M. Pueschel, T. Ben-Nun

Lecture 8: Scalable lock study and oblivious algorithms

Administration

Project Schedule:

- **November 15:** Send short report on status of project (max. 1 page plus one optional pages with plots only). It should include a brief summary what was done, how the work was divided, and initial results.
- **November 29 – December 4:** Half hour one-on-one meeting with project supervisor. Bring short presentation for at most 10 minutes, make sure to include plots with current results.
- **End of semester:** Project presentations during lecture/recitation hours.

Review of last lecture(s)

- **Lock implementation(s)**
  - Advanced locks (CLH + MCS)

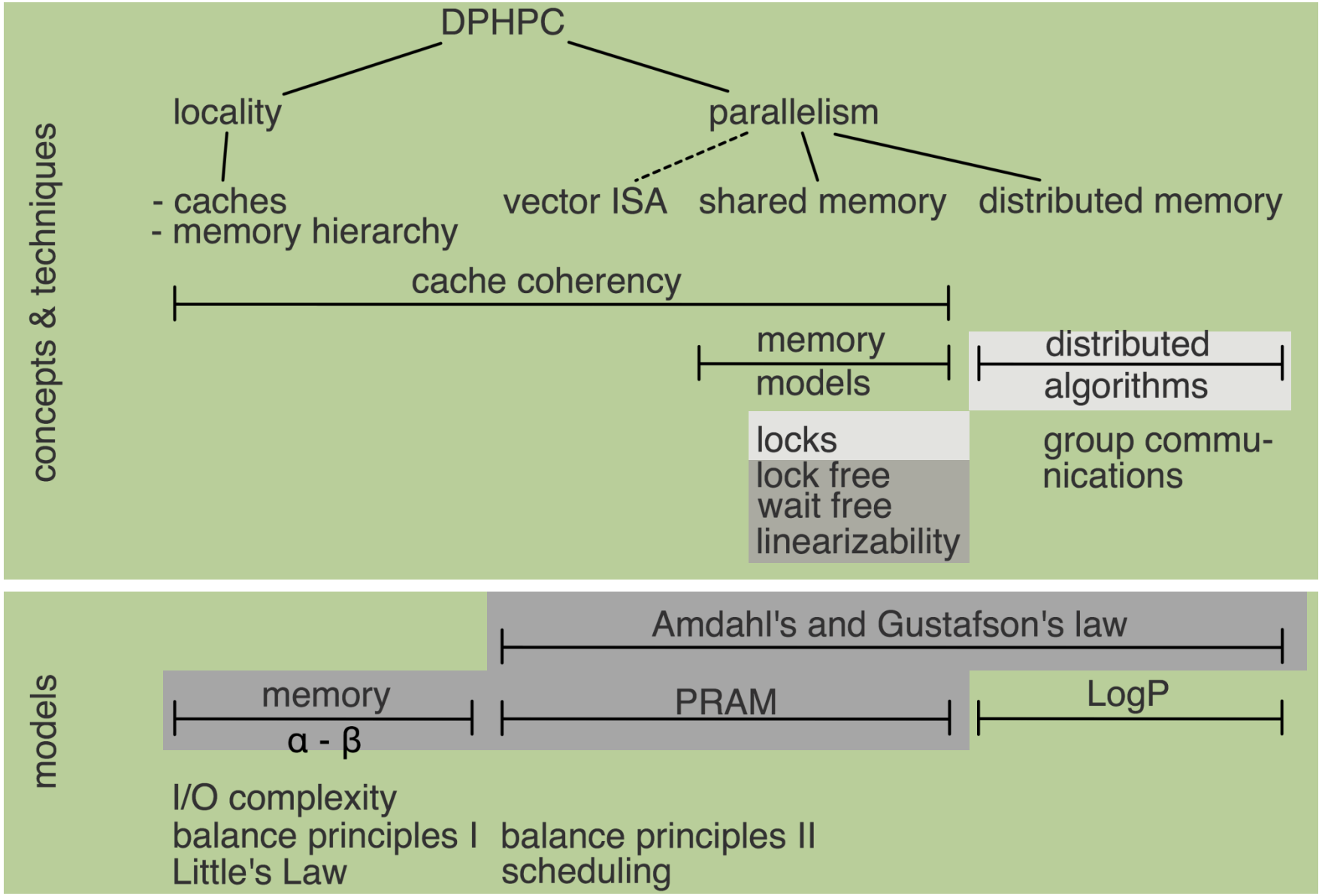
- **Started impossibility of wait-free consensus with atomic registers**
  - “perhaps one of the most striking impossibility results in Computer Science” (Herlihy, Shavit)

- **Theoretical background for performance**
  - Amdahl’s law
  - Models: PRAM, Work/Depth, simple alpha-beta (Hockney) model
  - Simple algorithms: reduce, mergesort, scan
  - Brent’s scheduling lemma + Little’s law
  - Roofline model

Learning goals for today

- **Case study about scalable locking**
  - Same concepts, realistic setting
- **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
- **Case study about data-centric parallel programming**

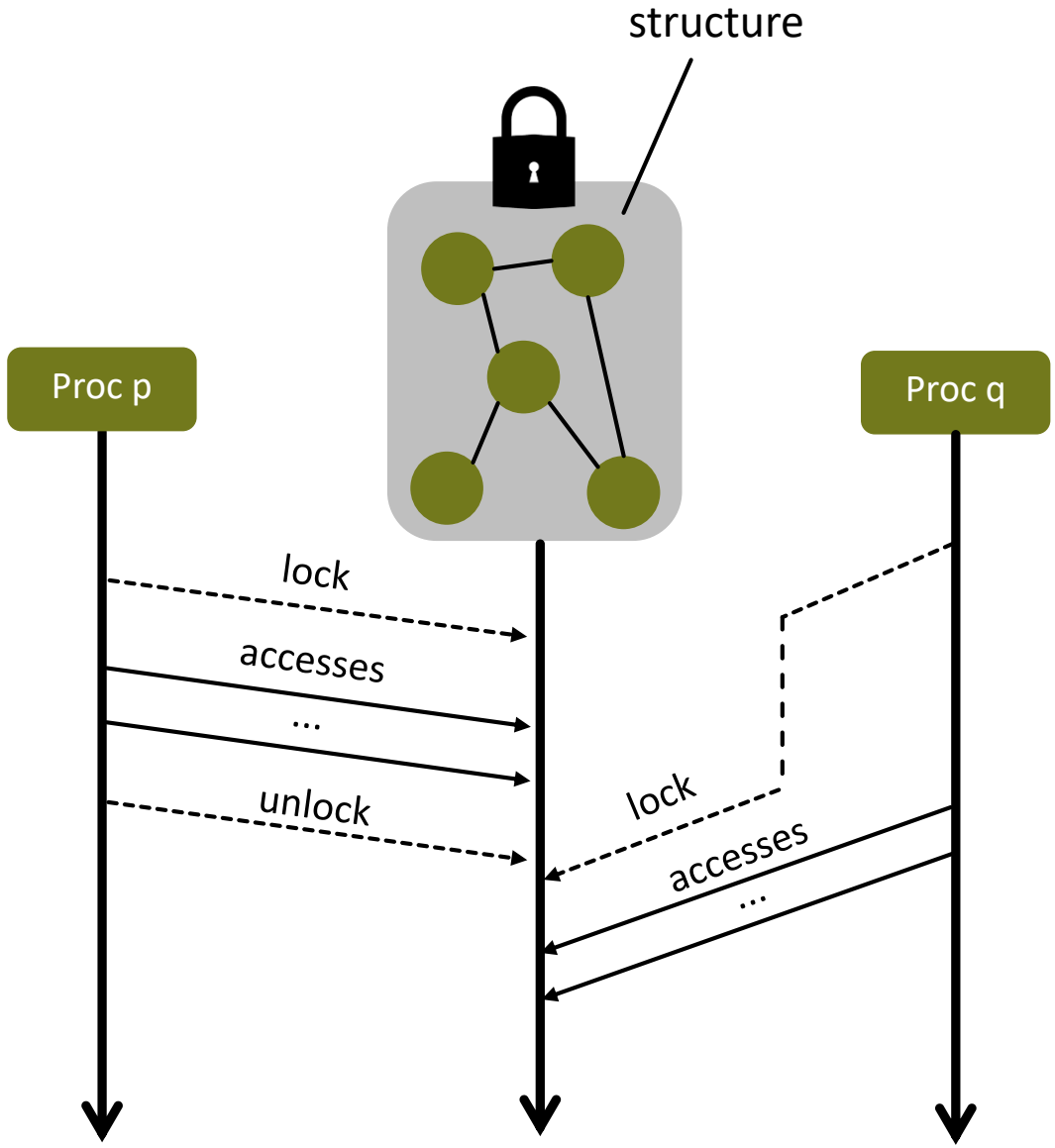
DPHPC Overview



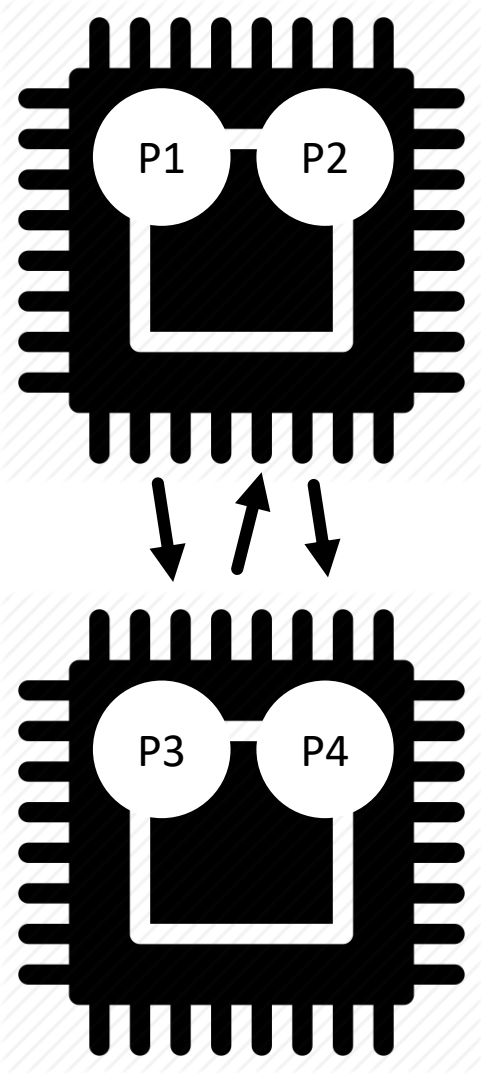
Case study: Fast Large-scale Locking in Practice

Inuitive semantics

Various performance penalties



Locks: Challenges



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

Locks: Challenges

We need intra- and inter-node topology-awareness

We need to cover arbitrary topologies

Locks: Challenges

We need to distinguish between readers and writers

Reader

Reader

Reader

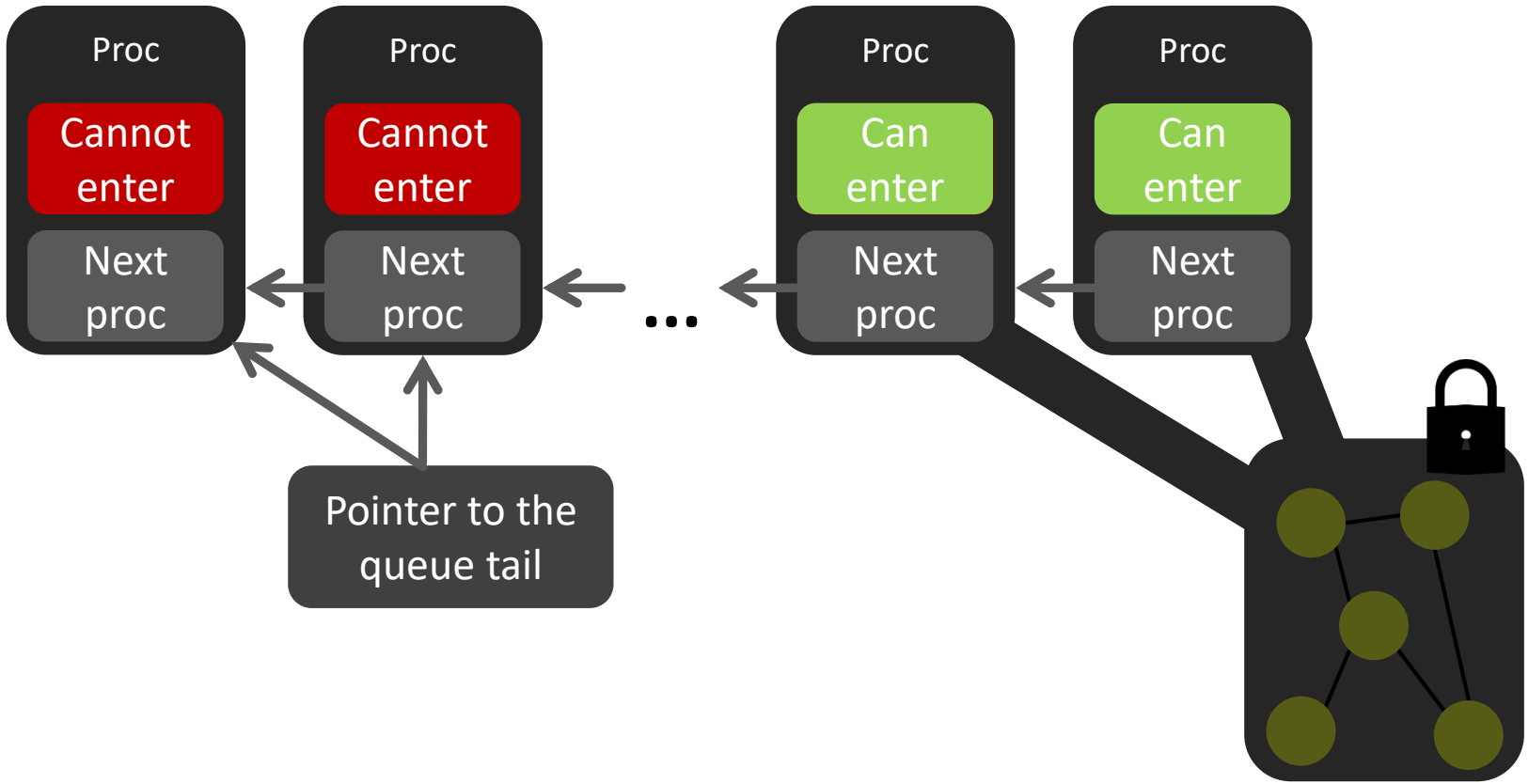
Writer

We need flexible performance for both types of processes

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

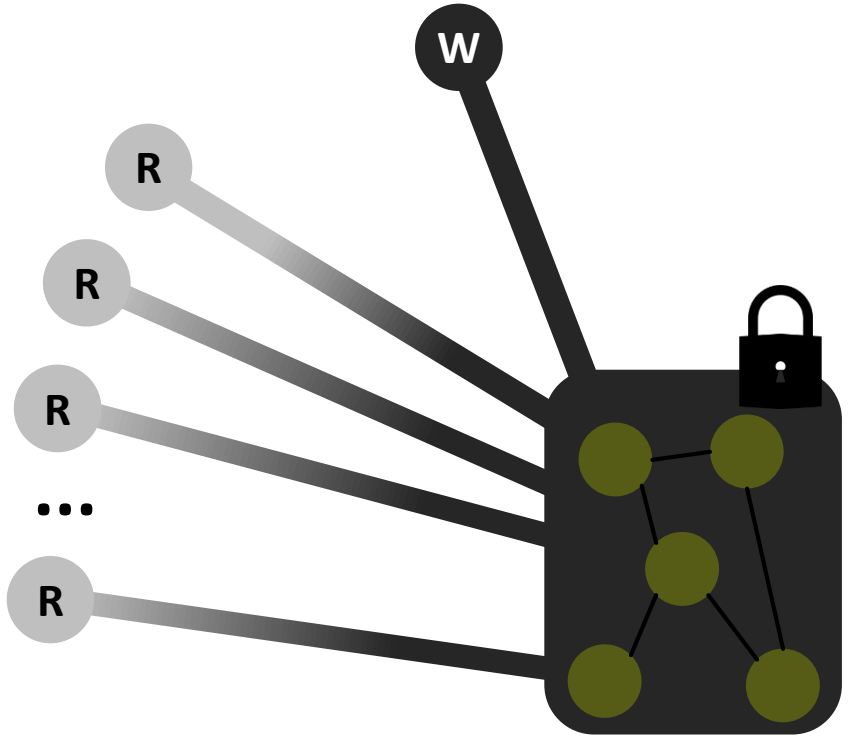
What will we use in the design?

