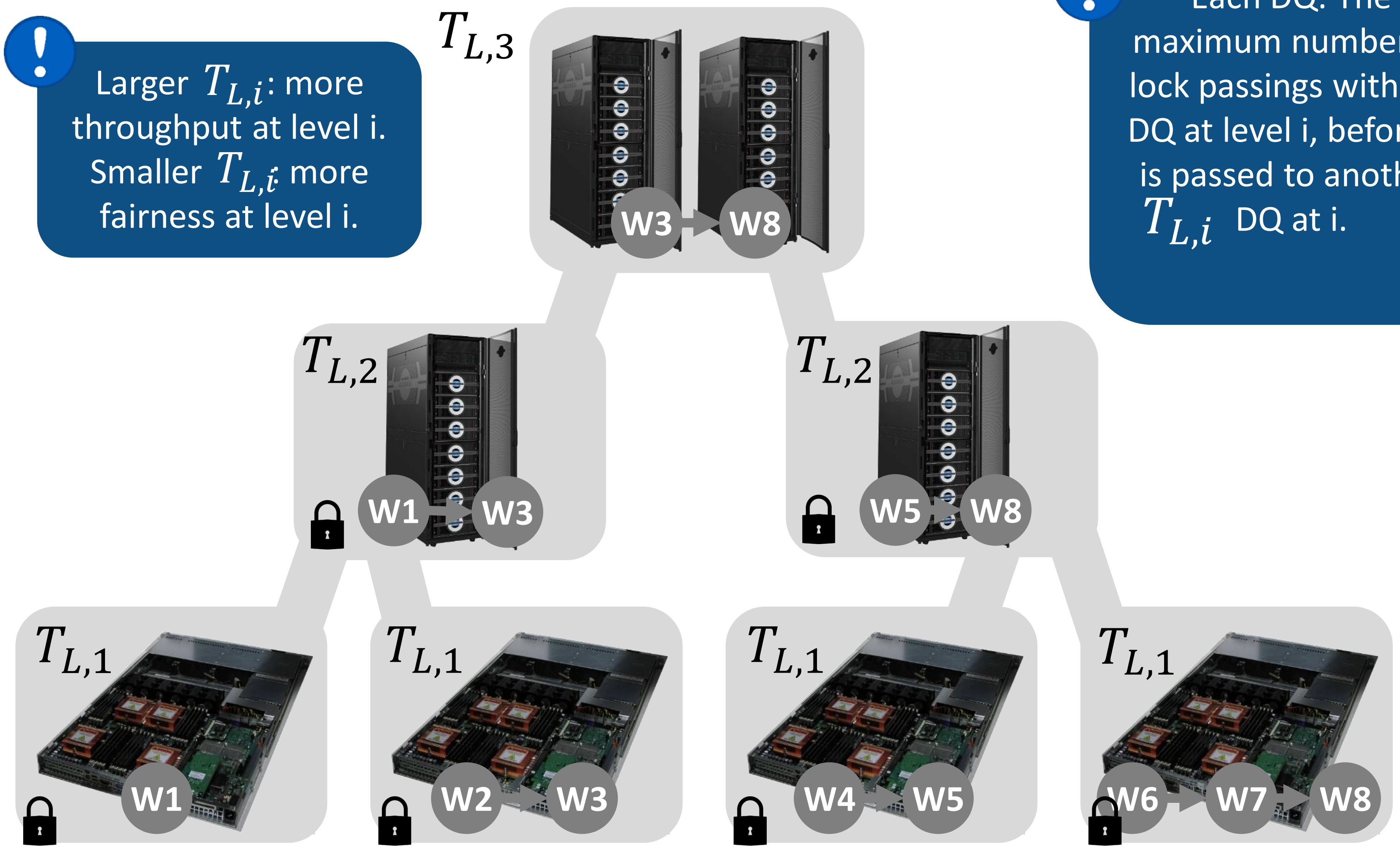
Each DQ: The maximum number of lock passings within a DQ at level i, before it is passed to another DQ at i. $T_{L,i}$

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed MCS Queues (DQs) - Throughput vs Fairness

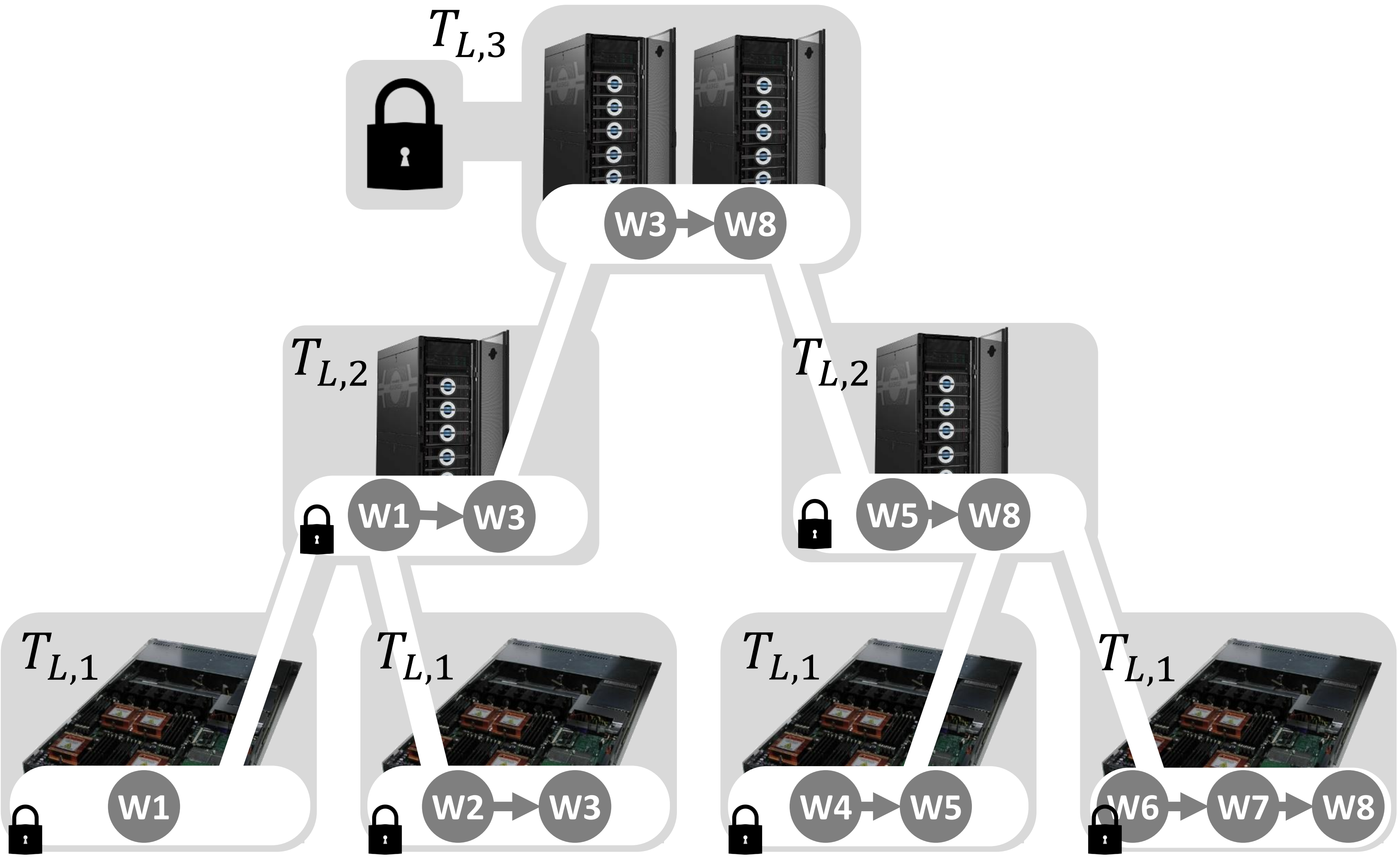
Larger $T_{L,i}$: more throughput at level i. Smaller $T_{L,i}$: more fairness at level i.

Each DQ: The maximum number of lock passings within a DQ at level i, before it is passed to another $T_{L,i}$ DQ at i.



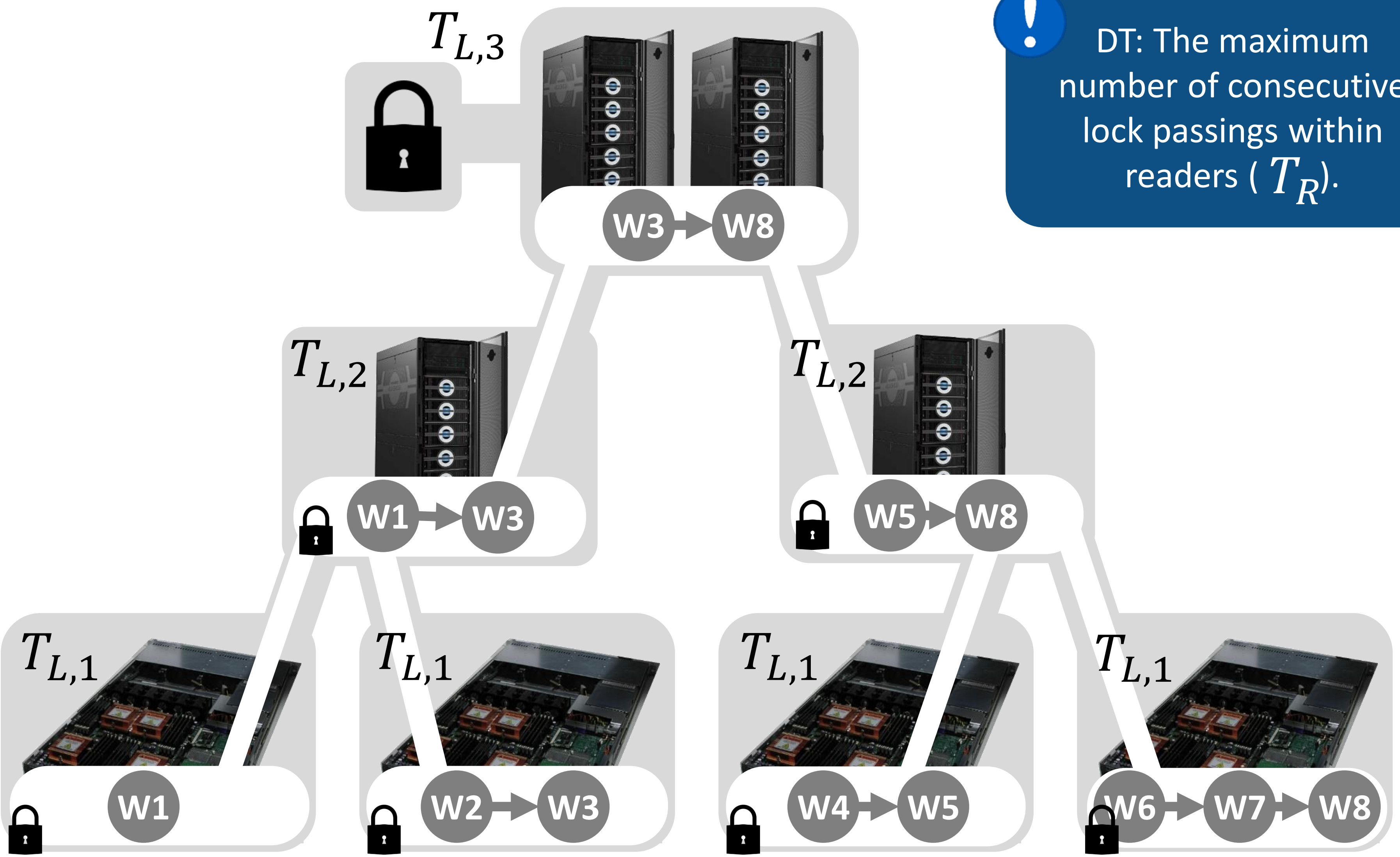
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Tree of Queues (DT) - Throughput of readers vs writers



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

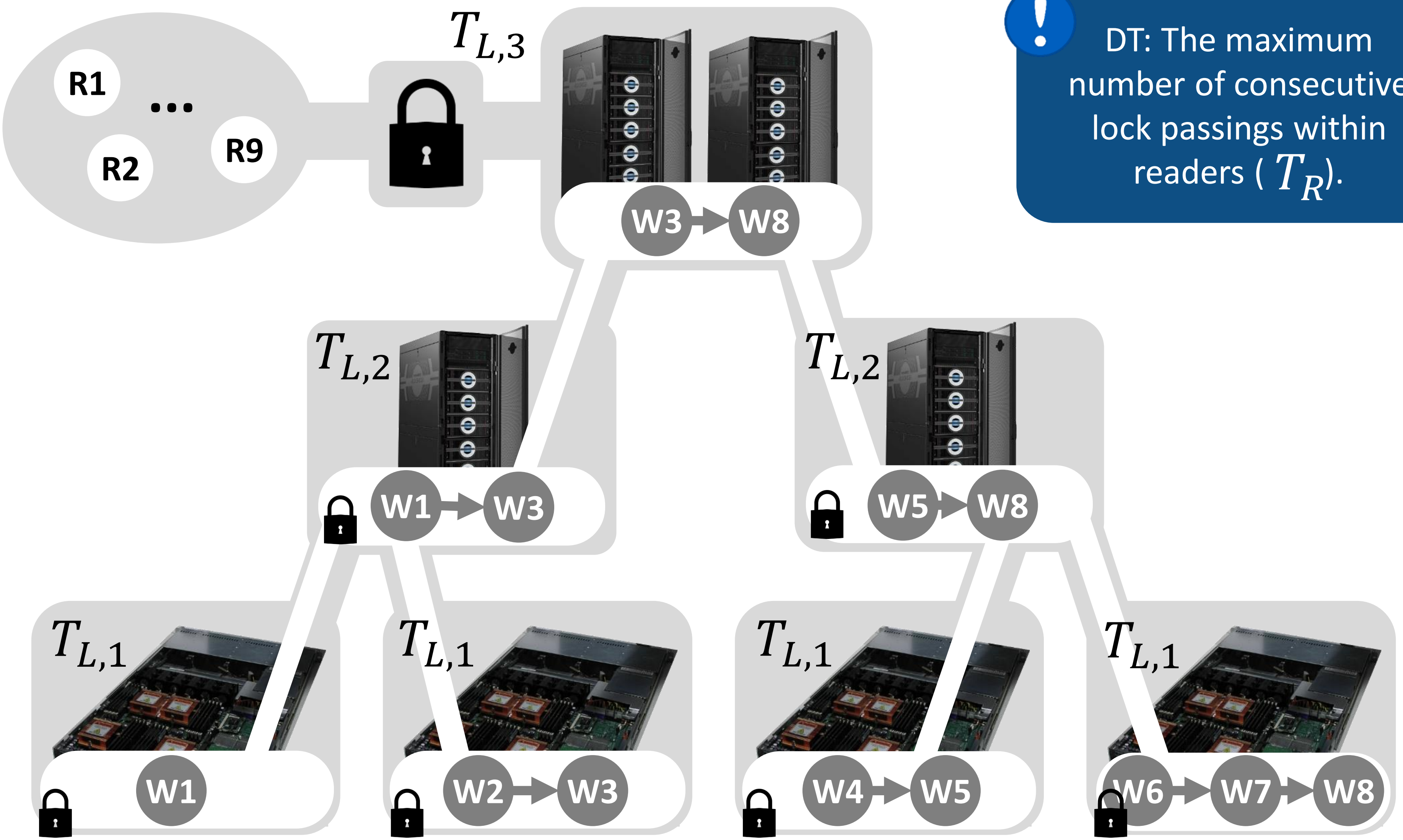
Distributed Tree of Queues (DT) - Throughput of readers vs writers



DT: The maximum number of consecutive lock passings within readers (T_R).

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Tree of Queues (DT) - Throughput of readers vs writers



DT: The maximum number of consecutive lock passings within readers (T_R).

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

DC: every *k*th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

DC: every kth compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

b|x|y

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

DC: every kth compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

A writer holds the lock

b|x|y

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

DC: every kth compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

A writer holds the lock — b|x|y — Readers that arrived at the CS

Distributed Counter (DC) - Latency of readers vs writers

DC: every kth compute node hosts a partial counter, all of which constitute the DC.

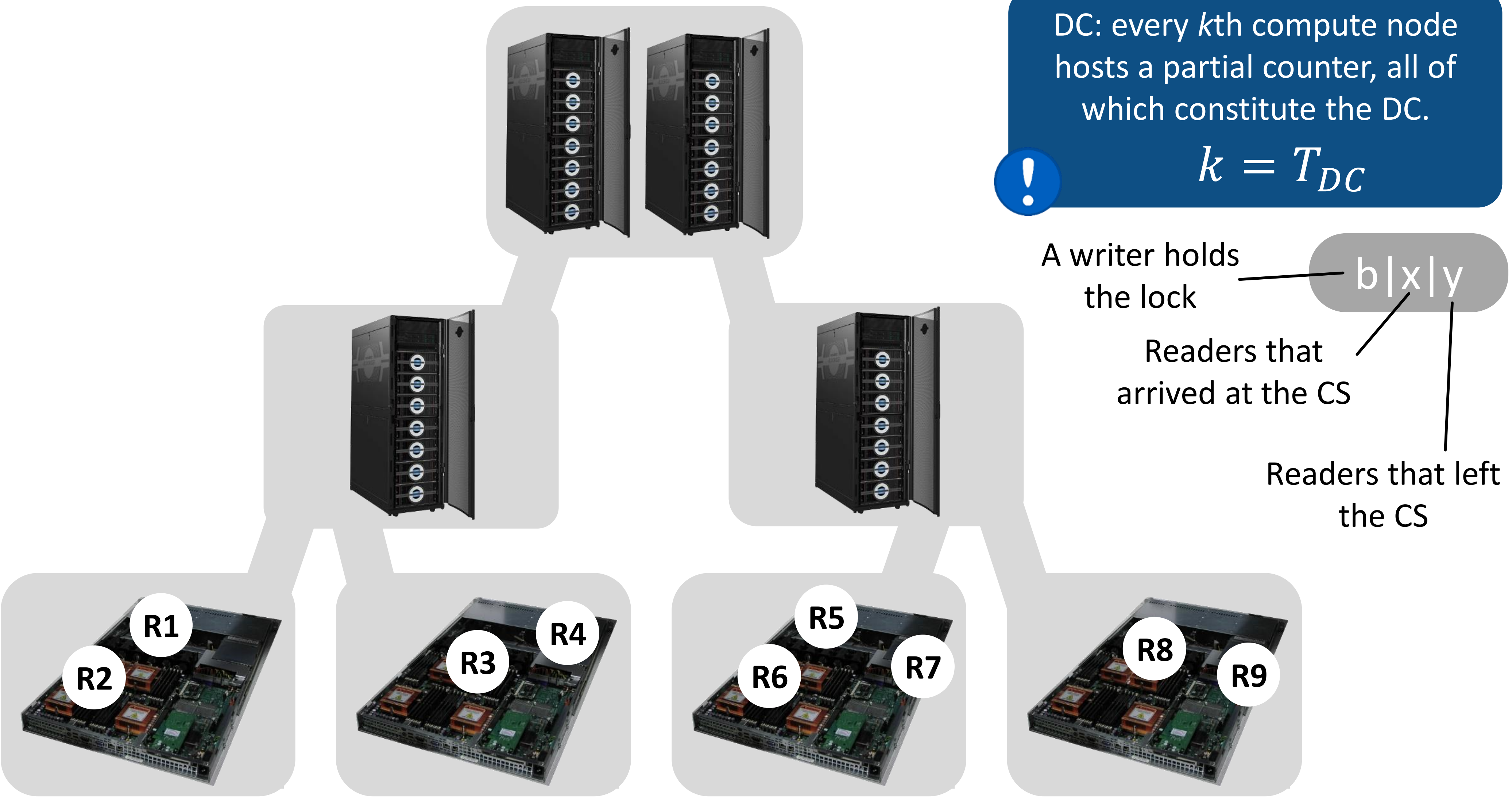
$$k = T_{DC}$$

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

DC: every *k*th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$



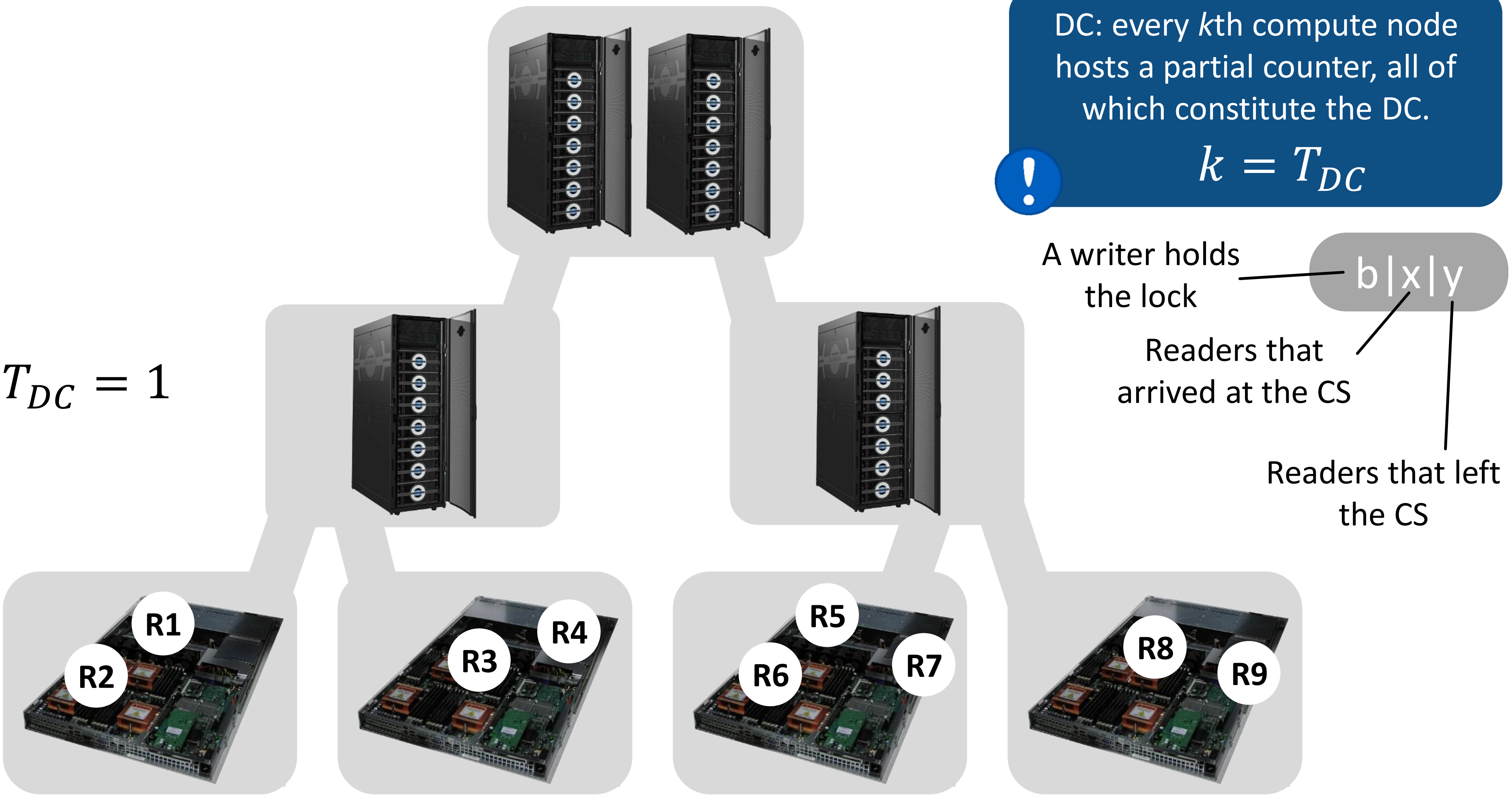
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

DC: every *k*th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

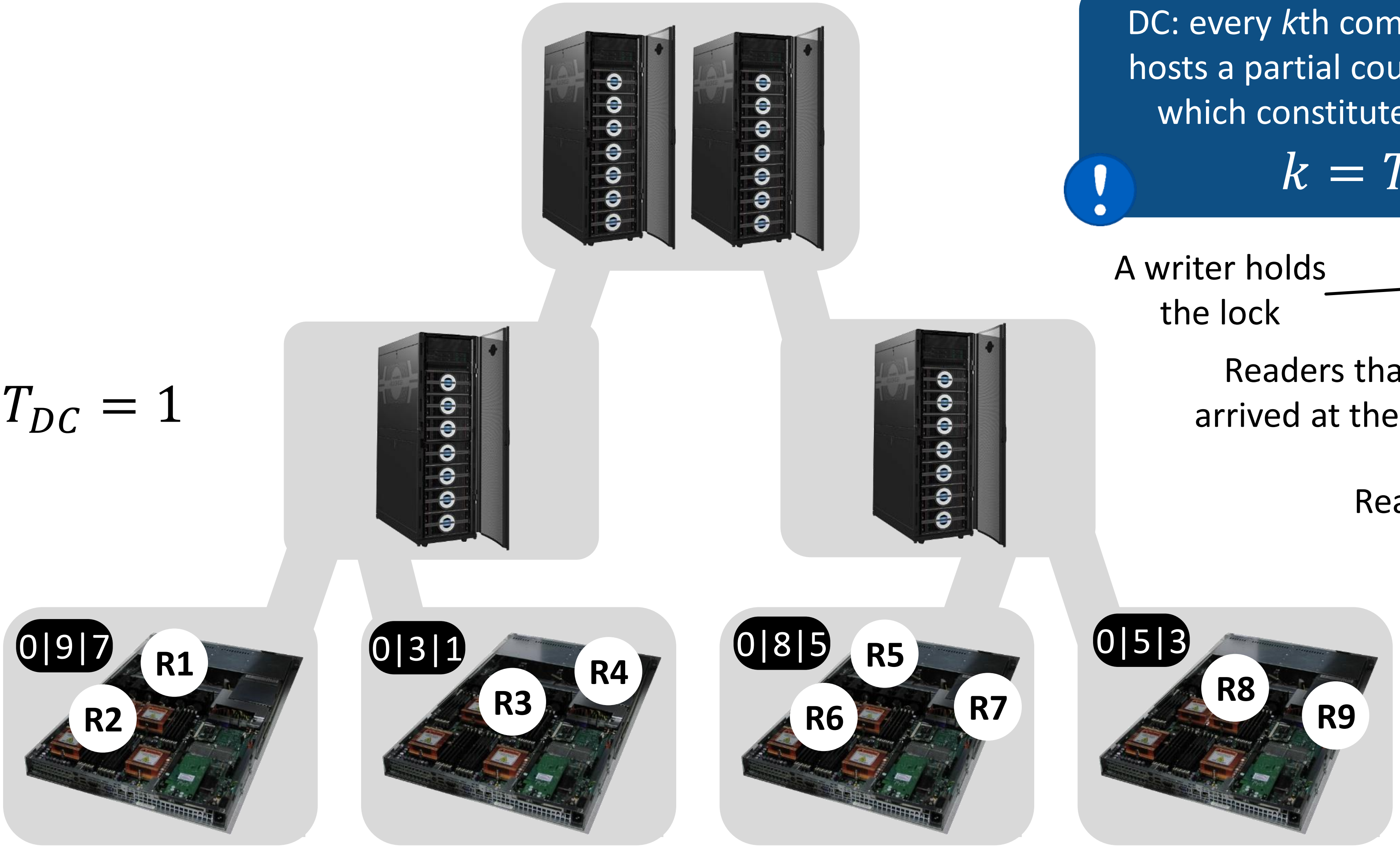
$T_{DC} = 1$



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

$T_{DC} = 1$



DC: every *k*th compute node hosts a partial counter, all of which constitute the DC.

$k = T_{DC}$

b|x|y

A writer holds the lock

Readers that arrived at the CS

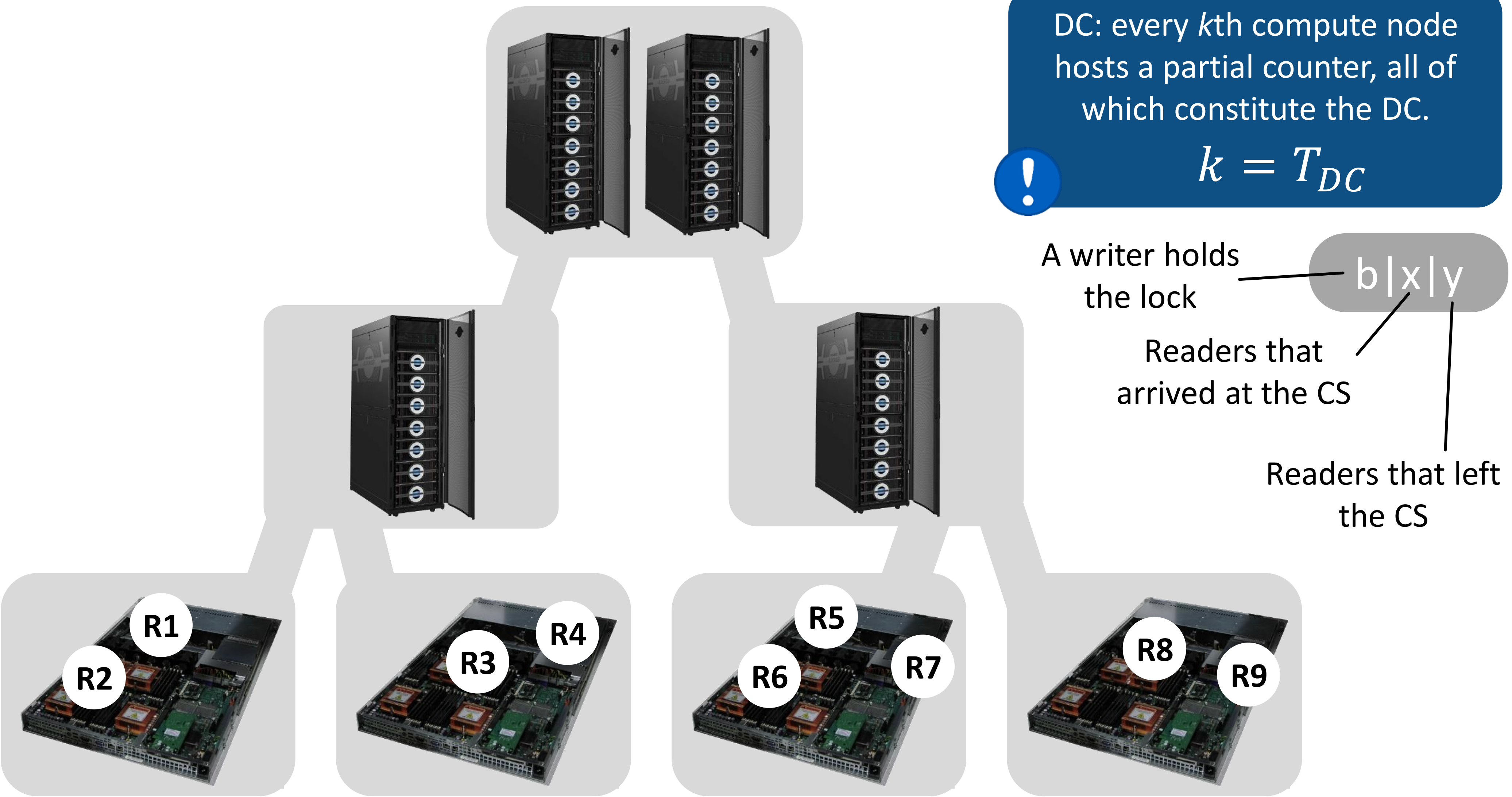
Readers that left the CS

P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers

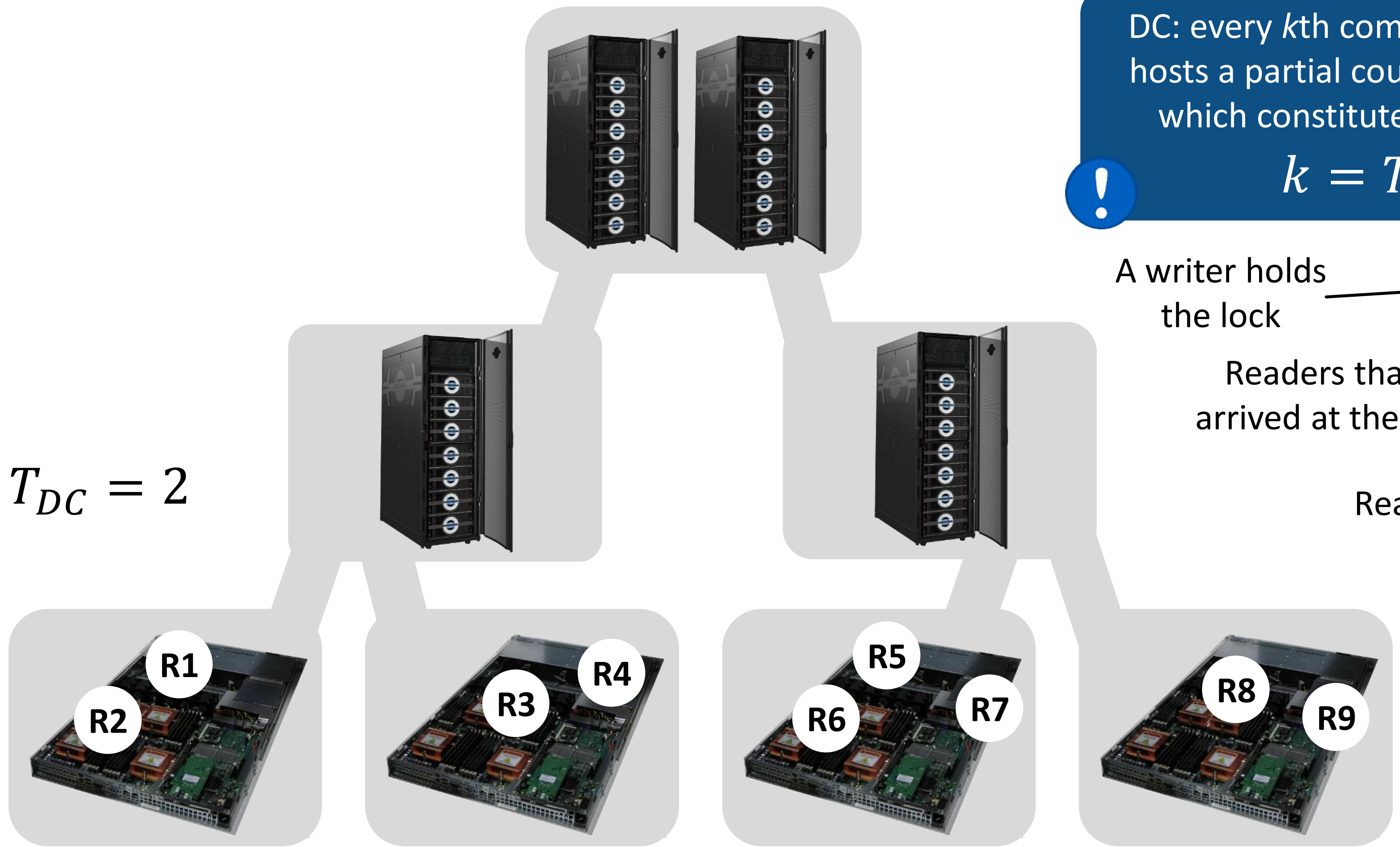
DC: every *k*th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Distributed Counter (DC) - Latency of readers vs writers



DC: every *k*th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

A writer holds the lock

b|x|y

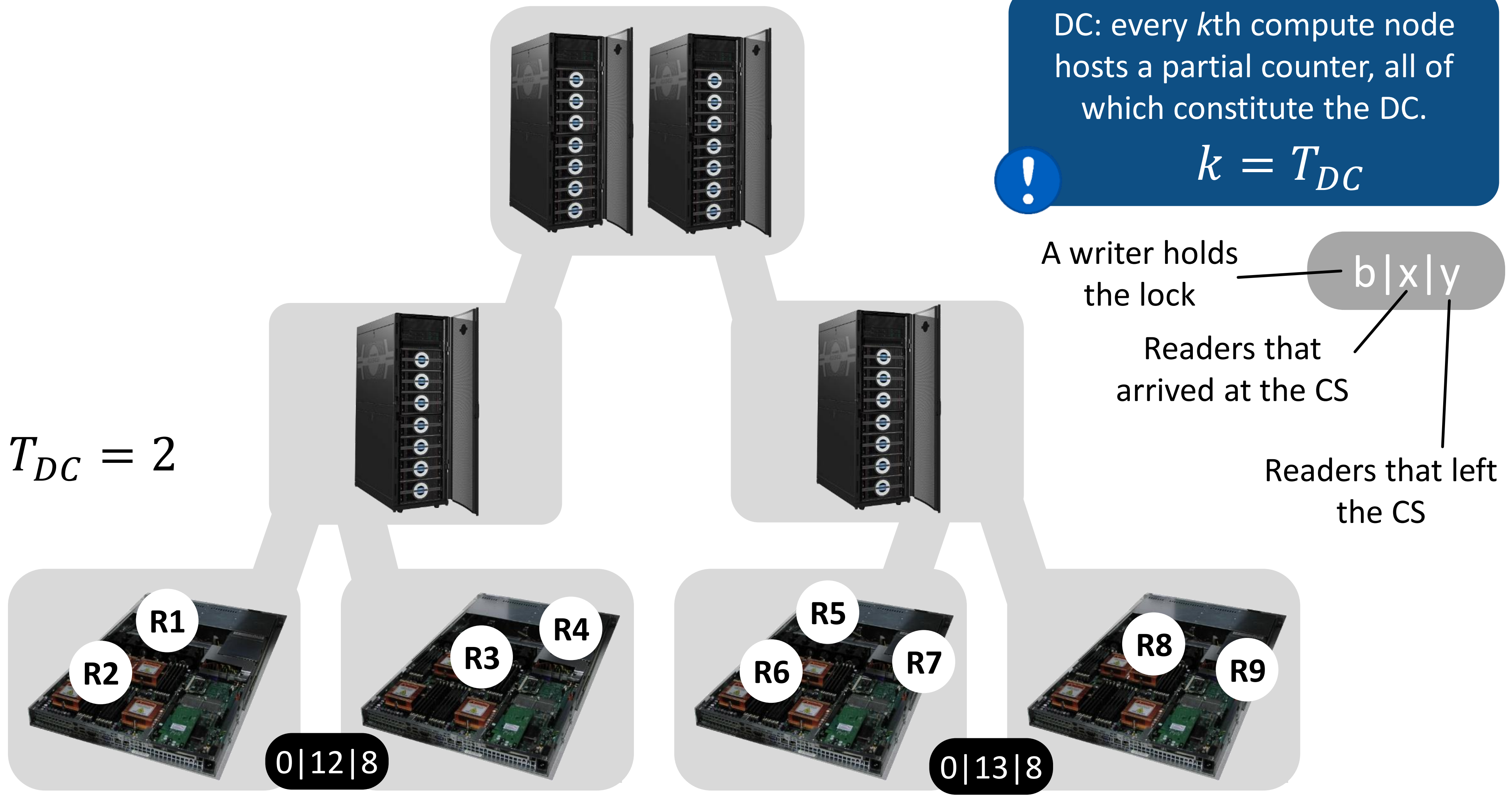
Readers that arrived at the CS

Readers that left the CS

Distributed Counter (DC) - Latency of readers vs writers

DC: every kth compute node hosts a partial counter, all of which constitute the DC.

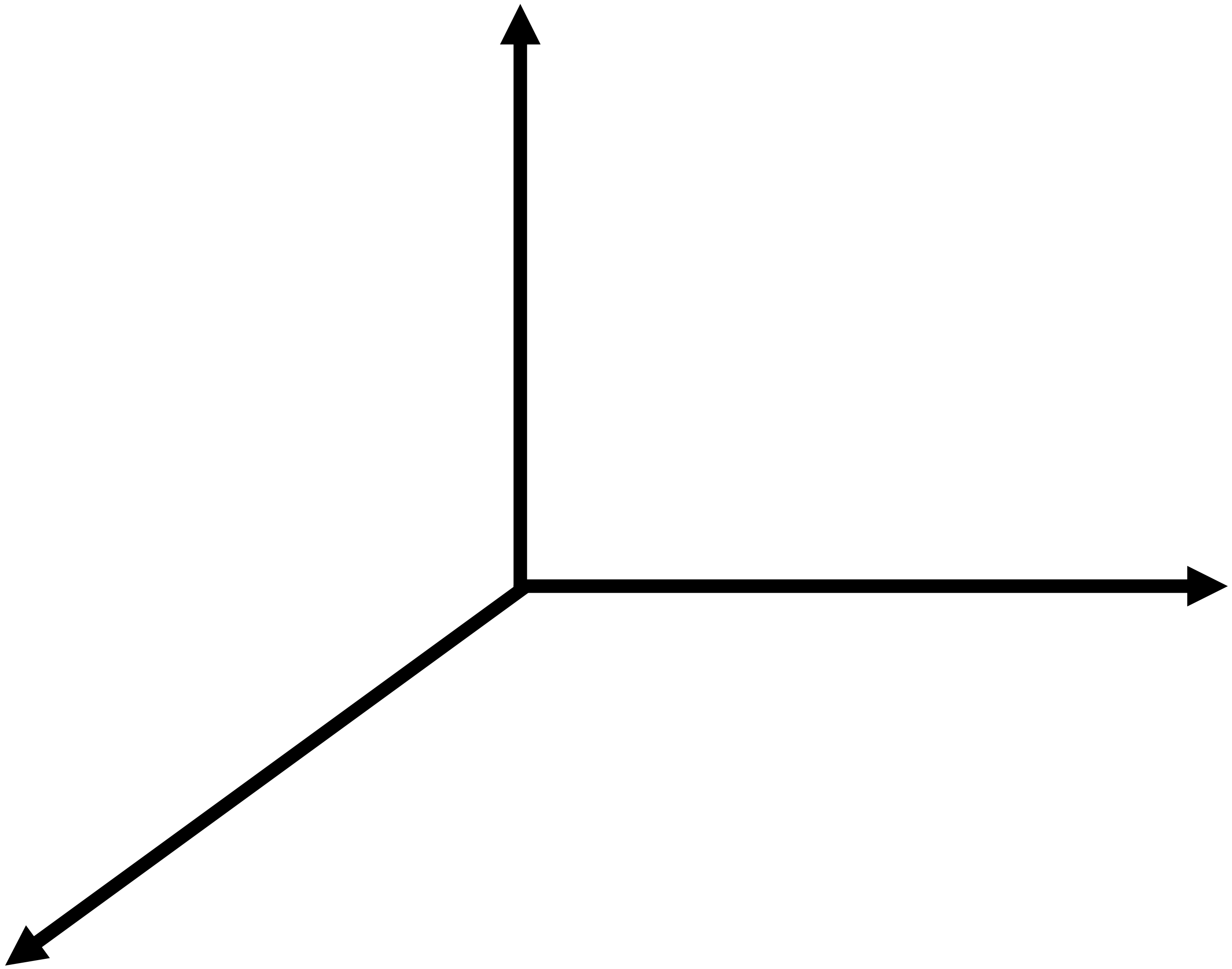
$$k = T_{DC}$$



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

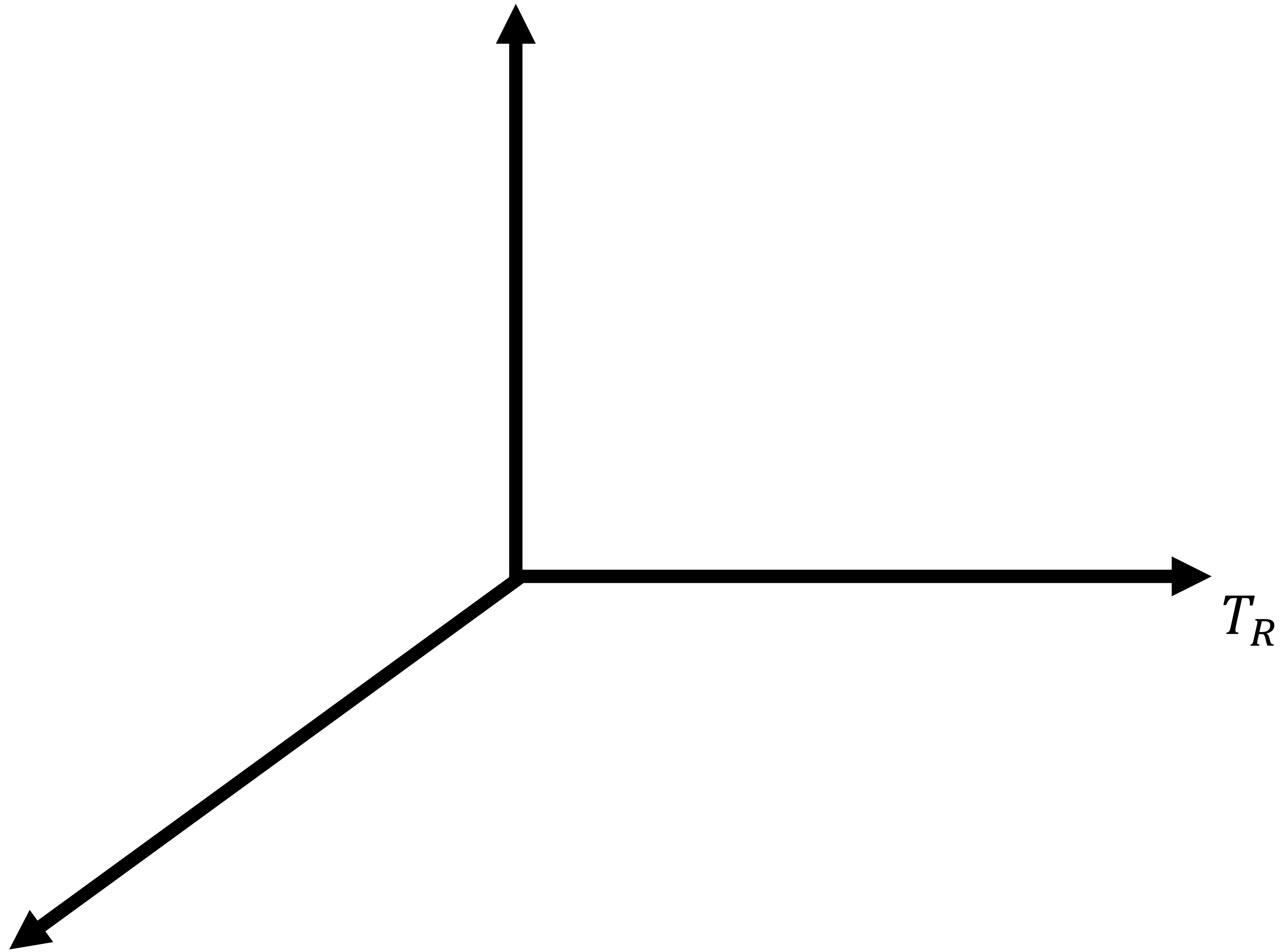
Design space

Design space

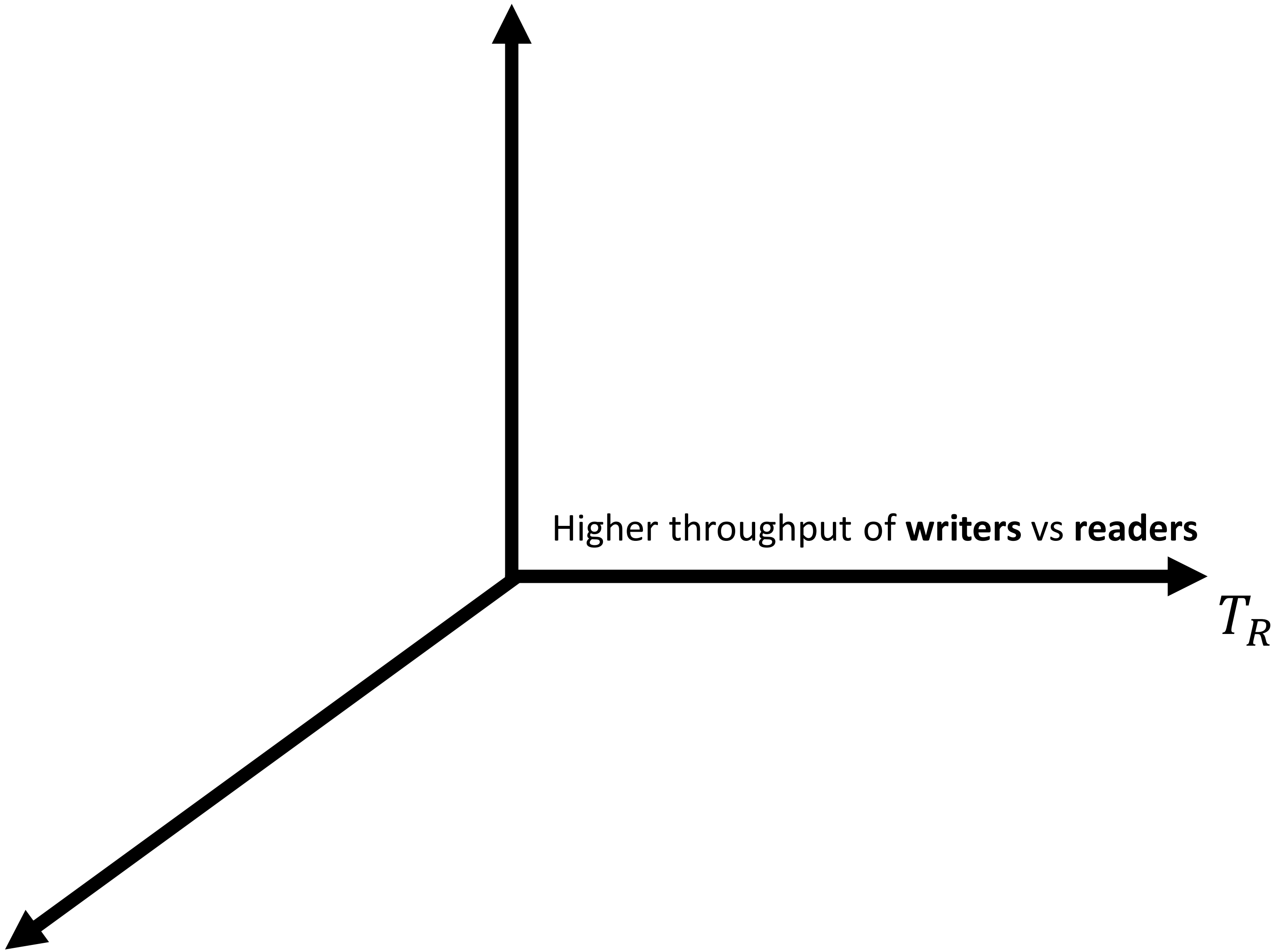


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Design space

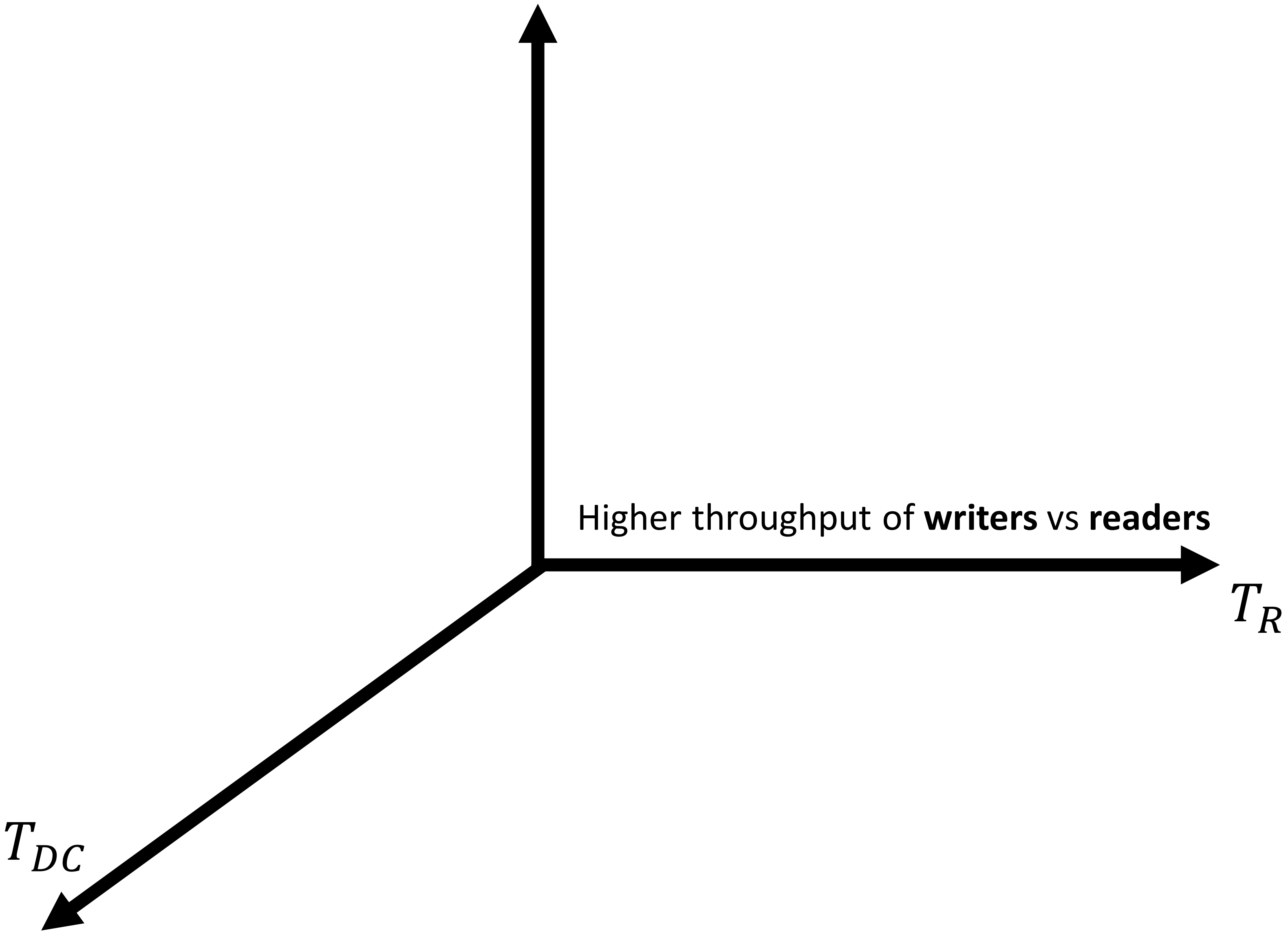


Design space



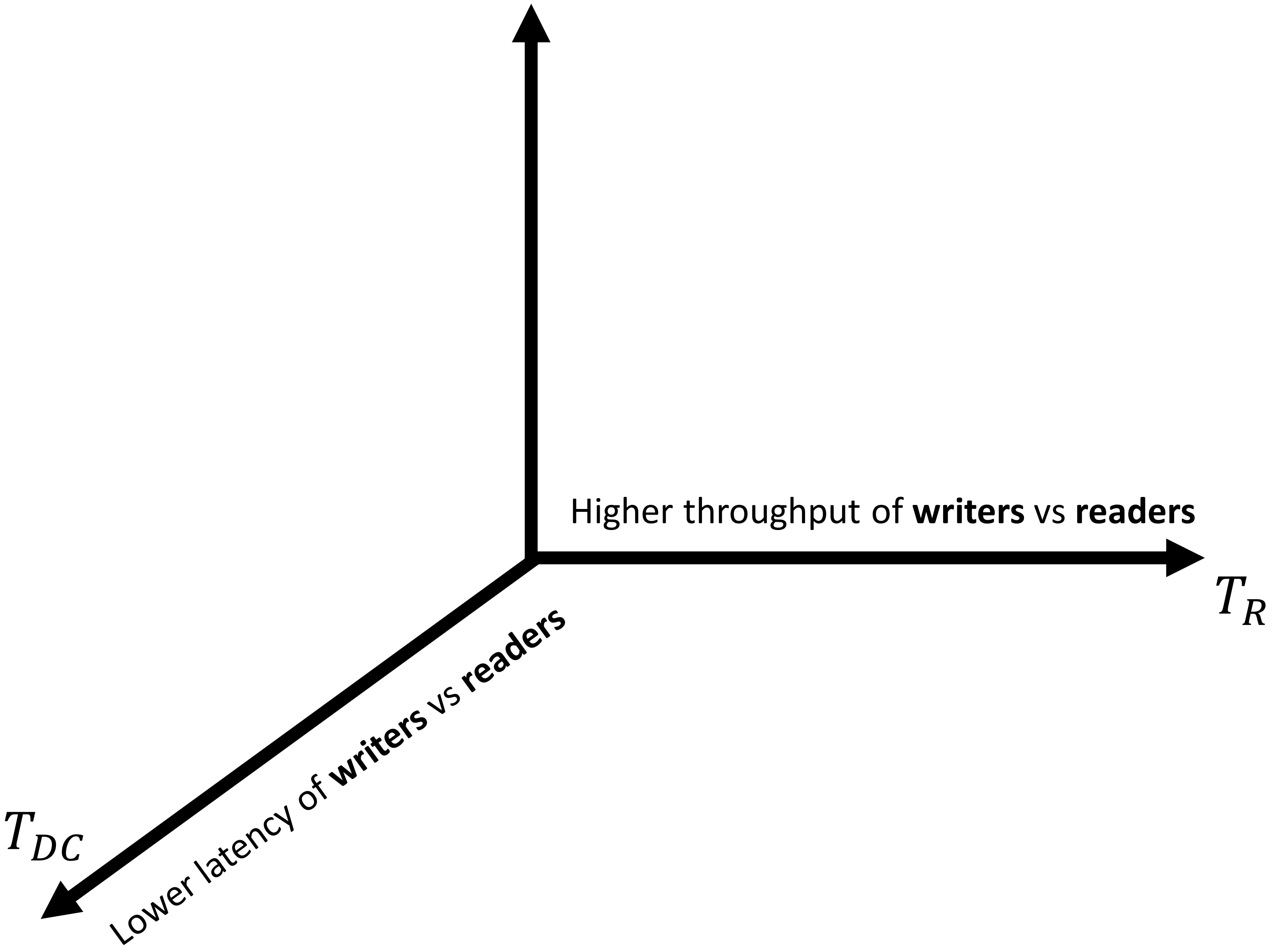
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Design space



