How to Sort 100TB of Data: Algorithm Design for Massive Datasets

Prof. Ke Yi

Hong Kong University of Science and Technology

Sort Benchmark Home Page

New: We are happy to announce the 2019 winners listed below. The new, 2019 records are listed in green. Congratulations to the winners!

Background

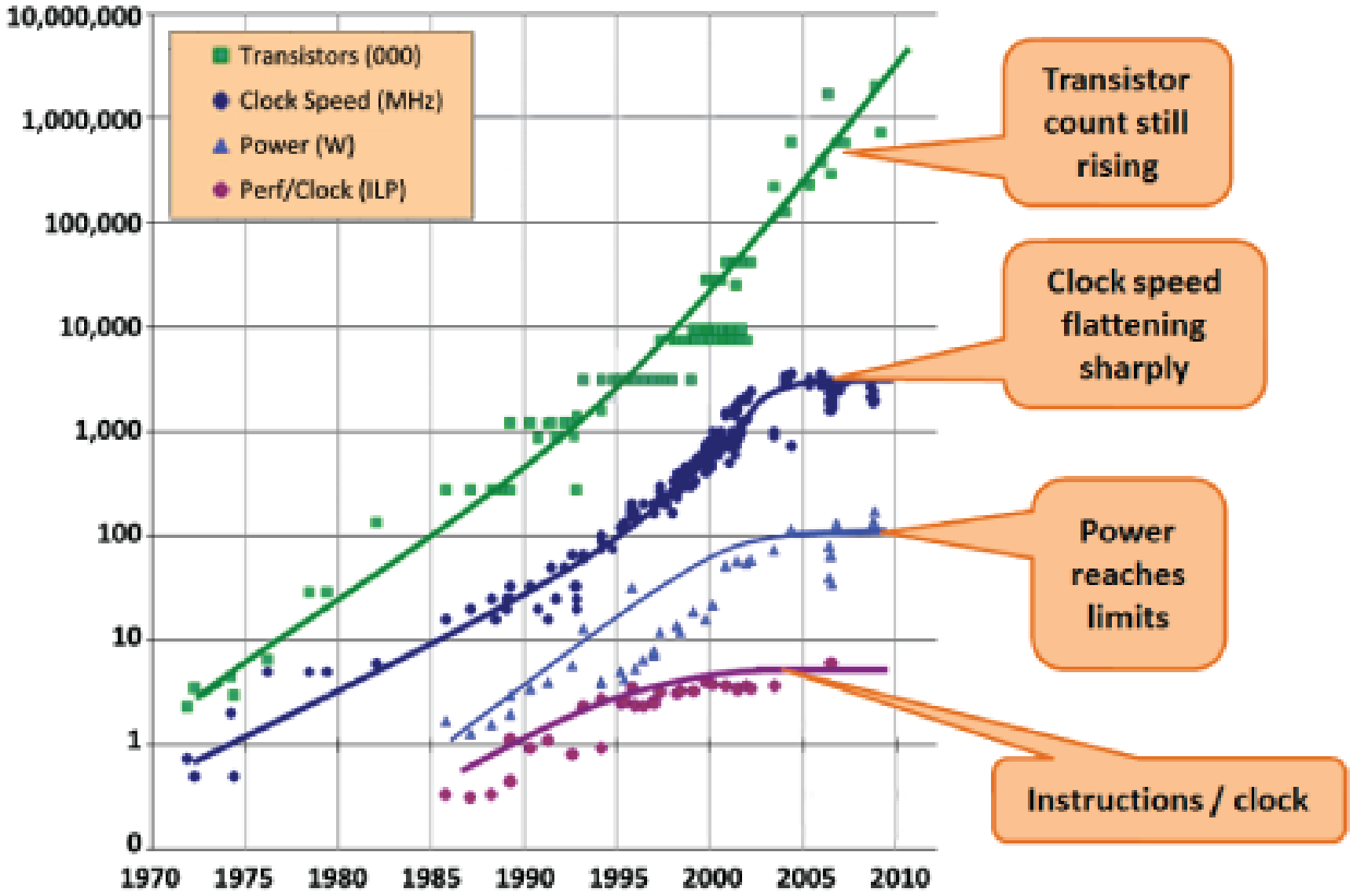
Until 2007, the sort benchmarks were primary defined, sponsored and administered by Jim Gray. Following Jim's disappearance at sea in January 2007, the sort benchmarks have been continued by a committee of past colleagues and sort benchmark winners. The Sort Benchmark committee members include:

- Chris Nyberg of Ordinal Technology Corp
- Mehul Shah of Amazon Web Services
- Naga Govindaraju of Microsoft

Top Results

	Daytona	Indy
Gray	2016, 44.8 TB/min **Tencent Sort** 100 TB in 134 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub	2016, 60.7 TB/min **Tencent Sort** 100 TB in 98.8 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub
Cloud	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.