Ingredient 1 - MCS Locks



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

Ingredient 2 - Reader-Writer Locks

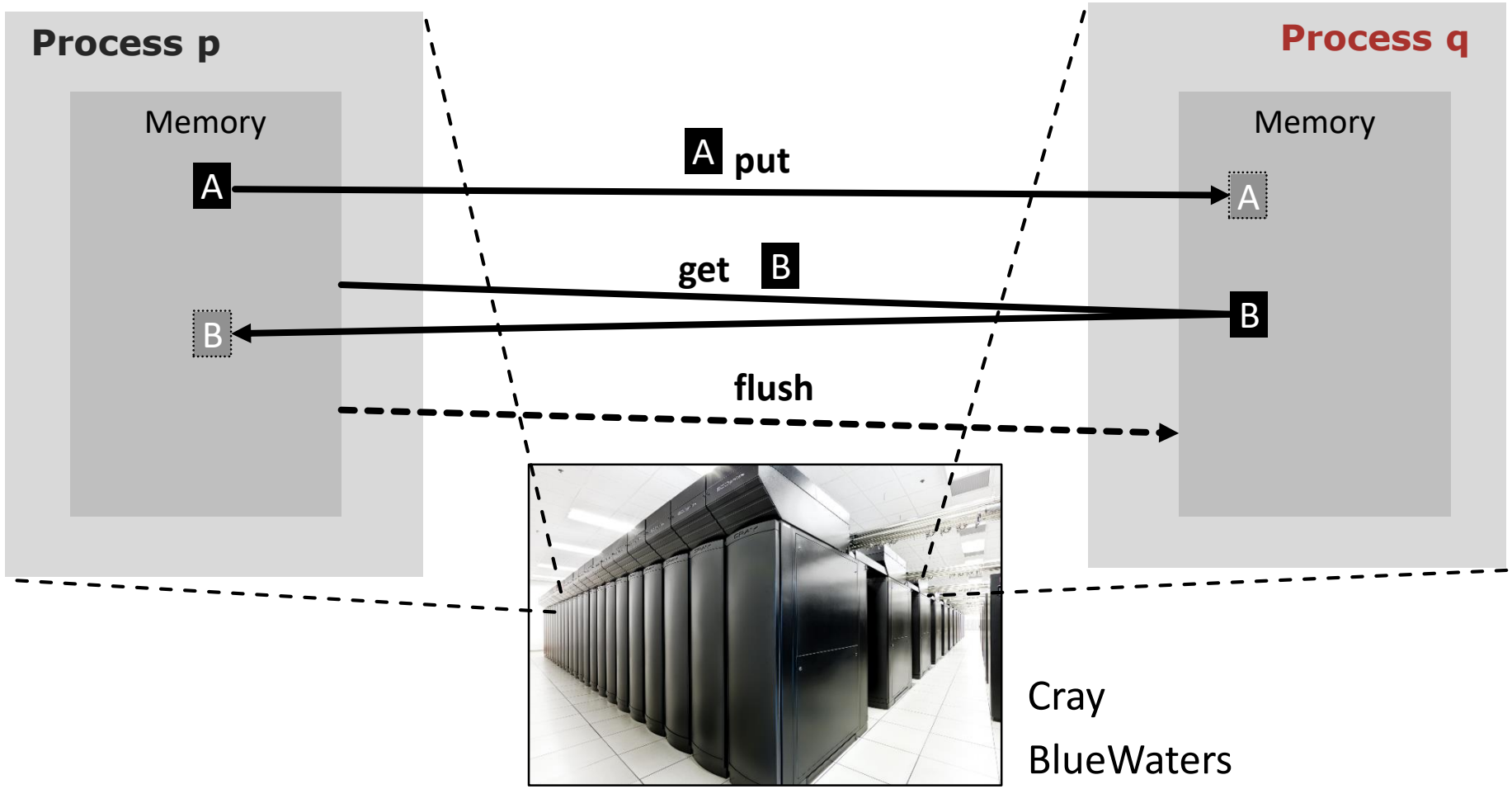


How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?

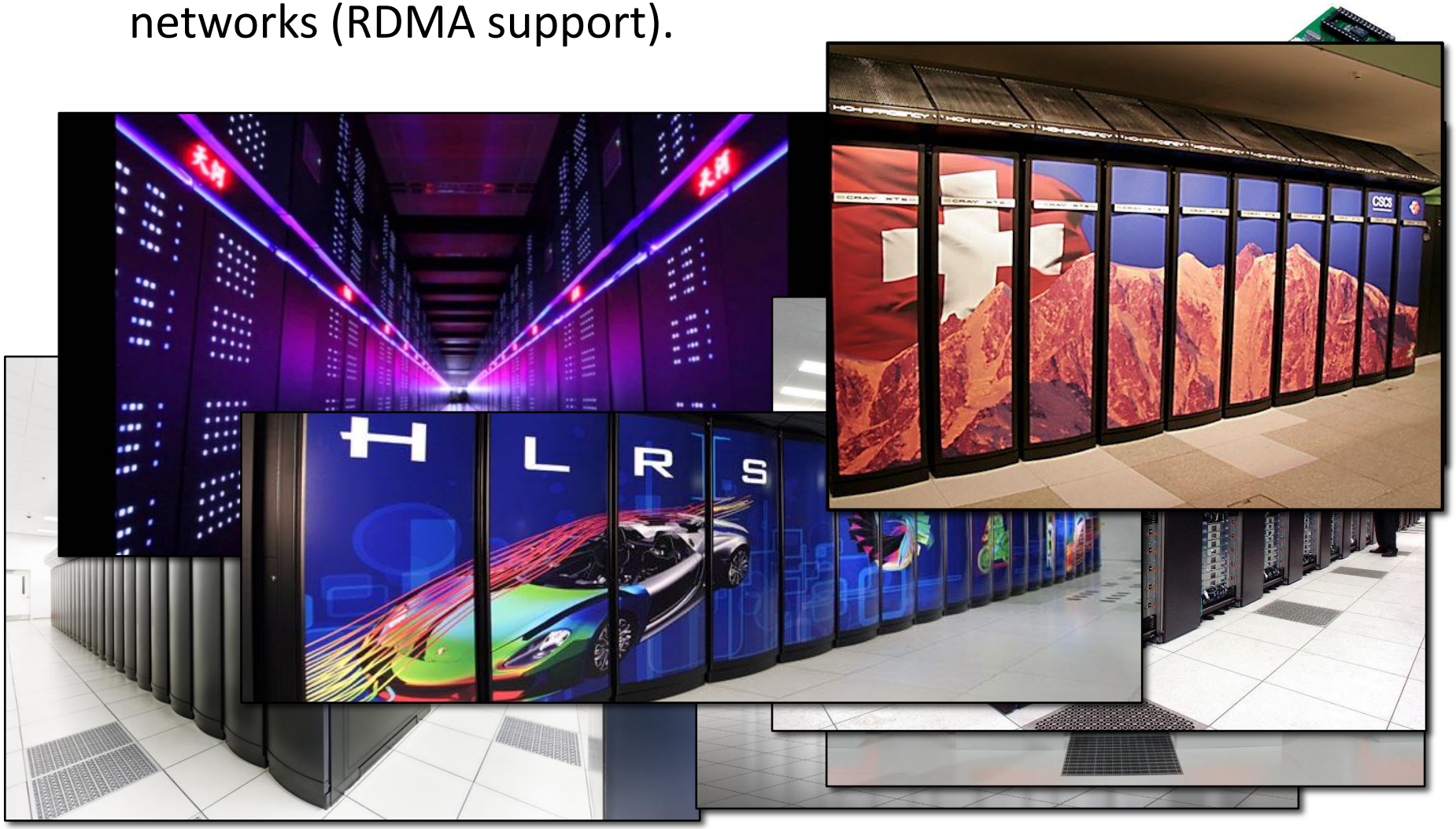
Remote Memory Access (RMA) Programming



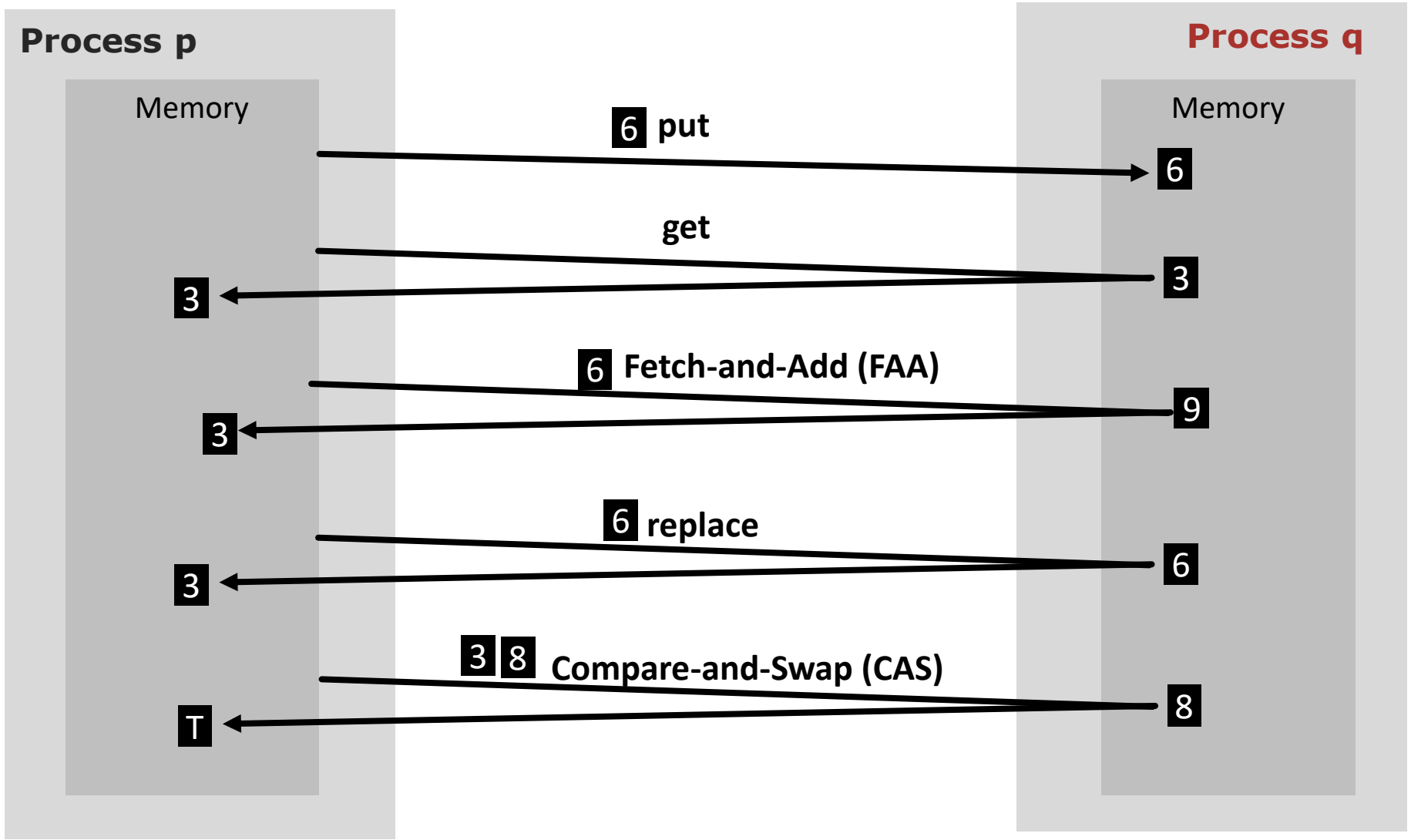
TH, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood: Remote Memory Access Programming in MPI-3, ACM TOPC'15

Remote Memory Access Programming

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



RMA-RW - Required Operations



Recitation recap: MPI RMA

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

  Accumulate (also atomic Put)

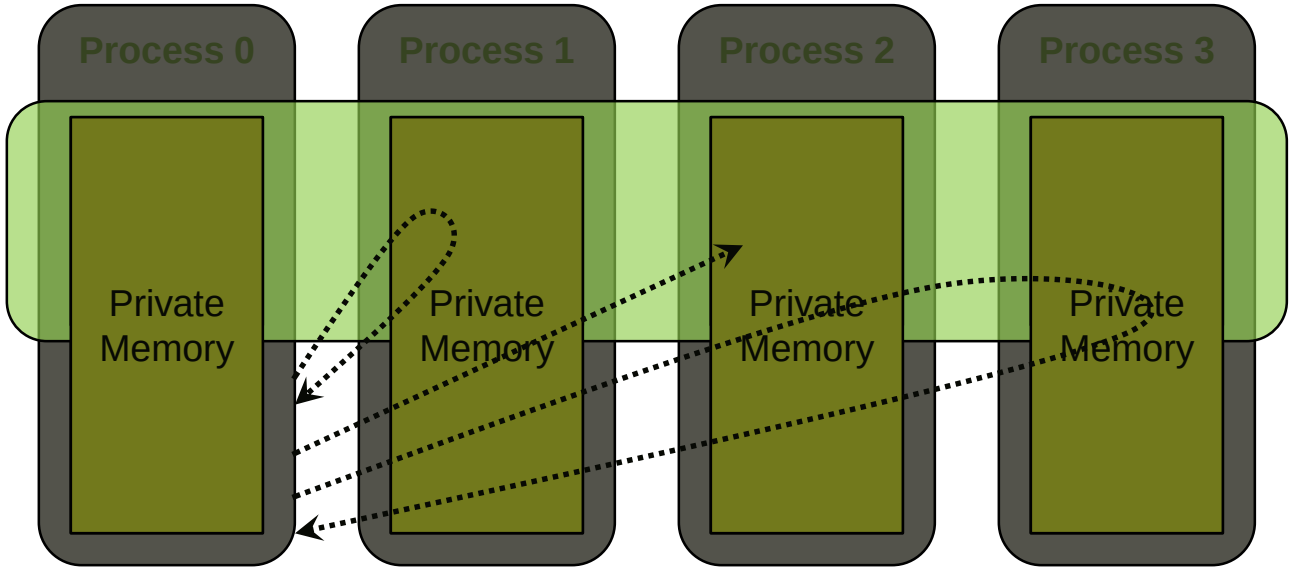
  Get_accumulate (also atomic Get)

  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?

