Shared Memory Parallel Stream Processing

Srikanta Tirthapura

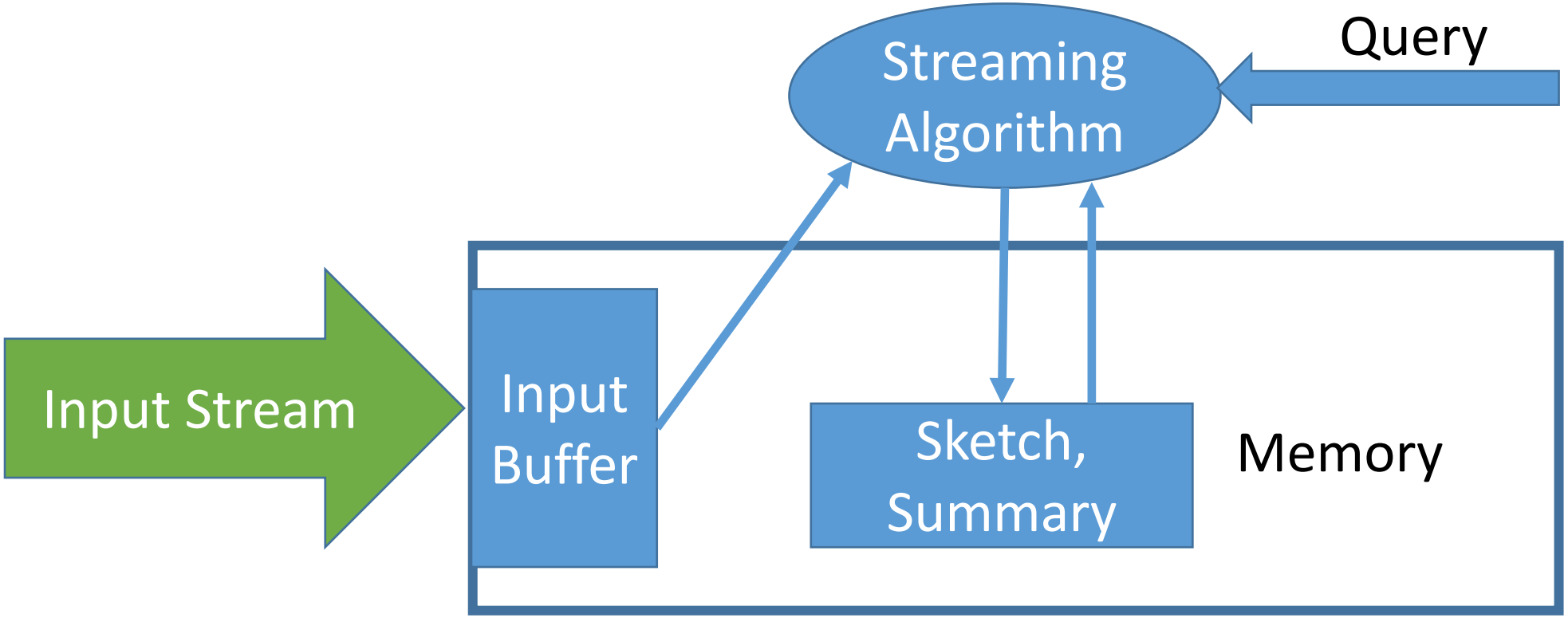
Iowa State University

snt@iastate.edu

Joint work with Kanat Tangwongsan, Kun-Lung Wu

Workshop on Data Summarization, Warwick, 2018

Sequential Streaming

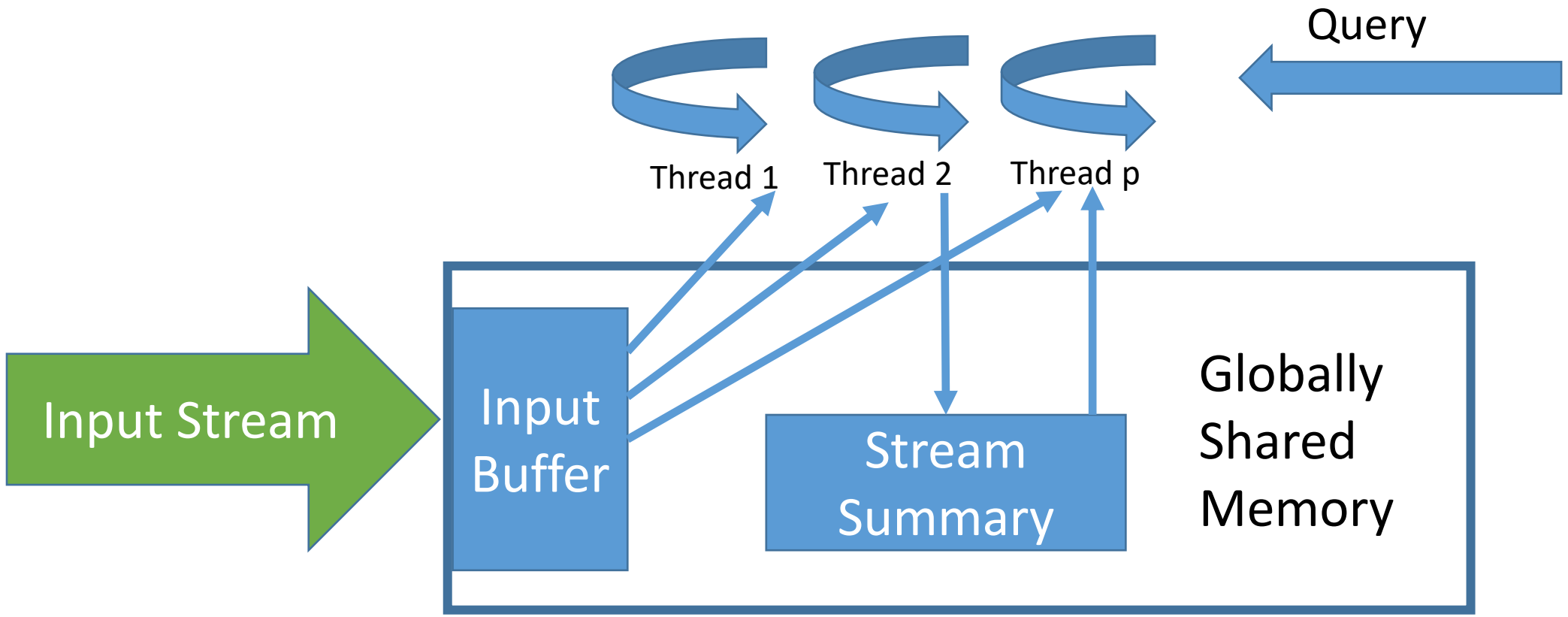


Shared Memory Parallel Machines

- Server machines commonly have tens to hundreds of cores and 100+ GB of shared memory

- How can we harness their power for stream processing?
  - How best to increase throughput?
  - Can we retain memory-accuracy tradeoffs?