As Transistor Count Increases, Clock Speed Levels Off



Source: Intel

Going Parallel/Distributed is the Only Way to Scale

The Frustration of Parallel Programming

- Race conditions

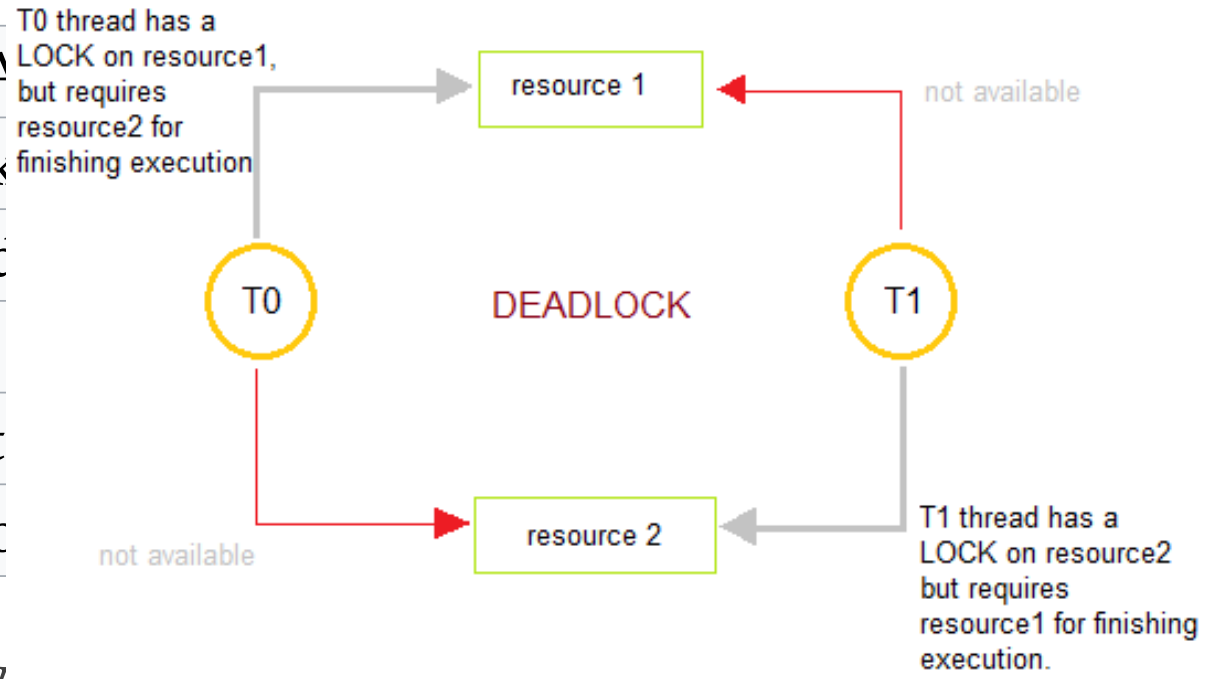
Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A: Write back to variable V	3B: Write back to variable V

- Intended result: add 2 to V
- But, what if 1A is executed between 1B and 3B?