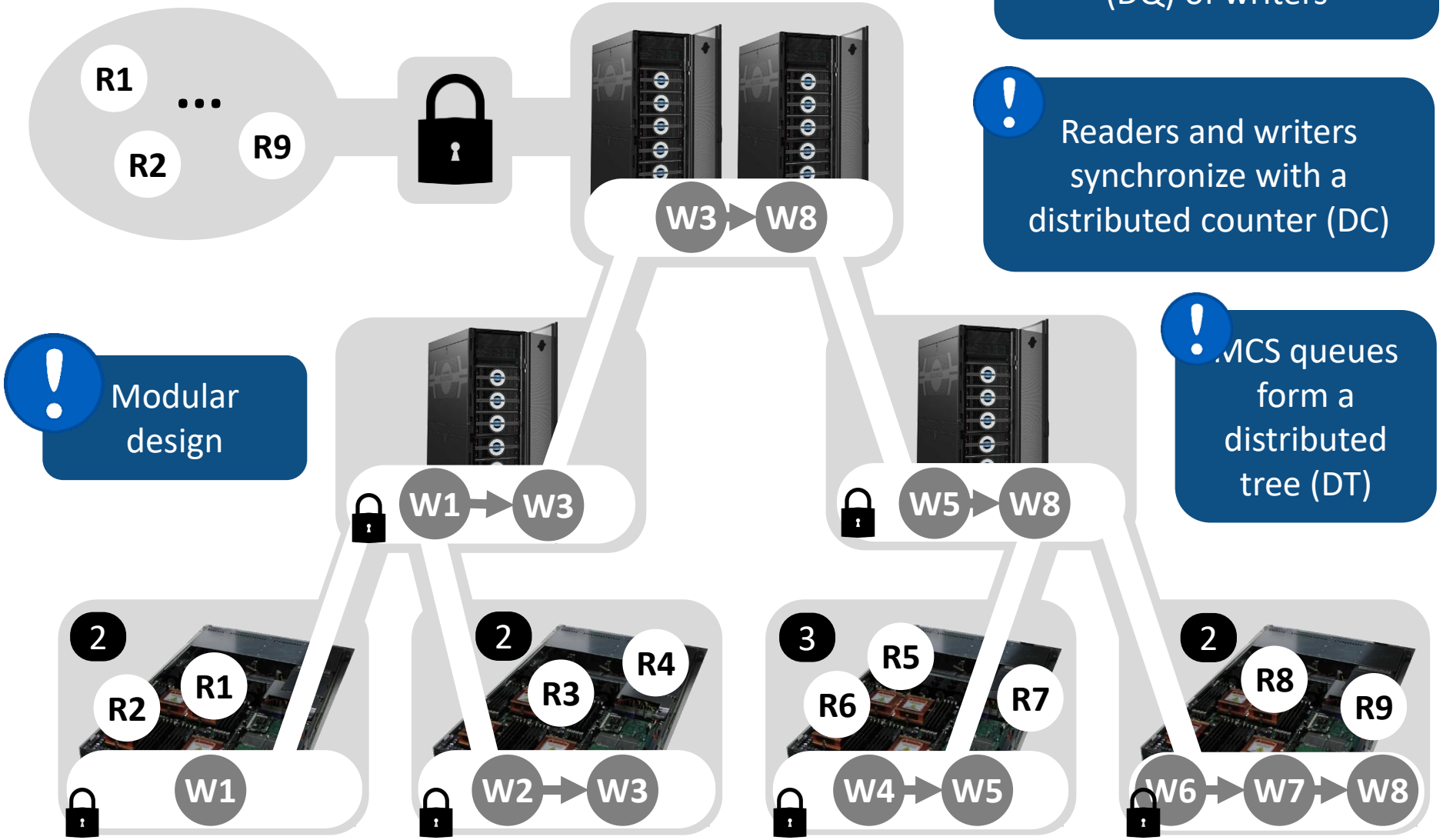
How to manage the design complexity?

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

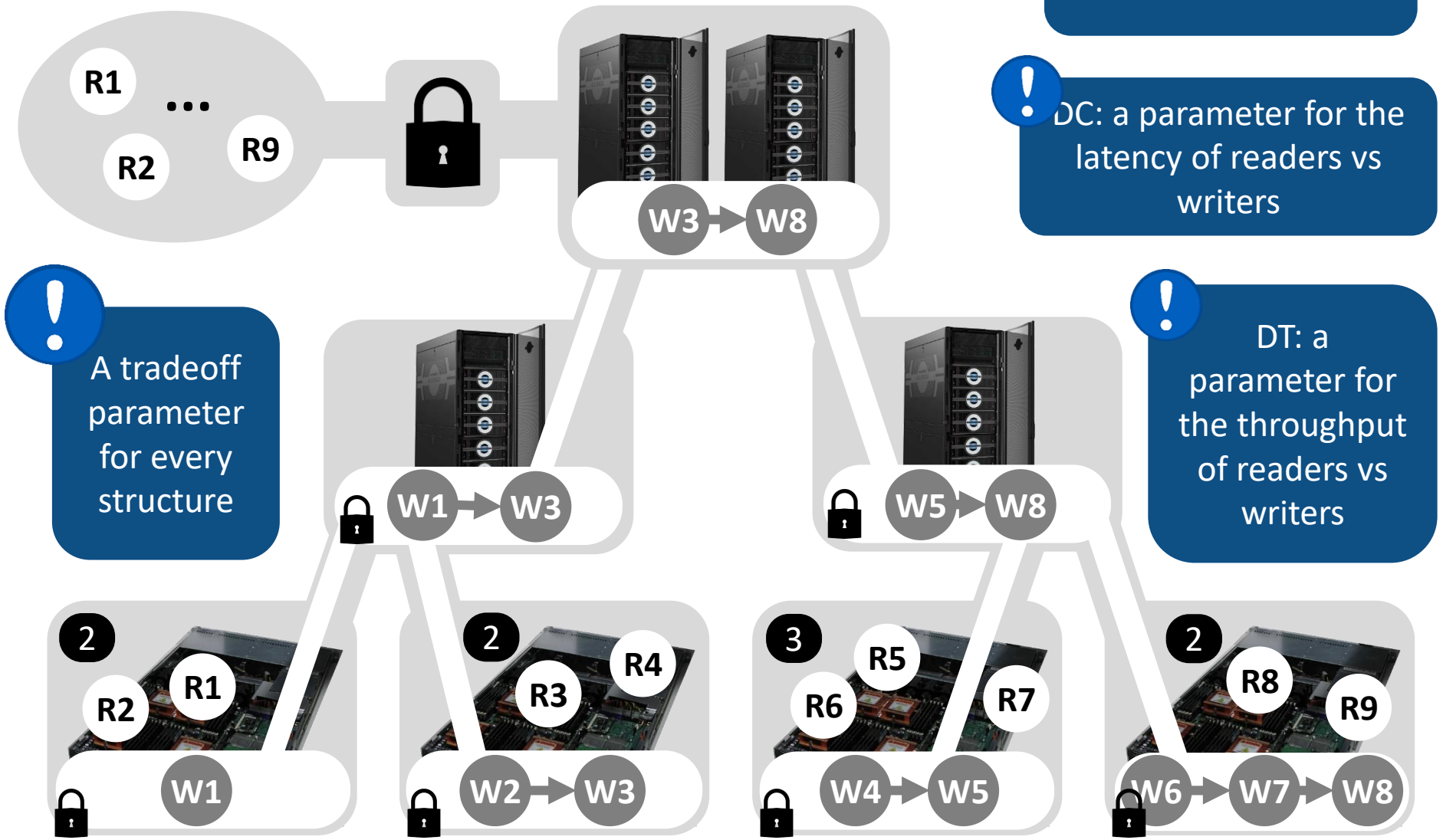
Readers and writers synchronize with a distributed counter (DC)

MCS queues form a distributed tree (DT)

Modular design



How to ensure tunable performance?



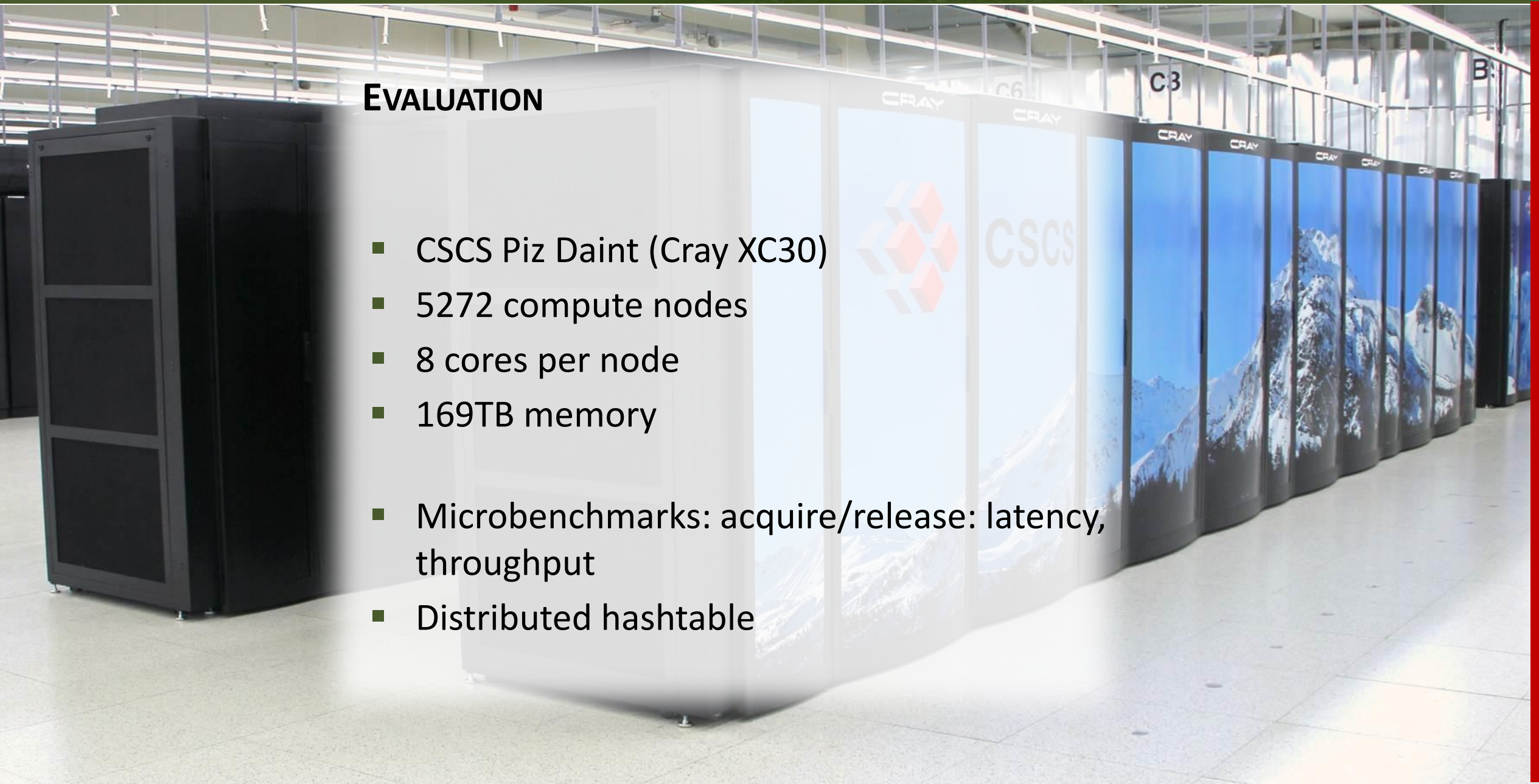
Each DQ: fairness vs throughput of writers

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



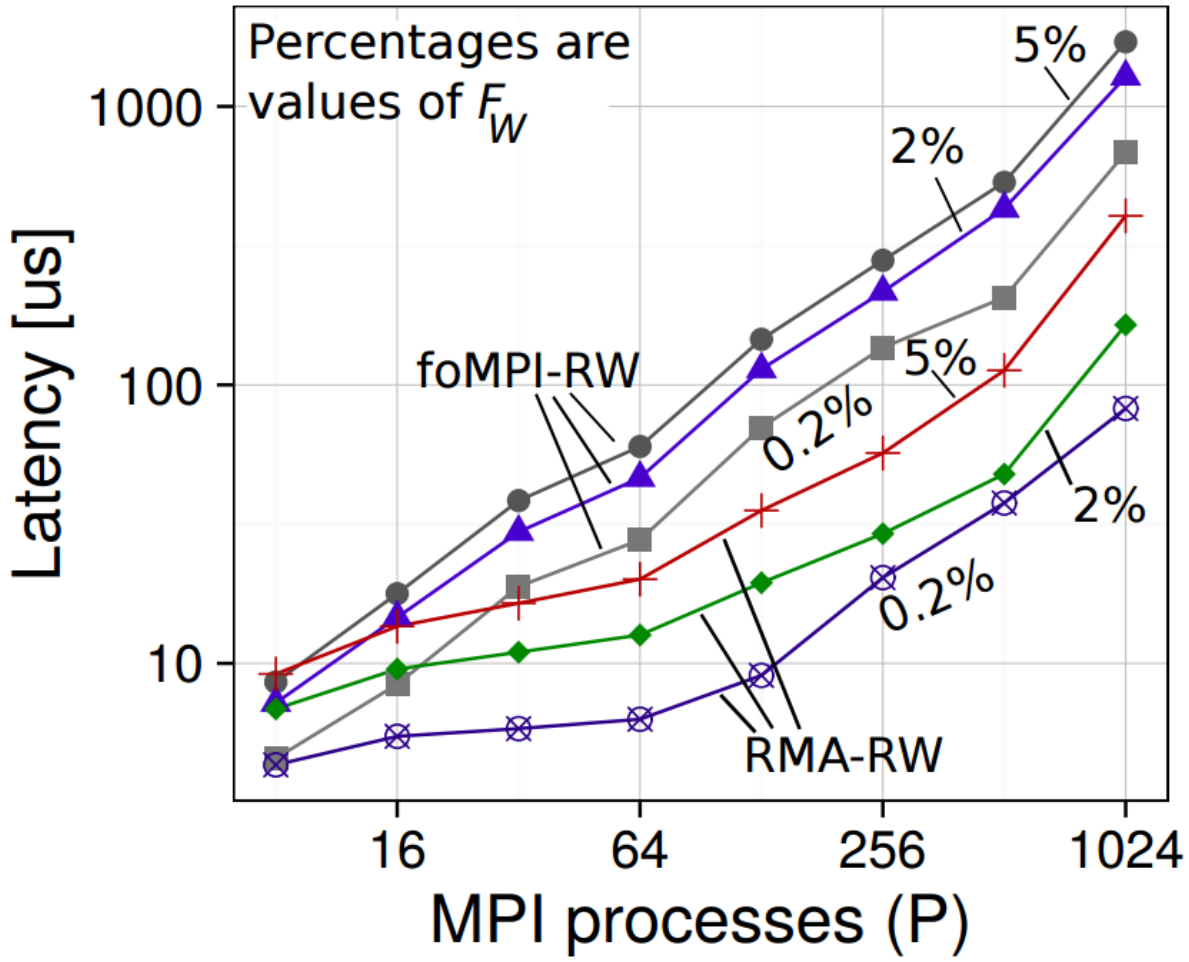
EVALUATION

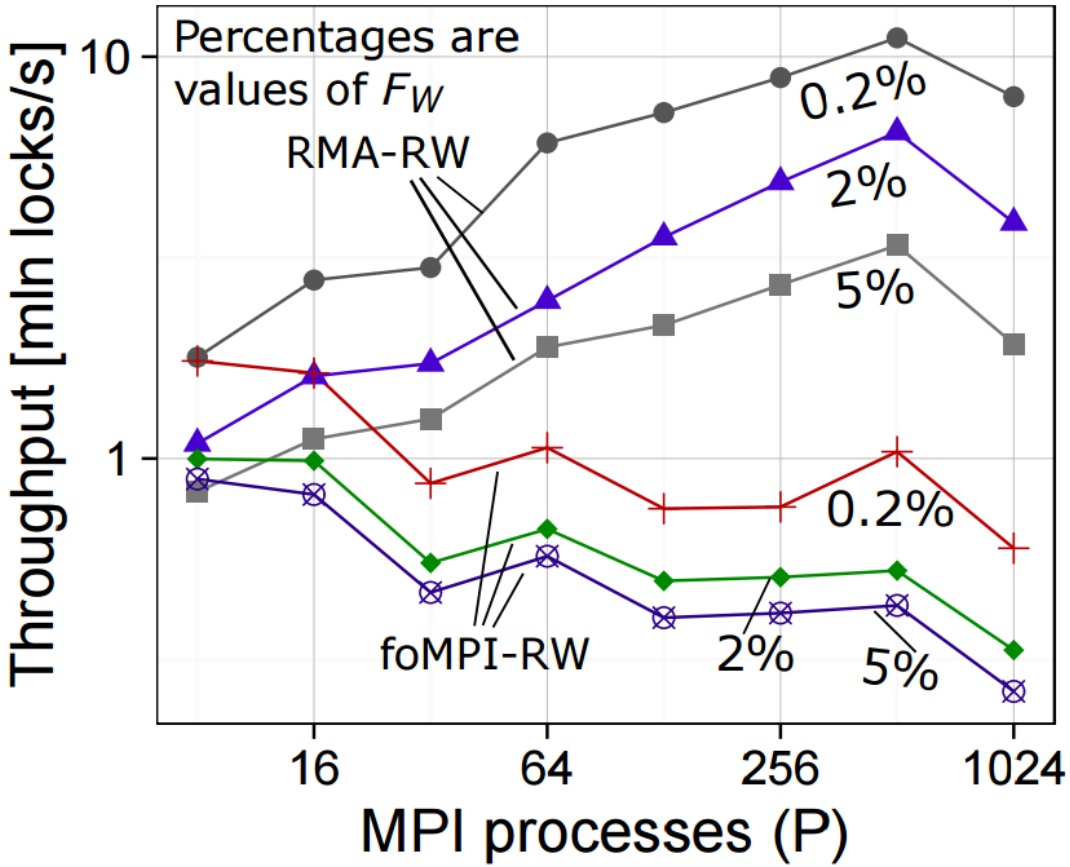
- CSCS Piz Daint (Cray XC30)
- 5272 compute nodes
- 8 cores per node
- 169TB memory

