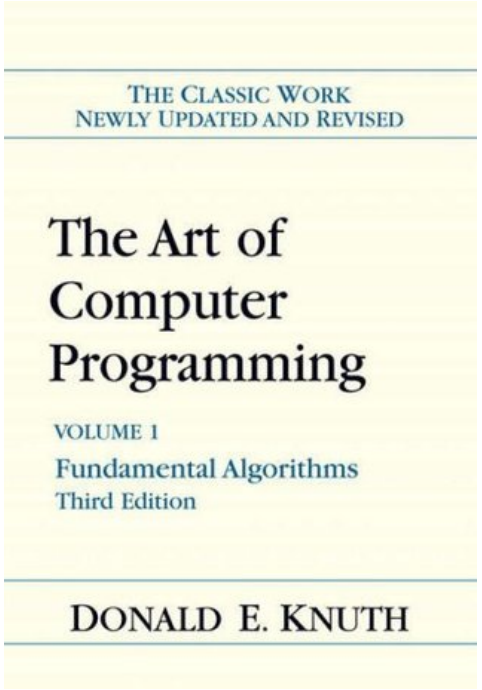
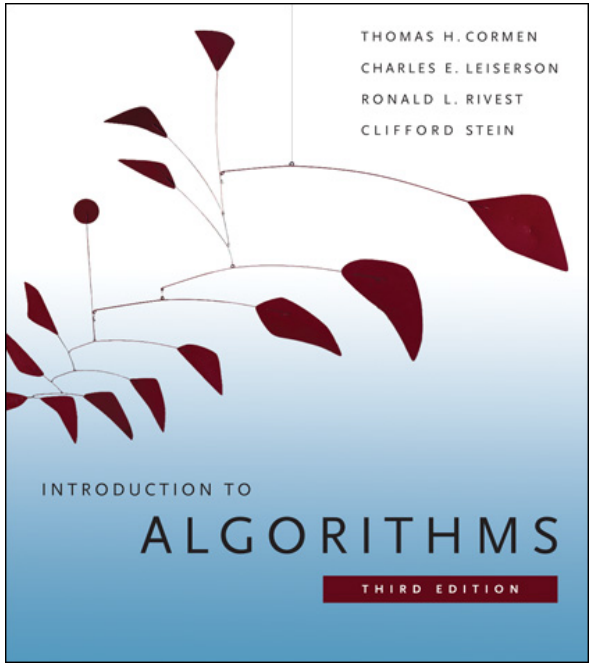
Write-efficient algorithms

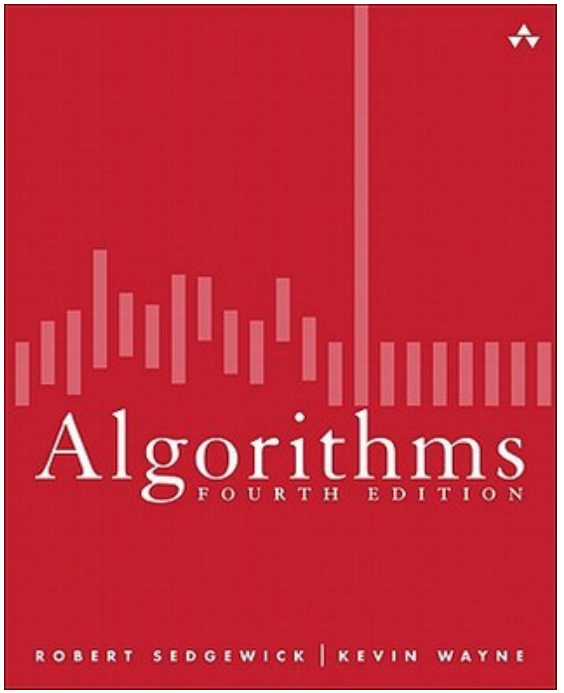
6.886 Algorithm Engineering

Yan Gu, May 7, 2019

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?

Intel® Optane™ DC Persistent Memory

BIG MEMORY BREAKTHROUGH FOR YOUR BIGGEST DATA CHALLENGES

Intel® Optane™ DC persistent memory represents a groundbreaking technology innovation.[1] [2] [3] [4] Delivered with the next-generation 2nd Generation Intel® Xeon® Scalable processors, this workload optimized technology will help businesses extract more actionable insights from data – from cloud and databases, to in-memory analytics, and content delivery networks.

What Is Intel® Optane™ DC Persistent Memory?

Intel® Optane™ DC persistent memory is an innovative memory technology that delivers a unique combination of affordable

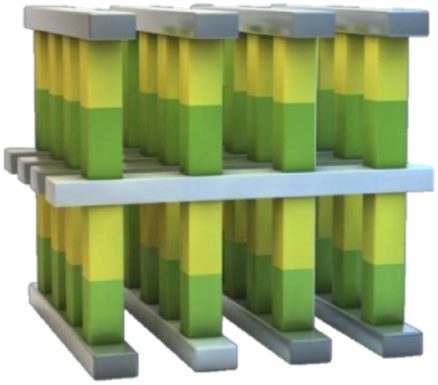
Emerging Memory Technologies

Motivation:

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

Emerging non-volatile main memory (NVRAMs) Technologies

- Persistent
- Significantly lower energy
- Higher density: 512GB per DIMM
- Read latencies approaching DRAM
- Random-access

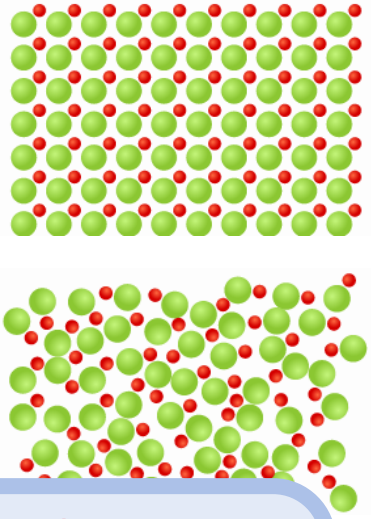


3D XPoint

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

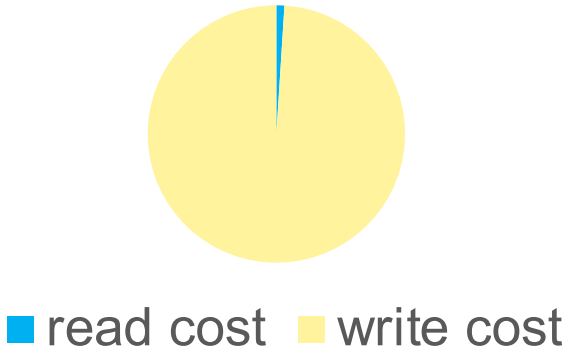


Writes incur higher energy costs, higher latency, and lower per-DIMM bandwidth (power envelope constraints)

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

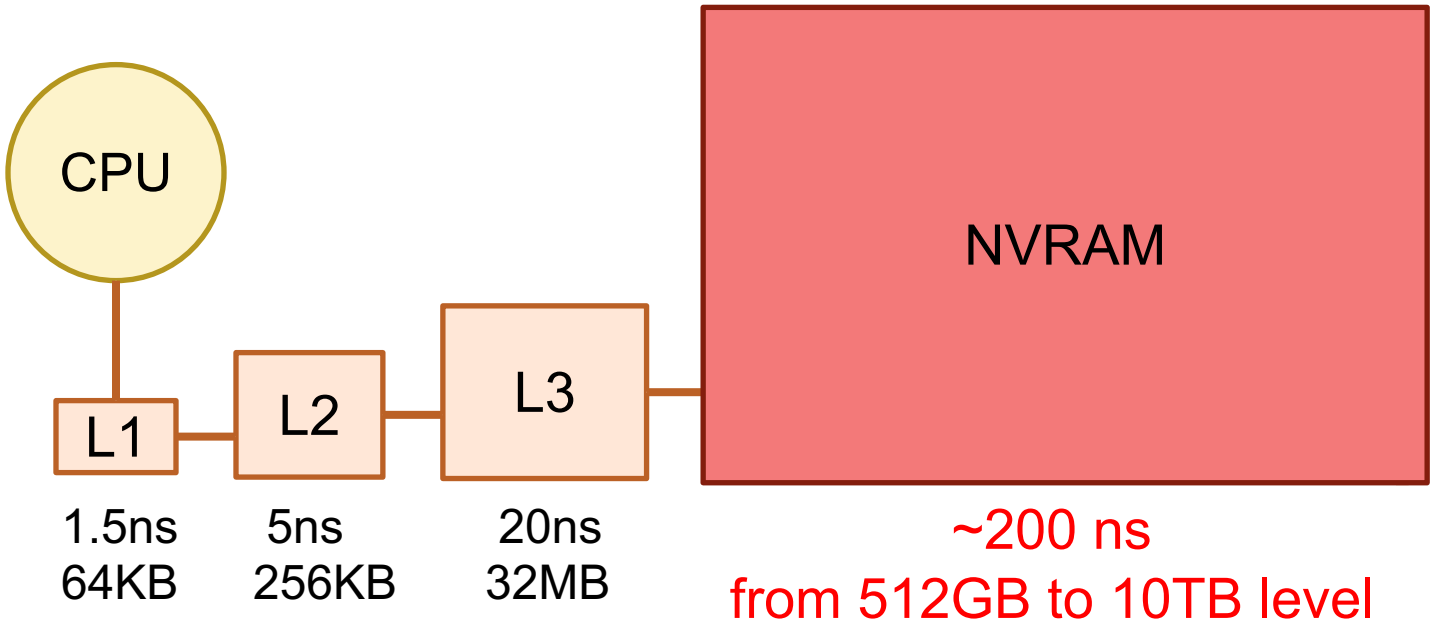


Sorting algorithm 2:
$200n$ reads and $2n$ writes on n elements
We can sort 25 million entries per joule