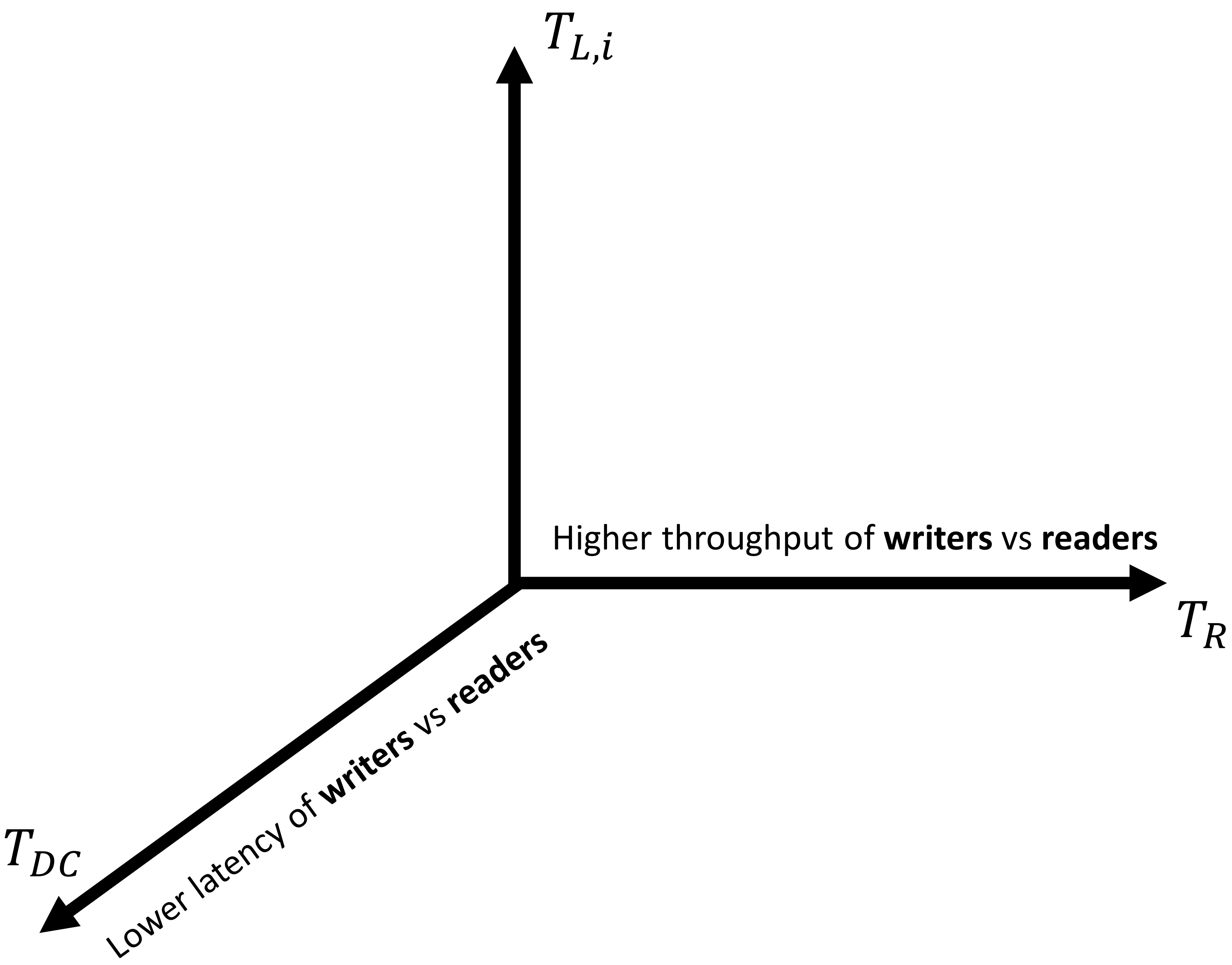
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Design space



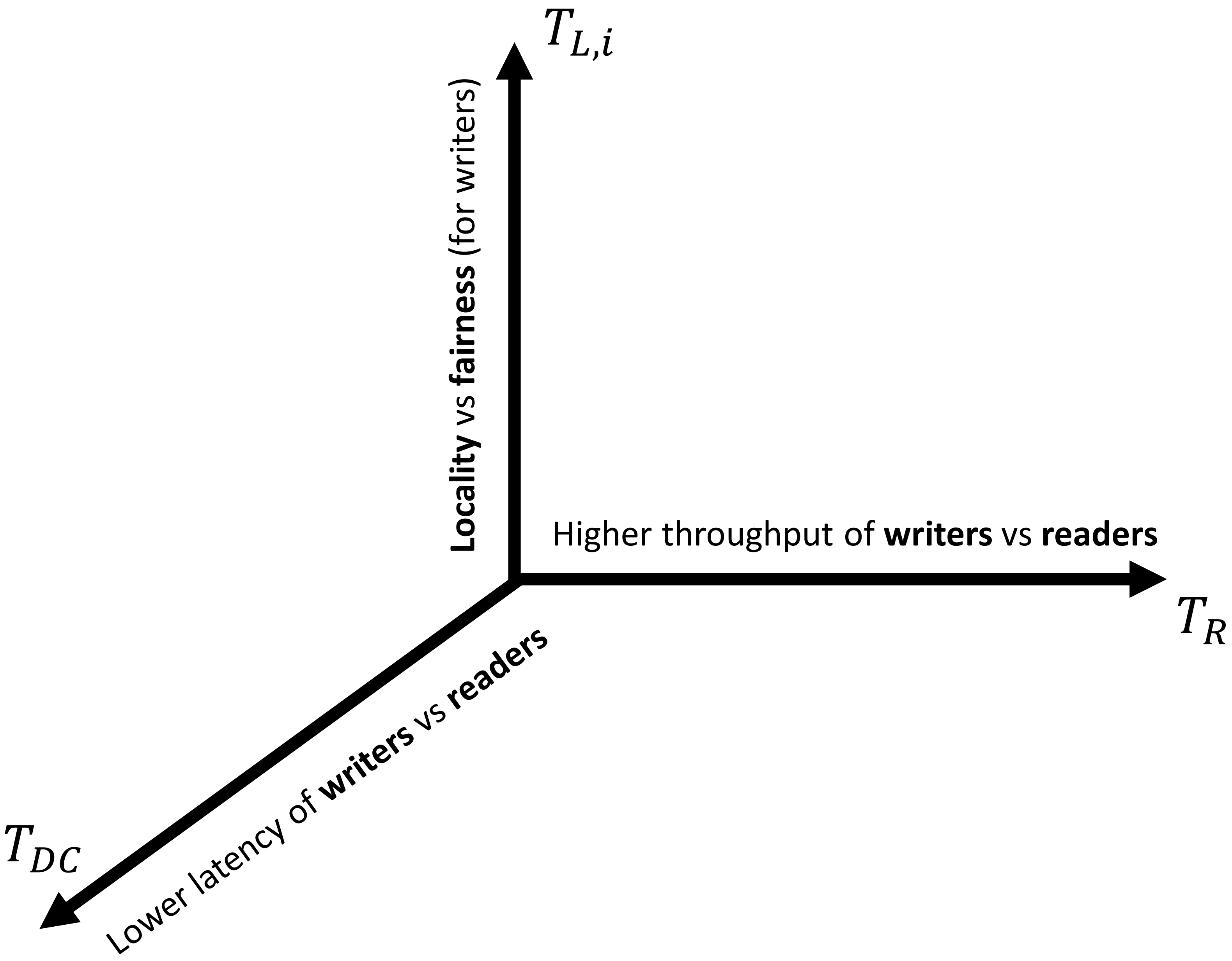
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Design space



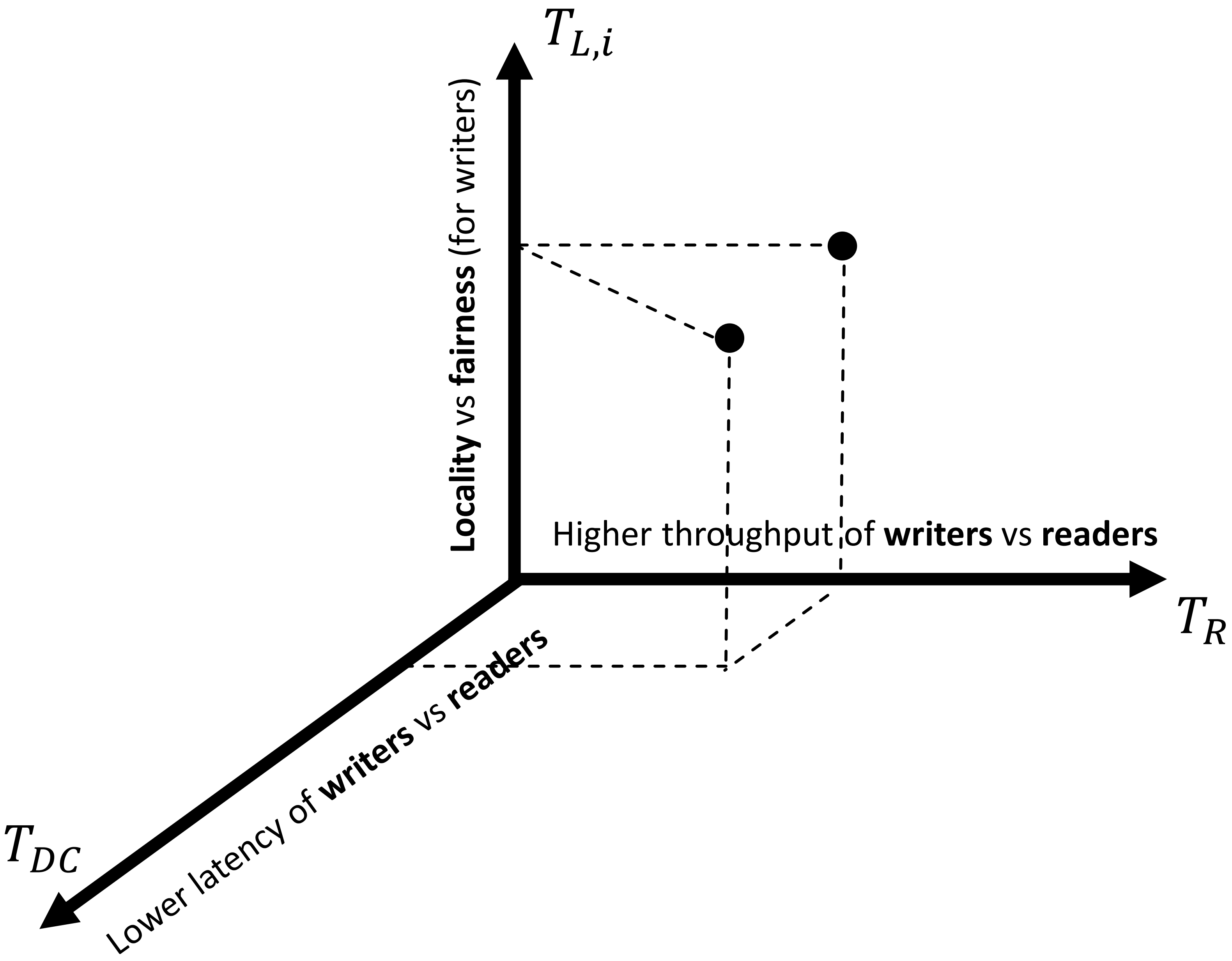
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Design space

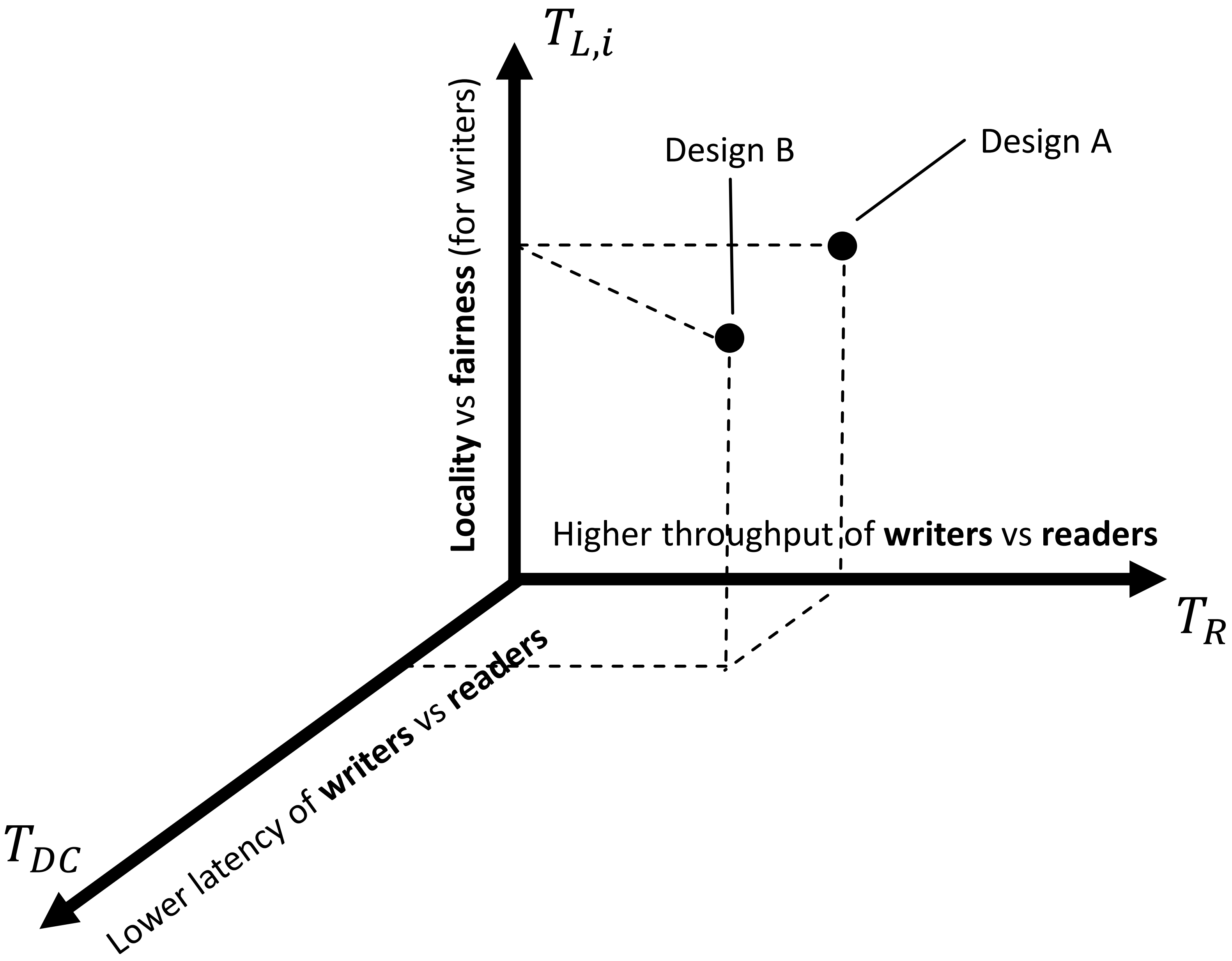


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Design space

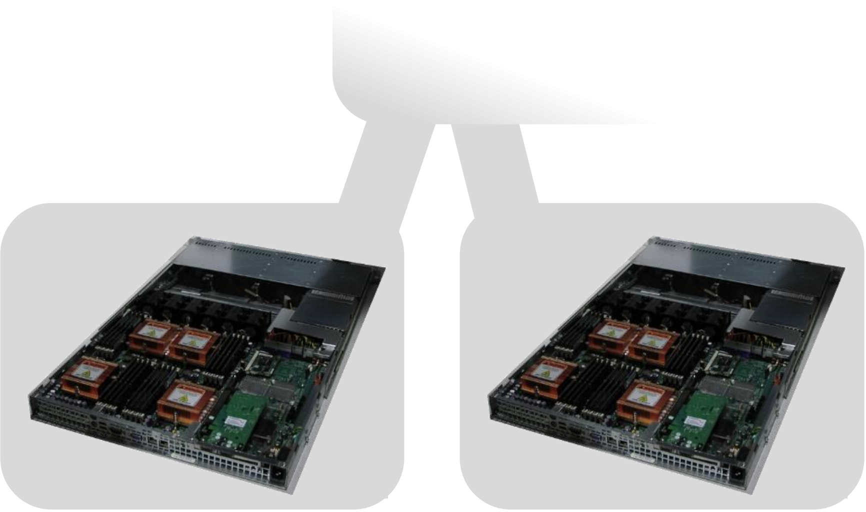


Design space



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

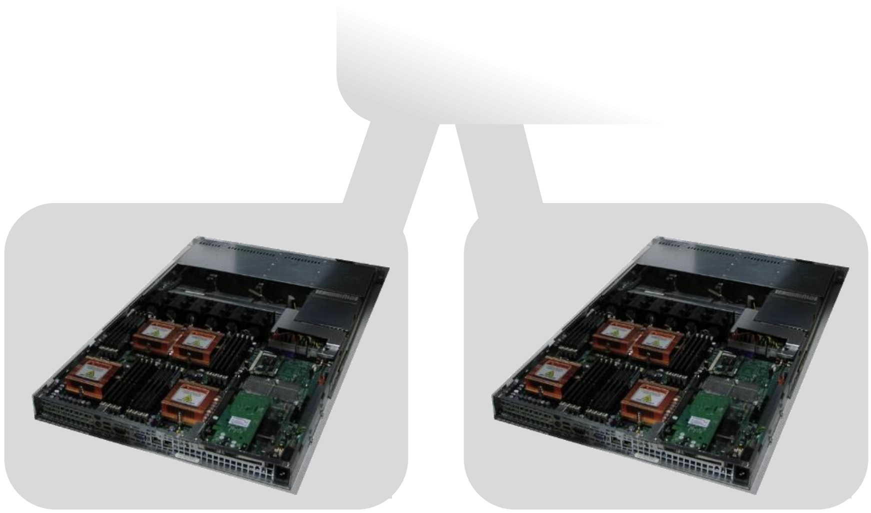
Lock Acquire by Readers



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper

Lock Acquire by Readers

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



Lock Acquire by Readers

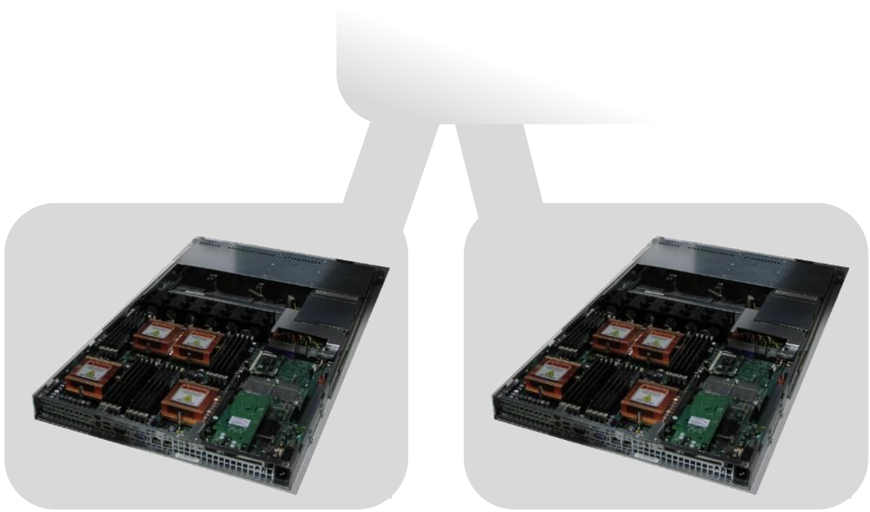
A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

b|x|y

A writer holds the lock

Readers that arrived at the CS

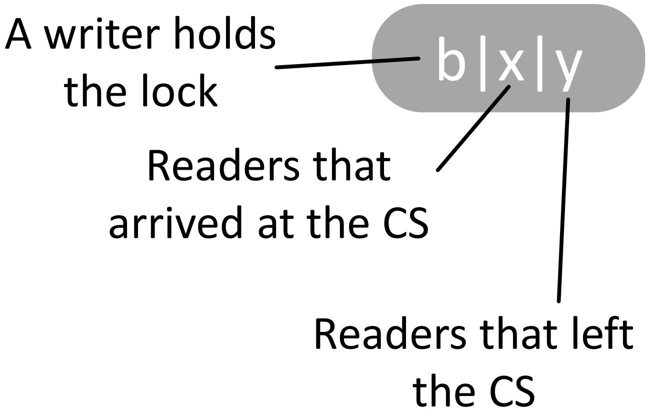
Readers that left the CS

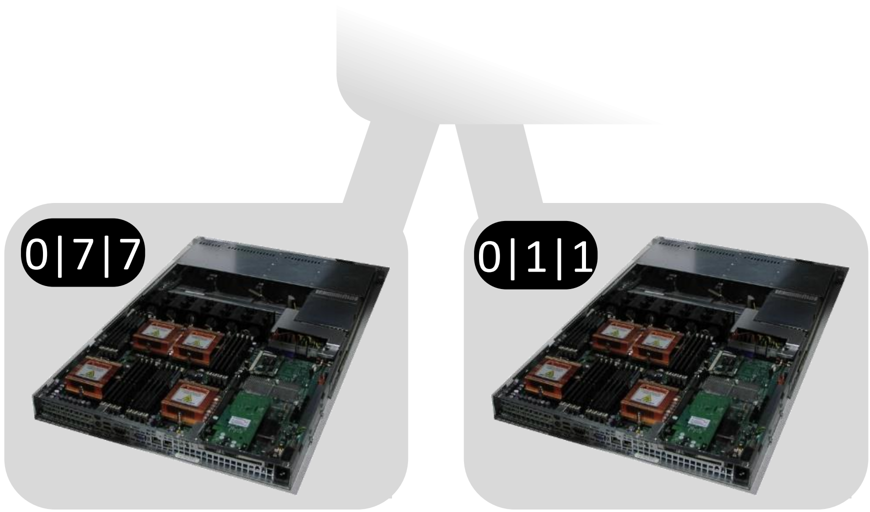


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation





Lock Acquire by Readers

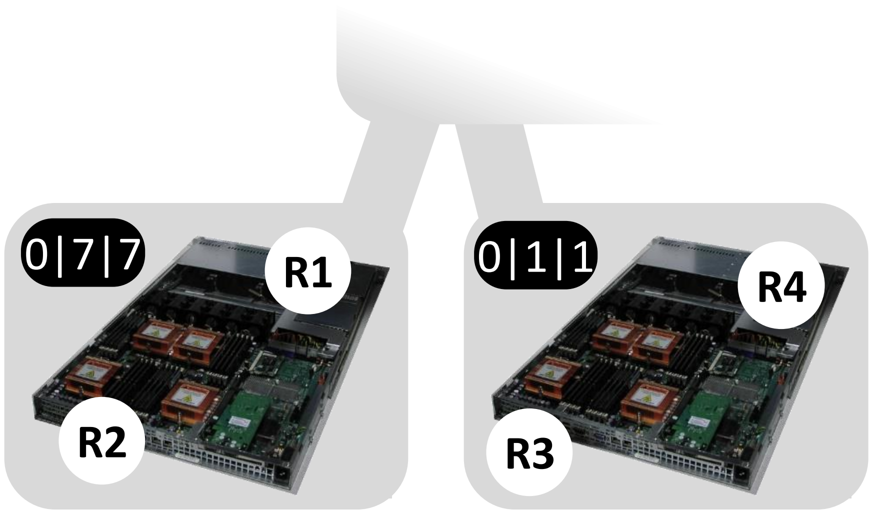
A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

A writer holds the lock

b|x|y

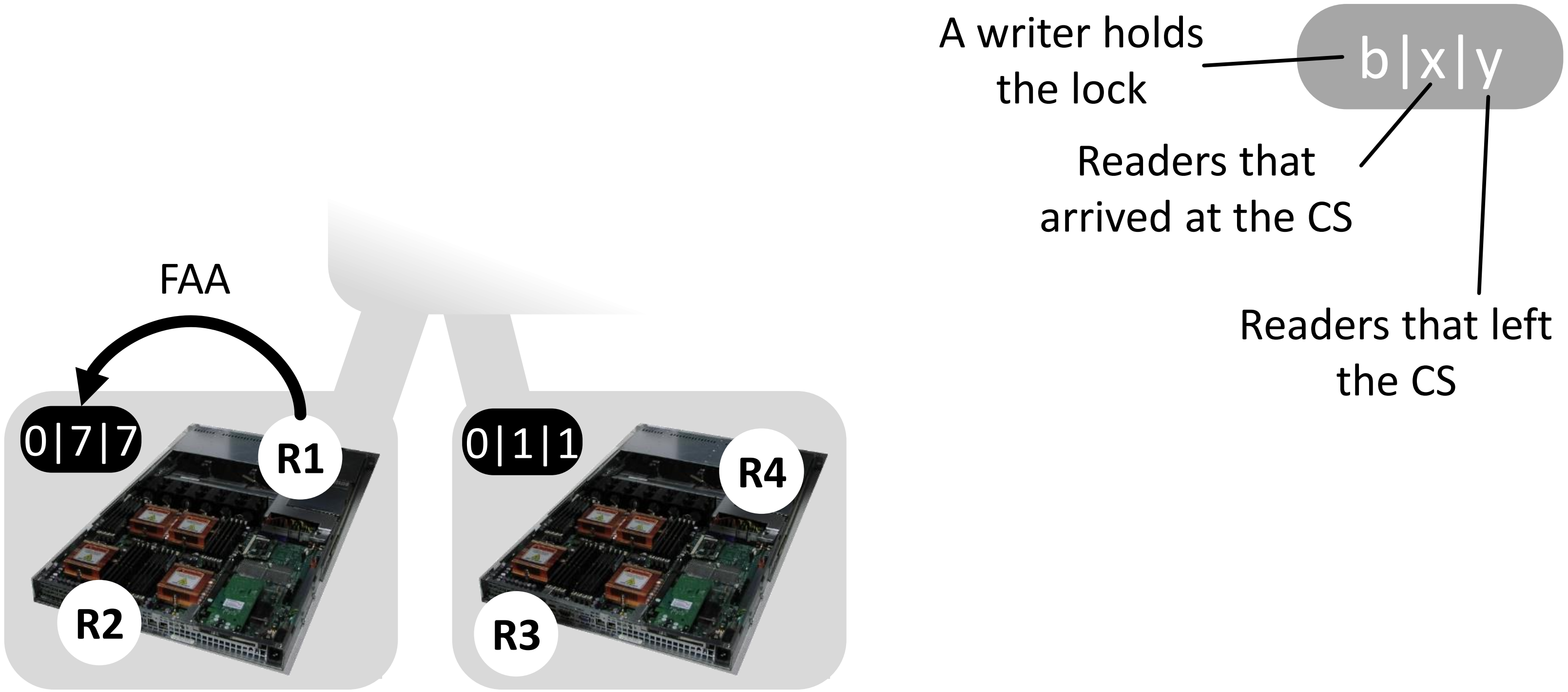
Readers that arrived at the CS

Readers that left the CS



Lock Acquire by Readers

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

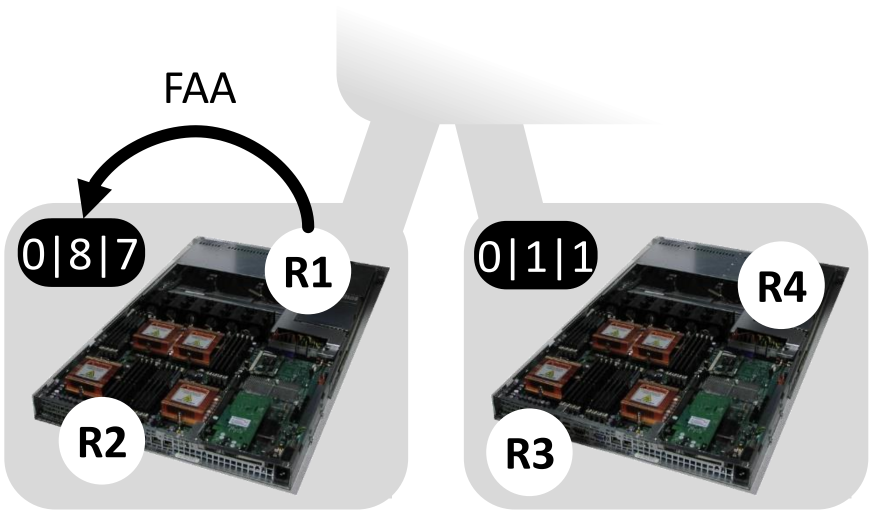
Lock Acquire by Readers

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

A writer holds the lock — b|x|y

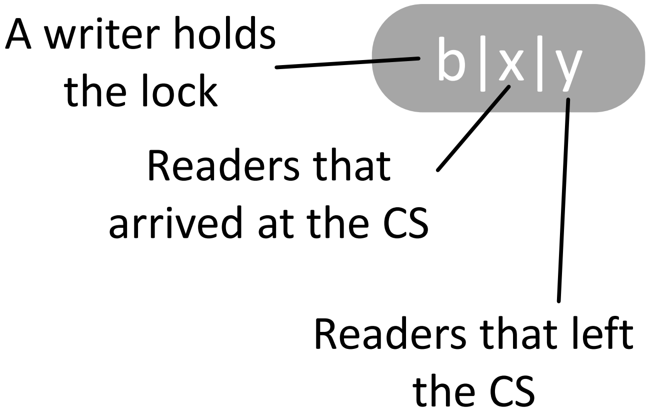
Readers that arrived at the CS

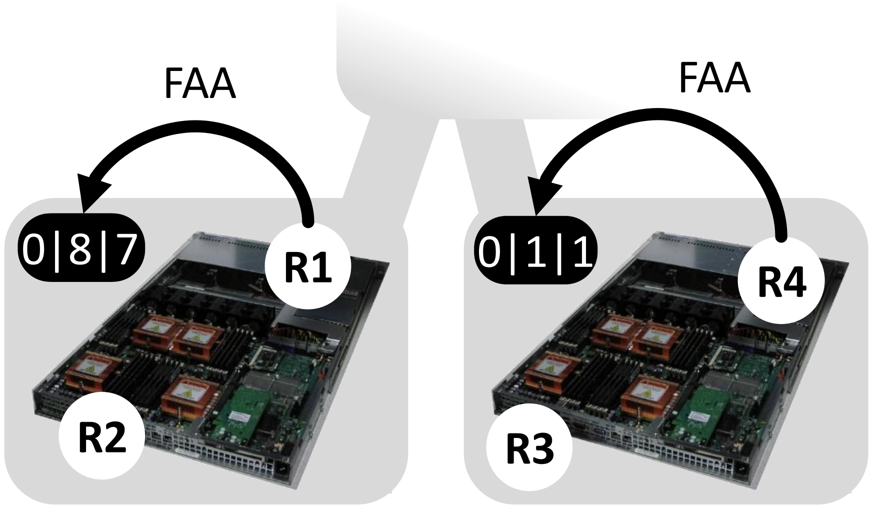
Readers that left the CS



Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

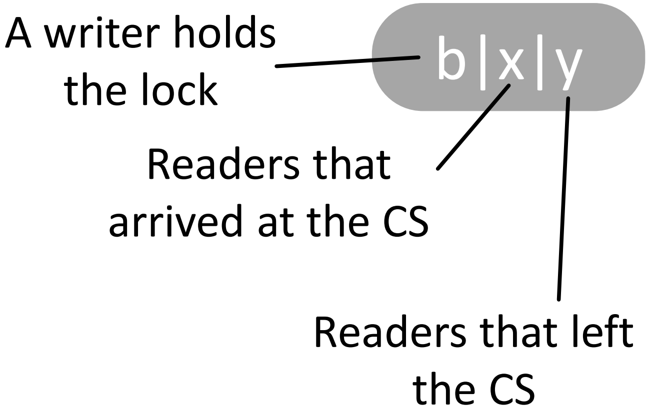


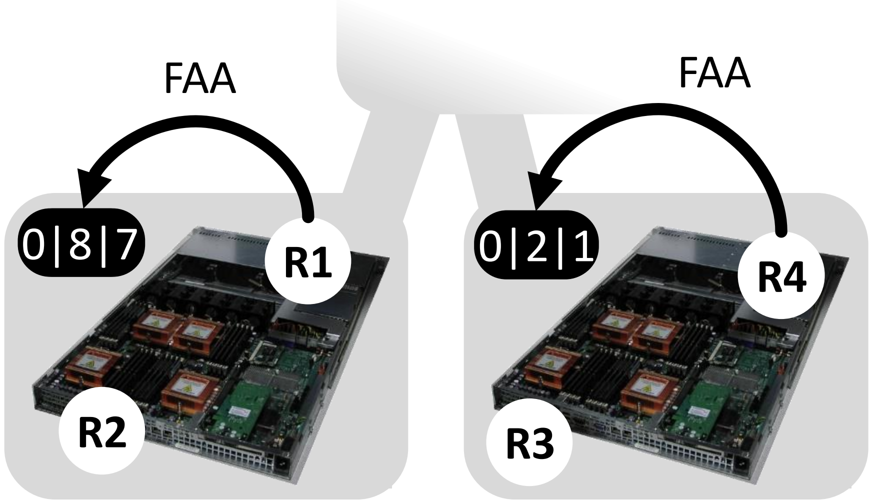


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Readers

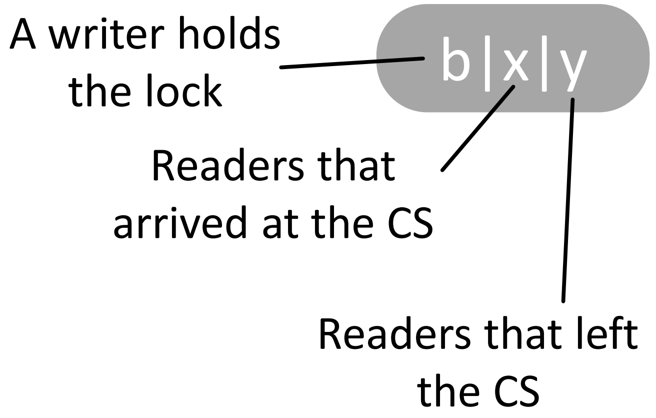
A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

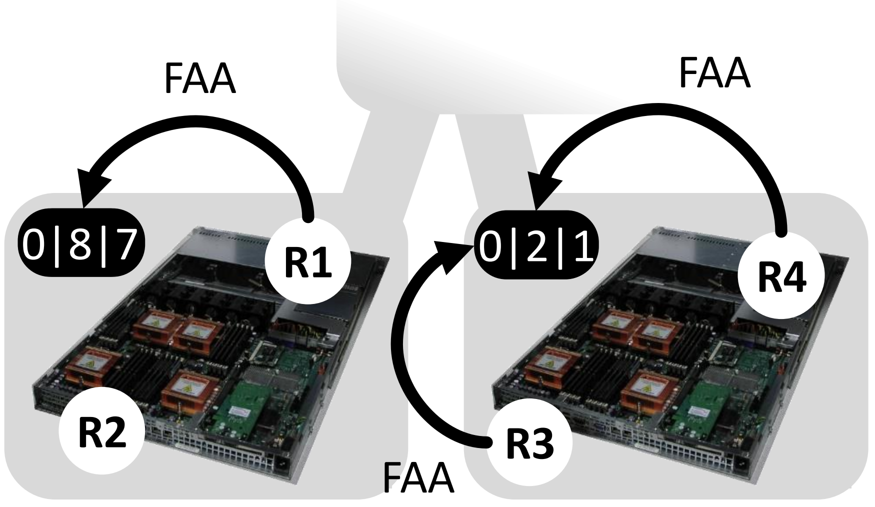




Lock Acquire by Readers

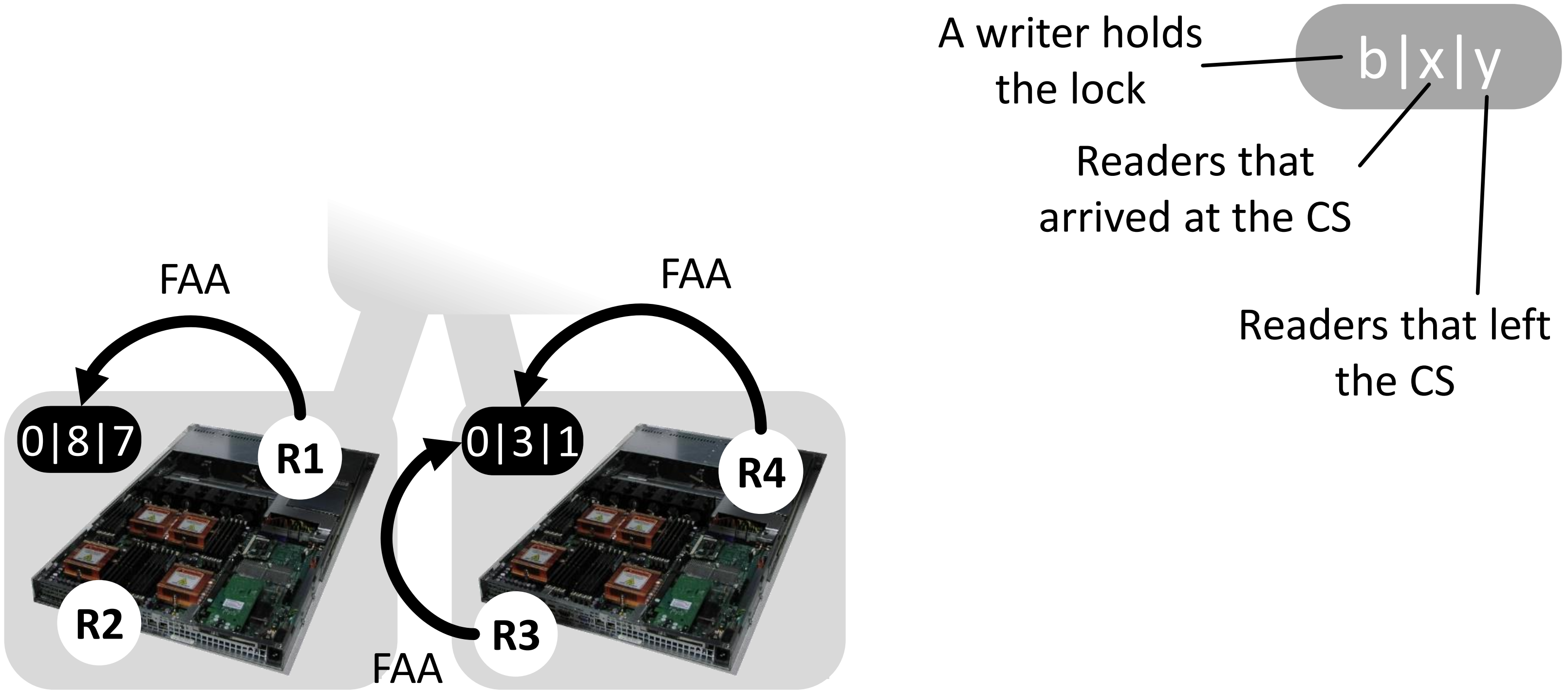
A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation





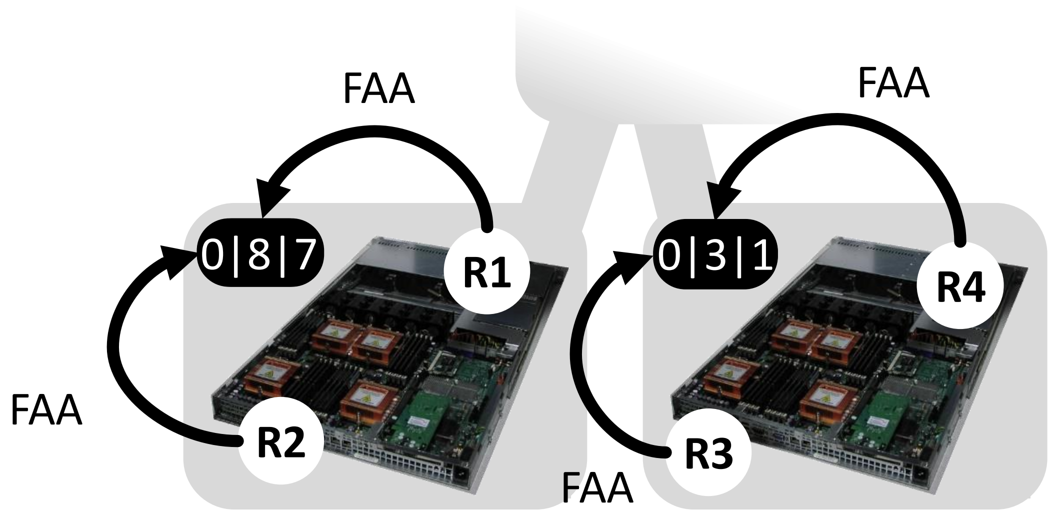
Lock Acquire by Readers

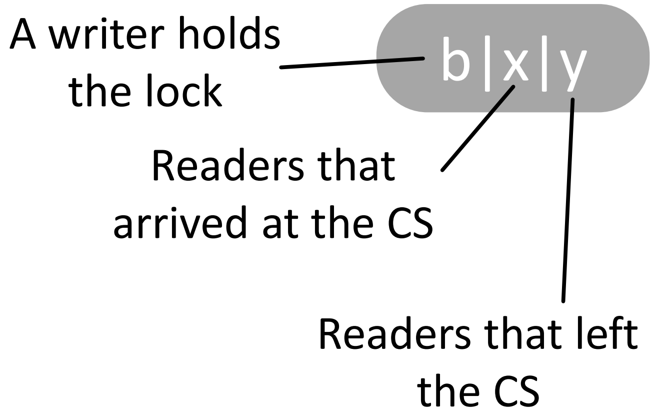
A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