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

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

Throughput, single-operation benchmark

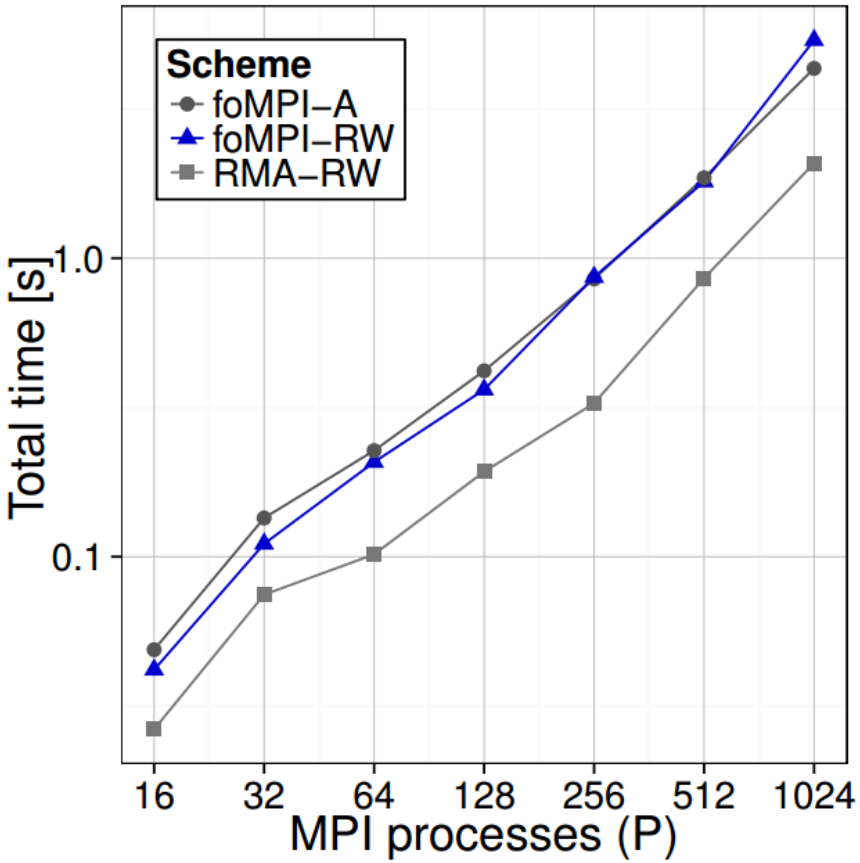




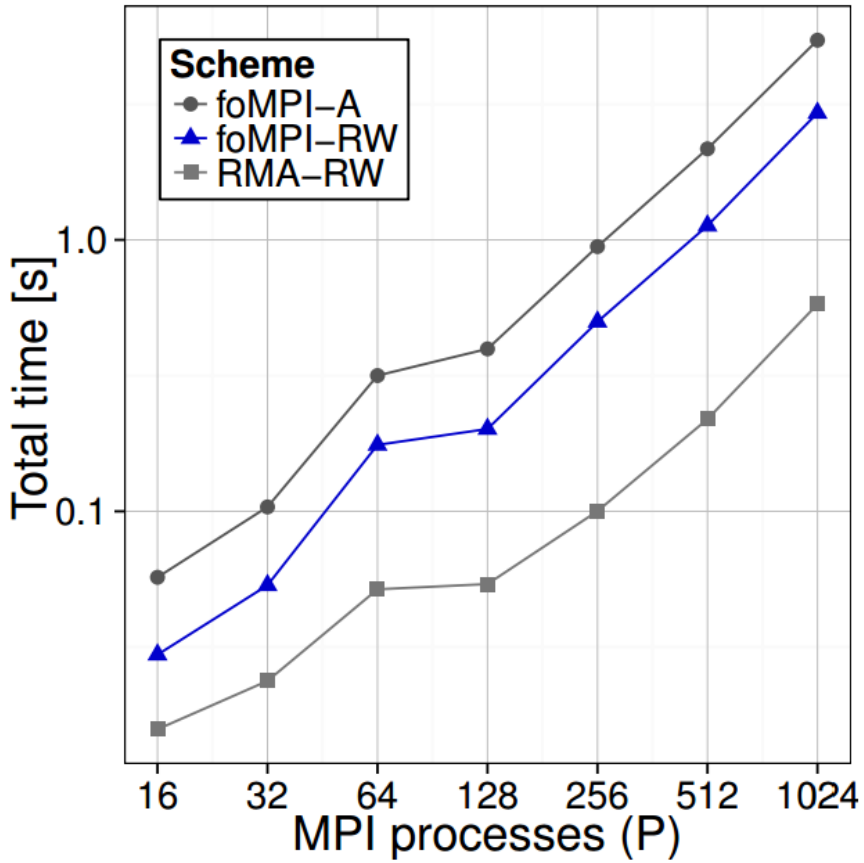
[1] R. Gerstenberger et al. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. ACM/IEEE Supercomputing 2013.

Evaluation - Distributed Hashtable

20% writers



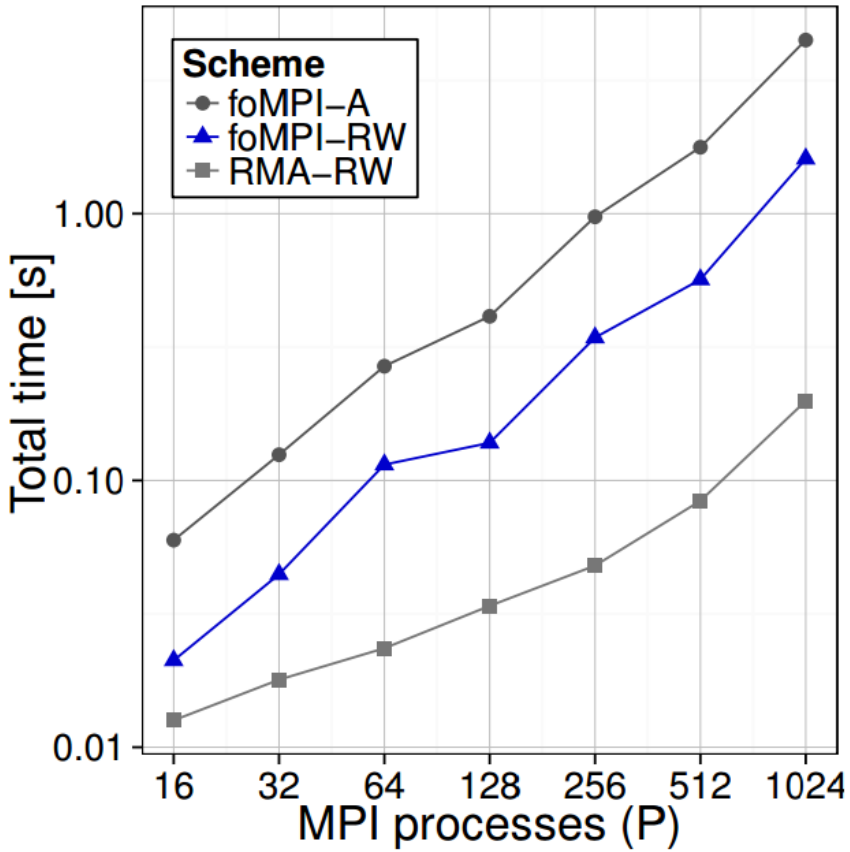
10% writers



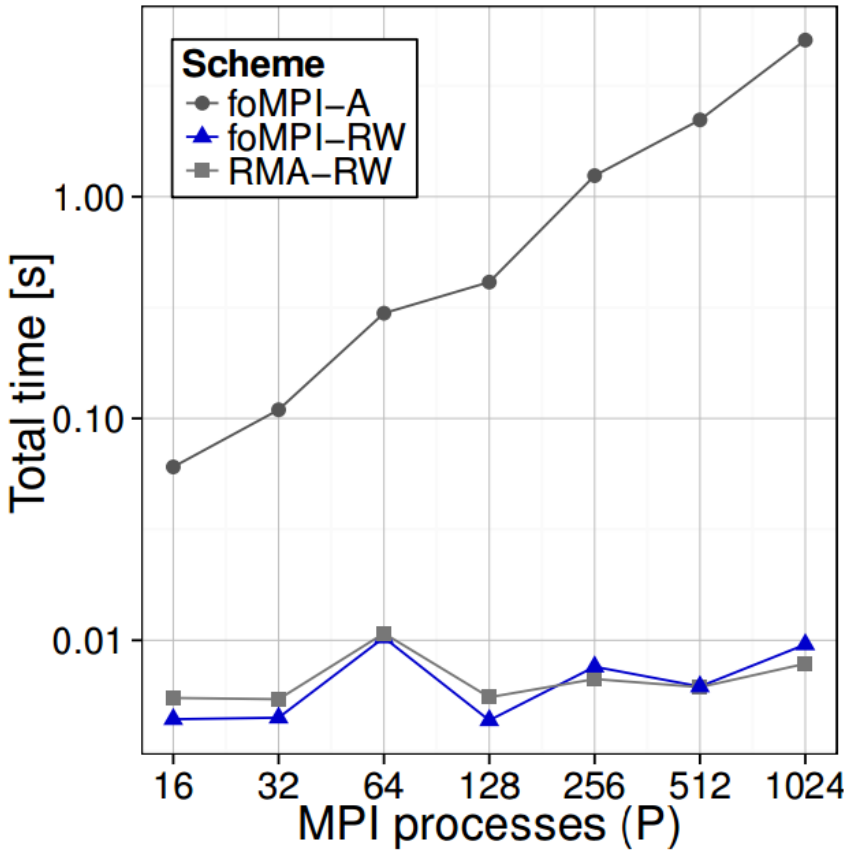
[1] R. Gerstenberger et al. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. ACM/IEEE Supercomputing 2013.

Evaluation - Distributed Hashtable

2% of writers



0% of writers

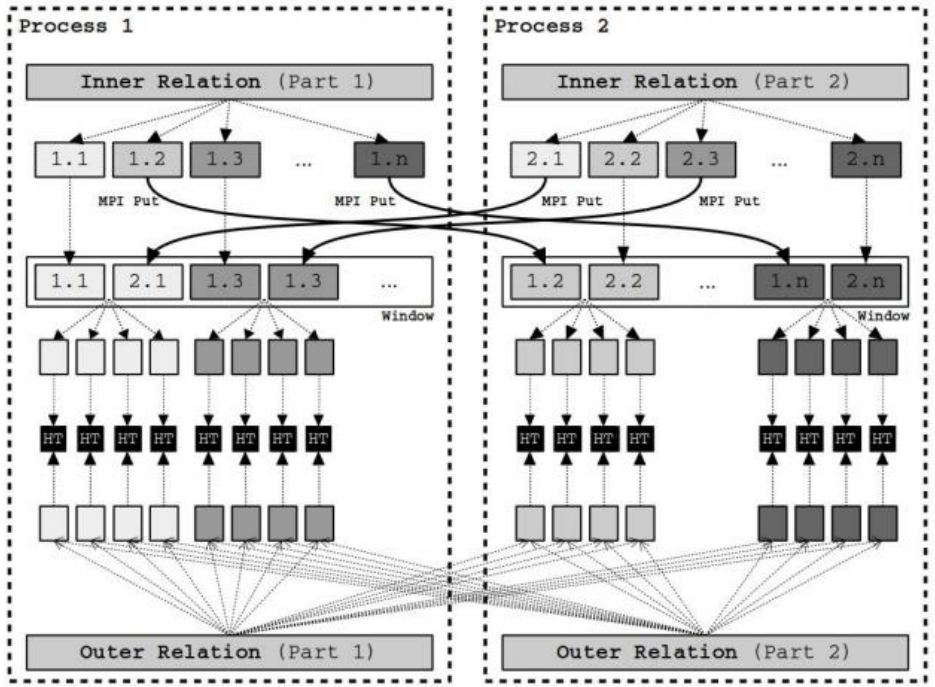


[1] R. Gerstenberger et al. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. ACM/IEEE Supercomputing 2013.