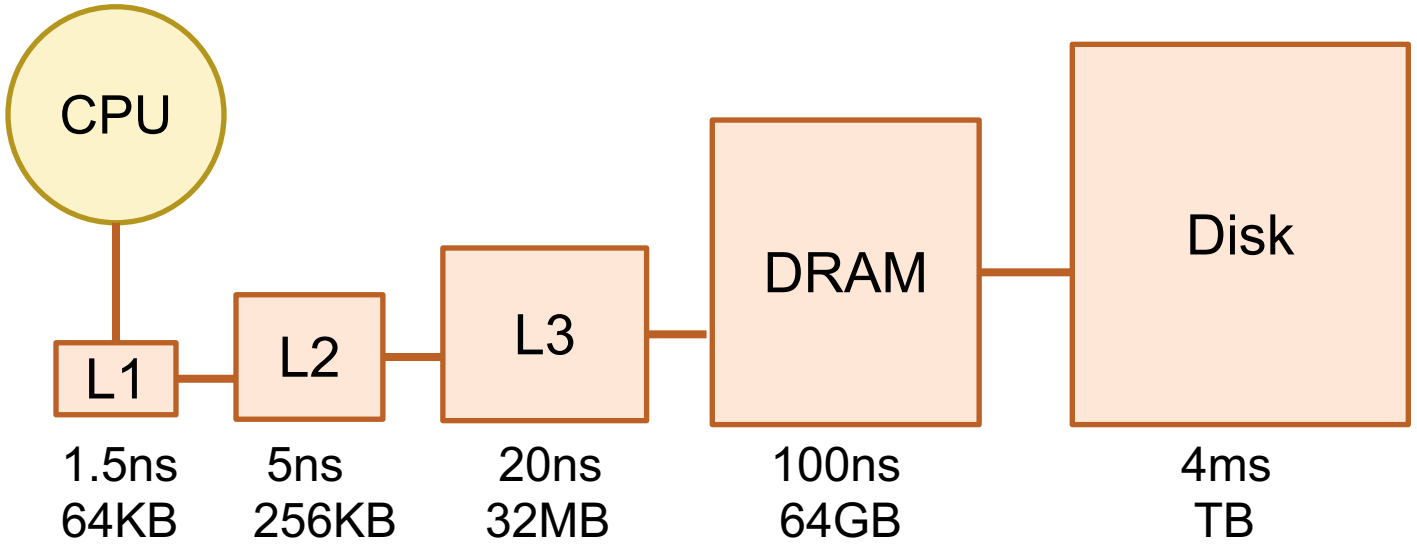
Why does it matter?

- 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 energy → Lower per-chip (memory) bandwidth
- **Let the parameter $\omega > 1$ be the cost for writes relative to reads**
  - Expected to be between 5 to 30

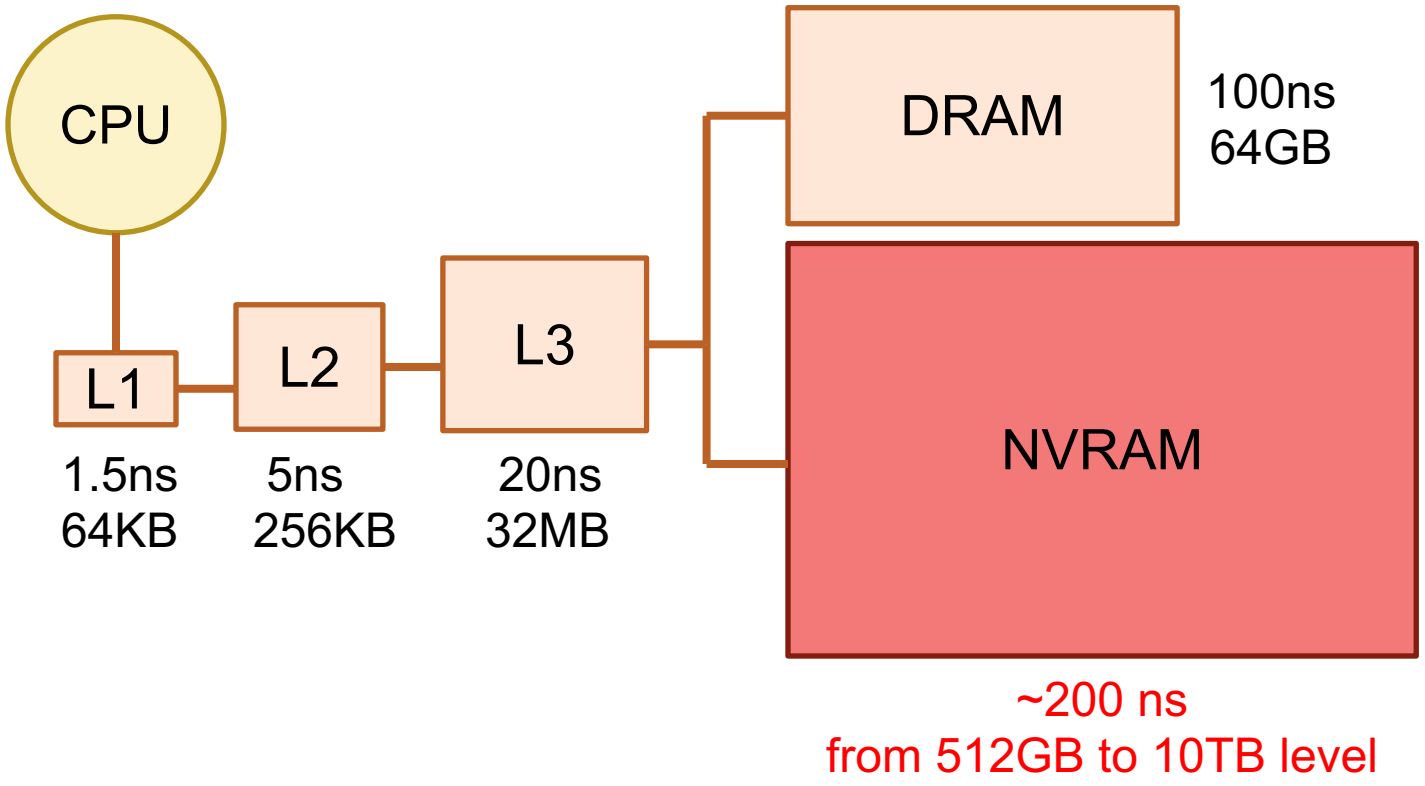
Evolution of the memory hierarchy



Evolution of the memory hierarchy



Evolution of the memory hierarchy



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

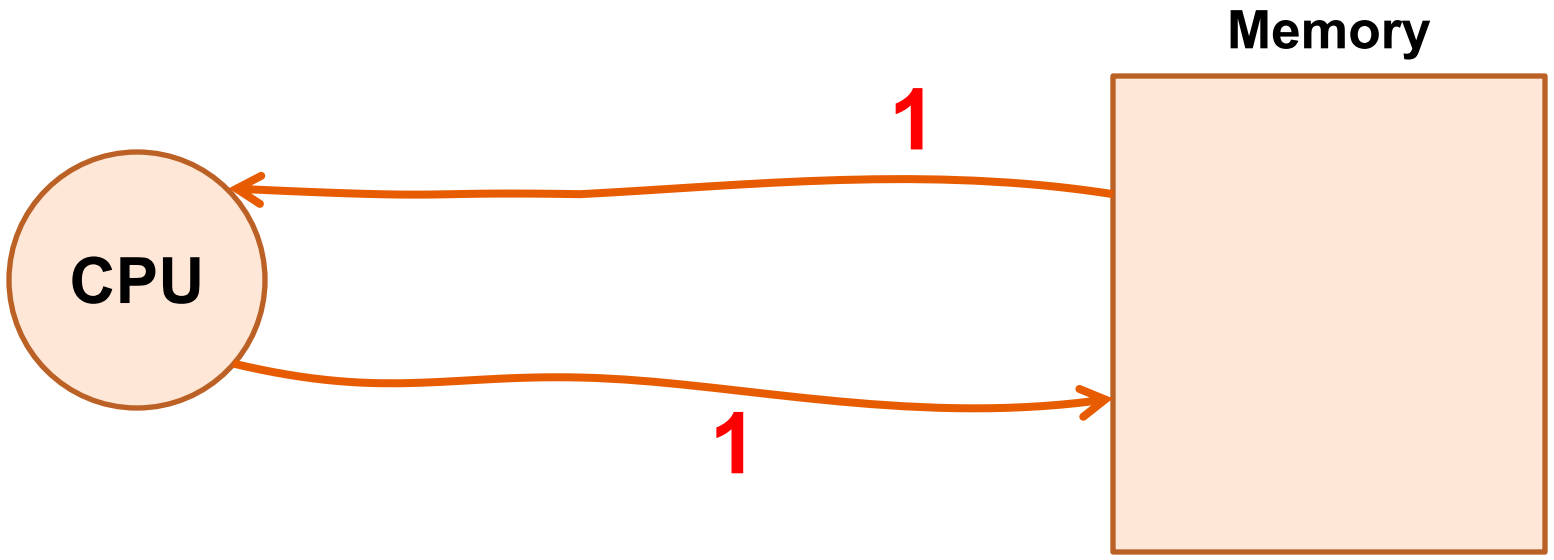
- The new NVRAMs raise the challenge to design write-efficient algorithms

- 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)

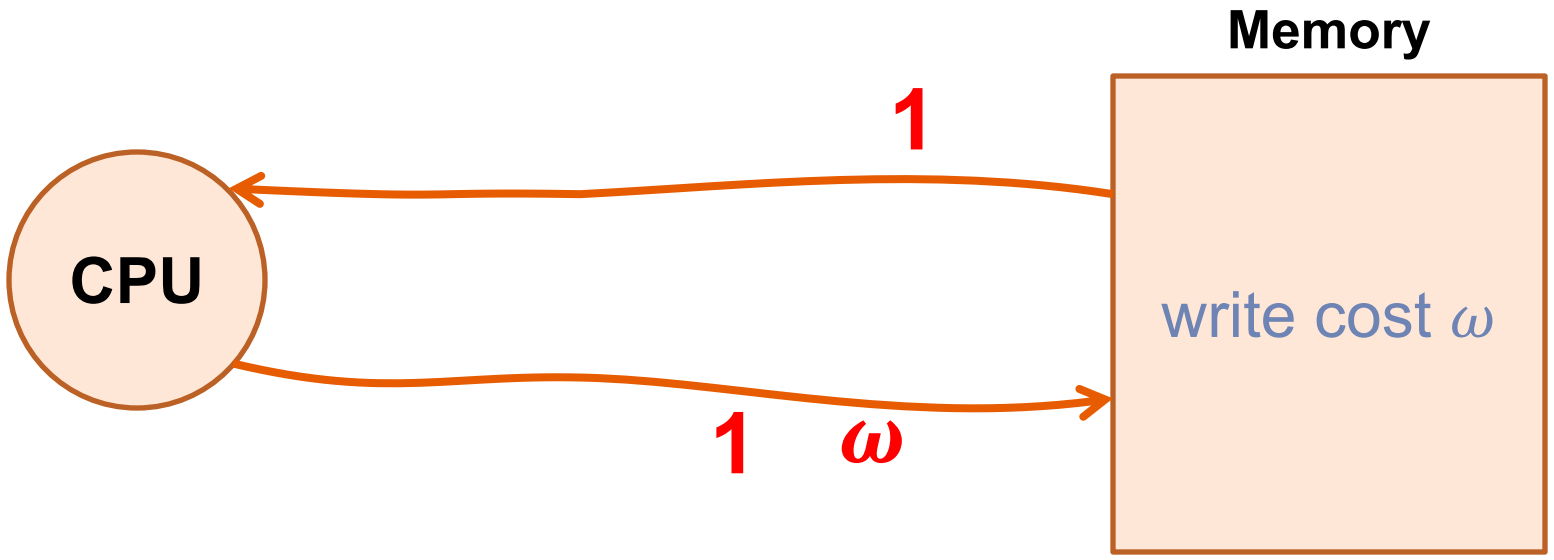
- Unit cost for:
  - Any instruction on $\Theta(\log n)$-bit words
  - Read/write a single memory location from an infinite memory



Read/write asymmetry in RAM?

