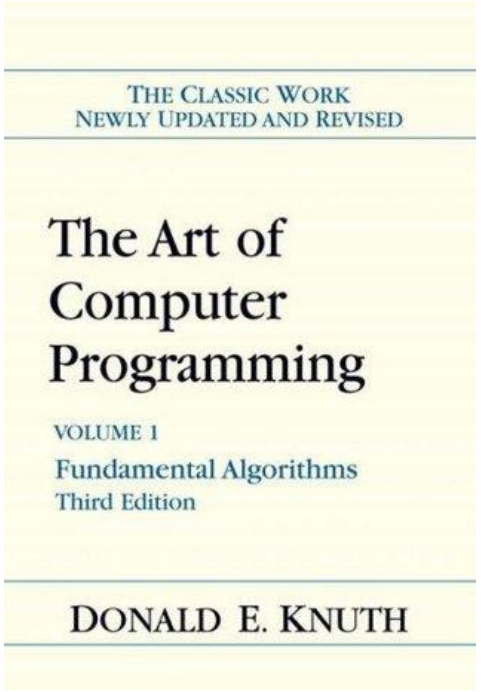
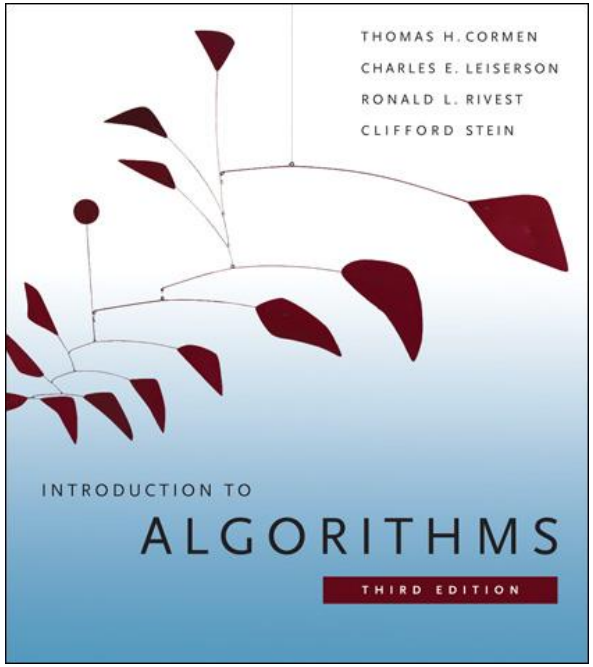
Write-efficient algorithms

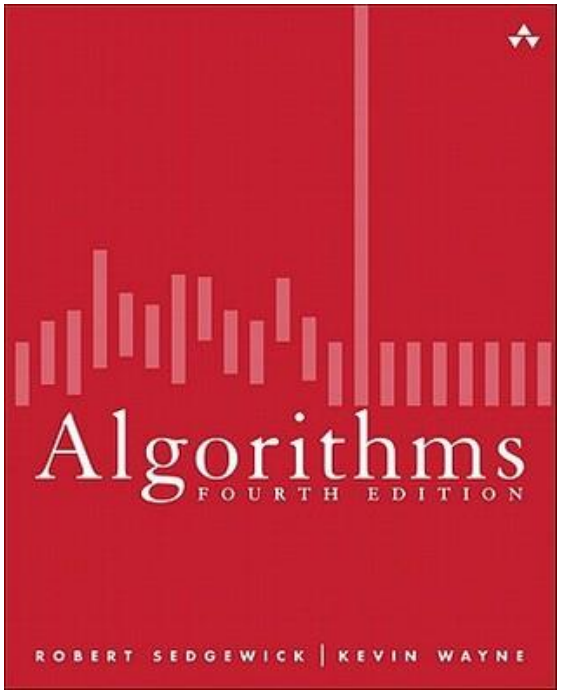
15-853: Algorithms in the Real World

Yan Gu, May 2, 2018

Classic Algorithms Research







…has focused on settings in which reads & writes to memory have equal cost

But what if they have very DIFFERENT costs? How would that impact Algorithm Design?

HP, SanDisk partner on memristor, ReRAM technology

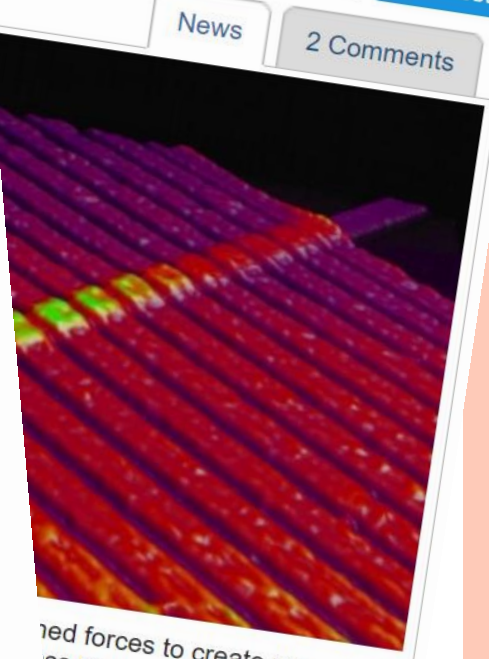
Published on 9th October 2015 by Gareth Halfacree

Hewlett Packard and SanDisk have announced a long-term partnership to develop a new technology for Storage Class Memory (SCM) combining the best of memristor and ReRAM.

The memory ind... home...

Like 2 Tweet

News 2 Comments



...ed forces to create memristor-...ss memory (SCM) products

...are combining their efforts ...ory (SCM) market.

Menu Communities Find Content Search Intel.com

Intel and Micron Produce Breakthrough Memory Technology

Posted by IntelPR in Intel Newsroom on Jul 28, 2015 9:00:28 AM

Tweet 1,203 Like 5.5k g+1 511

New Class of Memory Unleashes the Performance of PCs, Data Centers and More

NEWS HIGHLIGHTS

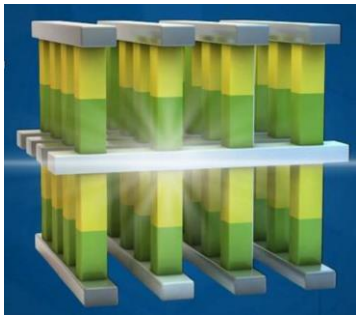
- Intel and Micron begin production on new class of non-volatile memory, creating the first new memory category in more than 25 years.
- New 3D XPoint™ technology brings non-volatile memory speeds up to 1,000 times faster[1] than NAND, the most popular non-volatile memory in the marketplace today.
- The companies invented unique material compounds and a cross point architecture for a memory technology that is 10 times denser than conventional memory[2].
- New technology makes new innovations possible in applications ranging from machine learning to real-time tracking of diseases and immersive 8K gaming.

SANTA CLARA, Calif., and BOISE, Idaho, July 28, 2015 – Intel Corporation and Micron Technology, Inc. today unveiled 3D XPoint™ technology, a non-volatile memory that has the potential to revolutionize any device, application or service that benefits from fast access to large sets of data. Now in production, 3D XPoint technology is a major breakthrough in memory process technology and

Emerging Memory Technologies

Motivation:

- DRAM is volatile
- DRAM energy cost is significant (~35% energy on data centers)
- DRAM density (bits/area) is limited



3D XPoint

Promising candidates:

- **Phase-Change Memory (PCM)**
- **Spin-Torque Transfer Magnetic RAM (STT-RAM)**
- **Memristor-based Resistive RAM (ReRAM)**
- **Conductive-bridging RAM (CBRAM)**

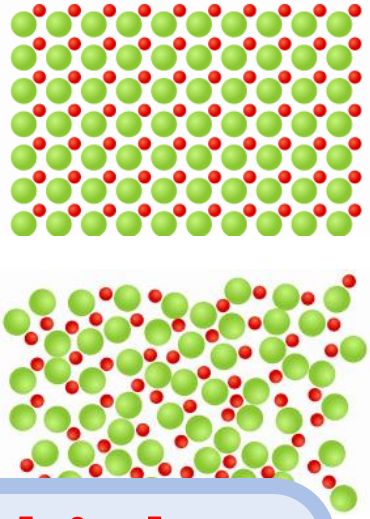
Key properties:

- **Persistent**, significantly lower energy, can be higher density (10x+)
- Read latencies approaching DRAM, random-access

Another Key Property:
Writes More Costly than Reads

In these emerging memory technologies, bits are stored as “states” of the given material

- No energy to retain state
- Small energy to read state
  - Low current for short duration
- Large energy to change state



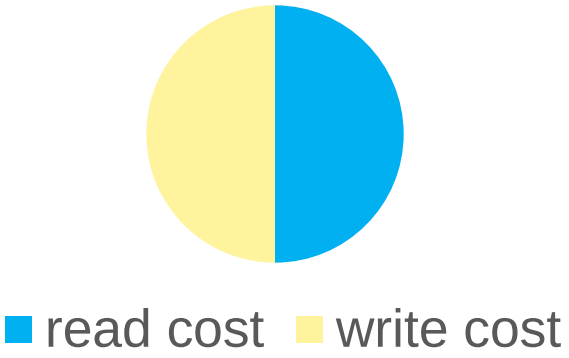
Writes incur higher energy costs, higher latency, and lower per-DIMM bandwidth (power envelope constraints)

Why does it matter?

- Consider the energy issue and assume a read costs 0.1 nJ and a write costs 10 nJ

Sorting algorithm 1:
$100n$ reads and $100n$ writes on n elements
We can sort <1 million entries per joule

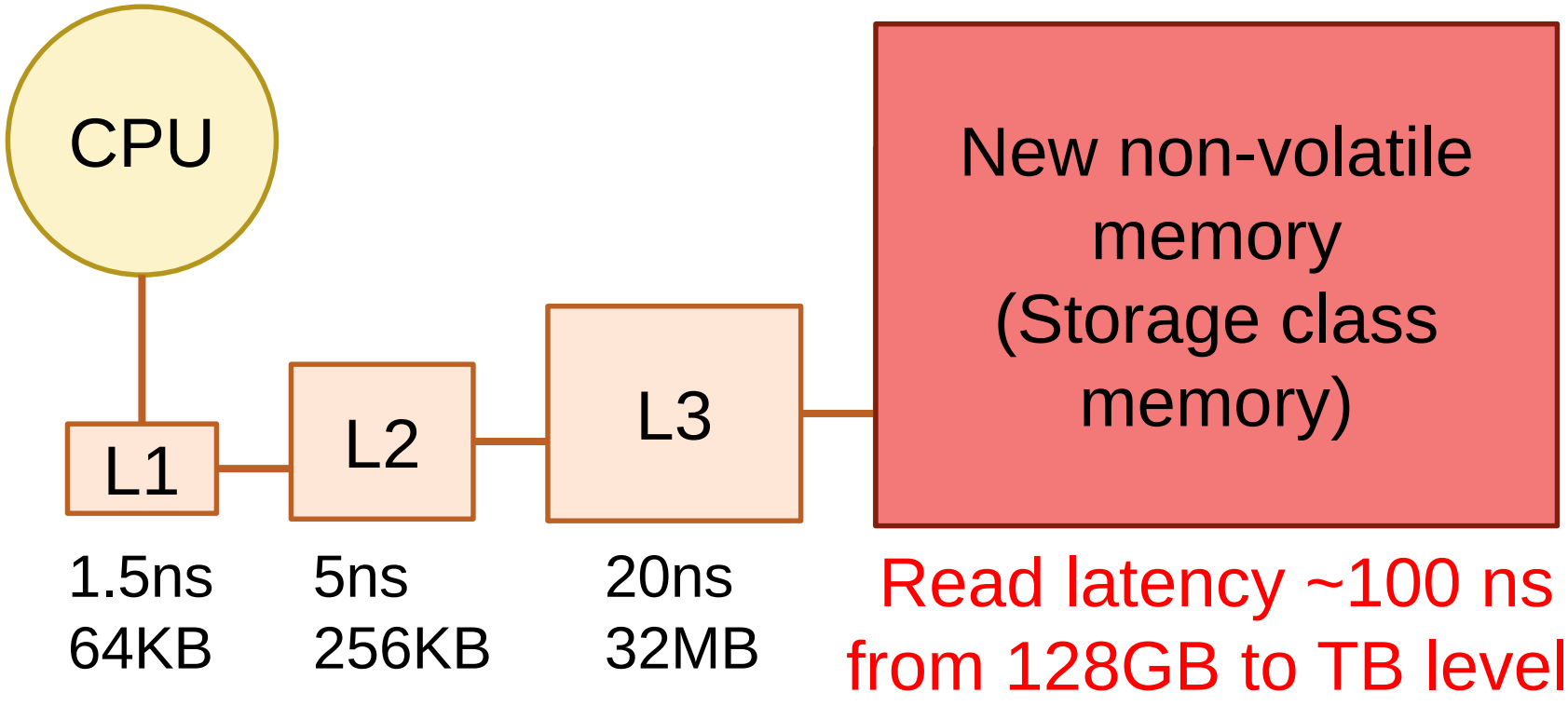
Sorting algorithm 2:
$200n$ reads and $2n$ writes on n elements
We can sort 25 million entries per joule



Why does it matter?

- Writes are significantly more costly than reads due to the cost to change the phases of materials
  - higher latency, lower per-chip bandwidth, higher energy costs
- Higher latency → Longer time for a write → Decrease per-chip (memory) bandwidth
- **Let the parameter $\omega > 1$ be the cost for writes relative to reads**
  - Expected to be between 5 to 30

Evolution on the memory hierarchy



Common latency / size for a current computer (server)

Impacts on Real-World Computation

- Databases: the data that is kept in the external memory can now be on the **main memory**
- Graph processing: large social networks nowadays contain ~billion vertices and >100 billion edges
- Geometry applications: can handle more precise meshes that support better effects

Summary

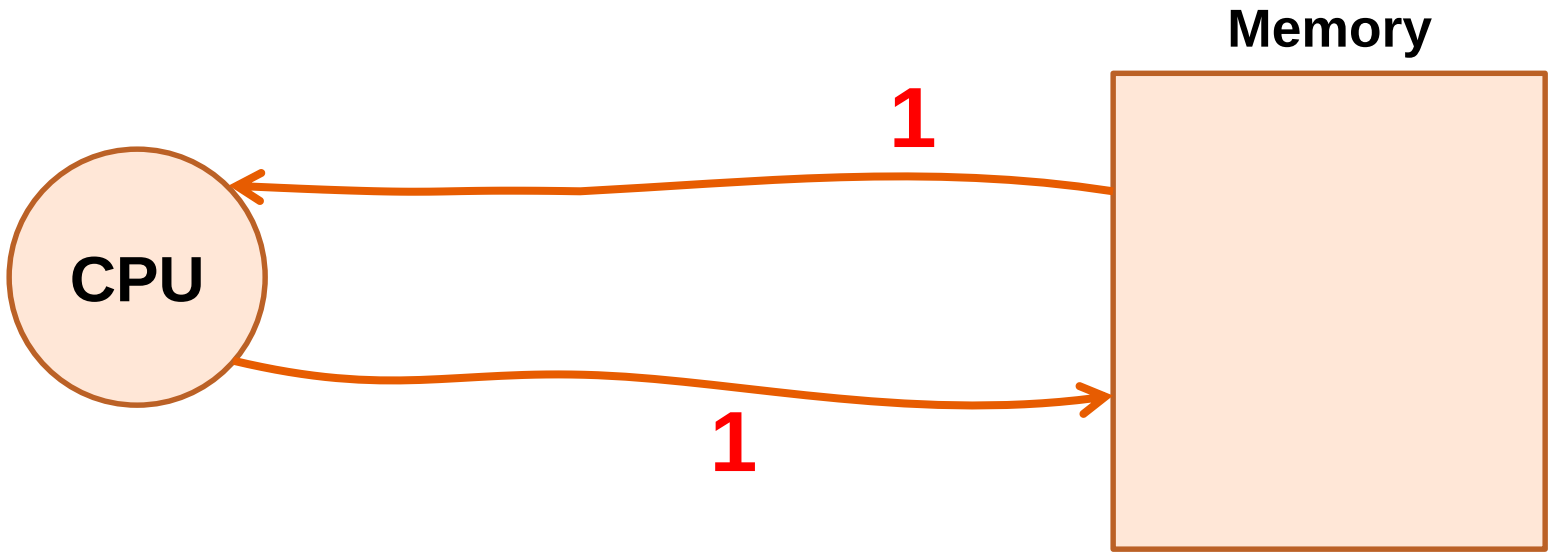
- The new non-volatile memory raises the challenge to design write-efficient algorithms

- What we need:
  - Modified cost models
  - New algorithms
  - New techniques to support efficient computation (cache policy, scheduling, etc.)
  - Experiment

New Cost Models

Random-Access Machine (RAM)

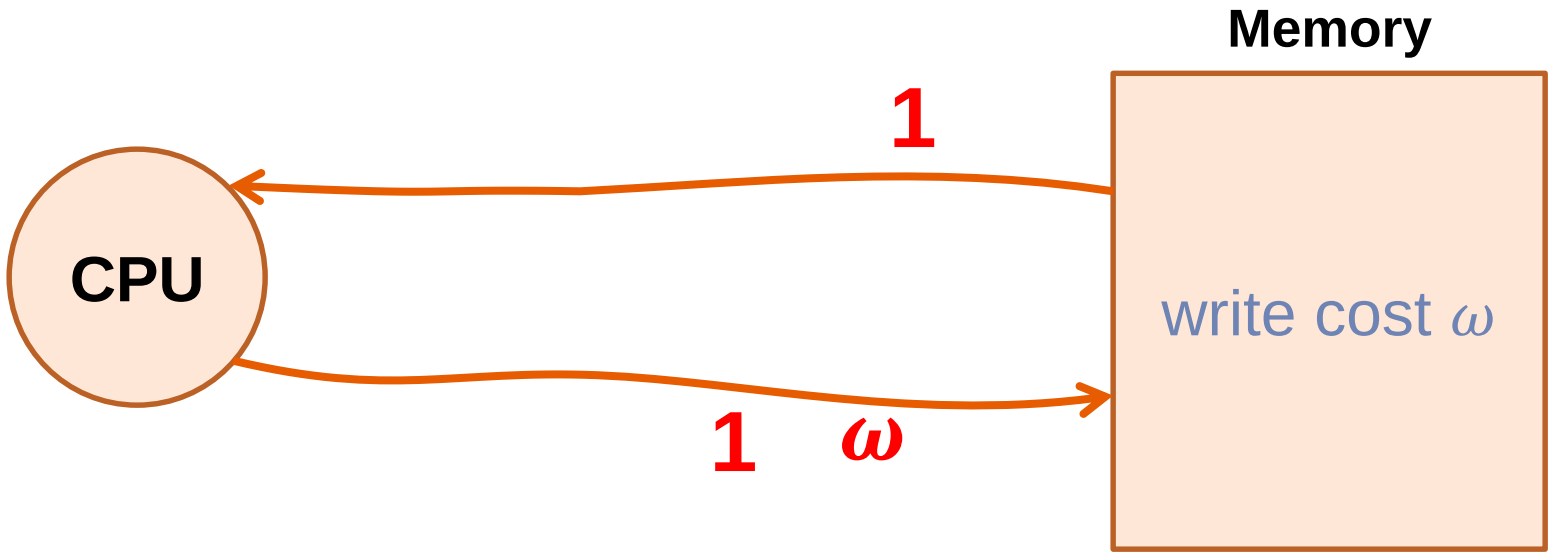
- Unit cost for:
  - Any instruction on $\Theta(\log n)$-bit words
  - Read/write a single memory location from an infinite memory



Read/write asymmetry in RAM?