Lock Acquire by Readers

! A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

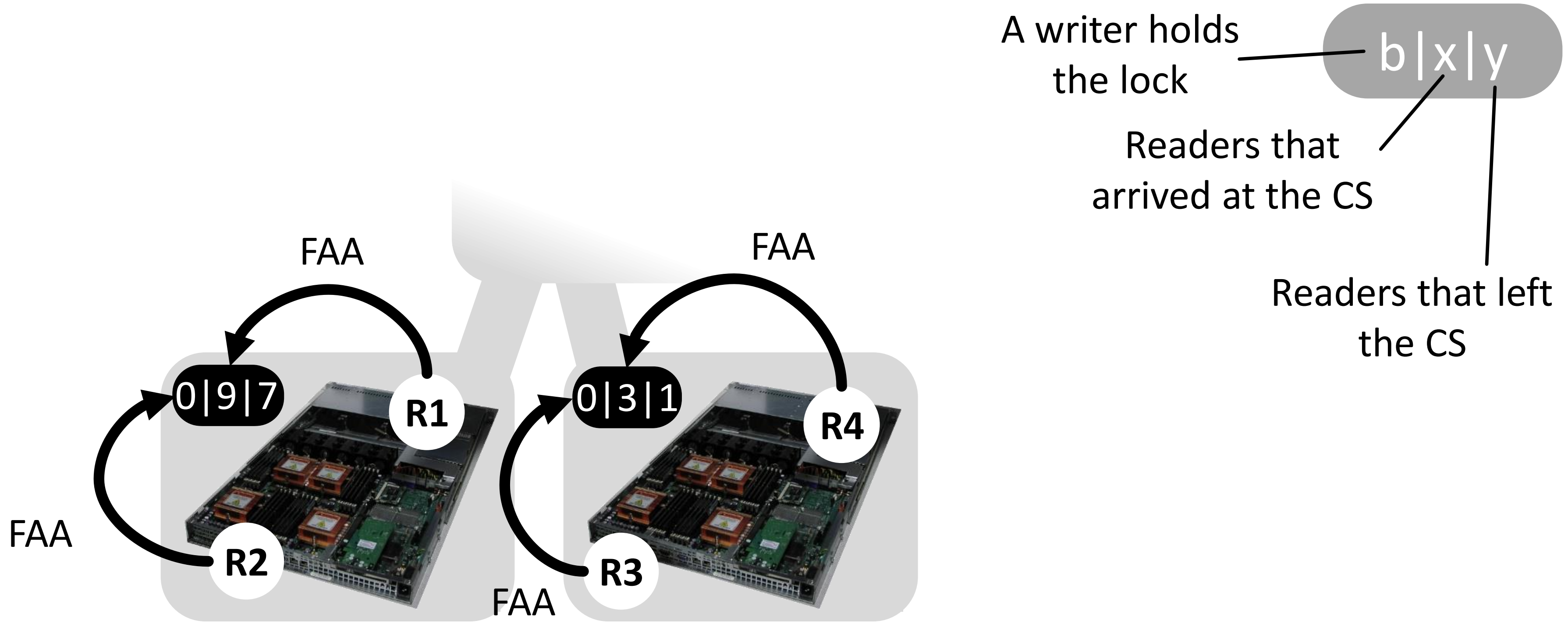




P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

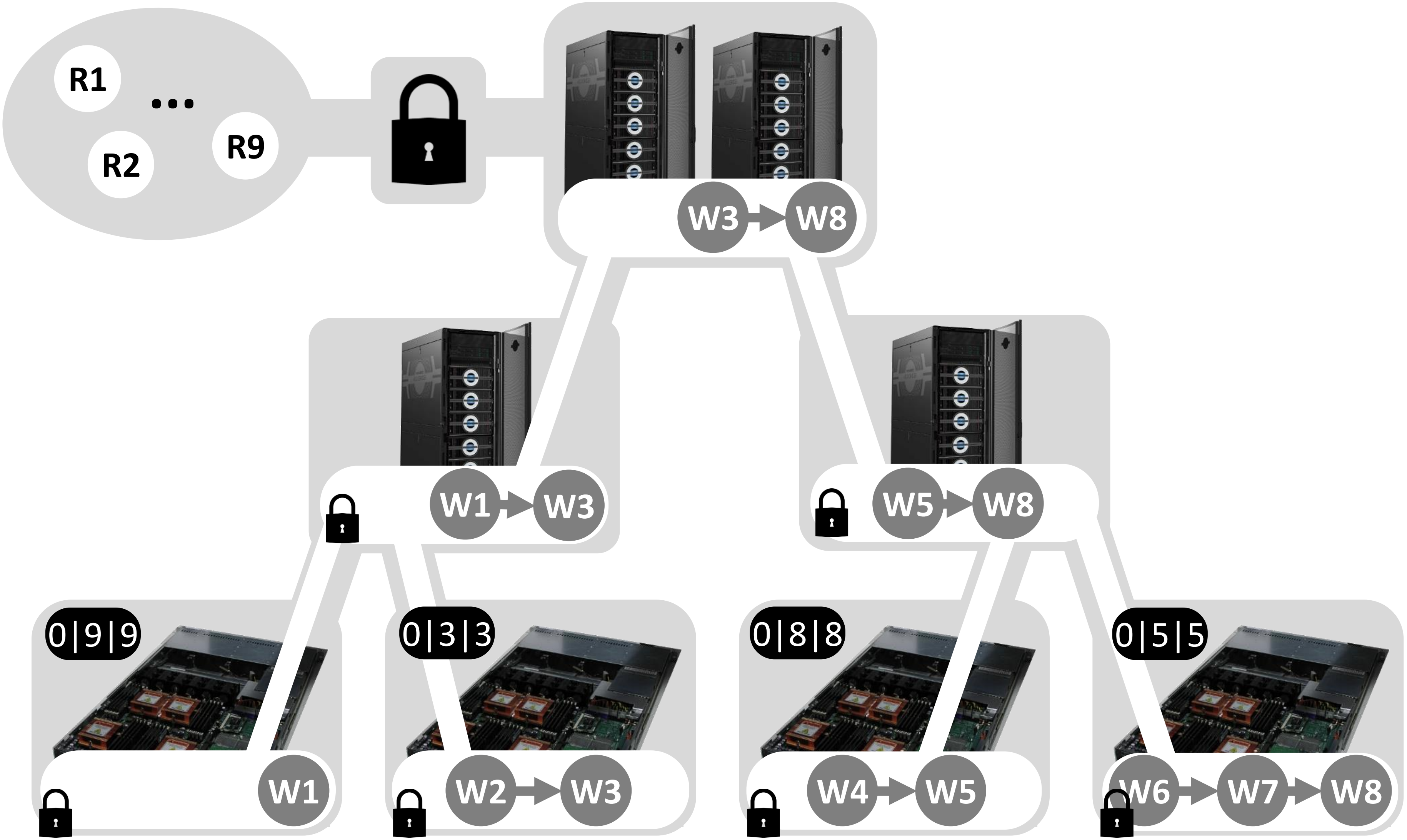
Lock Acquire by Readers

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation



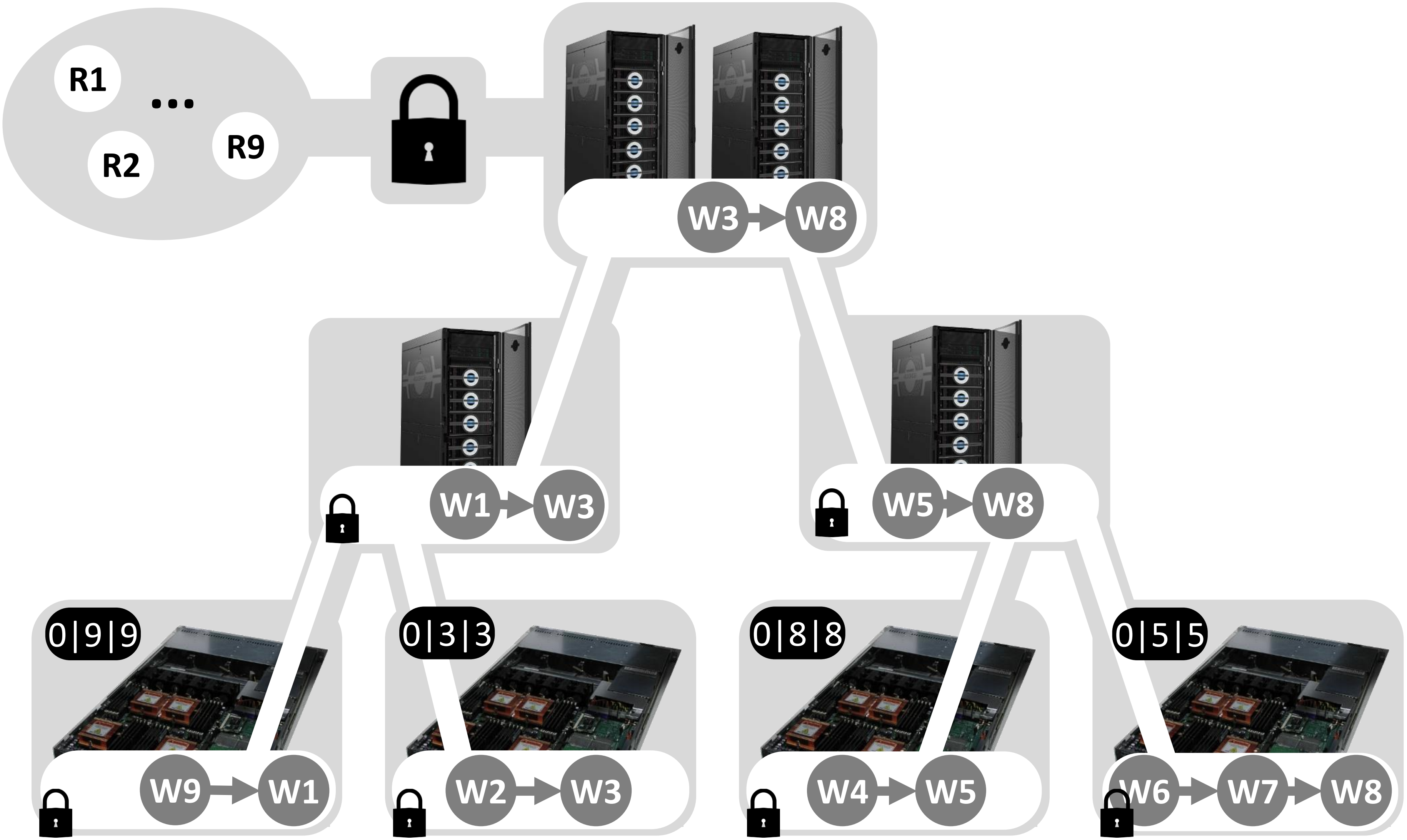
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Writers



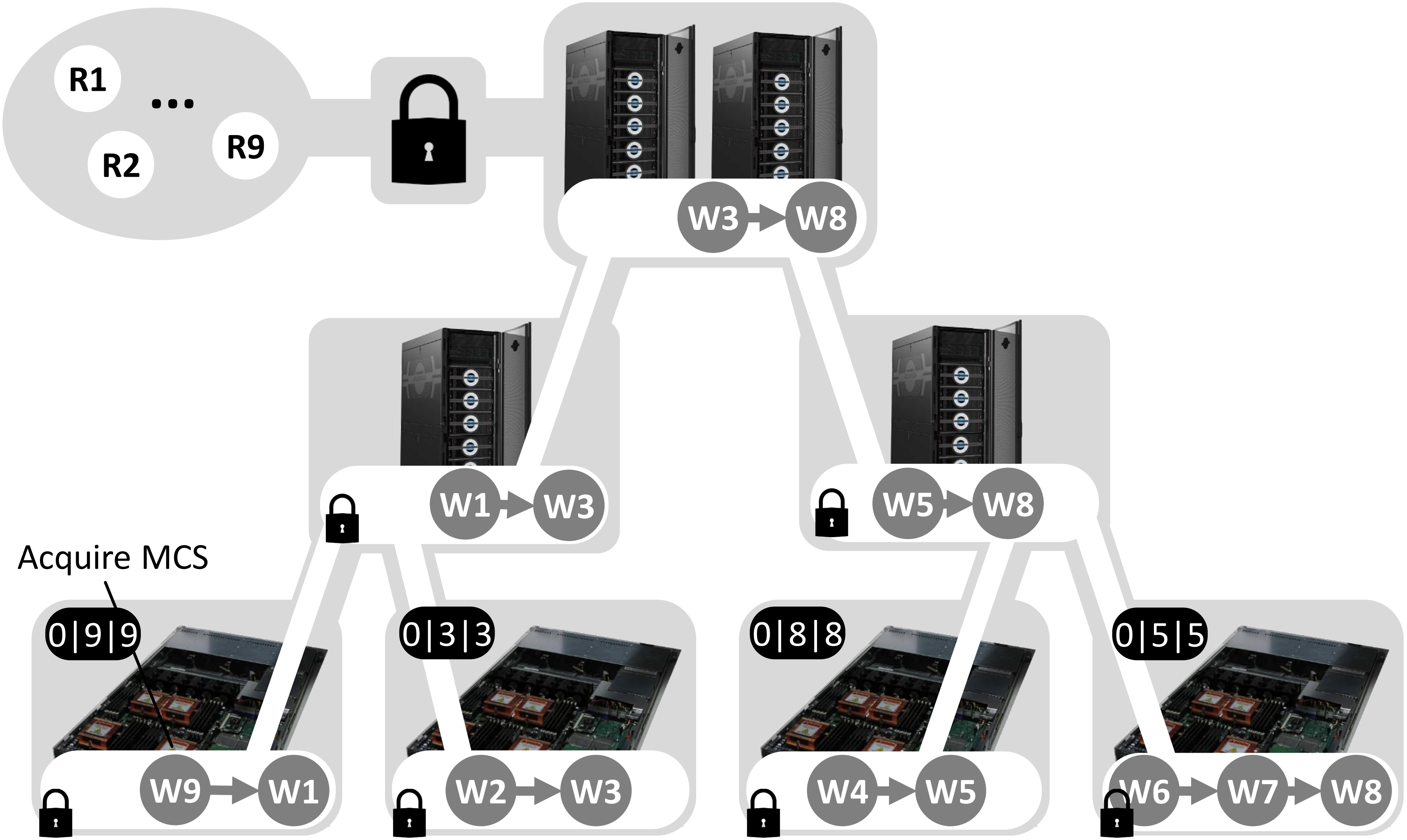
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Writers



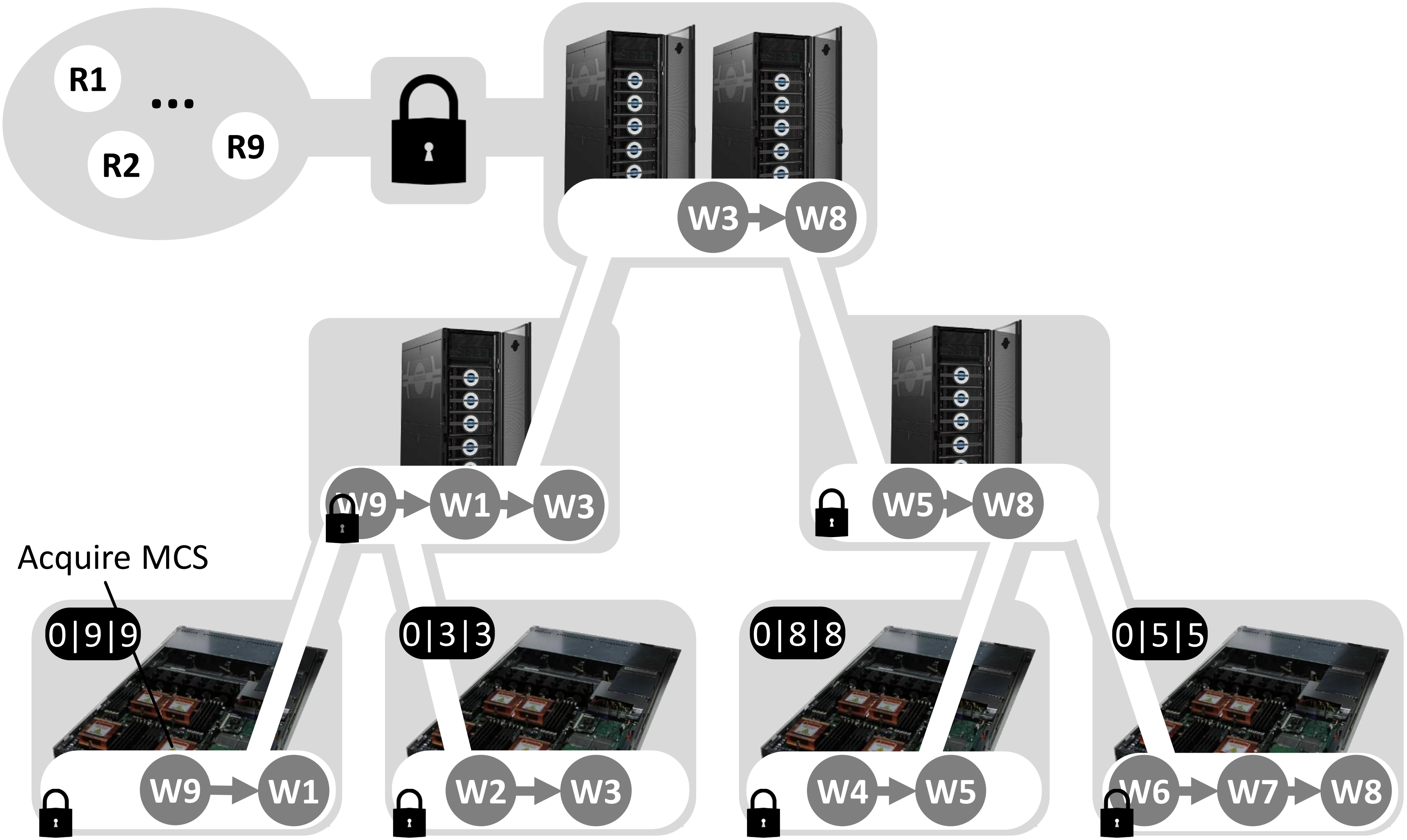
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Writers



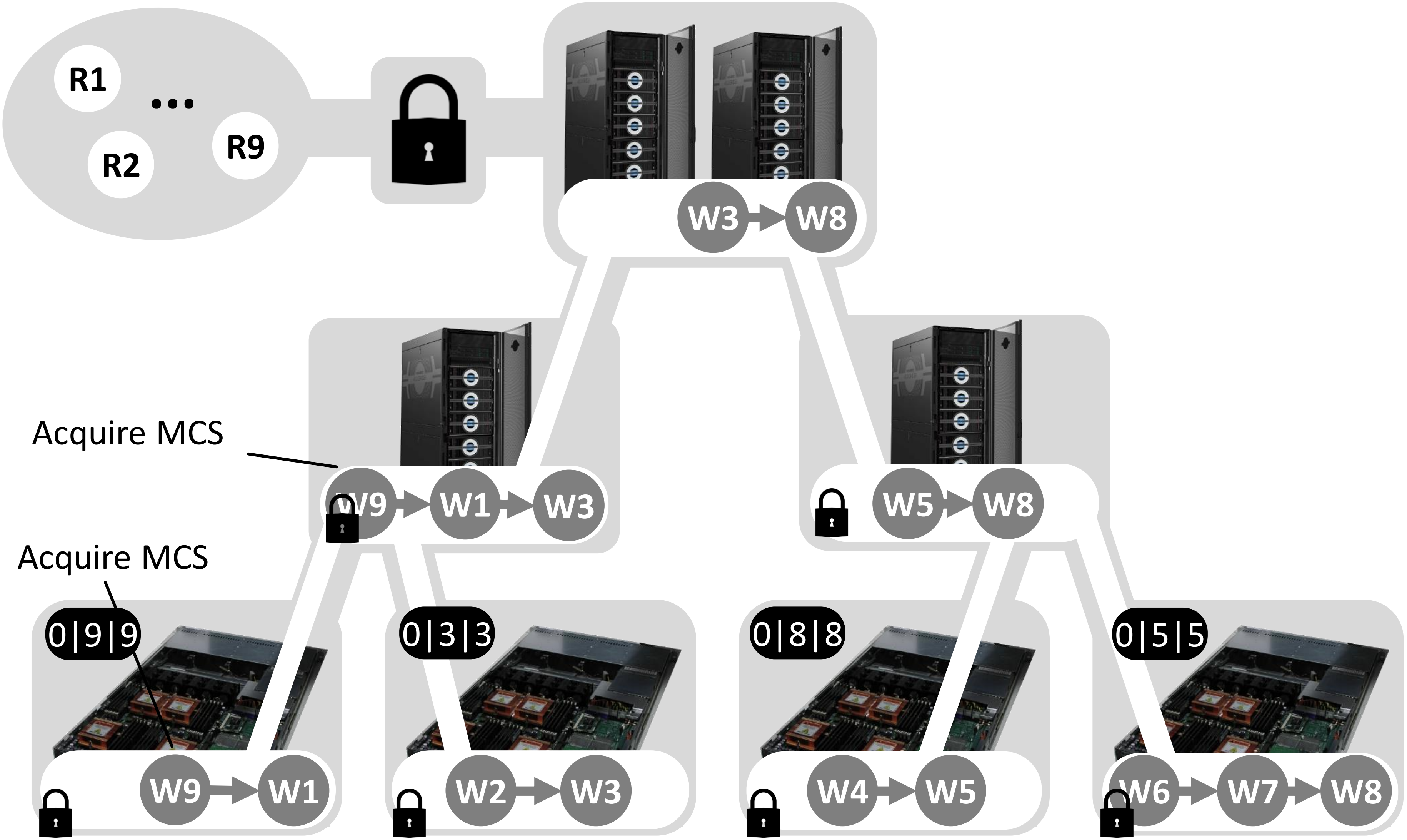
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC’16, best paper

Lock Acquire by Writers



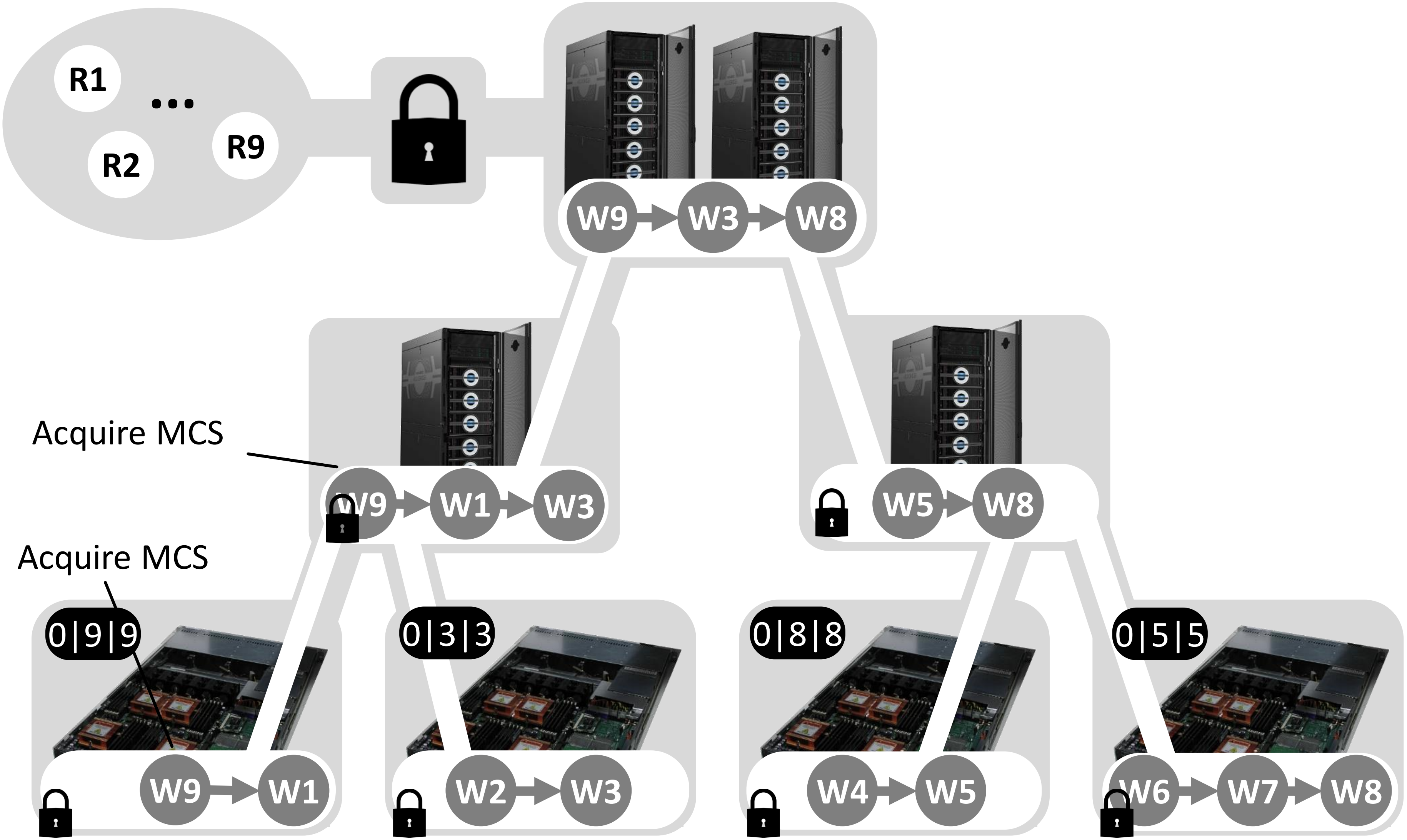
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Writers



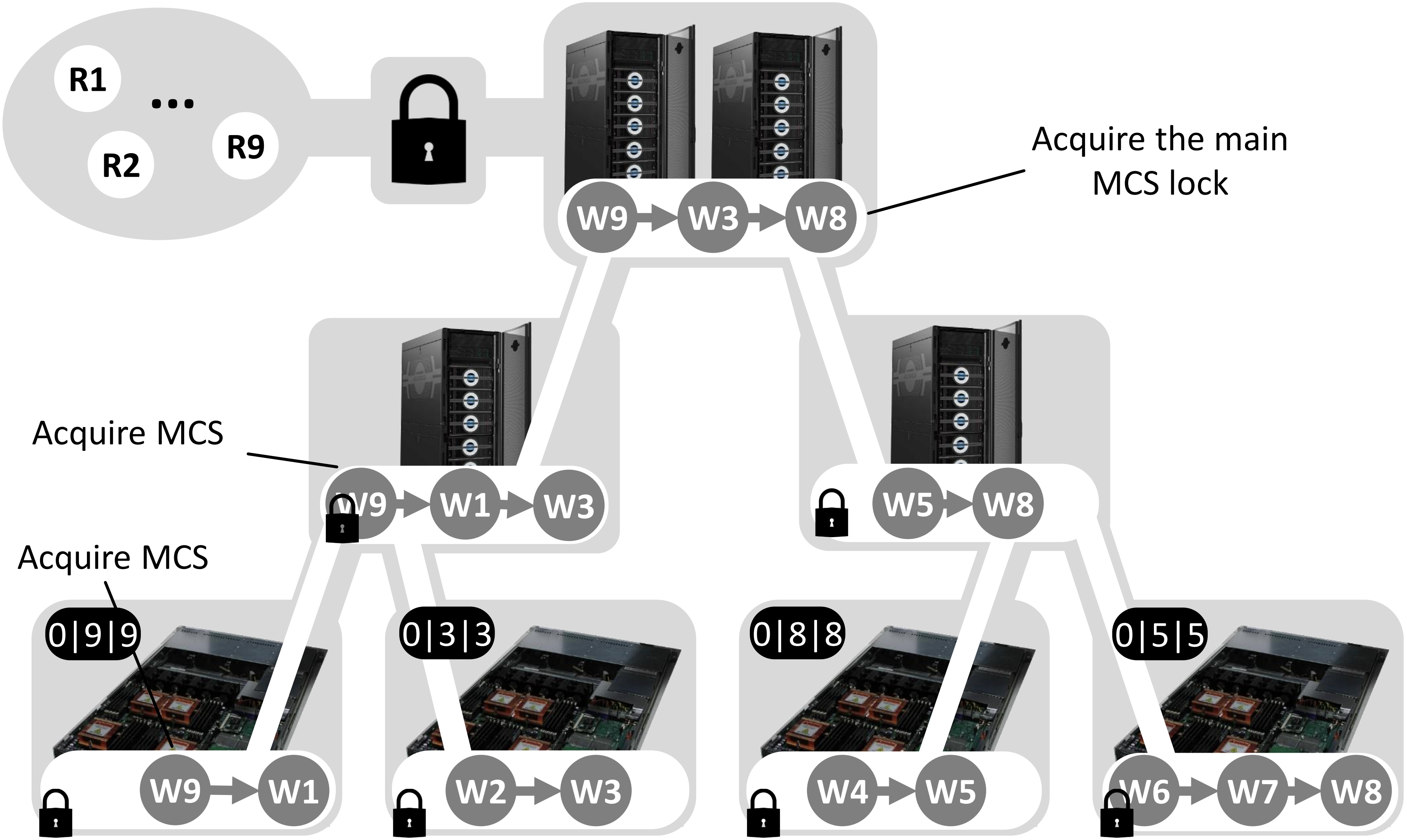
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Writers

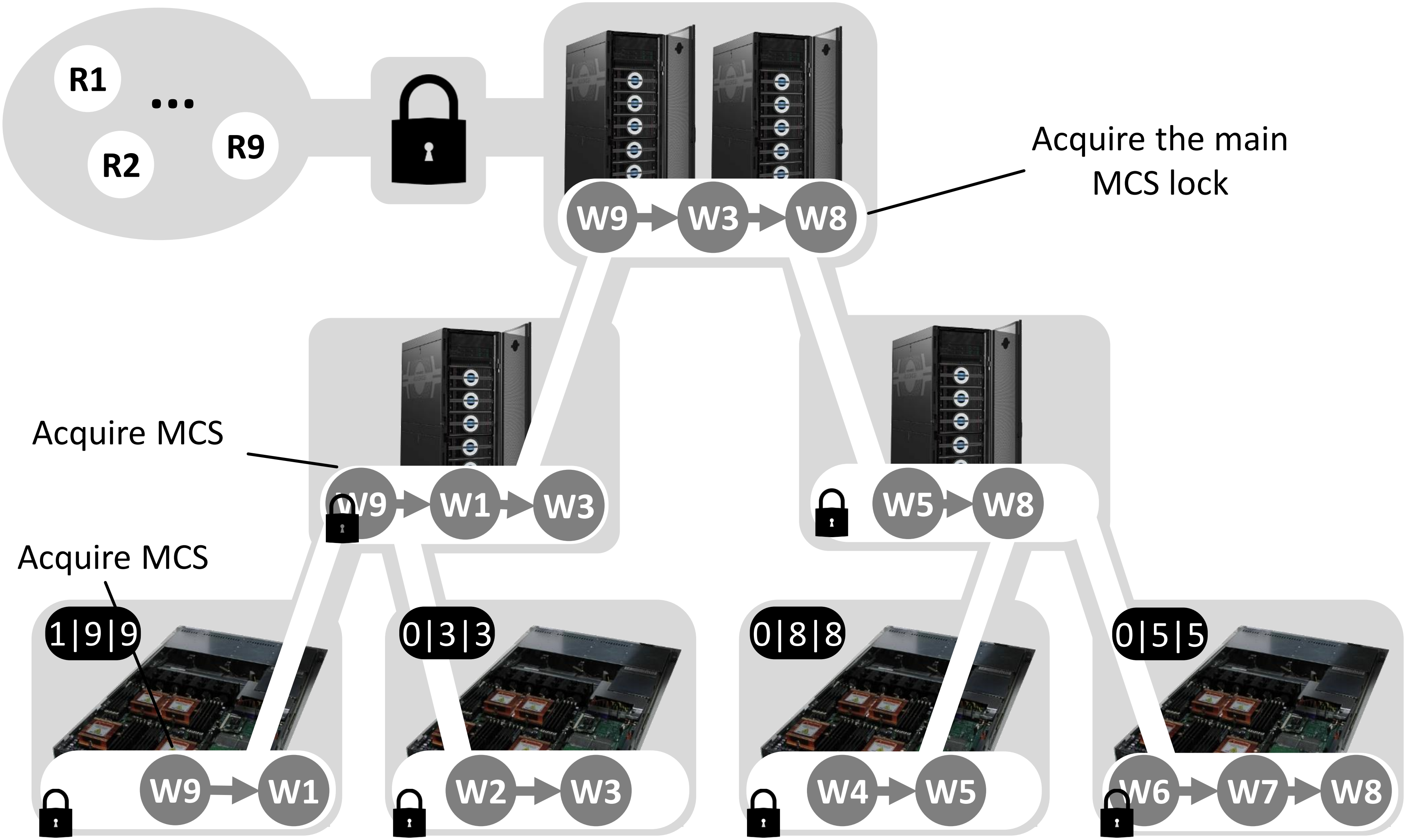


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

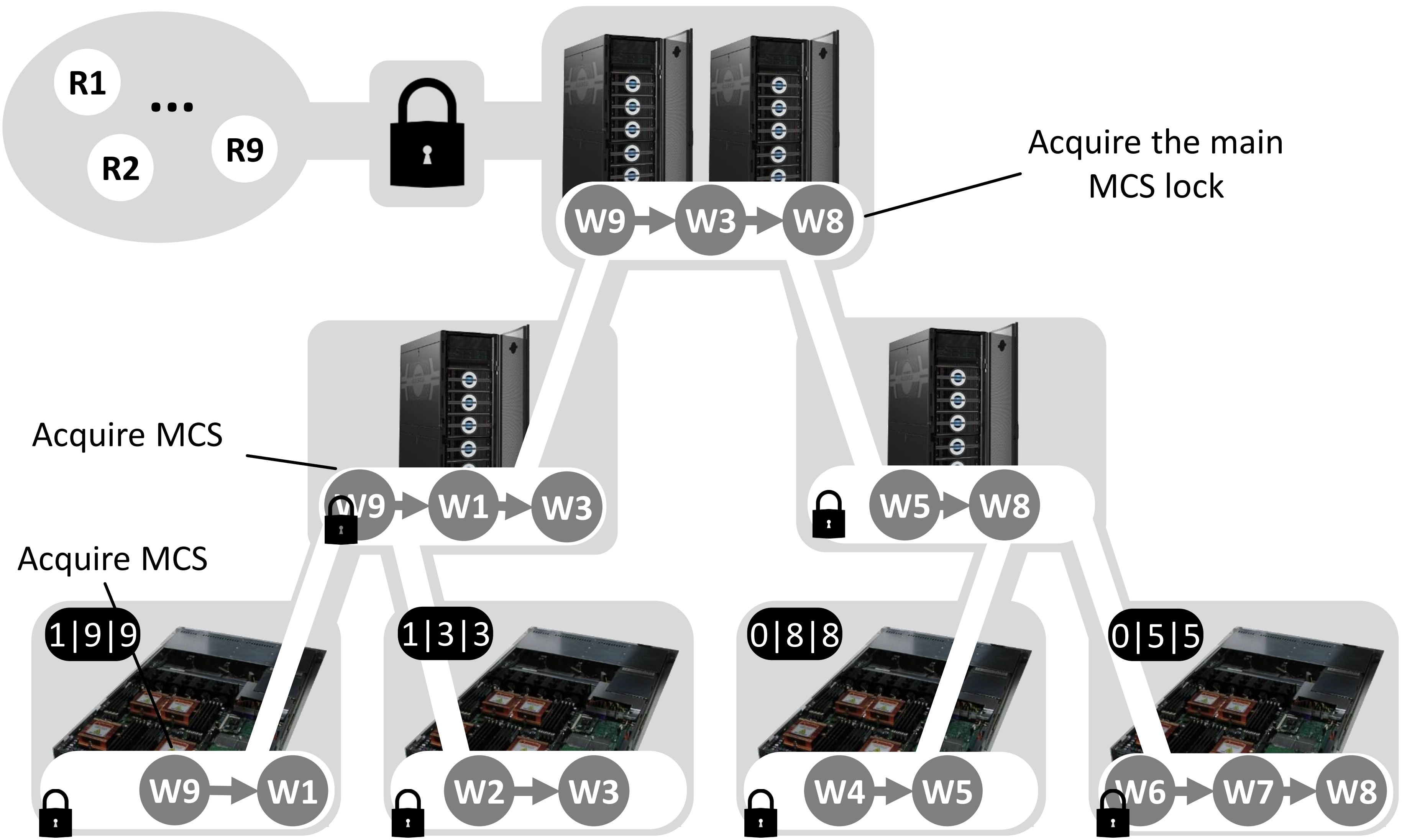
Lock Acquire by Writers



Lock Acquire by Writers

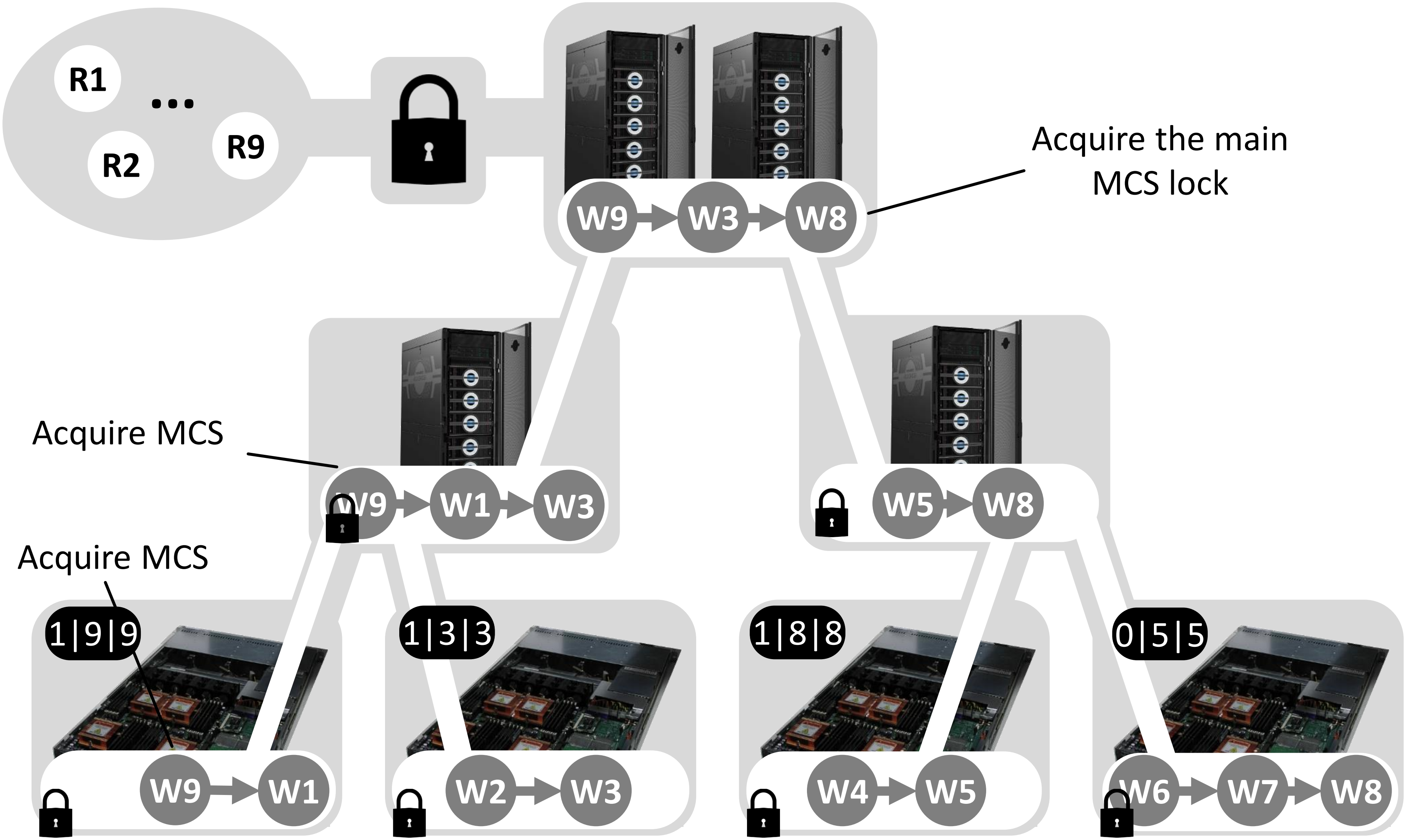


Lock Acquire by Writers



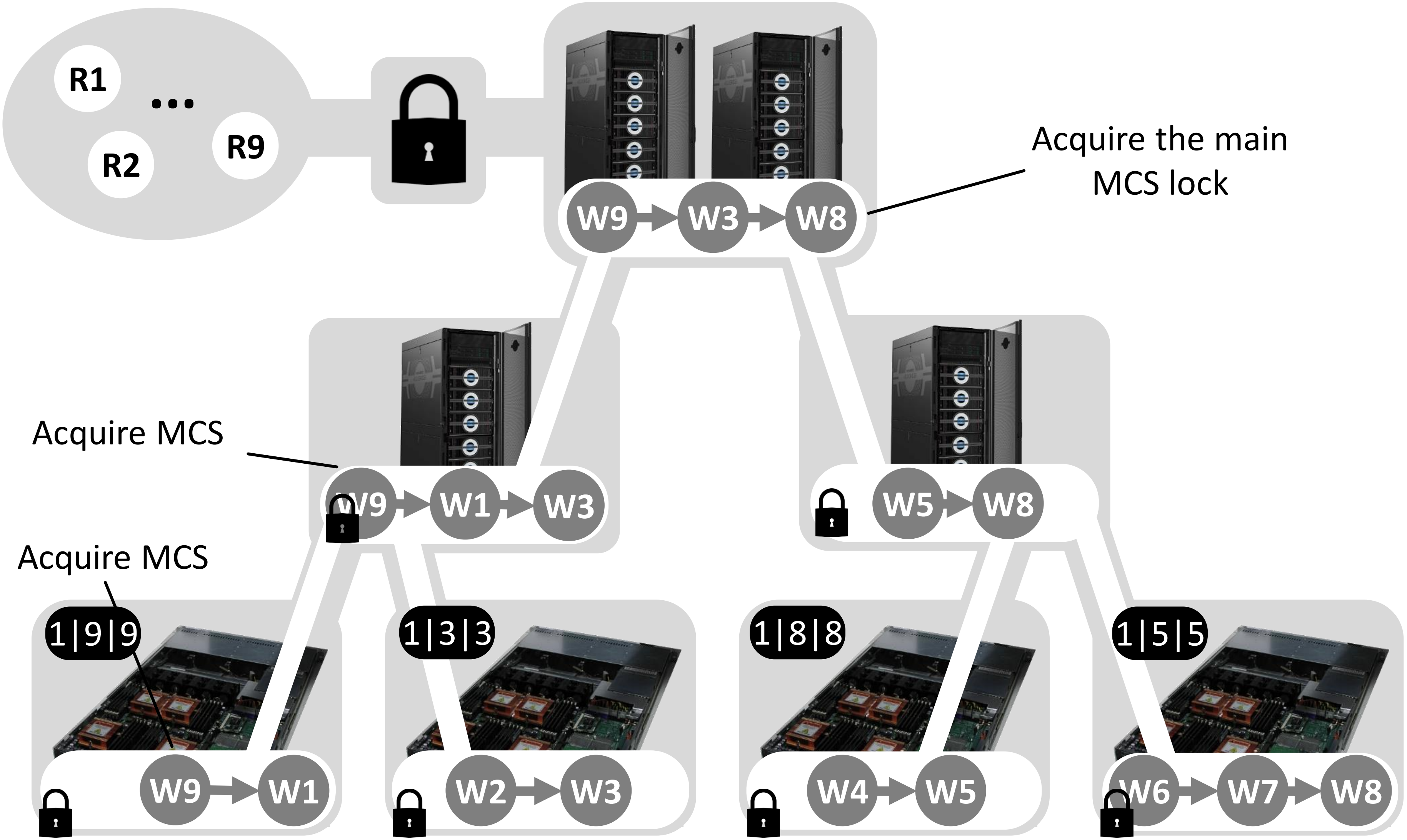
P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Writers

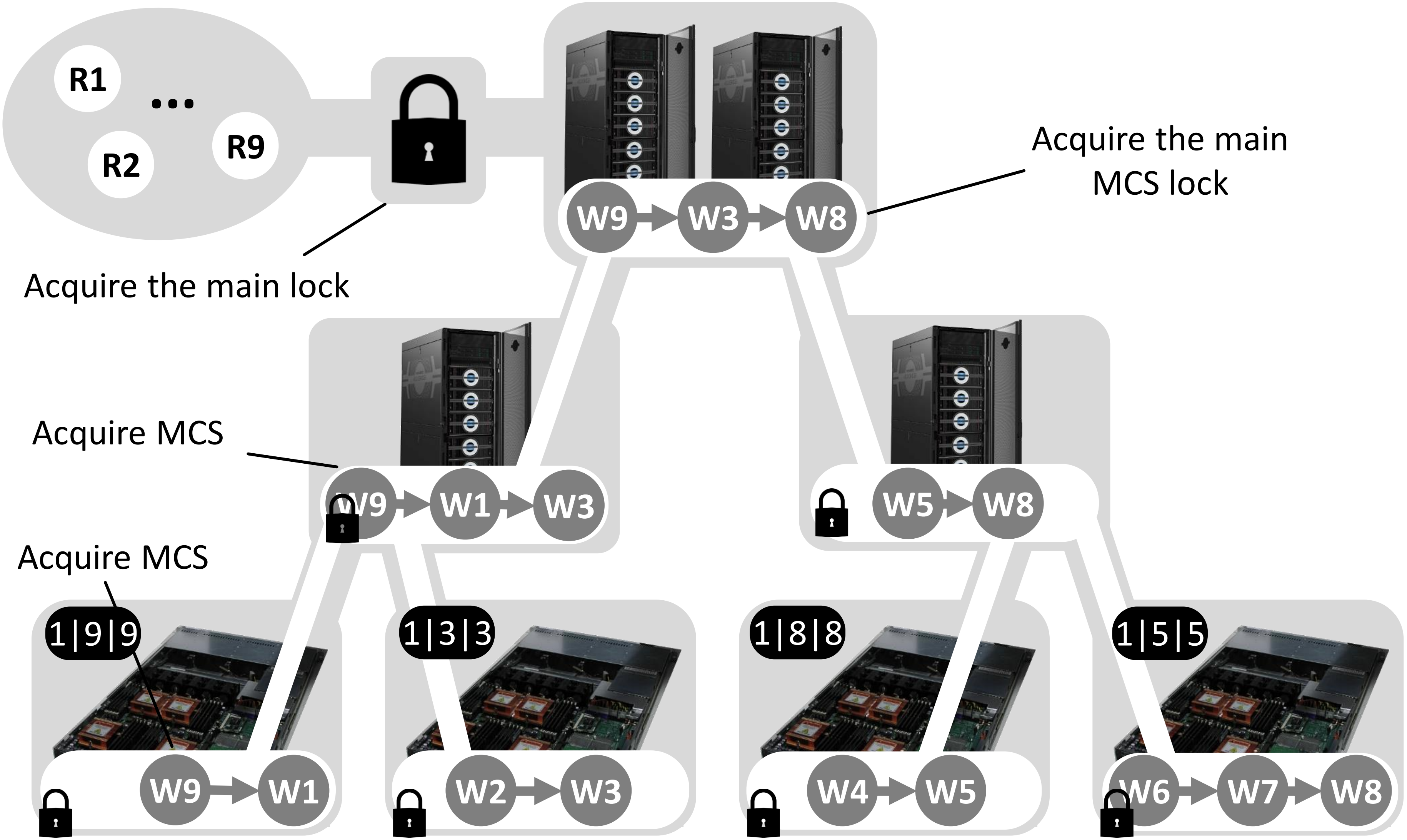


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Lock Acquire by Writers



Lock Acquire by Writers



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Evaluation

- CSCS Piz Daint (Cray XC30)
- 5272 compute nodes
- 8 cores per node
- 169TB memory

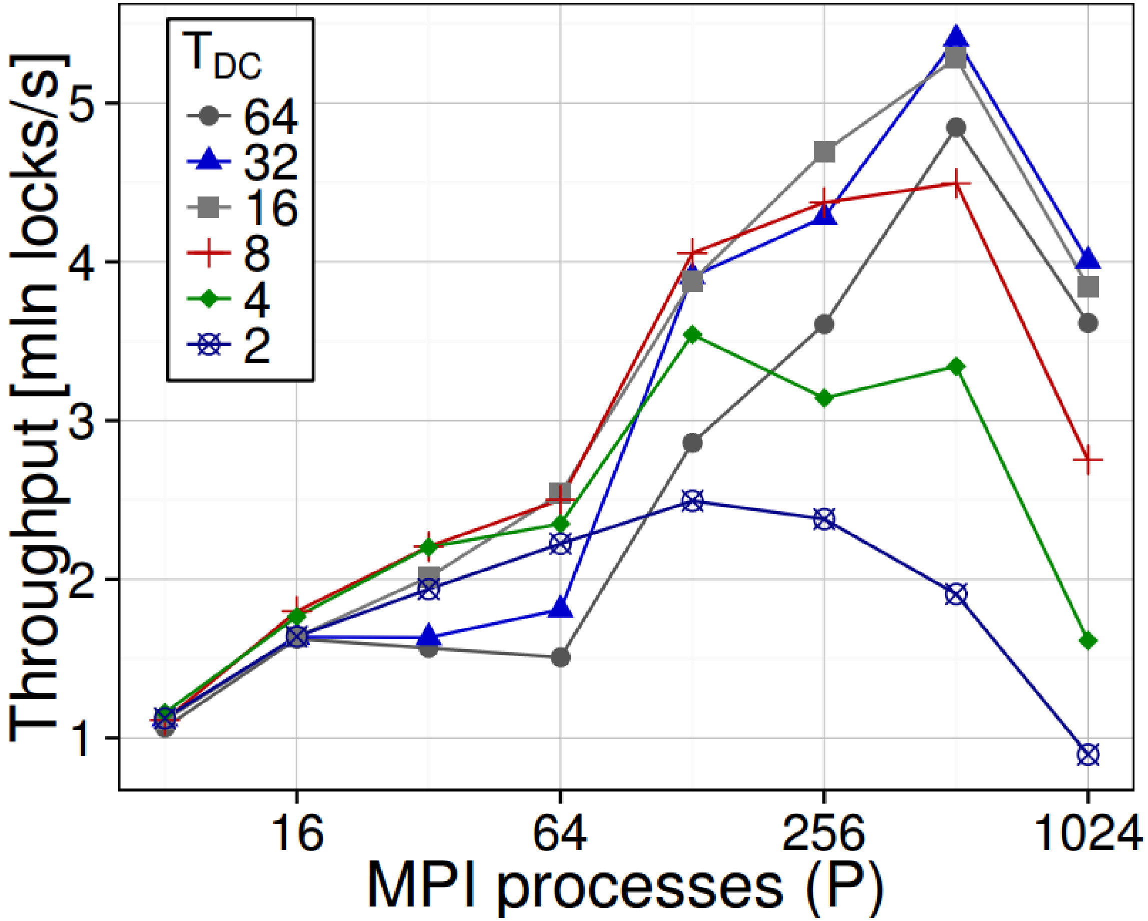
- Microbenchmarks: acquire/release: latency, throughput
- Distributed hashtable