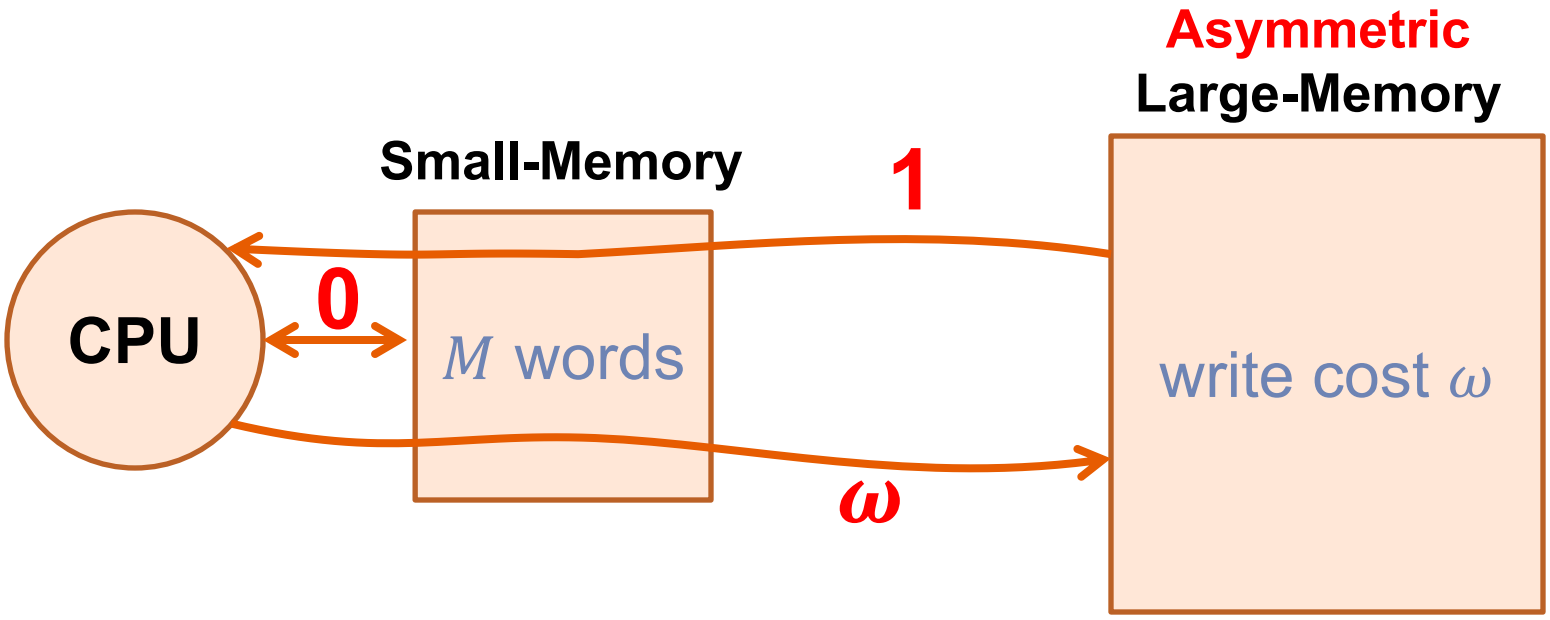
- A single write cost ω instead of 1

But every instruction writes something…



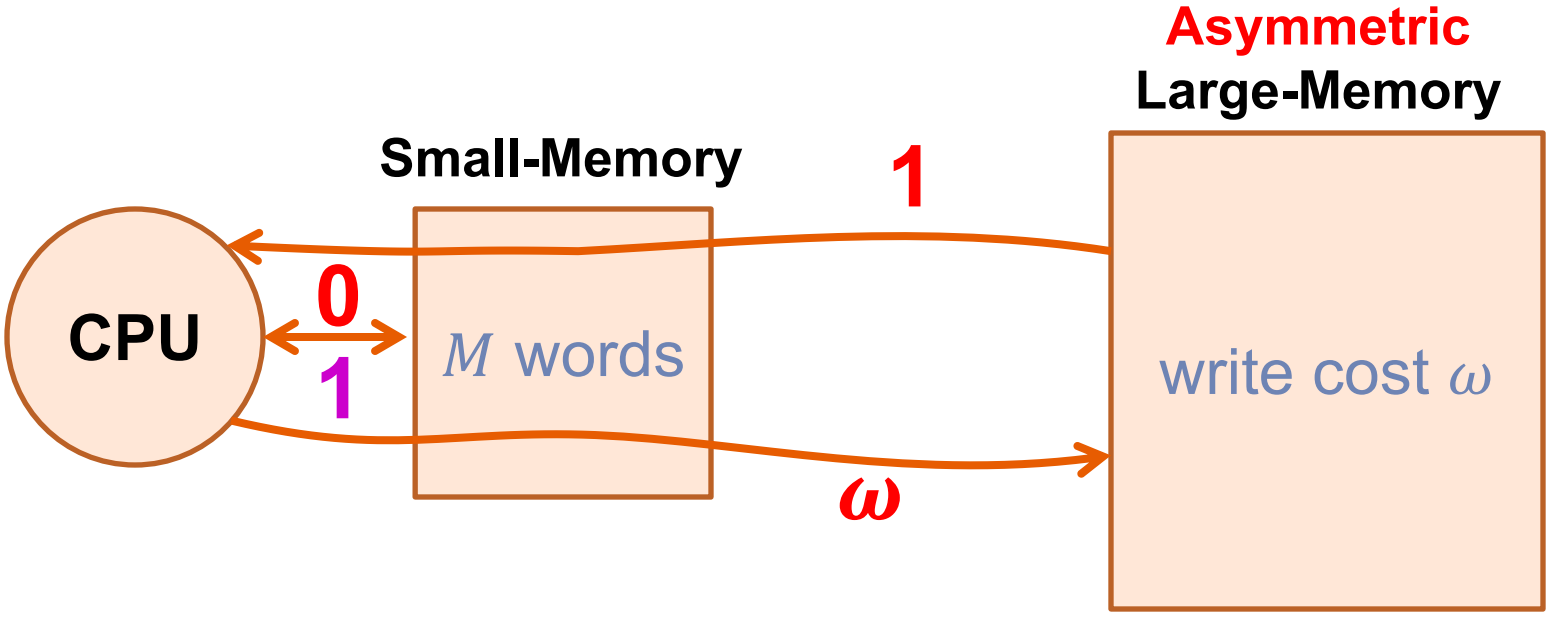
(M, ω)-Asymmetric RAM (ARAM)

- 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

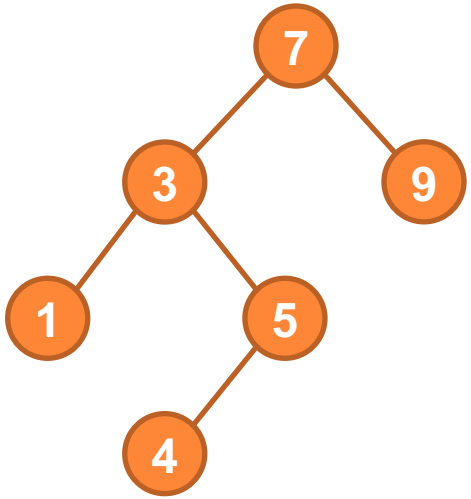
- Comparison sort on n elements
- Read and comparison (without writes): $\Omega(n \log n)$
- Write complexity: $\Omega(n)$

- 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

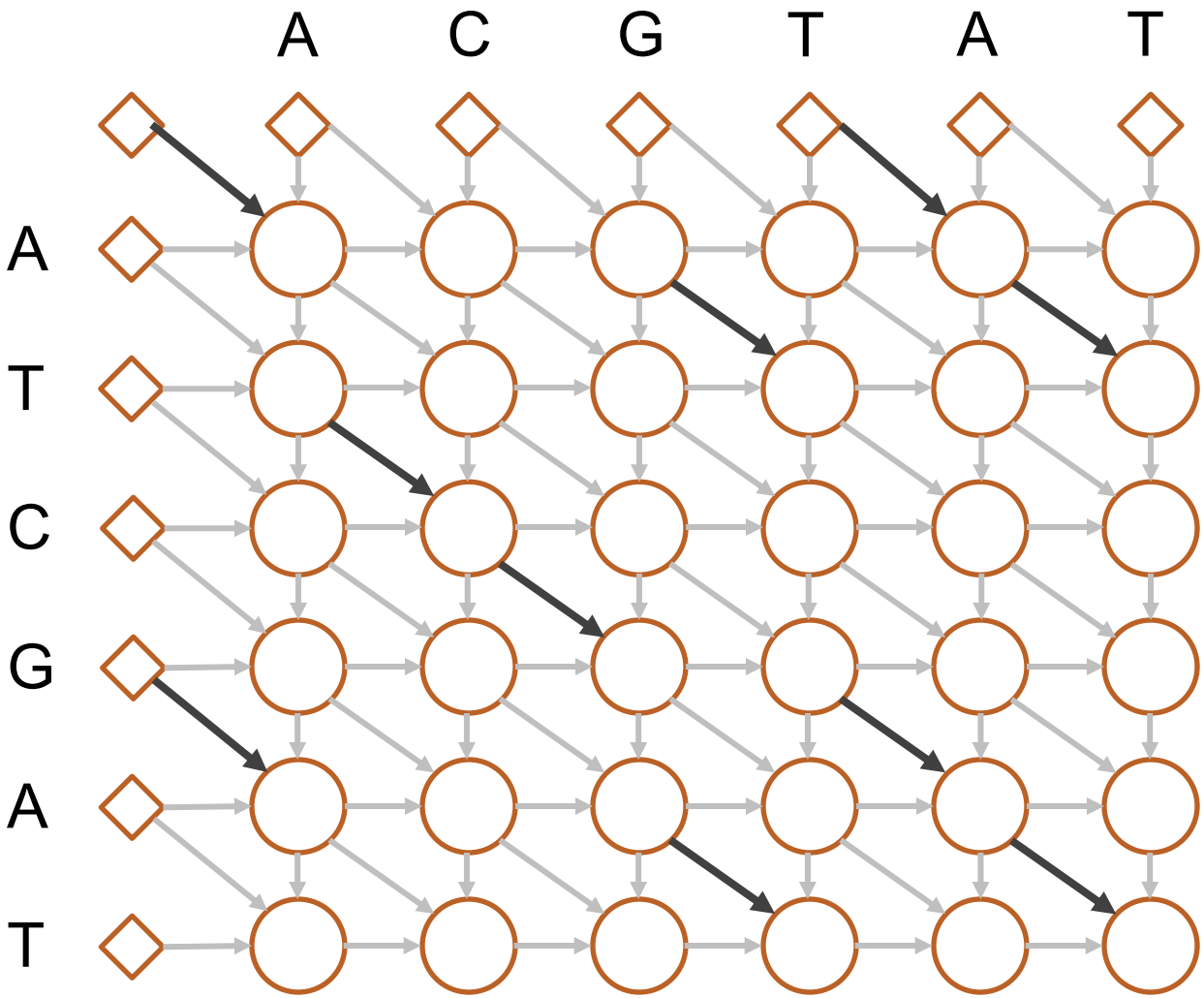
$M = O(1)$

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	$\Theta(m + n\omega)$	$O(m/n)$