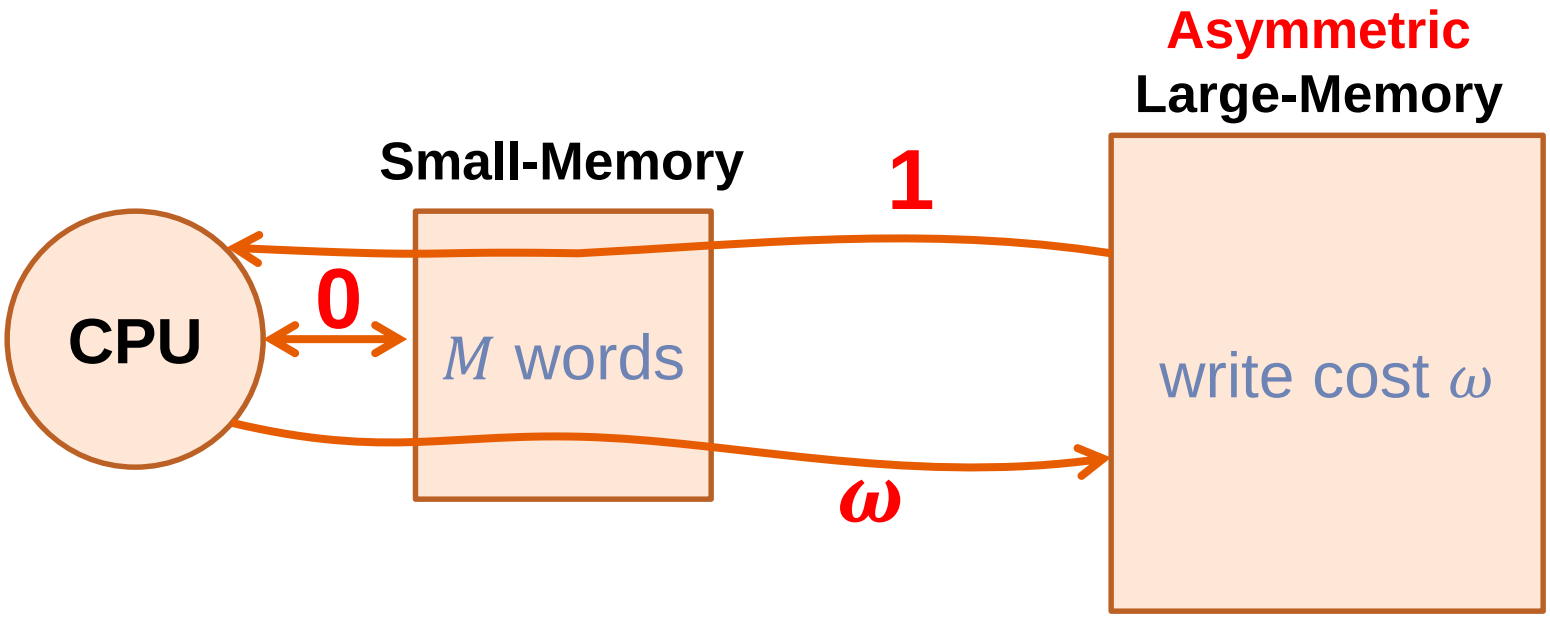
- A single write cost ω instead of 1

But every instruction writes something…



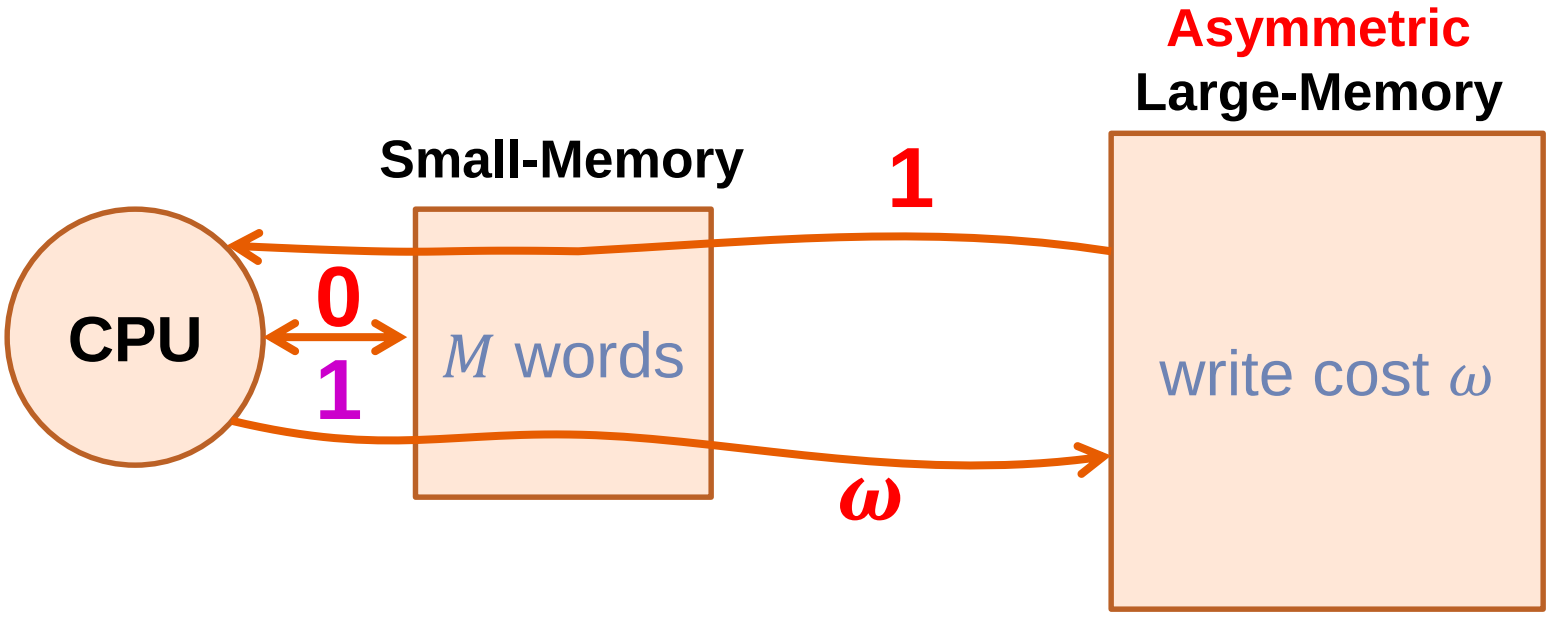
(M, ω)-Asymmetric RAM (ARAM)

- Comprise of:
  - a symmetric small-memory (cache) of size M, and
  - an asymmetric large-memory (main memory) of unbounded size, and an integer write cost ω
- I/O cost Q: instructions on cache are free



(M, ω)-Asymmetric RAM (ARAM)

- Comprise of:
  - a symmetric small-memory (cache) of size M, and
  - an asymmetric large-memory (main memory) of unbounded size, and an integer write cost ω
- I/O cost Q: instructions on cache are free
- time T: instructions on cache take $\mathbf{1}$ unit of time



Lower and Upper Bounds

Warm up: **Asymmetric** sorting

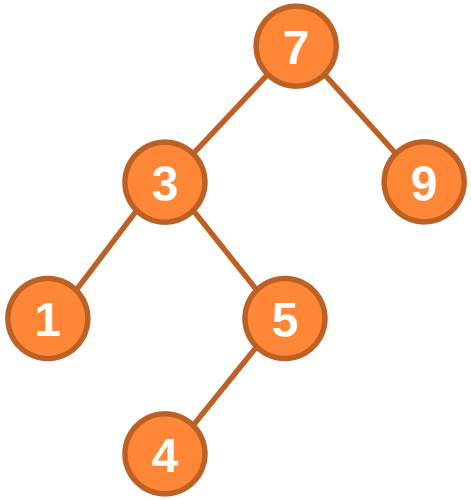
- Comparison sort on n elements
- Read and comparison (without writes): $\Omega(n \log n)$
- Write complexity: $\Omega(n)$

- Question: how to sort n elements using $O(n \log n)$ instructions (reads) and $O(n)$ writes?
  - Swap-based sorting (i.e. quicksort, heap sort) does not seem to work
  - Mergesort requires strictly n writes for $\log n$ rounds
  - Selection sort uses linear write, but not work (read) efficiency

Warm up: **Asymmetric** sorting

- Comparison sort on n elements
- Read complexity: $\Omega(n \log n)$
- Write complexity: $\Omega(n)$

- The algorithm: inserting each key in random order into a binary search tree. In-order traversing the tree gives the sorted array. ($O(\log n)$ tree depth w.h.p.)



- Using balanced BSTs (e.g. AVL trees) gives a deterministic algorithm, but more careful analysis is required

Trivial upper bounds

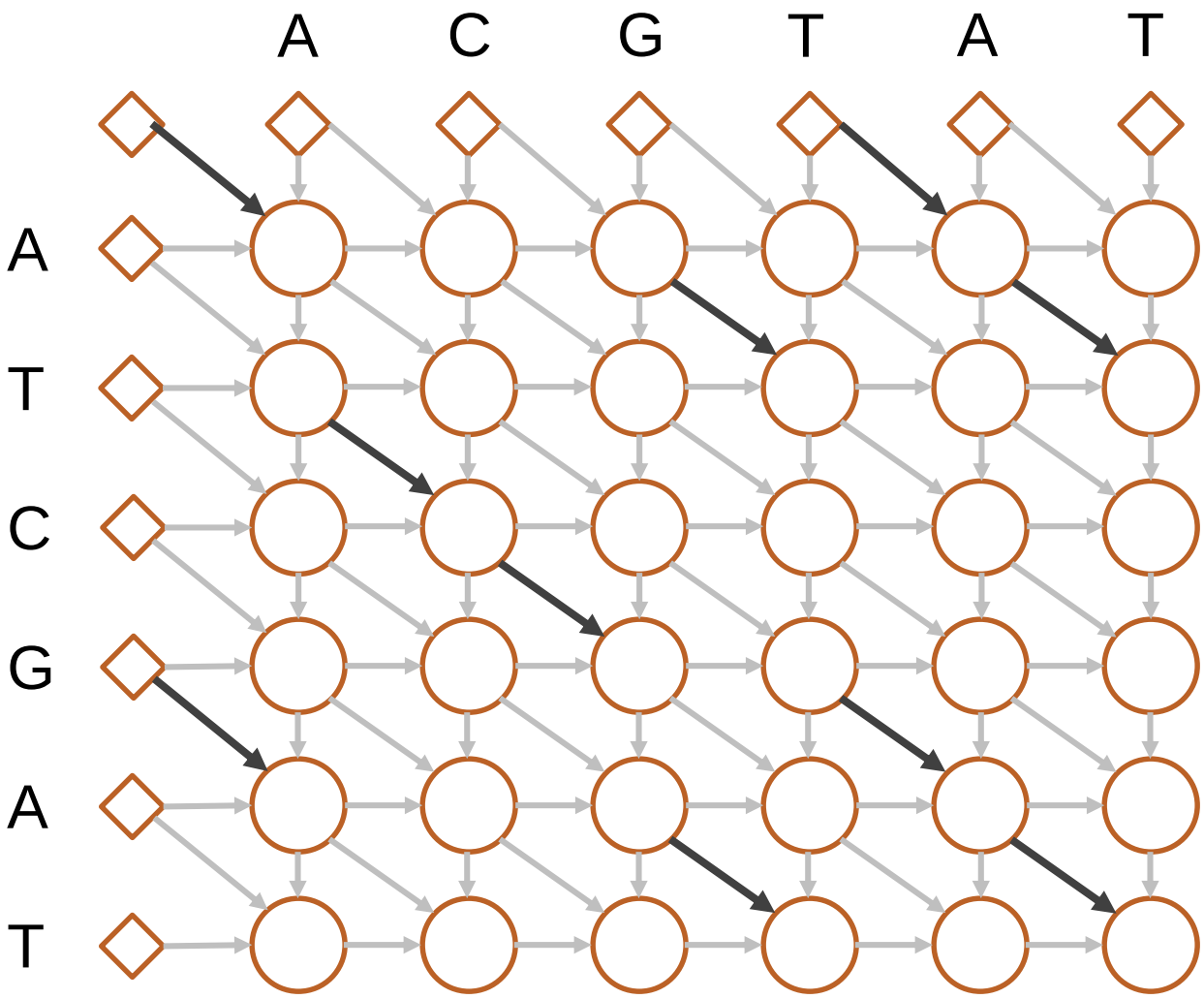
$M = O(1)$

Problem	I/O cost $Q(n)$ and time $T(n)$	Reduction ratio
Comparison sort	$\Theta(n \log n + \omega n)$	$O(\log n)$
Search tree, priority queue	$\Theta(\log n + \omega)$	$O(\log n)$
2D convex hull, triangulation	$O(n \log n + \omega n)$	$O(\log n)$
BFS, DFS, SCC, topological sort, block, bipartiteness, floodfill, biconnected components	$\Theta(m + n\omega)$	$O(m/n)$

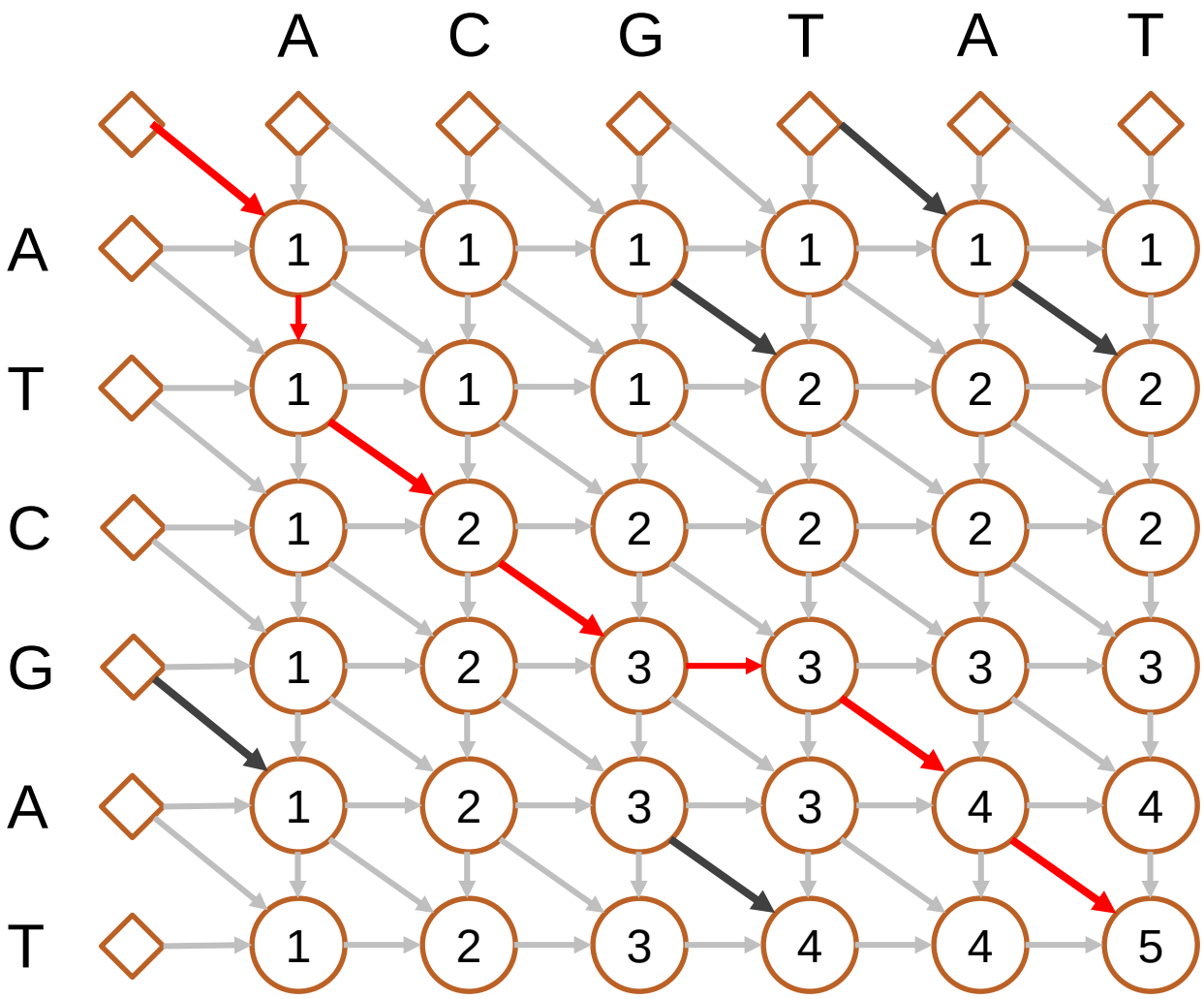
Lower bounds

Problem	I/O cost	
	Classic Algorithm	Lower bound
Sorting network	$\Theta\left(\omega n \frac{\log n}{\log M}\right)$	$\Theta\left(\omega n \frac{\log n}{\log \omega M}\right)$
Fast Fourier Transform	$\Theta\left(\omega n \frac{\log n}{\log M}\right)$	$\Theta\left(\omega n \frac{\log n}{\log \omega M}\right)$
Diamond DAG (LCS, edit distance)	$\Theta\left(\frac{n^2 \omega}{M}\right)$	$\Theta\left(\frac{n^2 \omega}{M}\right)$