The Frustration of Parallel Programming

- Use locks

Thread A
1A: Lock
2A: Read
3A: Add
4A: Writ
5A: Unlc

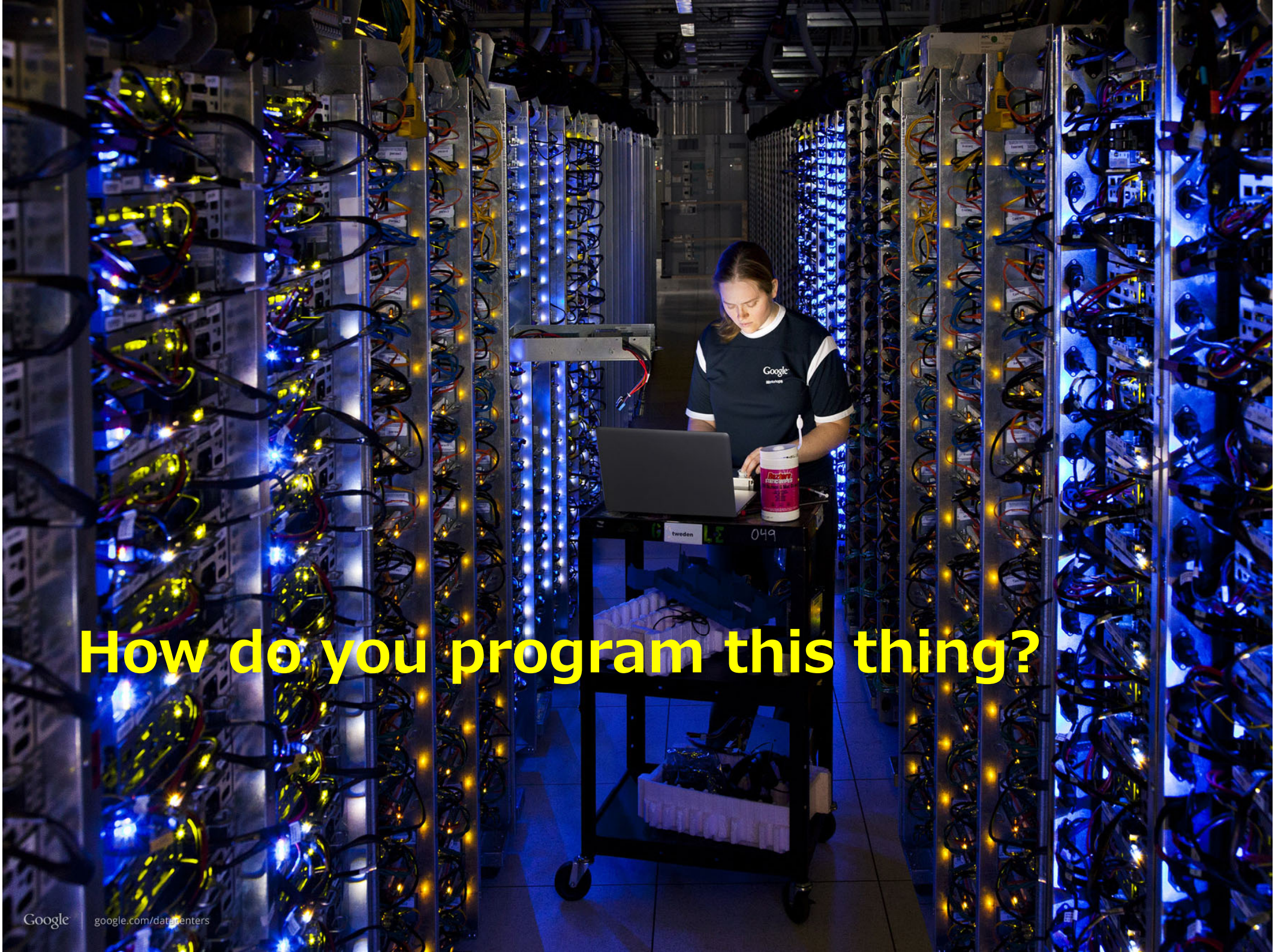


ble V

- How

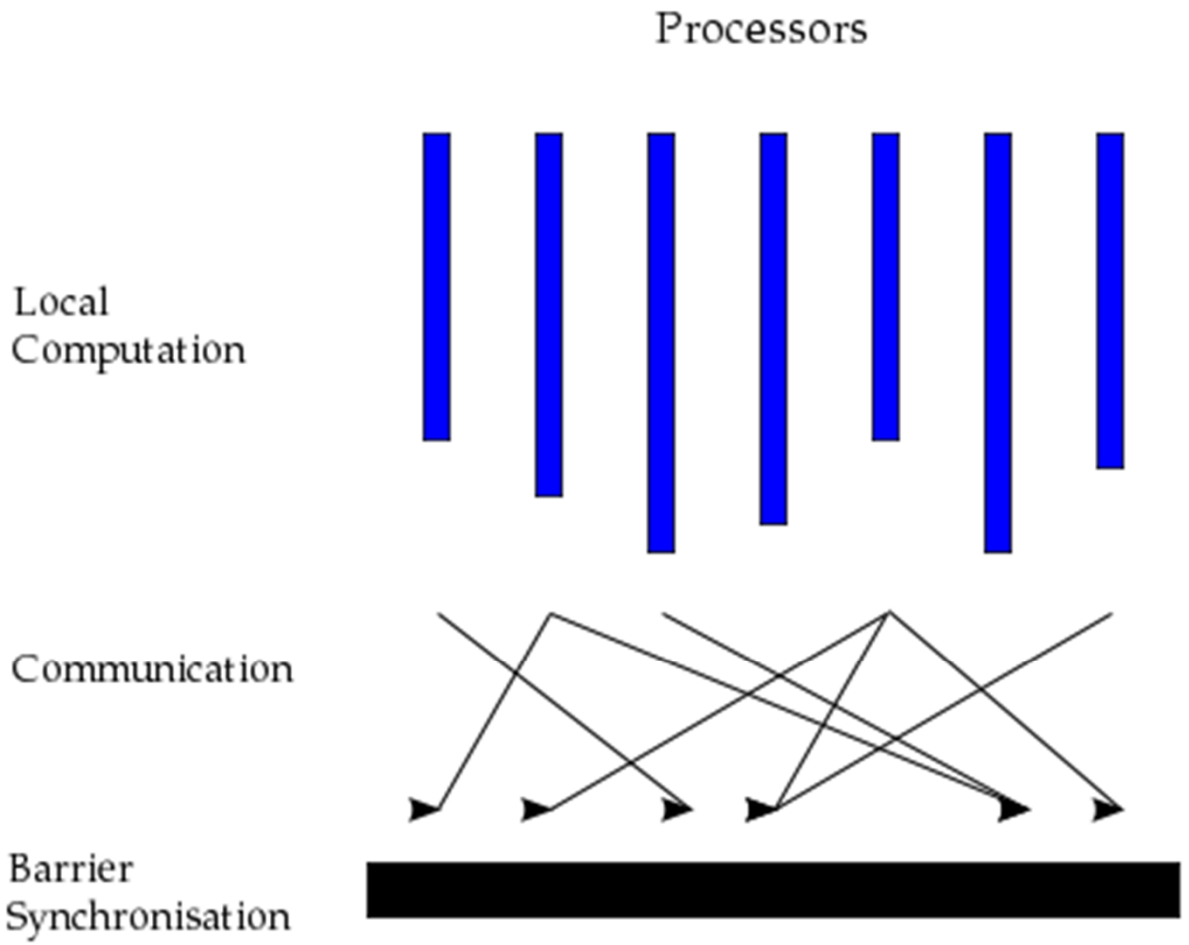
The Frustration of Parallel Programming

- Hard to debug: Race conditions and deadlocks are nondeterministic
- Most programming languages are low-level
  - The programmer needs to manage shared memory and/or communication
  - OpenMP is a good step forward, but still difficult for most programmers
- Programs written for multi-cores do not easily carry over to clusters