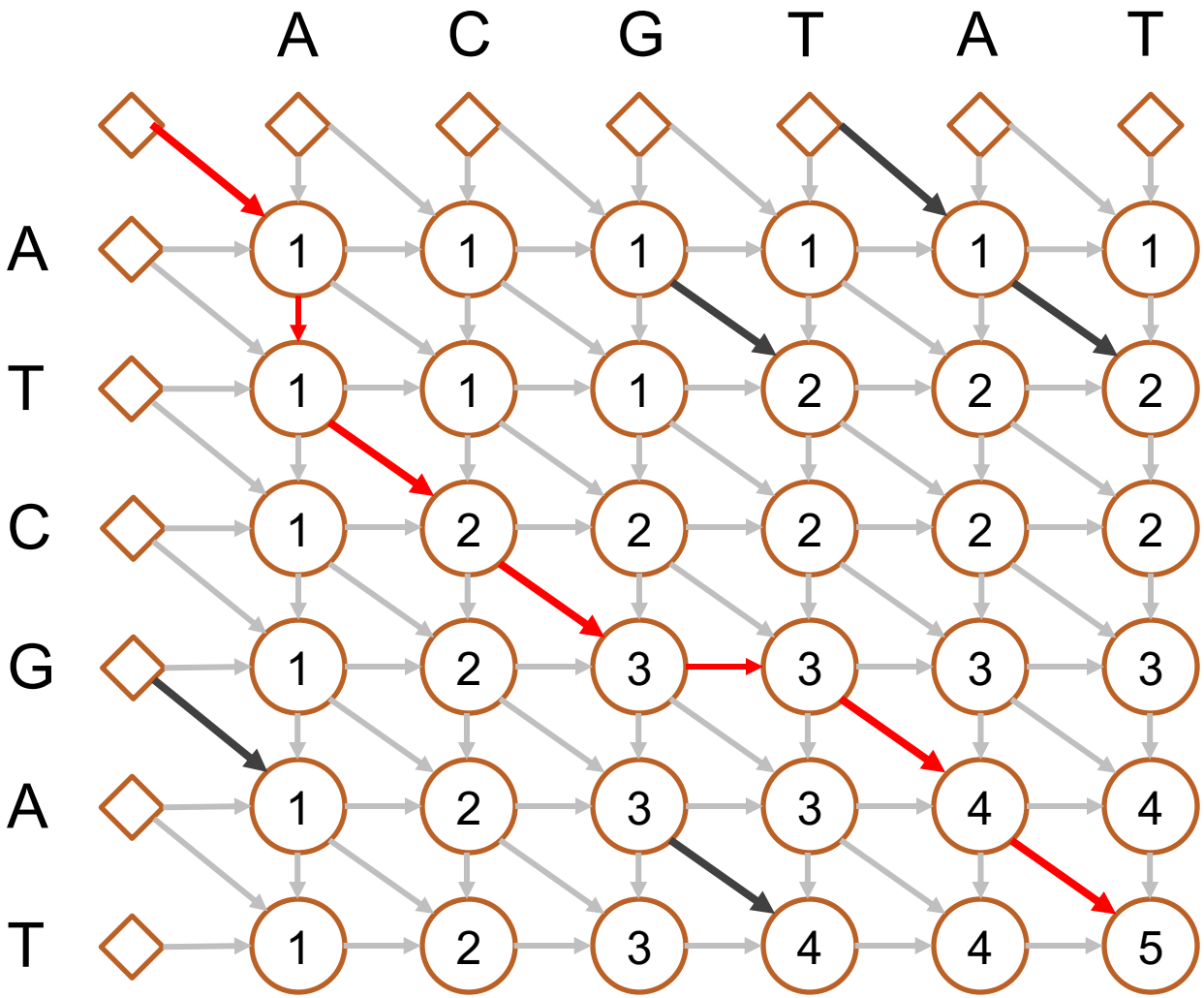
Lower bounds

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)



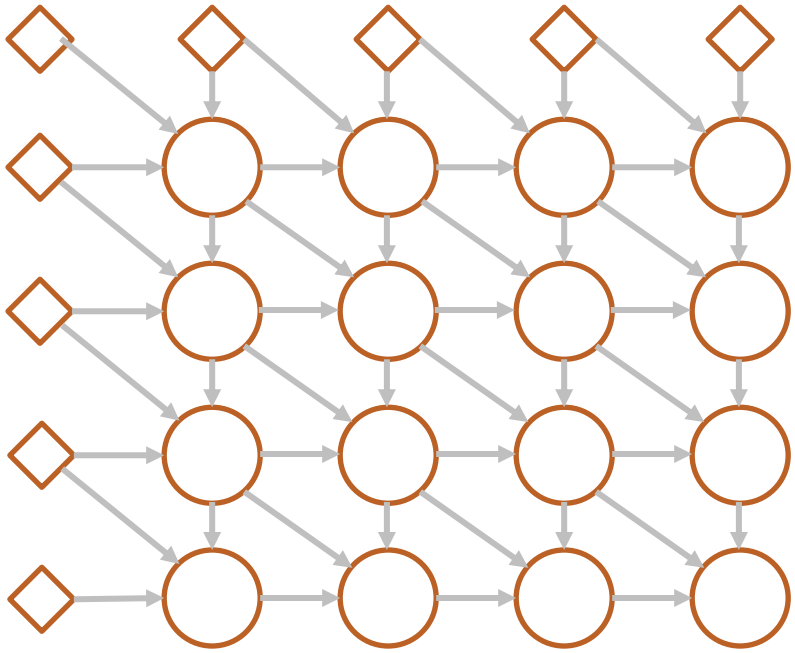
An example of Diamond DAG: Longest common sequence (LCS)



Computation DAG Rule

DAG Rule / pebbling game:

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

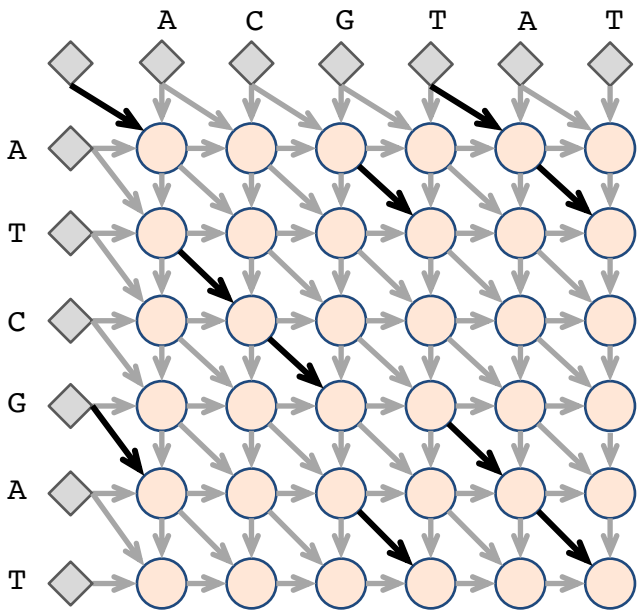


High-level proof idea

- 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

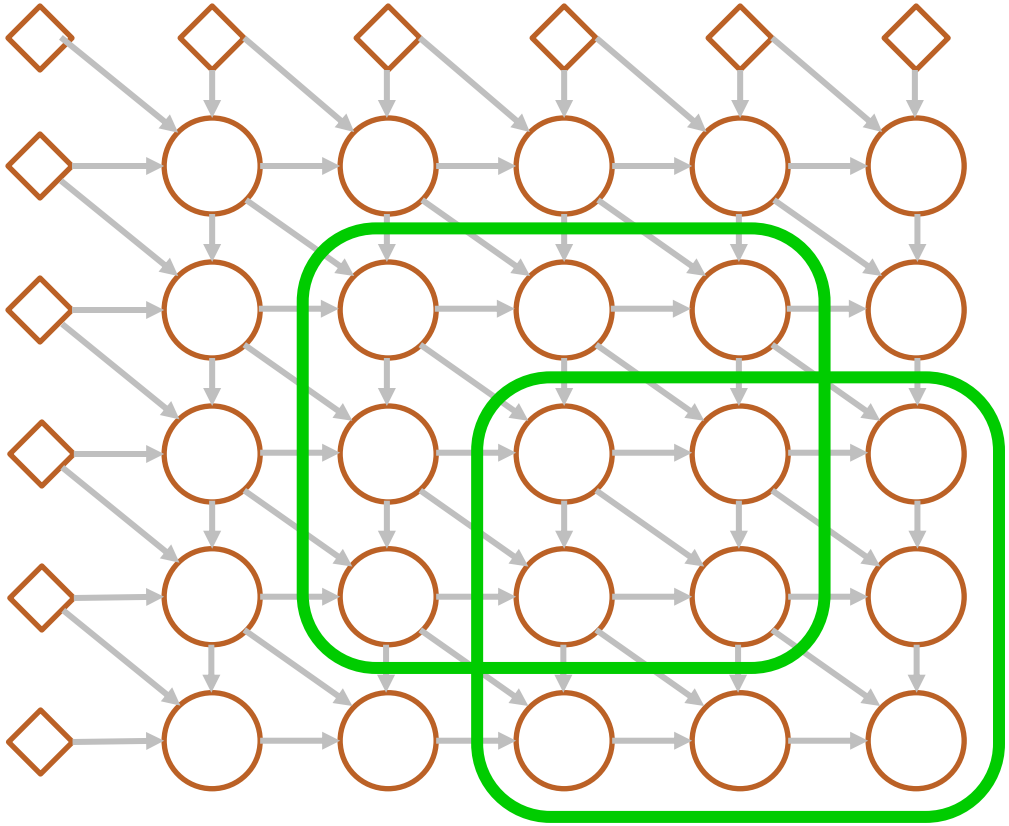
- (Input size is only $2n$)
- Building table explicitly ⇒ n^2 writes,
- but problem only inherently requires writing last value