An example of a diamond DAG: Longest common sequence (LCS)



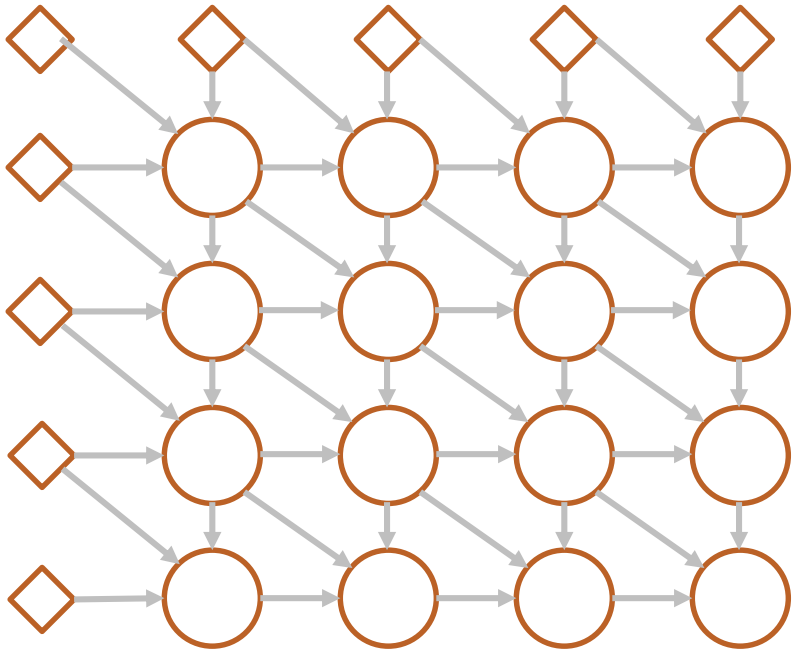
An example of Diamond DAG: Longest common sequence (LCS)



Computation DAG Rule

DAG Rule / pebbling game:

- To compute the value of a node, must have the values at all incoming nodes

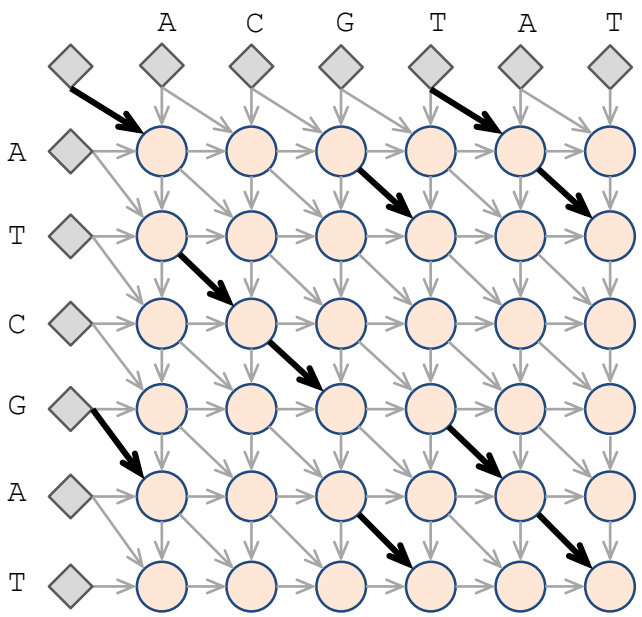


High-level proof idea

- To show that for the computational DAG, there exists a partitioning of the DAG that I/O cost is lower bounded

- However, since read and write has different cost, previous techniques (e.g. [HK81]) cannot directly apply

Standard Observations on DAG



DAG (or DP table) has size n^2

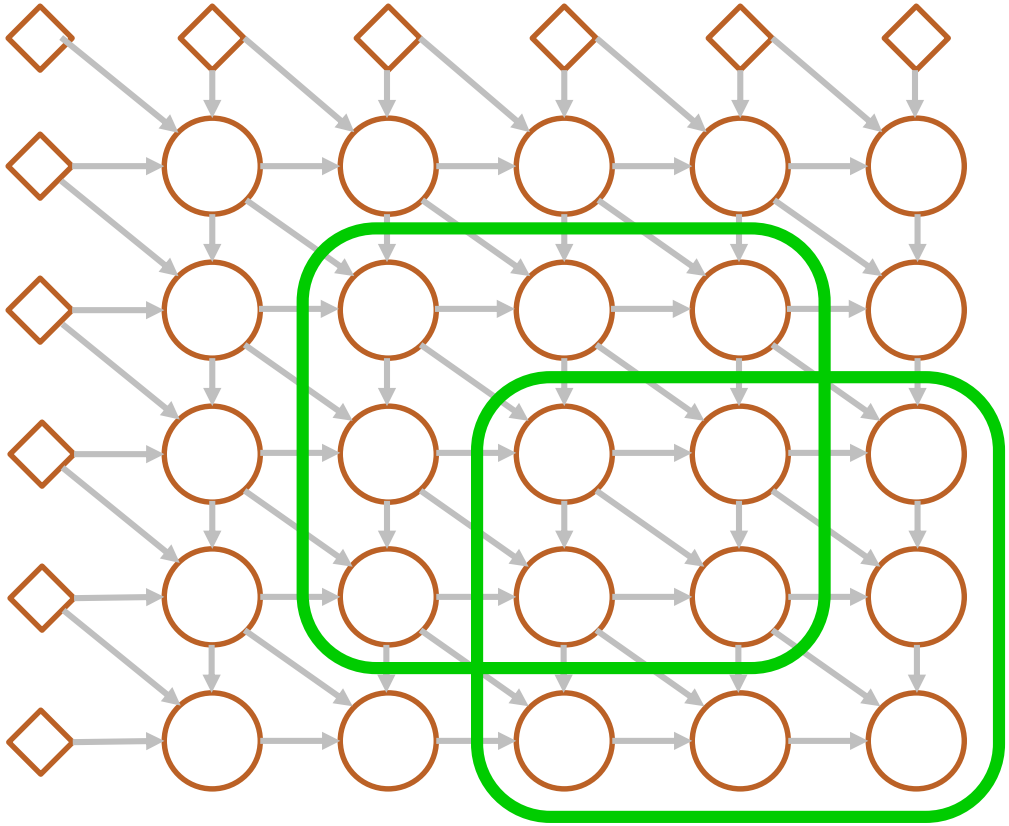
- (Input size is only $2n$)
- Building table explicitly ⇒ n^2 writes,
- but problem only inherently requires writing last value

- Compute some nodes in cache but don't write them out

Storage lower bound of subcomputation

(diamond DAG rule, Cook and Sethi 1976):

Solving an $k \times k$ sub-DAG requires k space to store intermediate value

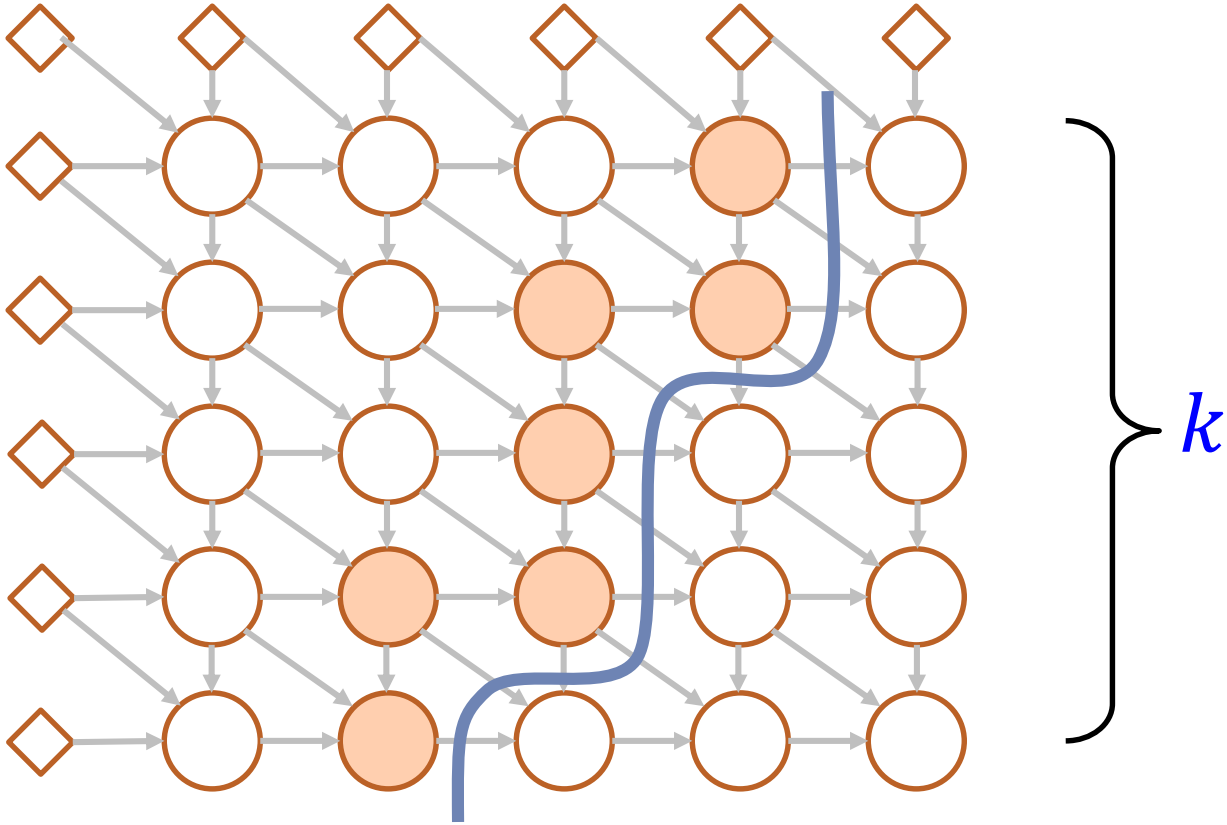


For $k > M$, some values need to be written out

Storage lower bound of subcomputation

(diamond DAG rule, Cook and Sethi 1976):

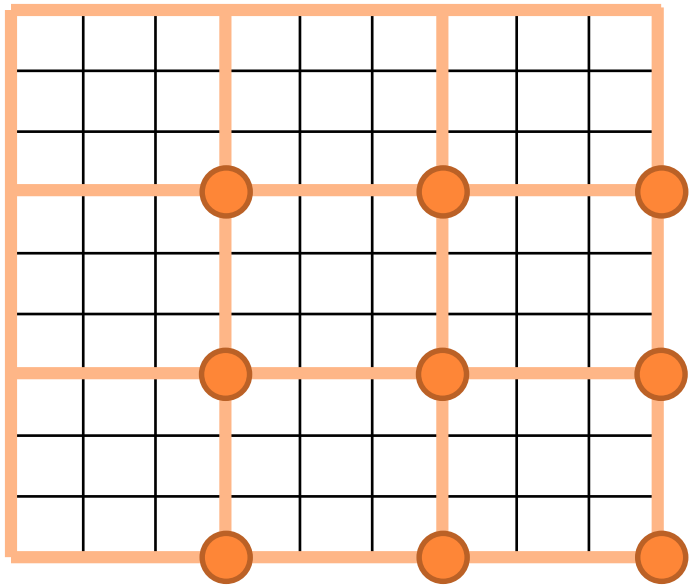
Solving an $k \times k$ sub-DAG requires k space to store intermediate value



For $k > M$, some values need to be written out

Proof sketch of lower bound

- Computing any $2M \times 2M$ diamond requires M writes to the large-memory
  - $2M$ storage space, M from small-memory
- To finish computation, every $2M \times 2M$ sub-DAG needs to be computed, which leads to $\Omega\left(\frac{n^2}{M}\right)$ writes

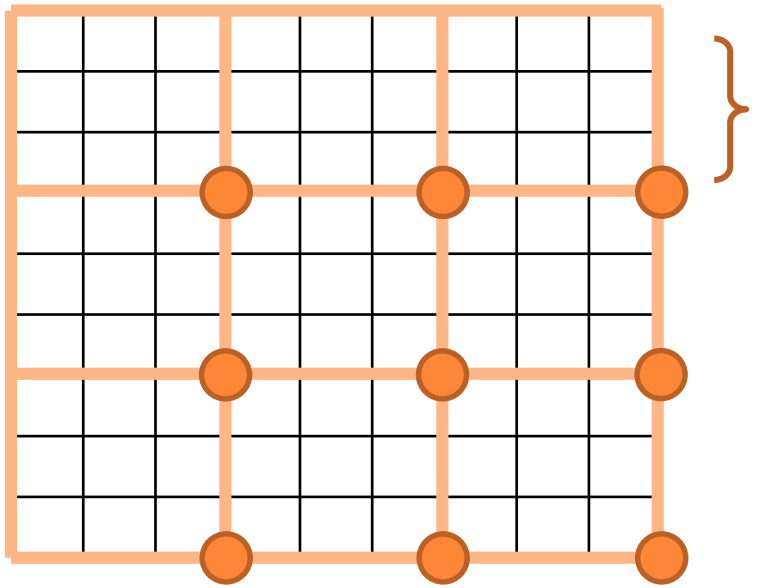


$$\Theta\left(\frac{n^2}{M^2}\right) \cdot \Omega(M) = \Omega\left(\frac{n^2}{M}\right)$$

#sub-problem #write

A matching algorithm for the lower bound

- Lower bound: $\Theta\left(\frac{n^2}{M}\right)$ writes
- This lower bound is tight when breaking down into $\frac{M}{2} \times \frac{M}{2}$ sub-DAGs, and read & write-out the boundary only



$$2 \cdot \frac{n}{M/2} \cdot n = O\left(\frac{n^2}{M}\right)$$

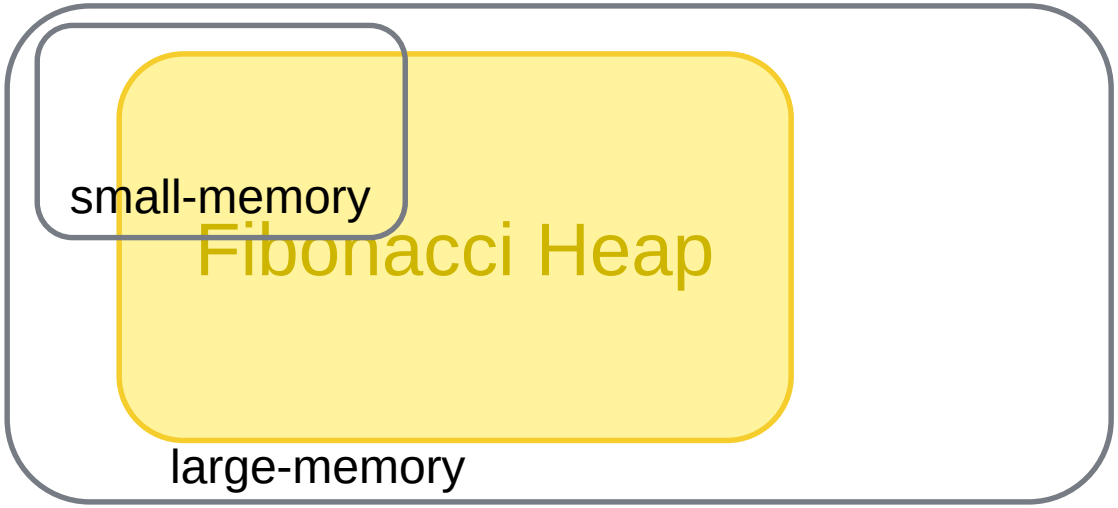
#row/column #read/write

Upper bounds on graph algorithms

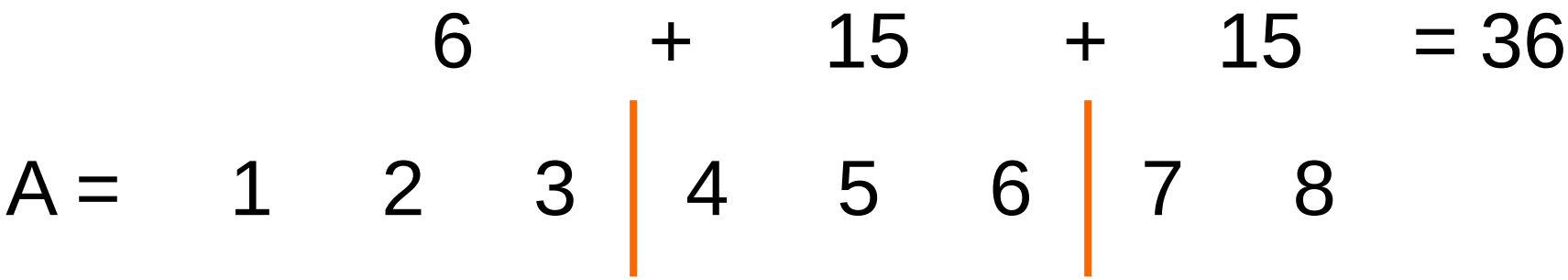
Problem	I/O cost $Q(n, m)$	
	Classic algorithms	New algorithms
Single-source shortest-path	$O\big(\omega(m + n\log n)\big)$	$O(\min(n(\omega + m/M),$ $\omega(m + n\log n),$ $m(\omega + \log n)))$
Minimum spanning tree	$O(m\omega)$	$O(\min(m\omega,$ $m\min(\log n, n/M) + \omega n))$