Shared Memory Parallel Streaming



Goals of Parallel Streaming

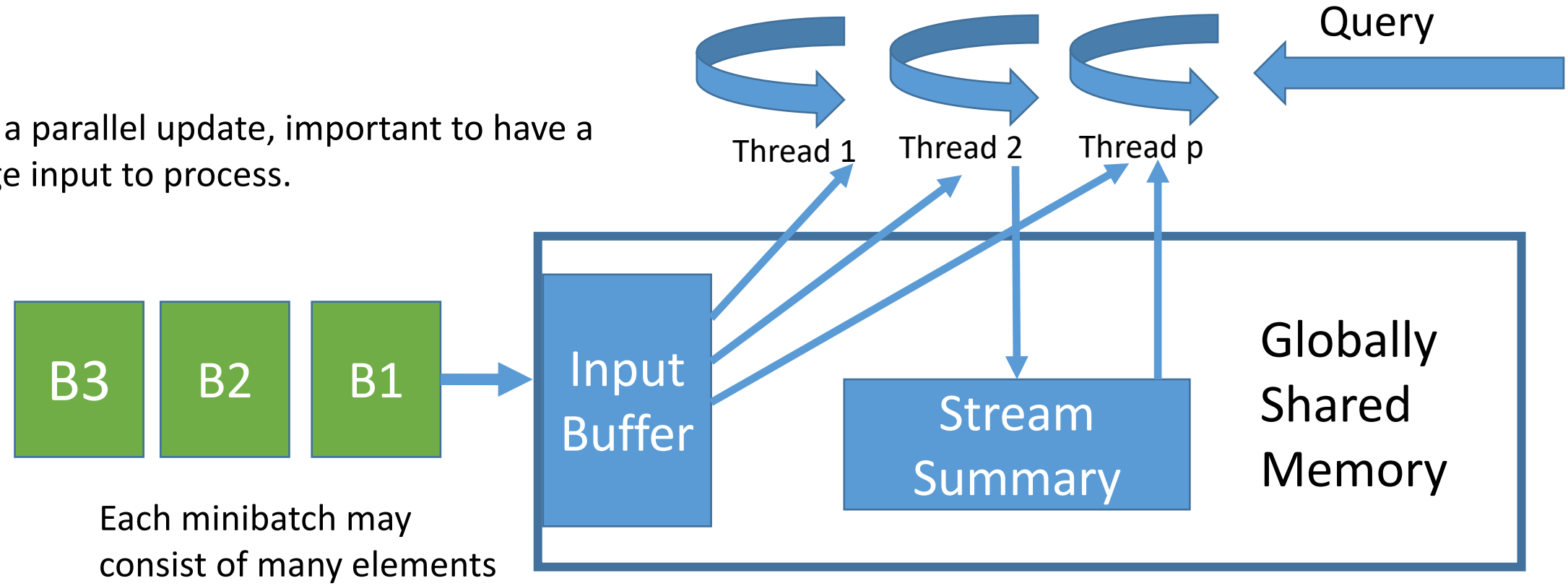
1. Parallel Update: When a new minibatch is received, can the sketch be updated in parallel, increasing the throughput?

2. Query: When a query is received, can it be answered quickly?

3. Accuracy: Can the memory/accuracy tradeoff of the sequential algorithm be preserved?

Parallel Streaming in Shared Memory Stream is a Sequence of "Mini Batches"

For a parallel update, important to have a large input to process.



Measure of Parallel Effectiveness: Work-Depth Model

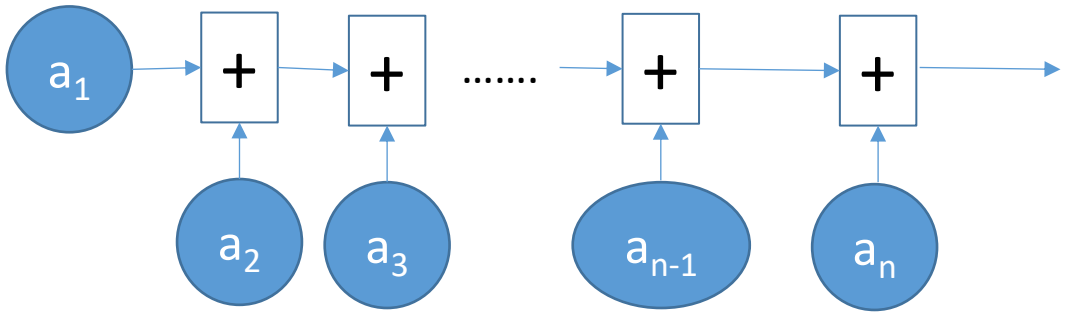
- Abstracts away the details of modeling a parallel computer
- Algorithm written as a computation graph (a DAG)

- **Work** of the algorithm is the total number (cost) of operations in the computation graph