- 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

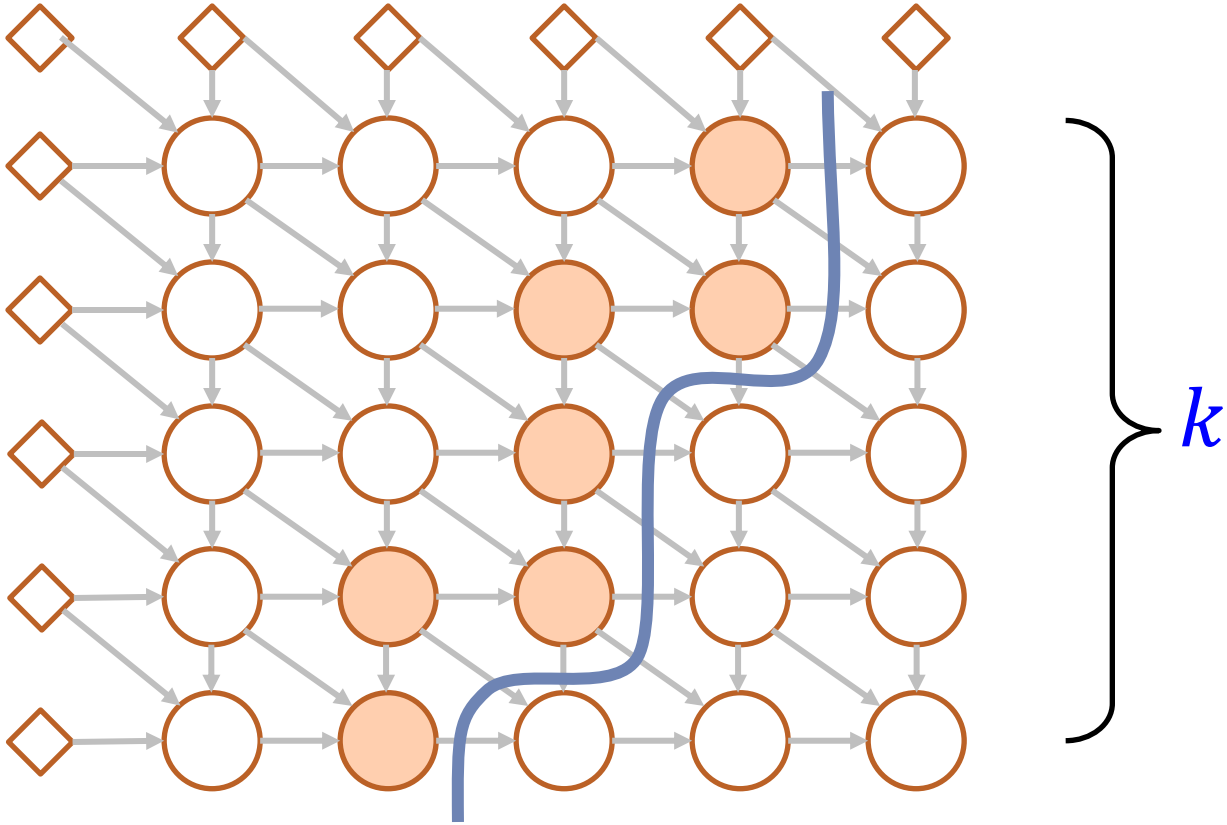


For $k > M$, some values need to be written out

Storage lower bound of subcomputation

(diamond DAG rule, Cook and Sethi 1976):

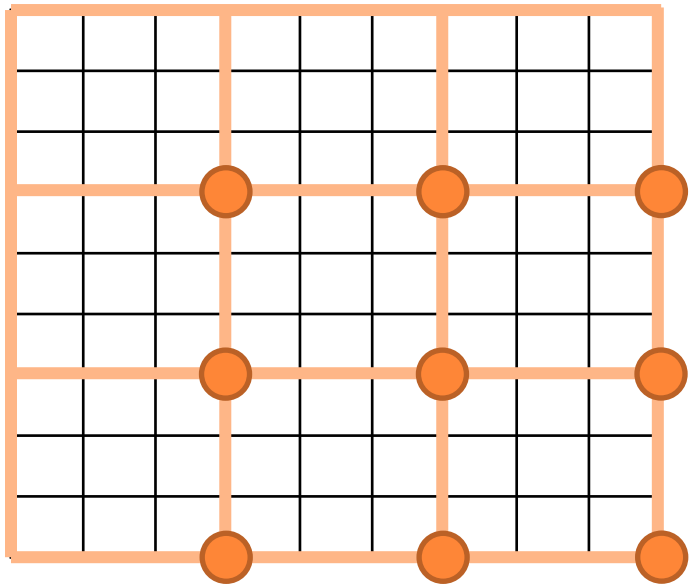
Solving an $k \times k$ sub-DAG requires k space to store intermediate value



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

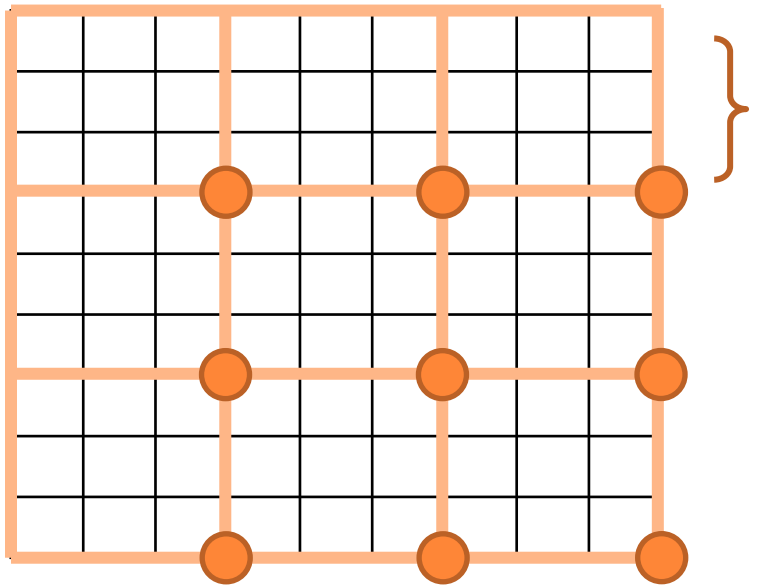


$$\Theta\left(\frac{n^2}{M^2}\right) \cdot \Omega(M) = \Omega\left(\frac{n^2}{M}\right)$$