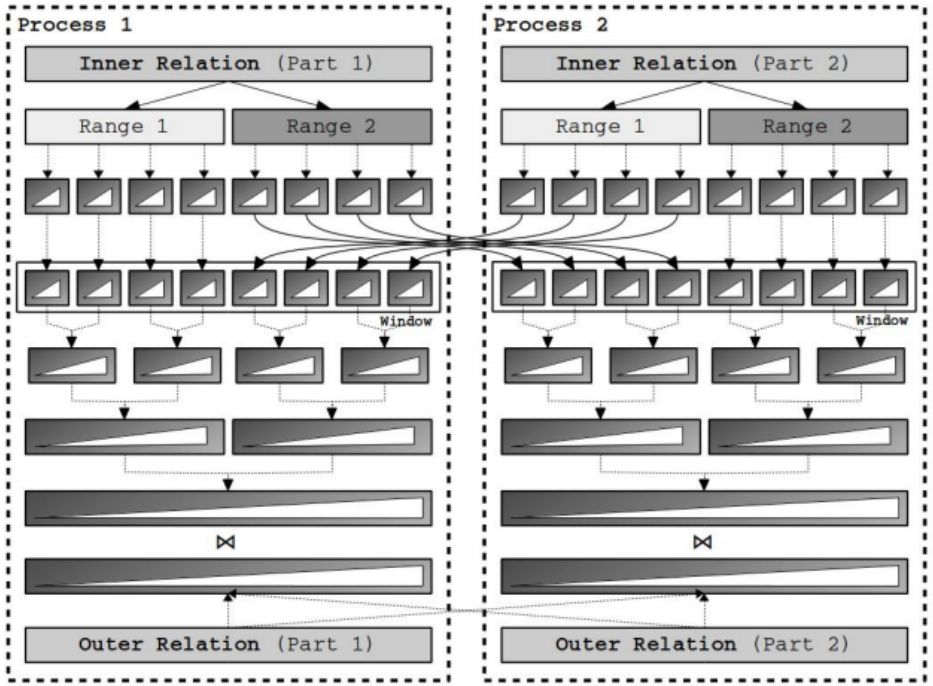
Another application area - Databases

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

Hash-Join



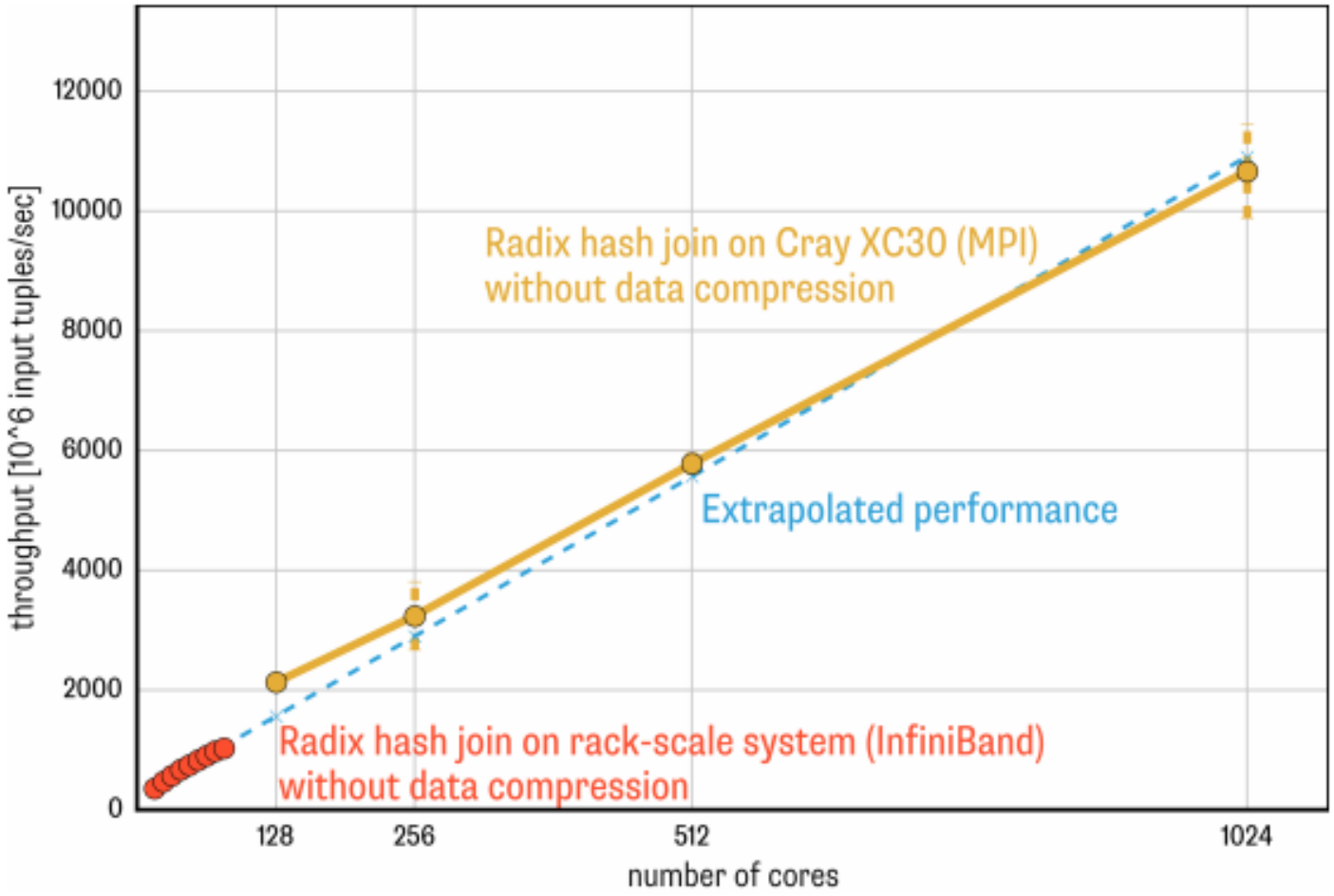
Sort-Join



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

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 **data-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"*

O(1)

Data-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? No
- Prefix sum on an array? Yes!
- Dense matrix multiplication? Yes
- Sparse matrix vector product? No
- Dense matrix vector product? Yes
- Queue-based breadth-first search? No

Obliviousness as property of an execution

*"An algorithm is **data-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 contain 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 data oblivious**
  - Can be programmatically (statically decided)
  - Sufficient for practical use
- **The middle example is not structurally oblivious but data oblivious**
  - First branch is always taken (assuming no overflow) but static dependency analysis is conservative

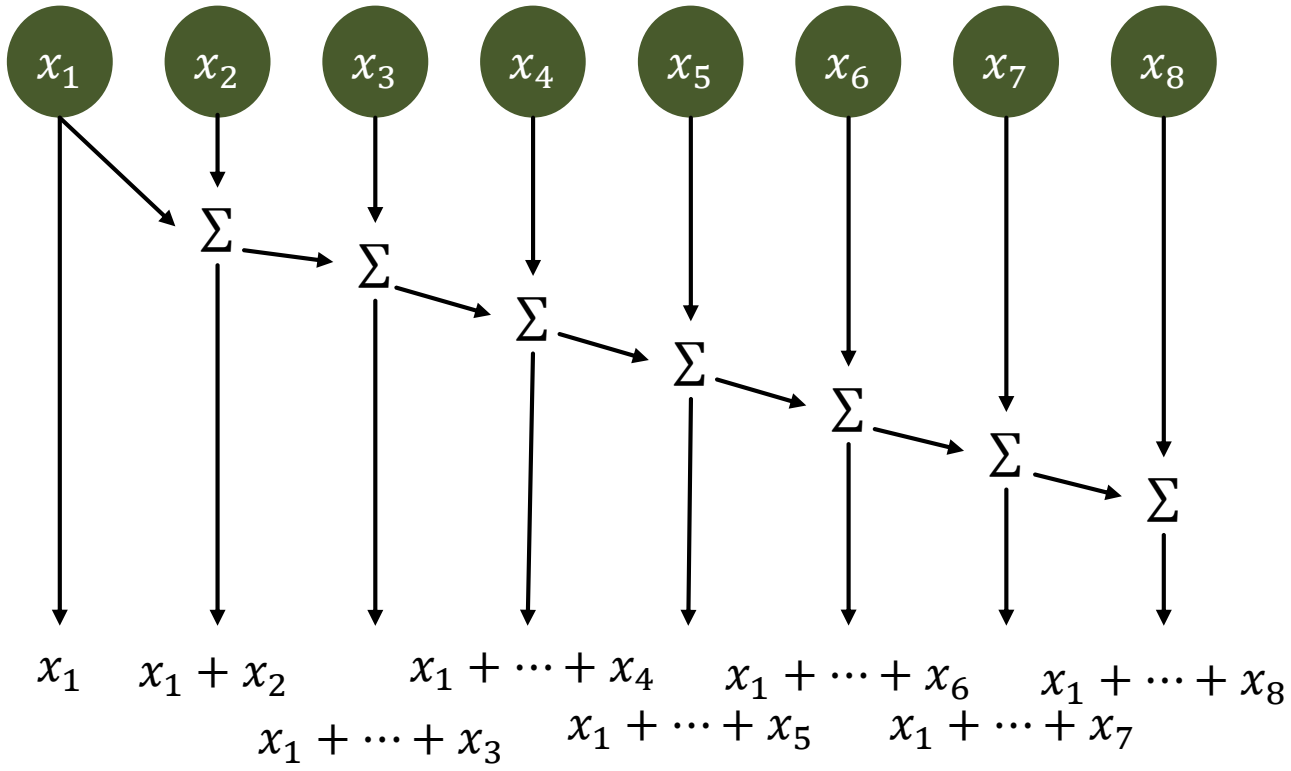
Why obliviousness?

- **We can easily reason about oblivious algorithms**
  - Execution DAG can be constructed "statically"
  - We have done this in the last week intuitively but we never looked 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!

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



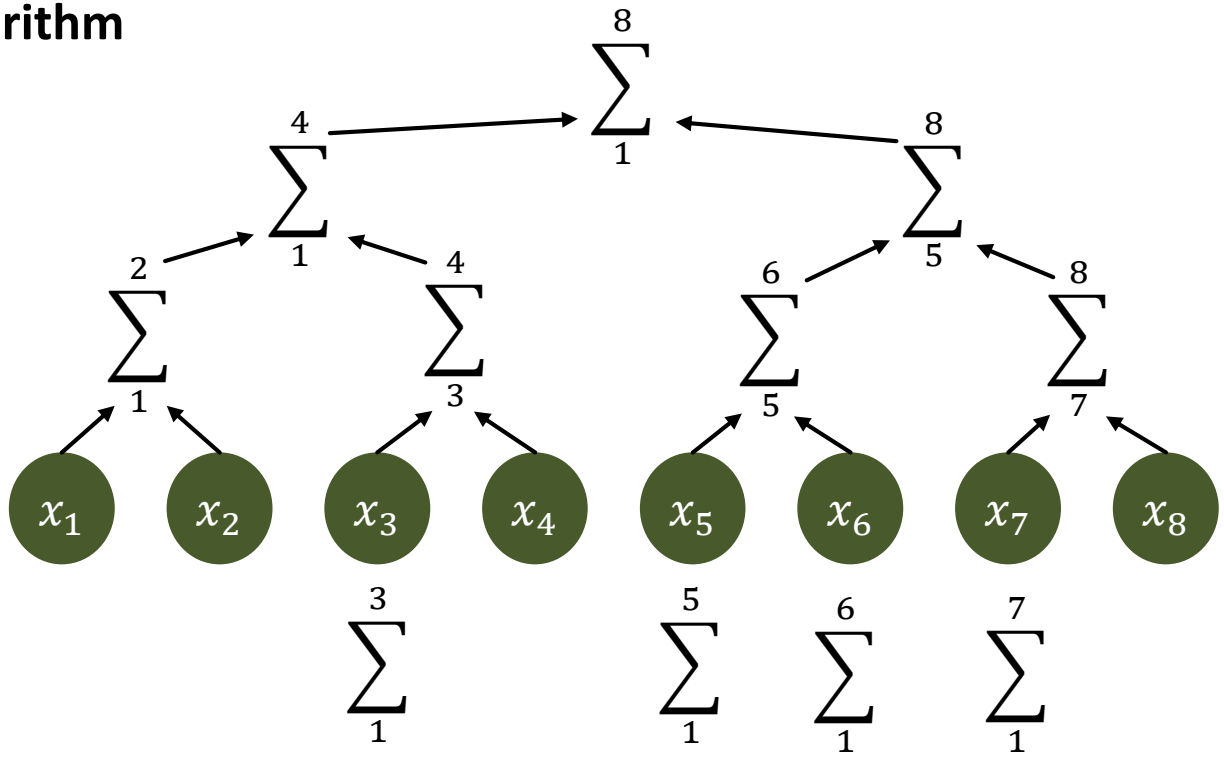
Class question: work and depth?

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

Class question: is this optimal?

What did we learn last week?

- **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)

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

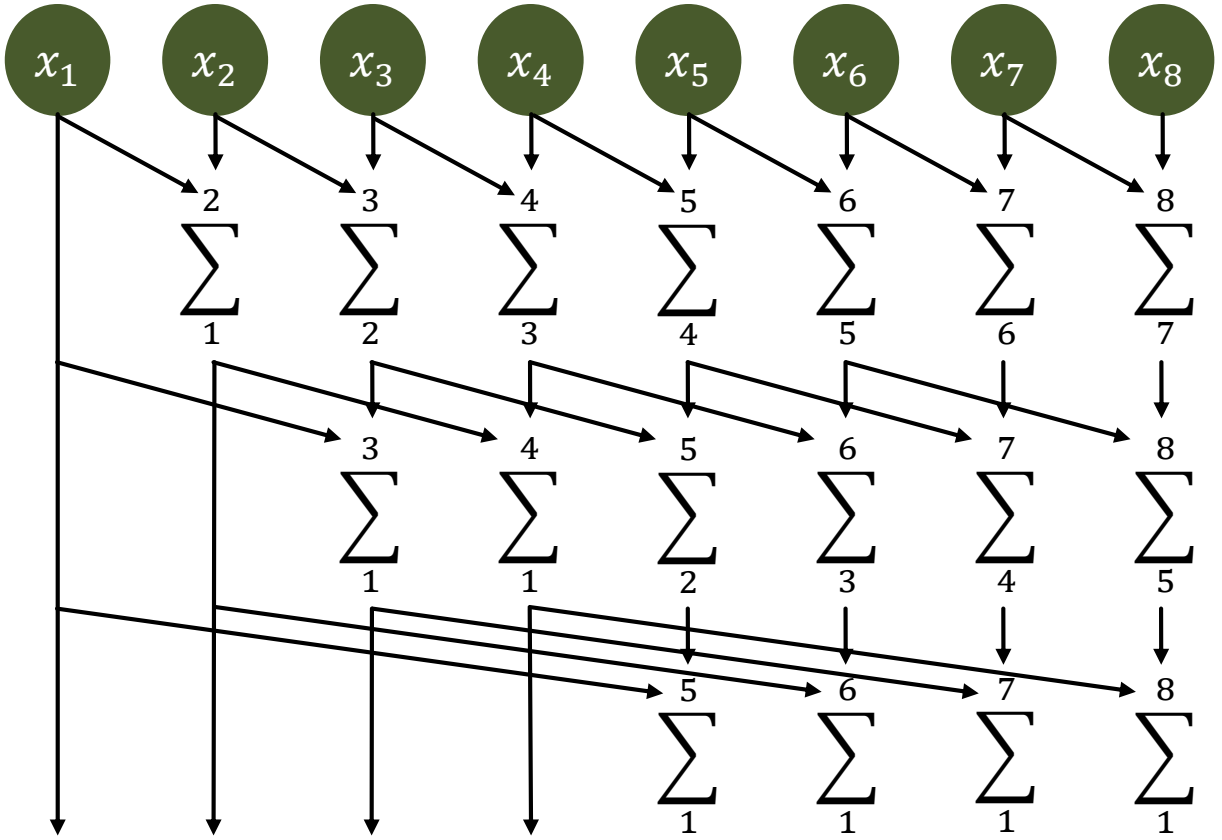
Class question: depth?
(needs to go up and down the tree)

$$\mathrm{D}(n) = 2\log_2 n - 1$$

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**



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

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

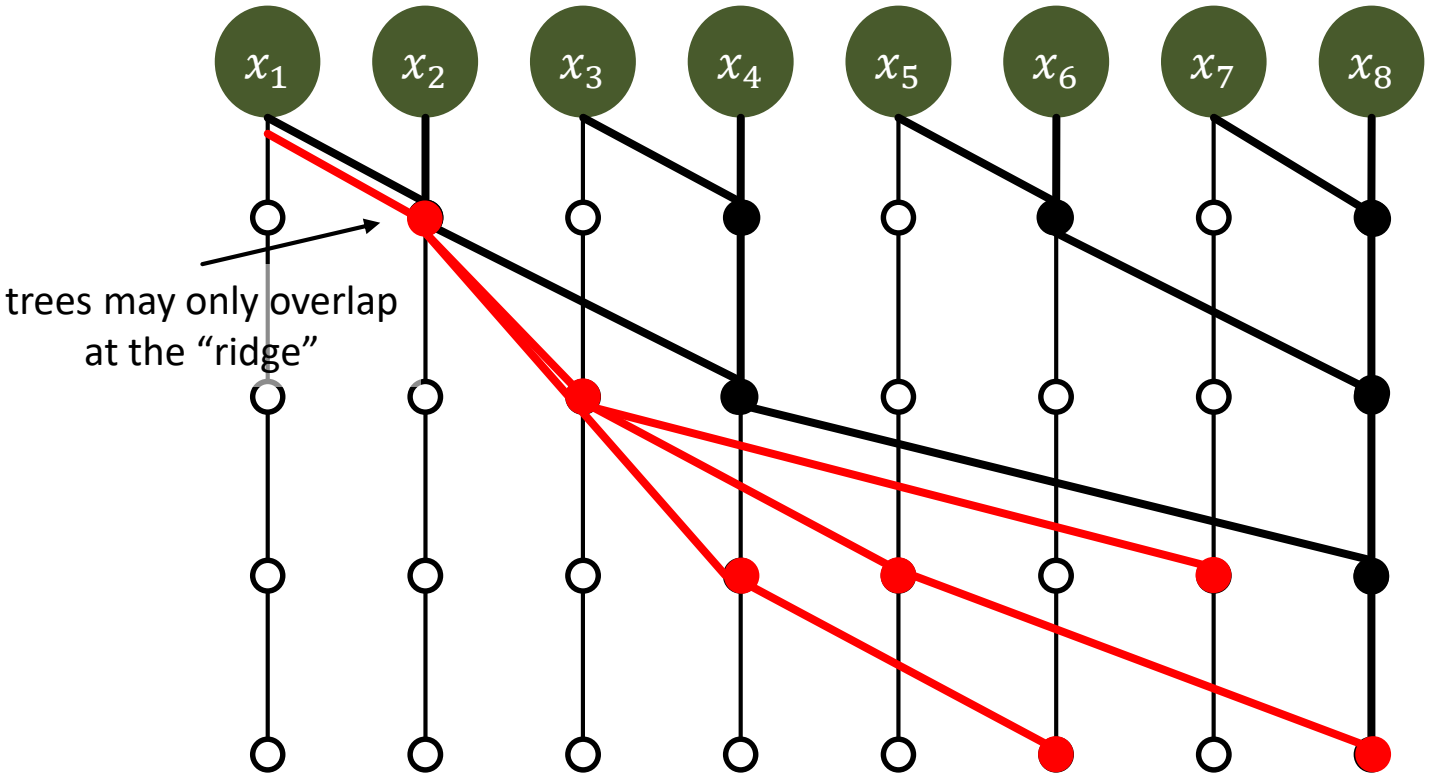
Class question: depth?