I/O cost of Dijkstra's algorithm

- Compute an SSSP requires $O(m)$ DECREASE-KEYs and $O(n)$ EXTRACT-MINs in Dijkstra's algorithm
  - Classic Fibonacci heap: $O\big(\omega(m + n\log n)\big)$
  - Balanced BST: $O\big(m(\omega + \log n)\big)$
  - Restrict the Fibonacci heap into the small-memory with size M: no writes to the large-memory to maintain the heap, $O(n/M)$ rounds to finish, $O\big(n(\omega + m/M)\big)$ I/O cost in total

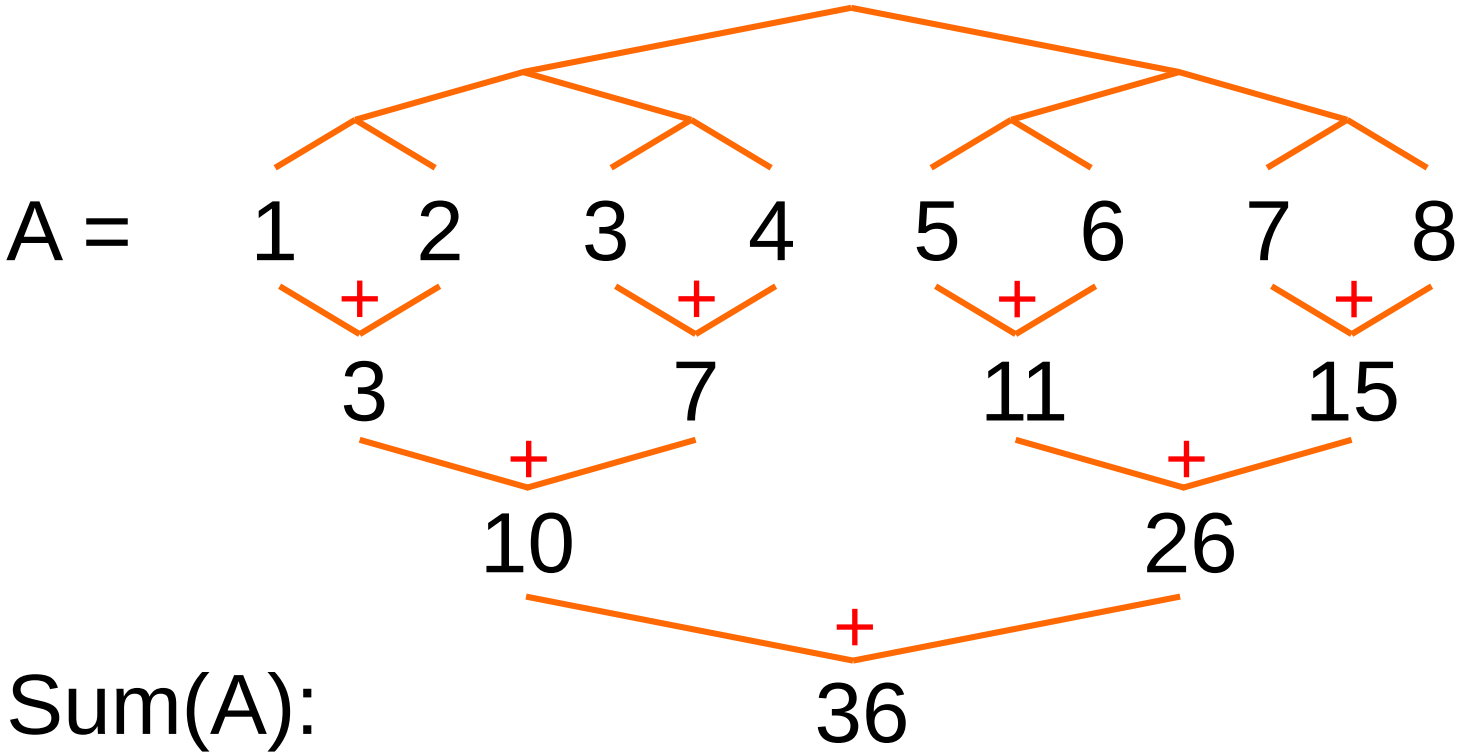


Parallel Computational Model and Parallel Write-efficient Algorithms

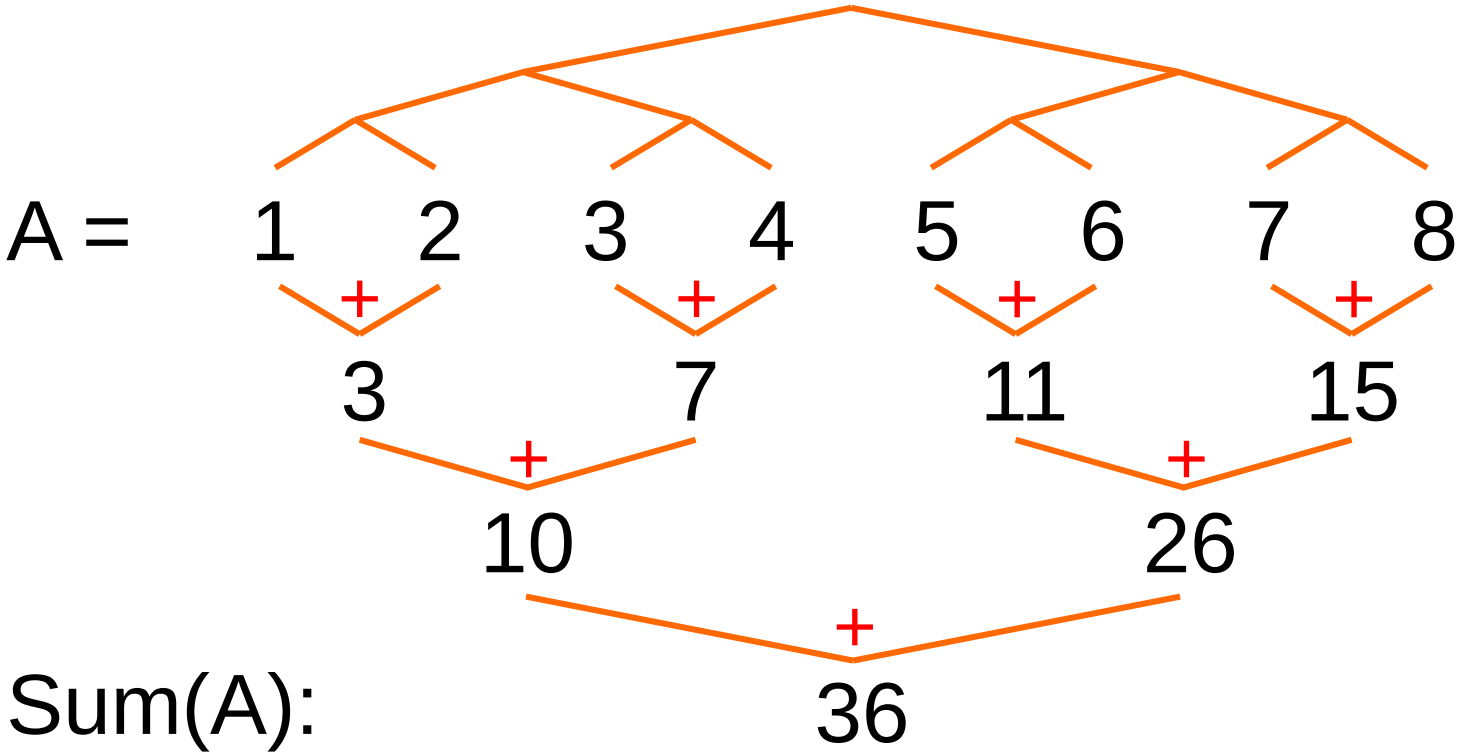


Sum(A): 36

- Cut the input array into smaller segments, sum each up individually, and finally sum up the sums
- Picking the appropriate number of segments can be annoying
  - Machine parameter, runtime environment, algorithmic details



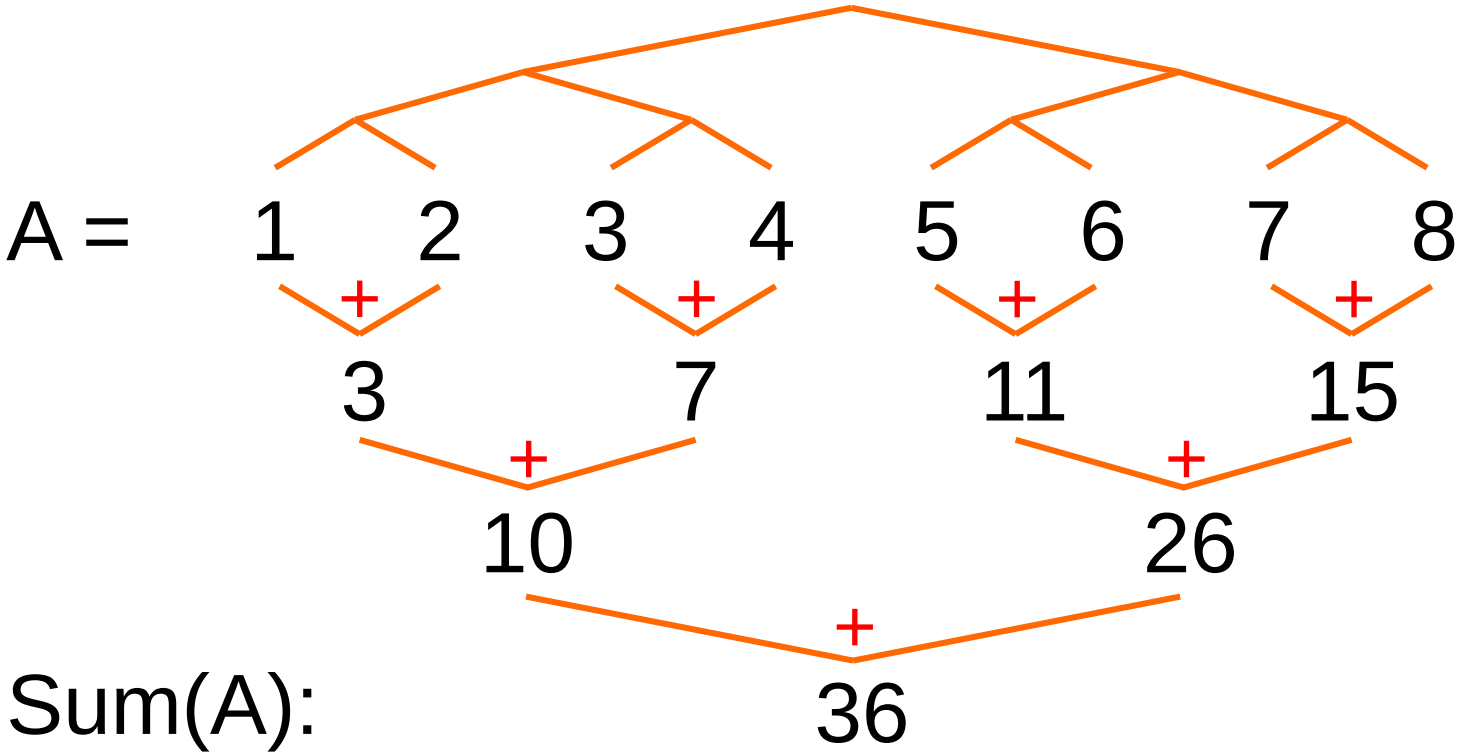

```
Function SUM(A)
        If  |A| = 1  then return  A(1)
        In Parallel
                a = SUM(first half of A)
                b = SUM(second half of A)
        return a + b
```



A work-stealing scheduler can run a computation on p cores using time:

$$O\left(\frac{n}{p} + \log n\right)$$

The work-stealing scheduler is used in OpenMP, CilkPlus, Intel TBB, MS PPL, etc.

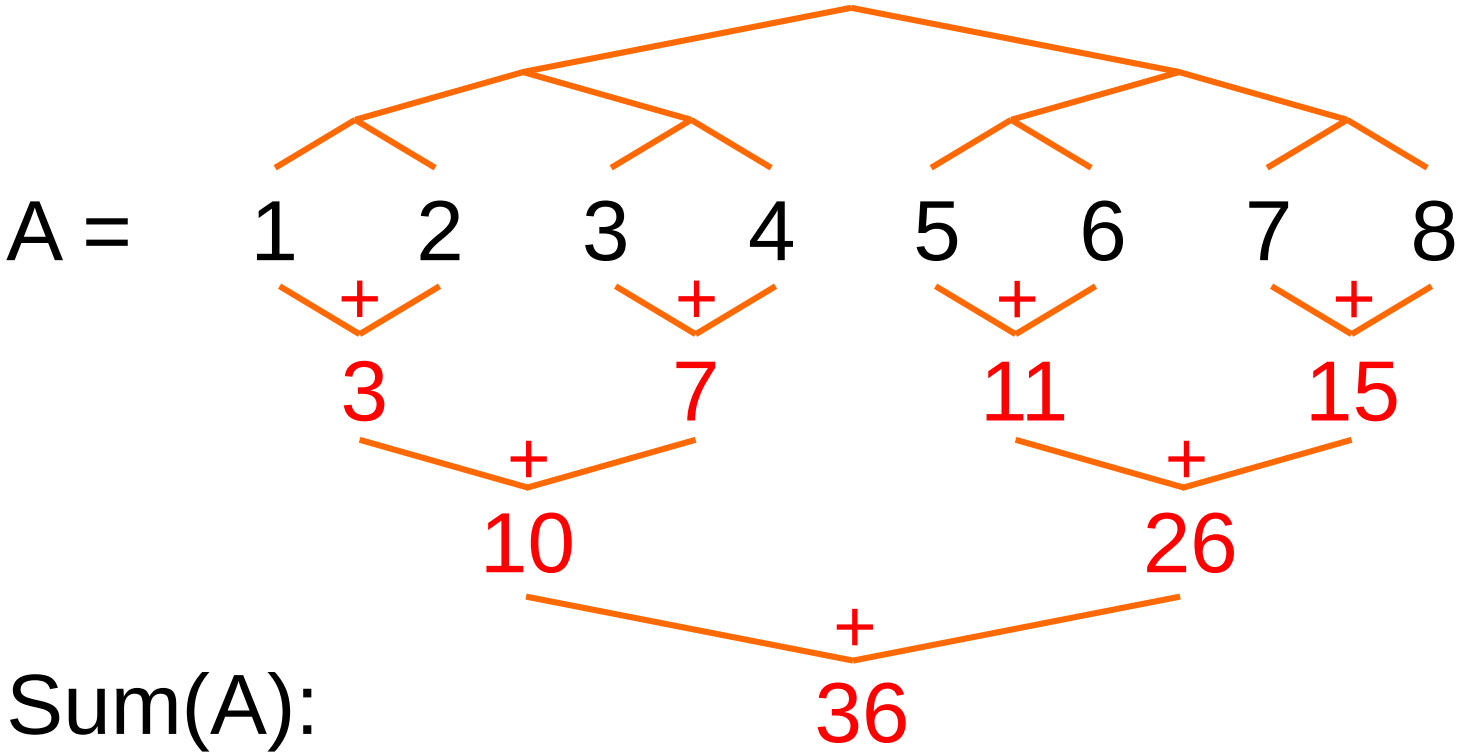


A work-stealing scheduler can run a computation on p cores using time:

$$\frac{W}{p} + O(D)$$

W: total computation

D: longest chain of all paths



Sum(A):

A work-stealing scheduler can run a computation on p cores using time:

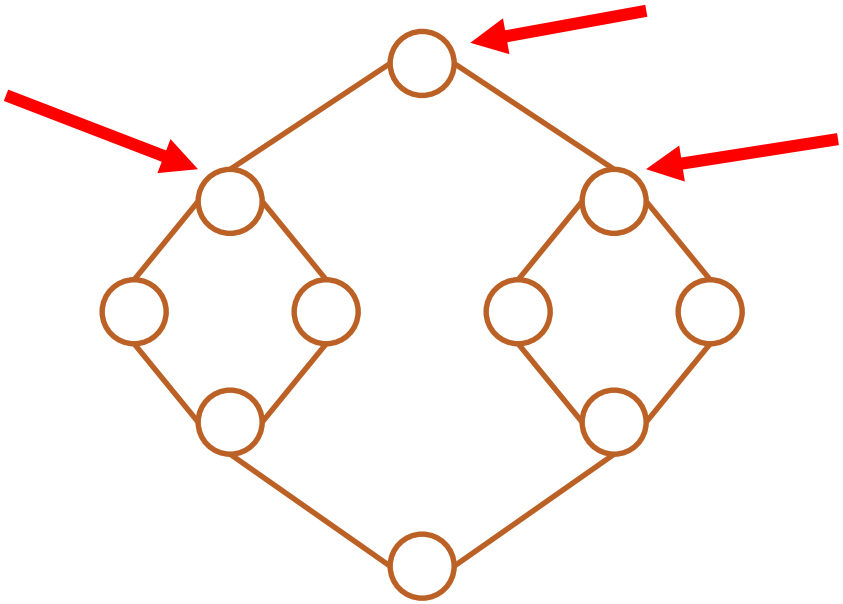
$$O\left(\omega\left(\frac{W}{P}+D\right)\right)$$