Evaluation - Distributed Counter Analysis

0|12|8

Throughput, 2% writers

Single-operation benchmark

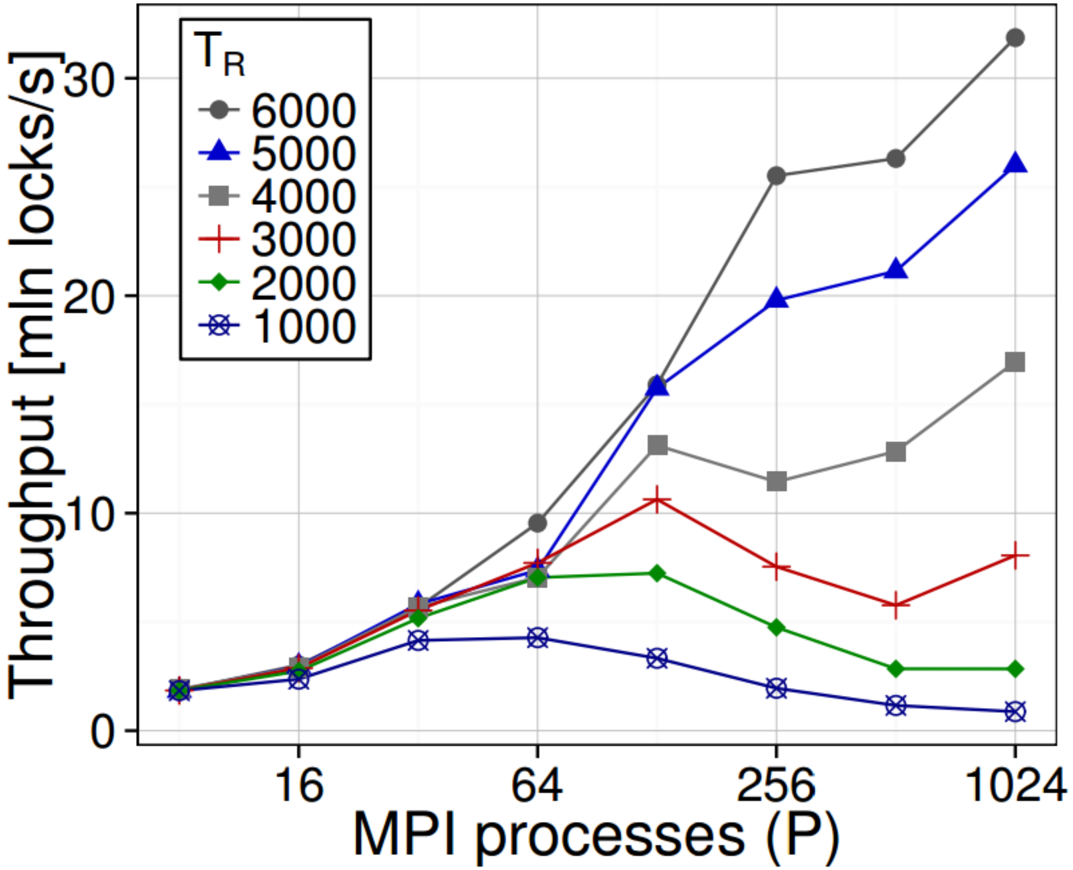


P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

Evaluation - Reader Threshold Analysis

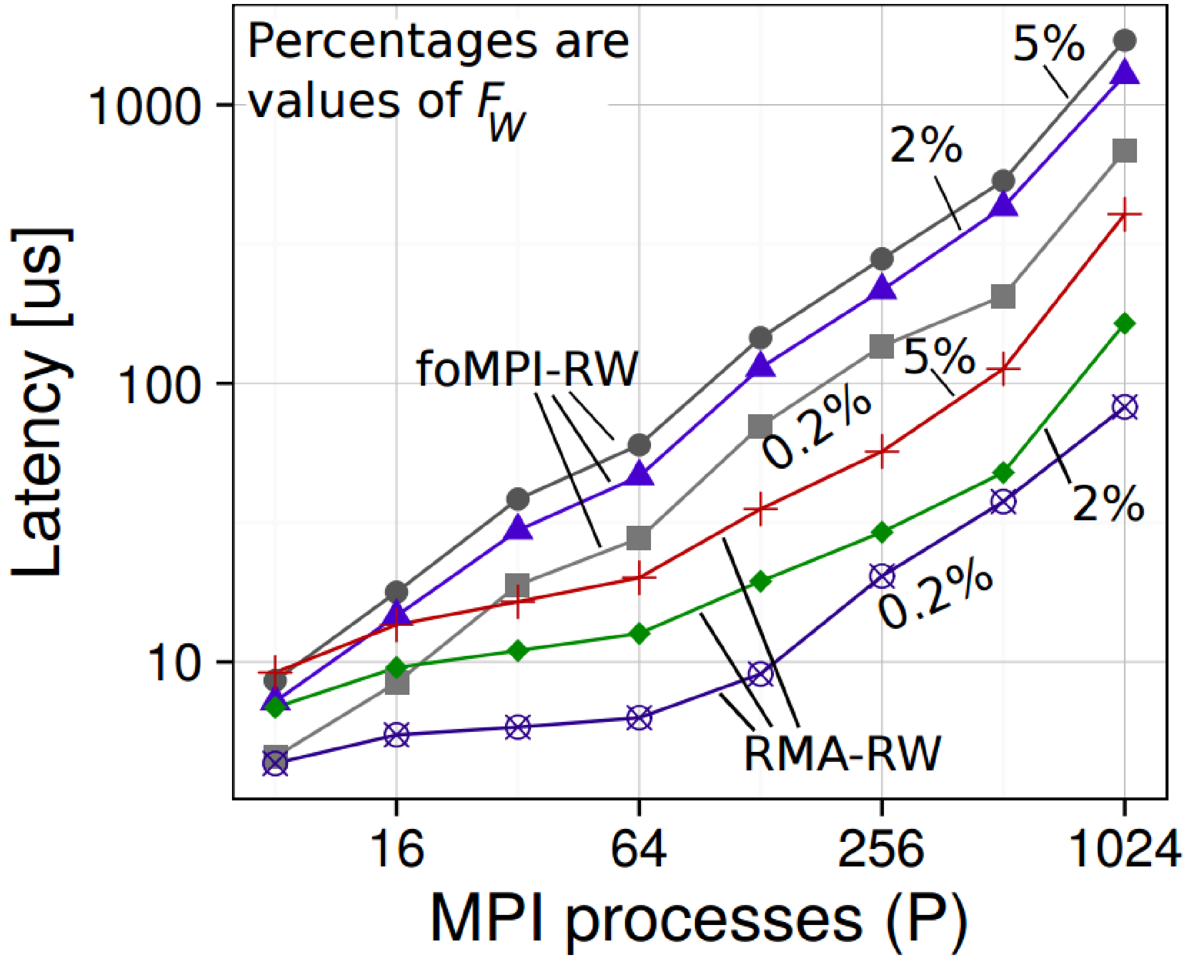
Throughput, 0.2% writers,

Empty-critical-section benchmark



P. Schmid, M. Besta, TH: High-Performance Distributed RMA Locks, ACM HPDC'16, best paper

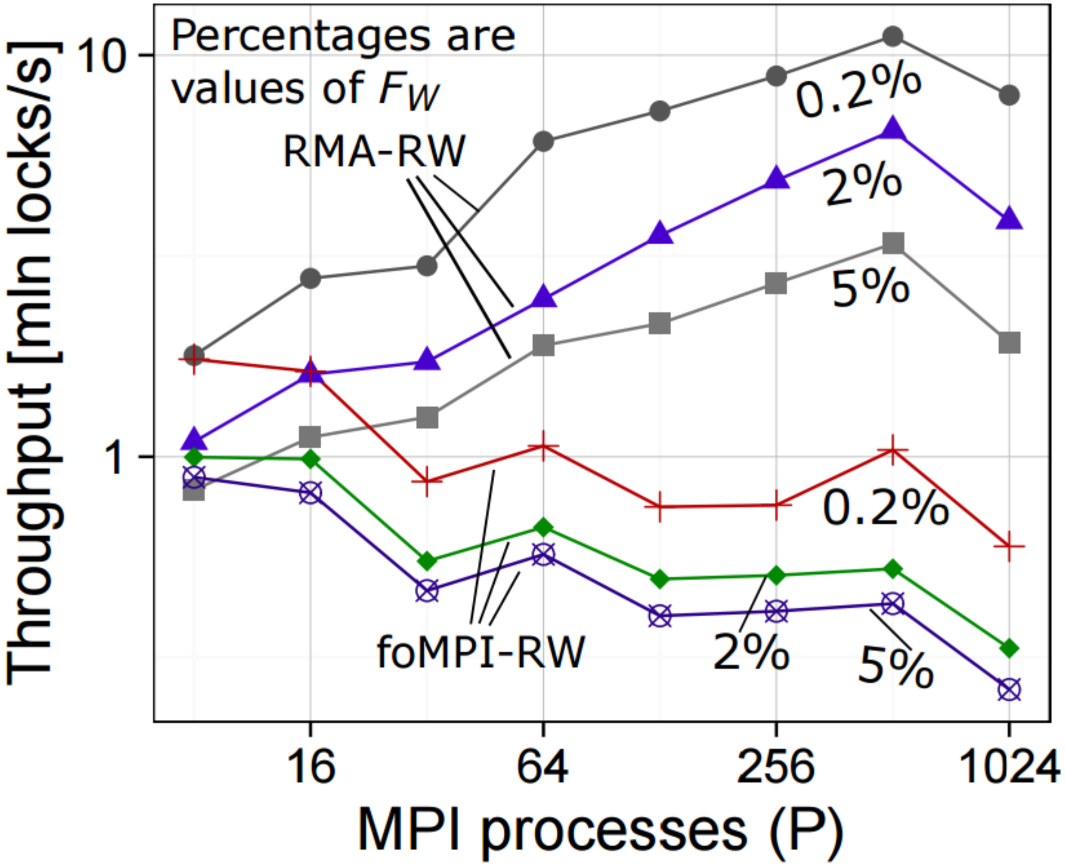
Evaluation - Comparison to the State-of-the-Art



[1] R. Gerstenberger et al. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. ACM/IEEE Supercomputing 2013.

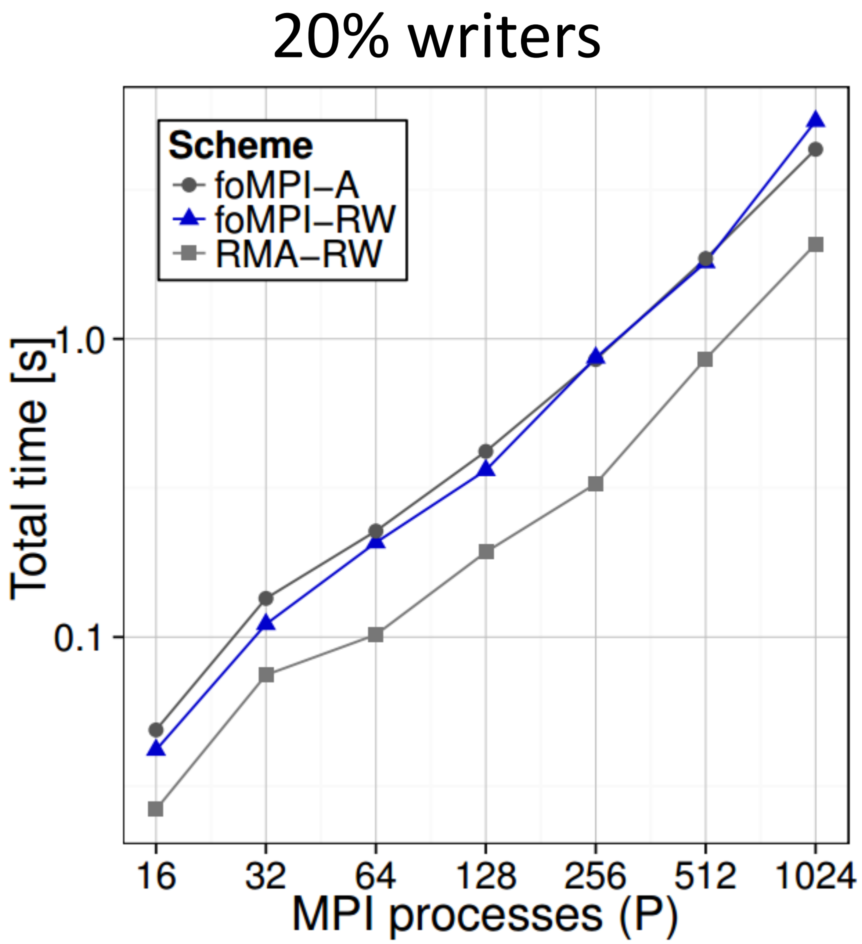
Evaluation - Comparison to the State-of-the-Art

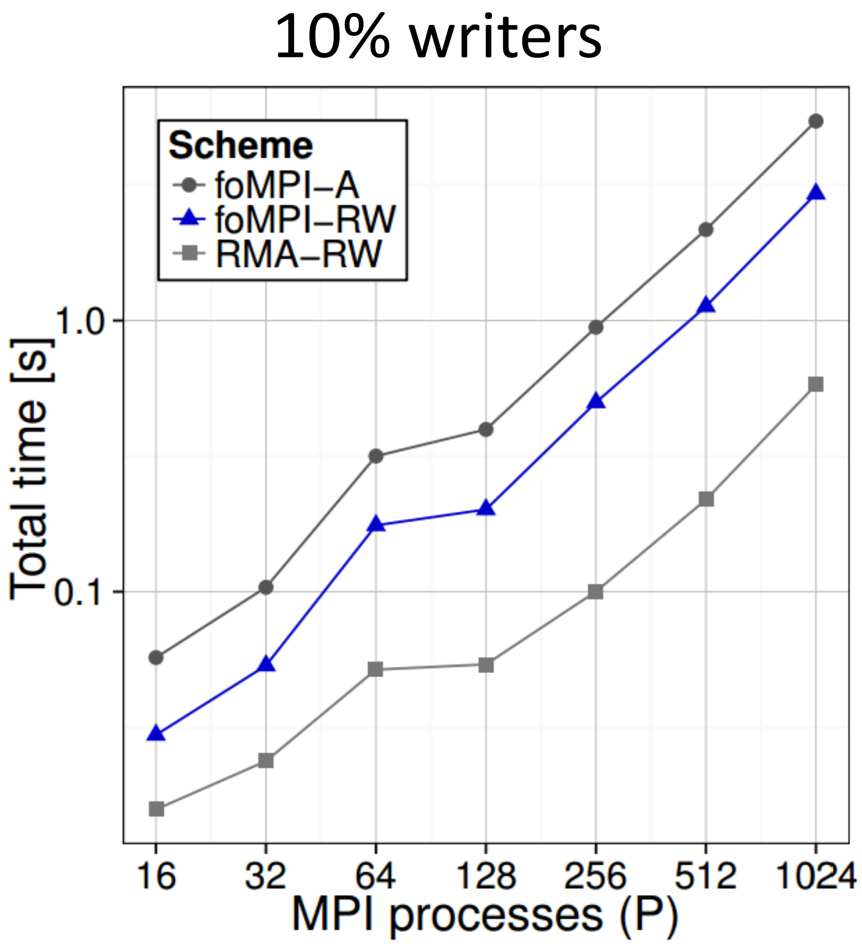
Throughput, single-operation benchmark



[1] R. Gerstenberger et al. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. ACM/IEEE Supercomputing 2013.

Evaluation - Distributed Hashtable

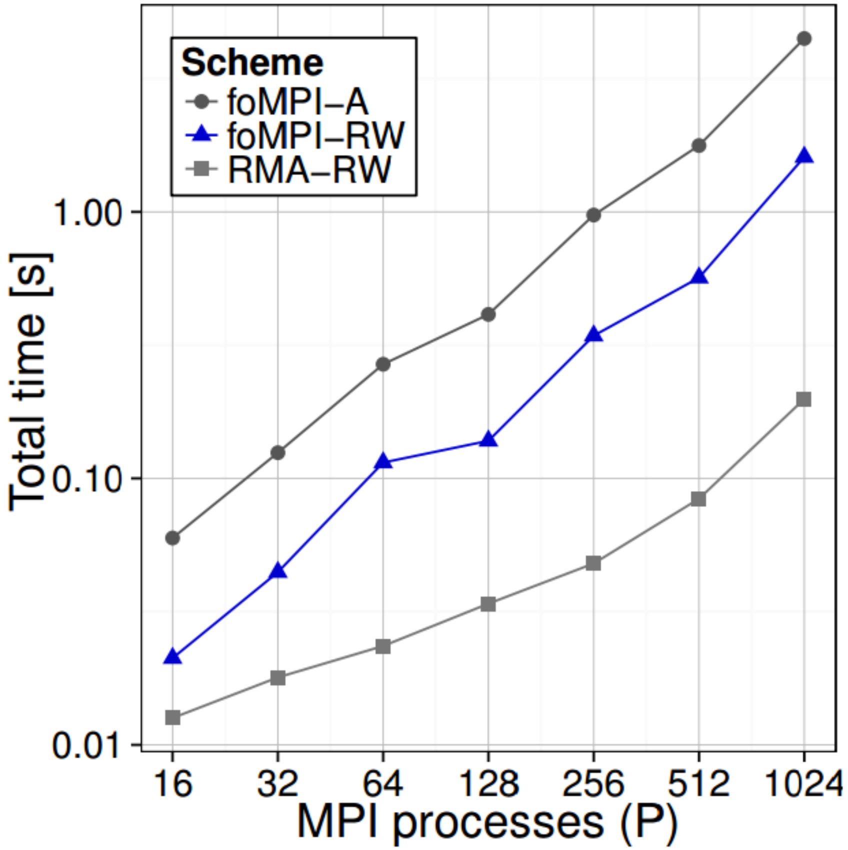




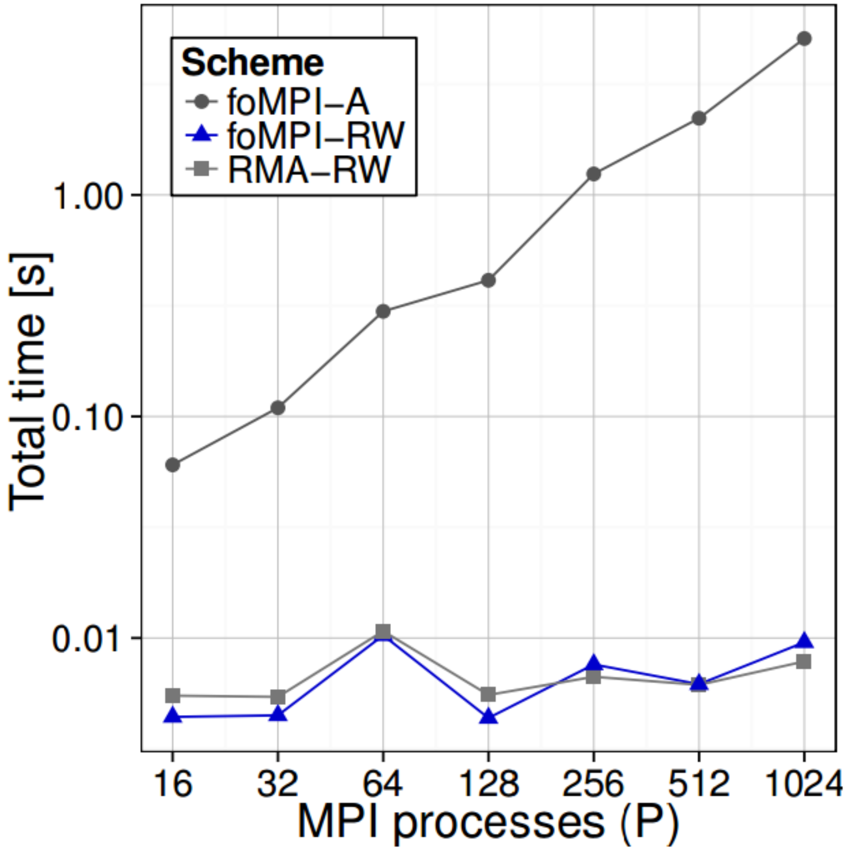
[1] R. Gerstenberger et al. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. ACM/IEEE Supercomputing 2013.

Evaluation - Distributed Hashtable

2% of writers



0% of writers

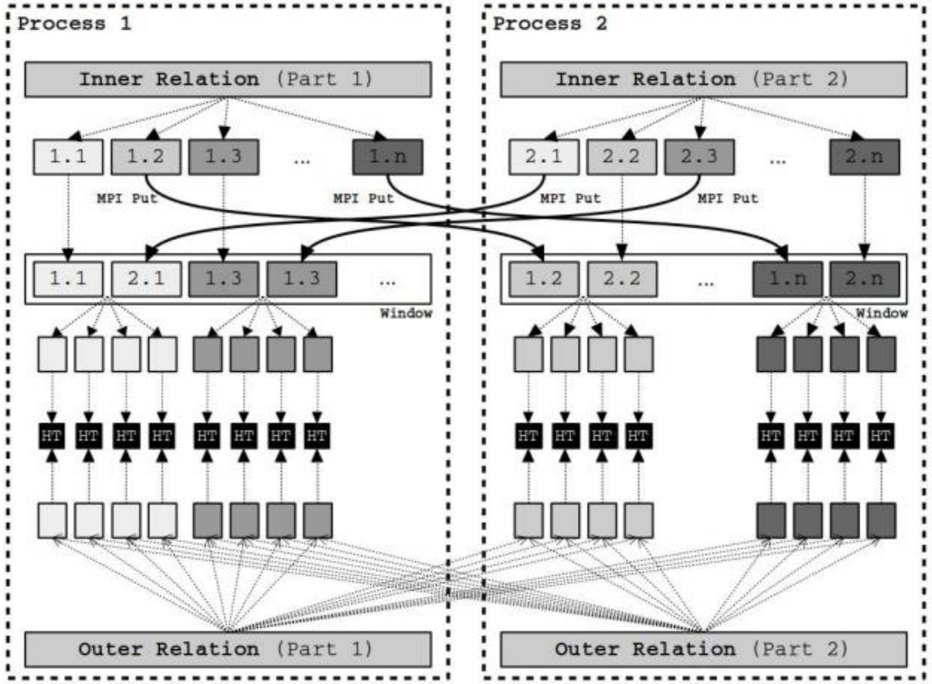


[1] R. Gerstenberger et al. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. ACM/IEEE Supercomputing 2013.

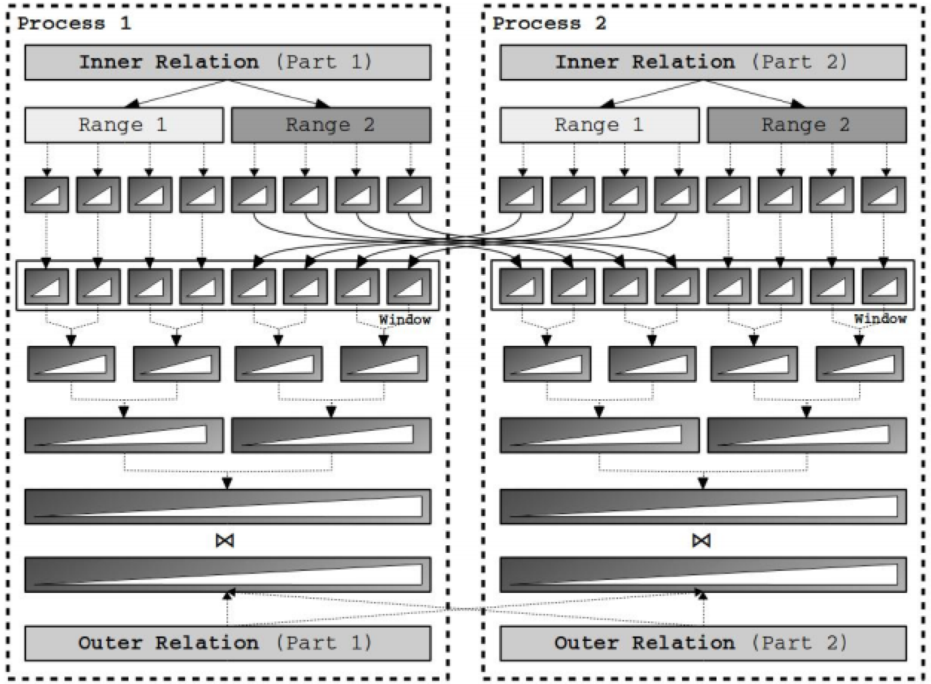
Another application area - Databases

- **MPI-RMA for distributed databases?**

Hash-Join



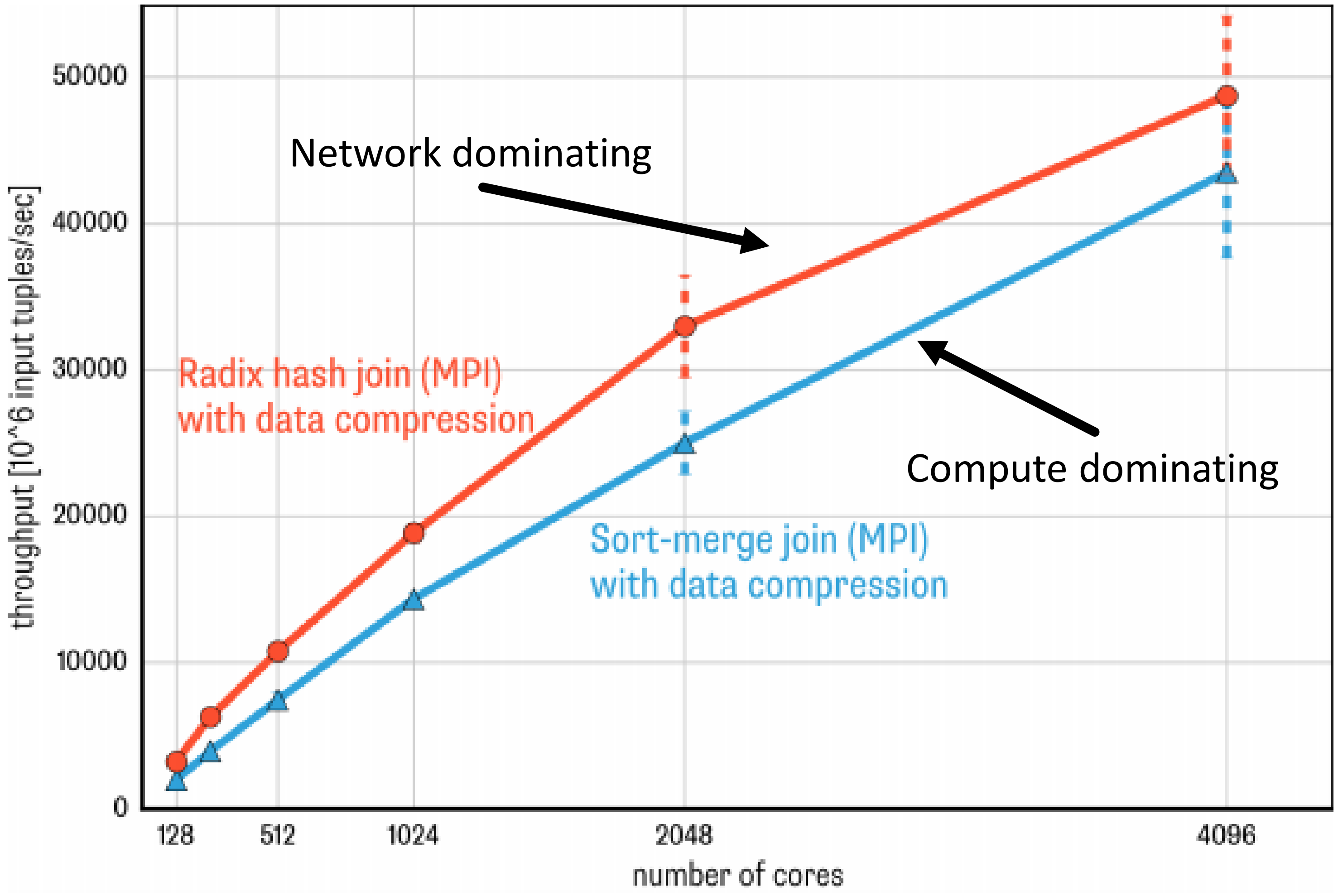
Sort-Join



C. Barthels, et al., TH: Distributed Join Algorithms on Thousands of Cores presented in Munich, Germany, VLDB Endowment, Aug. 2017

Another application area - Databases

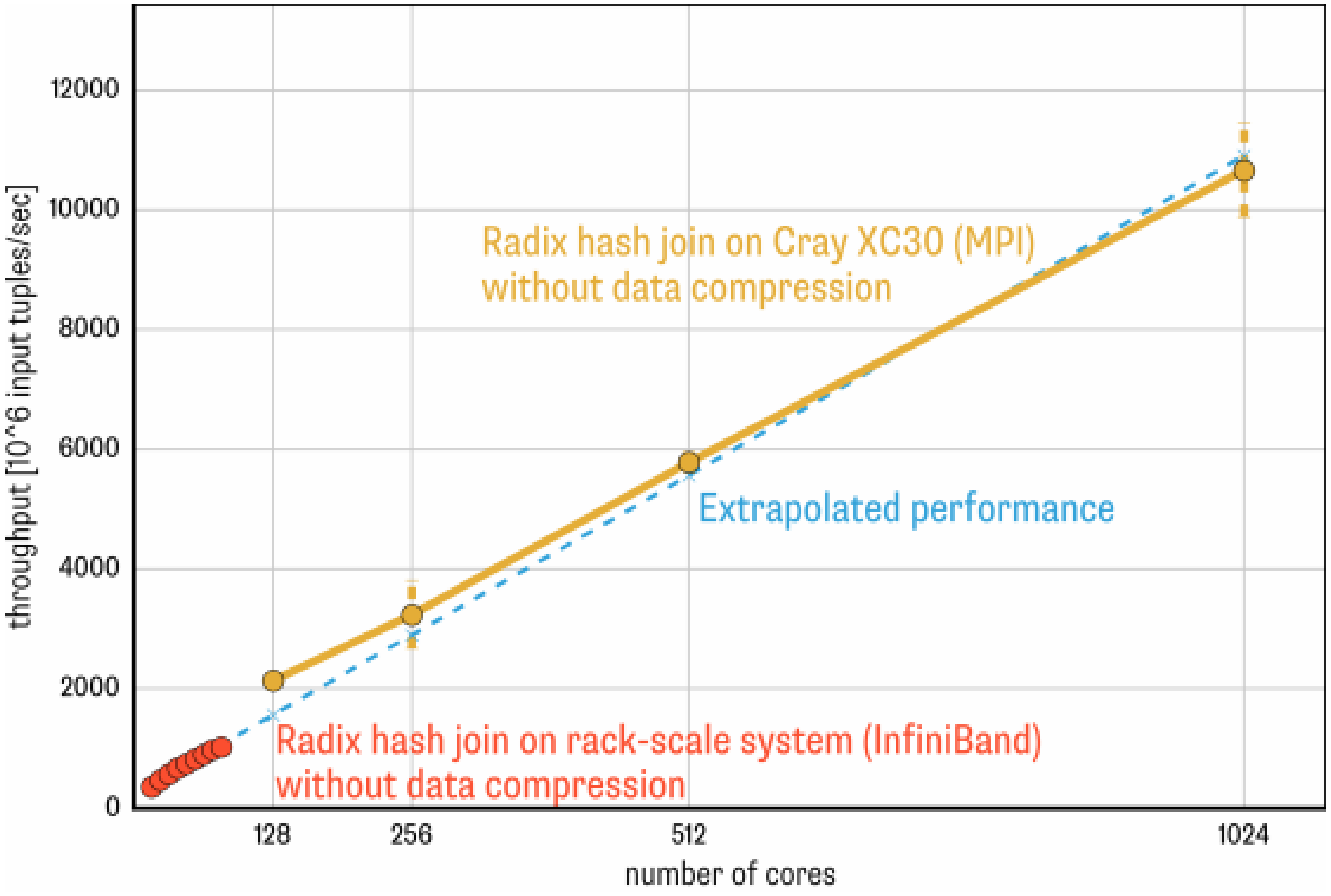
- **MPI-RMA for distributed databases on Piz Daint**



C. Barthels, et al., TH: Distributed Join Algorithms on Thousands of Cores presented in Munich, Germany, VLDB Endowment, Aug. 2017

Another application area - Databases

- **MPI-RMA for distributed databases on Piz Daint**



C. Barthels, et al., TH: Distributed Join Algorithms on Thousands of Cores presented in Munich, Germany, VLDB Endowment, Aug. 2017

Now on to parallel algorithms!

- **Oblivious parallel algorithms**
  - Fixed structure work-depth graphs
- **Nonoblivious parallel algorithms**
  - Data-dependent structure work-depth graphs
- **Data movement and I/O complexity**
  - Communication complexity

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- Quicksort?

Work/Depth in Practice – Oblivious Algorithms

"An algorithm is ***execution-oblivious*** *if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- Quicksort?
- Prefix sum on an array?

Work/Depth in Practice – Oblivious Algorithms

*“An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs”*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- Quicksort?
- Prefix sum on an array?
- Simple dense matrix multiplication?

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- Quicksort?
- Prefix sum on an array?
- Simple dense matrix multiplication?
- Dense matrix vector product?

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- Quicksort?
- Prefix sum on an array?
- Simple dense matrix multiplication?
- Dense matrix vector product?
- Sparse matrix vector product?

Work/Depth in Practice – Oblivious Algorithms

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

Execution oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- Quicksort?
- Prefix sum on an array?
- Simple dense matrix multiplication?
- Dense matrix vector product?
- Sparse matrix vector product?
- Queue-based breadth-first search?

Obliviousness as property of an execution

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- **Class question: Can an algorithm decide whether a program is oblivious or not?**

Obliviousness as property of an execution

*"An algorithm is **execution-oblivious** if, for each problem size, the sequence of instructions executed, the set of memory locations read and the set of memory locations written by each executed instruction are determined by the input size and are independent of the values of the other inputs"*

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int findmin(int n, a[n]) {
  for(int i=1; i<n; i++)
    if(a[i]<a[0]) a[0] = a[i];
}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- **Class question: Can an algorithm decide whether a program is oblivious or not?**
  - Answer: no, proof similar to decision problem whether a program always outputs zero or not

Structural obliviousness as stronger property

*“A program is **structurally-oblivious** if any value used in a conditional branch, and any value used to compute indices or pointers is structurally-dependent only in the input variable(s) that contains the problem size but not on any other input”*

Structural obliviousness as stronger property

*“A program is **structurally-oblivious** if any value used in a conditional branch, and any value used to compute indices or pointers is structurally-dependent only in the input variable(s) that contains the problem size but not on any other input”*

Structurally oblivious or not?

Structural obliviousness as stronger property

*"A program is **structurally-oblivious** if any value used in a conditional branch, and any value used to compute indices or pointers is structurally-dependent only in the input variable(s) that contains the problem size but not on any other input"*

Structurally oblivious or not?

```
int reduce(int n, arr[n]) {
 for(int i=0; i<n; ++i)
  sum += arr[i];
}
```

Structural obliviousness as stronger property

*“A program is **structurally-oblivious** if any value used in a conditional branch, and any value used to compute indices or pointers is structurally-dependent only in the input variable(s) that contains the problem size but not on any other input”*

Structurally oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int oblivious(int n, a[n], b[n]) {
  for(int i=0; i<n; ++i) {
    x = a[i] + 1;
    if (x > a[i]) b[i] = 1;
    else b[i] = 2;
} }
```

Structural obliviousness as stronger property

*"A program is **structurally-oblivious** if any value used in a conditional branch, and any value used to compute indices or pointers is structurally-dependent only in the input variable(s) that contains the problem size but not on any other input"*

Structurally oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int oblivious(int n, a[n], b[n]) {
  for(int i=0; i<n; ++i) {
    x = a[i] + 1;
    if (x > a[i]) b[i] = 1;
    else b[i] = 2;
} }
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

Structural obliviousness as stronger property

*"A program is **structurally-oblivious** if any value used in a conditional branch, and any value used to compute indices or pointers is structurally-dependent only in the input variable(s) that contains the problem size but not on any other input"*

Structurally oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int oblivious(int n, a[n], b[n]) {
  for(int i=0; i<n; ++i) {
    x = a[i] + 1;
    if (x > a[i]) b[i] = 1;
    else b[i] = 2;
}}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- **Clear that structurally oblivious programs are also execution oblivious**
  - Can be programmatically (statically decided)
  - Sufficient for practical use

Structural obliviousness as stronger property

*"A program is **structurally-oblivious** if any value used in a conditional branch, and any value used to compute indices or pointers is structurally-dependent only in the input variable(s) that contains the problem size but not on any other input"*

Structurally oblivious or not?

```
int reduce(int n, arr[n]) {
  for(int i=0; i<n; ++i)
    sum += arr[i];
}
```

```
int oblivious(int n, a[n], b[n]) {
  for(int i=0; i<n; ++i) {
    x = a[i] + 1;
    if (x > a[i]) b[i] = 1;
    else b[i] = 2;
}}
```