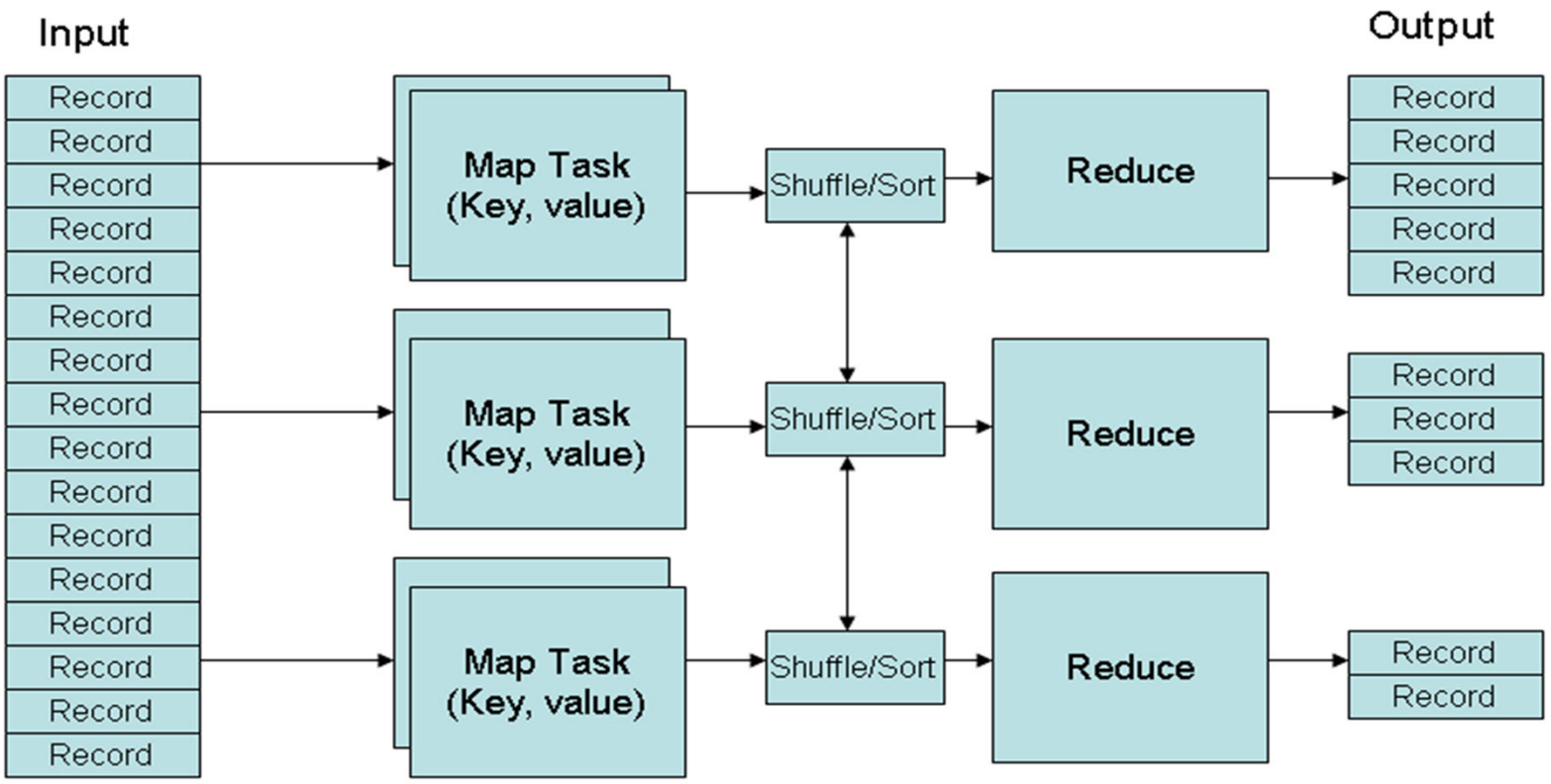

Google
tweden
049
How do you program this thing?
Google
google.com/datacenters

Valiant's BSP Model (1990)