W: $O(n)$

D: $O(\log n)$

Our new results on scheduler

- Assumptions:
  - Each processor has its symmetric private cache
  - Each processor can request to access data in other processor's cache, but cannot write anything
  - All communications are via asymmetric main memory
  - Any non-leaf task uses constant stack space



Our new results on scheduler

- Assumptions:
  - Each processor has its symmetric private cache
  - Each processor can request to access data in other processor's cache, but cannot write anything
  - All communications are via asymmetric main memory
  - Any non-leaf task uses constant stack space

With non-trivial but simple modifications to the work-stealing scheduler, a computation on p cores use time:

$$\frac{W}{p} + O(\omega D)$$

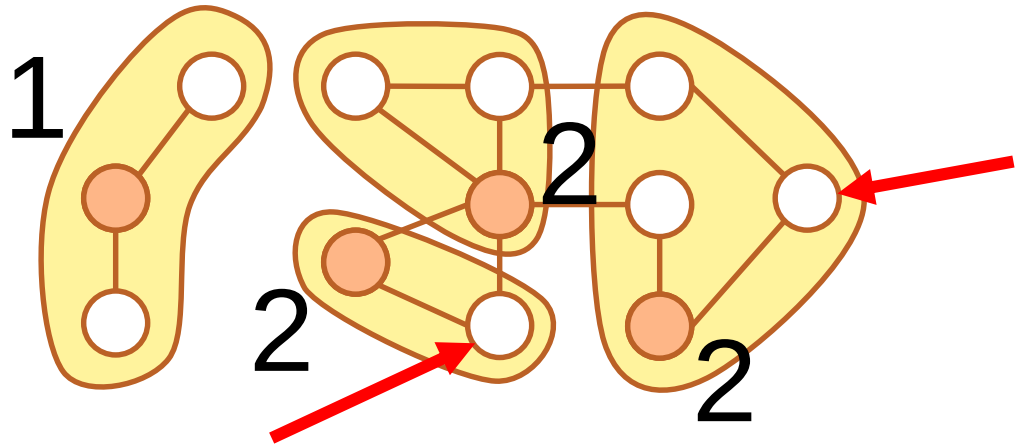
with memory size $M = O(D) + M_L$ (memory for base case)

Results of parallel algorithms

Problem	Work (W)	Depth (D)	Reduction of writes
Reduce	$\Theta(n+\omega)$	$\Theta(\log n+\omega)$	$\Theta(\log n)$
Ordered filter	$\Theta(n+\omega k)$ ↟	$O(\omega \log n)$ ↟	$\Theta(\log n)$
Comparison sort	$\Theta(n \log n+n\omega)$ ↟	$O(\omega \log n)$ ↟	$\Theta(\log n)$
List and tree contraction	$\Theta(n)$	$O(\omega \log n)$ ↟	$\Theta(\omega)$
Minimum spanning tree	$O(\alpha(n)m$ $+\omega n \log(\min(m/n,\omega)))$	$O(\omega \operatorname{polylog} n)$↟	$m/(n \cdot$ $\log(\min(m/n,\omega)))$
2D convex hull	$O(n \log k+\omega n \log\log k)$ §	$O(\omega \log^2 n)$ ↟	output-sensitive
BFS tree	$\Theta(\omega n+m)$ §	$\Theta(\omega\delta \log n)$ ↟	$O(m/n)$

k = output size
δ = graph diameter
↟ = with high probability
§ = expected

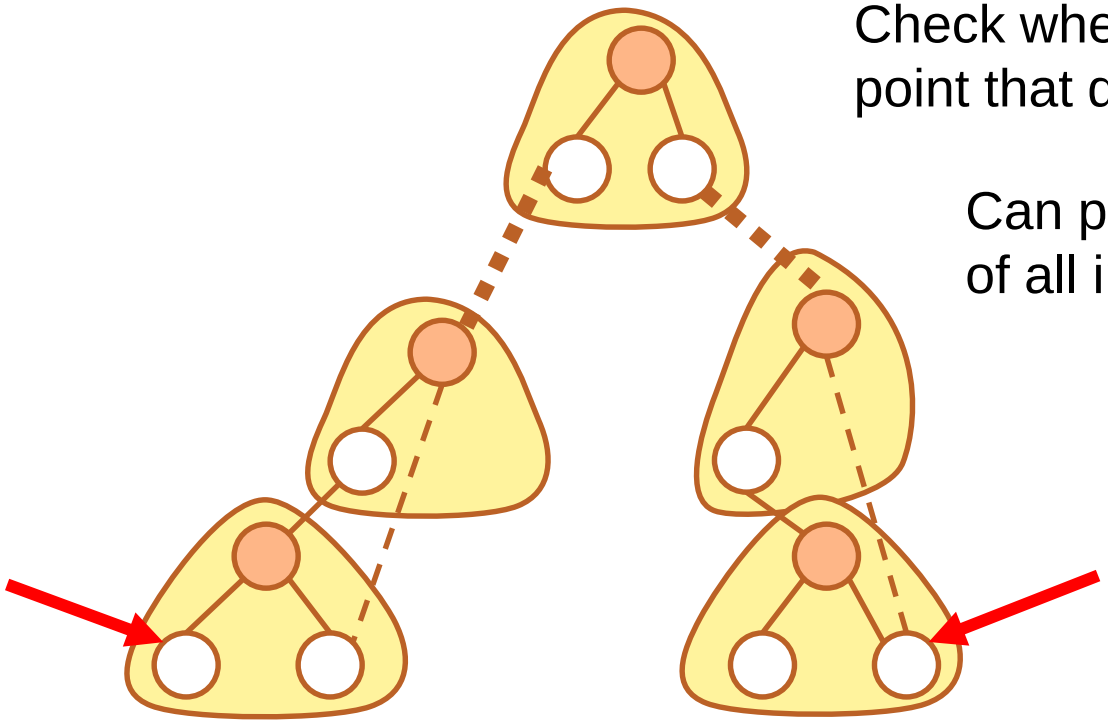
Graph Connectivity (Biconnectivity)



- Partition the graph into $\Theta(n/k)$ connected clusters, each with size no more than k
- Only store the connectivity information for the "center" of each cluster ($\Theta(n/k)$ size in total)
- Require $O(k^2)$ cost to retrieve a cluster
- The preprocessing uses $O(k^2 \cdot n/k) = O(nk)$ cost
- Each query needs $O(k^2)$ cost (check two clusters)

Graph Biconnectivity

- Our claim: answering a biconnectivity query only requires the information on the clusters graph and within at most 3 clusters



Check whether there is any articulation point that disconnects them

Can precompute and store results of all intermediate clusters

Need to specially check 3 clusters

Graph Connectivity / Biconnectivity

Problem	Work (W)	Depth (D)	Query
Undirected Connectivity, Biconnectivity	$\Theta(\min(m + \omega n, \sqrt{\omega} m))$	$\Theta\left(\text{poly}(\omega, \log n)\right)$	$\Theta(\omega)$

Our new approach (the implicit decomposition) can reduce the space utilization (and writes) by a factor of $\sqrt{\omega}$

We show a graph decomposition that uses sublinear space to represent, which can be a useful tool

Sequential upper bounds

$M = O(1)$

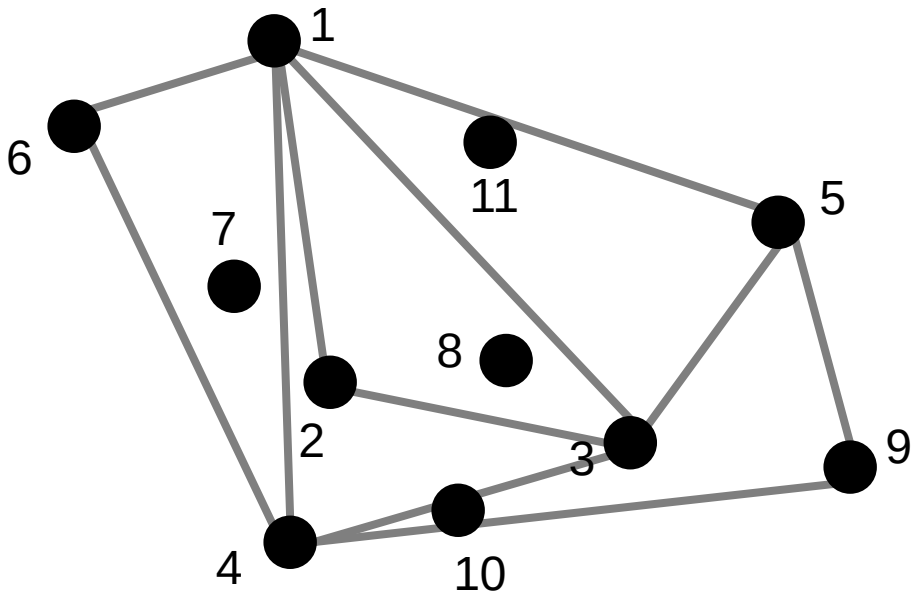
Problem	I/O cost $Q(n)$ and work $W(n)$	Reduction ratio
Comparison sort	$\Theta(n \log n + \omega n)$	$O(\log n)$
Search tree, priority queue	$\Theta(\log n + \omega)$	$O(\log n)$
2D convex hull, triangulation	$O(n \log n + \omega n)$	$O(\log n)$
BFS, DFS, SCC, topological sort, block, bipartiteness, floodfill, biconnected components	$\Theta(m + n\omega)$	$O(m/n)$

Randomized Incremental Algorithms

- A random permutation provides each “element” a unique random priority
- Incrementally inserting each element into the current configuration

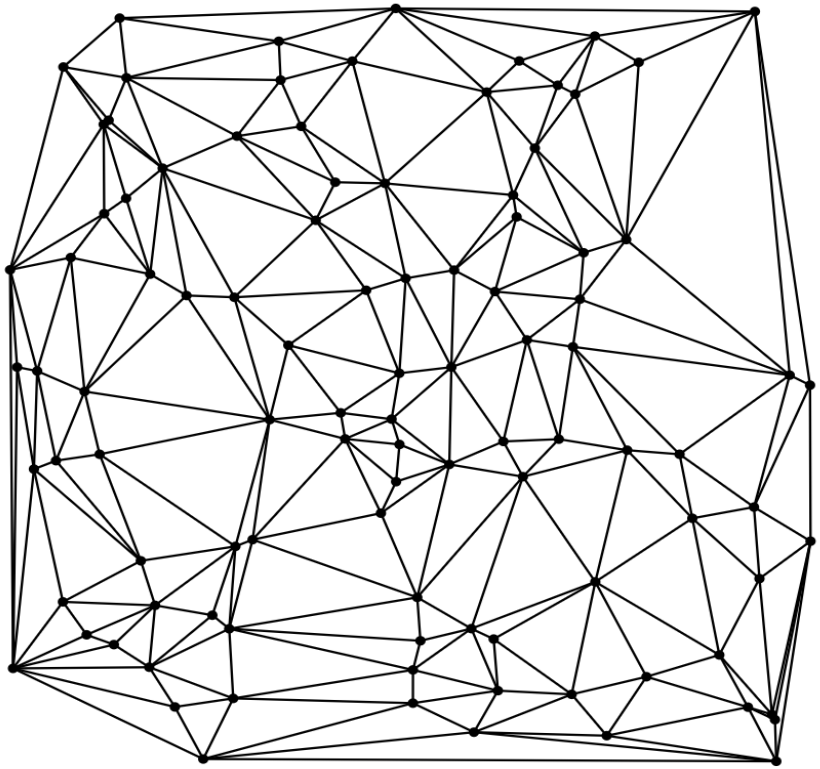
Randomized Incremental Algorithms

- A random permutation provides each “element” a unique random priority
- Incrementally inserting each element into the current configuration



Randomized Incremental Algorithms

- A random permutation provides each “element” a unique random priority
- Incrementally inserting each element into the current configuration



Randomized Incremental Algorithms

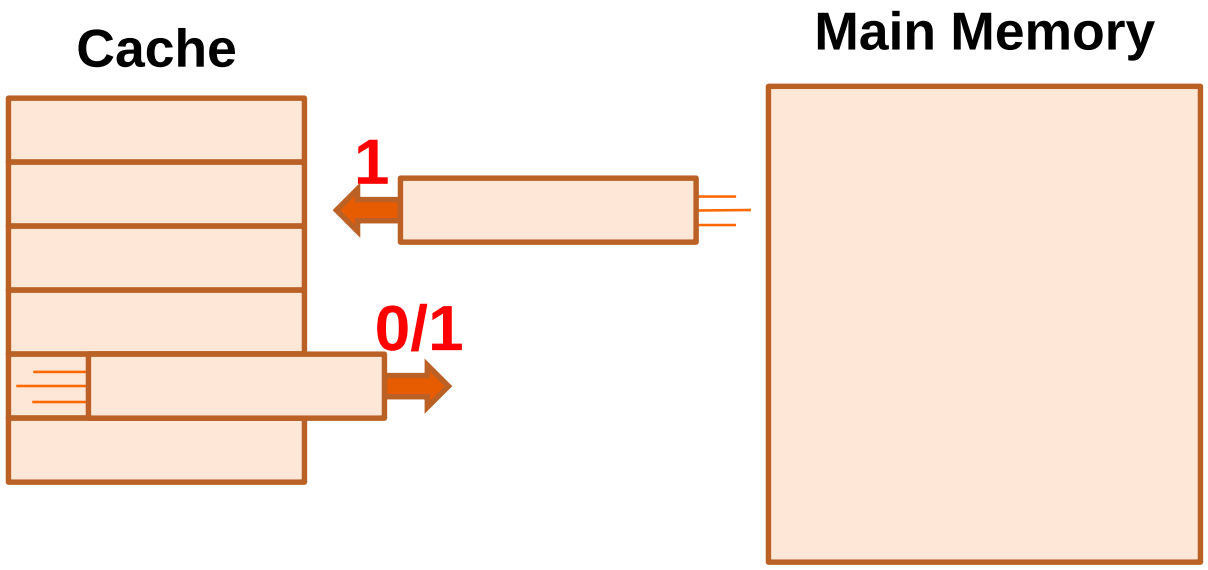
- Unfortunately, good parallel incremental algorithms for some geometry problems were unknown
- We describe a framework that can analyze the parallelism of many incremental algorithms
- Then we design a uniform approach to get the write-efficient versions of these algorithms

Problem	Work
Comparison sort, convex hull, Delaunay triangulation, k-d tree, interval tree, priority search tree	$\Theta(n \log n + \omega n)$
k-d linear programming, minimum enclosing disk	$\Theta(n + \omega n^{\epsilon})$

Cache Policy

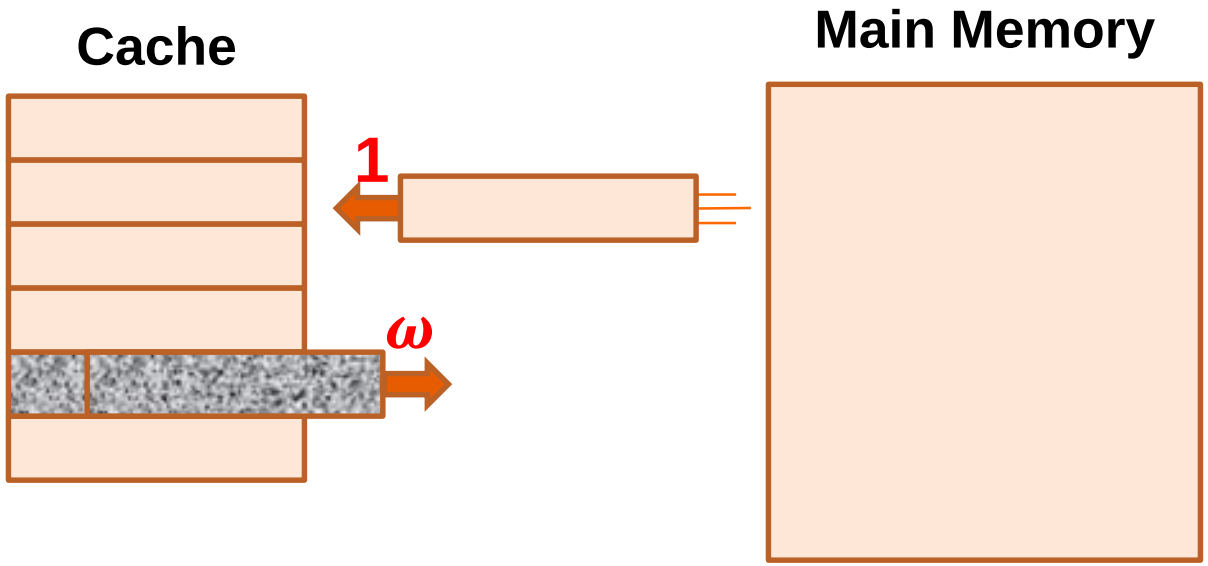
Cache policy: decide the block to evict when a cache miss occurs

- Least recent used (LRU) policy is the most practical implementation



Cache policy: decide the block to evict when a cache miss occurs

- Least recent used (LRU) policy is the most practical implementation



Challenge: LRU does not work well under the asymmetric setting

- Consider the sequence of repeated instructions:

  W(1), W(2), … , W(k-1), R(k), R(k+1), …, R(2k+1)

- A clever cache policy costs $k+2$ for this sequence with cache size k

W: write R: read

1	2	3	4	5	…	k-1	k k+1

Challenge: LRU does not work well under the asymmetric setting

- Consider the sequence of repeated instructions:

  W(1), W(2), … , W(k-1), R(k), R(k+1), …, R(2k+1)

- A clever cache policy costs $k + 2$ for this sequence with cache size k
- The LRU policy costs $(k - 1) \cdot \omega + k + 2$ with cache size $2k$

2k+1	1	2	4	5	…	2k-1	2k

Challenge: LRU does not work well under the asymmetric setting

- Consider the sequence of repeated instructions:

  W(1), W(2), … , W(k-1), R(k), R(k+1), …, R(2k+1)

- A clever cache policy costs $k + 2$ for this sequence with cache size k
- The LRU policy costs $(k - 1) \cdot \omega + k + 2$ with cache size $2k$

Classic LRU policy has an ω-time cost comparing to the clever policy!