$$\mathrm{D}(n) = \log_2 n$$

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: $W + D \geq 2n - 2$



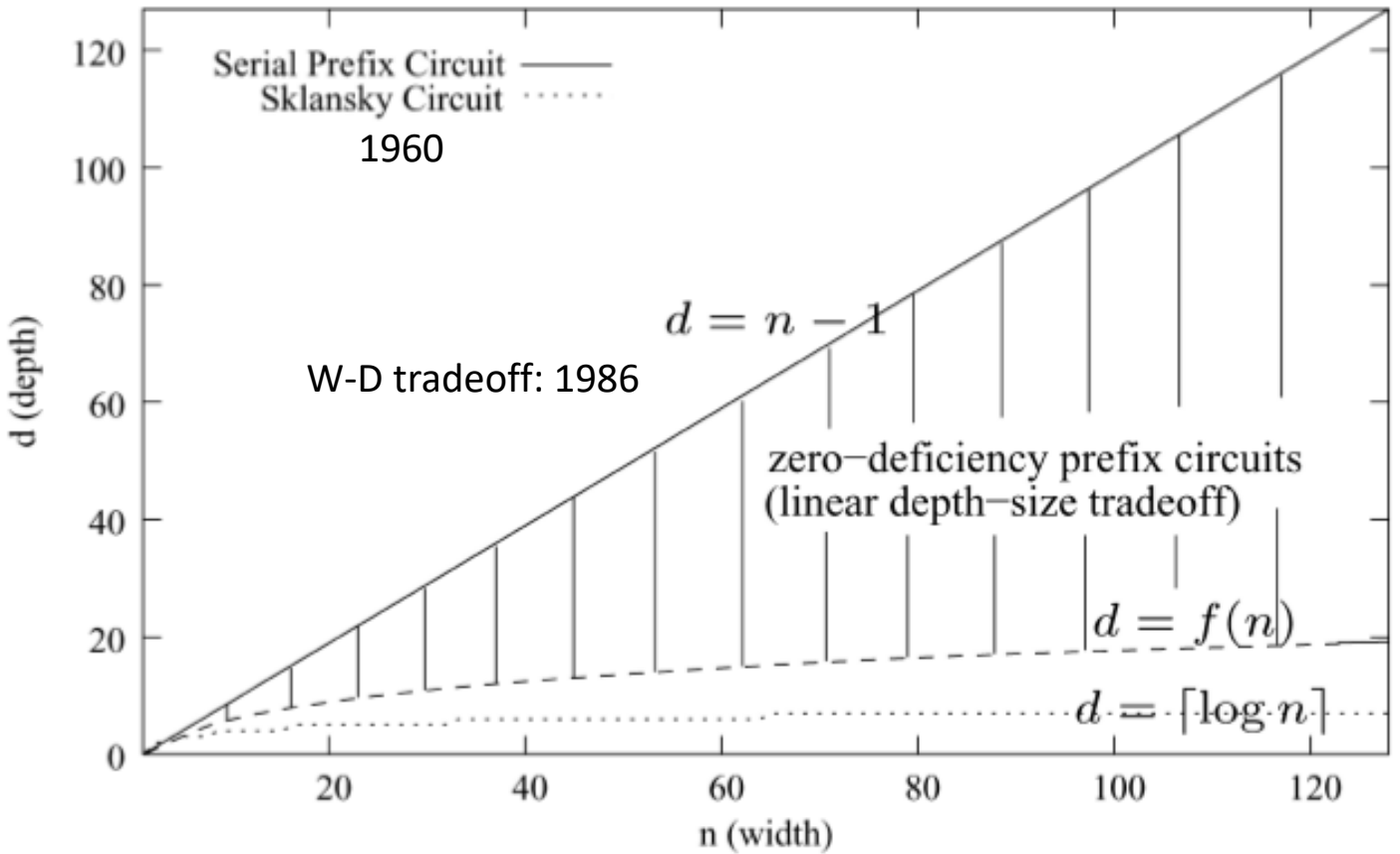
Output tree:
- leaves are all inputs, rooted at x_n
- binary due to binary operation
- $W = n - 1, D = D_o$

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", 2006

Work- and depth-optimal constructions

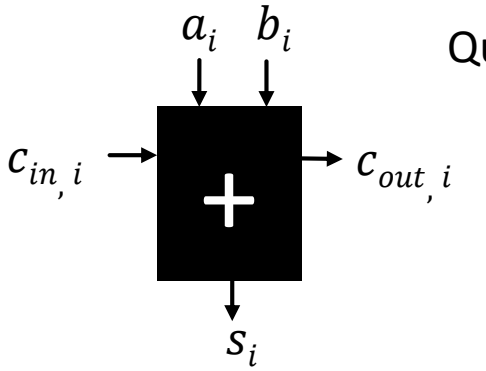
- **Work-optimal?**
  - Only sequential! Why?
  - $W = n - 1$, thus $D = 2n - 2 - W = n - 1$ q.e.d. ☹

- **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 (too 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

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

- **It's the simplest problem to demonstrate and prove 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



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

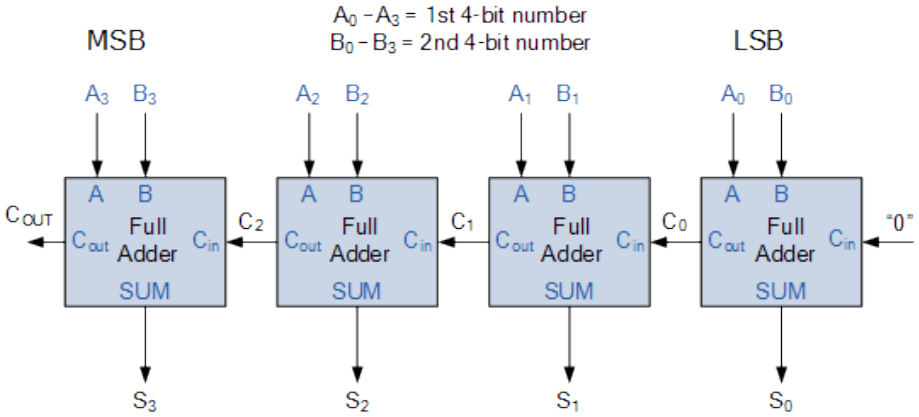
$$s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$$

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

Show example 4-bit addition!

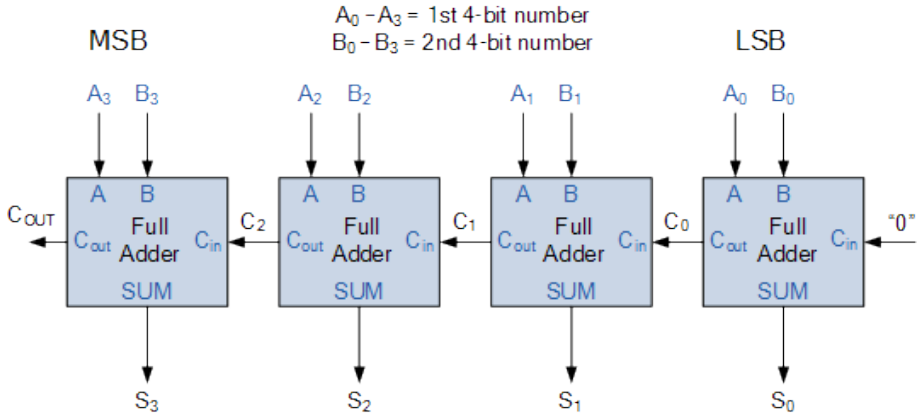
Question: what is work and depth?

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!** $c_{out,i} = a_i \text{ and } b_i \text{ or } c_{in,i} \text{ and } (a_i \text{ xor } b_i)$ $s_i = a_i \text{ xor } b_i \text{ xor } c_{in,i}$
  - Requires $c_{in,1}, c_{in,2}, \dots, c_{in,n}$ though ☹
- **Carry bits can be computed with a scan!**
  - Model carry bit as state starting with 0

    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 ($M_{a_i b_i}$):

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

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

Exercise: simplify!

- **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

Prefix sums as magic bullet for other seemingly sequential algorithms

- **Any time a sequential chain can be modeled as function composition!**
  - Let $f_1, \dots, f_n$ be an ordered set of functions and $f_0(x) = x$
  - Define ordered function compositions: $f_1(x); f_2(f_1(x)); \dots; f_n(\dots f_1(x))$
  - 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 $\boldsymbol{f_i(x) = a_i f_{i-1}(x) + b_i}$ 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: $f_2\binom{x}{1} = \begin{pmatrix} a_2 & b_2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} a_1 & b_1 \\ 0 & 1 \end{pmatrix} f_0\binom{x}{1} = \begin{pmatrix} a_1a_2 & a_2b_1 + b_2 \\ 0 & 1 \end{pmatrix}\binom{x}{1}$

- **Most powerful! Homework:**
  - Parallelize tridiagonal solve (e.g., Thomas' algorithm)
  - Parallelize string parsing

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

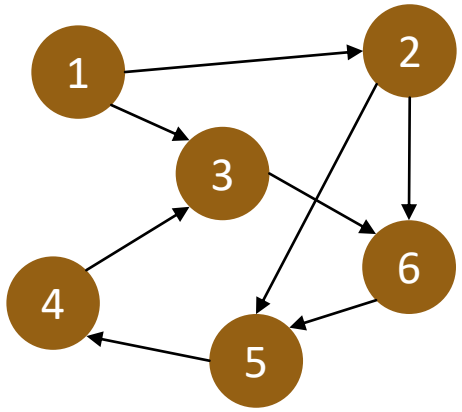
```
low = prefix_sum(!bit, sum)
high = n+1-backwards_prefix_sum(bit, sum)
new_idx = (bit == 0) : low ? high
b[new_idx-1] = a[i]
swap(a,b)
```

Show one example iteration!

Question: work and depth?

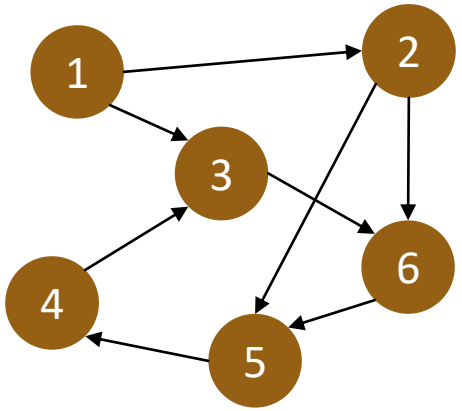
Oblivious graph algorithms

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



j \ i						
	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



j \ i						
	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

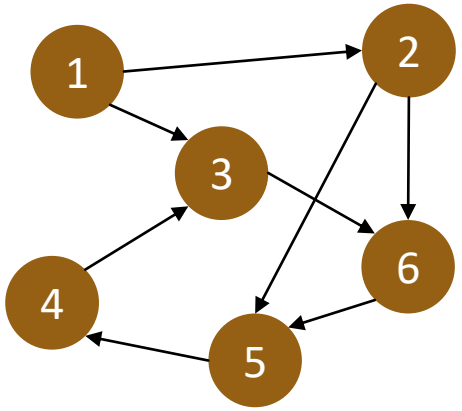
Weighted graph – general matrix

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)
    → *Multiplication by additive identity annihilates*
  - Semirings are denoted by tuples (S, +, *, 0, 1)
    *“Standard” ring of rational numbers: ($\mathbb{R}$, +, *, 0, 1)*
    Boolean semiring: ({0,1}, $\vee$, $\wedge$, 0, 1)
    Tropical semiring: ($\mathbb{R} \cup \{\infty\}$, min, +, ∞, 0) (also called min-plus semiring)

Oblivious shortest path search

- **Construct distance matrix from adjacency matrix by replacing all off-diagonal zeros with ∞**
- **Initialize distance vector d_0 of size n to ∞ everywhere but zero at start vertex**
  - E.g., $\mathbf{d_0} = (\infty, \mathbf{0}, \infty, \infty, \infty, \infty)^T$
    Show evolution when multiplied!



- **SSSP can be performed with n+1 matrix-vector multiplications!**
  - Question: total work and depth?
    $W = O(n^3), D = O(n \log n)$
  - Question: Is this good? Optimal?
    Dijkstra = $O(|E| + |V|\log|V|)$ ☹

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

- **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?**
  - Multiply the matrix $(A + I)$ n times with itself in the Boolean semiring!
  - Why?

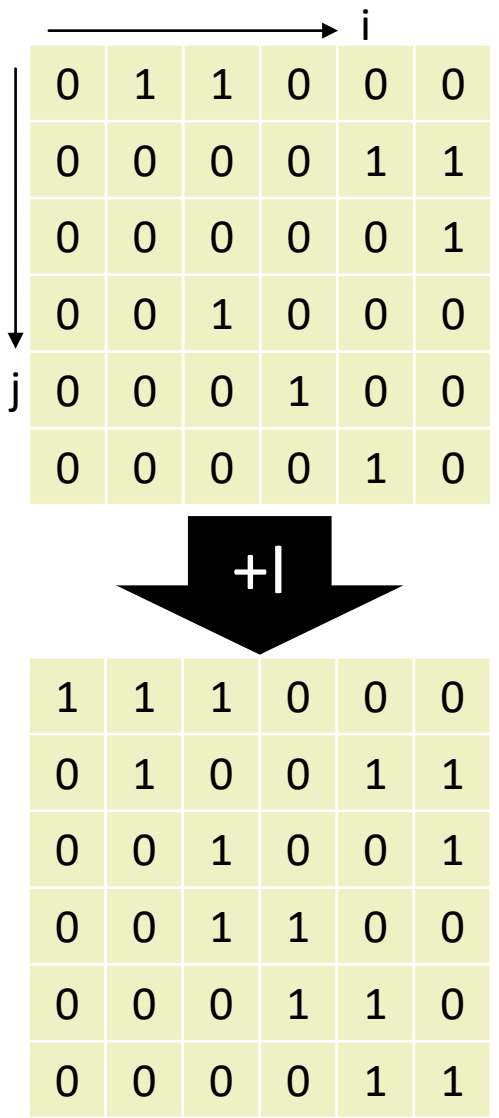
    Demonstrate that $(A + I)^2$ has 1s for each path of at most length 1

    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! $W = O(n^3 \log n)$ $D = O(\log^2 n)$

    $\lceil \log_2 n \rceil$ *multiplications (think $A^4 = A^{2^2}$)*

- **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 column (vertex) perform min-reduction to determine its component label!
  - Overall work and depth? $W = O(n^3 \log n)$, $D = O(\log^2 n)$



Many if not all graph problems have oblivious or tensor variants!