#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



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

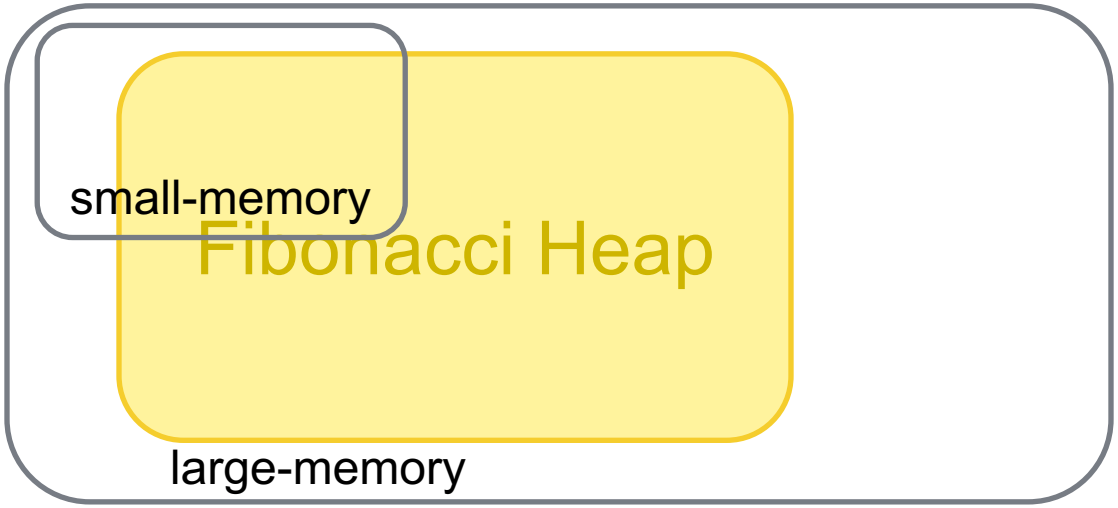
#row/column #read/write

Upper bounds on graph algorithms

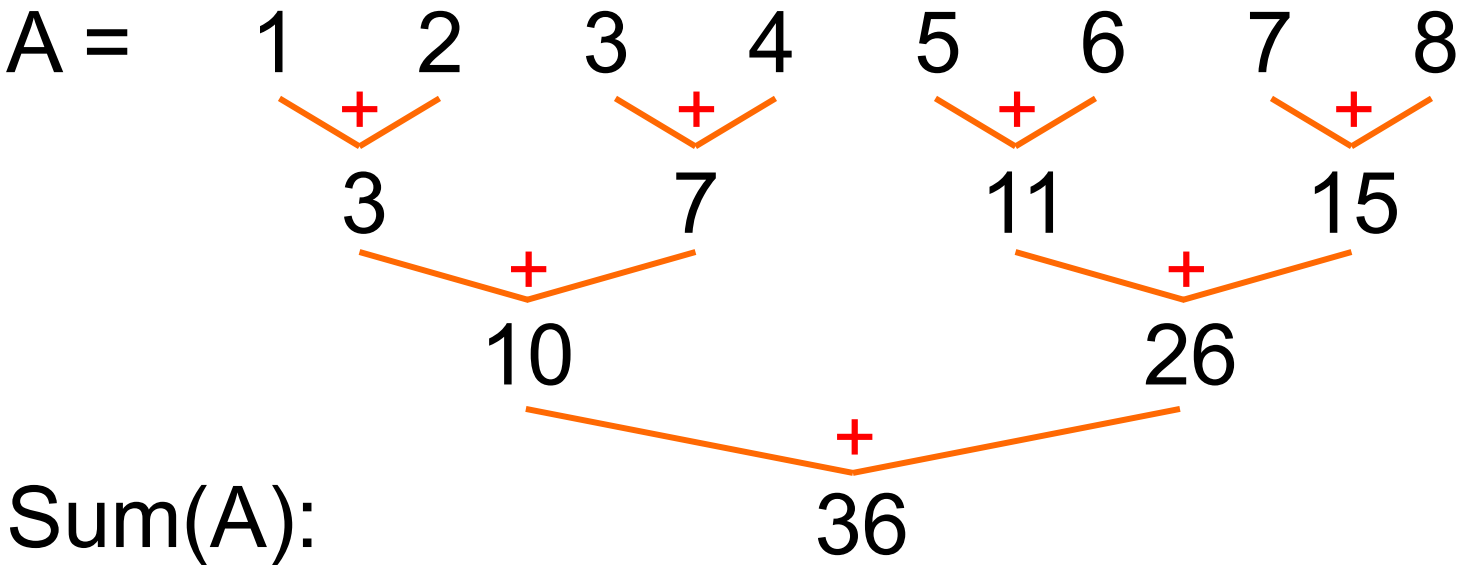
Problem	I/O cost $Q(n, m)$	
	Classic algorithms	New algorithms
Single-source shortest-path	$O\big(\omega(m + n\log n)\big)$	$O(\min(n(\omega + m/M),$ $\omega(m + n\log n),$ $m(\omega + \log n)))$
Minimum spanning tree	$O(m\omega)$	$O(\min(m\omega,$ $m\min(\log n, n/M) + \omega n))$

I/O cost of Dijkstra's algorithm

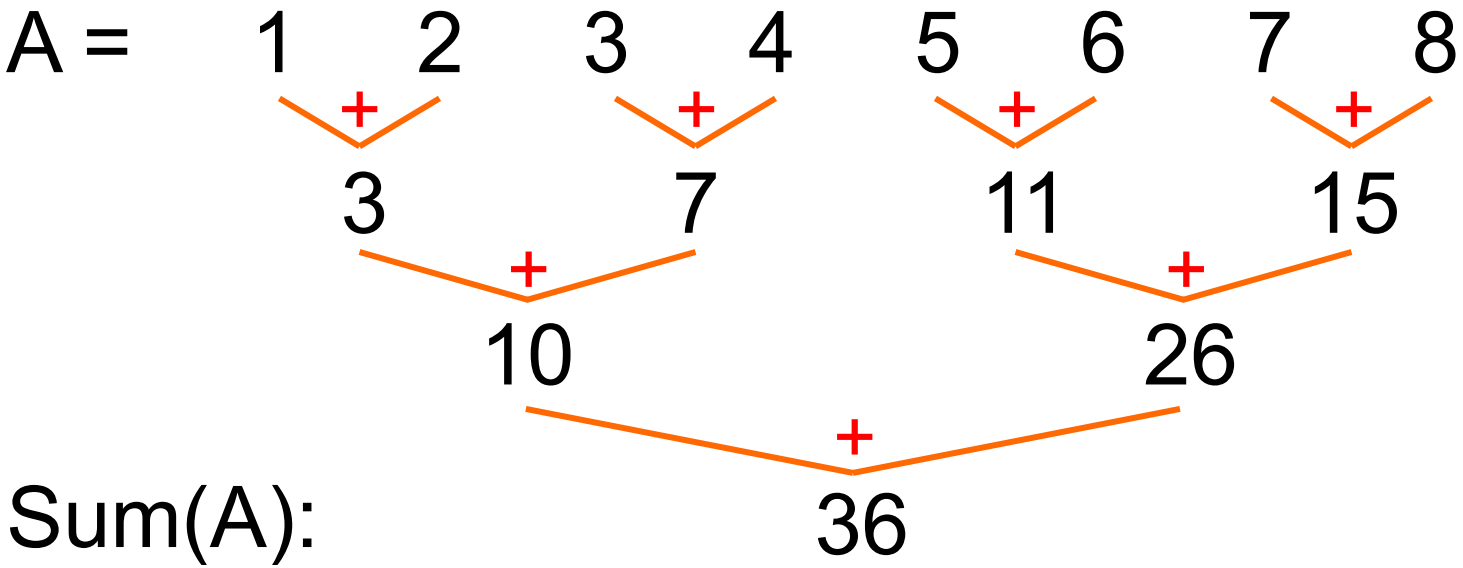
- Compute an SSSP requires $O(m)$ DECREASE-KEYs and $O(n)$ EXTRACT-MINs in Dijkstra's algorithm
  - Classic Fibonacci heap: $O\big(\omega(m + n\log n)\big)$
  - Balanced BST: $O\big(m(\omega + \log n)\big)$
  - Restrict the Fibonacci heap into the small-memory with size M: no writes to the large-memory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




```
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
```

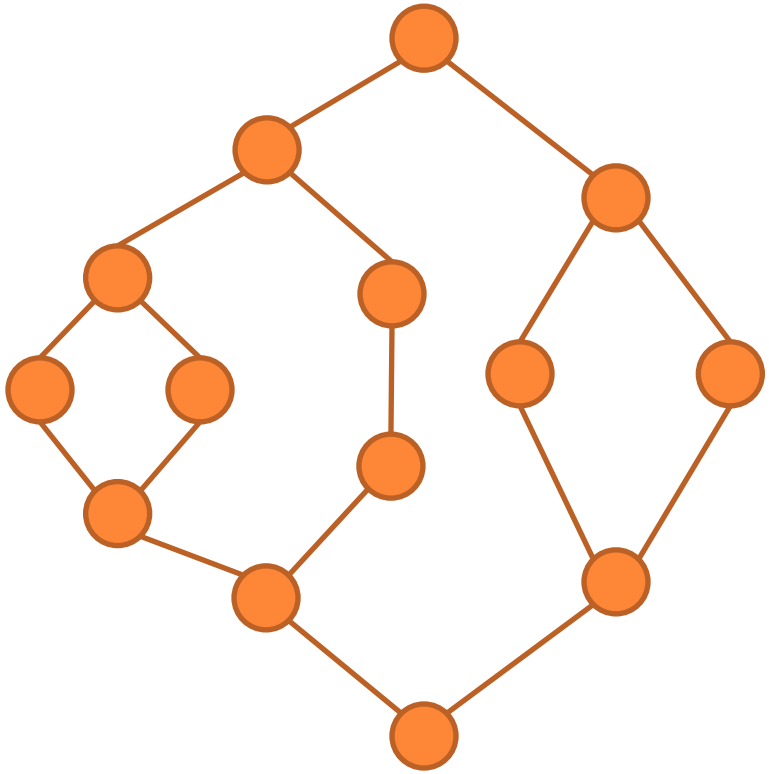


A work-stealing scheduler can run a computation on p cores using time:

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

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

The nested-parallel model



W (work): total computation

D (depth): longest chain of all paths

Using a work-stealing scheduler:

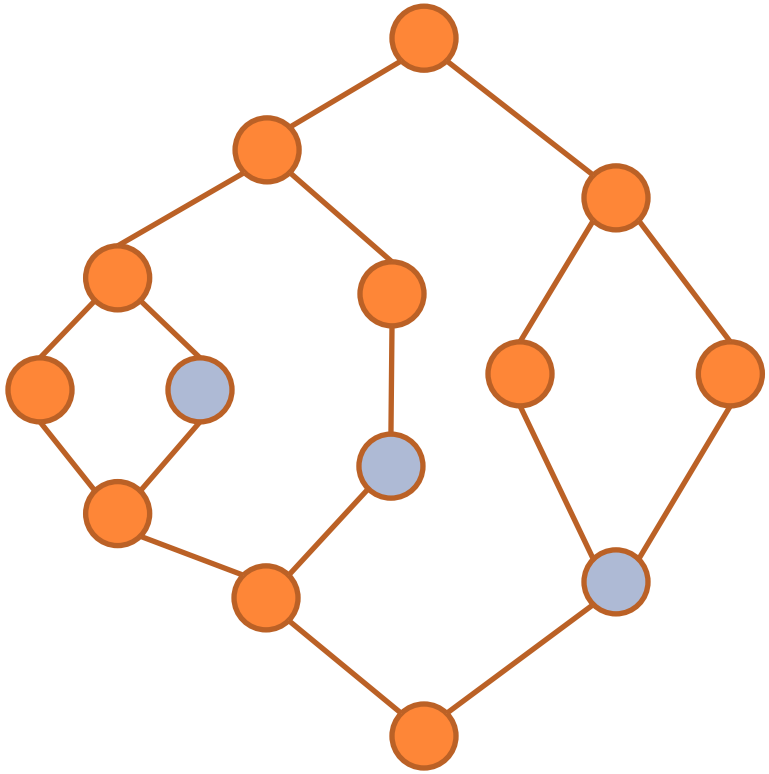
Extra work for scheduling (#steals):

$$O(pD)$$

Running time on p cores:

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

The asymmetric nested-parallel model



W (work): total computation w/ expensive writes

D (depth): longest unweighted chain of all paths

Using a work-stealing scheduler:

Extra work for scheduling (#steals):

$$O(pD)$$

Running time on p cores:

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

Results of parallel algorithms

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

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

Sequential upper bounds

$M = O(1)$

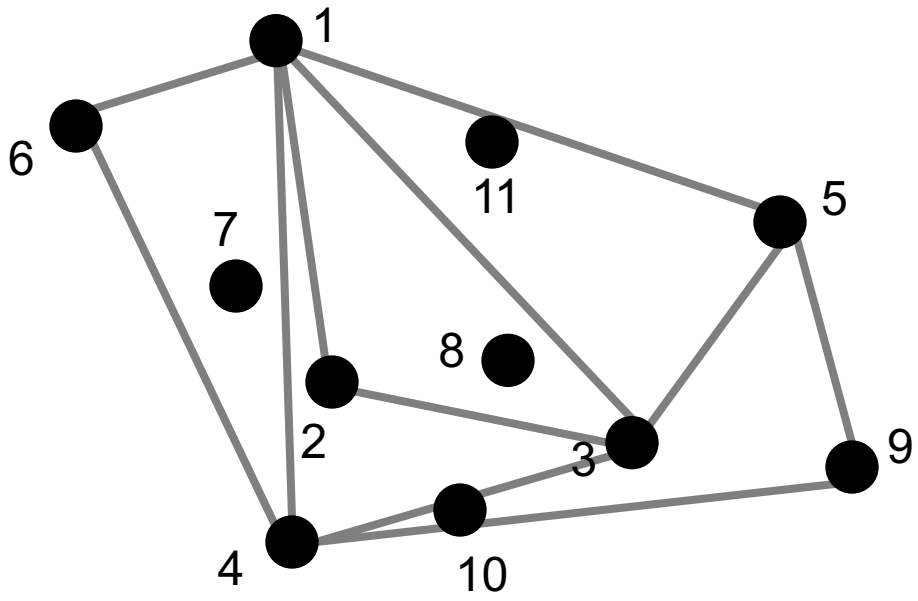
Problem	I/O cost $Q(n)$ and work $W(n)$	Reduction ratio
Comparison sort	$\Theta(n \log n + \omega n)$	$O(\log n)$
Search tree, priority queue	$\Theta(\log n + \omega)$	$O(\log n)$
2D convex hull, triangulation	$O(n \log n + \omega n)$	$O(\log n)$
BFS, DFS, SCC, topological sort, block, bipartiteness,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