Solution: The Asymmetric LRU policy

- The cache is separated into two equal-sized pools: a read pool and a write pool

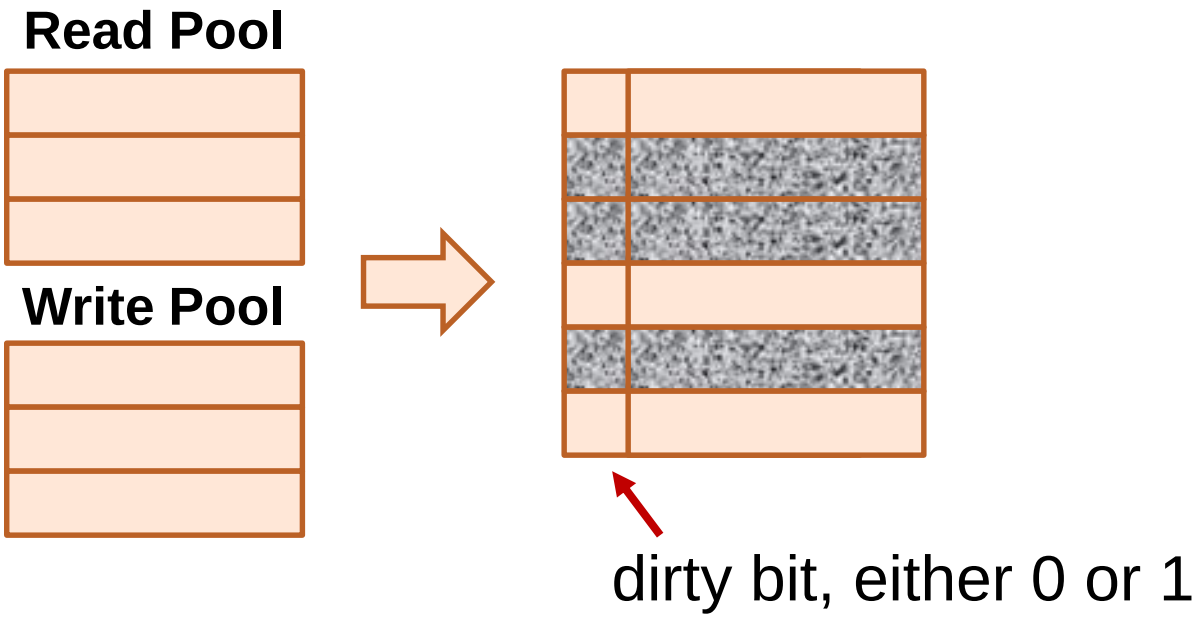
Solution: The Asymmetric LRU policy

- When reading a location, if the block is:
  - in the read pool, the read is free
  - in the write pool, the block will be copied to read pool
  - in neither, the block is loaded from main memory
- The rules for write pool are symmetric, but cost $\omega + 1$ since the blocks are all dirty and need to be written back

The new Asymmetric LRU policy is 3-competitive to the optimal policy

In practice,

- The cache does not need to be explicitly separated into two pools physically
- Use the **dirty bit** to identify and check, and change the eviction rule accordingly



Summary

Summary

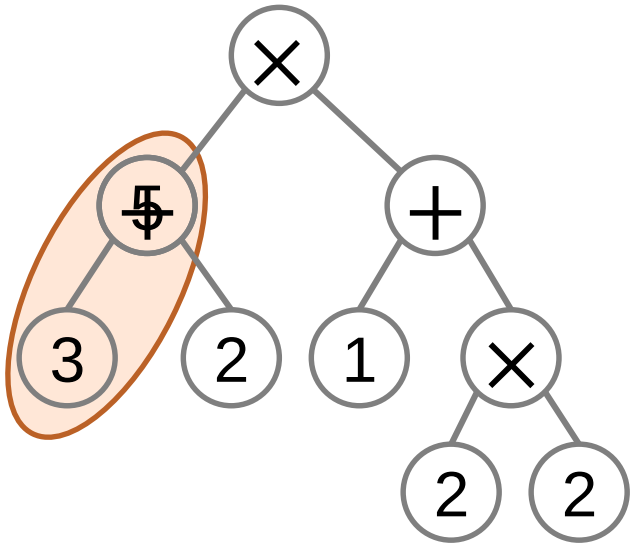
- The new emerging memories are upcoming, which rise the challenge of read/write asymmetry in algorithm design
- New cost models to capture this asymmetry
- New upper and lower bounds on a number of fundamental problems
- This area is still new — there are many other problems worth investigating

Thank you!

Tree contraction

- An abstract function that solves many problems in parallel, with linear work and logarithm depth
- Example: expression evaluation
  - Input: a binary rooted tree
  - Output: the final answer
- The ***rake*** operation:

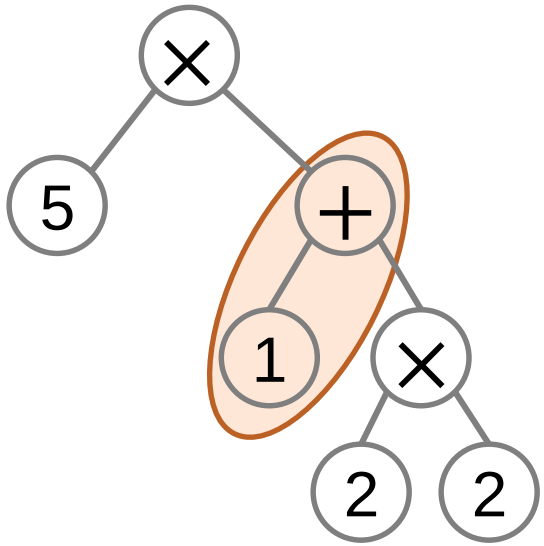
  Remove (evaluate) a child and its parent



Tree contraction

- An abstract function that solves many problems in parallel, with linear work and logarithm depth
- Example: expression evaluation
  - Input: a binary rooted tree
  - Output: the final answer
- The ***rake*** operation:

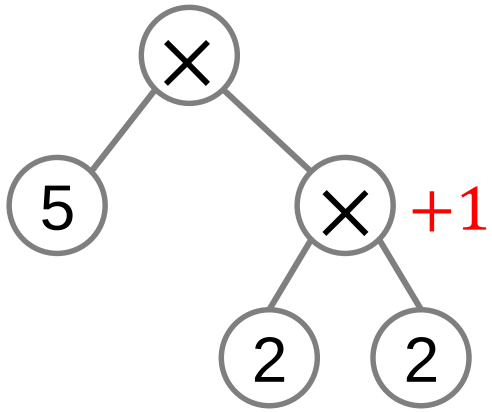
  Remove (evaluate) a child and its parent



Tree contraction

- An abstract function that solves many problems in parallel, with linear work and logarithm depth
- Example: expression evaluation
  - Input: a binary rooted tree
  - Output: the final answer
- The ***rake*** operation:

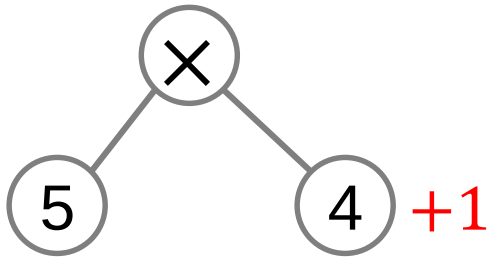
  Remove (evaluate) a child and its parent



Tree contraction

- An abstract function that solves many problems in parallel, with linear work and logarithm depth
- Example: expression evaluation
  - Input: a binary rooted tree
  - Output: the final answer
- The ***rake*** operation:

  Remove (evaluate) a child and its parent



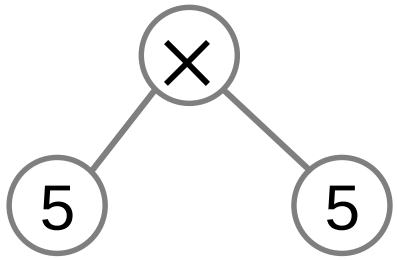
Tree contraction

- An abstract function that solves many problems in parallel, with linear work and logarithm depth
- Example: expression evaluation
  - Input: a binary rooted tree
  - Output: the final answer
- The ***rake*** operation:

  Remove (evaluate) a child and its parent

  Tree nodes decrease by a constant fraction

  $O(\log n)$ round to finish, linear work



Challenges of tree contraction

- Each rake operation corresponds to a write!
- The output of most of the applications (expression evaluation, subtree size, LCA queries) has sublinear size

Bottom-up? Top-down?

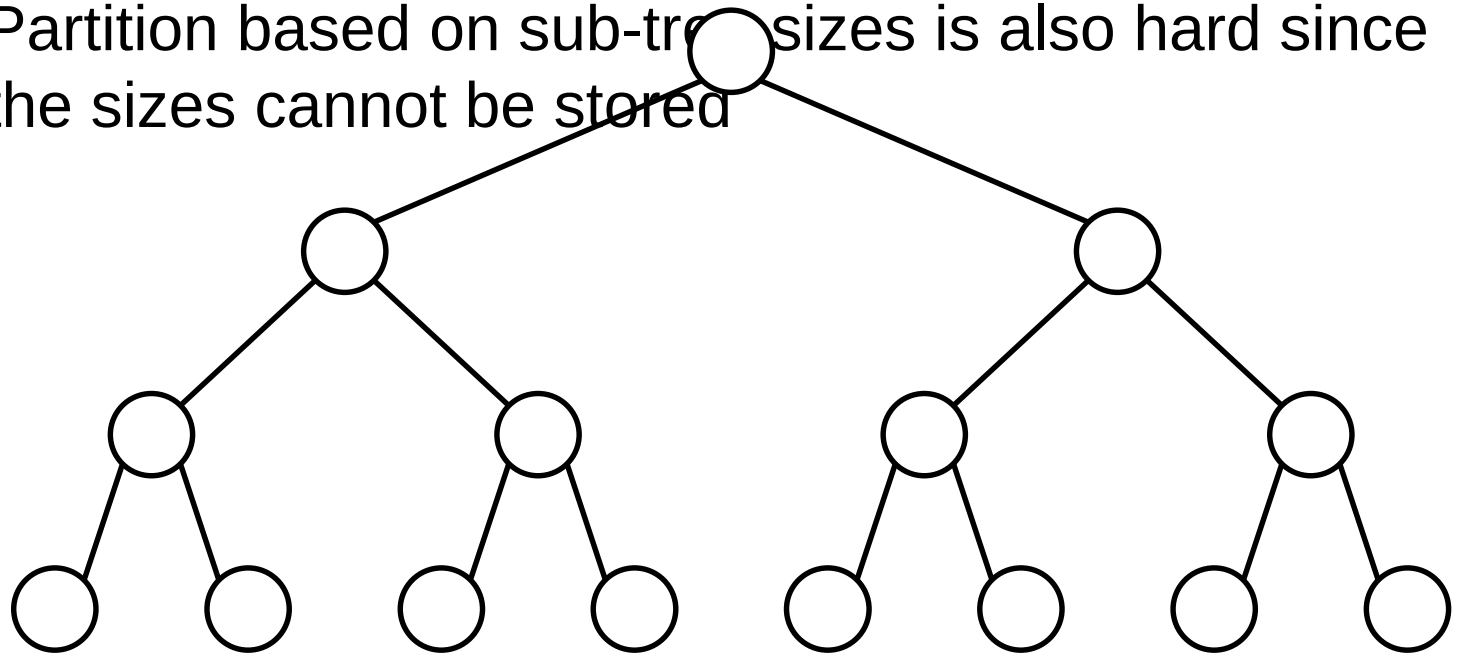
- Partition the tree into sufficiently smaller components, solve each one sequentially and independently?

How to partition the tree evenly without using writes?

How to solve each subproblem without using writes?

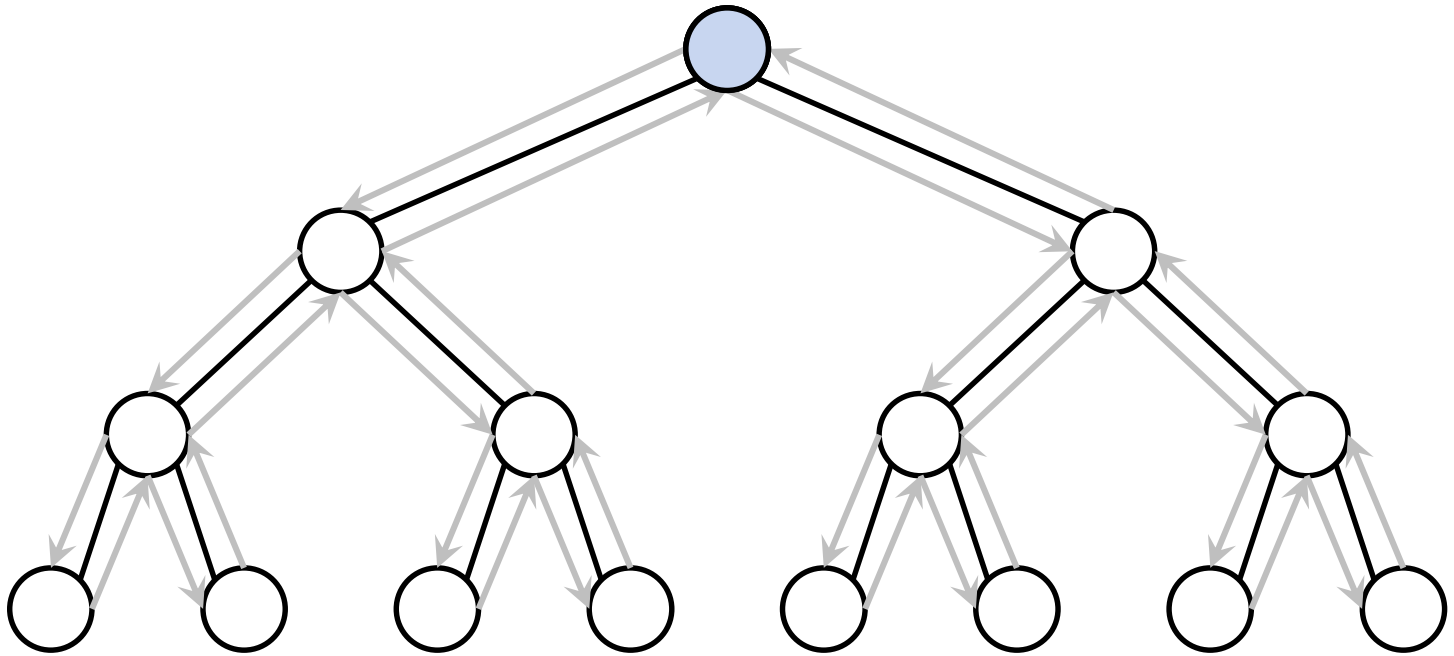
How to partition the tree evenly without using many writes?

- Goal: pick $O(s)$ nodes to partition a tree of size n, each component with size n/s
- Take s random nodes won't work: each will only expect to remove a small number of nodes, leaving the top part with expected size of $n - O(s \log n)$
- Partition based on sub-tree sizes is also hard since the sizes cannot be stored



How to partition the tree evenly without using many writes?

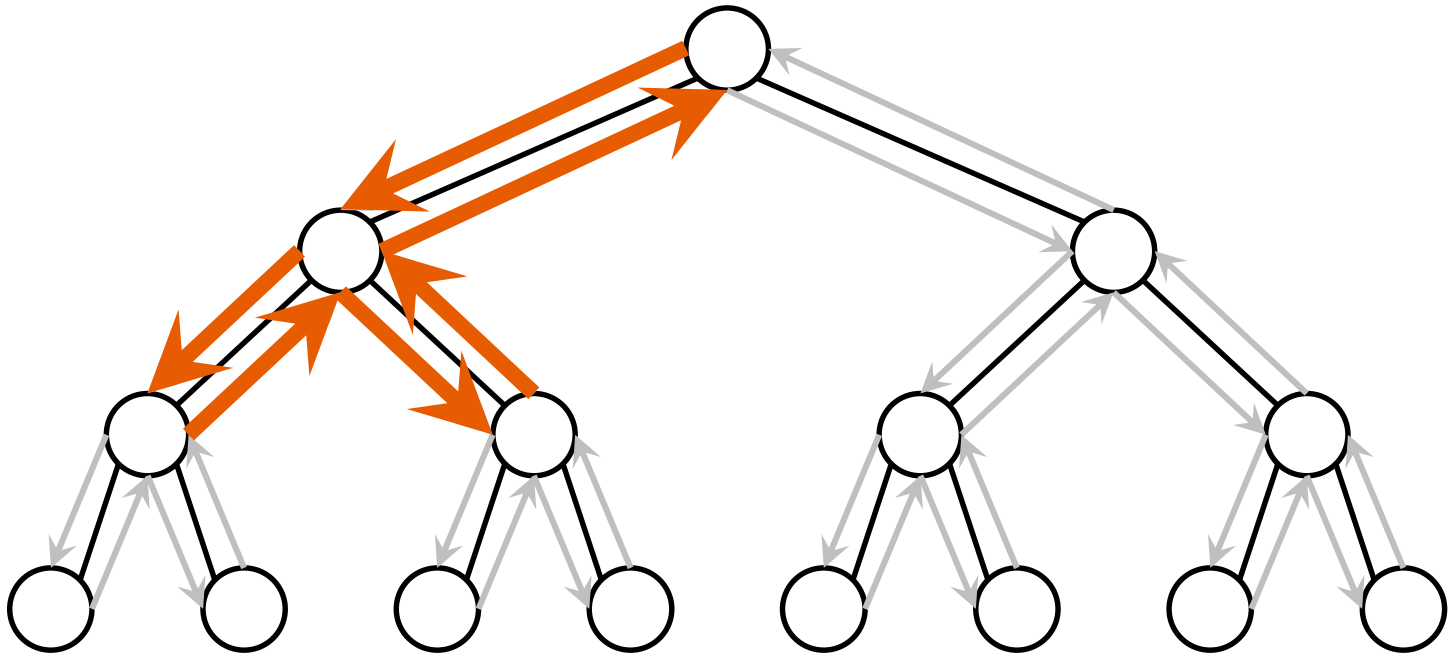
- Euler tour of a tree: the Euler circuit to traverse a tree (similar in a depth-first search)



How to partition the tree evenly without using many writes?