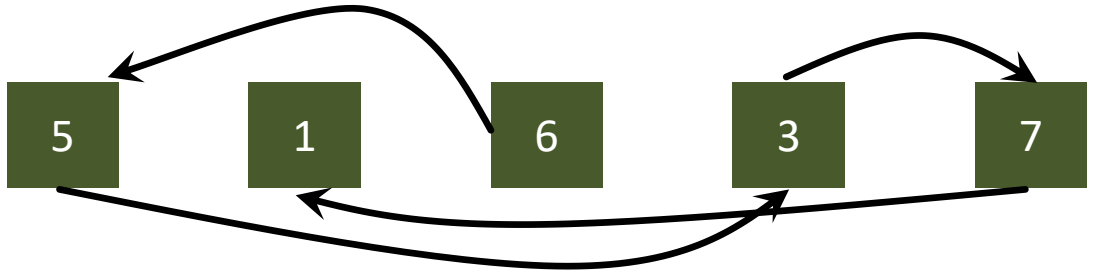
- **Not clear whether they are most efficient**
  - Efforts such as GraphBLAS exploit existing BLAS implementations and techniques
- **Generalizations to other algorithms possible**
  - Can everything be modeled as tensor computations on the right ring?
  - E. Solomonik, T. Hoefler: *"Sparse Tensor Algebra 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!
  - E.g., use sparse graphs for graph algorithms on semirings (if $|E| < |V|^2/\log|V|$)
    May recover some of the lost efficiency by computing zeros!
- **Now moving to non-oblivious ☺**

Nonoblivious parallel algorithms

- **Outline:**
  - Reduction on a linked list
  - Prefix sum on a linked list
  - Nonoblivious graph algorithms - connected components
  - Conflict graphs of bounded degree
- **Modeling assumptions:**
  - When talking about work and depth, we assume each loop iteration on a single PE is unit-cost (may contain multiple instructions!)

Reduction on a linked list

- **Given: n values in linked list, looking for sum of all values**




```
typedef struct elem {
  struct elem *next;
  int val;
} elem;
```

- **Sequential algorithm:**

```
set S={all elems}
while (S != empty) {
  pick some i ∈ S;
  S = S - i.next;
  i.val += i.next.val;
  i.next = i.next.next;
}
```

A set $I \subset S$ is called an **independent set** if no two elements in I are connected!

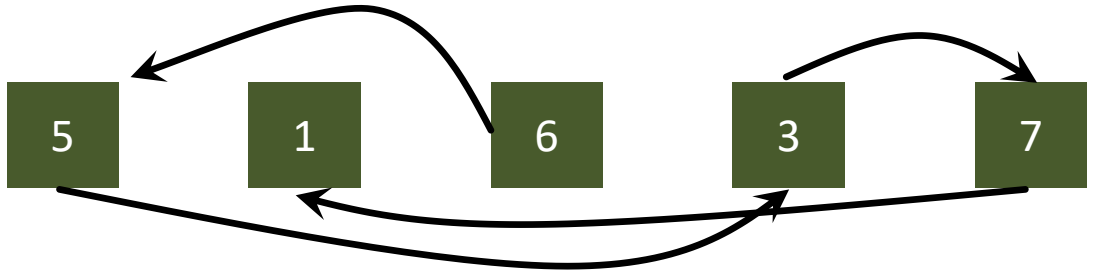
Are the following sets independent or not?

- {1}
- {1,5}
- {1,5,3}
- {7,6,5}
- {7,6,1}

Class question: What is the maximum size of an independent set of a linked list with n elements?

Parallel reduction on a linked list

- **Given: n values in linked list, looking for sum of all values**




```
typedef struct elem {
  struct elem *next;
  int val;
} elem;
```

A subset $I \subset S$ is called an **independent set** if no two elements in I are connected!

- **Parallel algorithm:**

```
set S={all elems}
while (S != empty) {
  pick independent subset I ∈ S;
  for(each i ∈ I do in parallel) {
    S = S – i.next;
    i.val += i.next.val;
    i.next = i.next.next;
  }
}
```

Basically the same algorithm, just working on independent subsets!

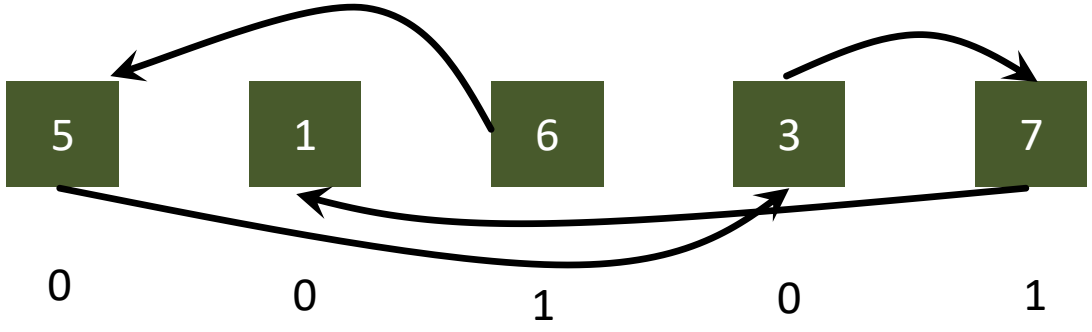
Class question: Assuming picking a maximum I is free, what are work and depth?

$$W = n - 1, D = \lceil \log_2 n \rceil$$

Is this optimal?

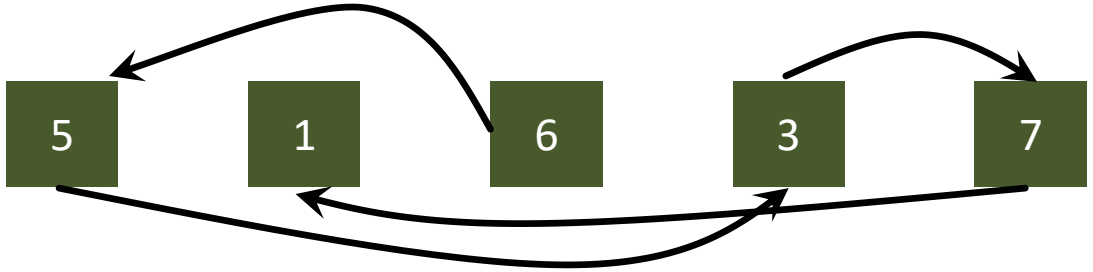
How to pick the independent set I?

- **That's now the whole trick!**
  - It's simple if all linked values are consecutive in an array – same as "standard" reduction!
    Can compute independent set up-front!
- **Irregular linked list though?**
  - Idea 1: find the order of elements → requires parallel prefix sum, D'oh!
  - Observation: if we pick $|I| > \lambda|V|$ in each iteration, we finish in logarithmic time!
- **Symmetry breaking:**
  - Assume p processes work on p consecutive nodes
  - How to find the independent set?
    They all look the same (well, only the first and last differ, they have no left/right neighbor)
    Local decisions cannot be made ☹
- **Introduce randomness to create local differences!**
  - Each node tosses a coin → 0 or 1
  - Let I be the set of nodes such that v drew 1 and $v.next$ drew 0!
    Show that I is indeed independent!
    What is the probability that $v \in I$? $P(v \in I) = \frac{1}{4}$



Optimizations

- **As the set shrinks, the random selection will get less efficient**
  - When p is close to n ($|S|$) then most processors will fail to make useful progress
  - Switch to a different algorithm
- **Recursive doubling!**




```
for (i=0; i ≤ ⌈log2n⌉; ++i) {
  for(each elem do in parallel) {
     elem.val += elem.next.val;
     elem.next = elem.next.next;
  }
}
```

- Show execution on our example!
- Algorithm computes prefix sum on the list!

  Result at original list head is overall sum

Class question: What are work and depth?

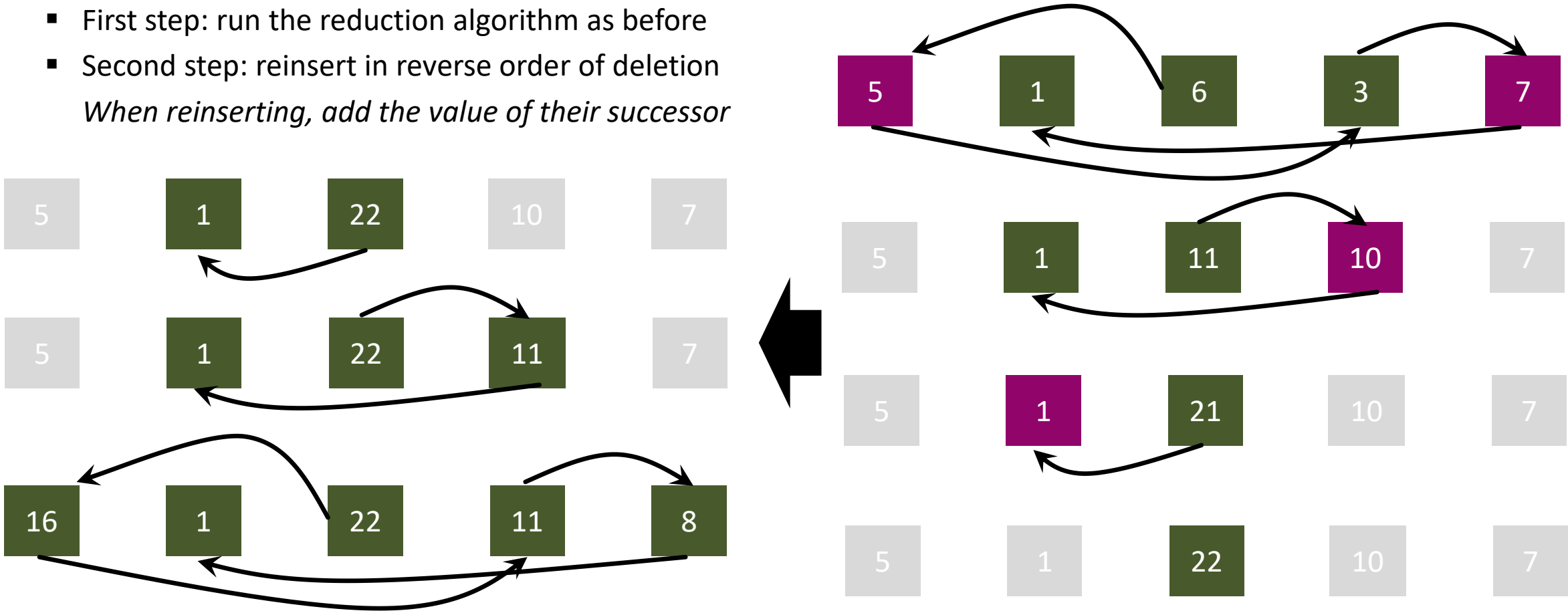
$$W = n\lceil\log_2 n\rceil,\ D = \lceil\log_2 \mathrm{n}\rceil$$

Prefix summation on a linked list

- **Didn't we just see it? Yes, but work-inefficient (if $p \ll n$)!**

  We extend the randomized symmetry-breaking reduction algorithms
  - First step: run the reduction algorithm as before
  - Second step: reinsert in reverse order of deletion

    When reinserting, add the value of their successor



Prefix summation on a linked list

- **Didn't we just see it? Yes, but work-inefficient (if $p \ll n$)!**

  We extend the randomized symmetry-breaking reduction algorithms
  - First step: run the reduction algorithm as before
  - Second step: reinsert in reverse order of deletion

    When reinserting, add the value of their successor

- **Class question: how to implement this in practice?**
  - Either recursion or a stack!
  - Design the algorithm as homework (using a parallel for loop)

Finding connected components as example

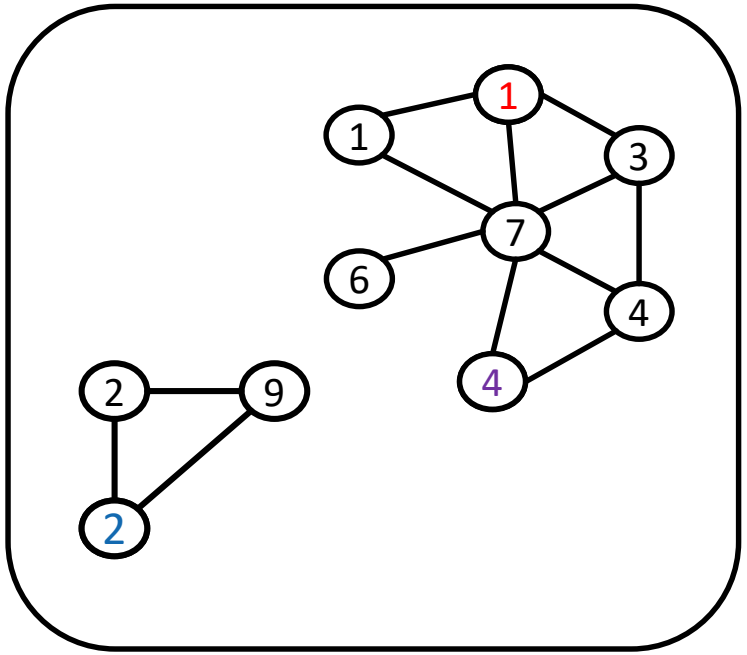
*A **connected component** of an undirected graph is a subgraph in which any two vertices are connected by a path and no vertex in the subgraph is connected to any vertices outside the subgraph. Each undirected graph G = (V,E) contains one or multiple (at most |V|) connected components.*

- **Straight forward and cheap to compute sequentially – question: how?**
  - Any traversal algorithm in work $O(|V| + |E|)$

    Seemingly trivial - becomes very interesting in parallel
  - Our oblivious semiring-based algorithm was $W = O(n^3 \log n)$, $D = O(\log^2 n)$

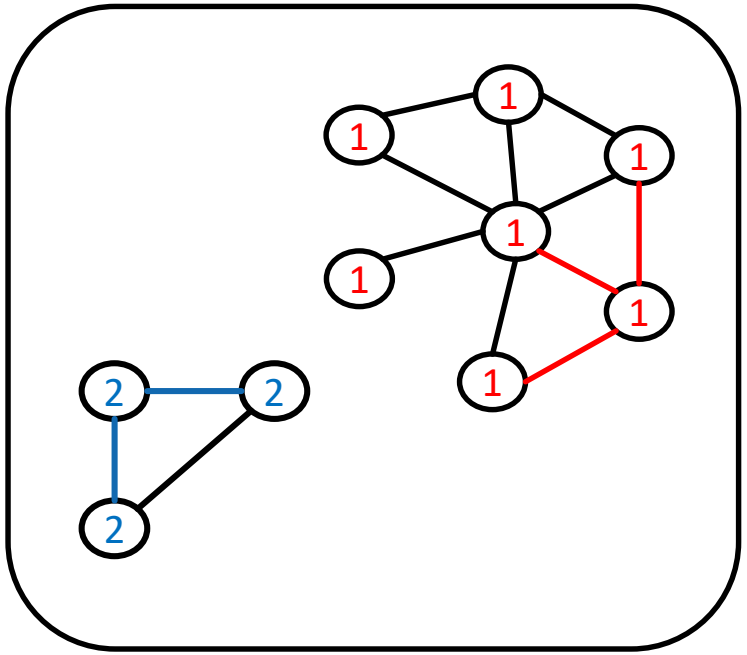
    FAR from work optimality! Question: can we do better by dropping obliviousness?