- **Depth** is the longest directed path from the input to the output

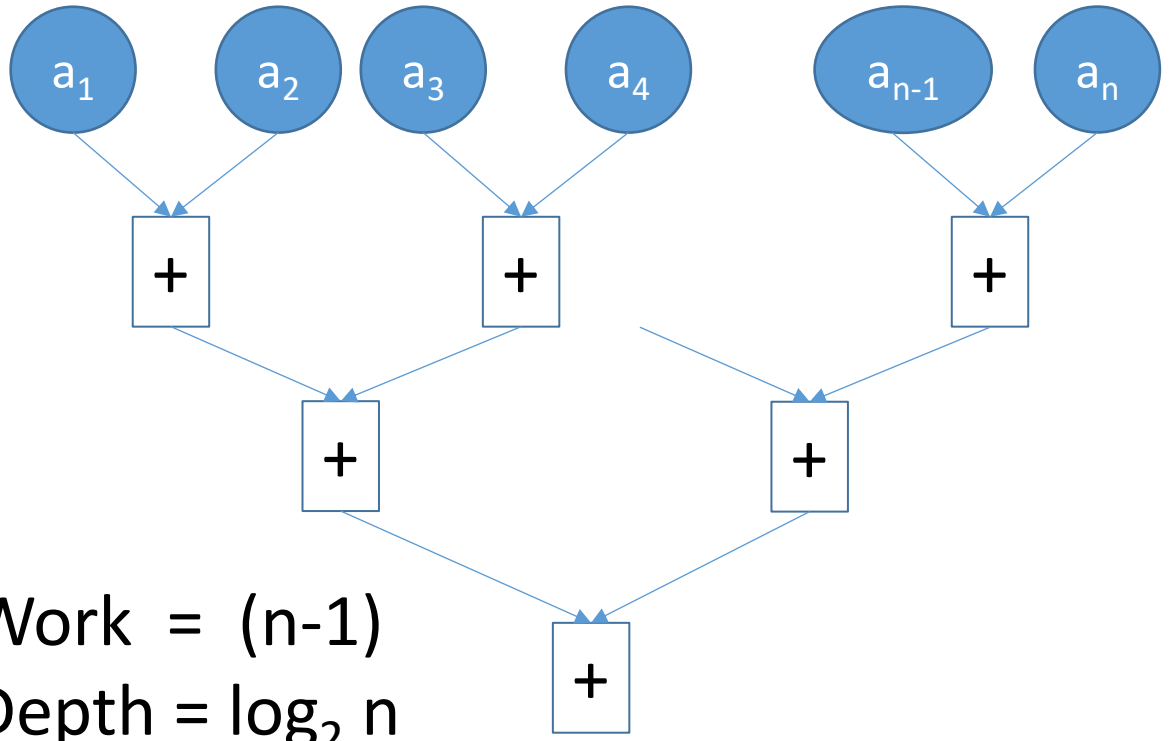
Example: Adding n numbers

Algorithm 1



Work = (n-1)
Depth = (n-1)

Algorithm 2



Work = (n-1)
Depth = $\log_2 n$

Work-Depth to Speedup, Efficiency

An algorithm written using the work-depth model can be translated to other related models, such as the PRAM model.

Brent's Theorem: Any algorithm that can be expressed as a circuit of total work W and depth D and with constant fan-in nodes in the circuit model can be executed in O(W/P + D) steps in the CREW PRAM model.

Addition in $O\left(\frac{n}{P} + \log_2 n\right)$ parallel time steps on a CREW PRAM, using P processors.

Speedup with P processors = $\mathrm{O}\left(\frac{n}{\frac{n}{P}+\log_2 n}\right)$

Speedup =$\Theta(P)$ as long as P = O(n/log n)

Algorithms written using the work-depth model can be translated into parallel programs using e.g. Cilk Plus (Intel)

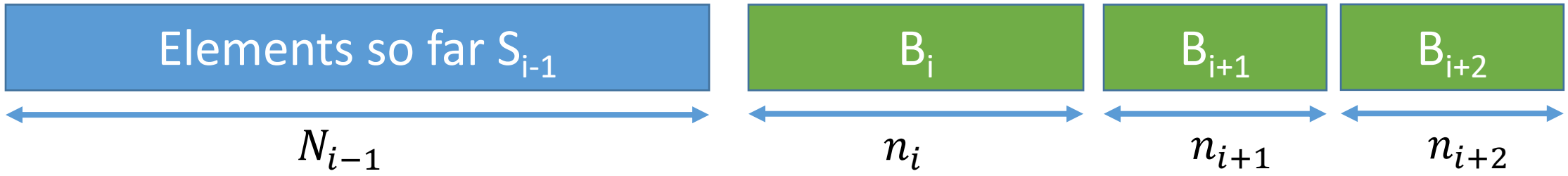
"Gold Standard" for Parallel Algorithms in Work-Depth model:

- Work-Efficient: Work is the same order as the best sequential algorithm
- Depth is poly-logarithmic in the input size

Overview

- Parallel Streaming in Shared Memory
- Aggregation Tasks
  - Random Sampling
  - Frequency Estimation (Heavy-Hitter Identification)
- Comparison with Distributed Streaming

Problem 1: Random Sampling



- Let n_i= size of minibatch B_i and $N_i = \sum_{j=1}^{i} n_j$ the number of elements so far. The desired sample size = s.

- Sampling Without Replacement: For each time i = 1 . . . t , maintain random sample R of size min{s, N_i} chosen uniformly without replacement from $S_i = \cup_{j=1}^{i} B_i$

Sequential Reservoir Sampling Without Replacement

- Initialization: R = {}, and Insert the first s elements in the stream into R
- When element e arrives at position i (i > s)
  - With probability s/i, replace a randomly chosen element in R with e
  - With probability (1-s/i), leave R unchanged
- When there is a query for a random sample
  - return R
- Easy to parallelize this algorithm when a minibatch arrives
- However…. Algorithm has a work of $O(n_i)$ for processing a minibatch B_i with n_i elements

Improved Work (Sequential) Reservoir Sampling

- When a new minibatch B_i arrives, skip directly to the elements that will be selected into the sample (Vitter 1985) – Algorithm Z uses a process to generate random variable $S(s, n)$ where n is the number of elements processed so far

- Total work (and depth) per minibatch = $O\left(s \log\left(\frac{N_{i+1}}{N_i}\right)\right)$, much better, but still linear in s

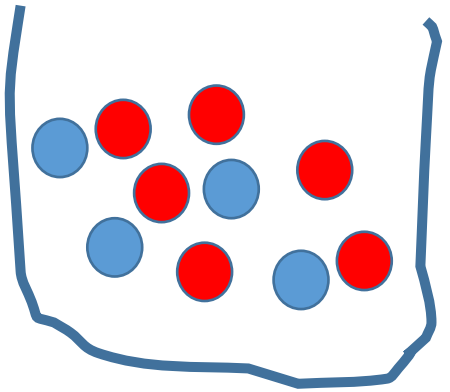
Work-Efficient Parallel Reservoir Sampling (1)