Bulk Synchronous Parallel



MapReduce

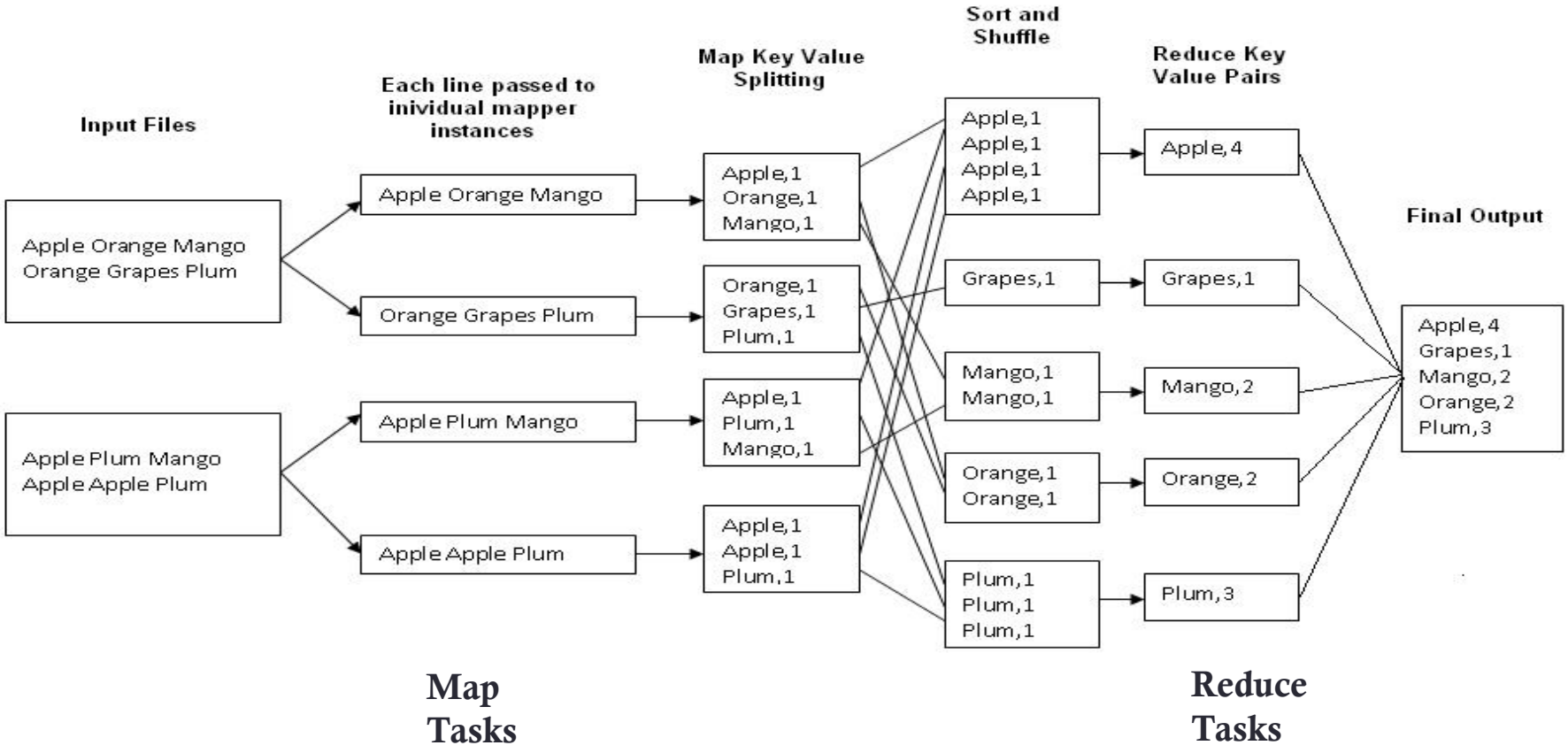


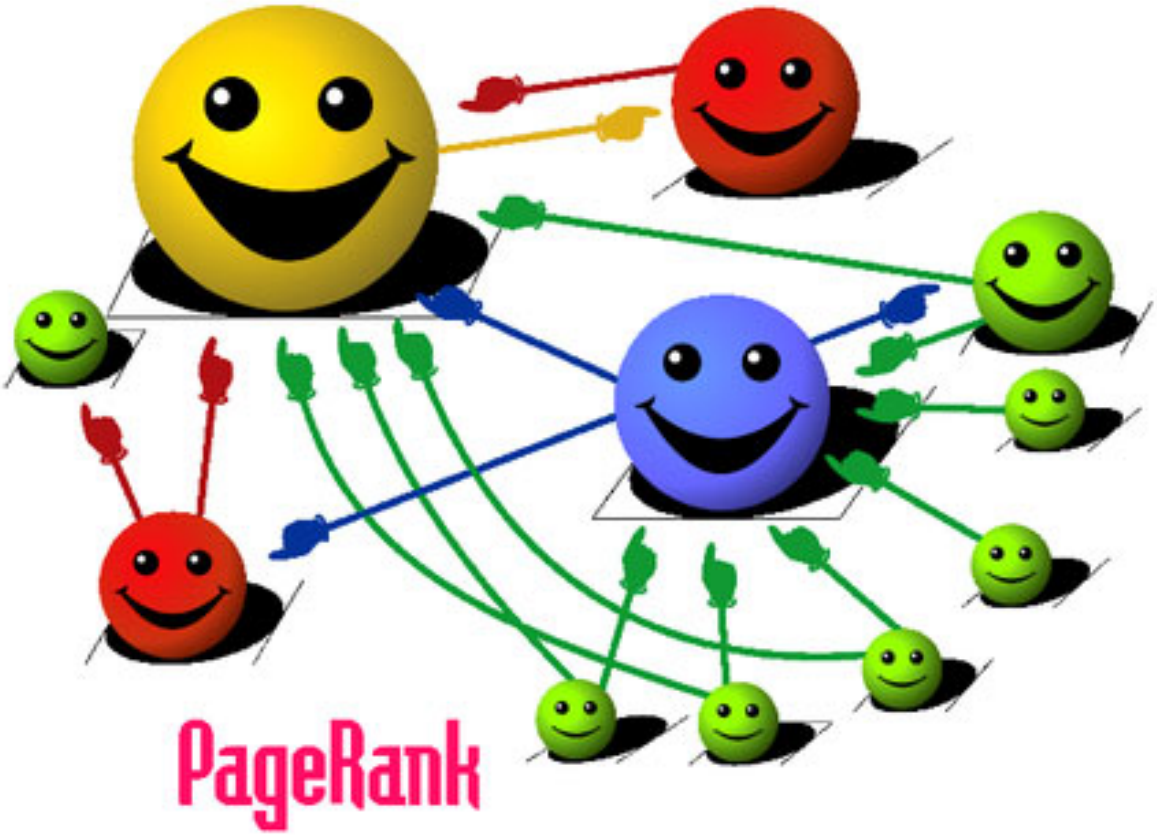
(key, value) pairs are used as the format for both data and intermediate results

(key, value) pair is sent to worker hash(key) mod p) by the shuffling stage

Example 1: Word Count

- **Job: Count the occurrences of each word in a data set**





Cartoon illustrating basic principle of PageRank. The size of each face is proportional to the total size of the other faces which are pointing to it.

Example 2: PageRank

- Algorithm:
  - Initialize all PR's to 1
  - Iteratively compute

$$PR(u) \leftarrow 0.15 + 0.85 \times \sum_{v \to u} \frac{PR(v)}{\text{outdegree}(v)}$$

- Data stored in adjacency list format: (src, PR, dst_1, dst_2, …)
- How to define the map and reduce function?
- Map:
  - (src, PR, dst_1, dst_2, …) → (dst_i, PR/outdegree(src)), $i = 1, 2, \ldots$
  - (src, PR, dst_1, dst_2, …) → (src, dst_1, dst_2, …) //can be optimized
- Reduce:
  - (dst, c_1,) + (dst, c_2) + … + (src, dst_1, dst_2, …) → (src, $0.15 + 0.85 \times \sum_i c_i$, dst_1, dst_2, …)

Performance Measurement

- Number of rounds
  - Ideally, a constant
  - $\log N$ is also tolerable
  - Wordcount: 1
  - Pagerank: 1 per iteration
- Maximum amount of work of a worker in a round
  - Wordcount: $O(N/p)$ assuming no skew in data
  - Pagerank: $O(V \cdot d_{max}/p)$, V: # vertices, d_{max}: max degree
- Space needed by each worker
  - Wordcount: $O(1)$, Pagerank: $O(1)$
- Total amount of work of all workers
  - Wordcount: $O(N)$ assuming no skew in data
  - Pagerank: $O(V \cdot d_{ave}) = O(E)$, E: # edges

Technique 1:
Divide and Conquer

Classical Divide-and-Conquer

- Classical D&C
  - Divide problem into 2 parts
  - Recursively solve each part
  - Combine the results together
- D&C under big data systems
  - Divide problem into p partitions, where (ideally) p is the number of executors in the system
  - Solve the problem on each partition
  - Combine the results together
- Example: sum(), reduce()

Prefix Sums

- Input: Sequence *x* of *n* elements, binary associative operator +
- Output: Sequence *y* of *n* elements, with $y_k = x_1 + \ldots + x_k$
- Example:

  x = [1, 4, 3, 5, 6, 7, 0, 1]

  y = [1, 5, 8, 13, 19, 26, 26, 27]
- Algorithm:
  - Compute sum for each partition
  - Compute the prefix sums of the p sums
  - Compute prefix sums in each partition
- $O(1)$ rounds, $O\left(\frac{N}{p}\right)$ work per worker, $O(1)$ space
  - Note: Master node needs to do $O(p)$ work.
  - Assume $p \ll N$

Variants of Prefix Sums

- Assign consecutive id's for each element
  - zipWithIndex()
- Given a list of words, find the first appearance of "spark"
- Given two long strings, compare them lexicographically
- Given a sequence of integers, check whether these numbers are monotonically decreasing.