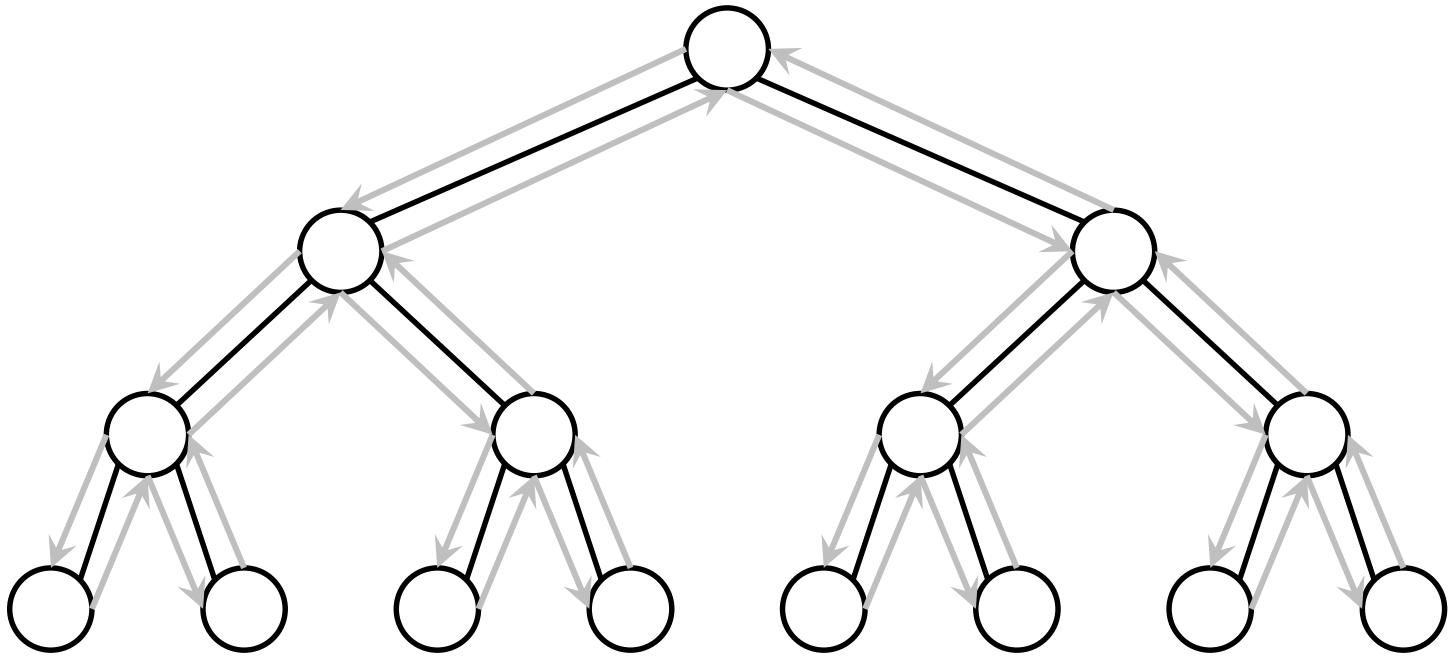
- Euler tour of a tree: the Euler circuit to traverse a tree (similar in a depth-first search)

- Can be finished with constant space



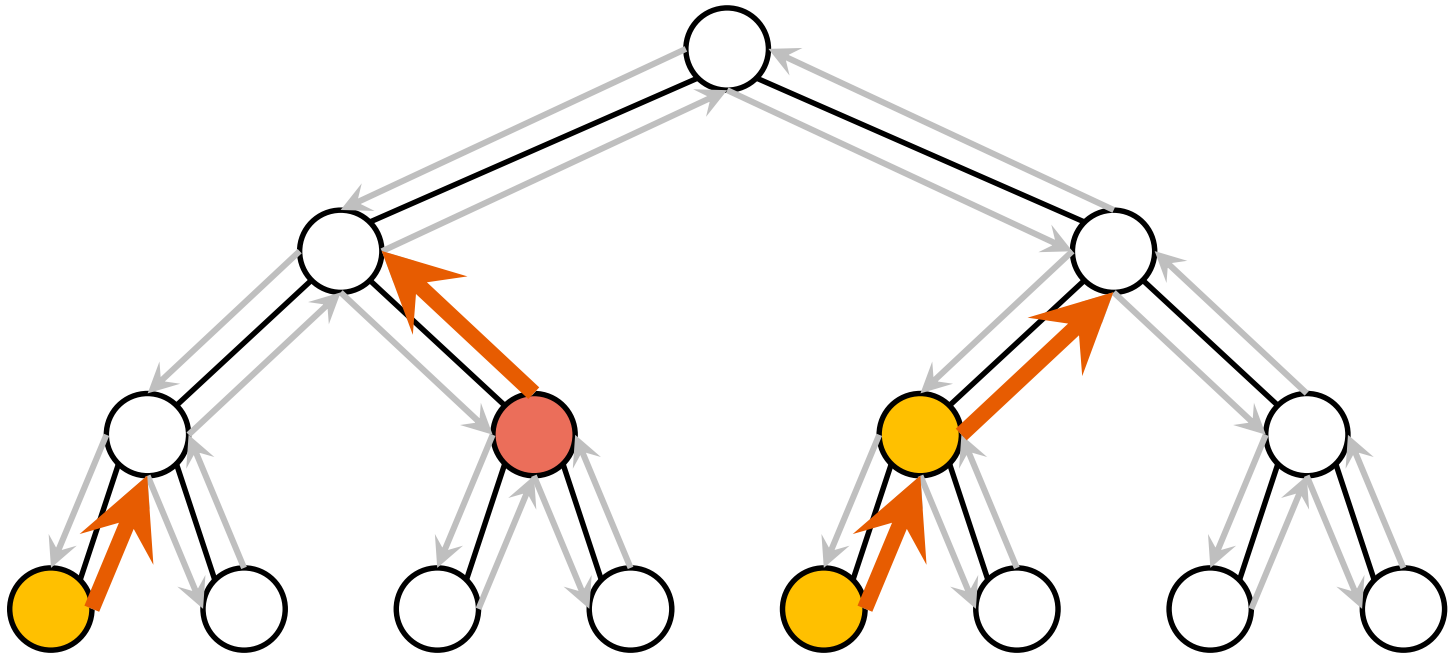
Selecting partition nodes

- Step 1: each node is marked with probability s/n (yellow)
- Step 2: for each yellow node, starting from all out-edges, traverse until next yellow nodes, and mark every (n/s)-th node (red)



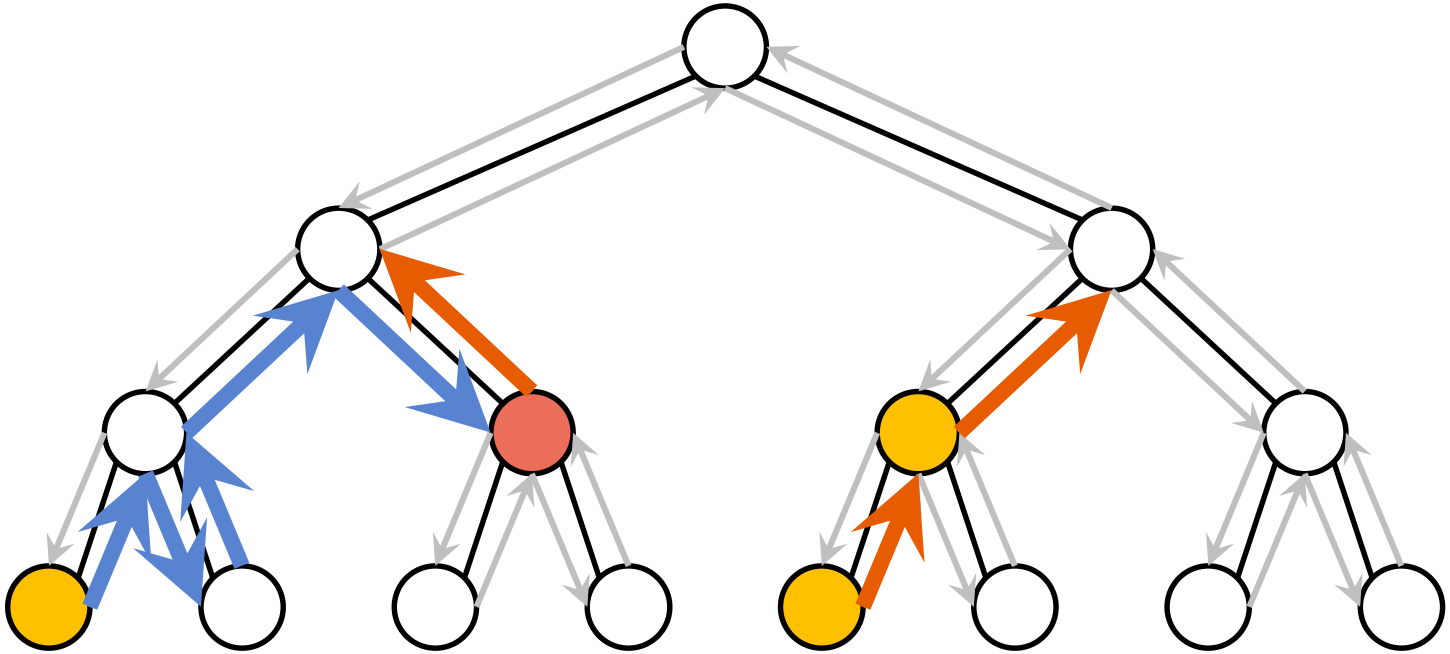
Selecting partition nodes

- Step 1: each node is marked with probability s/n (yellow)
- Step 2: marked $O(s)$ red nodes
- Step 3: for each marked node, starting from the edge to its parent, traverse based on the Euler tour until reaching the next colored node, and mark the highest node on the path (green)



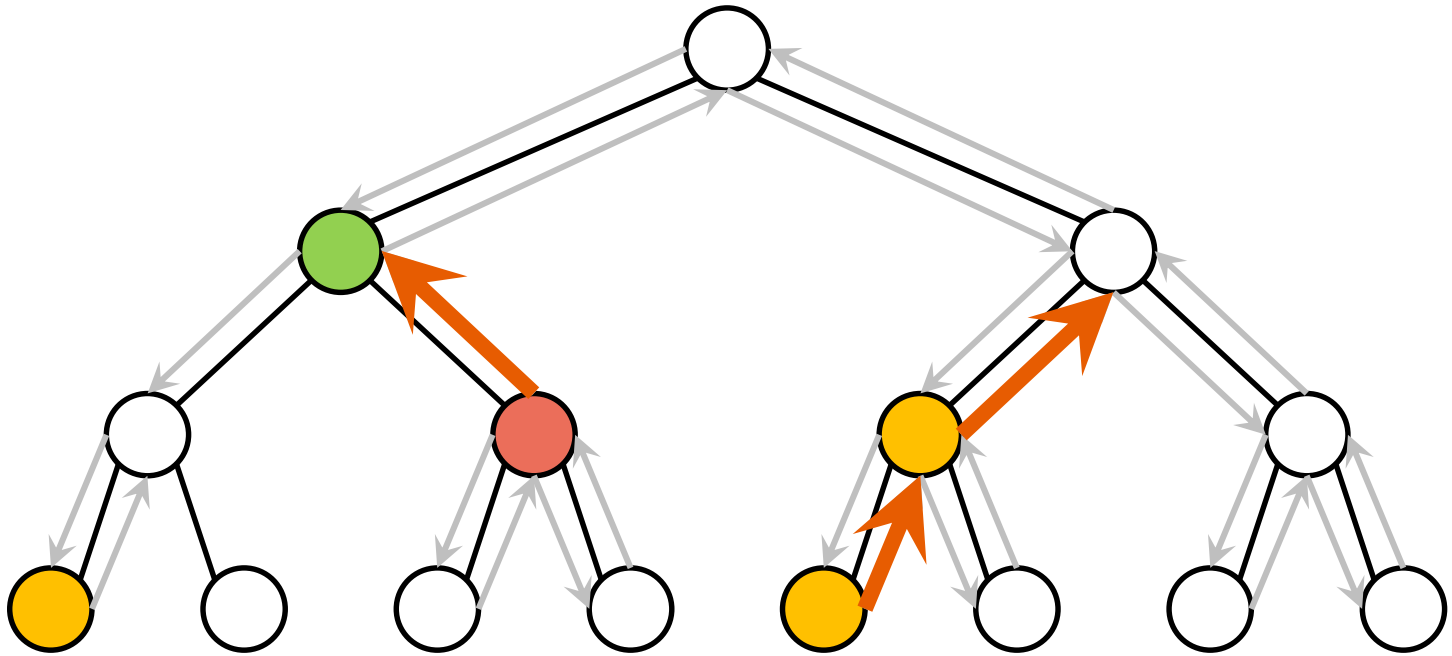
Selecting partition nodes

- Step 1: each node is marked with probability s/n (yellow)
- Step 2: marked $O(s)$ red nodes
- Step 3: for each marked node, starting from the edge to its parent, traverse based on the Euler tour until reaching the next colored node, and mark the highest node on the path (green)



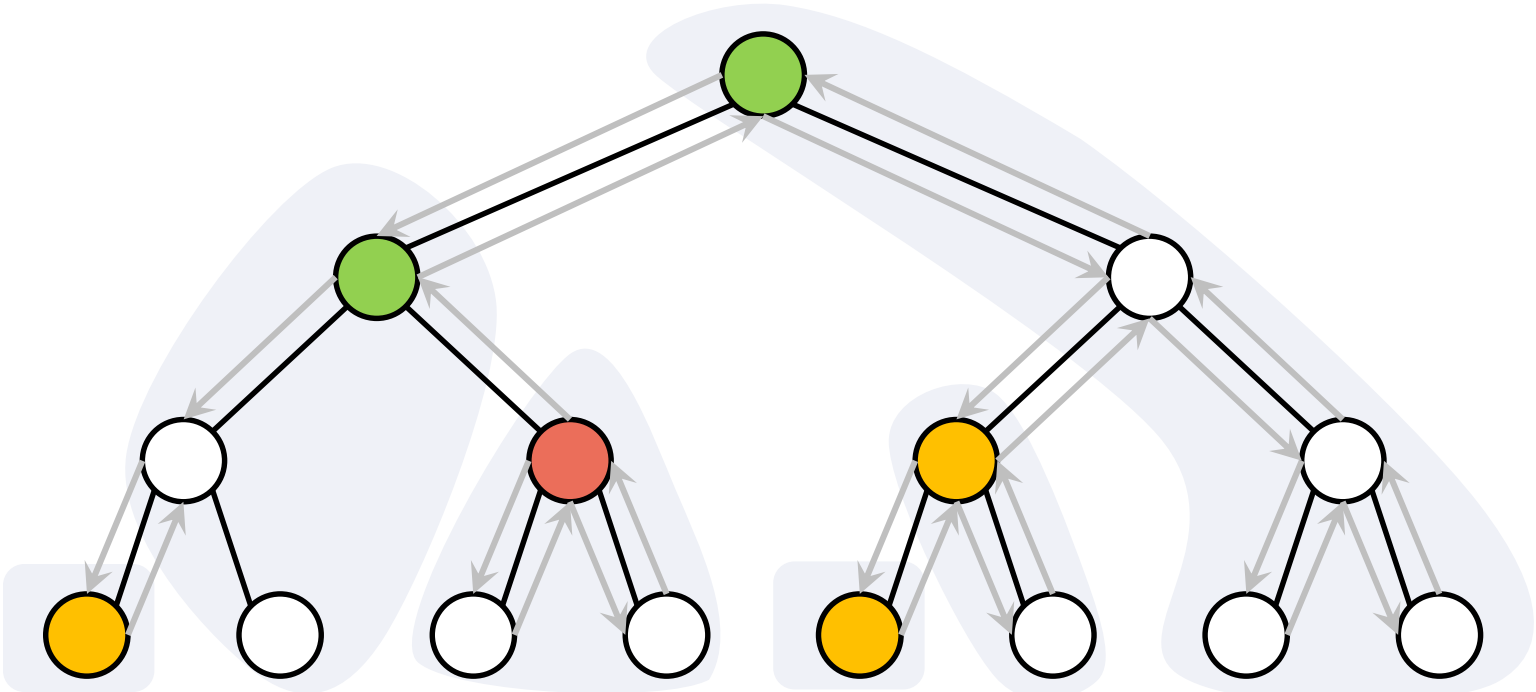
Selecting partition nodes

- Step 1: each node is marked with probability s/n (yellow)
- Step 2: marked $O(s)$ red nodes
- Step 3: for each marked node, starting from the edge to its parent, traverse based on the Euler tour until reaching the next colored node, and mark the highest node on the path (green)



Selecting partition nodes

- Step 1: each node is marked with probability s/n (yellow)
- Step 2: marked $O(s)$ red nodes
- Step 3: for each marked node, starting from the edge to its parent, traverse based on the Euler tour until reaching the next colored node, and mark the highest node on the path (green)