Hyper-geometric random variable H(p,q,r) for integers $p,q \leq r$

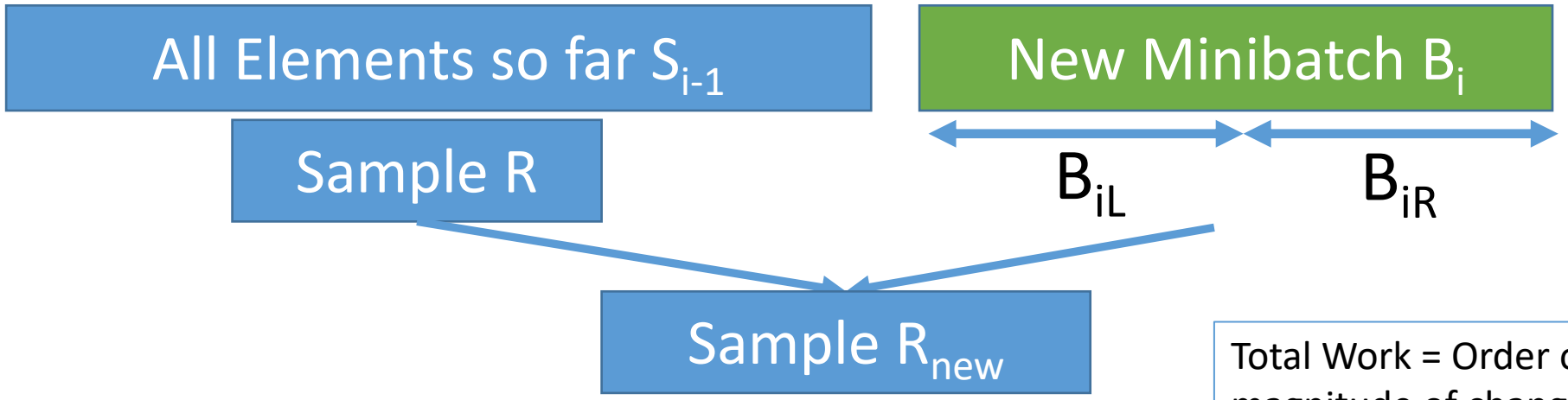
- Suppose there are q RED balls and (r − q) BLUE balls in an urn. Then H(p,q,r) is the number of RED balls drawn in p trials where in each trial, a ball is drawn at random from the urn without replacement.

- There is an algorithm for generating H(p,q,r) in O(1) time (e.g. Stadlober 1990)

p balls chosen out of r



Work-Efficient Parallel Reservoir Sampling (2)



1. Generate H = H(s,n_i,N_i) that equals how many samples in R_{new} come from B_i rather than from S_{i-1}
2. To choose H elements from B_i in parallel
   1. Generate H_L, H_R, how many samples in H come from B_{iL} and from B_{iR}
   2. In Parallel
      1. Generate H_L elements from B_{iL}
      2. Generate H_R elements from B_{iR}
3. Replace H randomly chosen elements in R by the newly chosen elements

Total Work = Order of magnitude of change in R

Work is optimal

Depth is logarithmic in s, n_i

See also Sanders et al. ACM TOMS 2018

Work-Efficient Parallel Reservoir Sampling

- Expected work = order of magnitude of change in the sample, which can be seen to be optimal

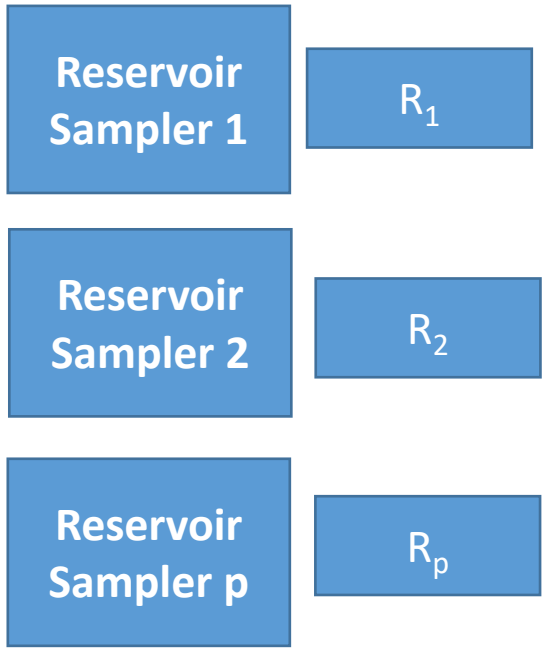
- Lemma: There is a work-efficient parallel algorithm with optimal (up to constant factors) total expected work to process a minibatch and depth logarithmic in sample size s and minibatch size n_i

Comparison with Vitter 1985 (Algorithm Z)

- Algorithm Z is sequential, while ours is parallel algorithm

- Algorithm Z applies in case when the size of the input N_i is unknown, and we have described the case when N_i is known – The unknown N_i case can be parallelized as well
  - If N_i is known, our algorithm is a better sequential algorithm as well