Sorting (Sample Sort)

- Step 1: Sampling
  - Master collects a sample of sp elements (will determine s later)
- Step 2: Choose splitters
  - Master picks every $(i \cdot s)$-th element in the sample as splitters, $i = 1, \ldots, p-1$
  - Broadcast them to all workers
- Step 3: Shuffling
  - Each worker partitions its data using the splitters
  - Send data to the target machine
- Step 4: Sort each partition
  - Each machine sorts all data received

Determining Sample Size

- Goal: No machine receives more than $(1+\epsilon)\frac{N}{p}$ elements w.h.p.
  - How large should s be?
- Let the elements be $a_1, \ldots, a_N$ in sorted order
- A sub-sequence $a_i, \ldots, a_{i+(1+\epsilon)\frac{N}{p}}$ is bad if it contains $< s$ sampled elements
  - Goal achieved if no sub-sequence is bad
- Consider a particular sub-sequence
  - $X =$ # sampled elements in it; $E[X] = \frac{sp}{N} \cdot (1+\epsilon)\frac{N}{p} = (1+\epsilon)s$
  - By Chernoff inequality: $\Pr[X < s] \leq \Pr\left[X < \left(1-\frac{\epsilon}{2}\right)E[X]\right] \leq e^{-\Omega(\epsilon^2 s)}$
- By union bound, $\Pr[\exists \text{ a bad subsequence}] \leq N \cdot e^{-\Omega(\epsilon^2 s)}$
  - It suffices to set $s = O\left(\frac{1}{\epsilon^2} \cdot \log N\right)$
  - Can you improve the $\log N$ term to a $\log \frac{p}{\epsilon}$?

Distributed Sampling

- Q: How to sample one element uniformly from n elements stored on p servers?
- A:
  - First randomly sample a server
  - Then ask that server to return an element randomly chosen from its N/p elements.
  - The probability of each element being sampled is $\frac{1}{p} \cdot \frac{p}{N} = \frac{1}{N}$
- Q: How to sample many elements at once?
- A: Do each of the two steps above in batch mode
  - First sample sp servers with replacement (this can be done at the master node).
  - If a server is sampled k times, we ask that server to return k samples (with replacement) from its local data.

Sample Sort: Summary

- $O(1)$ rounds
- $O\left(\frac{N}{p}\log p\right)$ work per worker
  - $O\left(\frac{N}{p}\log\frac{N}{p}\right)$ if comparison-based sorting is used in last step
- $O\left(\frac{N}{p}\right)$ space per worker
- $O(N\log p)$ total work
  - $O(N\log N)$ if comparison-based sorting is used in last step
- Now, can you solve the word count problem on skewed data?

Technique 2: Streaming Algorithms

Majority

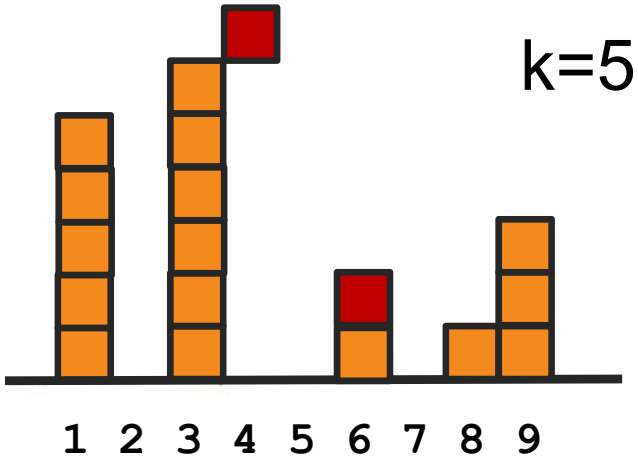
- Given a sequence of items, find the majority if there is one

- A A B C D B A A B B A A A A A A C C C D A B A A A
- Answer: A

- Trivial if we have O(n) memory
- Can you do it with O(1) memory and two passes?
  - First pass: find the possible candidate
  - Second pass: compute its frequency and verify that it is > n/2
- How about one pass?
  - Unfortunately, no

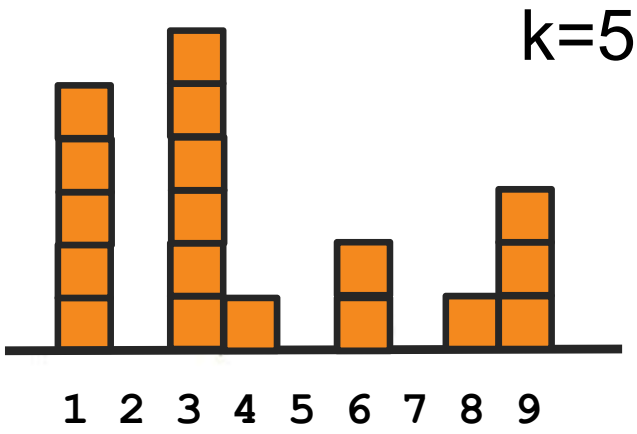
Heavy hitters

- Misra-Gries (MG) algorithm finds up to k items that occur more than 1/k fraction of the time in a stream
  - Estimate their frequencies with additive error $\leq$ N/(k+1)
- Keep k different candidates in hand. For each item in stream:
  - If item is monitored, increase its counter
  - Else, if < k items monitored, add new item with count 1
  - Else, decrease all counts by 1



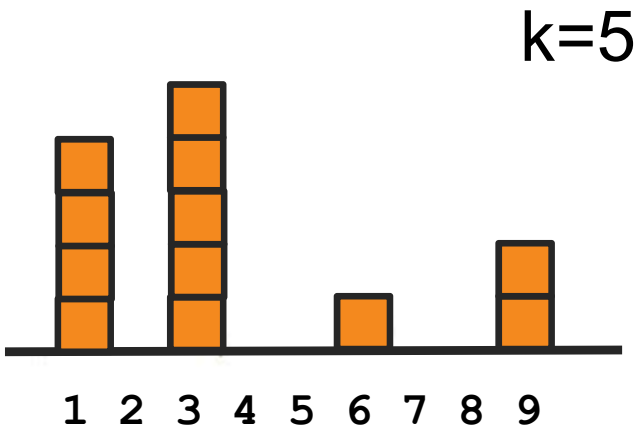
Heavy hitters

- Misra-Gries (MG) algorithm finds up to k items that occur more than 1/k fraction of the time in a stream
  - Estimate their frequencies with additive error $\leq$ N/(k+1)
- Keep k different candidates in hand. For each item in stream:
  - If item is monitored, increase its counter
  - Else, if < k items monitored, add new item with count 1
  - Else, decrease all counts by 1