```
int finditem(list_t list)
  item = list.head;
  while(item.value!=0 && item.next!=NULL)
    item=item.next;
}
```

- **Clear that structurally oblivious programs are also execution oblivious**
  - Can be programmatically (statically decided)
  - Sufficient for practical use
- **The middle example is not structurally oblivious but execution oblivious**
  - First branch is always taken (assuming no overflow) but static dependency analysis is conservative

Why obliviousness?

We can easily reason about oblivious algorithms

- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

$\log_2 n]$

Are both W and D optimal?

Yes!

Why obliviousness?

We can easily reason about oblivious algorithms

- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**

$\log_2 n]$

Are both W and D optimal?

Yes!

Why obliviousness?

We can easily reason about oblivious algorithms

- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**
  - Question: what is W(n) and D(n) of sequential summation?

$\log_2 n \rceil$

Are both W and D optimal?

Yes!

Why obliviousness?

We can easily reason about oblivious algorithms

- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**
  - Question: what is W(n) and D(n) of sequential summation?

    W(n)=D(n)=n-1

$\log_2 n]$

Are both W and D optimal?

Yes!

Why obliviousness?

We can easily reason about oblivious algorithms

- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**
  - Question: what is W(n) and D(n) of sequential summation?

    W(n)=D(n)=n-1

  - Question: is this optimal? How would you define optimality?

$\log_2 n]$

Are both W and D optimal?

Yes!

Why obliviousness?

We can easily reason about oblivious algorithms

- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**
  - Question: what is W(n) and D(n) of sequential summation?

    W(n)=D(n)=n-1

  - Question: is this optimal? How would you define optimality?

    Separate for W and D! Typically try to achieve both!

$\log_2 n]$

Are both W and D optimal?

Yes!

Why obliviousness?

We can easily reason about oblivious algorithms
- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**
  - Question: what is W(n) and D(n) of sequential summation?
    W(n)=D(n)=n-1
  - Question: is this optimal? How would you define optimality?
    Separate for W and D! Typically try to achieve both!
  - Question: what is W(n) and D(n) of the optimal parallel summation?
    $\log_2 n]$
    Are both W and D optimal?
    Yes!

Why obliviousness?

log 2 n] **log 2** log log 2 *2 log 2 log 2 n] nn] log 2 n]*

We **can easily reason about oblivious algorithms**

- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**
  - Question: what is W(n) and D(n) of sequential summation?

    W(n)=D(n)=n-1

  - Question: is this optimal? How would you define optimality?

    Separate for W and D! Typically try to achieve both!

  - Question: what is W(n) and D(n) of the optimal parallel summation?

    W(n)=n-1 D(n)=$\lceil \log_2 n \rceil$

    Are both W and D optimal?

    Yes!

Why obliviousness?

log 2 n**]** **log 2** log log 2 *2 log 2 log 2 n] nn] log 2 n]*

We **can easily reason about oblivious algorithms**
- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**
  - Question: what is W(n) and D(n) of sequential summation?

    W(n)=D(n)=n-1
  - Question: is this optimal? How would you define optimality?

    Separate for W and D! Typically try to achieve both!
  - Question: what is W(n) and D(n) of the optimal parallel summation?

    Are both W and D optimal?

    Are both W and D optimal?

    Yes!

Why obliviousness?

log 2 n] log 2 log log 2 2 log 2 log 2 n] nn] log 2 n]

We **can easily reason about oblivious algorithms**

- Execution DAG can be constructed "statically"
- We have done this in the last weeks intuitively but you never asked how to do it for BFS for example ☺

- **Simple example (that you know): parallel summation**
  - Question: what is W(n) and D(n) of sequential summation?

    W(n)=D(n)=n-1

  - Question: is this optimal? How would you define optimality?

    Separate for W and D! Typically try to achieve both!

  - Question: what is W(n) and D(n) of the optimal parallel summation?

    Are both W and D optimal?

    Yes!

    Yes!

Starting simple: optimality?

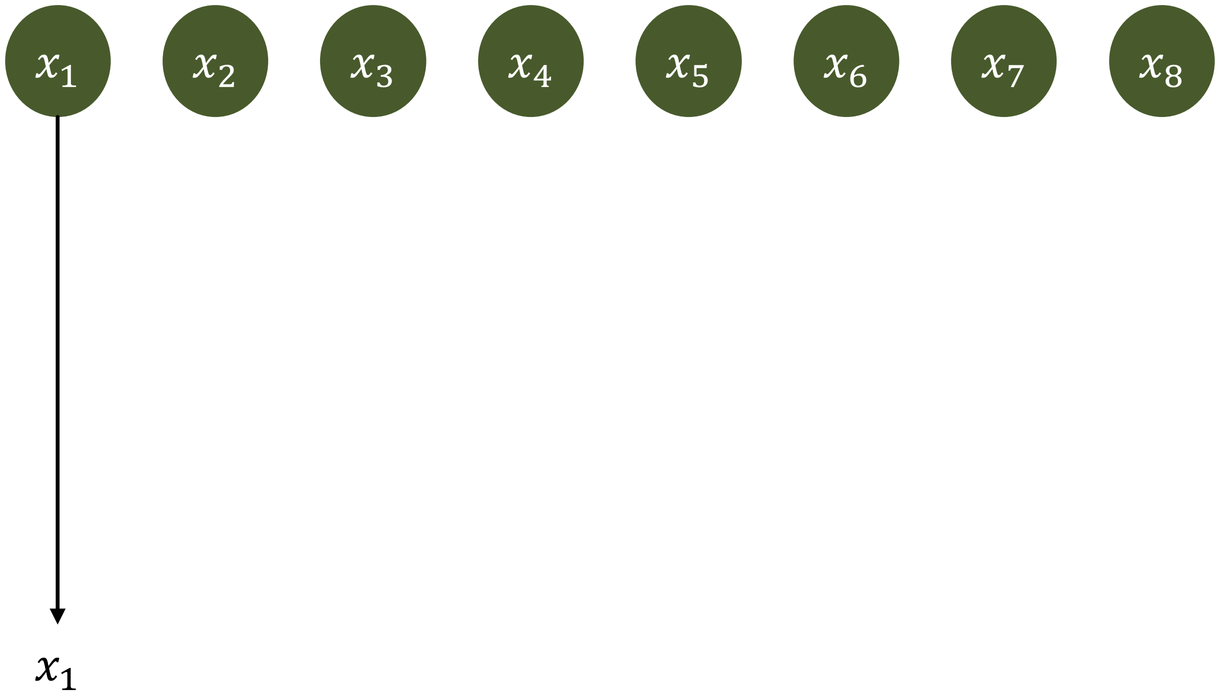
- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \ldots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \ldots; x_1 + x_2 + x_i \ldots + x_{n-1} + x_n]$
  - Simple serial schedule

Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \ldots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \ldots; x_1 + x_2 + x_i \ldots + x_{n-1} + x_n]$
  - Simple serial schedule

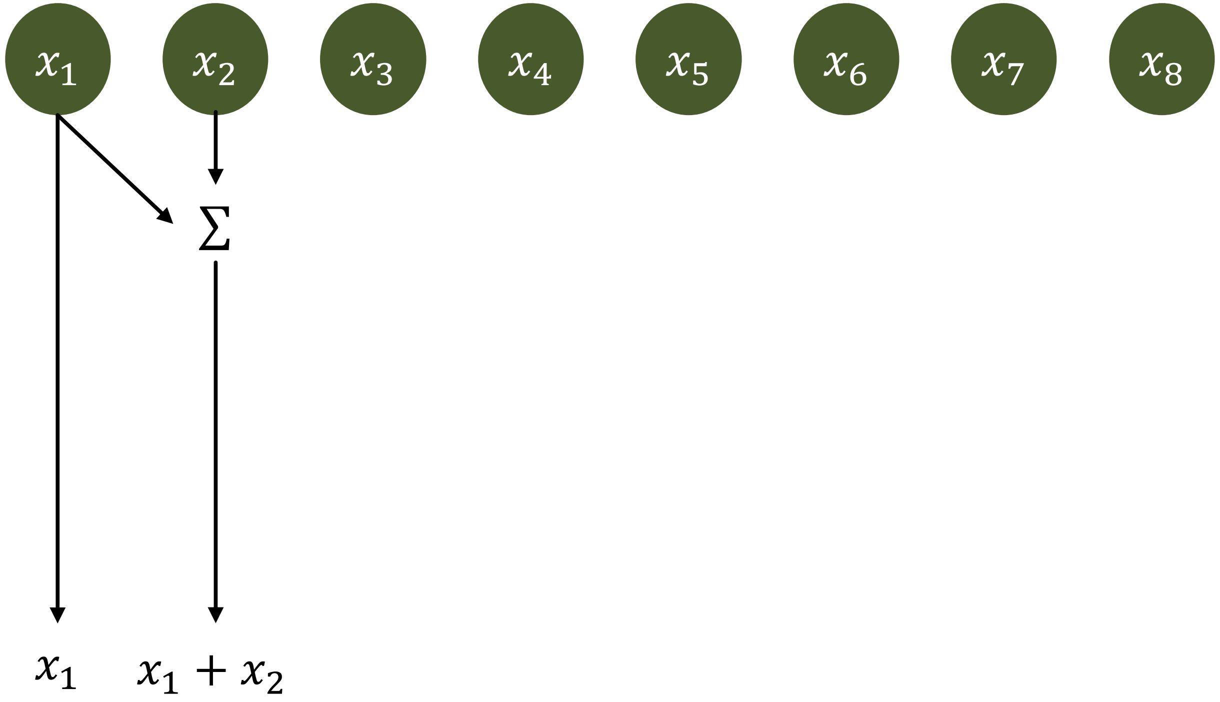
Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \dots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots; x_1 + x_2 + x_i \dots + x_{n-1} + x_n]$
  - Simple serial schedule



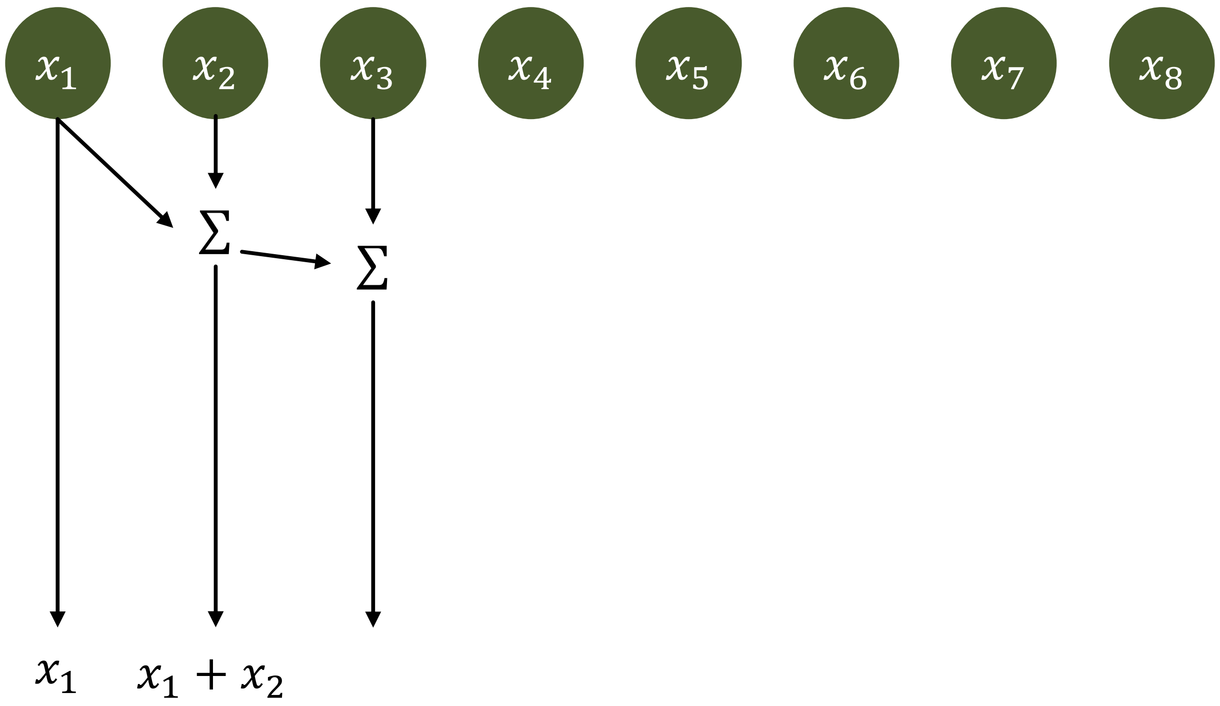
Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \ldots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \ldots; x_1 + x_2 + x_i \ldots + x_{n-1} + x_n]$
  - Simple serial schedule



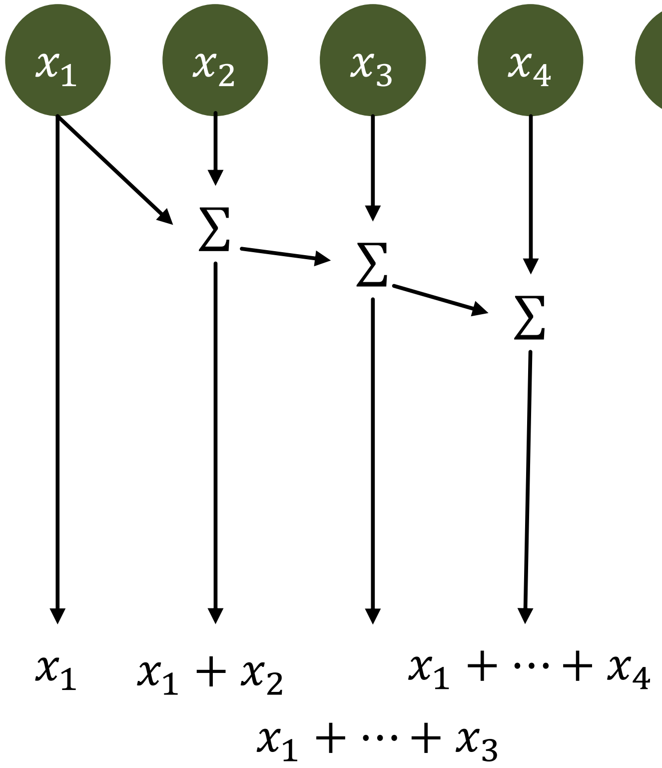
Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \dots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots; x_1 + x_2 + x_i \dots + x_{n-1} + x_n]$
  - Simple serial schedule



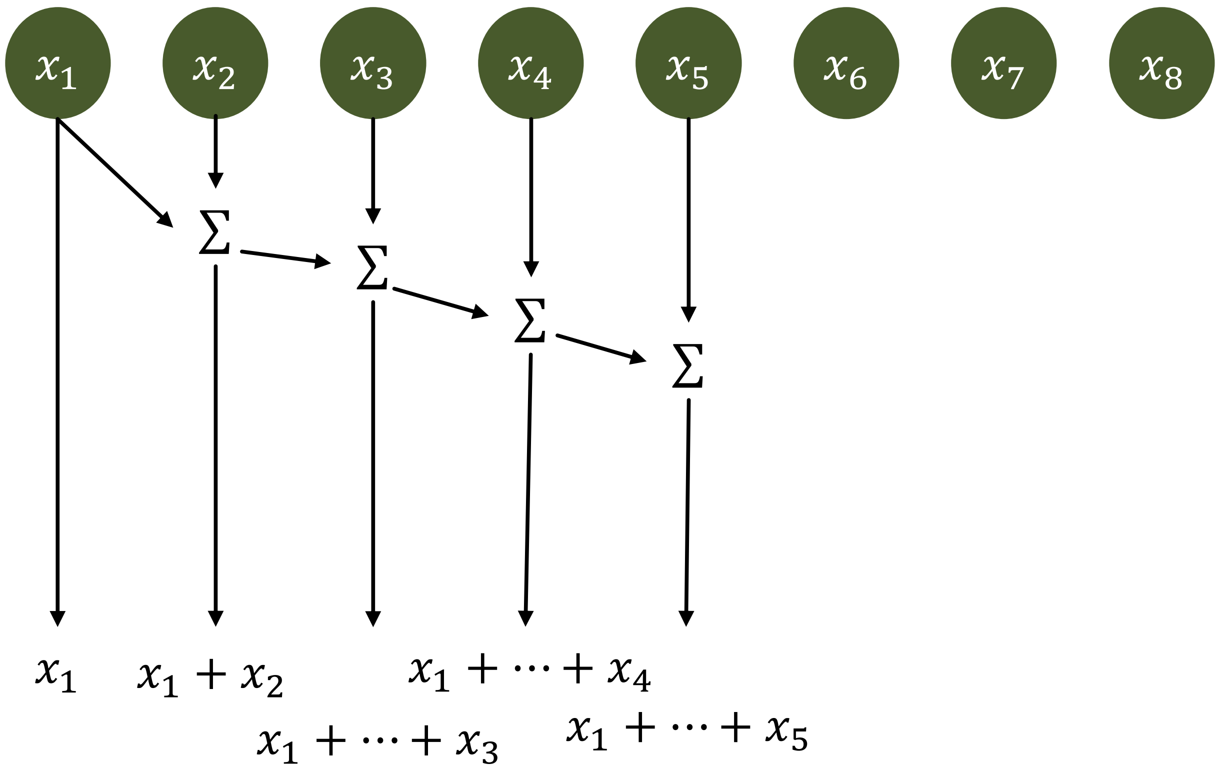
Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \dots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots; x_1 + x_2 + x_i \dots + x_{n-1} + x_n]$
  - Simple serial schedule



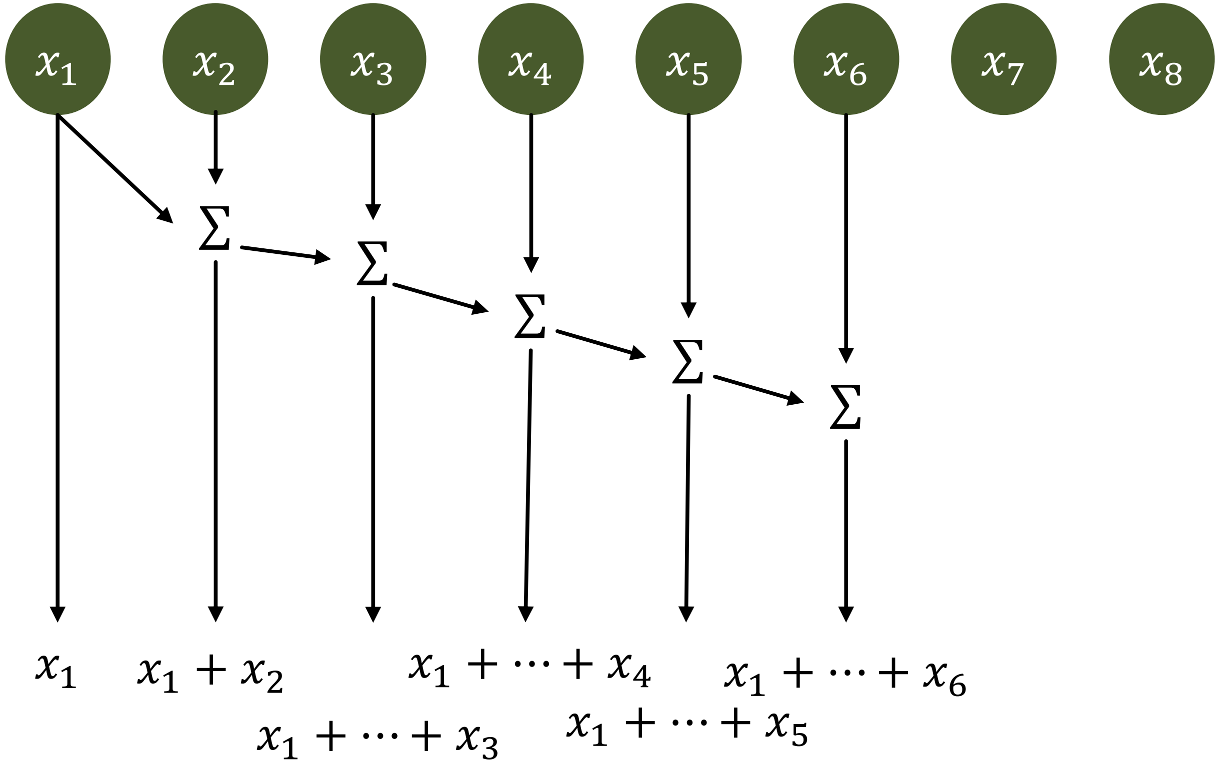
Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \ldots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \ldots; x_1 + x_2 + x_i \ldots + x_{n-1} + x_n]$
  - Simple serial schedule



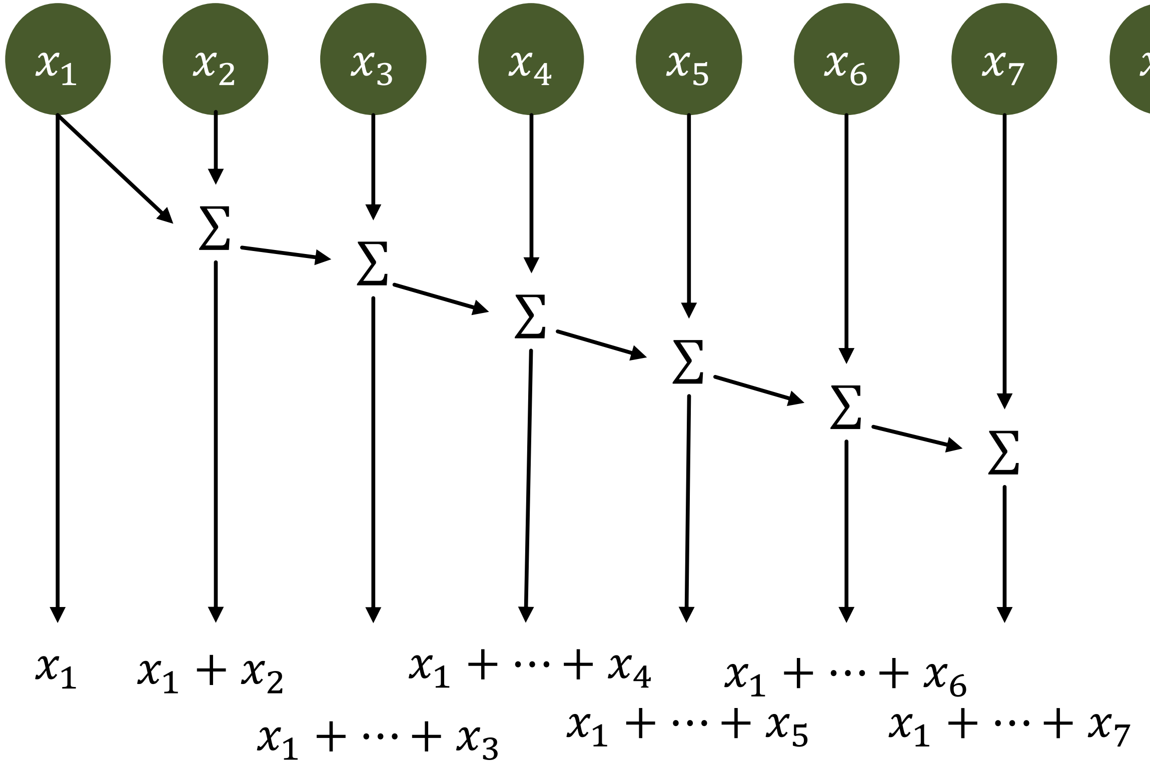
Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \dots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots; x_1 + x_2 + x_i \dots + x_{n-1} + x_n]$
  - Simple serial schedule



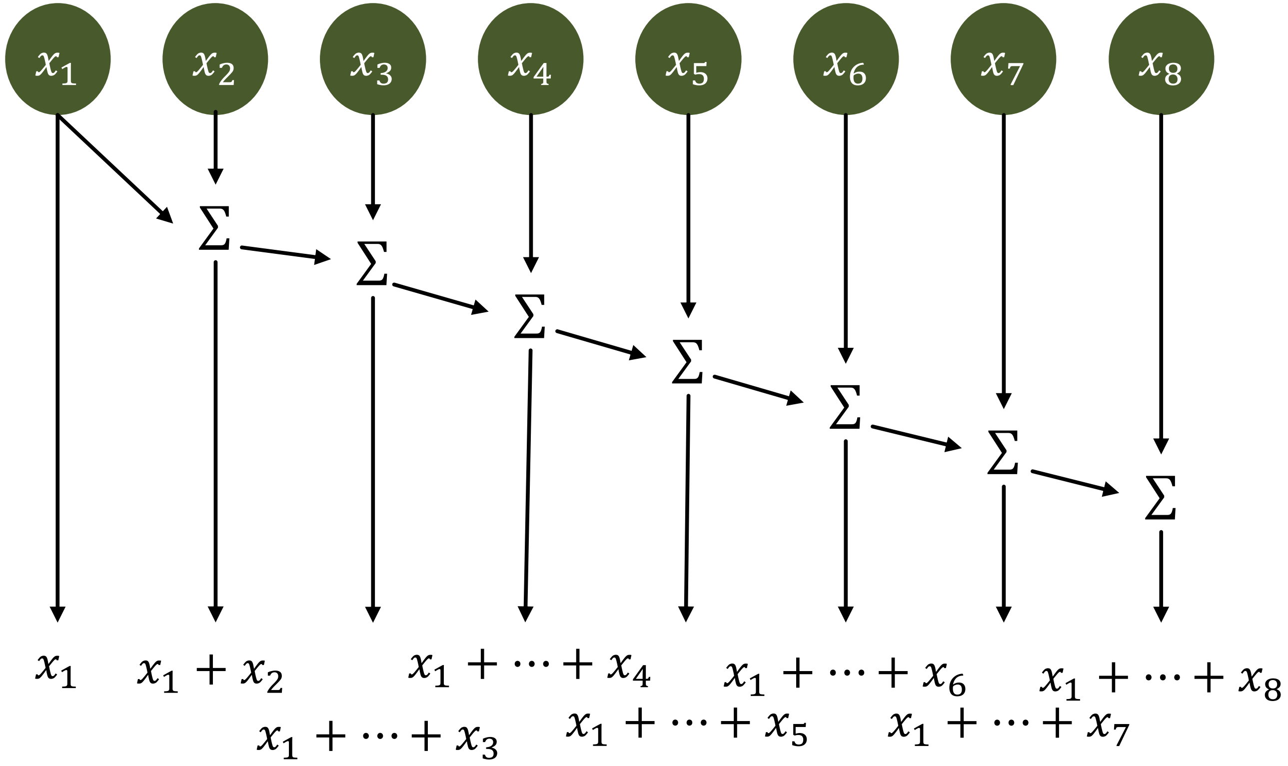
Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \dots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots; x_1 + x_2 + x_i \dots + x_{n-1} + x_n]$
  - Simple serial schedule



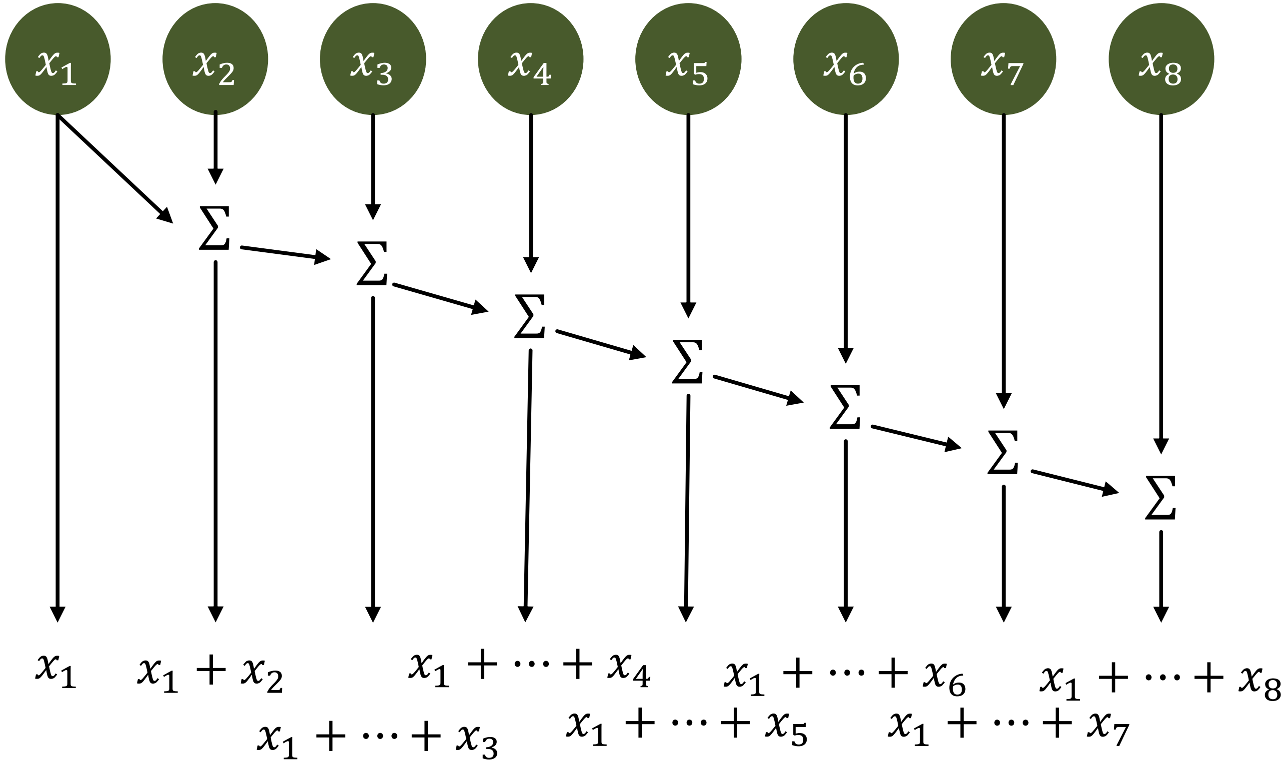
Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \dots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots; x_1 + x_2 + x_i \dots + x_{n-1} + x_n]$
  - Simple serial schedule



Starting simple: optimality?

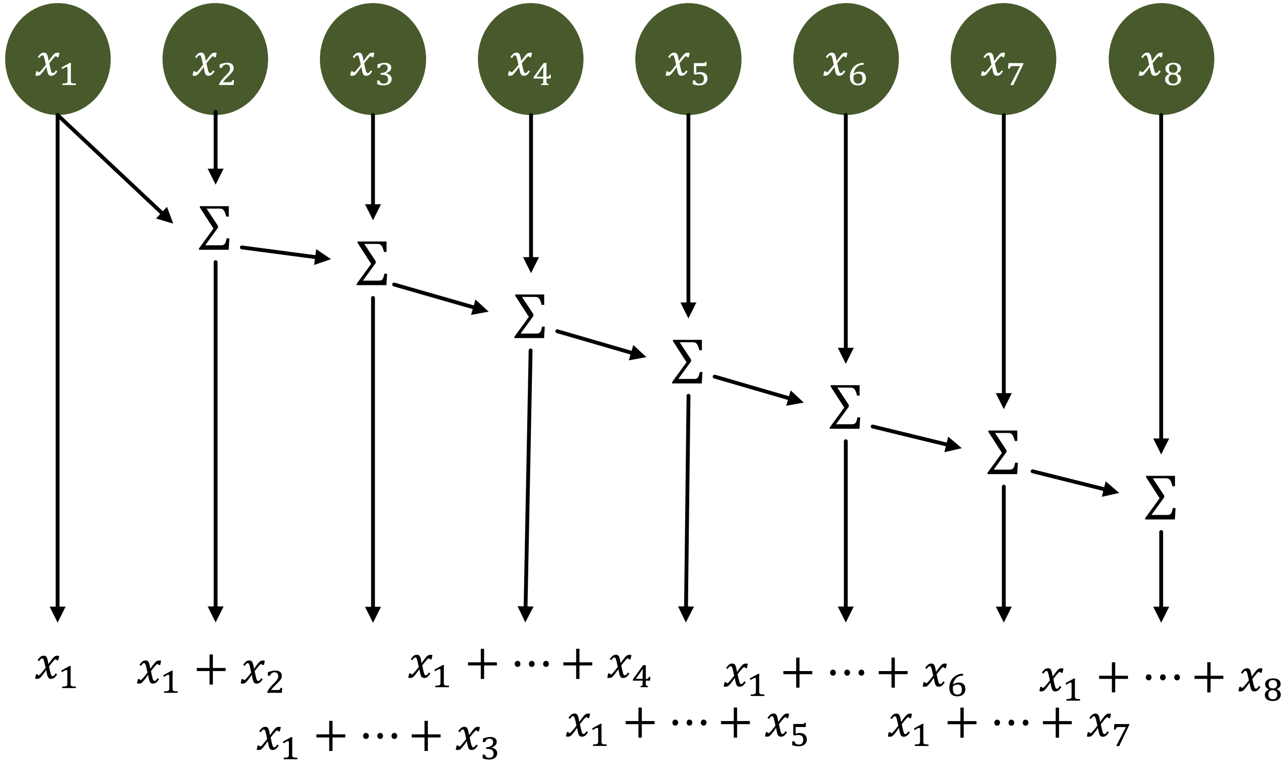
- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \ldots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \ldots; x_1 + x_2 + x_i \ldots + x_{n-1} + x_n]$
  - Simple serial schedule



Class question: work and depth?

Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \dots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots; x_1 + x_2 + x_i \dots + x_{n-1} + x_n]$
  - Simple serial schedule

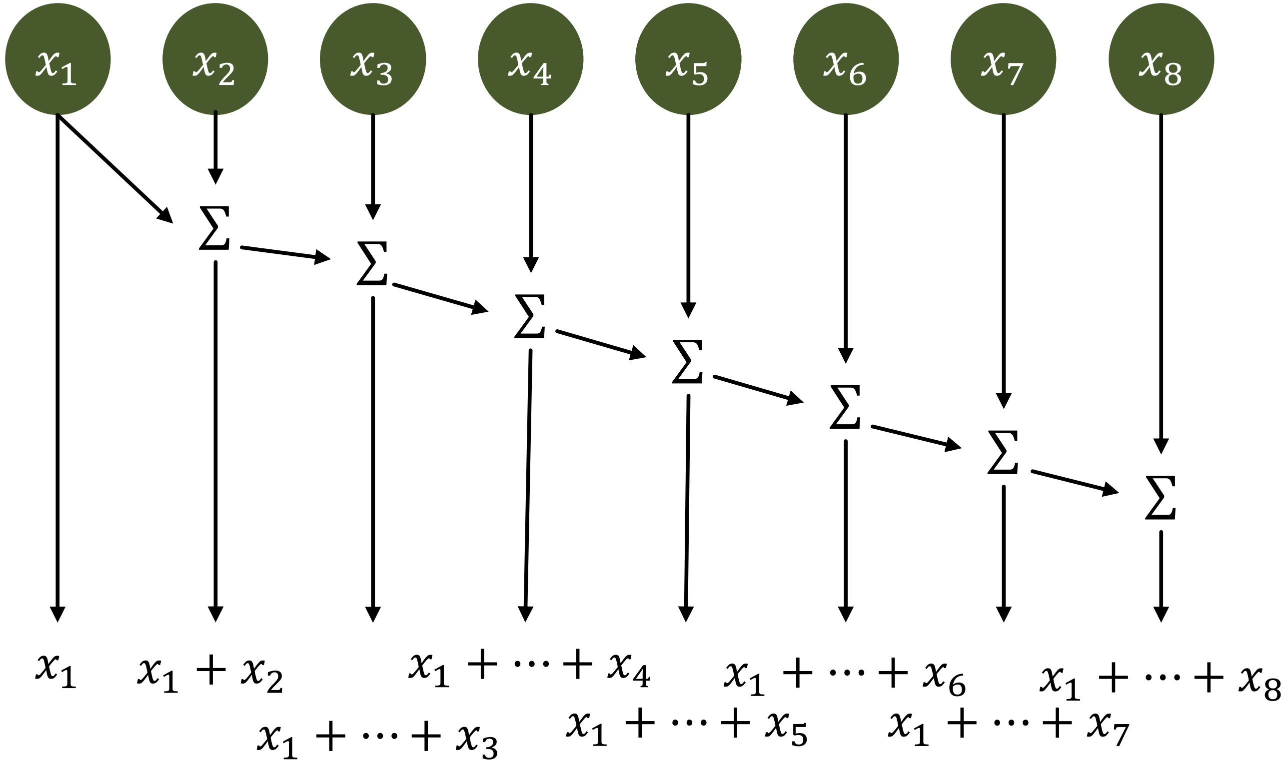


Class question: work and depth?

W(n) = n-1, D(n) = n-1

Starting simple: optimality?

- **Next example you know: scan!**
  - For a vector $[x_1, x_2, \ldots, x_n]$ compute vector of n results: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \ldots; x_1 + x_2 + x_i \ldots + x_{n-1} + x_n]$
  - Simple serial schedule



Class question: work and depth?

W(n) = n-1, D(n) = n-1

Class question: is this optimal?

What did we learn earlier?

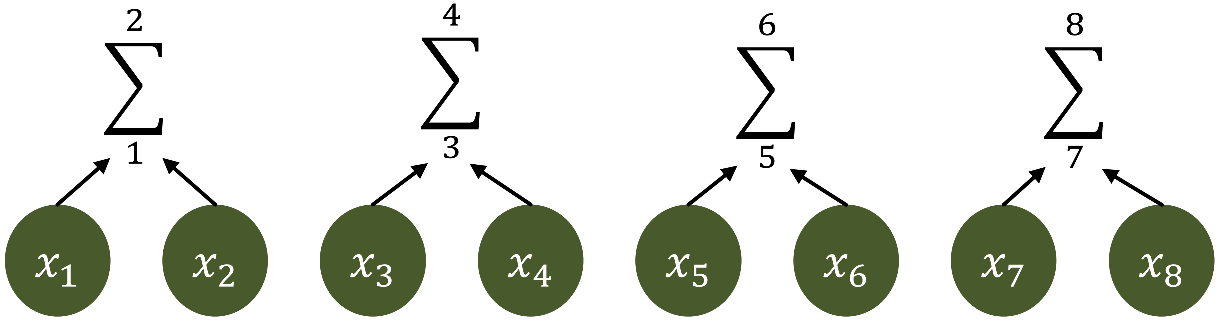
- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**

What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**

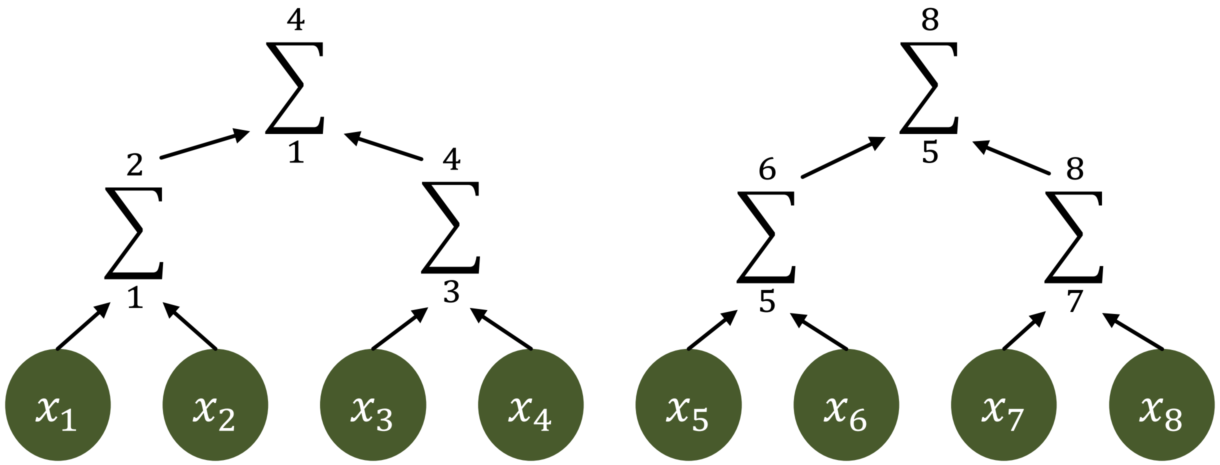
What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



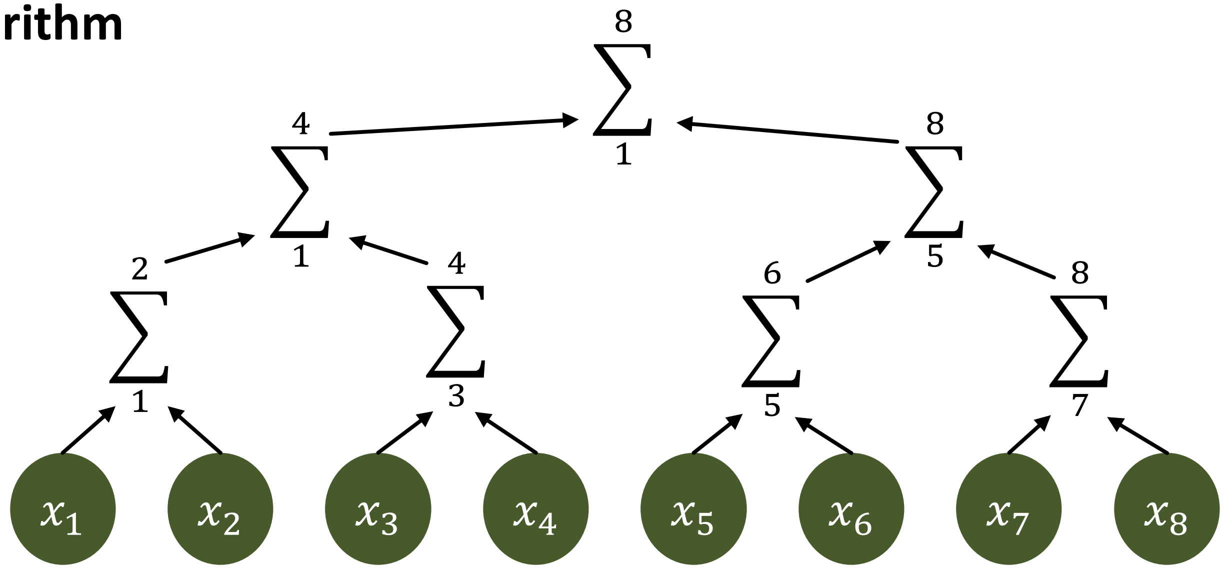
What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



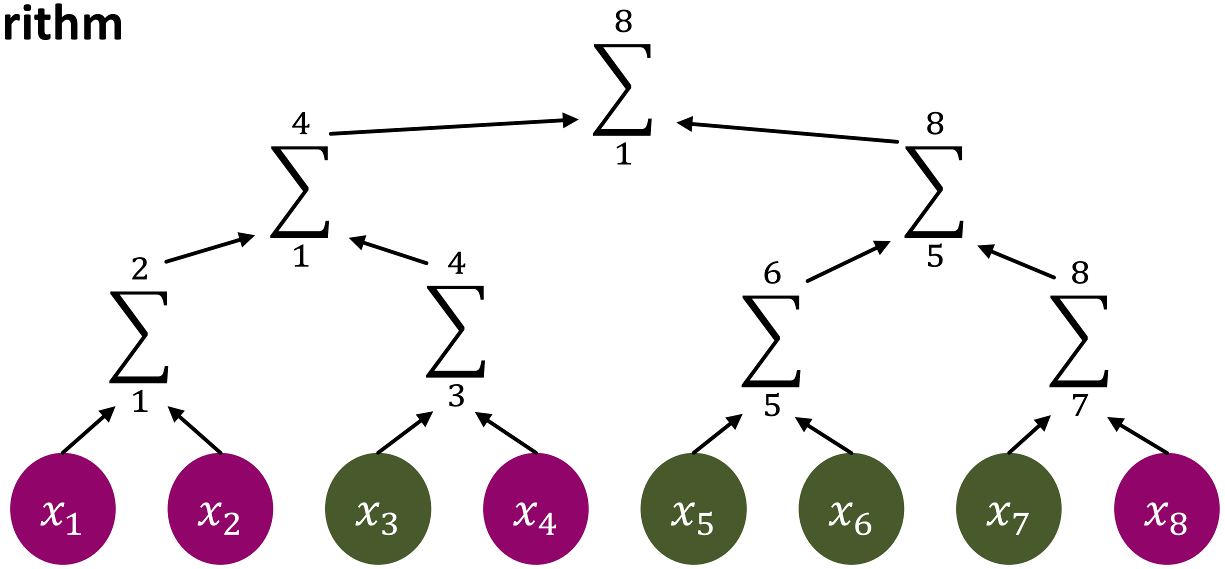
What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds “optimal”, doesn’t it? Well, let’s look at the constants!
- **Algorithm**



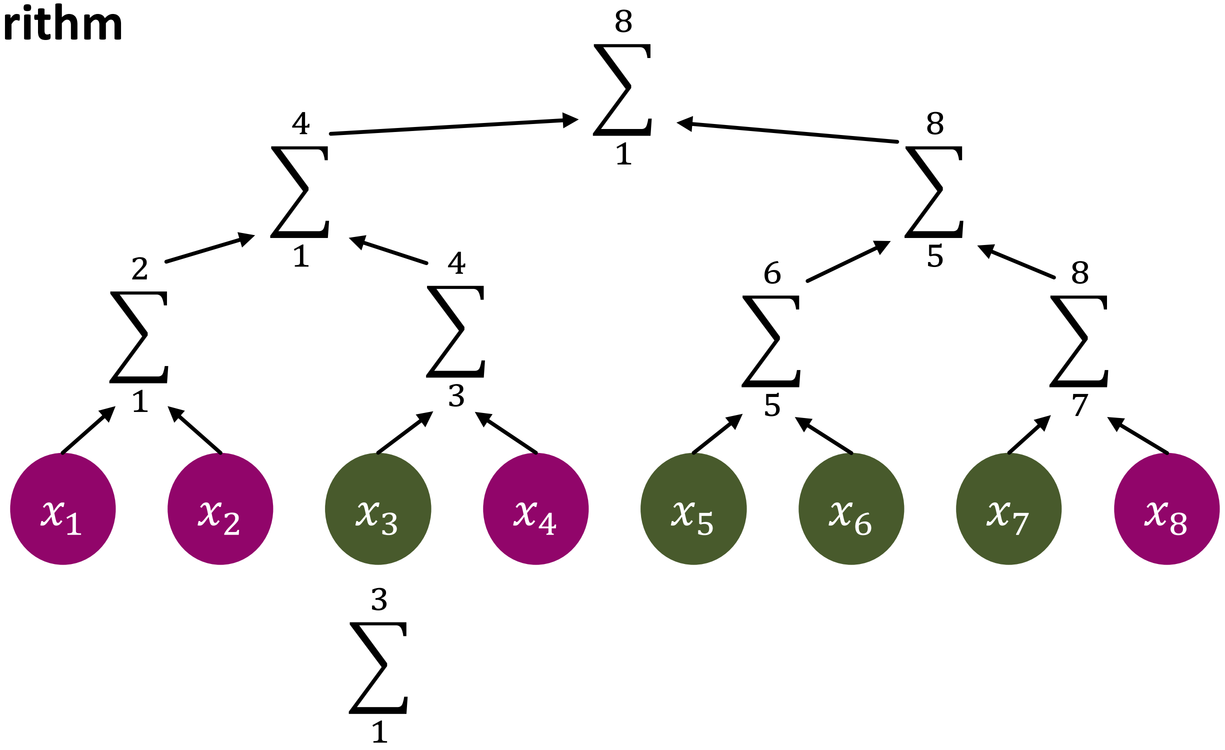
What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



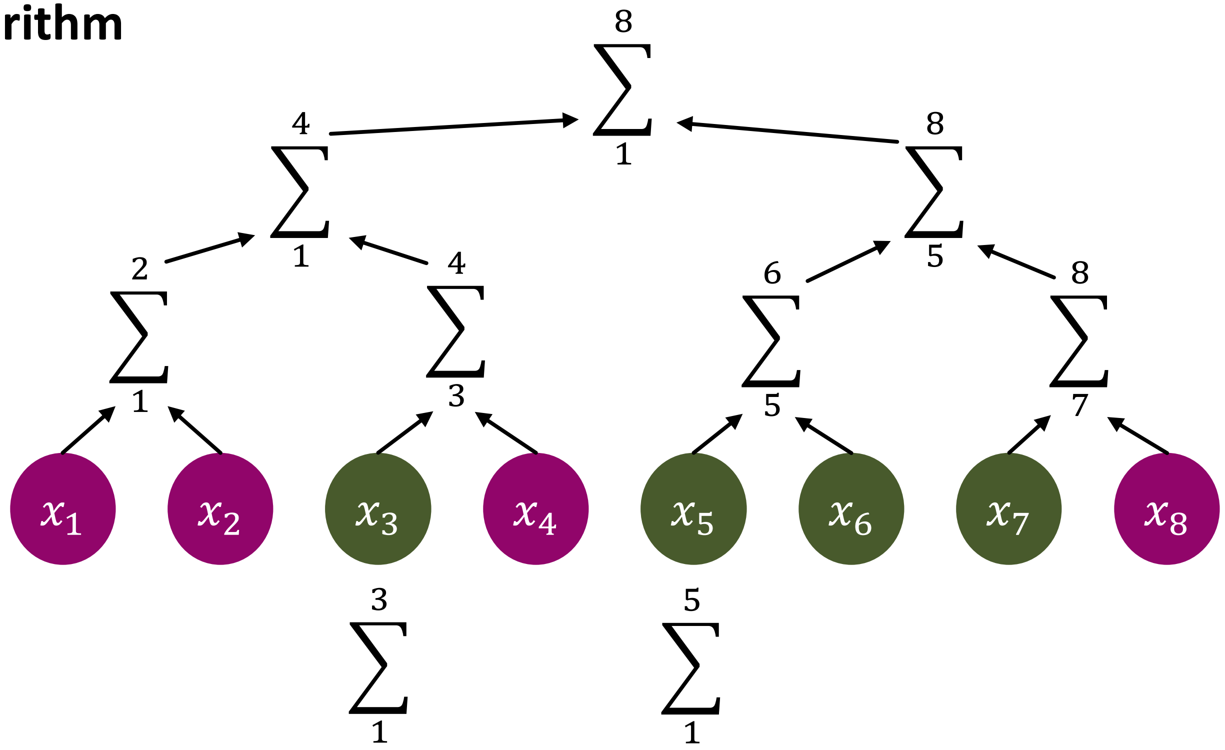
What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



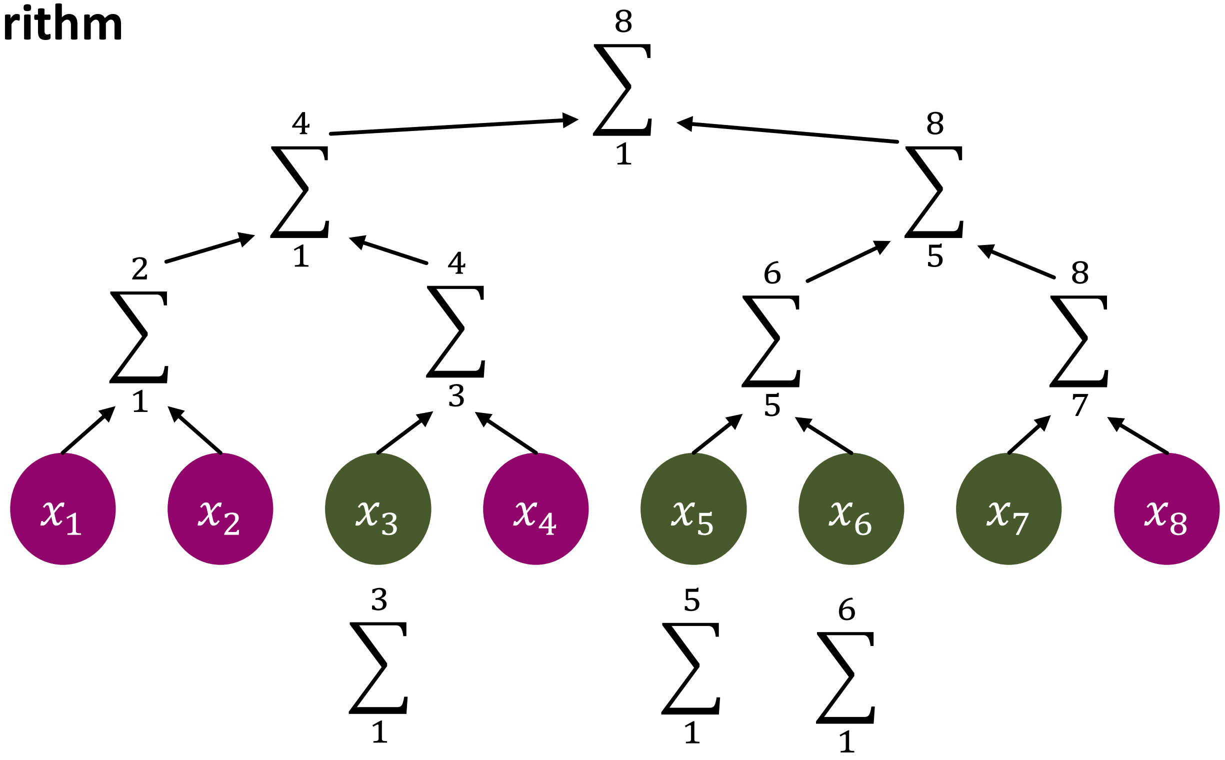
What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



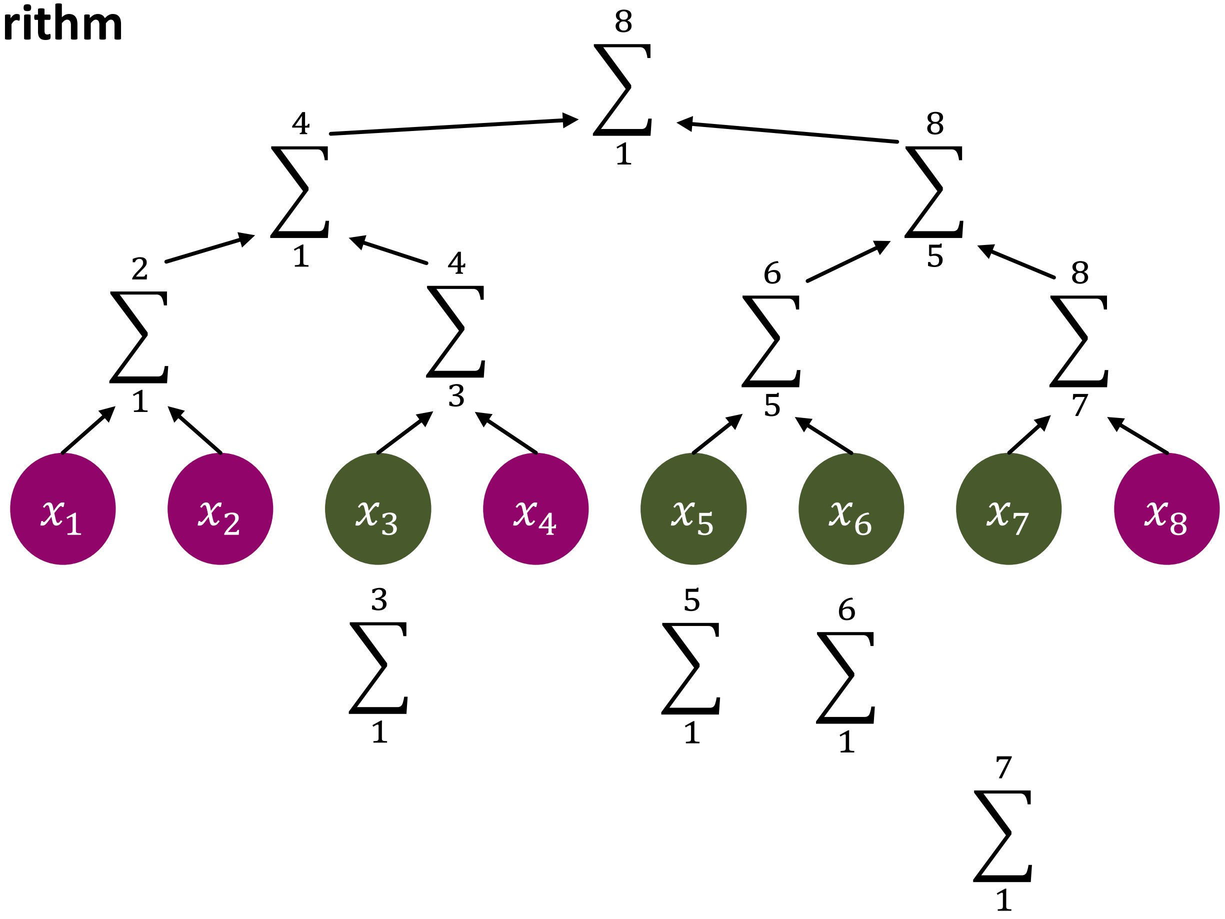
What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds “optimal”, doesn’t it? Well, let’s look at the constants!
- **Algorithm**



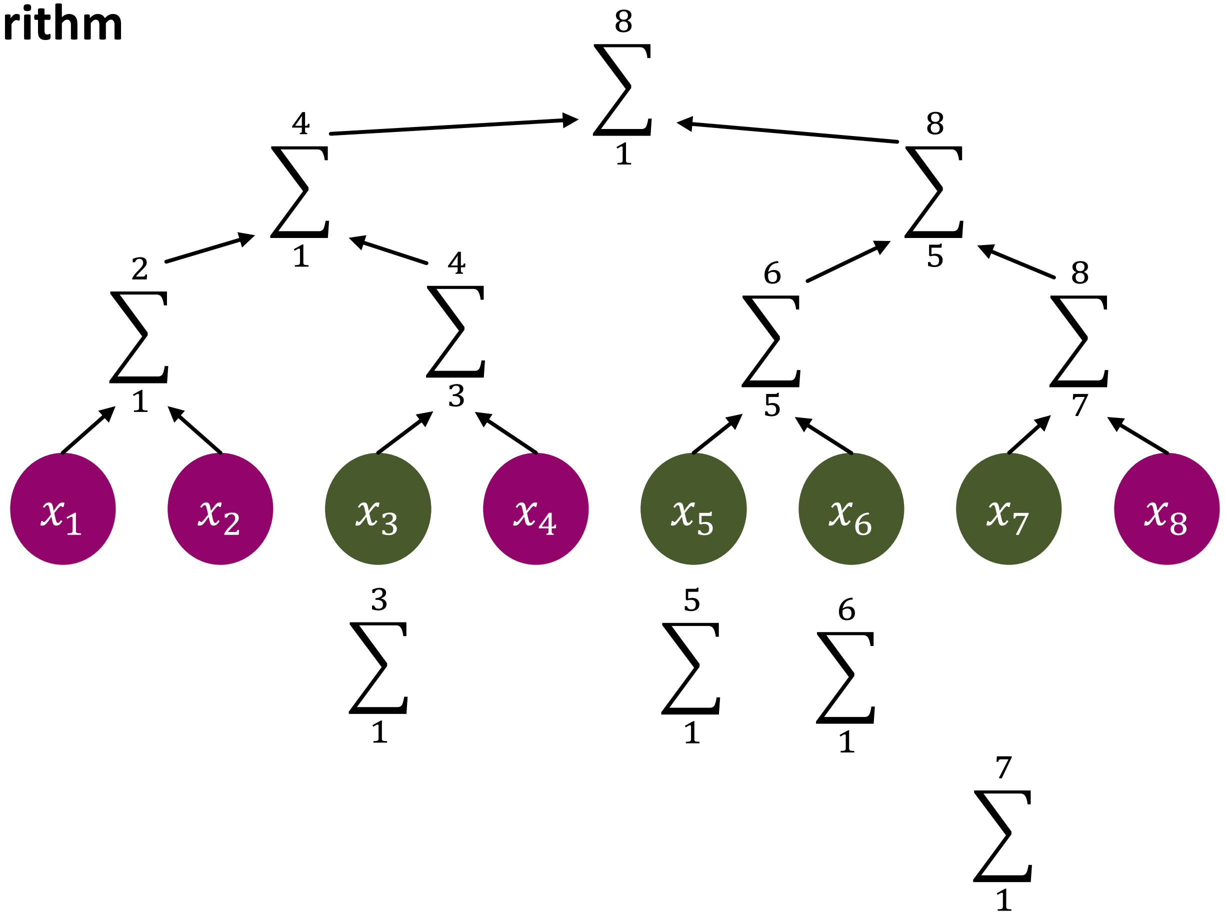
What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(\log n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**

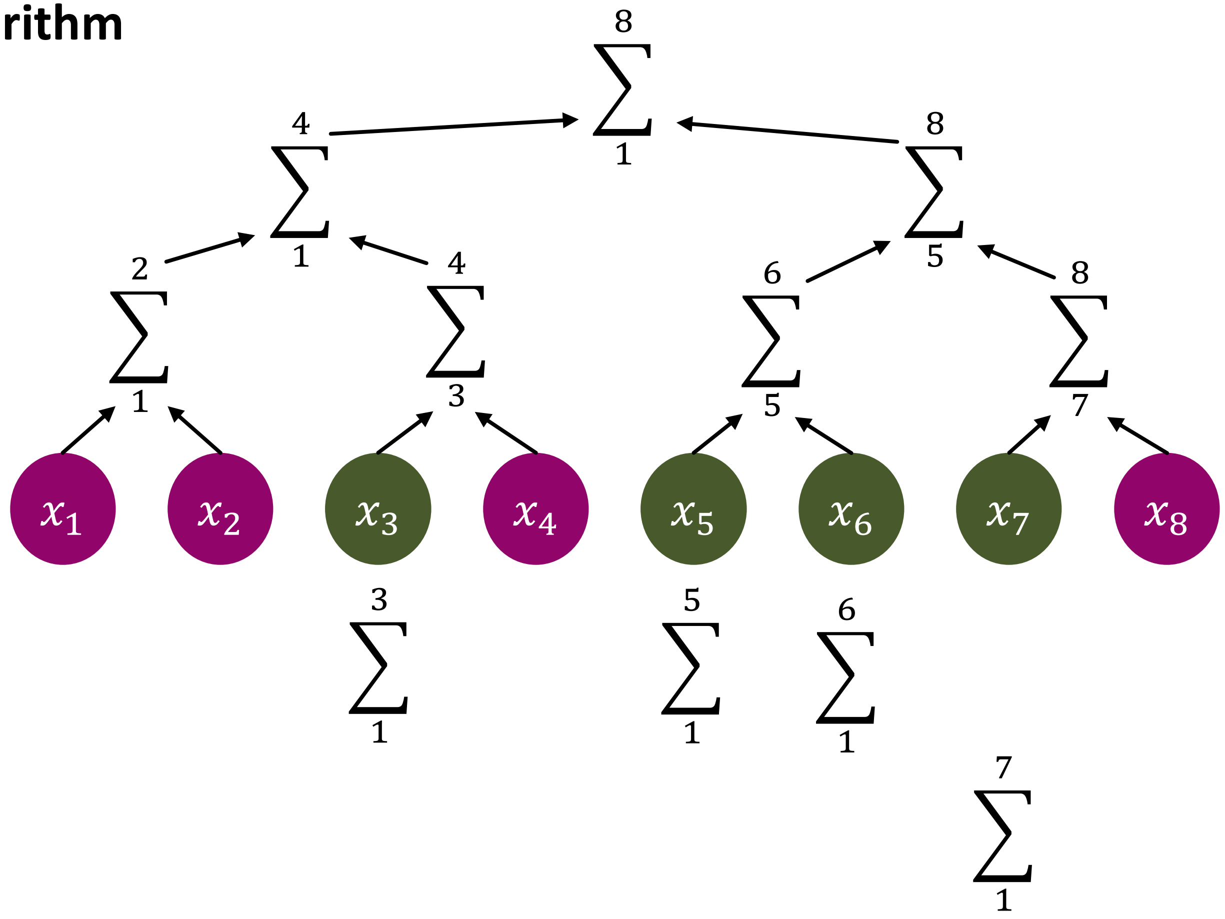


Class question: work?

(hint: after the way up, all powers of two are done, all others require another operation each)

What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(\log n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



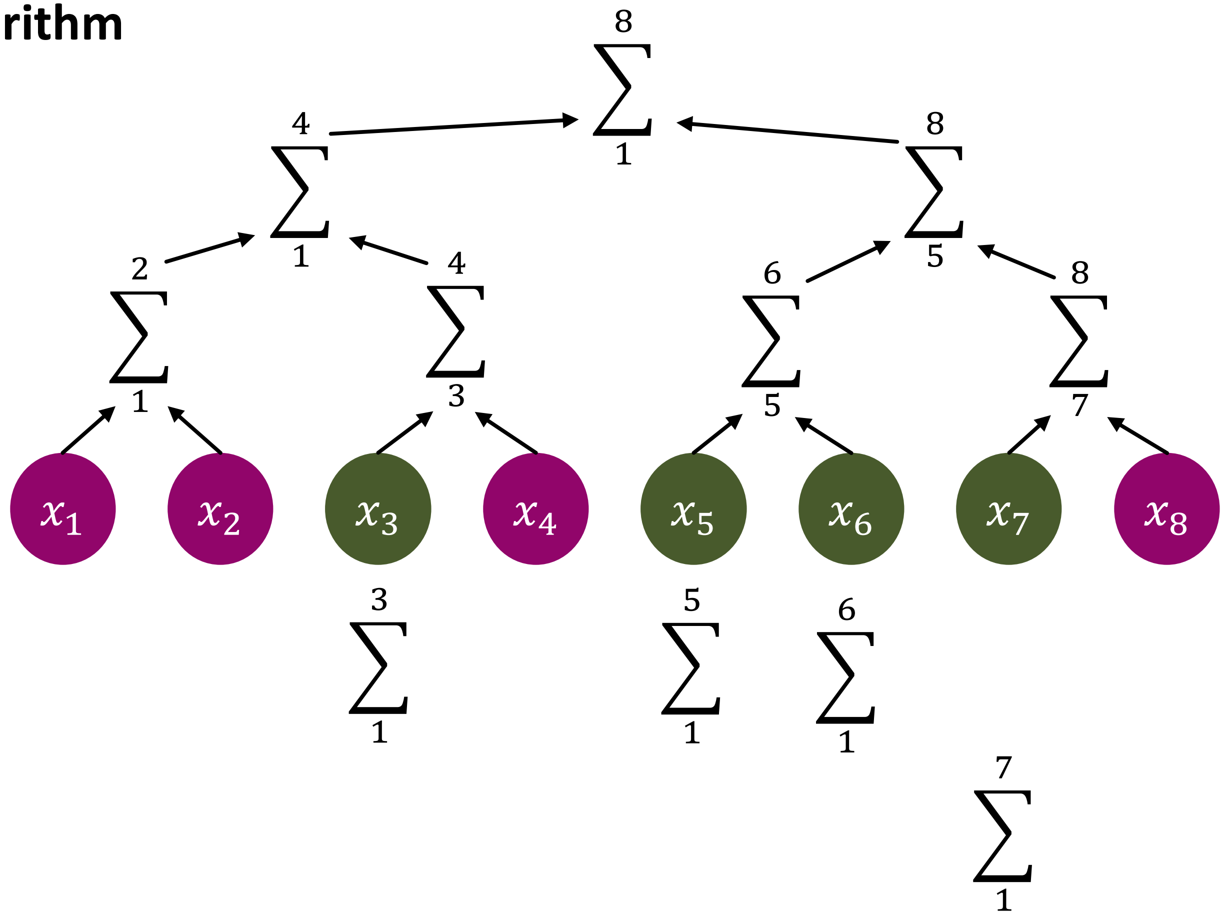
Class question: work?

(hint: after the way up, all powers of two are done, all others require another operation each)

$$W(n) = 2n - \log_2 n - 1$$

What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(log\ n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds “optimal”, doesn’t it? Well, let’s look at the constants!
- **Algorithm**



Class question: work?

(hint: after the way up, all powers of two are done, all others require another operation each)

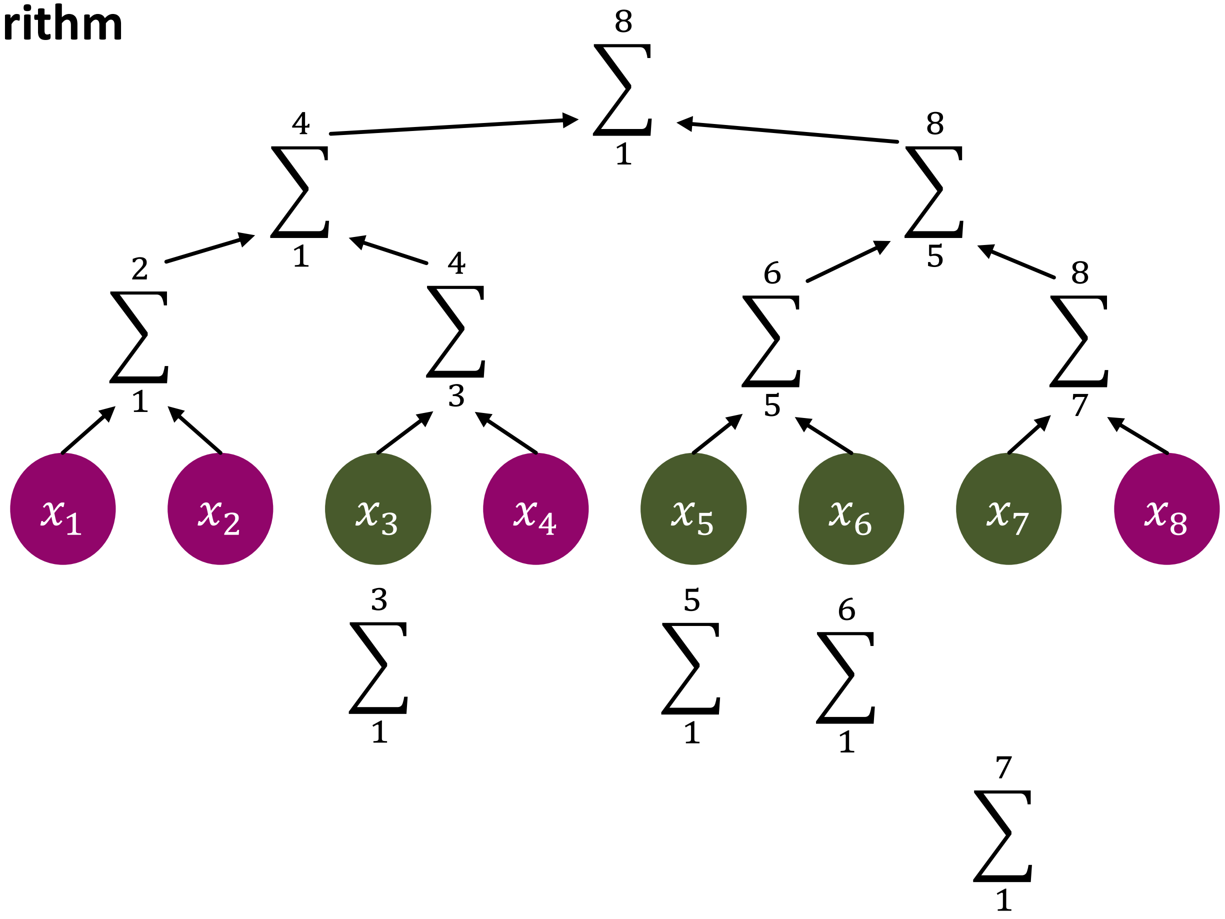
$$W(n) = 2n - \log_2 n - 1$$

Class question: depth?

(needs to go up and down the tree)

What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(\log n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



Class question: work?

(hint: after the way up, all powers of two are done, all others require another operation each)

$$W(n) = 2n - \log_2 n - 1$$

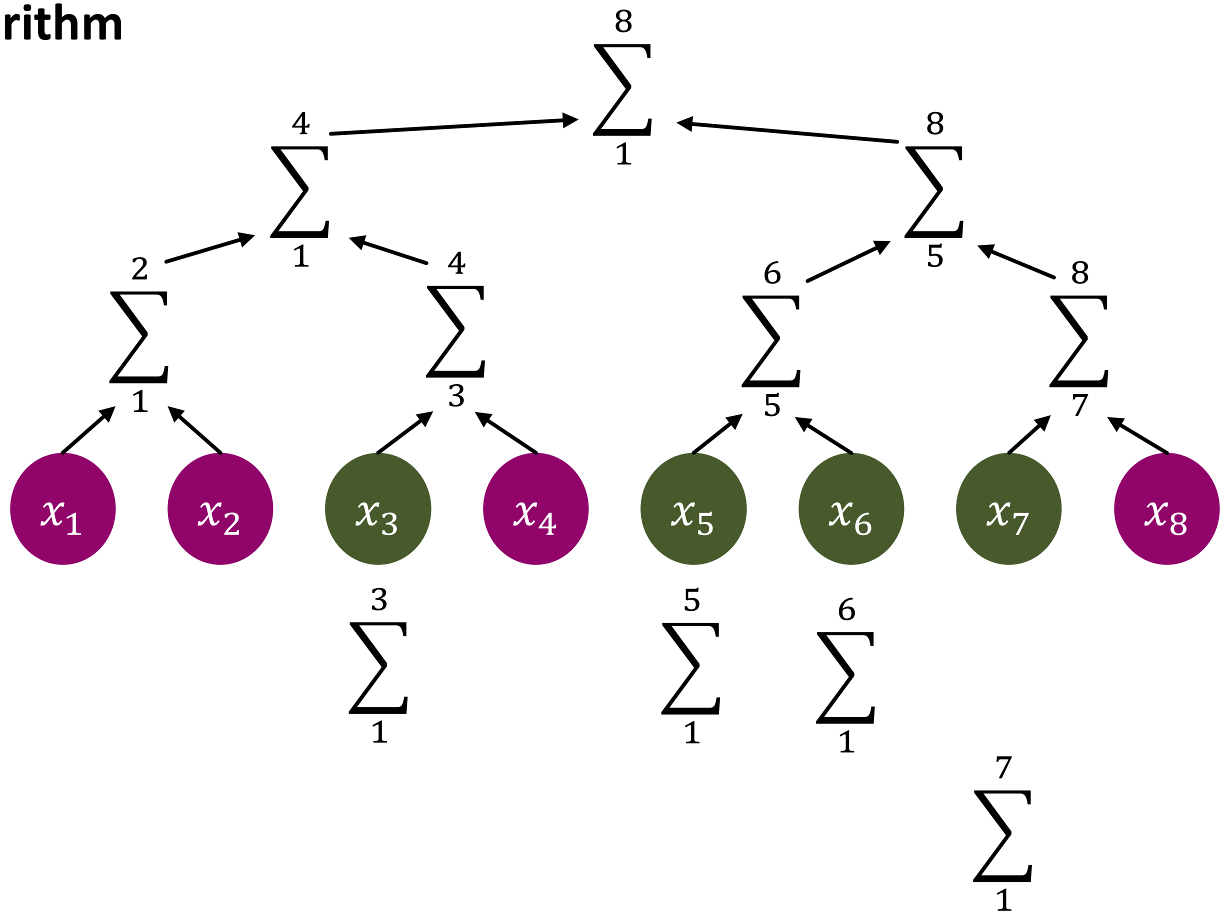
Class question: depth?

(needs to go up and down the tree)

$$D(n) = 2\log_2 n - 1$$

What did we learn earlier?

- **Recursive to get to $W = O(n)$ and $D = O(\log n)$! Assume $n = 2^k$ for simplicity!**
  - Sounds "optimal", doesn't it? Well, let's look at the constants!
- **Algorithm**



Class question: work?

(hint: after the way up, all powers of two are done, all others require another operation each)

$$W(n) = 2n - \log_2 n - 1$$

Class question: depth?

(needs to go up and down the tree)

$$D(n) = 2\log_2 n - 1$$

Class question: what happened to optimality?

Oh no, not good, another algorithm to the rescue!

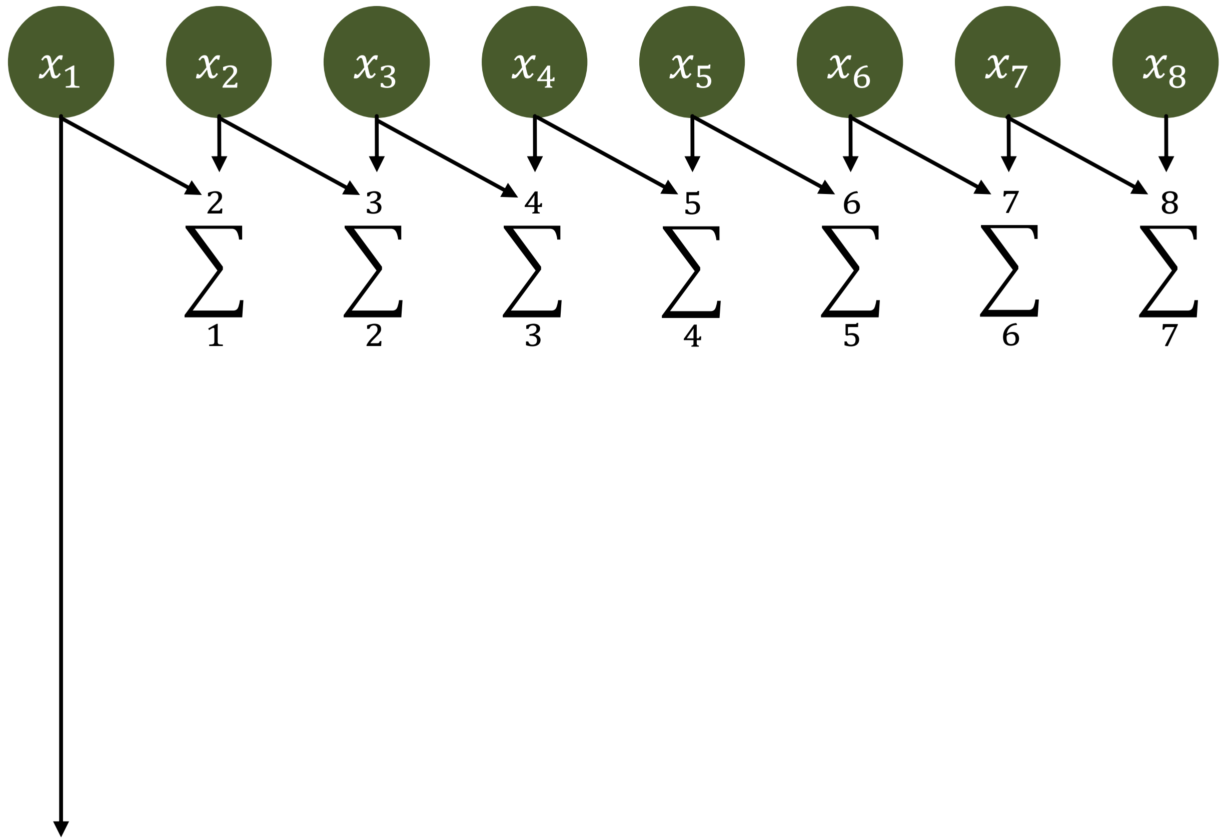
- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**

Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**

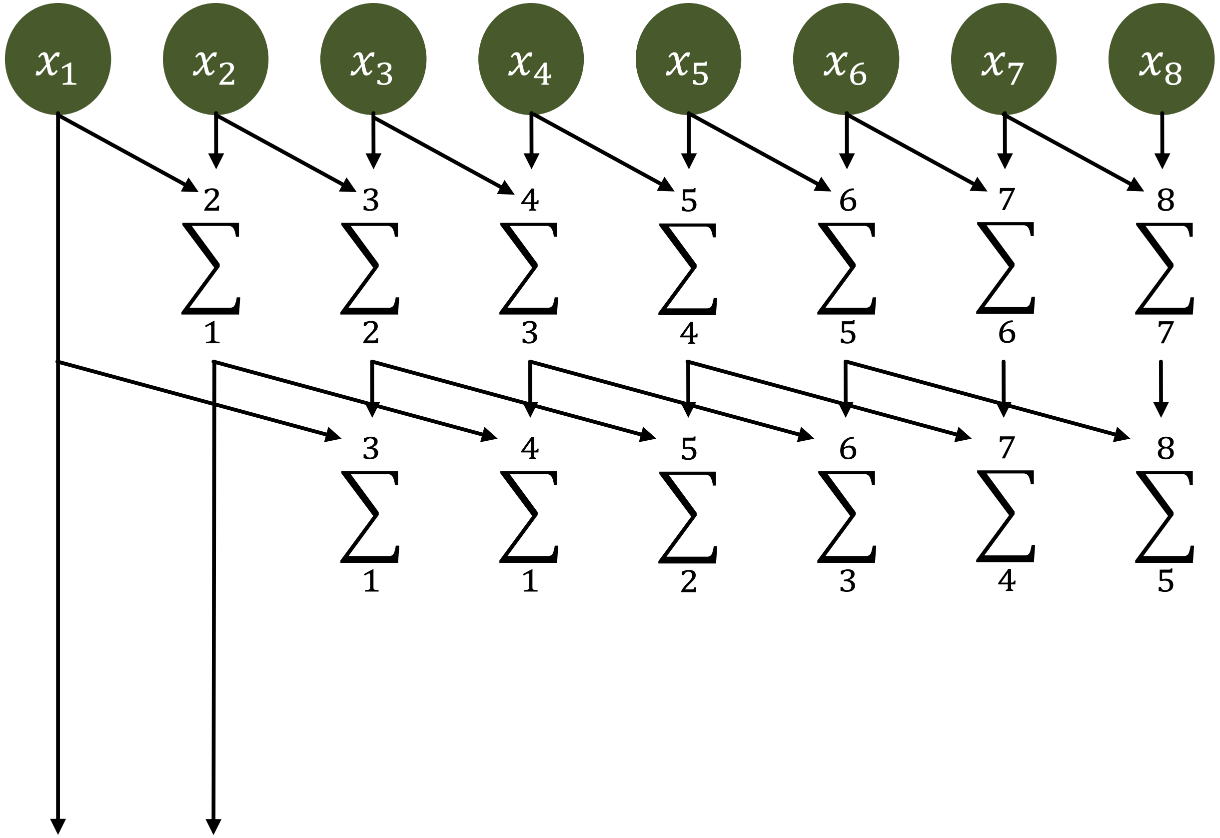
Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**



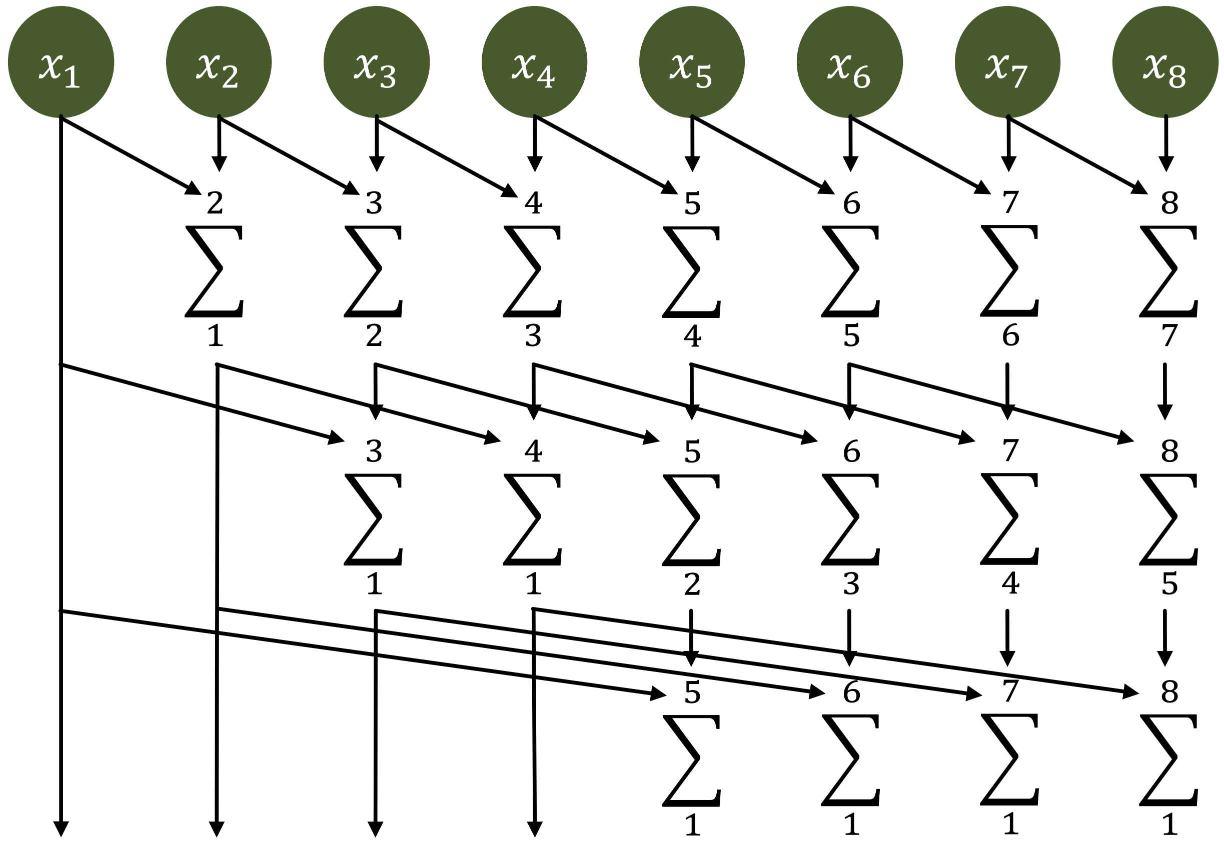
Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**



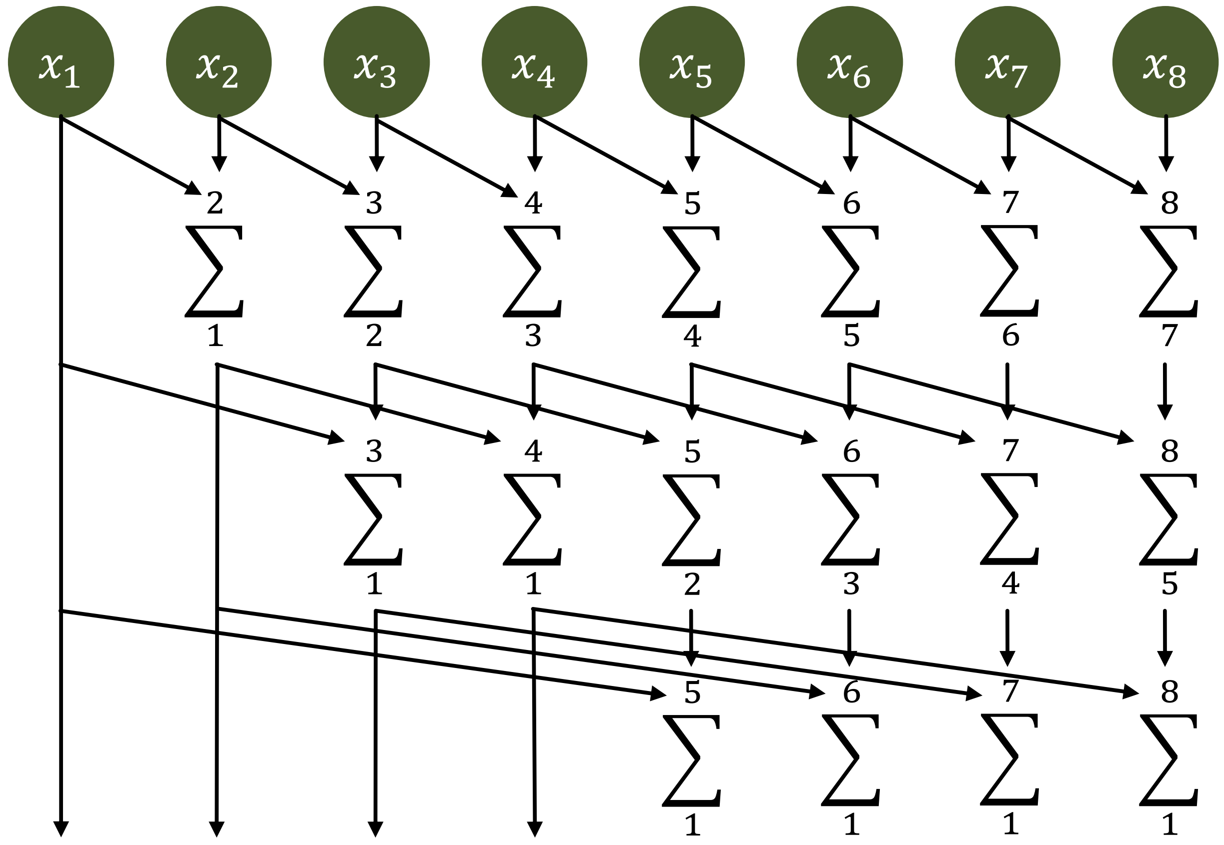
Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**



Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**

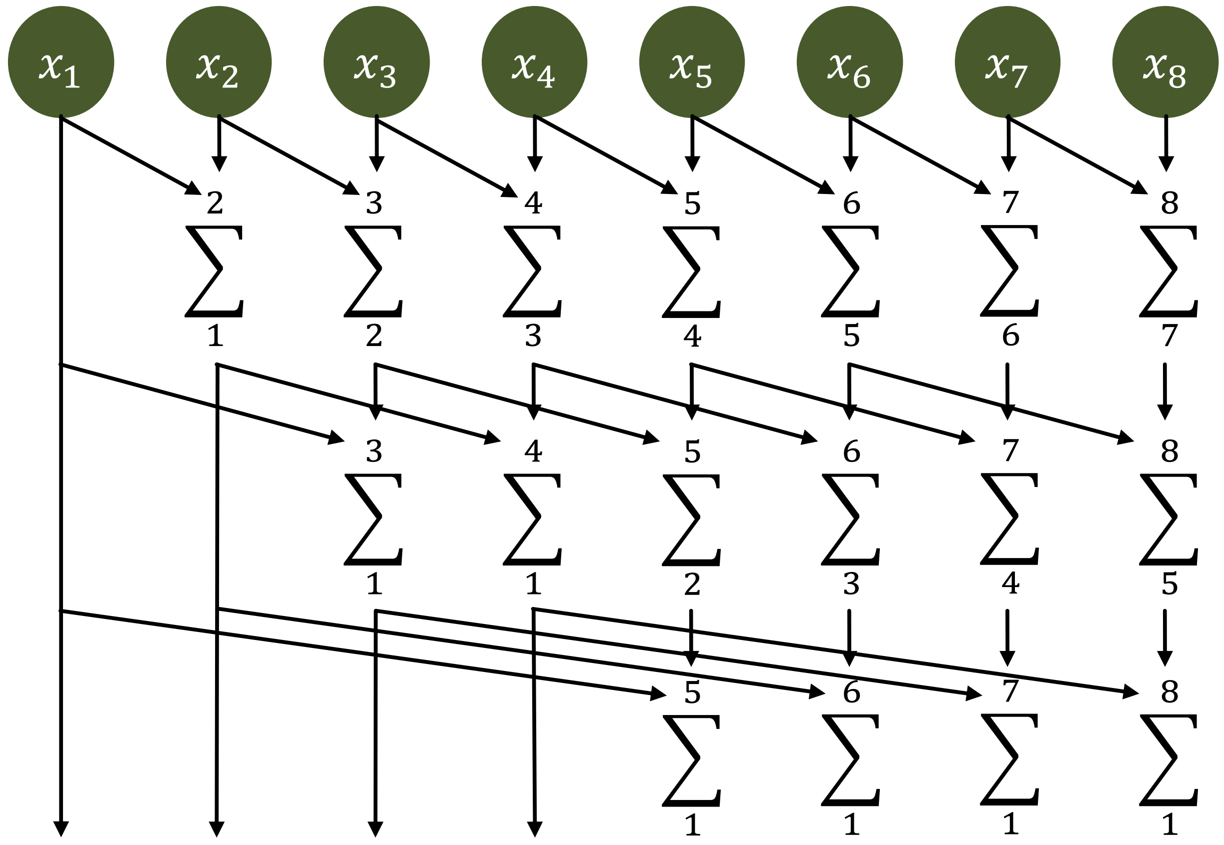


Class question: work?

(hint: number of count number of omitted ops)

Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**

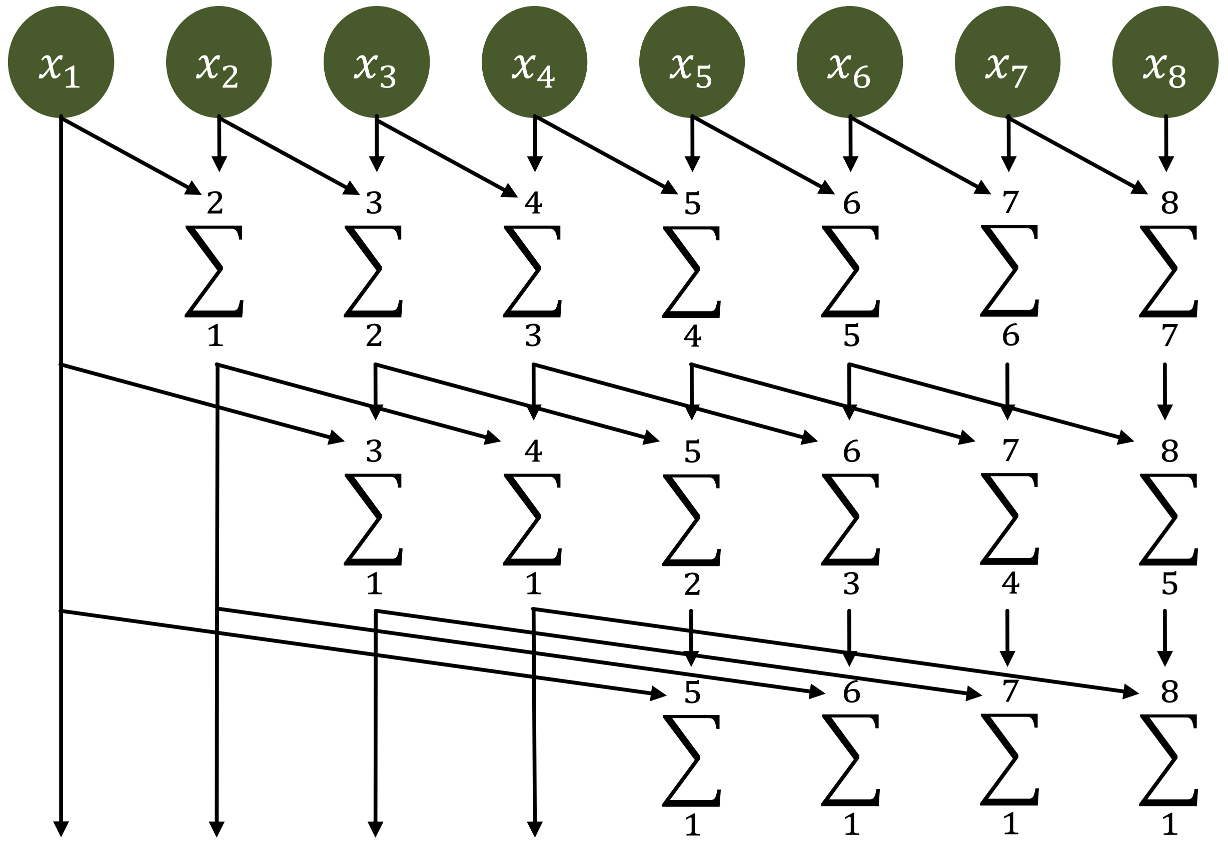


Class question: work?
(hint: number of count number of omitted ops)

$$W(n) = n\log_2 n - n + 1$$

Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**



Class question: work?

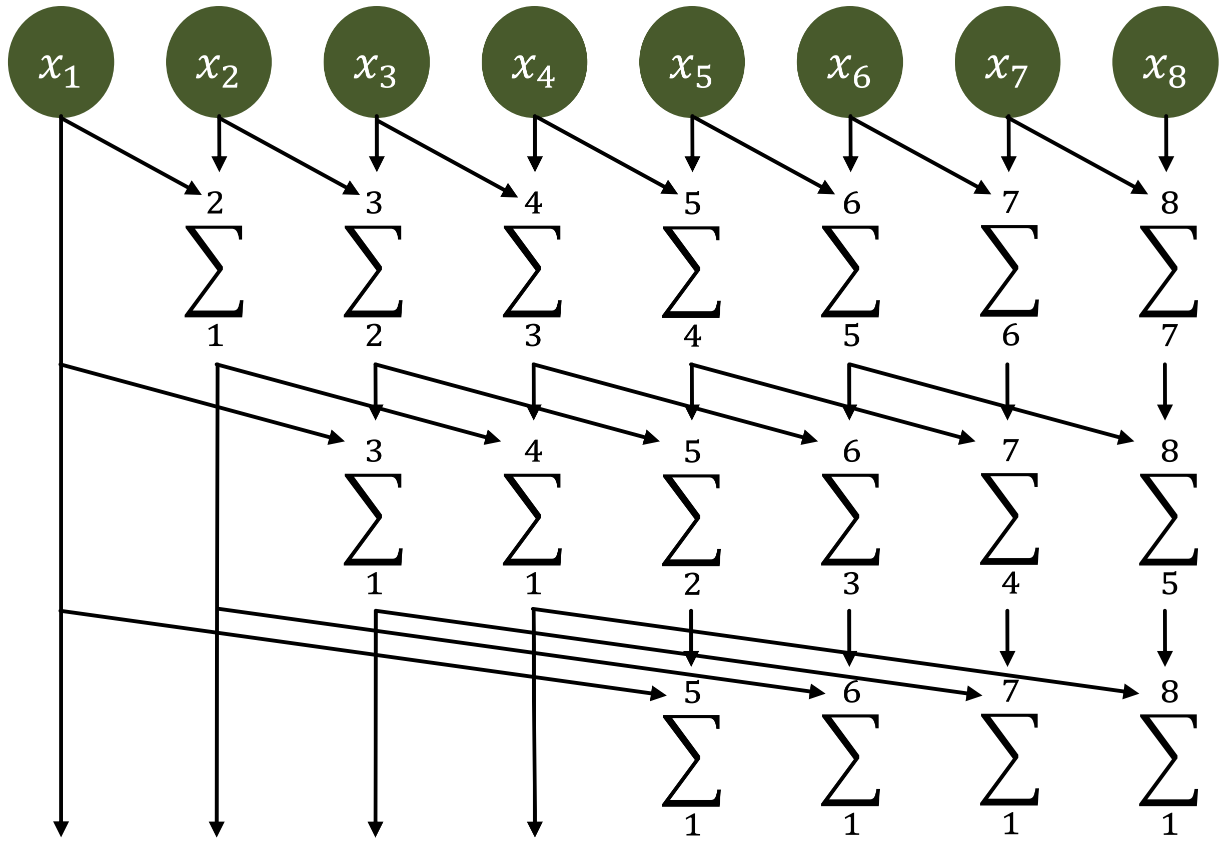
(hint: number of count number of omitted ops)

$$W(n) = n \log_2 n - n + 1$$

Class question: depth?

Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**



Class question: work?
(hint: number of count number of omitted ops)

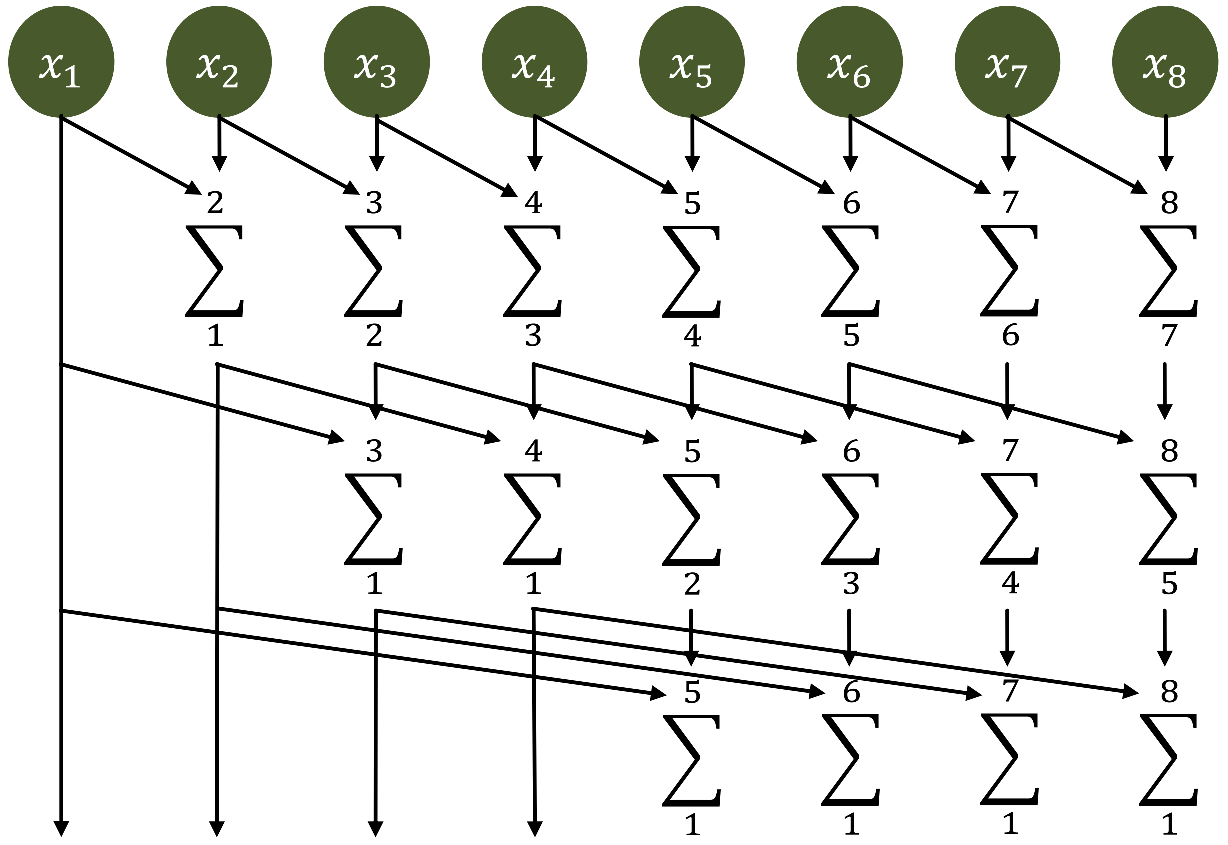
$$W(n) = n\log_2 n - n + 1$$

Class question: depth?

$$D(n) = \log_2 n$$

Oh no, not good, another algorithm to the rescue!

- **Dissemination/recursive doubling – another well-known algorithmic technique – similar to trees**



Class question: work?
(hint: number of count number of omitted ops)

$$W(n) = n \log_2 n - n + 1$$

Class question: depth?

$$D(n) = \log_2 n$$

Class question: is this now optimal?

Oh no, three non-optimal algorithms so far!

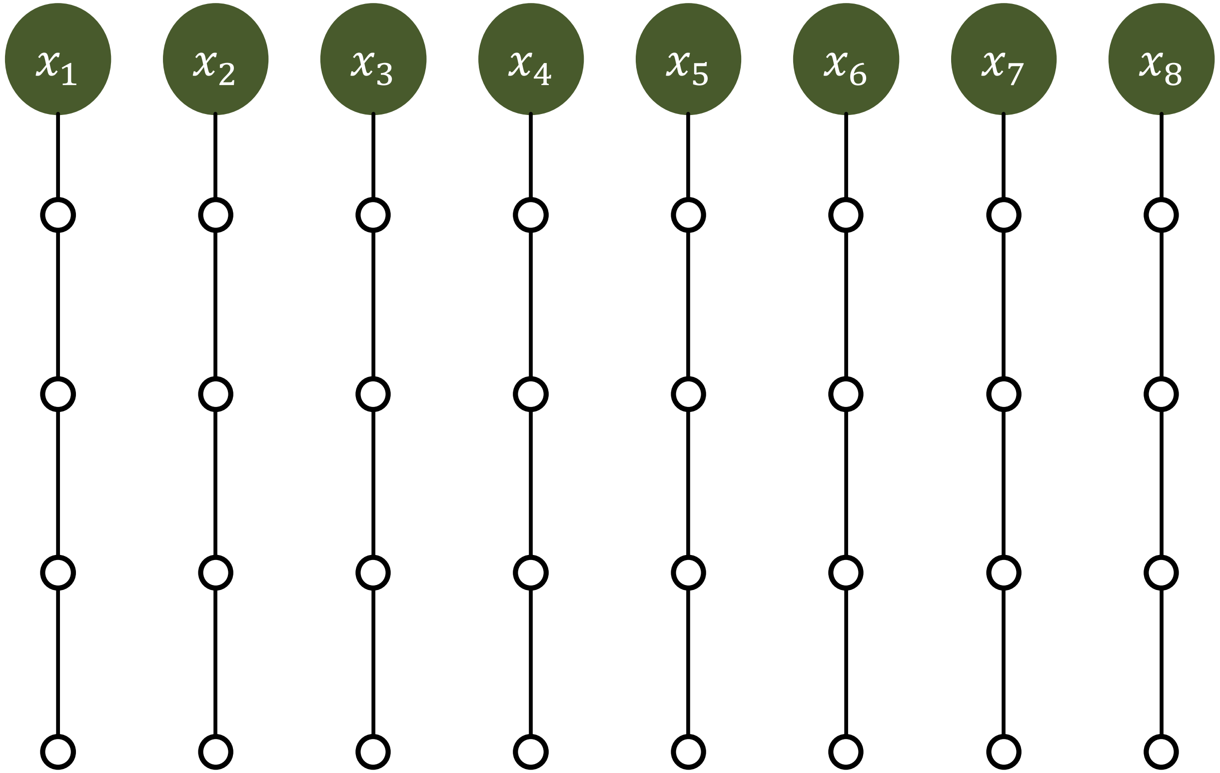
- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$

Oh no, three non-optimal algorithms so far!

- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$

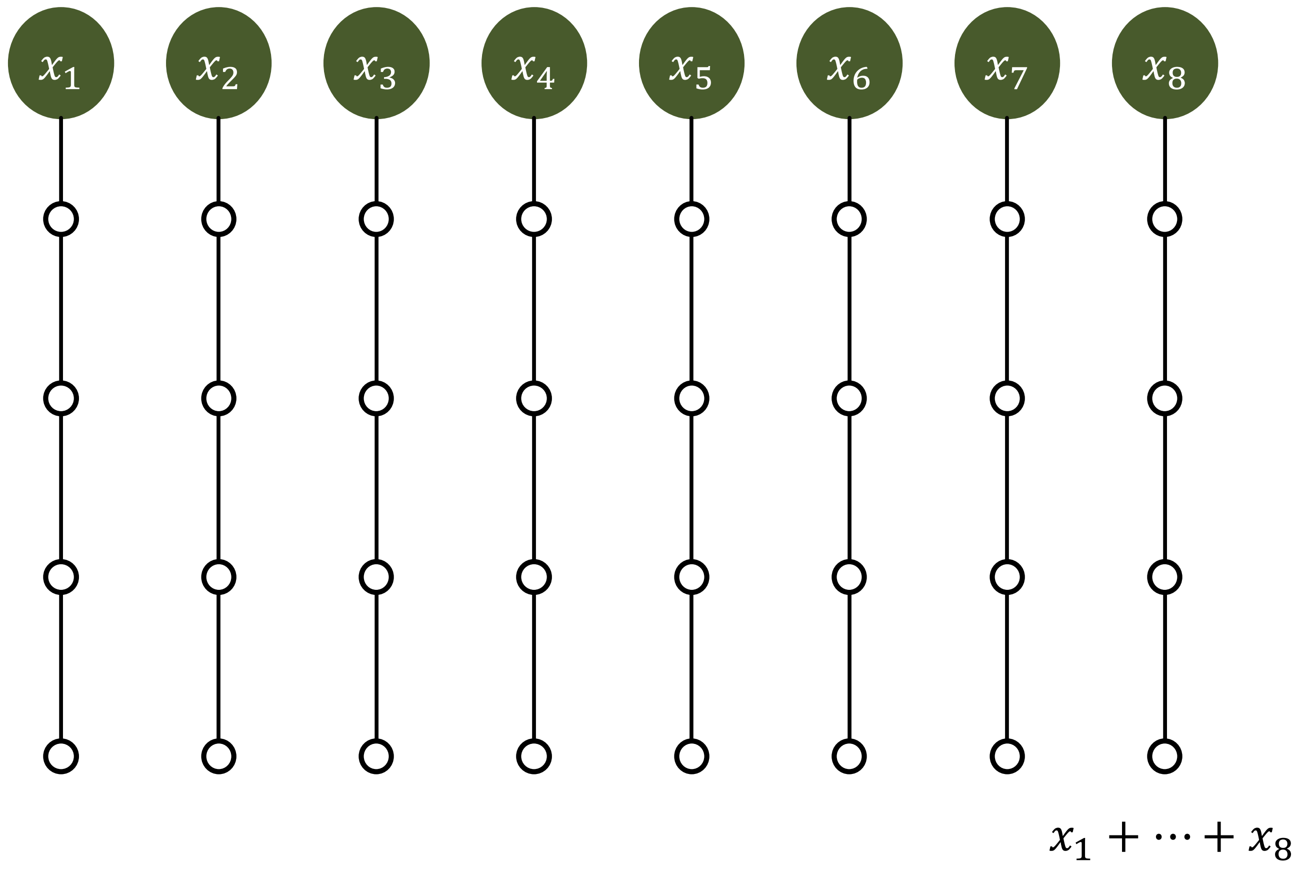
Oh no, three non-optimal algorithms so far!

- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$



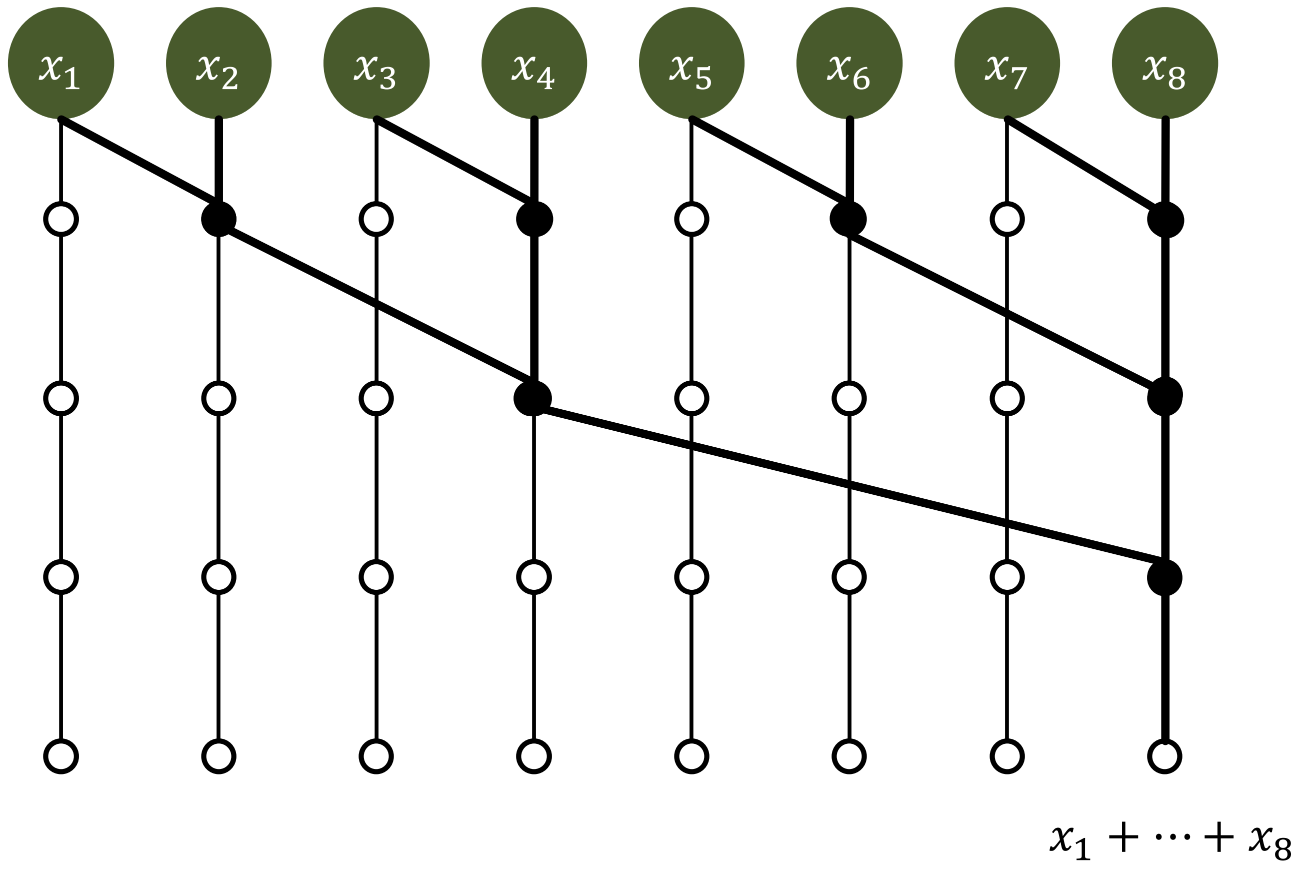
Oh no, three non-optimal algorithms so far!

- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$



Oh no, three non-optimal algorithms so far!

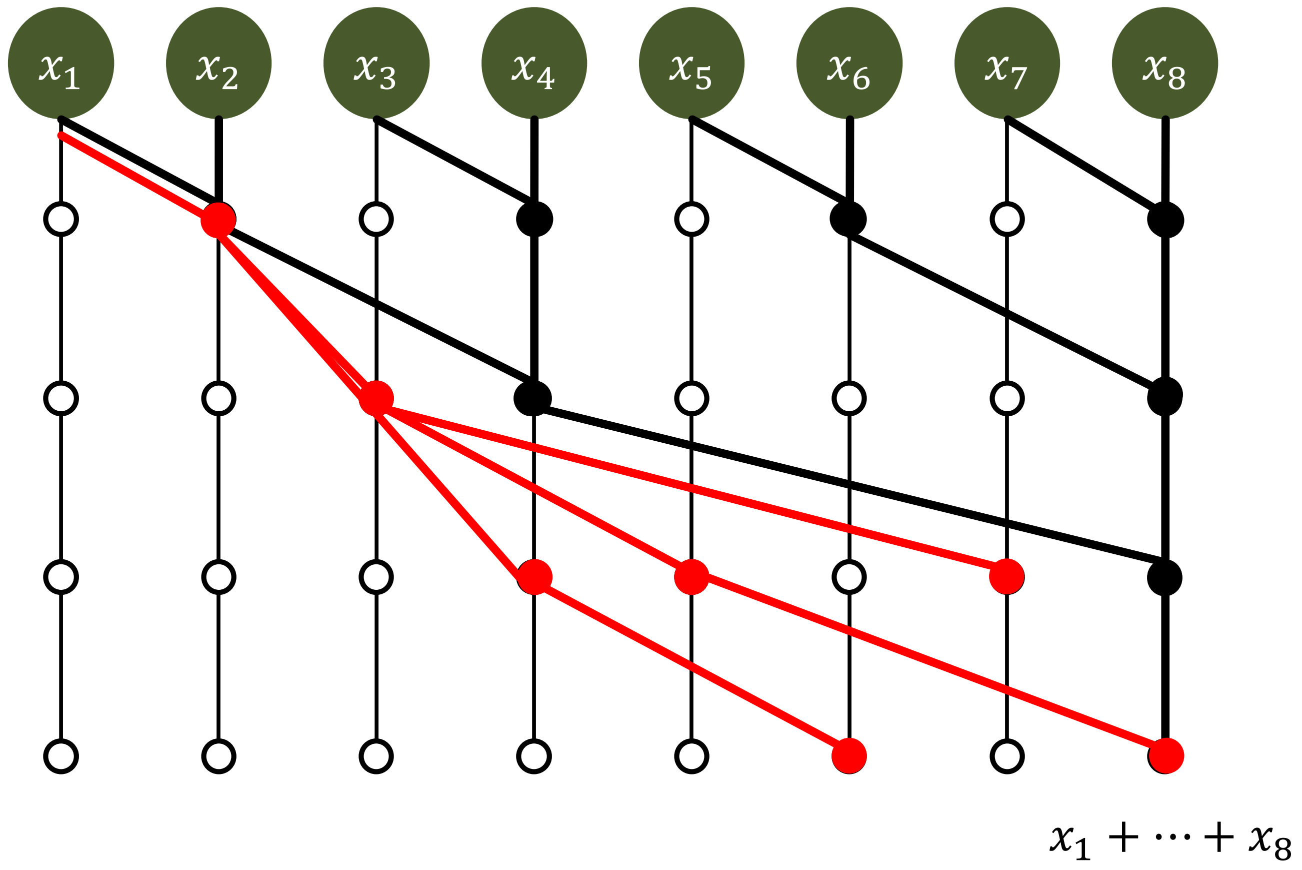
- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$



Output tree:
- leaves are all inputs, rooted at x_n
- binary due to binary operation
- $W = n - 1, D = D_o$

Oh no, three non-optimal algorithms so far!

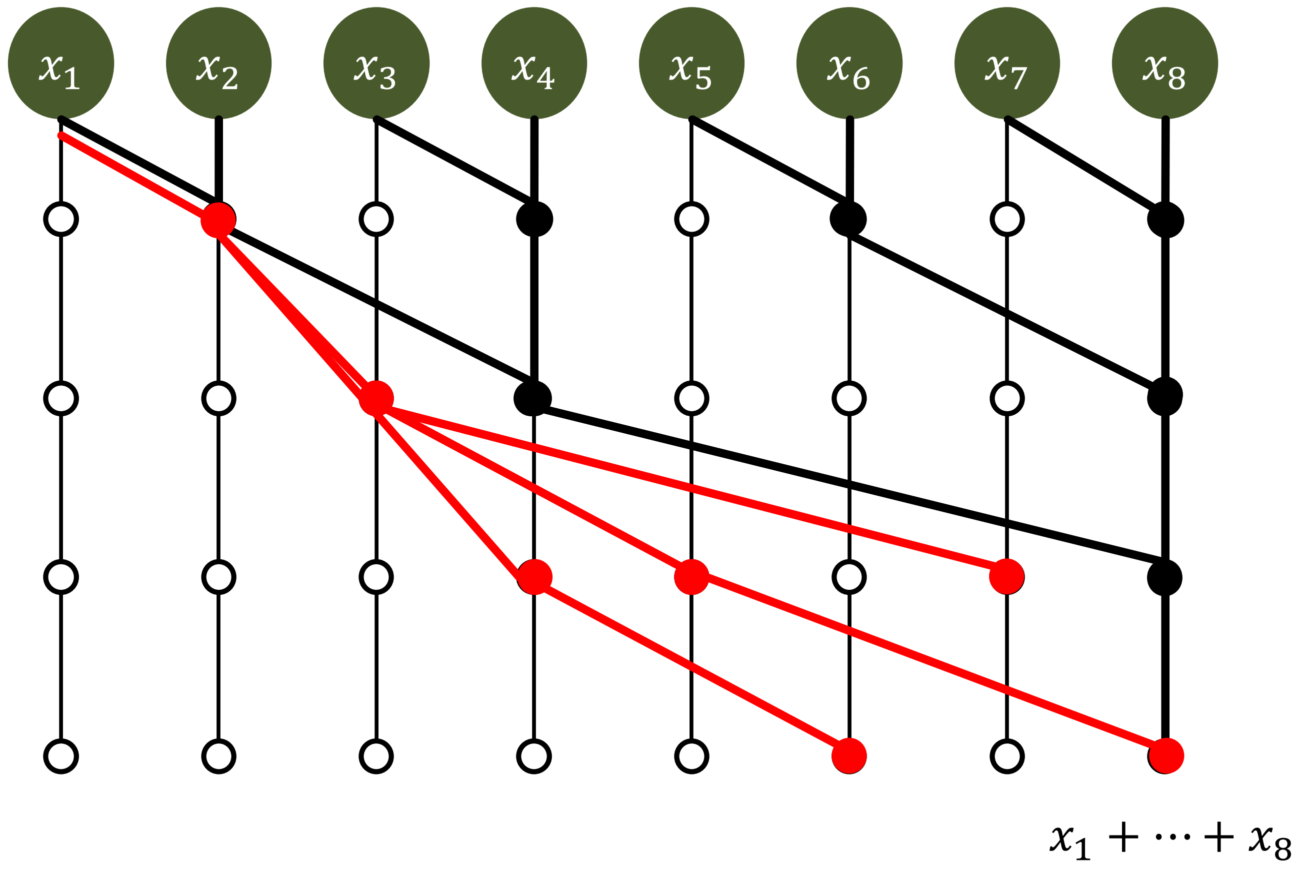
- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$



Output tree:
- leaves are all inputs, rooted at x_n
- binary due to binary operation
- $W = n - 1, D = D_o$

Oh no, three non-optimal algorithms so far!

- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$

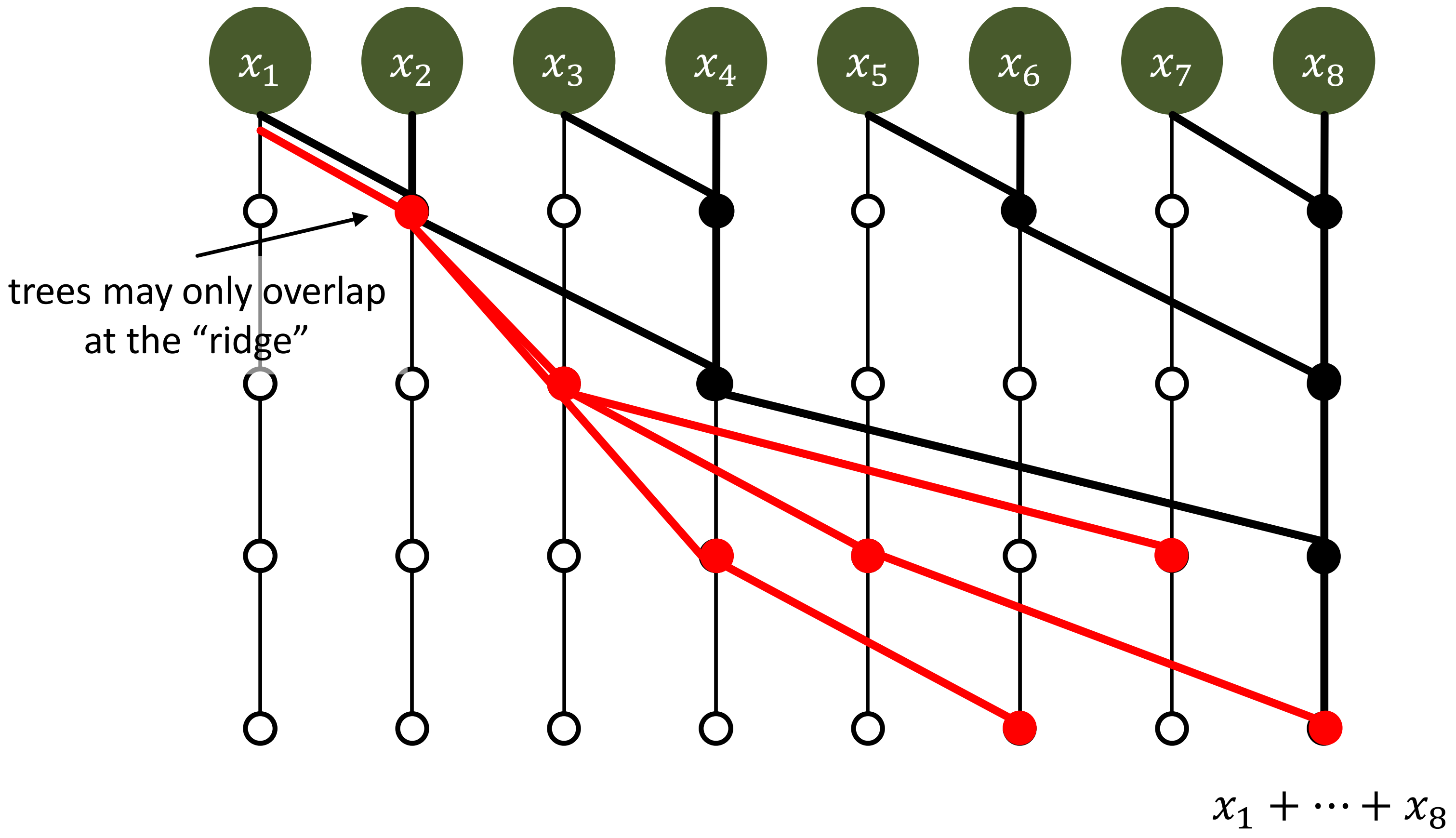


Output tree:
- leaves are all inputs, rooted at x_n
- binary due to binary operation
- $W = n - 1, D = D_o$

Input tree:
- rooted at x_1, leaves are all outputs
- not binary (simultaneous read)
- $W = n - 1$

Oh no, three non-optimal algorithms so far!

- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$

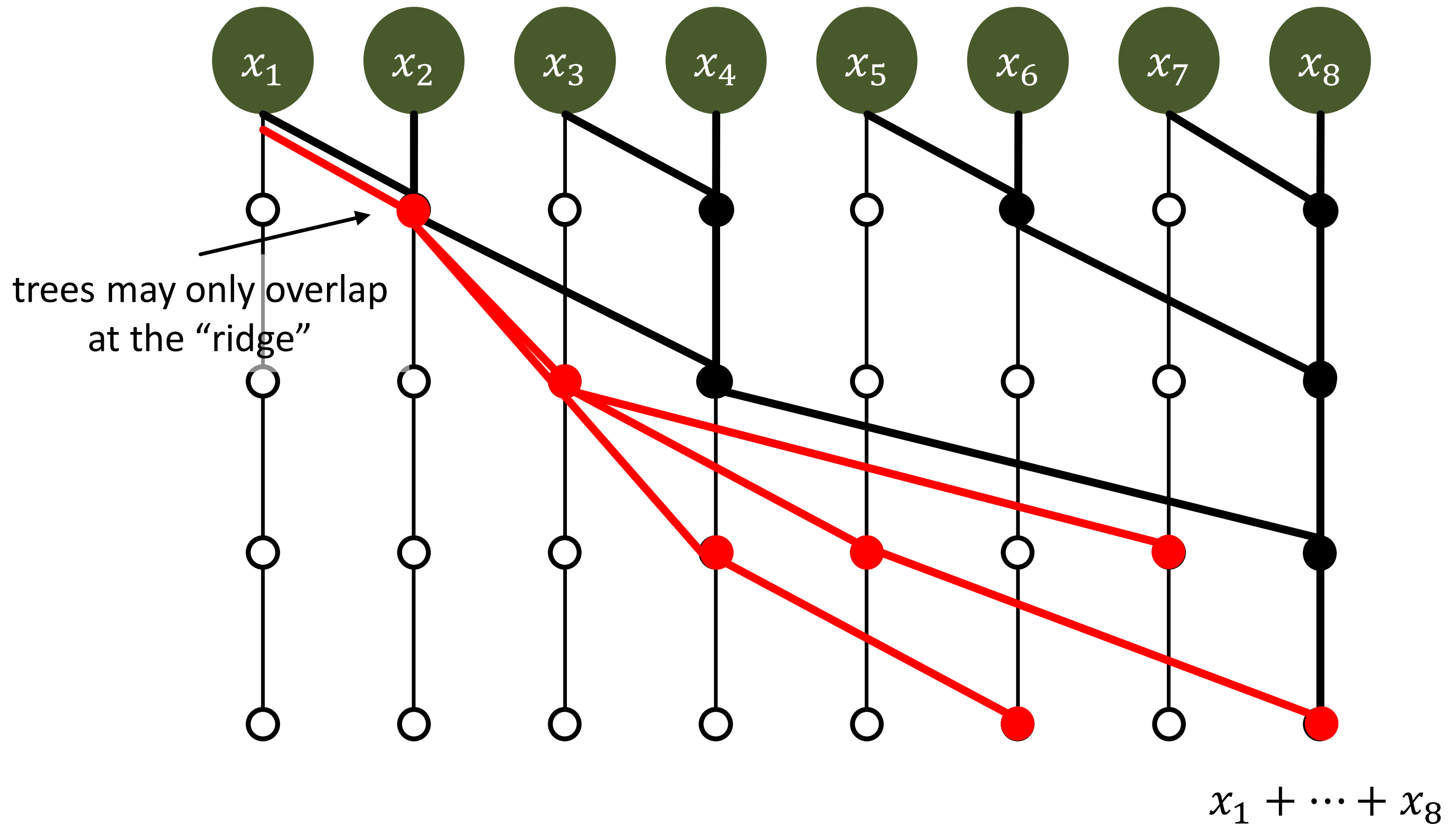


Output tree:
- leaves are all inputs, rooted at x_n
- binary due to binary operation
- $W = n - 1, D = D_o$

Input tree:
- rooted at x_1, leaves are all outputs
- not binary (simultaneous read)
- $W = n - 1$

Oh no, three non-optimal algorithms so far!

- **Obvious question: is there a depth- and work-optimal algorithm?**
  - This took years to settle! The answer is surprisingly: no
  - We know, for parallel prefix: $W + D \geq 2n - 2$



Output tree:

- leaves are all inputs, rooted at x_n
- binary due to binary operation
- $W = n - 1, D = D_o$

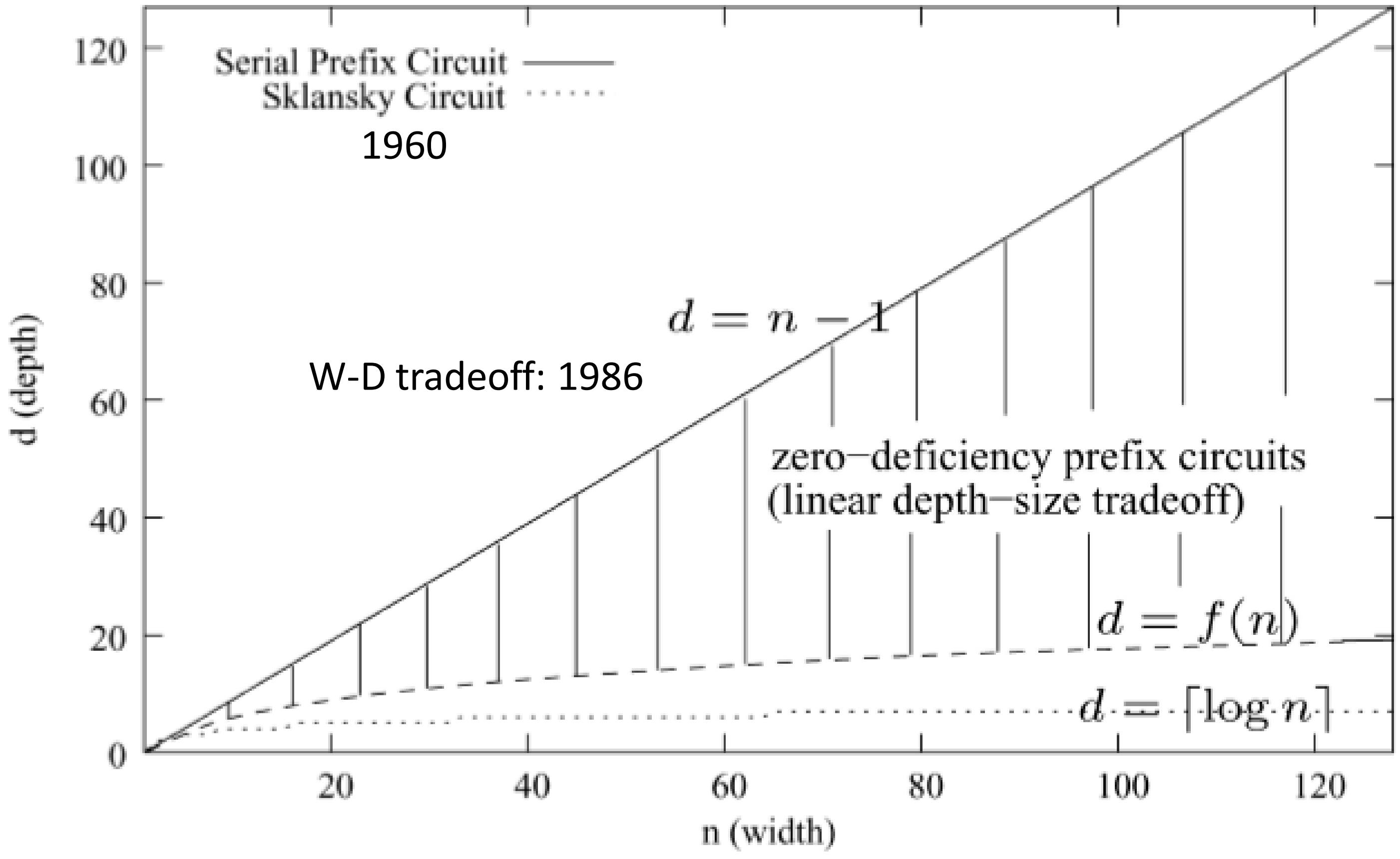
Input tree:

- rooted at x_1, leaves are all outputs
- not binary (simultaneous read)
- $W = n - 1$

Ridge can be at most D_o long!
Now add trees and subtract shared vertices:
$(n-1) + (n-1) - D_o = 2n - 2 - D_o \leq W$
q.e.d.

Work-Depth Tradeoffs and deficiency

"The deficiency of a prefix circuit c is defined as $\mathrm{def}(c) = W_c + D_c - (2n-2)$*"*



Latest 2006 result for zero-deficiency construction for $n > F(D+3) - 1$ ($f(n)$ is inverse)

From Zhu et al.: "Construction of Zero-Deficiency Parallel Prefix Circuits"

Work- and depth-optimal constructions

Work-optimal?

- $= n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth $D = \lceil \log_2 n \rceil$, $W \leq 4n$
    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)
    Has an unbounded fan-out! May thus not be practical ☹

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

Work- and depth-optimal constructions

Work-optimal?

- Only sequential! Why?
- $= n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth $D = \lceil \log_2 n \rceil$, $W \leq 4n$
    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)
    Has an unbounded fan-out! May thus not be practical ☹

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

Work- and depth-optimal constructions

$\boldsymbol{nn}$−1, thus DD=$2nn$−2−WW=nn−1 q.e.d. ⍰

Work-optimal**?**

- Only sequential! Why?
- $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth

    $D = \lceil \log_2 n \rceil$, $W \leq 4n$

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)

    Has an unbounded fan-out! May thus not be practical ☹

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

Work- and depth-optimal constructions

nn−1, thus DD=*2nn−2−WW=nn−1* q.e.d. ?

Work-optimal**?**

- Only sequential! Why?
- $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth
    $D = \lceil \log_2 n \rceil$, $W \leq 4n$
    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)
    Has an unbounded fan-out! May thus not be practical ☹

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

Work- and depth-optimal constructions

⌈ log 2 n⌉ log 2 log log 2 2 log 2 log 2 n⌉ *nn⌉ log 2 n⌉ , WW≤4nn*

nn−1, thus *DD*=*2nn*−*2*−*WW*=*nn*−*1* q.e.d. ?

Work-optimal**?**

- Only sequential! Why?
- $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)

    Has an unbounded fan-out! May thus not be practical ☹

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)

    Has an unbounded fan-out! May thus not be practical ☹

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

Work- and depth-optimal constructions

⌈ **log 2** n⌉ **log 2 log log 2 2 log 2** **log 2** n⌉ *nn⌉ log 2 n⌉ , WW≤4nn*

nn–1, thus DD=*2nn−2−WW=nn−1* q.e.d. ?

Work-optimal**?**

- Only sequential! Why?
- $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)

    Has an unbounded fan-out! May thus not be practical ☹

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)

- **Depth-optimal with bounded fan-out?**

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

Work- and depth-optimal constructions

⌈ log 2 n⌉ log 2 log log 2 2 log 2   log 2 n⌉ *nn⌉ log 2 n⌉ , WW≤4nn*

nn–1, thus DD=$2nn-2-WW=nn-1$ q.e.d. ⍰

Work-optimal**?**

- Only sequential! Why?
- $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)

    Has an unbounded fan-out! May thus not be practical ☹

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)
- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem

- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

Work- and depth-optimal constructions

⌈ log 2 n⌉ log 2 log log 2 2 log 2 log 2 n⌉ *nn⌉ log 2 n⌉ , WW≤4nn*

nn−1, thus *DD=2nn−2−WW=nn−1* q.e.d. ?

Work-optimal**?**

- Only sequential! Why?
- $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **Depth-optimal?**