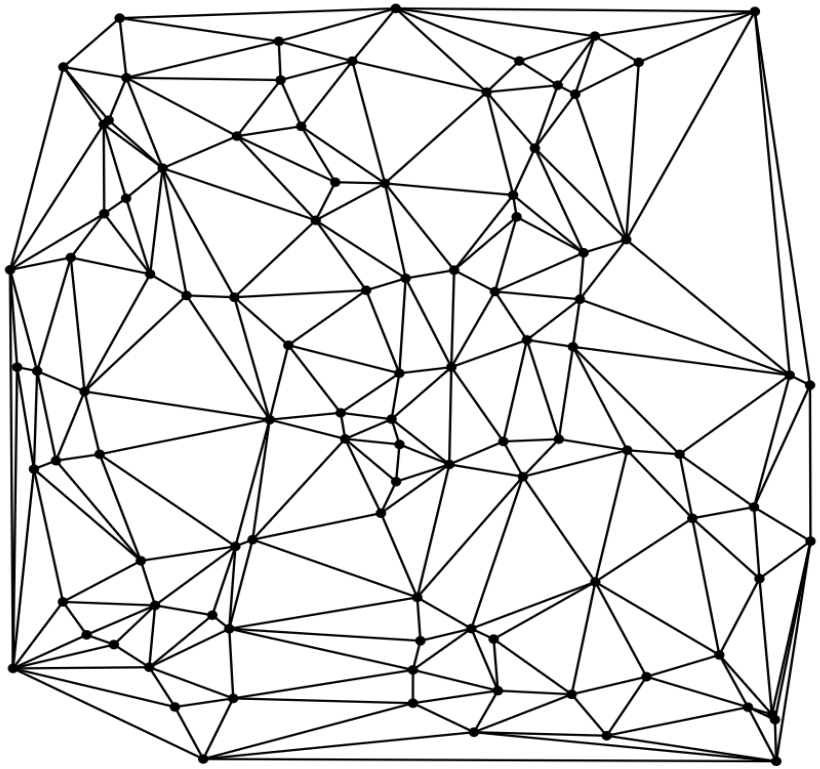
Randomized Incremental Algorithms

- 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

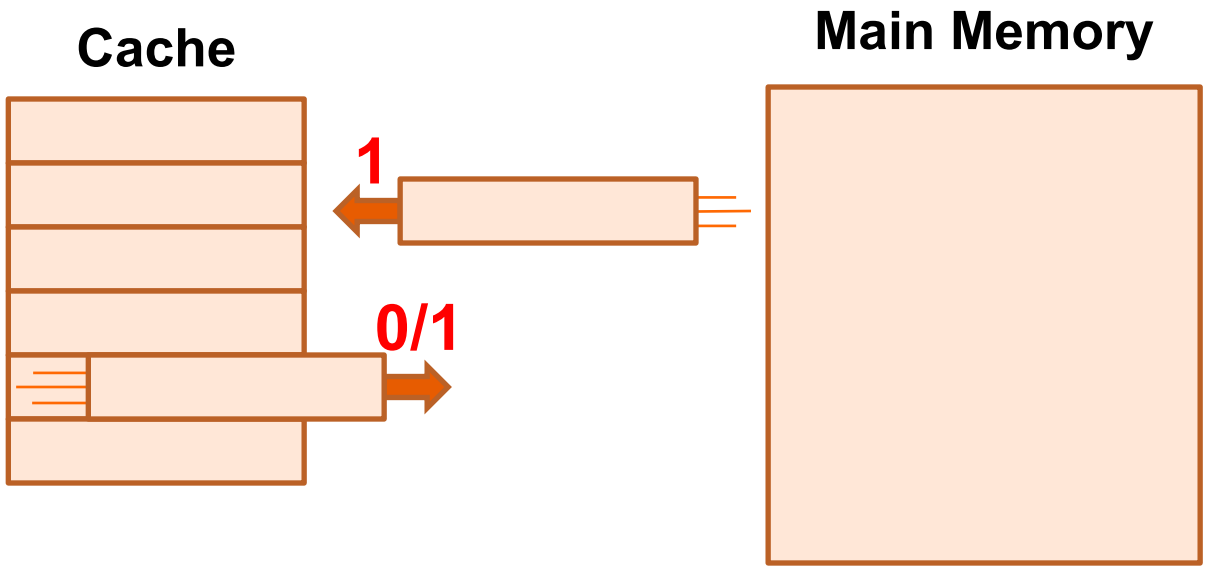
- Unfortunately, good parallel incremental algorithms for some geometry problems were unknown
- We describe a framework that can analyze the parallelism of many incremental algorithms
- Then we design a uniform approach to get the write-efficient versions of these algorithms

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

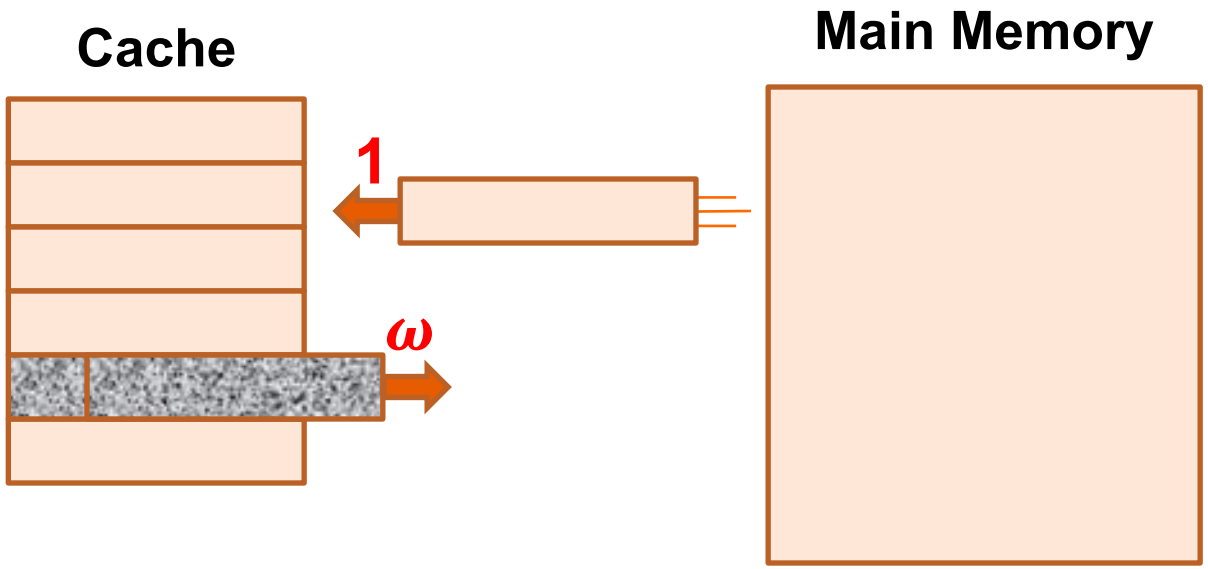
Cache policy: decide the block to evict when a cache miss occurs

- 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:

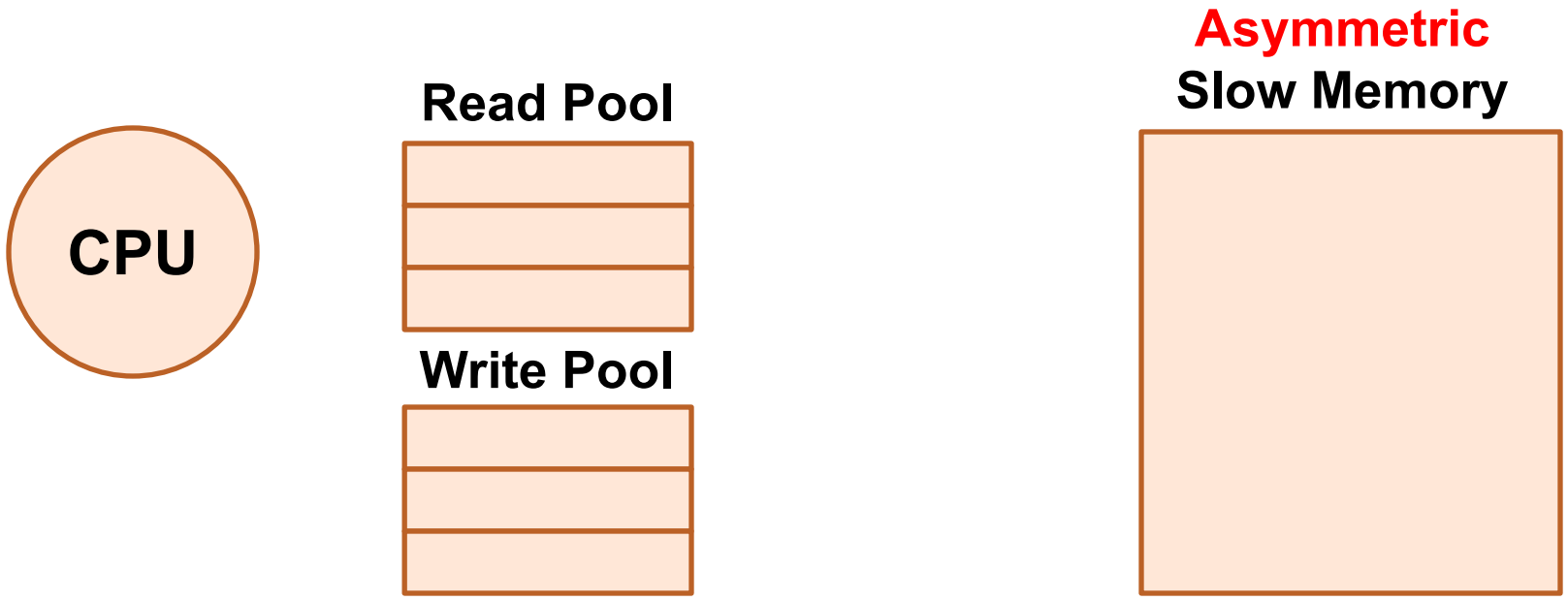
  W(1), W(2), … , W(k-1), R(k), R(k+1), …, R(2k+1)