Heavy hitters

- Misra-Gries (MG) algorithm finds up to k items that occur more than 1/k fraction of the time in a stream
  - Estimate their frequencies with additive error $\leq$ N/(k+1)
- Keep k different candidates in hand. For each item in stream:
  - If item is monitored, increase its counter
  - Else, if < k items monitored, add new item with count 1
  - Else, decrease all counts by 1



Streaming MG analysis

- N = total input size
- Error analysis
  - True count ∈ [counter, counter + # decrements]
  - Each decrement corresponds to deleting (k+1) distinct items from stream
  - At most N/(k+1) decrements on each unique key
  - So error ≤ N/(k+1)
- Note:
  - We can easily keep track of # decrements, so the actual error guarantee can be smaller than N/(k+1)
  - On real date sets, the true count is usually closer to the upper bound, i.e., counter + # decrements

Challenge: The Maximum Subarray Problem

Input: Profit history of a company of the years.

Year	1	2	3	4	5	6	7	8	9
Profit (M$)	-3	2	1	-4	5	2	-1	3	-1

Problem: Find the span of years in which the company earned the most

Answer: Year 5-8 , 9 M$

Formal definition:

Input: An array of numbers $A[1 \ldots n]$, both positive and negative

Output: Find the maximum $V(i,j)$, where $V(i,j) = \sum_{k=i}^{j} A[k]$

Challenge: Can you solve this problem in $O(1)$ rounds, $O(N/p)$ work per worker, and $O(1)$ space per worker?

Technique 3: Graph Algorithms

The Pregel Model for Graph Computation

- Vertex-centric computation
- The Pregel model
- Each vertex has a local value and a binary state (active/inactive)
  - In each round (superstep), each vertex executes a user-defined program:
    1. If active, the vertex sends messages to neighbors
    2. Aggregates messages (inactive vertices become active if messages are received)
    3. Updates local value and optionally set its state to inactive
  - Whole computation terminates when no active vertices

Example: PageRank

- Initialization:
  - local value = 1, status = active for all vertices
- User-defined program

```
for each neighbor v
  send_message(v, val / outdegree)

val = sum(all messages m received) * 0.85 + 0.15
if number of rounds > threshold:
  set status to inactive
```

Example: BFS

- Initialization:
  - local value = 0, status = active at starting vertex
  - local value = ∞, status = inactive at all other vertices
- User-defined program

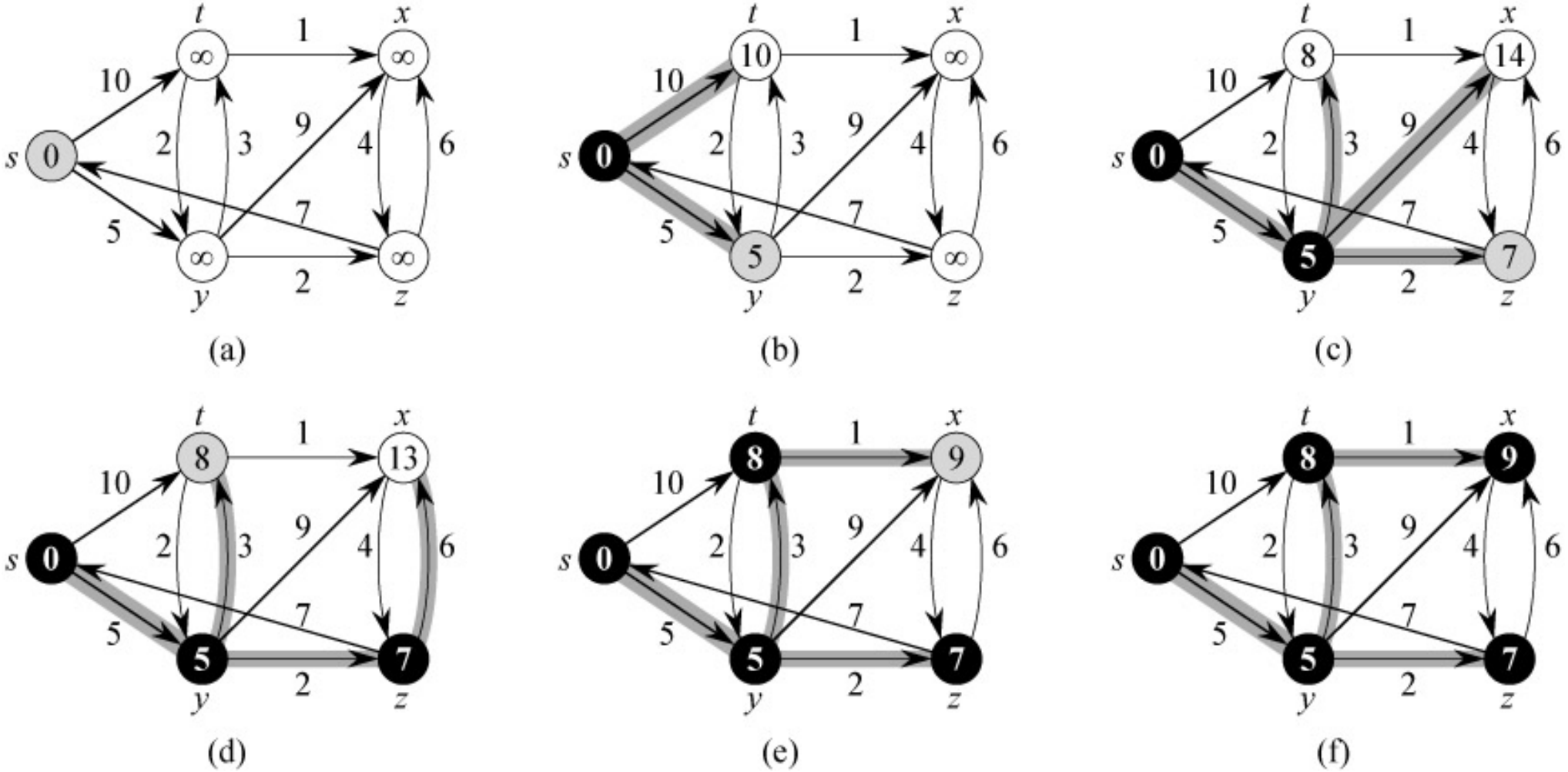
```
for each neighbor v
  send_message(v, val+1)

new_val = min(all messages m received)
if new_val < val then
  val = new_val
Else
  set status to inactive
```