Selecting partition nodes

- Step 1: each node is marked with probability s/n (yellow)
- Step 2: marked $O(s)$ red nodes
- Step 3: for each marked node, starting from the edge to its parent, traverse based on the Euler tour until reaching the next colored node, and mark the highest node on the path (green)

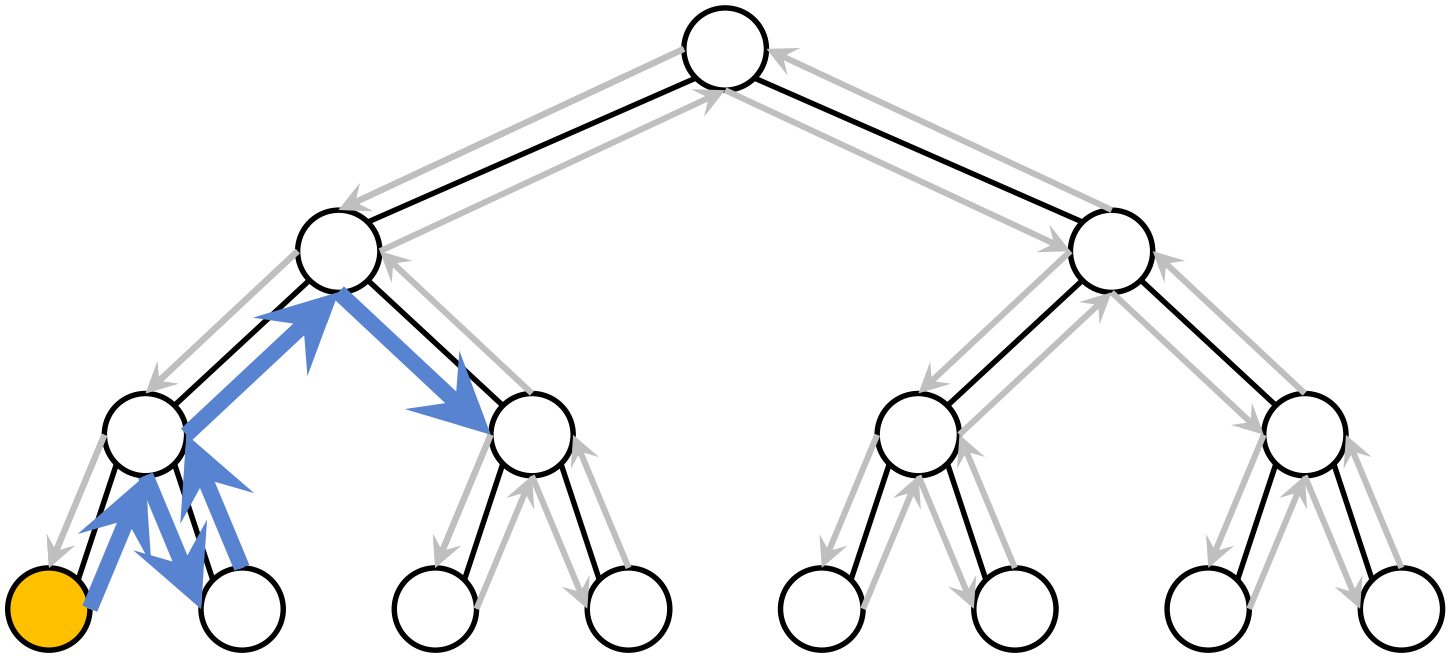
In total, we expect to mark no more than $O(s)$ nodes (writes), and linear work

What is the depth? (work of step 2)

How good this partition is?

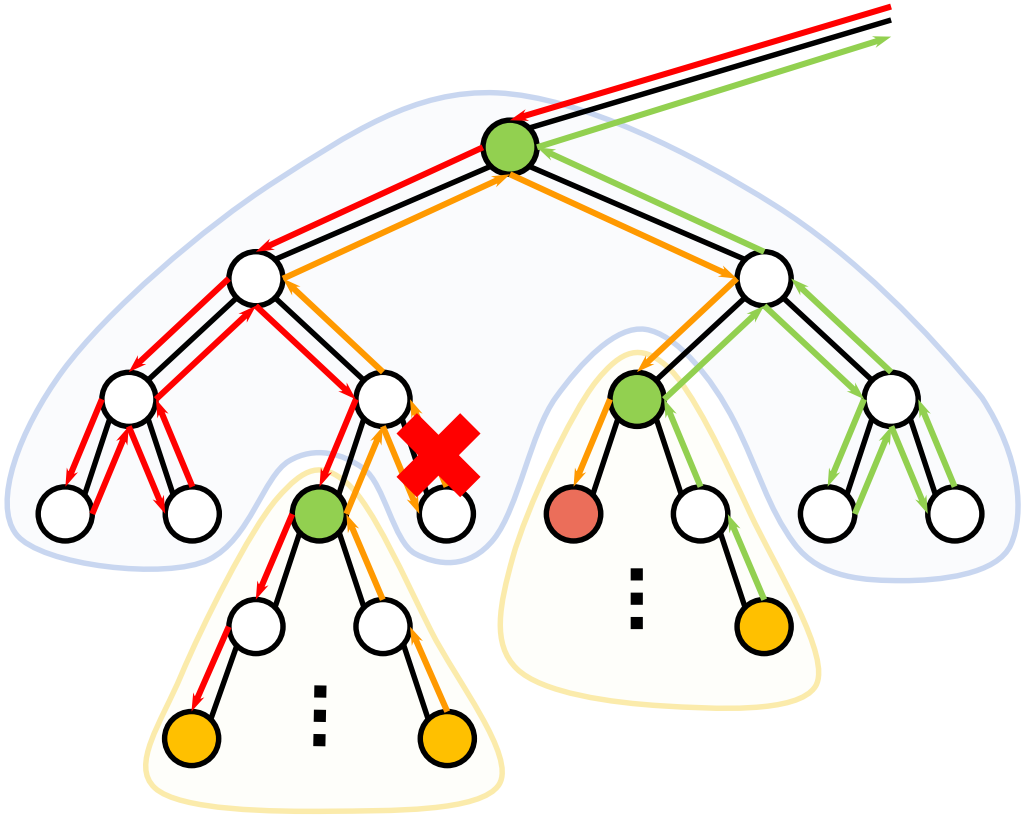
The gap between two yellow nodes

- The probability p that neither of the next k tree nodes are yellow is $\left(1 - \frac{s}{n}\right)^k$
- When $k = \frac{cn \log n}{s}$, $p = n^{-c}$ (high probability bound)
- For example, when $s = n/\omega$, then $k = O(\omega \log n)$



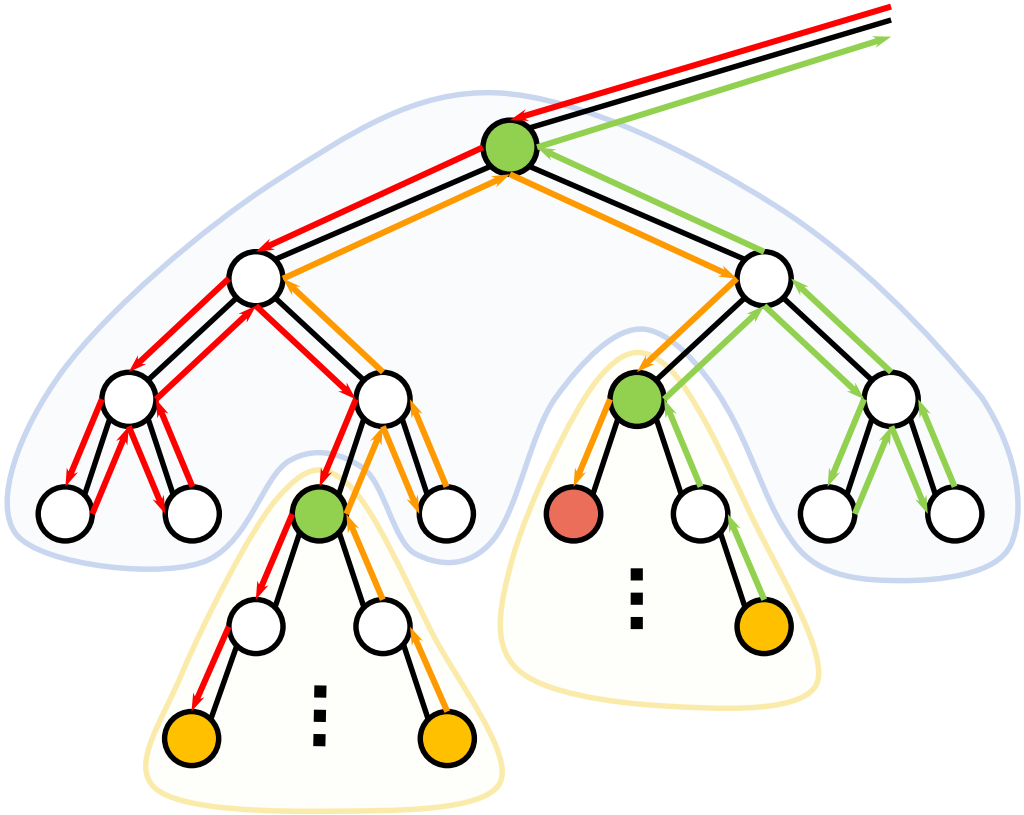
The average size of the component after partitioning

- $\Theta(n/s)$ (optimal)
- Each component contains at most three segments between two yellow and red node, each with size $\leq n/s$



The average size of the component after partitioning

- $\Theta(n/s)$ (optimal)
- Each component contains at most three segments between two yellow and red node, each with size $\leq n/s$



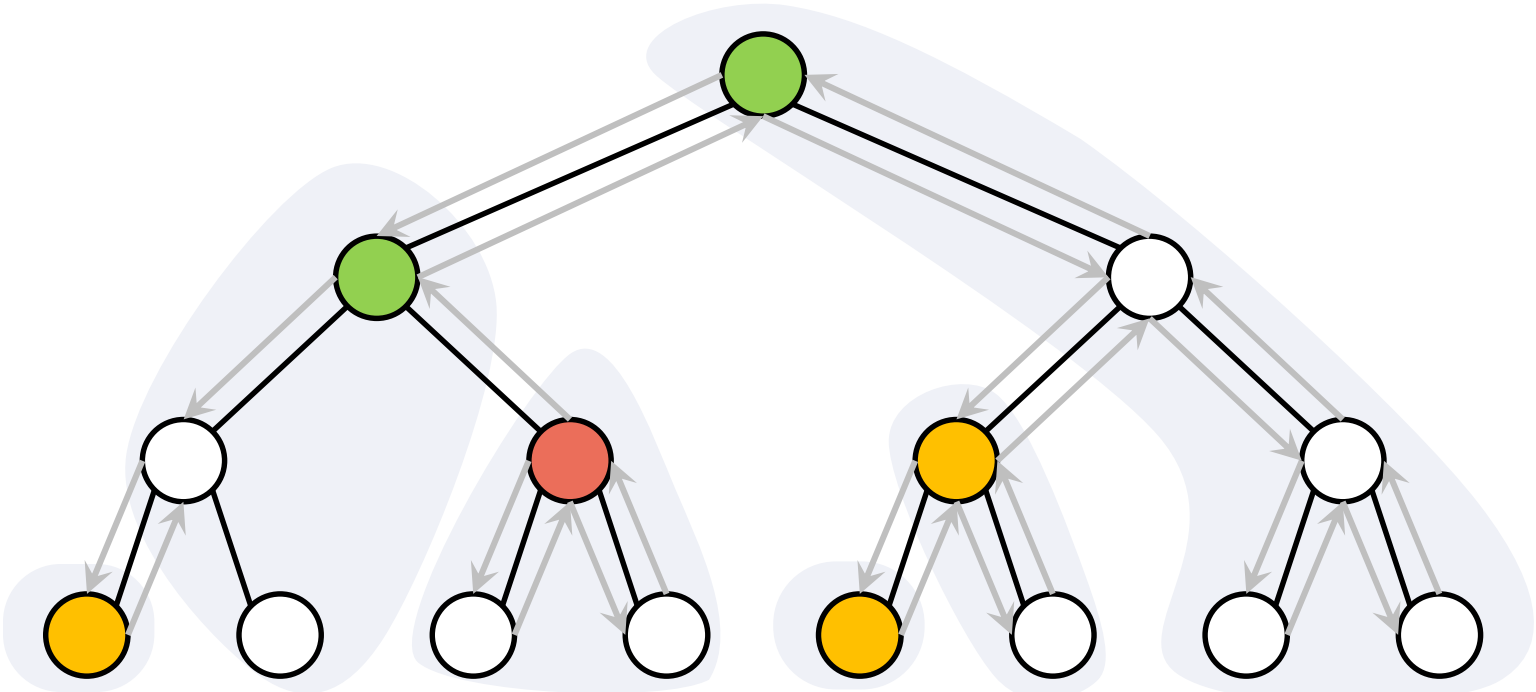
The tree-partitioning algorithm

- Requires linear work, $O(n \log n / s)$ depth, $O(s)$ writes, and partitions the tree with component size $O(n/s)$

- In particular, when $s = n/\omega$, the algorithm has $O(\omega \log n)$ depth, $O(n/\omega)$ writes, and generates components with size $O(\omega)$

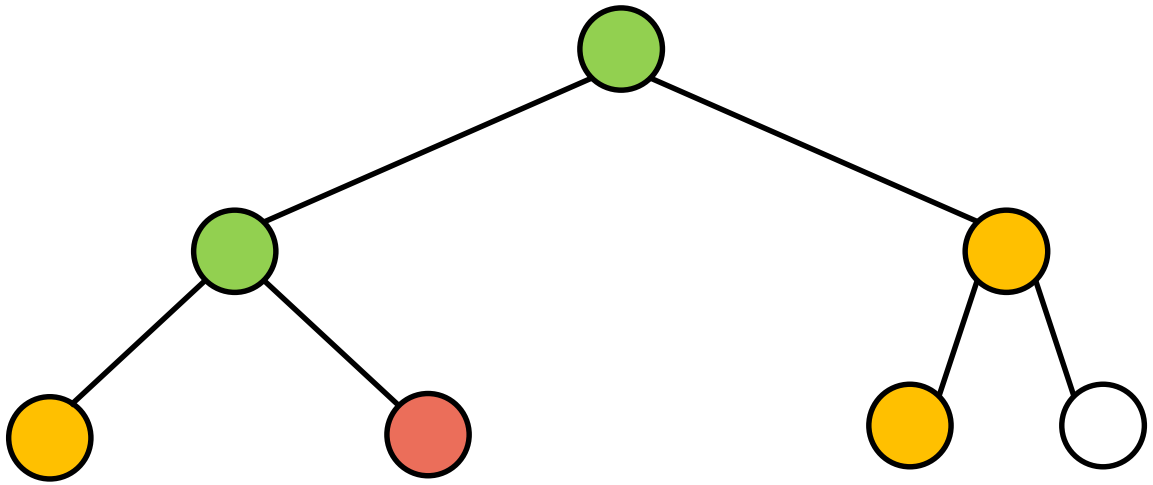
Contracting each component

- With $O(\omega)$ fast-memory, we can sequentially contract each component with linear work and depth in term of the component size



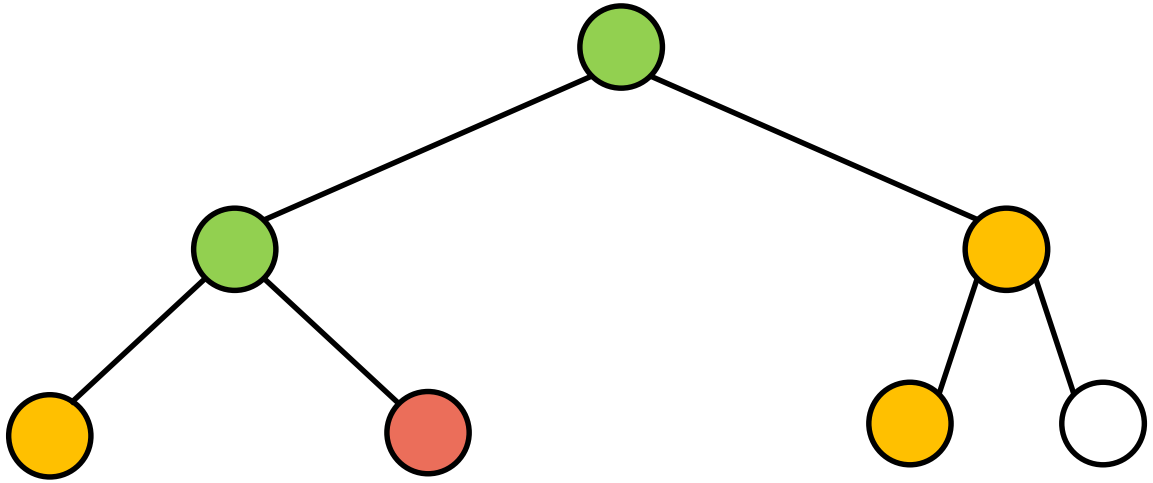
Contracting each component

- With $O(\omega)$ fast-memory, we can sequentially contract each component with linear work and depth in term of the component size
- The tree will contains $O(n/\omega)$ nodes



Final contraction

- The tree will contains $O(n/\omega)$ nodes
- Use any classic tree-contraction algorithm with linear work and writes, and logarithm of depth



Write-efficient tree-contraction algorithm

- Given a tree with n nodes, the algorithm requires:
  - $O(n)$ work and reads
  - $O(n/\omega)$ writes
  - $O(\omega \log n)$ depth
- Comparing to the classic tree-contraction algorithm, the number of writes can be reduced by $O(\omega)$

Tree applications: LCA, subtree size, tree distances, tree isomorphism, maximal subtree isomorphism, and any associative operation on trees

Graph applications: graph isomorphism, computing the 3-connected components, and finding an explicit planar embedding

Other applications: expression evaluation, common subexpression elimination, line breaking, etc.