  - Ladner and Fischer propose a construction for work-efficient circuits with minimal depth

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)

    Has an unbounded fan-out! May thus not be practical ☹

    Simple set of recursive construction rules (boring for class, check 1980's paper if needed)
- **Depth-optimal with bounded fan-out?**
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits
  - Some constructions exist, interesting open problem
  - Nice research topic to define optimal circuits

But why do we care about this prefix sum so much?

It's the simplest problem to demonstrate W-D tradeoffs

- And it's one of the most important parallel primitives

- $= \; a + b$ **(n-bit numbers)**
  - Starting with single-bit (full) adder for bit i

But why do we care about this prefix sum so much?

It's the simplest problem to demonstrate W-D tradeoffs

- And it's one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!

- $= a + b$ **(n-bit numbers)**
  - Starting with single-bit (full) adder for bit i

But why do we care about this prefix sum so much?

= aa+bb (n-bit numbers)

It's the simplest problem to demonstrate W-D tradeoffs

- And it's one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!

- **Simple first example: binary adder – $s = a + b$ (n-bit numbers)**
  - Starting with single-bit (full) adder for bit i

But why do we care about this prefix sum so much?

= aa+bb (n-bit numbers)

It’s the simplest problem to demonstrate W-D tradeoffs

- And it’s one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!

  - Starting with single-bit (full) adder for bit i
  - Starting with single-bit (full) adder for bit i

But why do we care about this prefix sum so much?

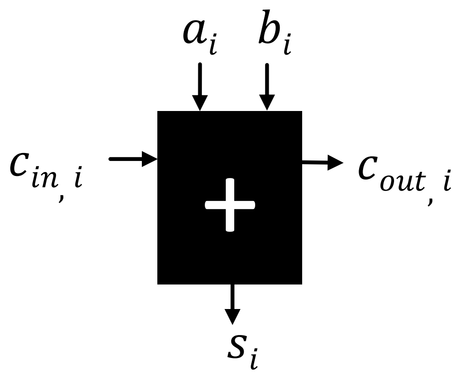
= aa+bb (n-bit numbers)

It's the simplest problem to demonstrate W-D tradeoffs

- And it's one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!

  - Starting with single-bit (full) adder for bit i
  - Starting with single-bit (full) adder for bit i



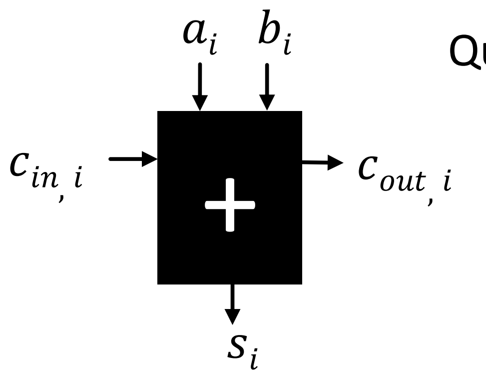
But why do we care about this prefix sum so much?

= aa+bb **(n-bit numbers)**

It's the simplest problem to demonstrate W-D tradeoffs

- And it's one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!
  - Starting with single-bit (full) adder for bit i
  - Starting with single-bit (full) adder for bit i



Question: what are the functions for s_i and $c_{out,i}$?

But why do we care about this prefix sum so much?

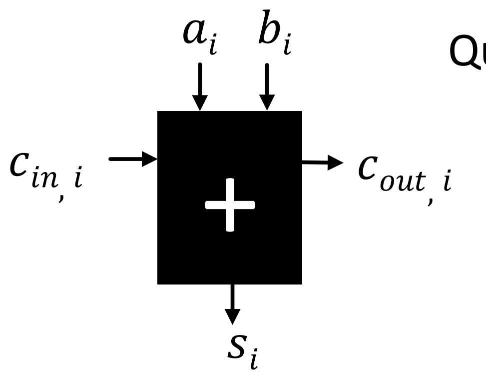
= aa+bb (n-bit numbers)

It's the simplest problem to demonstrate W-D tradeoffs

- And it's one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!

  - Starting with single-bit (full) adder for bit i
  - Starting with single-bit (full) adder for bit i



Question: what are the functions for s_i and $c_{out,i}$?

$$s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$$

$$c_{out,i} = a_i \text{ and } b_i \text{ or } c_{in,i} \text{ and } (a_i \text{ xor } b_i)$$

But why do we care about this prefix sum so much?

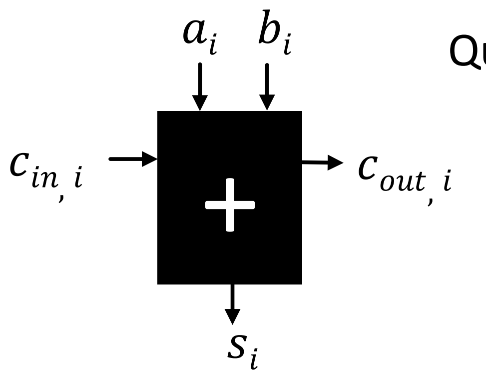
= aa+bb (n-bit numbers)

It’s the simplest problem to demonstrate W-D tradeoffs

- And it’s one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!

  - Starting with single-bit (full) adder for bit i
  - Starting with single-bit (full) adder for bit i

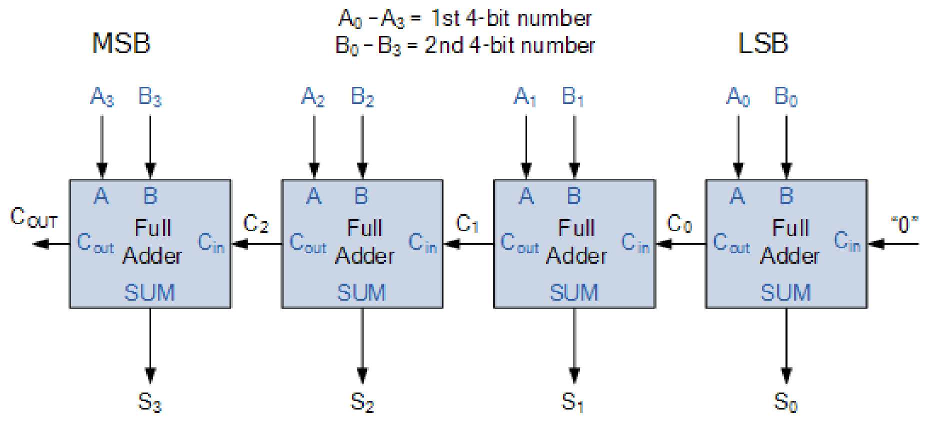


Question: what are the functions for s_i and $c_{out,i}$?

$$s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$$

$$c_{out,i} = a_i \text{ and } b_i \text{ or } c_{in,i} \text{ and } (a_i \text{ xor } b_i)$$

Example 4-bit ripple carry adder



source: electronics-tutorials.ws

But why do we care about this prefix sum so much?

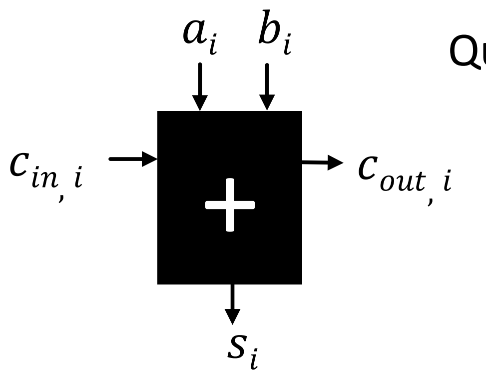
= aa+bb (n-bit numbers)

It's the simplest problem to demonstrate W-D tradeoffs

- And it's one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!

  - Starting with single-bit (full) adder for bit i
  - Starting with single-bit (full) adder for bit i



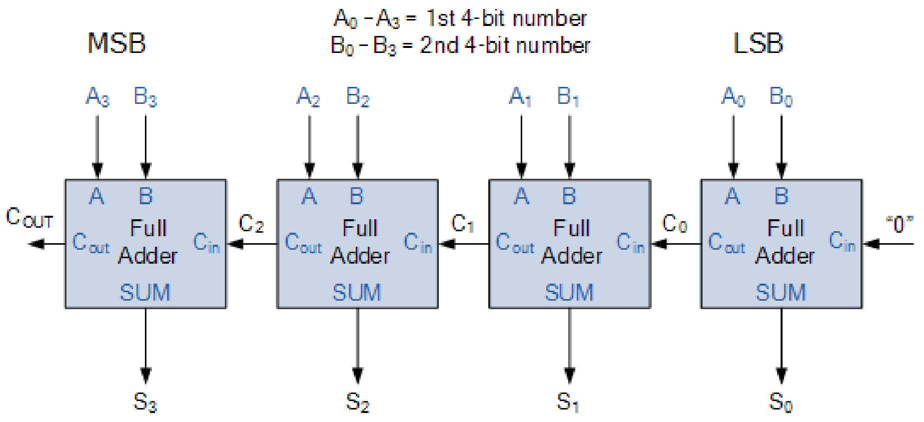
Question: what are the functions for s_i and $c_{out,i}$?

$$s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$$

$$c_{out,i} = a_i \text{ and } b_i \text{ or } c_{in,i} \text{ and } (a_i \text{ xor } b_i)$$

Show example 4-bit addition!

Example 4-bit ripple carry adder



source: electronics-tutorials.ws

But why do we care about this prefix sum so much?

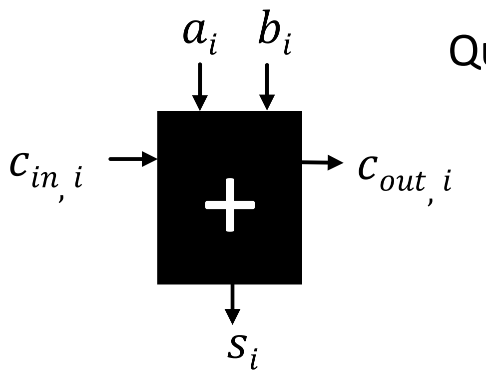
= aa+bb (n-bit numbers)

It's the simplest problem to demonstrate W-D tradeoffs

- And it's one of the most important parallel primitives

- **Prefix summation as function composition is extremely powerful!**
  - Many seemingly sequential problems can be parallelized!

  - Starting with single-bit (full) adder for bit i
  - Starting with single-bit (full) adder for bit i



Question: what are the functions for s_i and $c_{out,i}$?

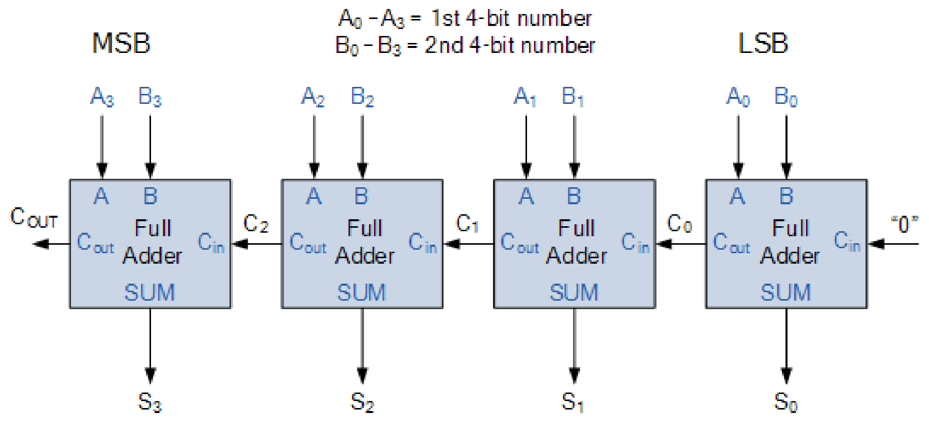
$$s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$$

$$c_{out,i} = a_i \text{ and } b_i \text{ or } c_{in,i} \text{ and } (a_i \text{ xor } b_i)$$

Show example 4-bit addition!

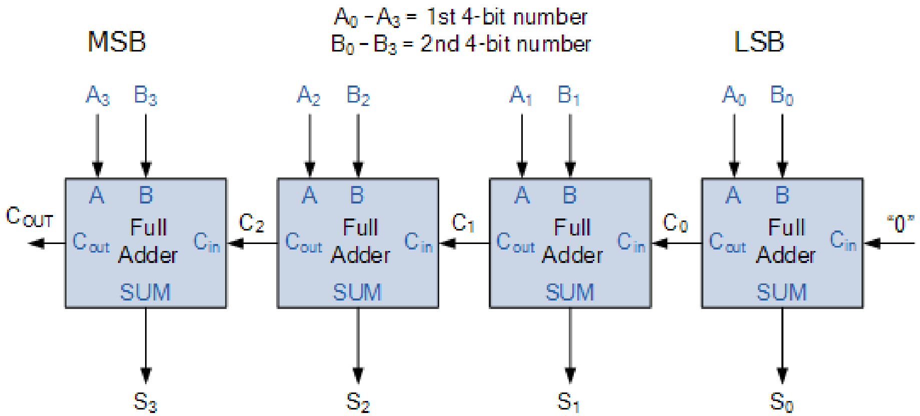
Question: what is work and depth?

Example 4-bit ripple carry adder



source: electronics-tutorials.ws

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

We only want s!

- $, c_{in,2}, \dots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0

    Encode state as 1-hot vector: $q_0 = \binom{1}{0}$, $q_1 = \binom{0}{1}$
  - Each full adder updates the carry bit state according to a_i and b_i

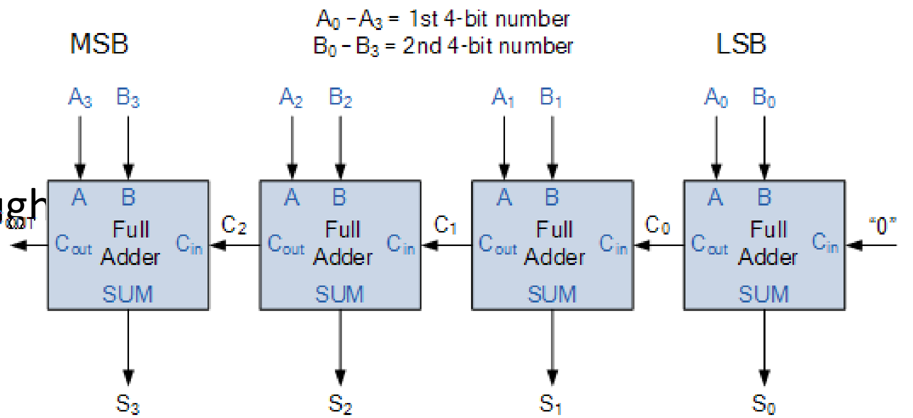
    State update is now represented by matrix operator, depending on a_i and b_i ($M_{a_i b_i}$):

    $$M_{00} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, M_{10} = M_{01} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace "+" with "and" and "*" with "or"

    Prefix sum on the states computes now all carry bits in parallel!

- **Example: a=011, b=101 → M_{11}, M_{10}, M_{01}**
  - Scan computes: $M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$; $M_{11}M_{10} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$; $M_{11}M_{10}M_{01} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$ in parallel
  - All carry states and s_i can now be computed in parallel by multiplying scan result with q_0

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

1 c in,1 , c in,2 cc c in,2 iinn,2 $c_{out,i}$ = ..., a_i and b_i or $c_{in,i}$ and (a_i xor b_i) though

We **only want s!** $\quad s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$

- Requires $c_{in,n}, 1_{in,1}, c_{in,2}, \ldots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0

    Encode state as 1-hot vector: $q_0 = \binom{1}{0}$, $q_1 = \binom{0}{1}$
  - Each full adder updates the carry bit state according to a_i and b_i

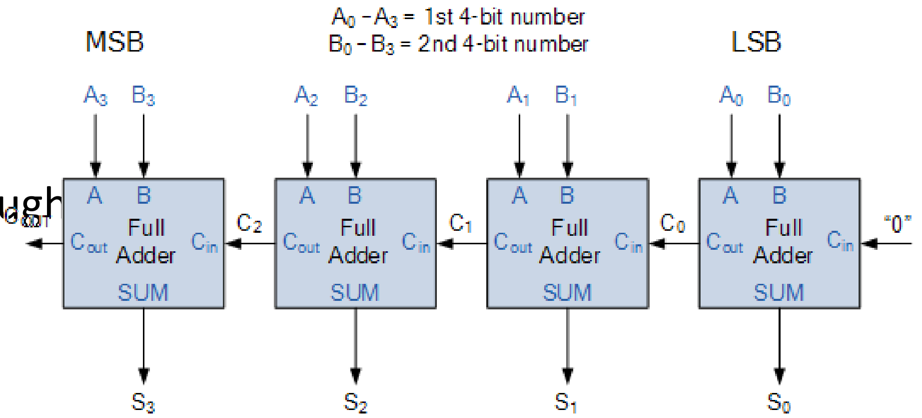
    State update is now represented by matrix operator, depending on a_i and b_i ($M_{a_i b_i}$):

    $$M_{00} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, M_{10} = M_{01} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace "+" with "and" and "*" with "or"

    Prefix sum on the states computes now all carry bits in parallel!

- **Example: a=011, b=101 → M_{11}, M_{10}, M_{01}**
  - Scan computes: $M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$; $M_{11}M_{10} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$; $M_{11}M_{10}M_{01} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$ in parallel
  - All carry states and s_i can now be computed in parallel by multiplying scan result with q_0

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

$1\ c\ in,1\ ,\ c\ in,2\ cc\ c\ in,2\ iinn,2$ $c_{out,i} = a_i$ and b_i or $c_{in,i}$ and $(a_i$ xor $b_i)$ $,\dots,\ c\ in,n\ c\ c\ in,n\ iinn,n\ c\ in,n$ though

We **only want s!** $s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$

- Requires $c\ in, n, 1\ in, 1\ , c_{in,2}, \dots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0

    Encode state as 1-hot vector: $q_0 = \binom{1}{0}$, $q_1 = \binom{0}{1}$
  - Each full adder updates the carry bit state according to a_i and b_i

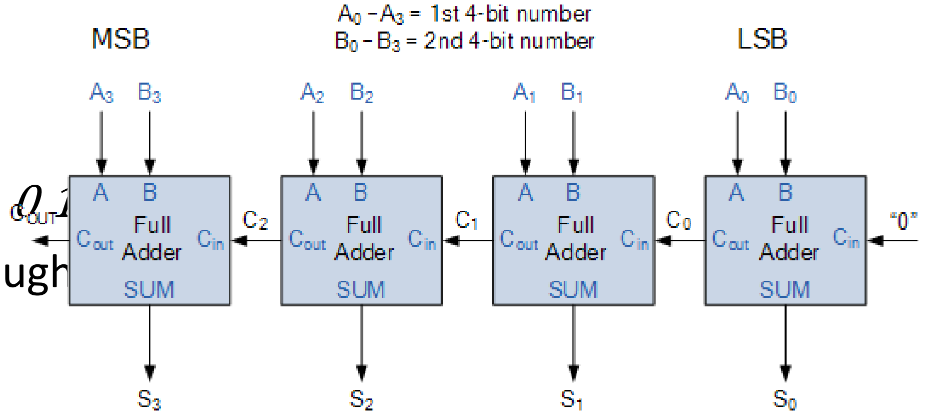
    State update is now represented by matrix operator, depending on a_i *and* b_i ($M_{a_i b_i}$):

    $$M_{00} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, M_{10} = M_{01} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace "+" with "and" and "*" with "or"

    Prefix sum on the states computes now all carry bits in parallel!

- **Example: a=011, b=101 → M_{11}, M_{10}, M_{01}**
  - Scan computes: $M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$; $M_{11}M_{10} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$; $M_{11}M_{10}M_{01} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$ in parallel

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

$q\,0 =\; 1\,0\;\; 1\,0\,1\,1\,0\,0\,1\,0\;\; 1\,0$ $c_{out} = a_i$ and b_i or c_{in} and (a_i xor b_i) $\ldots$

$1\; c\; in,1\,,\; c\; in,2\; \ldots,\; c\; in,n$ $s = a_i$ xor b_i xor c_{in} though

We **only want s!**

- Requires $c\; in, n, 1\; in, 1\,, c_{in,2}, \ldots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0

    Encode state as 1-hot vector: $q\;0\,0\,0\; = \binom{1}{0}, q_1 = \binom{0}{1}$

    Encode state as 1-hot vector: $q_0 = \binom{1}{0}, q_1 = \binom{0}{1}$
  - Each full adder updates the carry bit state according to a_i and b_i

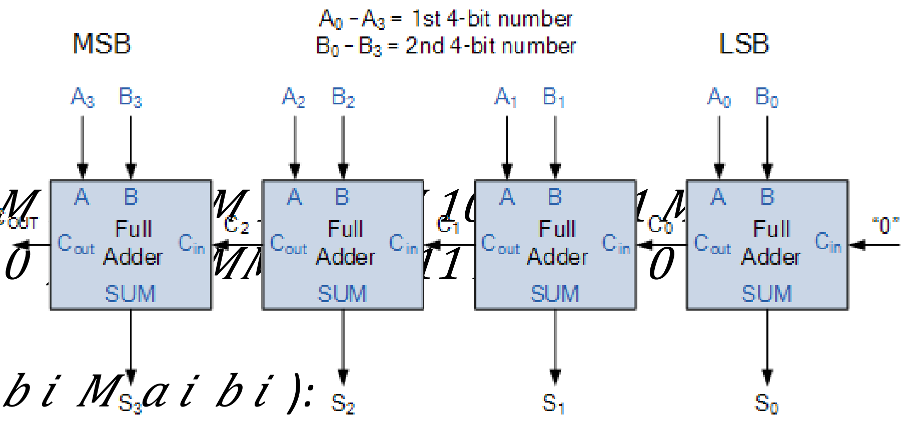
    State update is now represented by matrix operator, depending on a_i and b_i ($M_{a_i b_i}$):

    $$M_{00} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, M_{10} = M_{01} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace "+" with "and" and "*" with "or"

    Prefix sum on the states computes now all carry bits in parallel!

- **Example: a=011, b=101 → M_{11}, M_{10}, M_{01}**
  - Scan computes: $M_{..} = \begin{pmatrix} 0 & 0 \\ \ldots \end{pmatrix}$; $M_{..} M_{..} = \begin{pmatrix} 0 & 0 \\ \ldots \end{pmatrix}$; $M_{..} M_{..} M_{..} = \begin{pmatrix} 0 & 0 \\ \ldots \end{pmatrix}$ in parallel

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

0 M 00 = 1 1 0 0 1 1 0 0 1 1 0 0 0 1 1 1 0 0 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 1 0 , M
$c_{out,i} = (a_i$ and $b_i)$ or $(c_{in,i}$ and $(a_i$ xor $b_i))$

01 01 M 01 = 0 1 1 0 0 1 1 0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 0

$s_i = a_i$ xor b_i xor $c_{in,i}$

0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1

and b i bb b i ii b i (M a i b i MM M a i b i a i aa a i ii a i b i bb b i ii b i M a i b i):

and b i bb b i ii b i

q 0 = 1 0 1 0 1 1 0 0 1 0 1 0 , q 1 qq q 1 1 q 1 = 0 1 0 1 0 0 1 1 0 1 0 1

1 c in,1 , c in,2 cc c in,2 iinn,2 c in,2 ,…, c in,n cc c in,n iinn,nn c in,n though ?

We **only want s!**

- Requires $c\ in, n, 1\ in, 1$, $c_{in,2}, \dots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0

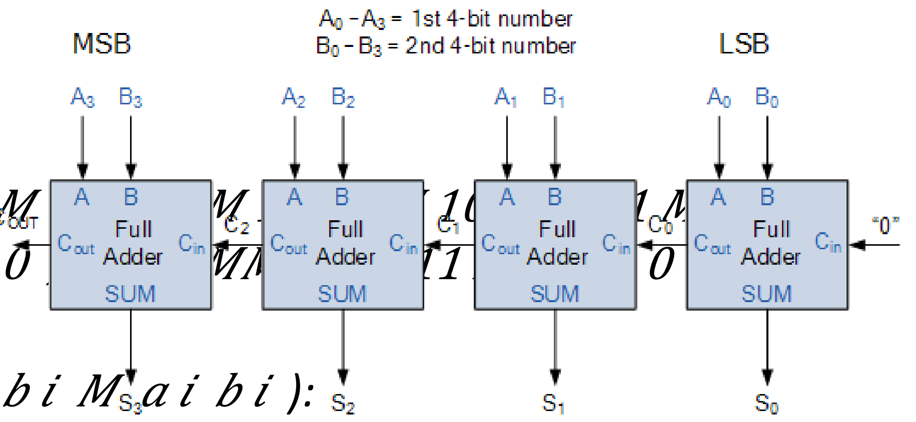
    $M\ 0\ 00\ 00 = \begin{pmatrix}1 & 1\\0 & 0\end{pmatrix}, M_{10} = M_{01} = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}, M_{11} = \begin{pmatrix}0 & 0\\1 & 1\end{pmatrix}$

    Encode state as 1-hot vector: $q_0 = \binom{1}{0}$, $q_1 = \binom{0}{1}$
  - Each full adder updates the carry bit state according to a_i and b_i

    State update is now represented by matrix operator, depending on a_i *and* b_i $(M_{a_i b_i})$:

    $M_{00} = \begin{pmatrix}1 & 1\\0 & 0\end{pmatrix}, M_{10} = M_{01} = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}, M_{11} = \begin{pmatrix}0 & 0\\1 & 1\end{pmatrix}$

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

$c_{out,i} = a_i$ and b_i or $c_{in,i}$ and $(a_i$ xor $b_i)$

$s_i = a_i$ xor b_i xor $c_{in,i}$

0 M 00 = 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 , M ...

01 01 M 01 = 0 1 1 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 0 ...

0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1

and b i bb b i ii b i (M a i b i MM M a i b i a i aa a i ii a i b i bb b i ii b i M a i b i):

and b i bb b i ii b i

q 0 = 1 0 1 0 1 1 0 0 1 0 1 0 , q 1 qq q 1 1 q 1 = 0 1 0 1 0 0 1 1 0 1 0 1

1 c in,1 , c in,2 cc c in,2 iinn,2 c in,2 ,..., c in,n cc c in,n iinn,nn c in,n though ?

We **only want s!**

- Requires $c_{in,1}, c_{in,2}, \ldots, c_{in,n}$ though ☹

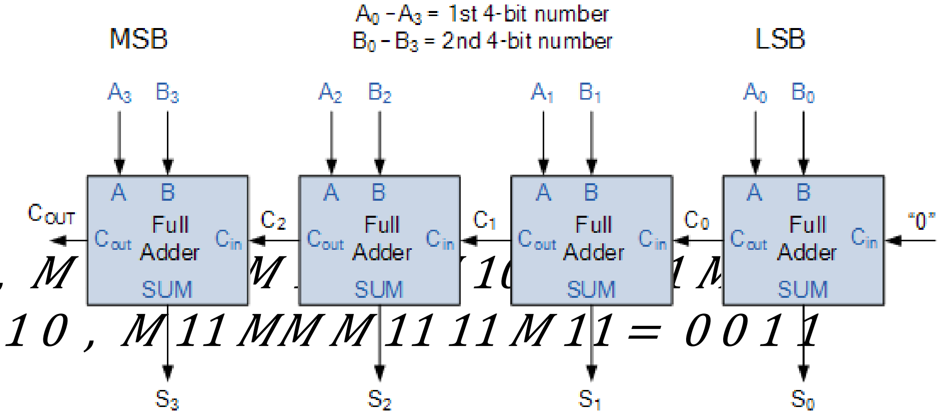
- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace "+" with "and" and "*" with "or"

    Prefix sum on the states computes now all carry bits in parallel!
  - Each full adder updates the carry bit state according to a_i and b_i

    State update is now represented by matrix operator, depending on a_i and b_i ($M_{a_i b_i}$):

$$M_{00} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, M_{10} = M_{01} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, M_{11} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

M 10 MM M 10 1100 M 10 $c_{out,i} = a_i$ and b_i or $c_{in,i}$ and $(a_i \text{ xor } b_i)$ **M 01 MM M 01 0011 M 01**

$s_i = a_i$ xor b_i xor $c_{in,i}$

0 M 00 = 1 1 0 0 1 1 0 0 1 1 1 0 0 1 1 1 0 0 0 1 1 0 0 1 1 0 0 1 1 0 0 , M

01 01 M 01 = 0 1 1 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 0 , M 11 MM M 11 11 M 11 = 0 0 1 1
0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1

and b i bb b i ii b i (M a i b i MM M a i b i a i aa a i ii a i b i bb b i ii b i M a i b i):

and b i bb b i ii b i

q 0 = 1 0 1 0 1 1 0 0 1 0 1 0 , q 1 qq q 1 1 q 1 = 0 1 0 1 0 0 1 1 0 1 0 1

1 c in,1 , c in,2 cc c in,2 iinn,2 c in,2 ,…, c in,n cc c in,n iinn,nn c in,n though ⍰

We **only want** s**!**

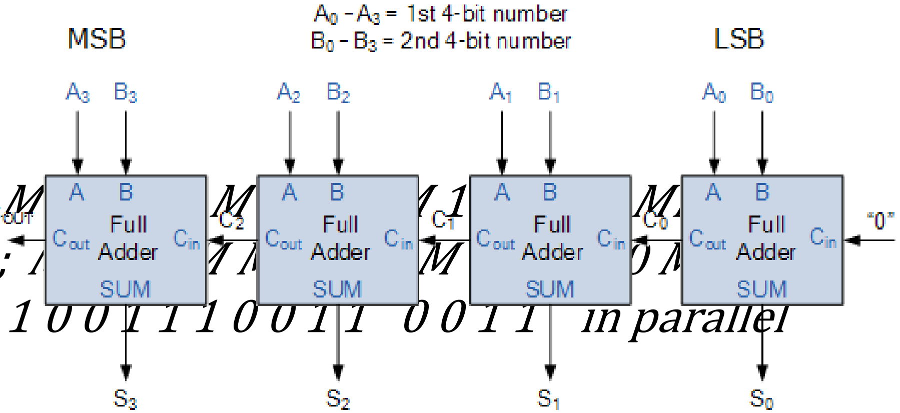
- Requires $c\ in, n, 1\ in, 1$, $c_{in,2}, \dots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace "+" with "and" and "*" with "or"

    Prefix sum on the states computes now all carry bits in parallel!
  - Each full adder updates the carry bit state according to a_i and b_i

- **Example: a=011, b=101 →** $\boldsymbol{M\ 11\ ,11\ 1\ 11\ , M_{10}, M_{01}}$

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

$c_{out,i} = a_i$ and b_i or $c_{in,i}$ and (a_i xor b_i)

$s_i = a_i$ xor b_i xor $c_{in,i}$

1 M 11 = 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 ; M ... M 1 ... M
10 M 10 = 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 ; ... M ... 0 M
10 10 M 10 M 01 MM M 01 01 M 01 = 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 in parallel

M 10 MM M 10 1100 M 10 , M 01 MM M 01 0011 M 01

0 M 00 = 1 1 0 0 1 1 0 0 1 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 0 , M 10 MM M 10 10 M 10 = M 01 MM M
01 01 M 01 = 0 1 1 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 0 , M 11 MM M 11 11 M 11 = 0 0 1 1
0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1

and b i bb b i ii b i (M a i b i MM M a i b i a i aa a i ii a i b i bb b i ii b i M a i b i):

and b i bb b i ii b i

q 0 = 1 0 1 0 1 1 0 0 1 0 1 0 , q 1 qq q 1 1 q 1 = 0 1 0 1 0 0 1 1 0 1 0 1

1 c in,1 , c in,2 cc c in,2 iinn,2 c in,2 ,..., c in,n cc c in,n iinn,nn c in,n though ?

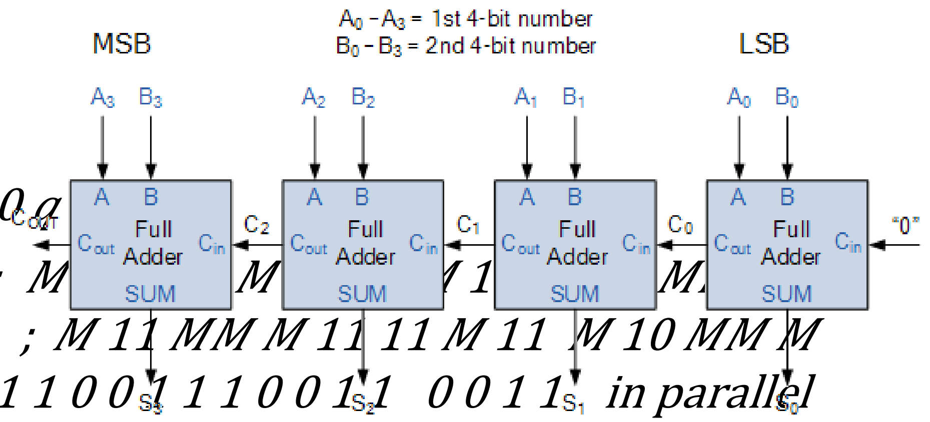
We **only want** s**!**

- Requires c in, n, 1 in, 1 , $c_{in,2}, \dots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace “+” with “and” and “*” with “or”

    Prefix sum on the states computes now all carry bits in parallel!

Seems very sequential, can this be parallelized?



source: electronics-tutorials.ws

can now be computed in parallel by multiplying scan result with $c_{out,i} = a_i$ and b_i or $c_{in,i}$ and $(a_i$ xor $b_i)$ $s_i = a_i$ xor b_i xor $c_{in,i}$

1 M 11 = 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 ; M ... M ... M 1 ... M
10 M 10 = 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 ; M 11 MM M 11 11 M 11 M 10 MM M
10 10 M 10 M 01 MM M 01 01 M 01 = 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1 in parallel

M 10 MM M 10 1100 M 10 , M 01 MM M 01 0011 M 01

0 M 00 = 1 1 0 0 1 1 0 0 1 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 1 1 0 0 , M 10 MM M 10 10 M 10 = M 01 MM M 01 01 M 01 = 0 1 1 0 0 1 1 0 0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 0 , M 11 MM M 11 11 M 11 = 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0 0 1 1

and *b i bb b i ii b i (M a i b i MM M a i b i a i aa a i ii a i b i bb b i ii b i M a i b i):*

and *b i bb b i ii b i*

q 0 = 1 0 1 0 1 1 0 0 1 0 1 0 , q 1 qq q 1 1 q 1 = 0 1 0 1 0 0 1 1 0 1 0 1

1 c in,1 , c in,2 cc c in,2 iinn,2 c in,2 ,..., c in,n cc c in,n iinn,nn c in,n though ☹

We **only want** s**!**

- Requires $c\ in, n, 1\ in, 1$, $c_{in,2}, \dots, c_{in,n}$ though ☹

- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0
  - Operator composition is defined on algebraic ring ({0, 1, or, and}) – i.e., replace "+" with "and" and "*" with "or"

Prefix sums as magic bullet for other seemingly sequential algorithms

Any time a sequential chain can be modeled as function composition!

- , … , f_n be an ordered set of functions and $f_0(x) = x$
- Define ordered function compositions: $f_1(x); f_2(f_1(x)); \ldots ; f_n(\ldots f_1(x))$
- If we can write function composition $g(x) = f_i(f_{i-1}(x)$ as $g = f_i \circ f_{i-1}$ then we can compute $\circ$ with a prefix sum!

  We saw an example with the adder (M_{ab} were our functions)

- **Example: linear recurrence $\boldsymbol{f_i(x) = a_i f_{i-1}(x) + b_i}$ with $\boldsymbol{f_0(x)}$=x**
  - Write as matrix form $f_i\binom{x}{1} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} f_{i-1}\binom{x}{1}$
  - Function composition is now simple matrix multiplication!

    For example: $f_2\binom{x}{1} = \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix} f_0\binom{x}{1} = \begin{pmatrix} a_1 a_2 & a_2 b_1 + b_2 \\ 0 & 1 \end{pmatrix}\binom{x}{1}$

- **Most powerful! Homework:**
  - Parallelize tridiagonal solve
  - Parallelize string parsing

Prefix sums as magic bullet for other seemingly sequential algorithms

$\boldsymbol{f}$ **1 ,…,** $\boldsymbol{f\ n\ ff\ f\ n\ nn\ f\ n}$ be an ordered set of functions and $f\ 0\ ff\ f\ 0\ 0\ f\ 0\ \ x\ xx\ x = xx$