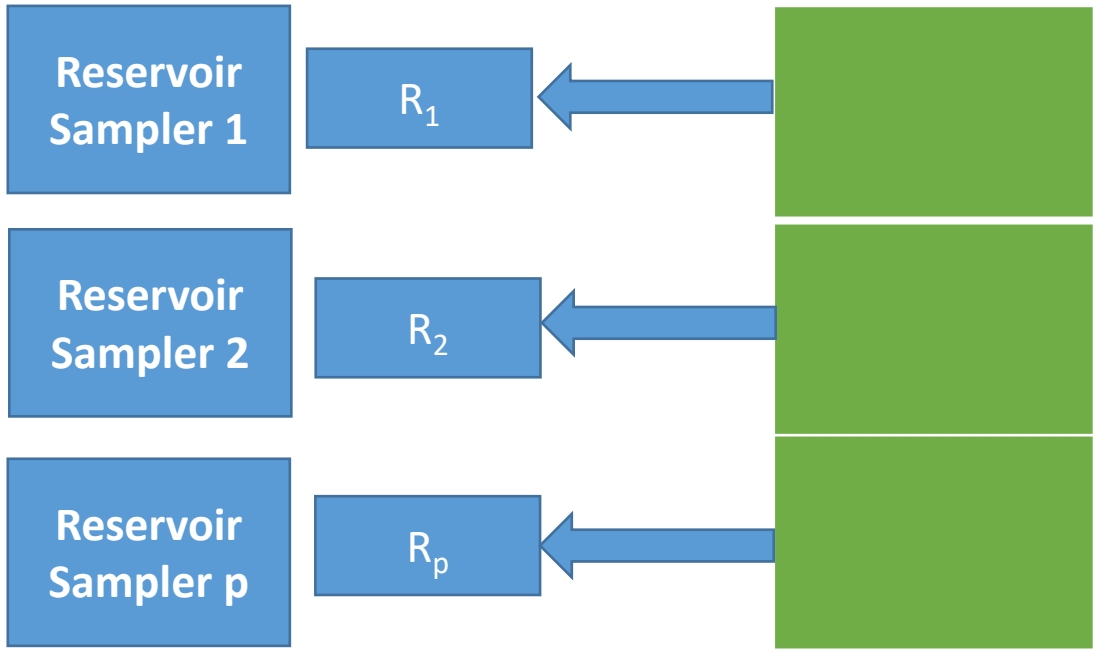
Alternate Approach Using Independent Data Structures



Many Independent Reservoir Samplers

New Batch B_i

Alternate Approach Using Independent Data Structures



Many Independent Reservoir Samplers

New Batch B_i

This Approach has Two Problems

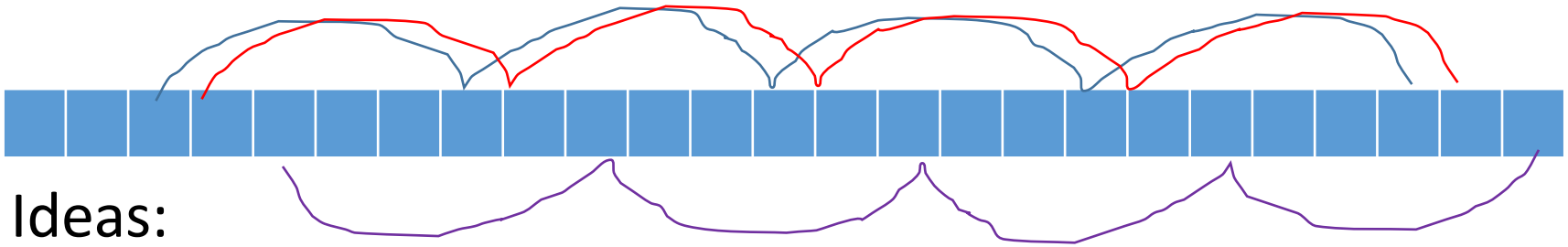
- More space than necessary
- Processing Work is too high

Parallel Stream Sampling from a Sliding Window

- Two versions:
  - Fixed Size Window: Window size W is fixed and known at processing time
    - Sequential Algorithm uses a reduction to the case of infinite window (Braverman, Ostrovsky, Zaniolo, 2009)
    - Similar reduction in parallel
  - Variable Size Window: Window size is a parameter that is provided at query time – common solutions use "bottom-k" sampling (BDM02, GL08, XTB08, BOZ09)
    - Each stream element gets a randomly chosen weight from [0,1]
    - Sample equals the set of s elements with the smallest weights within window
    - Invariant: Maintain only those elements which are not dominated by s or more elements that came later
    - Parallelizing this case needs an alternate view of work-efficient reservoir sampling than is provided by Vitter's Algorithm Z
- Tangwongsan and Tirthapura, "Parallel Random Sampling in the Minibatch Streaming Model", submitted, 2018.

Work-Efficient Parallel Sampling from a Variable-Length Sliding Window

- Lemma: Let W be the maximum possible window size. For fixed $s \leq W$, there is a data structure for Variable Window Sampling, consuming expected $O(s + s\log(W/s))$ space, supporting
  - insert(B_i) in $O(s + s \ln(W/s))$ work and $O(\log W)$ depth
  - sample(q, t) in O(q) work and $O(\log W)$ depth.



- Algorithm Ideas:
  - Size 1 sampler for a sliding window
  - Divide new minibatch B_i into multiple "tracks" – each track skips s elements to get to the next one
  - Sample elements in different tracks in parallel, and post-process the result

Overview

- Parallel Streaming in Shared Memory
- Aggregation Tasks
  - Random Sampling
  - Frequency Estimation (Heavy-Hitter Identification)
- Comparison with Distributed Streaming

Problem 2: Heavy Hitter Identification (Frequency Estimation)

- Consider a stream where each element e is chosen from {1,2...,M}. Let f_e denote the number of occurrences of element e. N is the total number of elements in the stream. Given thresholds $\varepsilon < \phi < 1$.

- ε-Accurate Frequency Estimation: For each element e, maintain an estimate g_e such that $(f_e - \varepsilon N) \leq g_e \leq f_e$

- Heavy Hitter Identification: Output all elements e such that $f_e \geq \phi N$ and no element such that $f_e < (\phi - \varepsilon)N$

Sequential Misra-Gries Algorithm (MG)

- The MG summary (MGS) is a set of up to S = 1/ε <item,frequency> pairs

- When item e arrives:
  1. If e in MGS, then increment frequency of e
  2. Else
     1. If MGS has less than S items, add <e,1>
     2. Else, decrement frequency of each item in MGS by 1, and remove the item if its frequency reaches 0