- 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



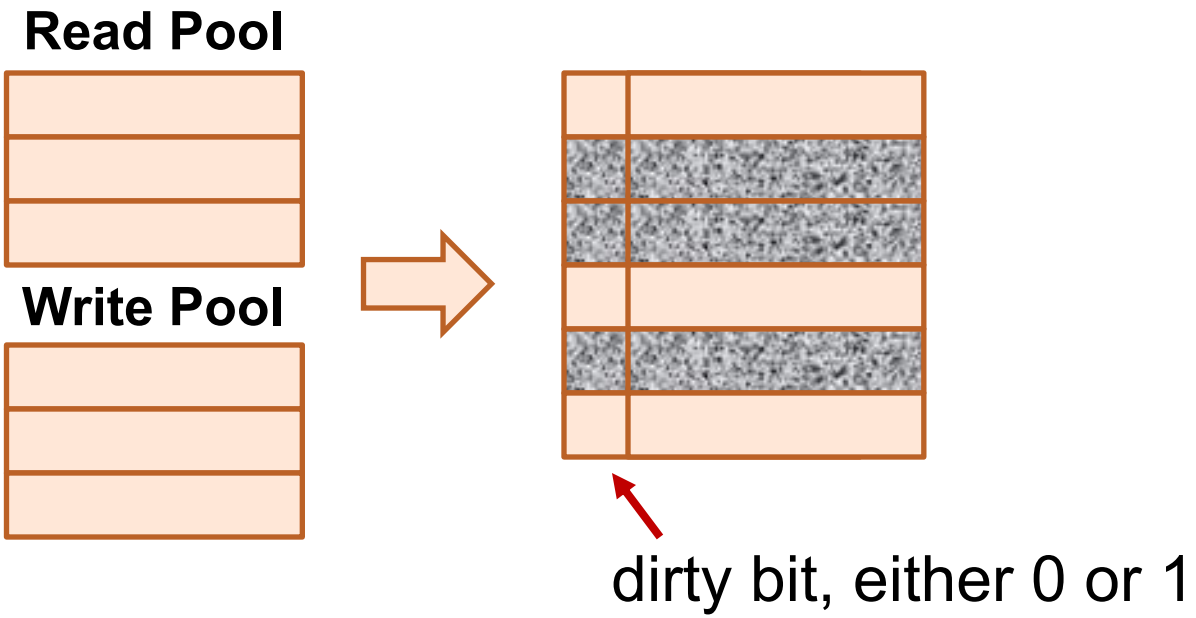
Solution: The Asymmetric LRU policy

- 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



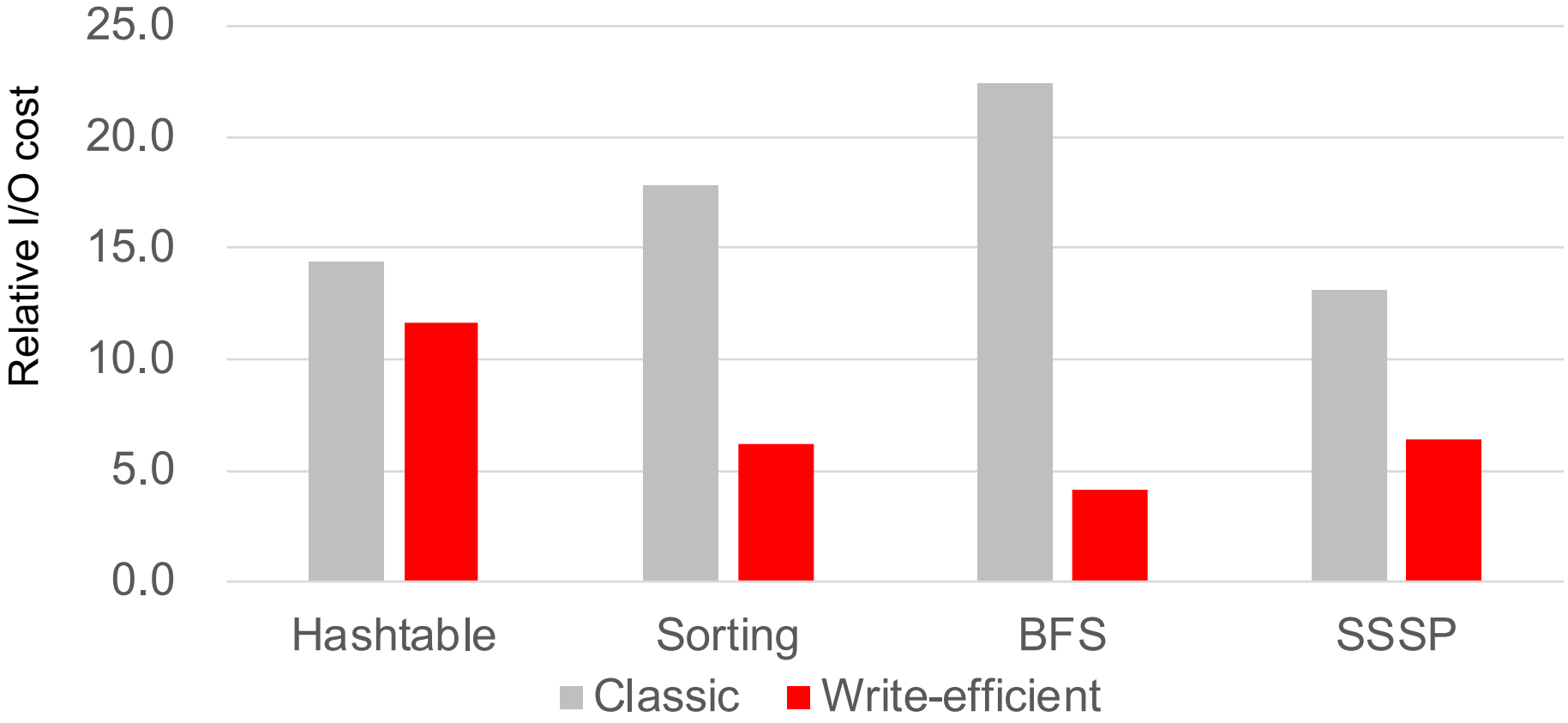
Experiment Analysis

Software simulator [ESA18]

- Can measure the number of reads and writes of an algorithm

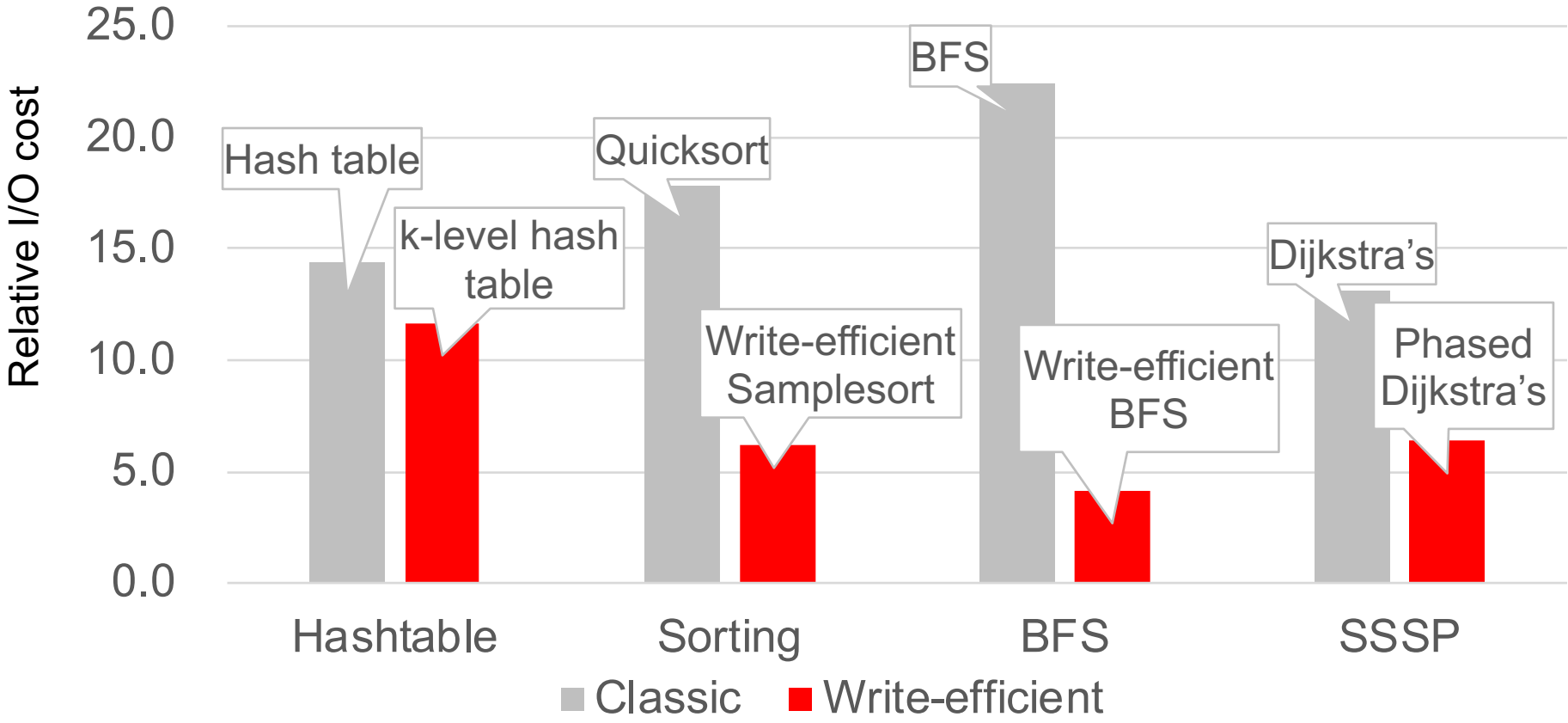
Experimental results

- Weighted reads and writes assuming writes are 6x more expensive



Experimental results

- Weighted reads and writes assuming writes are 6x more expensive



Summary

Summary

- The new NVRAMs are available, 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
- New implementation with better performance
- This area is still new — there are many other problems worth investigating