Any **time a sequential chain can be modeled as function composition!**

- Let $f\ 1\ 1\ 1, \ldots, f_n$ be an ordered set of functions and $f_0(x) = x$
- Define ordered function compositions: $f_1(x); f_2(f_1(x)); \ldots; f_n(\ldots f_1(x))$
- If we can write function composition $g(x) = f_i(f_{i-1}(x)$ as $g = f_i \circ f_{i-1}$ then we can compute $\circ$ with a prefix sum!

  We saw an example with the adder (M_{ab} were our functions)

- **Example: linear recurrence $\boldsymbol{f_i(x) = a_i f_{i-1}(x) + b_i}$ with $\boldsymbol{f_0(x)}$=x**
  - Write as matrix form $f_i\begin{pmatrix} x \\ 1 \end{pmatrix} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} f_{i-1}\begin{pmatrix} x \\ 1 \end{pmatrix}$
  - Function composition is now simple matrix multiplication!

    For example: $f_2\begin{pmatrix} x \\ 1 \end{pmatrix} = \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix} f_0\begin{pmatrix} x \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 a_2 & a_2 b_1 + b_2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ 1 \end{pmatrix}$

- **Most powerful! Homework:**
  - Parallelize tridiagonal solve
  - Parallelize string parsing

Prefix sums as magic bullet for other seemingly sequential algorithms

f 1 (xx); f 2 ff f 2 2 f 2 (f 1 ff f 1 1 f 1 x xx x); … ; f n ff f n nn f n (… f 1 ff f 1 1 f 1 x xx x)

f 1 ,…, f n ff f n nn f n be an ordered set of functions and f 0 ff f 0 0 f 0 x xx x =xx

Any **time a sequential chain can be modeled as function composition!**

- Define ordered function compositions: f 1 1 1 (x); $f_2(f_1(x))$; … ; $f_n(\dots f_1(x))$
- Define ordered function compositions: $f_1(x)$; $f_2(f_1(x))$; … ; $f_n(\dots f_1(x))$
- If we can write function composition $g(x) = f_i(f_{i-1}(x)$ as $g = f_i \circ f_{i-1}$ then we can compute $\circ$ with a prefix sum!

  We saw an example with the adder (M_{ab} were our functions)

- **Example: linear recurrence $f_i(x) = a_i f_{i-1}(x) + b_i$ with $f_0(x)$=x**
  - Write as matrix form $f_i\binom{x}{1} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} f_{i-1}\binom{x}{1}$
  - Function composition is now simple matrix multiplication!

    For example: $f_2\binom{x}{1} = \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix} f_0\binom{x}{1} = \begin{pmatrix} a_1 a_2 & a_2 b_1 + b_2 \\ 0 & 1 \end{pmatrix}\binom{x}{1}$

- **Most powerful! Homework:**
  - Parallelize tridiagonal solve

Prefix sums as magic bullet for other seemingly sequential algorithms

were our functions)

***f f g x* =*f i ii g x* =*f i* (*f i*−1 *f f f i*−1 *ii*−1 *f i*−1 *x xx x* as *gg*= *f i f f f i ii f i* ∘ *f i*−1 *f f f i*−1 *ii*−1 *f i*−1**
then we can compute ∘ with a prefix sum!

***f* 1 (*xx*); *f* 2 *f f f* 2 2 *f* 2 (*f* 1 *f f f* 1 1 *f* 1 *x xx x*); … ; *f n f f f n nn f n* (… *f* 1 *f f f* 1 1 *f* 1 *x xx x*)**

f* 1 ,…, *f n f f f n nn f n be an ordered set of functions and *f* 0 *f f f* 0 *0 f 0 x xx x* =*xx*

Any **time a sequential chain can be modeled as function composition!**

We saw an example with the adder (M ab   ab b ab were our functions)

- Define ordered function compositions: $f_1(x); f_2(f_1(x)); \ldots ; f_n(\ldots f_1(x))$
- If we can write function composition $g(x) = f_i(f_{i-1}(x)$ as $g = f_i \circ f_{i-1}$ then we can compute ∘ with a prefix sum!

  We saw an example with the adder (M_{ab} were our functions)

- **Example: linear recurrence $\boldsymbol{f_i(x) = a_i f_{i-1}(x) + b_i}$ with $\boldsymbol{f_0(x)}$=x**
  - Write as matrix form $f_i\binom{x}{1} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} f_{i-1}\binom{x}{1}$
  - Function composition is now simple matrix multiplication!

    For example: $f_2\binom{x}{1} = \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix} f_0\binom{x}{1} = \begin{pmatrix} a_1 a_2 & a_2 b_1 + b_2 \\ 0 & 1 \end{pmatrix}\binom{x}{1}$

Prefix sums as magic bullet for other seemingly sequential algorithms

a i aa a i ii a i f i−1 ff f i−1 ii−11 f i−1 x xx x + b i bb b i ii b i with **f 0 ff f 0 00 f 0 x xx x** =x
were our functions)

f f g x =*f i ii g x* =*f i* (*f i*−1 *f f f i*−1 *ii*−1 *f i*−1 *x xx x* as *gg*= *f i f f f i ii f i* ∘ *f i*−1 *f f f i*−1 *ii*−1 *f i*−1
then we can compute ∘ with a prefix sum!

f 1 (xx); f 2 ff f 2 2 f 2 (f 1 ff f 1 1 f 1 x xx x); … ; f n ff f n nn f n (… f 1 ff f 1 1 f 1 x xx x)

f 1 ,…, f n ff f n nn f n be an ordered set of functions and *f* 0 *f f f* 0 *0 f 0 x xx x* =*xx*

Any **time a sequential chain can be modeled as function composition!**

We saw an example with the adder (M ab ab b ab were our functions)

- **Example: linear recurrence f i x = i i i $(x) = a_i f_{i-1}(x) + b_i$ with $f_0(x)$=x**
  - If we can write function composition $g(x) = f_i(f_{i-1}(x)$ as $g = f_i \circ f_{i-1}$ then we can compute ∘ with a prefix sum!
  - *We saw an example with the adder (M_{ab} were our functions)*

- **Example: linear recurrence $f_i(x) = a_i f_{i-1}(x) + b_i$ with $f_0(x)$=x**
  - Write as matrix form $f_i\binom{x}{1} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} f_{i-1}\binom{x}{1}$
  - Function composition is now simple matrix multiplication!

Prefix sums as magic bullet for other seemingly sequential algorithms

x 1 x 1 = a i b i 0 1 a i b i 0 1 a i aa a i ii a i a i b i 0 1 b i bb b i ii b i a i b i 0 1 0 a i b i 0 1 1 a i b i 0 1 a i b i 0 1 f i−1 ff f i−1 ii−1 f i−1 x 1 x 1 xx x 1 1 x 1 x 1

a i aa a i ii a i f i−1 ff f i−1 ii−11 f i−1 x xx x + b i bb b i ii b i with f 0 ff f 0 00 f 0 x xx x =x

were our functions)

ff g x =f i ii g x =f i (f i−1 ff f i−1 ii−1 f i−1 x xx x as gg= f i ff f i ii f i ∘ f i−1 ff f i−1 ii−1 f i−1 then we can compute ∘ with a prefix sum!

f 1 (xx); f 2 ff f 2 2 f 2 (f 1 ff f 1 1 f 1 x xx x); … ; f n ff f n nn f n (… f 1 ff f 1 1 f 1 x xx x)

f 1 ,…, f n ff f n nn f n be an ordered set of functions and f 0 ff f 0 0 f 0 x xx x =xx

Any **time a sequential chain can be modeled as function composition!**

We saw an example with the adder (M ab ab b ab were our functions)

- Write as matrix form f i x 1 1 1 x 1 i i i $\binom{x}{1} = \begin{pmatrix} a_i & b_i \\ 0 & 1 \end{pmatrix} f_{i-1} \binom{x}{1}$
- If we can write function composition $g(x) = f_i(f_{i-1}(x)$ as $g = f_i \circ f_{i-1}$ then we can compute $\circ$ with a prefix sum!

  We saw an example with the adder (M_{ab} were our functions)

- **Example: linear recurrence $f_i(x) = a_i f_{i-1}(x) + b_i$ with $f_0(x)$=x**

Prefix sums as magic bullet for other seemingly sequential algorithms

f 2 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1 = *a* 2 *b* 2 0 1 *a* 2 *b* 2 0 1 *a* 2 *aa a* 2 2 *a* 2 *a* 2 *b* 2 0 1 *b* 2 *bb b* 2 2 *b* 2 *a* 2 *b* 2 0 1 0 *a* 2 *b* 2 0 1 1 *a* 2 *b* 2 0 1 *a* 2 *b* 2 0 1 *a* 1 *b* 1 0 1 *a* 1 *b* 1 0 1 *a* 1 *aa a* 1 1 *a* 1 *a* 1 *b* 1 0 1 *b* 1 *bb b* 1 1 *b* 1 *a* 1 *b* 1 0 1 0 *a* 1 *b* 1 0 1 1 *a* 1 *b* 1 0 1 *a* 1 *b* 1 0 1 *f* 0 *ff f* 0 0 *f* 0 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1 = *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *aa a* 1 1 *a* 1 *a* 2 *aa a* 2 2 *a* 2 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 2 *b* 1 *a* 2 *aa a* 2 2 *a* 2 *bb a* 2 *b* 1 1 *a* 2 *b* 1 + *b* 2 *bb b* 2 2 *b* 2 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 0 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1

x 1 *x* 1 = *a i b i* 0 1 *a i b i* 0 1 *a i aa a i ii a i a i b i* 0 1 *b i bb b i ii b i a i b i* 0 1 0 *a i b i* 0 1 1 *a i b i* 0 1 *a i b i* 0 1 *f i*−1 *ff f i*−1 *ii*−1 *f i*−1 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1

a i aa a i ii a i f i*−1 *ff f i*−1 *ii*−11 *f i*−1 *x xx x* + *b i bb b i ii b i with ***f* 0 *ff f* 0 00 *f* 0 *x xx x*** =x were our functions)

ff g x =*f i ii g x* =*f i* (*f i*−1 *ff f i*−1 *ii*−1 *f i*−1 *x xx x* as *gg*= *f i ff f i ii f i* ∘ *f i*−1 *ff f i*−1 *ii*−1 *f i*−1 then we can compute ∘ with a prefix sum!

f 1 (*xx*); *f* 2 *ff f* 2 2 *f* 2 (*f* 1 *ff f* 1 1 *f* 1 *x xx x*); … ; *f n ff f n nn f n* (… *f* 1 *ff f* 1 1 *f* 1 *x xx x*)

f 1 ,…, *f n ff f n nn f n* be an ordered set of functions and *f* 0 *ff f* 0 *0 f 0 x xx x =xx*

Any **time a sequential chain can be modeled as function composition!**

We saw an example with the adder (M ab ab b ab were our functions)

Prefix sums as magic bullet for other seemingly sequential algorithms

f 2 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1 = *a* 2 *b* 2 0 1 *a* 2 *b* 2 0 1 *a* 2 *aa a* 2 2 *a* 2 *a* 2 *b* 2 0 1 *b* 2 *bb b* 2 2 *b* 2 *a* 2 *b* 2 0 1 0 *a* 2 *b* 2 0 1 1 *a* 2 *b* 2 0 1 *a* 2 *b* 2 0 1 *a* 1 *b* 1 0 1 *a* 1 *b* 1 0 1 *a* 1 *aa a* 1 1 *a* 1 *a* 1 *b* 1 0 1 *b* 1 *bb b* 1 1 *b* 1 *a* 1 *b* 1 0 1 0 *a* 1 *b* 1 0 1 1 *a* 1 *b* 1 0 1 *a* 1 *b* 1 0 1 *f* 0 *ff f* 0 0 *f* 0 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1 = *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *aa a* 1 1 *a* 1 *a* 2 *aa a* 2 2 *a* 2 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 2 *b* 1 *a* 2 *aa a* 2 2 *a* 2 *bb a* 2 *b* 1 1 *a* 2 *b* 1 + *b* 2 *bb b* 2 2 *b* 2 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 0 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1

x 1 *x* 1 = *a i b i* 0 1 *a i b i* 0 1 *a i aa a i ii a i a i b i* 0 1 *b i bb b i ii b i a i b i* 0 1 0 *a i b i* 0 1 1 *a i b i* 0 1 *a i b i* 0 1 *f i*–1 *ff f i*–1 *ii*–1 *f i*–1 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1

a i aa a i ii a i f i*–1 *ff f i*–1 *ii*–11 *f i*–1 *x xx x* + *b i bb b i ii b i with ***f* 0 *ff f* 0 00 *f* 0 *x xx x*** =x **were our functions)**

ff g x =*f i ii g x* =*f i* (*f i*–1 *ff f i*–1 *ii*–1 *f i*–1 *x xx x* as *gg*= *f i ff f i ii f i* ∘ *f i*–1 *ff f i*–1 *ii*–1 *f i*–1 **then we can compute ∘ with a prefix sum!**

f 1 (*xx*); *f* 2 *ff f* 2 2 *f* 2 (*f* 1 *ff f* 1 1 *f* 1 *x xx x*); … ; *f n ff f n nn f n* (… *f* 1 *ff f* 1 1 *f* 1 *x xx x*)

f* 1 ,…, *f n ff f n nn f n be an ordered set of functions and *f* 0 *ff f* 0 *0 f 0 x xx x* =*xx*

Any **time a sequential chain can be modeled as function composition!**

We saw an example with the adder (M ab ab b ab were our functions)

Prefix sums as magic bullet for other seemingly sequential algorithms

f 2 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1 = *a* 2 *b* 2 0 1 *a* 2 *b* 2 0 1 *a* 2 *aa a* 2 2 *a* 2 *a* 2 *b* 2 0 1 *b* 2 *bb b* 2 2 *b* 2 *a* 2 *b* 2 0 1 0 *a* 2 *b* 2 0 1 1 *a* 2 *b* 2 0 1 *a* 2 *b* 2 0 1 *a* 1 *b* 1 0 1 *a* 1 *b* 1 0 1 *a* 1 *aa a* 1 1 *a* 1 *a* 1 *b* 1 0 1 *b* 1 *bb b* 1 1 *b* 1 *a* 1 *b* 1 0 1 0 *a* 1 *b* 1 0 1 1 *a* 1 *b* 1 0 1 *a* 1 *b* 1 0 1 *f* 0 *ff f* 0 0 *f* 0 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1 = *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *aa a* 1 1 *a* 1 *a* 2 *aa a* 2 2 *a* 2 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 2 *b* 1 *a* 2 *aa a* 2 2 *a* 2 *bb a* 2 *b* 1 1 *a* 2 *b* 1 + *b* 2 *bb b* 2 2 *b* 2 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 0 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *a* 1 *a* 2 *a* 2 *b* 1 + *b* 2 0 1 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1

x 1 *x* 1 = *a i b i* 0 1 *a i b i* 0 1 *a i aa a i ii a i a i b i* 0 1 *b i bb b i ii b i a i b i* 0 1 0 *a i b i* 0 1 1 *a i b i* 0 1 *a i b i* 0 1 *f i*–1 *ff f i*–1 *ii*–1 *f i*–1 *x* 1 *x* 1 *xx x* 1 1 *x* 1 *x* 1

a i aa a i ii a i f i*–1 *ff f i*–1 *ii*–11 *f i*–1 *x xx x* + *b i bb b i ii b i with ***f* 0 *ff f* 0 00 *f* 0 *x xx x*** =x **were our functions)**

ff g x =*f i ii g x* =*f i* (*f i*–1 *ff f i*–1 *ii*–1 *f i*–1 *x xx x* as *gg*= *f i ff f i ii f i* ∘ *f i*–1 *ff f i*–1 *ii*–1 *f i*–1 **then we can compute ∘ with a prefix sum!**

***f* 1 (*xx*); *f* 2 *ff f* 2 2 *f* 2 (*f* 1 *ff f* 1 1 *f* 1 *x xx x*); … ; *f n ff f n nn f n* (… *f* 1 *ff f* 1 1 *f* 1 *x xx x*)**

f* 1 ,…, *f n ff f n nn f n be an ordered set of functions and *f* 0 *ff f* 0 *0 f 0 x xx x =xx*

Any **time a sequential chain can be modeled as function composition!**

We saw an example with the adder (M ab ab b ab were our functions)

Another use for prefix sums: Parallel radix sort

- stably sort all values by the i-th bit

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - stably sort all values by the i-th bit

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - stably sort all values by the i-th bit

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

- **Now on n processors**

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

- **Now on n processors**
  - Each processor owns single k-bit number, each iteration

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

- **Now on n processors**
  - Each processor owns single k-bit number, each iteration

    low = prefix_scan(!bit, sum)

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

- **Now on n processors**
  - Each processor owns single k-bit number, each iteration

    low = prefix_scan(!bit, sum)

    high = n-1-backwards_prefix_scan(bit, sum)

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

- **Now on n processors**
  - Each processor owns single k-bit number, each iteration

    low = prefix_scan(!bit, sum)

    high = n-1-backwards_prefix_scan(bit, sum)

    new_idx = (bit == 0) : low ? high

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

  Iteration 0: 101 111 010 011 110 001

  Iteration 1: 010 110 101 111 011 001

  Iteration 2: 101 001 010 110 111 011

  Iteration 3: 001 010 011 101 110 111

- **Now on n processors**
  - Each processor owns single k-bit number, each iteration

  low = prefix_scan(!bit, sum)

  high = n-1-backwards_prefix_scan(bit, sum)

  new_idx = (bit == 0) : low ? high

  b[new_idx] = a[i]

  swap(a,b)

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

- **Now on n processors**
  - Each processor owns single k-bit number, each iteration

    low = prefix_scan(!bit, sum)

    high = n-1-backwards_prefix_scan(bit, sum)

    new_idx = (bit == 0) : low ? high

    b[new_idx] = a[i]

    swap(a,b)

Show one example iteration!

Another use for prefix sums: Parallel radix sort

- **Radix sort works bit-by-bit**
  - Sorts k-bit numbers in k iterations
  - In each iteration i stably sort all values by the i-th bit
  - Example, k=1:

    Iteration 0: 101 111 010 011 110 001

    Iteration 1: 010 110 101 111 011 001

    Iteration 2: 101 001 010 110 111 011

    Iteration 3: 001 010 011 101 110 111

- **Now on n processors**
  - Each processor owns single k-bit number, each iteration

    low = prefix_scan(!bit, sum)

    high = n-1-backwards_prefix_scan(bit, sum)

    new_idx = (bit == 0) : low ? high

    b[new_idx] = a[i]

    swap(a,b)

Show one example iteration!

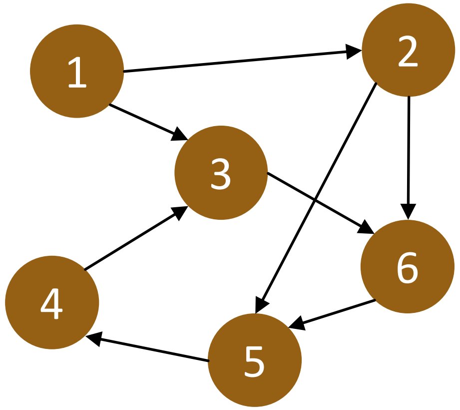
Question: work and depth?

Oblivious graph algorithms

- **Seems paradoxical but isn't (may just not be most efficient)**
  - Use adjacency matrix representation of graph – "compute with all zeros"

Oblivious graph algorithms

- **Seems paradoxical but isn't (may just not be most efficient)**
  - Use adjacency matrix representation of graph – "compute with all zeros"



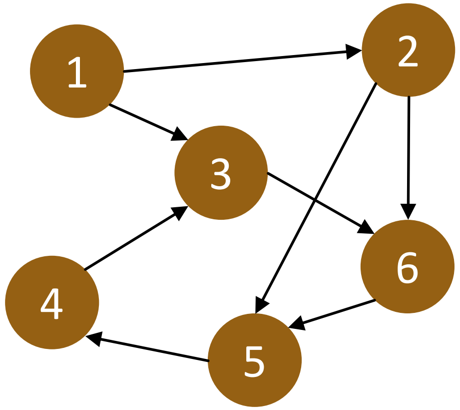
i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

Unweighted graph – binary matrix

Oblivious graph algorithms

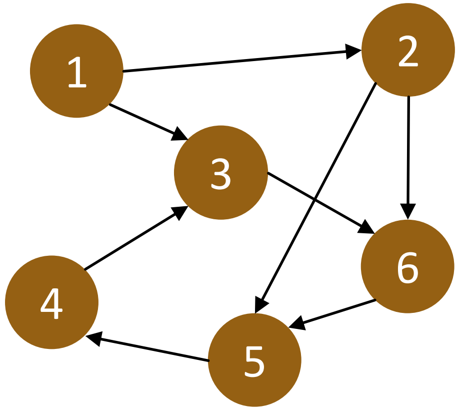
- **Seems paradoxical but isn't (may just not be most efficient)**
  - Use adjacency matrix representation of graph – "compute with all zeros"



0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

(i: columns, j: rows)

Unweighted graph – binary matrix



0	2	3	0	0	0
0	0	0	0	3	1
0	0	0	0	0	2
0	0	4	0	0	0
0	0	0	7	0	0
0	0	0	0	8	0

(i: columns, j: rows)

Weighted graph – general matrix

Algebraic semirings

A semiring is an algebraic structure that

*, +, *, 0, 1)*

Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

Tropical semiring: (ℝ ∪ {∞}, min, +, ∞, 0) (also called min-plus semiring)

Algebraic semirings

A semiring is an algebraic structure that

- Has two binary operations called "addition" and "multiplication"

*, +, *, 0, 1)*

Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

Tropical semiring: (ℝ ∪ {∞}, min, +, ∞, 0) (also called min-plus semiring)

Algebraic semirings

A semiring is an algebraic structure that

- Has two binary operations called “addition” and “multiplication”
- Addition must be associative ((a+b)+c = a+(b+c)) and commutative ((a+b=b+a)) and have an identity element

*, +, *, 0, 1)*

Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

Tropical semiring: (ℝ ∪ {∞}, min, +, ∞, 0) (also called min-plus semiring)

Algebraic semirings

A semiring is an algebraic structure that

- Has two binary operations called “addition” and “multiplication”
- Addition must be associative ((a+b)+c = a+(b+c)) and commutative ((a+b=b+a)) and have an identity element
- Multiplication must be associative and have an identity element

*, +, *, 0, 1)*

Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

Tropical semiring: (ℝ ∪ {∞}, min, +, ∞, 0) (also called min-plus semiring)

Algebraic semirings

A semiring is an algebraic structure that

- Has two binary operations called “addition” and “multiplication”
- Addition must be associative ((a+b)+c = a+(b+c)) and commutative ((a+b=b+a)) and have an identity element
- Multiplication must be associative and have an identity element
- Multiplication distributes over addition (a*(b+c) = a*b+a*c) and multiplication by additive identity annihilates

*, +, *, 0, 1)*

Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

Tropical semiring: ($\mathbb{R} \cup \{\infty\}$, min, +, ∞, 0) (also called min-plus semiring)

Algebraic semirings

A semiring is an algebraic structure that

- Has two binary operations called “addition” and “multiplication”
- Addition must be associative ((a+b)+c = a+(b+c)) and commutative ((a+b=b+a)) and have an identity element
- Multiplication must be associative and have an identity element
- Multiplication distributes over addition (a*(b+c) = a*b+a*c) and multiplication by additive identity annihilates
- Semirings are denoted by tuples (S, +, *, 0, 1)

*, +, *, 0, 1)*

Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