Parallel Heavy Hitter Identification

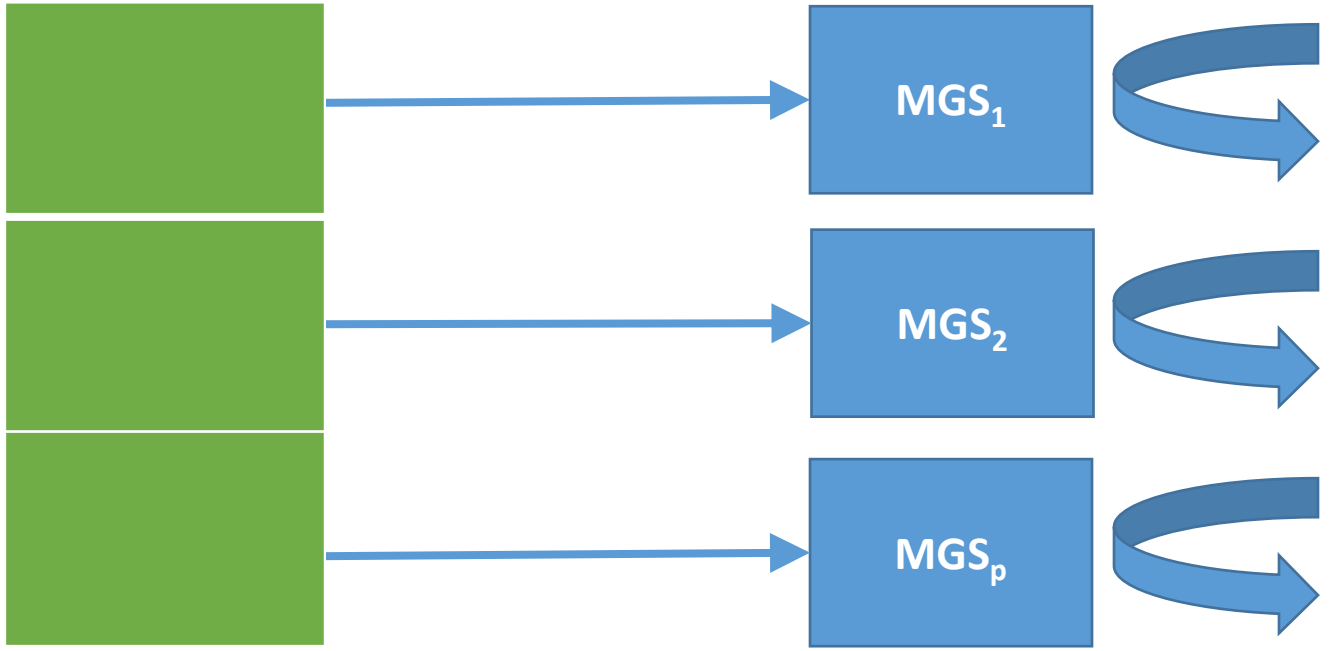
- Theorem: Let $\varepsilon < 1$. There is a parallel algorithm for ε-accurate heavy-hitter identification that
  - requires $O(1/\varepsilon)$ space,
  - adding a mini-batch of size M takes time $O(M + 1/\varepsilon)$ work
  - depth of the algorithm for adding a mini-batch poly-logarithmic in $(M+1/\varepsilon)$.

Parallel MG Using Independent Data Structures

New Batch B_i

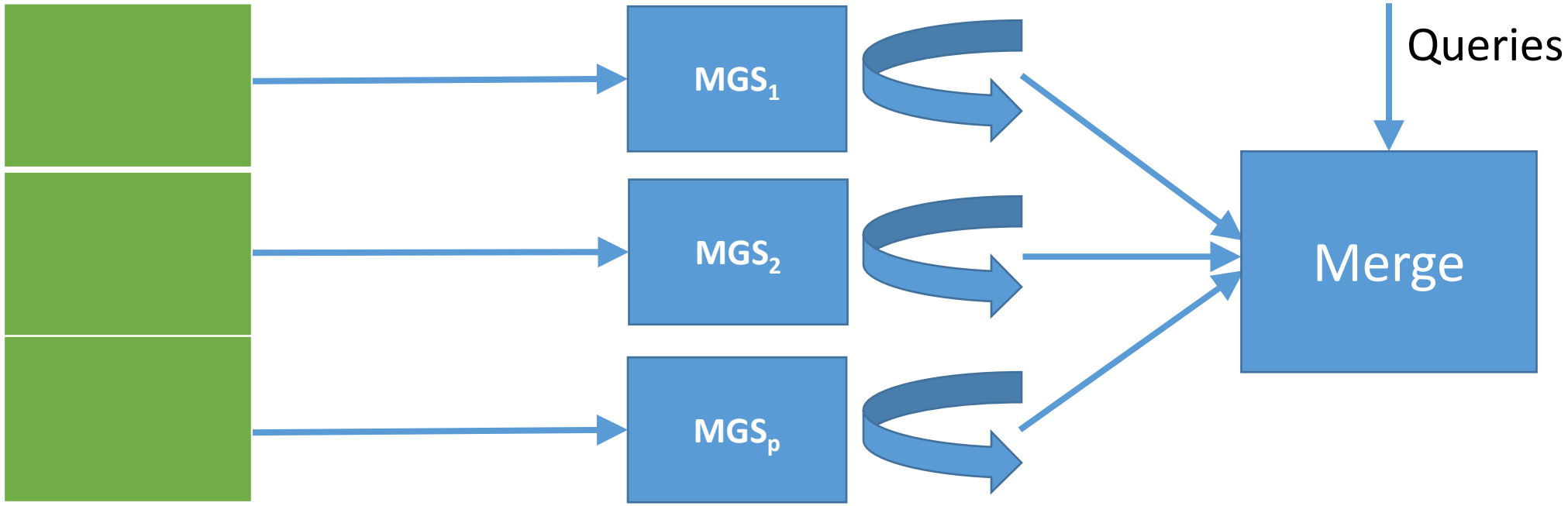
Many Independent Copies of the MG data structure