Shortest Path: Dijkstra's Algorithm

```
Dijkstra(G, s):
for each v ∈ V do
    v.d ← ∞, v.p ← nil, v.color ← white
s.d ← 0
create a min priority queue Q on V with d as key
while Q ≠ ∅
    u ← Extract-Min(Q)
    u.color ← black
    for each v ∈ Adj[u] do
        if v.color = white and u.d + w(u, v) < v.d then
            v.p ← u
            v.d ← u.d + w(u, v)
            Decrease-Key(Q, v, v.d)
```

- This is an inherently sequential algorithm!

Dijkstra's Algorithm: Example



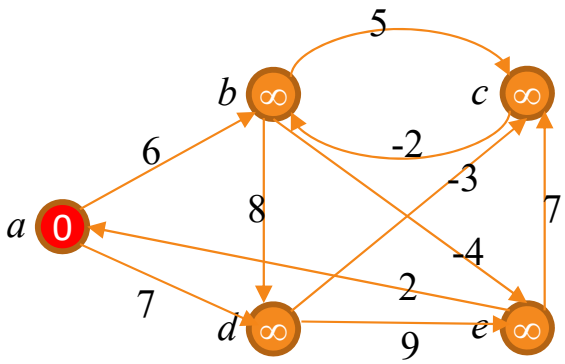
Note: All the shortest paths found by Dijkstra's algorithm form a tree (shortest-path tree).

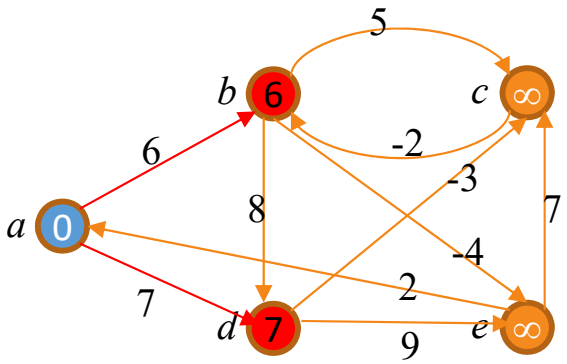
Bellman-Ford (implemented in GraphX / GraphFrames)

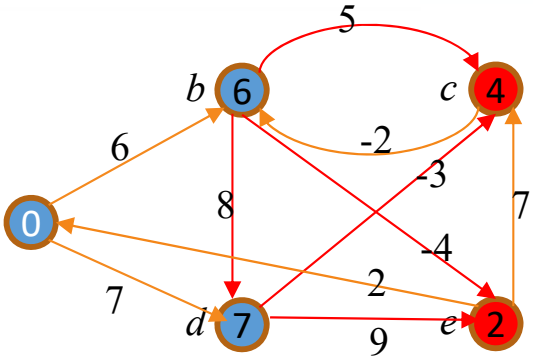
- Initialization:
  - local value = 0, status = active at starting vertex
  - local value = ∞, status = inactive at all other vertices
- User-defined program

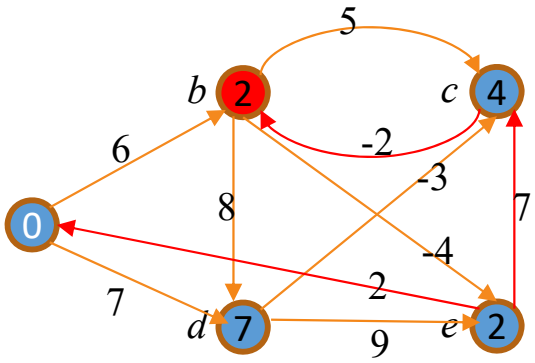
```
for each neighbor v
    send_message(v, val + dist(self, v))

new_val = min(all messages m received)
if new_val < val then
  val = new_val
else
  set status to inactive
```

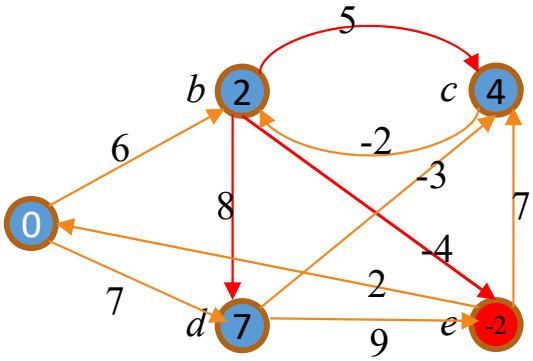
- Can be much faster (less rounds) than Dijkstra's algorithm on shallow graphs
- But may do more total work.
- It also supports negative-weight edges
  - Dijkstra's algorithm cannot handle negative-weight edges


a
0
b
∞
c
∞
d
∞
e
∞
5
6
-2
-3
8
7
-4
2
7
9


a
0
b
6
c
∞
d
7
e
∞
5
6
-2
-3
8
7
-4
2
7
9


0
b
6
c
4
d
7
e
2
5
6
-2
-3
8
7
-4
2
7
9


0
b
2
c
4
d
7
e
2
5
6
-2
-3
8
7
-4
2
7
9


0
b
2
c
4
d
7
e
-2
5
6
-2
-3
8
7
-4
2
7
9