Tropical semiring: $(\mathbb{R} \cup \{\infty\}$*, min, +,* ∞*, 0) (also called min-plus semiring)*

Algebraic semirings

, +, *, *0, 1)*

A **semiring is an algebraic structure that**

- Has two binary operations called “addition” and “multiplication”
- Addition must be associative ((a+b)+c = a+(b+c)) and commutative ((a+b=b+a)) and have an identity element
- Multiplication must be associative and have an identity element
- Multiplication distributes over addition (a*(b+c) = a*b+a*c) and multiplication by additive identity annihilates
- Semirings are denoted by tuples (S, +, *, 0, 1)

*“Standard” ring of rational numbers: (ℝ, +, *, 0, 1)*

Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

Tropical semiring: (ℝ ∪ {∞}, min, +, ∞, 0) (also called min-plus semiring)

Algebraic semirings

, ∧, 0, 1)

, **+, *,** *0, 1)*

A **semiring is an algebraic structure that**

- Has two binary operations called “addition” and “multiplication”
- Addition must be associative ((a+b)+c = a+(b+c)) and commutative ((a+b=b+a)) and have an identity element
- Multiplication must be associative and have an identity element
- Multiplication distributes over addition (a*(b+c) = a*b+a*c) and multiplication by additive identity annihilates
- Semirings are denoted by tuples (S, +, *, 0, 1)

  Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

  Boolean semiring: ({0,1}, ∨, ∧, 0, 1)

  Tropical semiring: $(\mathbb{R} \cup \{\infty\}$*, min, +, ∞, 0) (also called min-plus semiring)*

Algebraic semirings

∪{∞}, min, +, ∞, 0) (also called min-plus semiring)

, ∧, 0, 1)

, **+, *,** *0, 1)*

A **semiring is an algebraic structure that**

- Has two binary operations called “addition” and “multiplication”
- Addition must be associative ((a+b)+c = a+(b+c)) and commutative ((a+b=b+a)) and have an identity element
- Multiplication must be associative and have an identity element
- Multiplication distributes over addition (a*(b+c) = a*b+a*c) and multiplication by additive identity annihilates
- Semirings are denoted by tuples (S, +, *, 0, 1)

  Tropical semiring: ($\mathbb{R}$ ∪ {∞}, *min, +,* ∞, *0) (also called min-plus semiring)*

  Boolean semiring: ({0,1}, ∨, ∧, *0, 1)*

  Tropical semiring: ($\mathbb{R}$ ∪ {∞}, *min, +,* ∞, *0) (also called min-plus semiring)*

Oblivious shortest path search

0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

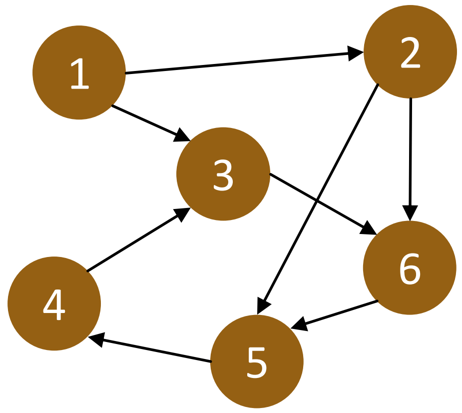
Oblivious shortest path search

-
- **Construct distance matrix from adjacency matrix by replacing all off-diagonal zeros with ∞**

0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

Oblivious shortest path search

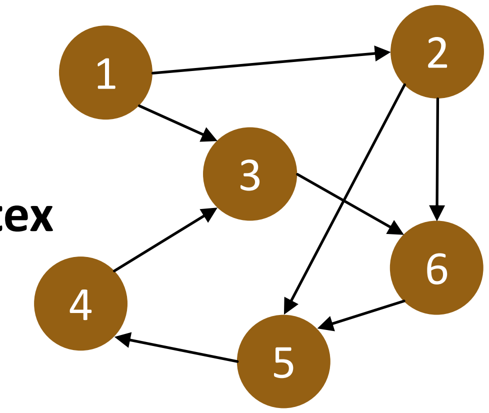
-
- **Construct distance matrix from adjacency matrix by replacing all off-diagonal zeros with ∞**



0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

Oblivious shortest path search

- f size n to ∞ everywhere but zero at start vertex
-
- Initialize distance vector $d\ 0\ o\ 0\ 0\ 0$ of size n to ∞ everywhere but zero at start vertex

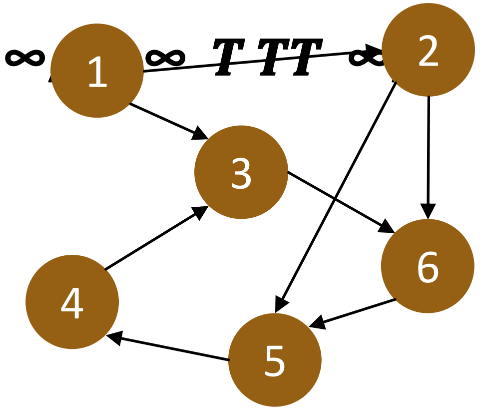


0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

Oblivious shortest path search

- $\infty, 0, \infty, \infty, \infty, \infty \;\; T \;\; \infty, 0, \infty, \infty, \infty, \infty \; \infty, 00, \infty, \infty, \infty, \infty \; \infty, 0, \infty, \infty, \infty, \infty \;\; \infty, 0, \infty, \infty, \infty, \infty \;\; T \, TT \;\; \infty$ $0, \infty, \infty, \infty, \infty \;\; T$
- f size n to ∞ everywhere but zero at start vertex
-

Show evolution when multiplied!



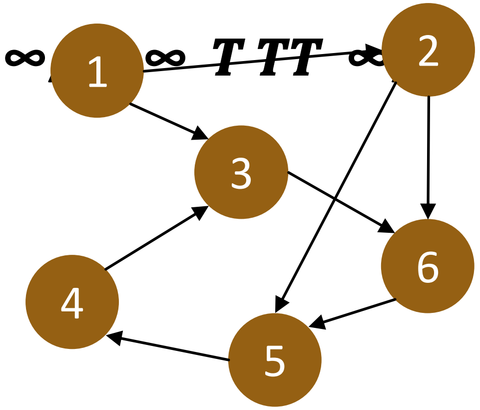
0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

Oblivious shortest path search

- $\infty, 0, \infty, \infty, \infty, \infty\ T\ \infty, 0, \infty, \infty, \infty, \infty\ \infty, 00, \infty, \infty, \infty, \infty\ \infty, 0, \infty, \infty, \infty, \infty\ \ \infty, 0, \infty, \infty, \infty\ T\ TT\ \infty$
  $0, \infty, \infty, \infty, \infty\ T$
- f size n to ∞ everywhere but zero at start vertex
-

  Show evolution when multiplied!

- **SSSP can be performed with n+1 matrix-vector multiplications!**

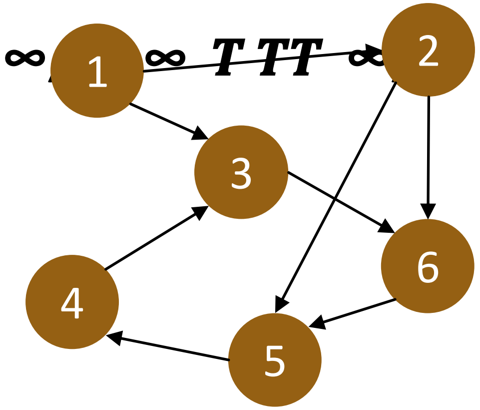


0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

Oblivious shortest path search

- ∞, 0,∞, ∞, ∞, ∞ T ∞, 0,∞, ∞, ∞, ∞ ∞, 00,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ T TT ∞ 0,∞, ∞, ∞, ∞ T
- f size n to ∞ everywhere but zero at start vertex
-

  Show evolution when multiplied!



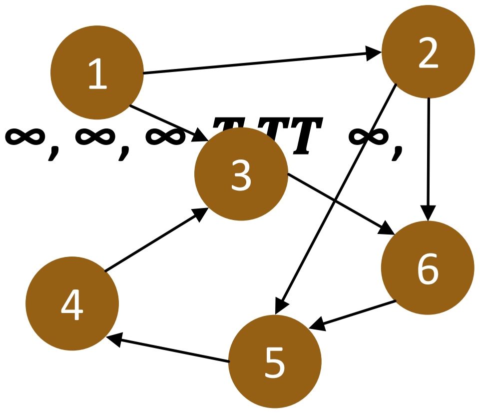
- **SSSP can be performed with n+1 matrix-vector multiplications!**
  - Question: total work and depth?

0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

Oblivious shortest path search

- $OO(n 3 nn n 3 3 n 3), DD=OO(nn \log nn)$
- ∞, 0,∞, ∞, ∞, ∞ T ∞, 0,∞, ∞, ∞, ∞ ∞, 00,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ TTT ∞, 0,∞, ∞, ∞, ∞ T
- **f size n to ∞ everywhere but zero at start vertex**
-

  Show evolution when multiplied!



- **SSSP can be performed with n+1 matrix-vector multiplications!**
  - Question: total work and depth?

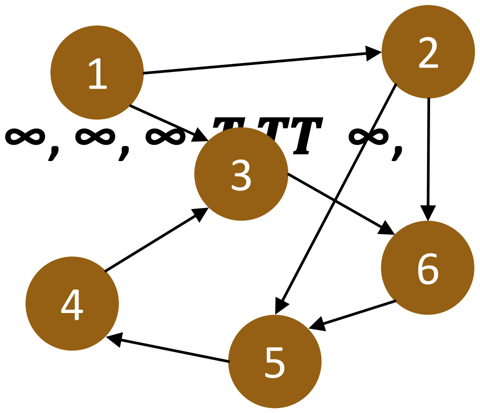
    $W = O(n^3), D = O(n \log n)$

0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

Oblivious shortest path search

- *OO*(*n* 3 *nn n* 3 3 *n* 3), *DD*=*OO*(*nn* log *nn*)
- **∞, 0,∞, ∞, ∞, ∞ *T* ∞, 0,∞, ∞, ∞, ∞ ∞, 00,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ *TTT* ∞, 0,∞, ∞, ∞, ∞ *T***
- **f size n to ∞ everywhere but zero at start vertex**
-

  Show evolution when multiplied!



- **SSSP can be performed with n+1 matrix-vector multiplications!**
  - Question: total work and depth?
  - Question: Is this good? Optimal?

0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

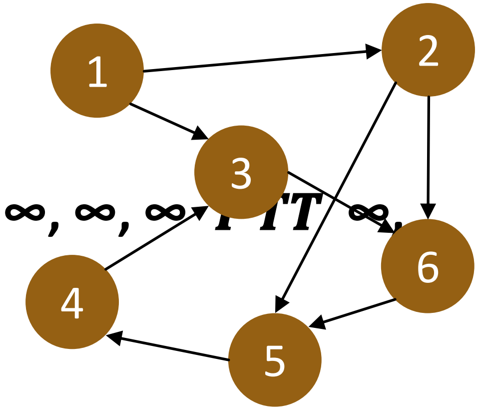
Oblivious shortest path search

- E EE $E + V$ VV $V \log V$ VV V)
- OO(n 3 nn n 3 3 n 3), DD=OO(nn log nn)
- ∞, 0,∞, ∞, ∞, ∞ T ∞, 0,∞, ∞, ∞, ∞ ∞, 00,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ TT ∞, 0,∞, ∞, ∞, ∞ T
- f size n to ∞ everywhere but zero at start vertex
-

  Show evolution when multiplied!

- **SSSP can be performed with n+1 matrix-vector multiplications!**
  - Question: total work and depth?
  - Question: Is this good? Optimal?

  Dijkstra = $O(|E| + |V|\log|V|)$ ☹

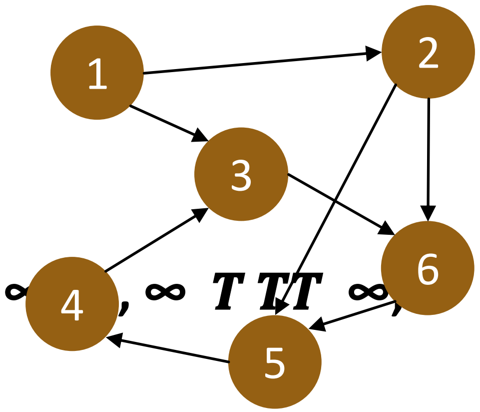


0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

Oblivious shortest path search

- OO(n 3 nn n 3 3 n 3 log nn), DD=OO(log 2 log log 2 2 log 2 nn)
- E EE E + V VV V log V VV V)
- OO(n 3 nn n 3 3 n 3), DD=OO(nn log nn)
- ∞, 0,∞, ∞, ∞, ∞ T ∞, 0,∞, ∞, ∞, ∞ ∞, 00,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ ∞, 0,∞, ∞, ∞, ∞ T T T ∞, 0,∞, ∞, ∞, ∞ T
- f size n to ∞ everywhere but zero at start vertex
-

Show evolution when multiplied!



0	2	3	∞	∞	∞
∞	0	∞	∞	3	1
∞	∞	0	∞	∞	2
∞	∞	4	0	∞	∞
∞	∞	∞	7	0	∞
∞	∞	∞	∞	8	0

- **SSSP can be performed with n+1 matrix-vector multiplications!**
  - Question: total work and depth?
  - Question: Is this good? Optimal?

    Dijkstra = $O(|E| + |V|\log|V|)$ ☹
- **Homework:**
  - Define a similar APSP algorithm with

    $W = O(n^3 \log n)$, $D = O(\log^2 n)$

Oblivious connected components

Question: How could we compute the transitive closure of a graph?

- $A + I)$ n times with itself in the Boolean semiring!
- Why?

  Demonstrate that $(A + I)^2$ *has 1s for each path of at most length 1*

  By induction show that $(A + I)^k$ *has 1s for each path of at most length k*

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff ${(A_I)^n}_{ij} = 1$ and $L_{ij} = \infty$ otherwise
  - For each row (vertex) perform min-reduction to determine its component label!
  - Overall work and depth?

    $W = O(n^3 \log n)$, $D = O(\log^2 n)$

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

Oblivious connected components

Question: How could we compute the transitive closure of a graph?

- $A + I)$ n times with itself in the Boolean semiring!
- Why?

  Demonstrate that $(A + I)^2$ *has 1s for each path of at most length 1*

  By induction show that $(A + I)^k$ *has 1s for each path of at most length k*

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise
  - For each row (vertex) perform min-reduction to determine its component label!
  - Overall work and depth?

    $W = O(n^3 \log n)$, $D = O(\log^2 n)$

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

AA+II) nn times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Multiply the matrix $(A + I)$ n times with itself in the Boolean semiring!
- Why?

  Demonstrate that $(A + I)^2$ has 1s for each path of at most length 1

  By induction show that $(A + I)^k$ has 1s for each path of at most length k

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise
  - For each row (vertex) perform min-reduction to determine its component label!
  - Overall work and depth?

    $W = O(n^3 \log n)$, $D = O(\log^2 n)$

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

AA+*II*) ***nn*** times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  Demonstrate that A + I A + I 2 has 1s for each path of at most length 1

  Demonstrate that $(A+I)^2$ *has 1s for each path of at most length 1*

  By induction show that $(A+I)^k$ *has 1s for each path of at most length k*

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise
  - For each row (vertex) perform min-reduction to determine its component label!
  - Overall work and depth?

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

*AA***+II) nn** times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  By induction show that A + I A + I k has 1s for each path of at most length k

  Demonstrate that $(A + I)^2$ *has 1s for each path of at most length 1*

  By induction show that $(A + I)^k$ *has 1s for each path of at most length k*

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise
  - For each row (vertex) perform min-reduction to determine its component label!

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

AA+*II*) *nn* times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  By induction show that A + I A + I k has 1s for each path of at most length k

- **What is work and depth of transitive closure?**

  By induction show that $(A+I)^k$ *has 1s for each path of at most length k*

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise
  - For each row (vertex) perform min-reduction to determine its component label!

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

OO(n 3 nn n 3 3 n 3 log nn) DD=OO(log 2 log log 2 2 log 2 nn)

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

*AA***+*II*) *nn*** times with itself in the Boolean semiring!

Question**: How could we compute the transitive closure of a graph?**

- Why?

  By induction show that A + I A + I k has 1s for each path of at most length k

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise

i (→), j (↓)

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

OO(n 3 nn n 3 3 n 3 log nn) DD=OO(log 2 log log 2 2 log 2 nn)

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

AA+*II*) *nn* times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  By induction show that A + I A + I k has 1s for each path of at most length k

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise

i → ; j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

OO(n 3 nn n 3 3 n 3 log nn) DD=OO(log 2 log log 2 2 log 2 nn)

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

AA+*II*) ***nn*** times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  By induction show that A + I A + I k has 1s for each path of at most length k

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix $L_{ij} = j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

jj iff A I n ij A I n A I A I AA A I II A I A I A I n nn A I n A I n ij iijj A I n
iijj L ij =∞ otherwise

OO(n 3 nn n 3 3 n 3 log nn) DD=OO(log 2 log log 2 2 log 2 nn)

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

AA+*II*) *nn* times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  By induction show that $A + I$ $A + I$ k *has 1s for each path of at most length k*

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - Create label matrix L ij $=$ ij j ij $= j$ iff $(A_I)^n{}_{ij} = 1$ and $L_{ij} = \infty$ otherwise

- **How to get to connected components from a transitive closure matrix?**

i → ; j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

jj iff A I n ij A I n A I A I AA A I II A I A I A I n nn A I n A I n ij iijj A I n
iijj L ij =∞ otherwise

OO(n 3 nn n 3 3 n 3 log nn) DD=OO(log 2 log log 2 2 log 2 nn)

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

AA+***II*) *nn*** times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  By induction show that A + I A + I k has 1s for each path of at most length k

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - For each row (vertex) perform min-reduction to determine its component label!

- **How to get to connected components from a transitive closure matrix?**

i (→), j (↓)

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

jj iff A I n ij A I n A I A I AA A I II A I A I A I n nn A I n A I n ij iijj A I n
iijj L ij =∞ otherwise

OO(n 3 nn n 3 3 n 3 log nn) DD=OO(log 2 log log 2 2 log 2 nn)

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

AA+*II*) *nn* times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  By induction show that A + I A + I k has 1s for each path of at most length k

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - For each row (vertex) perform min-reduction to determine its component label!
  - Overall work and depth?
- **How to get to connected components from a transitive closure matrix?**

i (→), j (↓)

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

OO(n 3 nn n 3 3 n 3 log nn), DD=OO(log 2 log log 2 2 log 2 nn)

jj iff   A I   n ij   A I   n   A I   A I AA A I II A I   A I   A I   n nn   A I   n   A I   n ij iijj   A I   n
iijj L ij =∞ otherwise

OO(n 3 nn n 3 3 n 3 log nn) DD=OO(log 2 log log 2 2 log 2 nn)

II A+I   A+I k kk A+I k has 1s for each path of at most length k

II A+I   A+I 2 2 A+I 2 has 1s for each path of at most length 1

*AA***+*II*) *nn*** times with itself in the Boolean semiring!

Question**: How could we compute the transitive closure of a graph?**

- Why?

  By induction show that   A + I A + I   k has 1s for each path of at most length k

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - For each row (vertex) perform min-reduction to determine its component label!
  - Overall work and depth?

    $W = O(n^3 \log n)$, $D = O(\log^2 n)$

(i → columns, j ↓ rows)

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Oblivious connected components

OO(n 3 nn n 3 3 n 3 log nn), DD=OO(log 2 log log 2 2 log 2 nn)

OO(n 3 nn n 3 3 n 3 log nn), DD=OO(log 2 log log 2 2 log 2 nn)

jj iff A I n ij A I n A I A I AA A I II A I A I A I n nn A I n A I n ij iijj A I n
iijj L ij =∞ otherwise

OO(n 3 nn n 3 3 n 3 log nn) DD=OO(log 2 log log 2 2 log 2 nn)

II A+I A+I k kk A+I k has 1s for each path of at most length k

II A+I A+I 2 2 A+I 2 has 1s for each path of at most length 1

AA+II) nn times with itself in the Boolean semiring!

Question: **How could we compute the transitive closure of a graph?**

- Why?

  By induction show that A + I A + I k has 1s for each path of at most length k

- **What is work and depth of transitive closure?**
  - Repeated squaring! $W = O(n^3 \log n)$ $D = O(\log^2 n)$
- **How to get to connected components from a transitive closure matrix?**
  - Each component needs unique label
  - For each row (vertex) perform min-reduction to determine its component label!

    $W = O(n^3 \log n)$, $D = O(\log^2 n)$

i →, j ↓

0	1	1	0	0	0
0	0	0	0	1	1
0	0	0	0	0	1
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

+I ↓

1	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	1
0	0	1	1	0	0
0	0	0	1	1	0
0	0	0	0	1	1

Many if not all graph problems have oblivious or tensor variants!

- $< |V|^2/\log|V|)$

Many if not all graph problems have oblivious or tensor variants!

- **Not clear whether they are most efficient**
  - Efforts such as GraphBLAS exploit existing BLAS implementations and techniques
  - $< |V|^2/\log|V|)$

Many if not all graph problems have oblivious or tensor variants!

- **Not clear whether they are most efficient**
  - Efforts such as GraphBLAS exploit existing BLAS implementations and techniques

- **Generalizations to other algorithms possible**
  - Can everything be modeled as tensor computations on the right ring?
  - E. Solomonik, TH: *"Sparse Tensor Algebra as a Parallel Programming Model"*
  - Much of machine learning/deep learning is oblivious

- $< |V|^2/\log|V|)$

Many if not all graph problems have oblivious or tensor variants!

- V **2** $V\,VV\,V$ V **2** 2 V 2 $/\log|VV|)$
- *Not* **clear whether they are most efficient**
  - Efforts such as GraphBLAS exploit existing BLAS implementations and techniques

- **Generalizations to other algorithms possible**
  - Can everything be modeled as tensor computations on the right ring?
  - E. Solomonik, TH: *"Sparse Tensor Algebra as a Parallel Programming Model"*
  - Much of machine learning/deep learning is oblivious

- **Many algorithms get non-oblivious though**
  - All sparse algorithms are data-dependent!

    May recover some of the lost efficiency by computing zeros!

Many if not all graph problems have oblivious or tensor variants!

- $\boldsymbol{V}$ **2** $\boldsymbol{V\,VV\,V}$ $\boldsymbol{V}$ **2** $2\;V\;2\;/\log|VV|)$
- *Not* **clear whether they are most efficient**
  - Efforts such as GraphBLAS exploit existing BLAS implementations and techniques

- **Generalizations to other algorithms possible**
  - Can everything be modeled as tensor computations on the right ring?
  - E. Solomonik, TH: *"Sparse Tensor Algebra as a Parallel Programming Model"*
  - Much of machine learning/deep learning is oblivious

- **Many algorithms get non-oblivious though**
  - All sparse algorithms are data-dependent!

    May recover some of the lost efficiency by computing zeros!

- **Now moving to non-oblivious ☺**