Parallel MG Using Independent Data Structures



New Batch B_i

Parallel MG Using Independent Data Structures



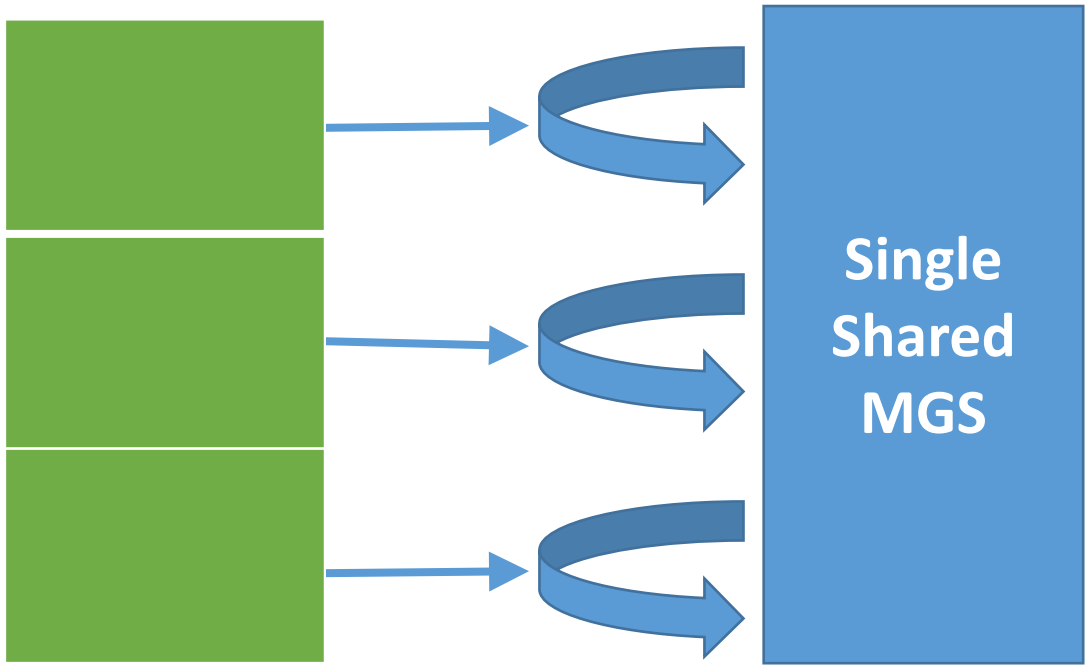
A Mergeable Summary for Frequent Elements Can be Used ("Mergeable Summaries", Agarwal et al. ACM TODS 2013)

Drawbacks with Independent Data Structure Approach

1. Poor Memory-Accuracy Tradeoff: Total workspace is a factor of p times more than the sequential MG algorithm, for the same accuracy guarantee

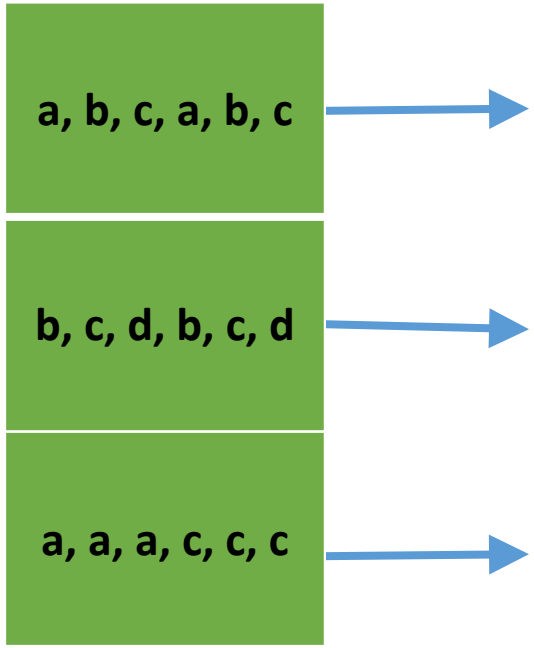
2. Merging still a bottleneck for parallelization