Connected Components



Label Propagation

Connected Components

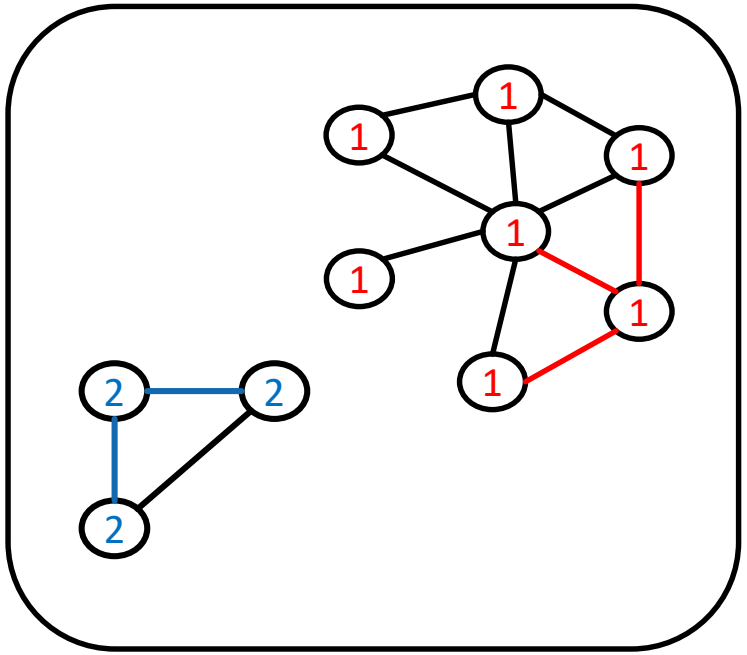


Label Propagation

Work = $\mathcal{O}(D \cdot m)$

Depth = $\mathcal{O}(D)$

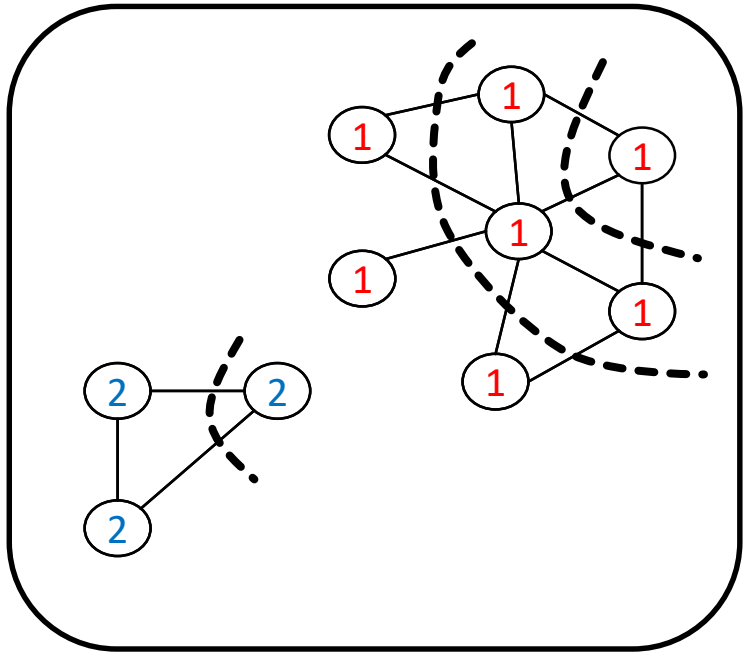
Connected Components



Label Propagation

Work = $\mathcal{O}(D \cdot m)$

Depth = $\mathcal{O}(D)$



Breadth-First Search

Work = $\mathcal{O}(m + n)$

Depth = $\mathcal{O}(\sum_{i=1}^{C} D(c_i))$

Finding connected components as example

*A **connected component** of an undirected graph is a subgraph in which any two vertices are connected by a path and no vertex in the subgraph is connected to any vertices outside the subgraph. Each undirected graph G = (V,E) contains one or multiple (at most |V|) connected components.*

- **Straight forward and cheap to compute sequentially – question: how?**
  - Any traversal algorithm in work $O(|V| + |E|)$

    Seemingly trivial - becomes very interesting in parallel
  - Our oblivious semiring-based algorithm was $W = O(n^3 \log n)$, $D = O(\log^2 n)$

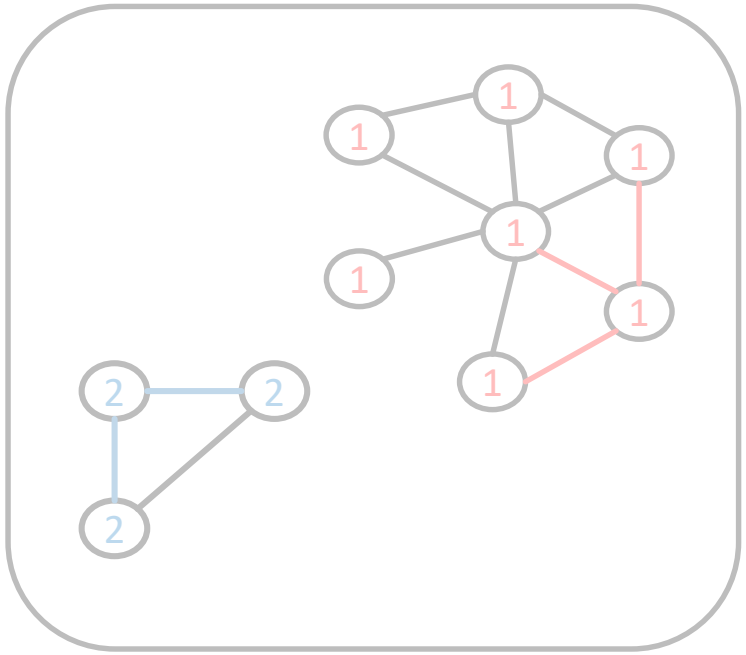
    FAR from work optimality! Question: can we do better by dropping obliviousness?
- **Let's start simple – assuming concurrent read/write is free**
  - Arbitrary write wins
- **Concept of supervertices**
  - A supervertex represents a set of vertices in a graph

  1. Initially, each vertex is a (singleton) supervertex
  2. Successively merge neighboring supervertices
  3. When no further merging is possible → each supervertex is a component

  - Question is now only about the merging strategy

*A **fixpoint algorithm** proceeds iteratively and monotonically until it reaches a final state that is not left by iterating further.*

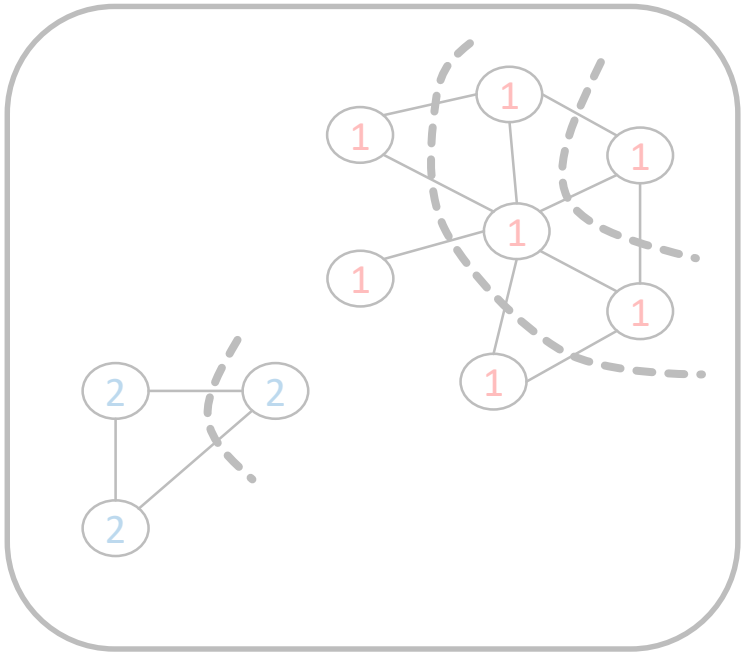
Connected Components



Label Propagation

Work $= \mathcal{O}(D \cdot m)$

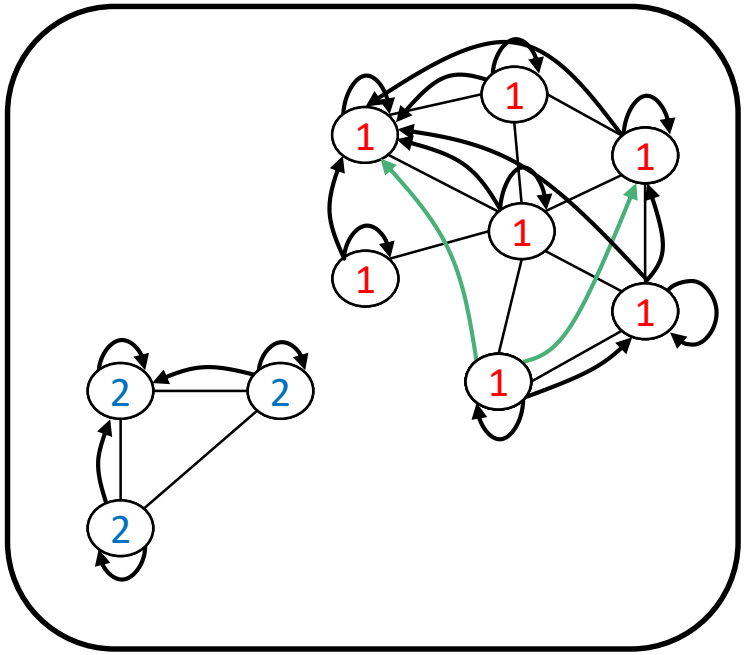
Depth $= \mathcal{O}(D)$



Breadth-First Search

Work $= \mathcal{O}(m + n)$

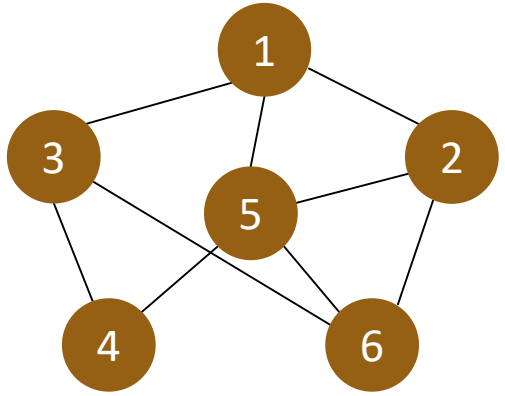
Depth $= \mathcal{O}(\sum_{i=1}^{C} D(c_i))$



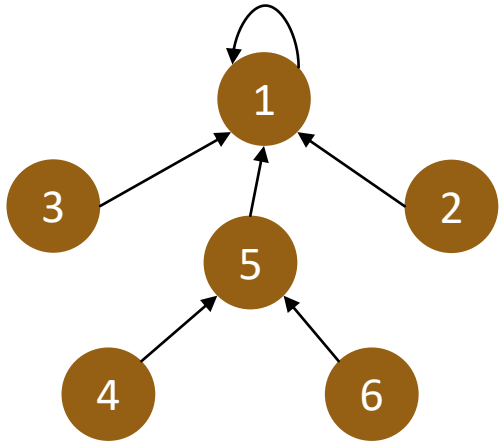
Shiloach/Vishkin

Shiloach/Vishkin's algorithm

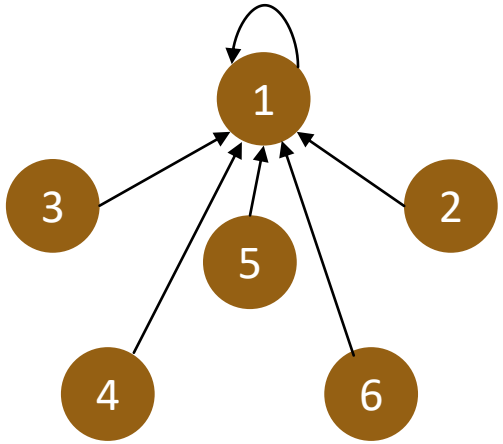
- **Pointer graph/forest:**
  - Define pointer array P, $P[i]$ is a pointer from i to some other vertex
  - We call the graph defined by P (excluding self loops) the pointer graph
  - During the algorithm, $P[i]$ forms a forest such that $\forall i$: $(i, P[i])$ there exists a path from i to $P[i]$ in the original graph!
  - Initially, all $P[i] = i$
  - The algorithm will run until each forest is a directed star pointing at the (smallest-id) root of the component
- **Supervertices:**
  - Initially, each vertex is its own supervertex
  - Supervertices induce a graph - S_i and S_j are connected iff $\exists (u, v) \in E$ with $u \in S_i$ and $v \in S_j$
  - A supervertex is represented by its tree in P



graph with single component



possible forest formed by P

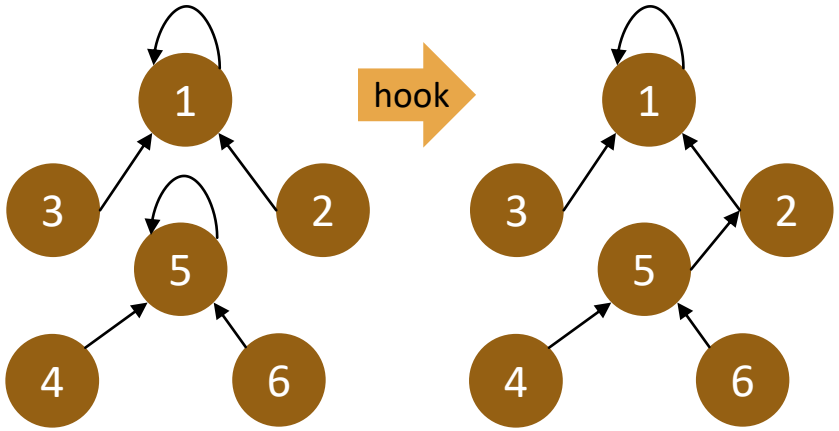


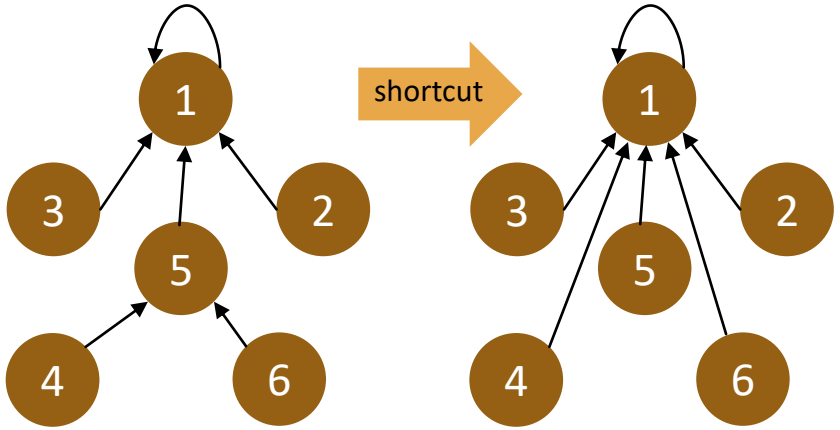
star formed by P

Shiloach/Vishkin's algorithm – key components

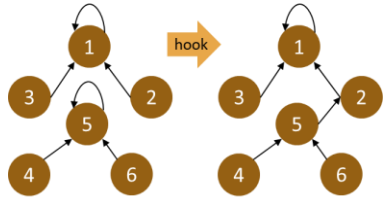
- **Algorithm proceeds in two operations:**
  - Hook – merge connected supervertices (must be careful to not introduce cycles!)
  - Shortcut – turn trees into stars

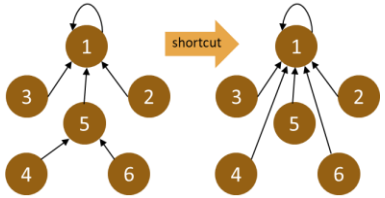
  Repeat two steps iteratively until fixpoint is reached!





Shiloach/Vishkin's algorithm – Proof





- **Correctness proofs:**
  - Lemma 1: The shortcut operation converts rooted trees to rooted stars. Proof: obvious
  - Theorem 1: The pointer graph always forms a forest (set of rooted trees). Proof: shortcut doesn't violate, hook works on rooted stars, connects only to smaller label star, no cycles

- **Performance proofs:**
  - Lemma 2: The number of iterations of the outer loop is at most $\log_2 n$. Proof: consider connected component, if it has two supervertices before hook, number of supervertices is halved, if no hooking happens, component is done
  - Lemma 2: The number of iterations of the inner loop in shortcut is at most $\log_2 n$. Proof: consider tree of height > 2 at some iteration, the height of the tree halves during that iteration
  - Corollary: Class question: work and depth? $W = O(n^2 \log n), D = O(\log^2 n)$ (assuming conflicts are free!)

- **Algorithm was recently improved for practical architectures**
  - M. Sutton, TBN, A. Barak, "Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling", IPDPS'18
  - Introducing CAS, random sampling