Parallel MG Using Shared Data Structure



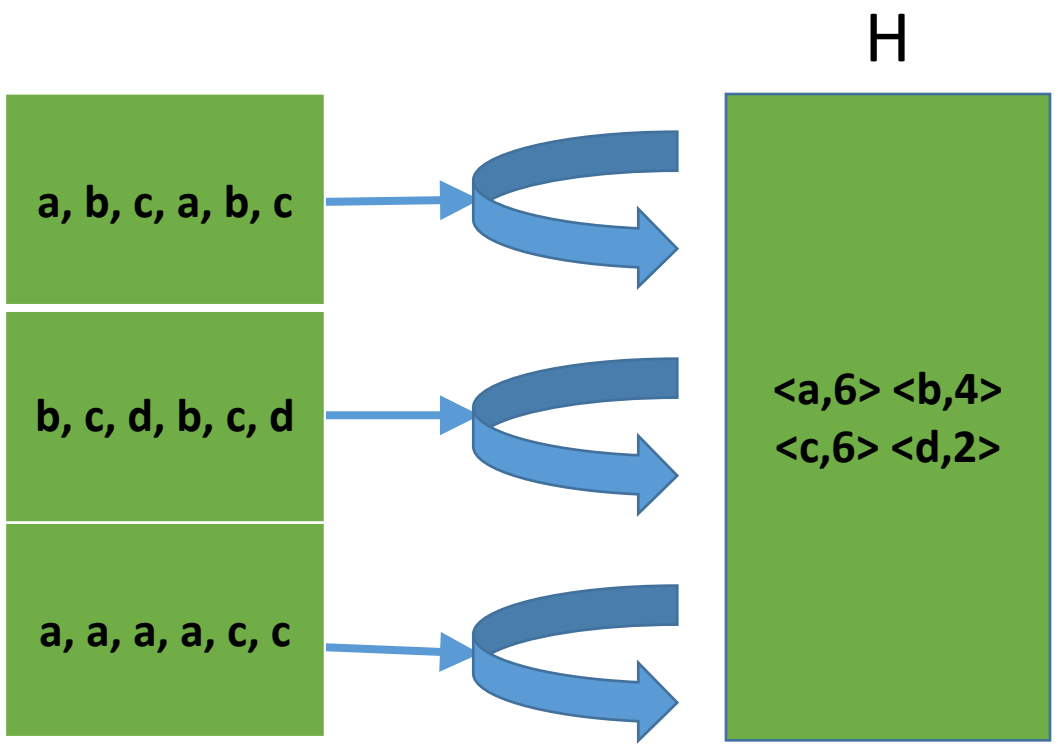
New Batch B_i

Parallel MG Using Shared Data Structure



New Batch B_i

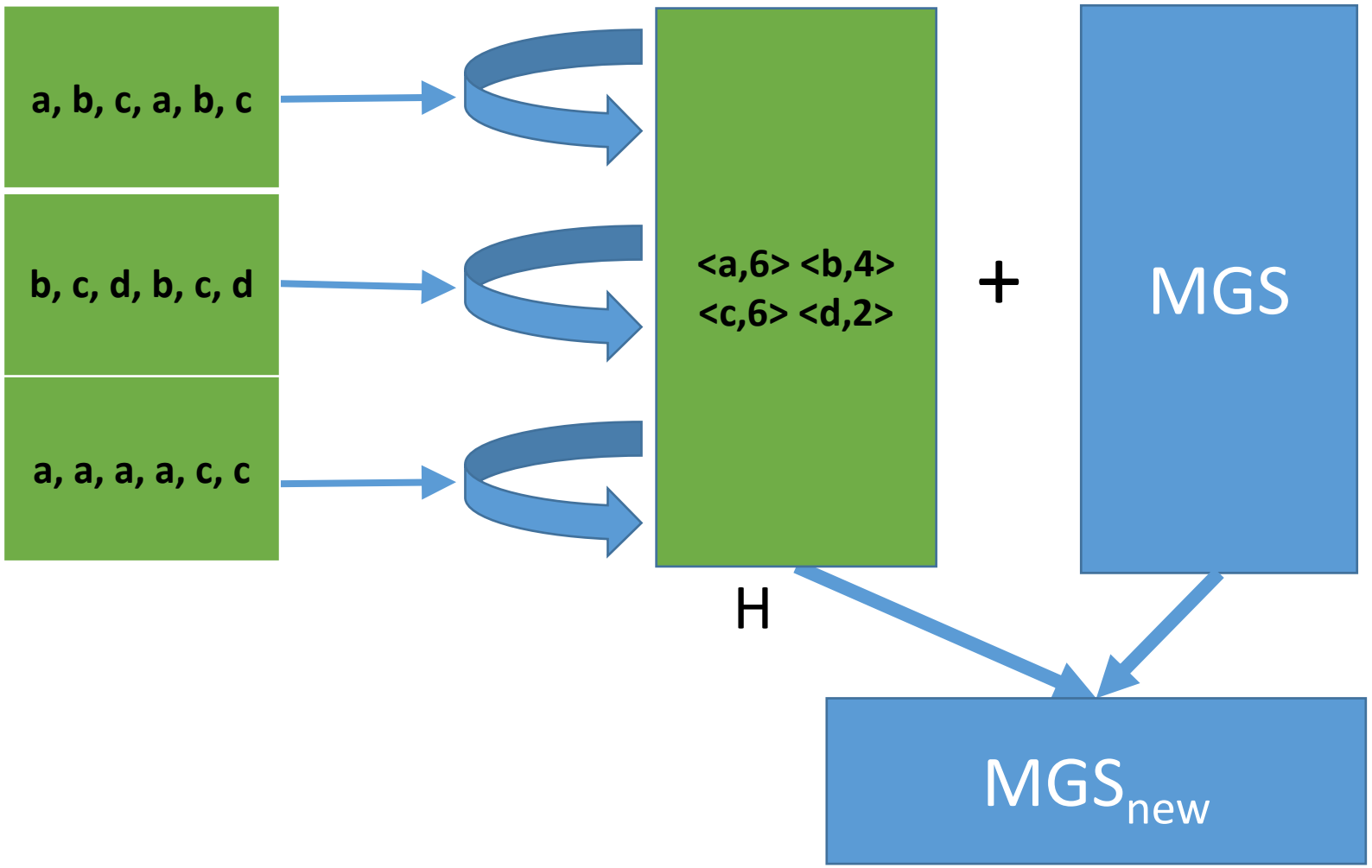
Parallel MG Using a Shared Data Structure



Theorem: There is a parallel algorithm **buildHist** that takes a sequence a_1, a_2, a_3,.., a_n, and builds a histogram of their frequencies, using O(n) work and O(log n) depth.

Building Block: **Parallel Integer Sort (Rajasekaran and Reif, 89)** that takes a sequence of integer keys of length n, each a between 0 and O(n) and produces a sorted sequence using O(n) work and O(log n) depth.

Parallel MG Using Shared Data Structure



Combining MGS and H to get MGS_{new}

1. **Parallel Merging of Histograms:** MGS and H are histograms consisting of <item, freq> pairs. Add up corresponding frequencies to get a new histogram H_{new} (done in parallel using linear work, using an algorithm based on hashing)
   - Problem: H_{new} may have more than S non-zero entries

2. Using a parallel version of QuickSelect, find an integer ρ such that at most S elements of H_{new} have a frequency of ρ or greater. (linear work parallel algorithm)

3. Subtract ρ from each element in H_{new} to get a summary MGS_{new} with at most S non-zero entries. This is a MG-summary of all elements so far.

Correctness: A single decrement of a specific item's counter is still accompanied by a decrement to at least S counters corresponding to distinct items.

Parallel Streaming Heavy Hitter Identification

- Theorem: Let $\varepsilon < 1$. There is a parallel algorithm for ε-accurate heavy-hitter identification that requires $O(1/\varepsilon)$ space, such that adding a mini-batch of size M takes work $O(M + 1/\varepsilon)$ work. The depth of the algorithm is poly-logarithmic in $(M + 1/\varepsilon)$.

- Observation:
  - Matches the space-accuracy tradeoff of the sequential Misra-Gries algorithm
  - Work-Efficient, if we set the mini-batch size $M = \Omega(1/\varepsilon)$

- Also handle the sliding windows version of this problem

- Further details in: "Parallel Streaming Frequency-Based Aggregates", Tangwongsan, Tirthapura, Wu, ACM SPAA 2014

Shared vs. Independent Data Structures for Heavy Hitter Identification

Single Shared Data Structure Provides

- Better space-accuracy trade-off than Independent Data Structures
- Does not require a further merge step during answering a query

Similar Results hold for other frequency-based aggregates

- Sketch-based algorithms, such as Count-Min Sketch

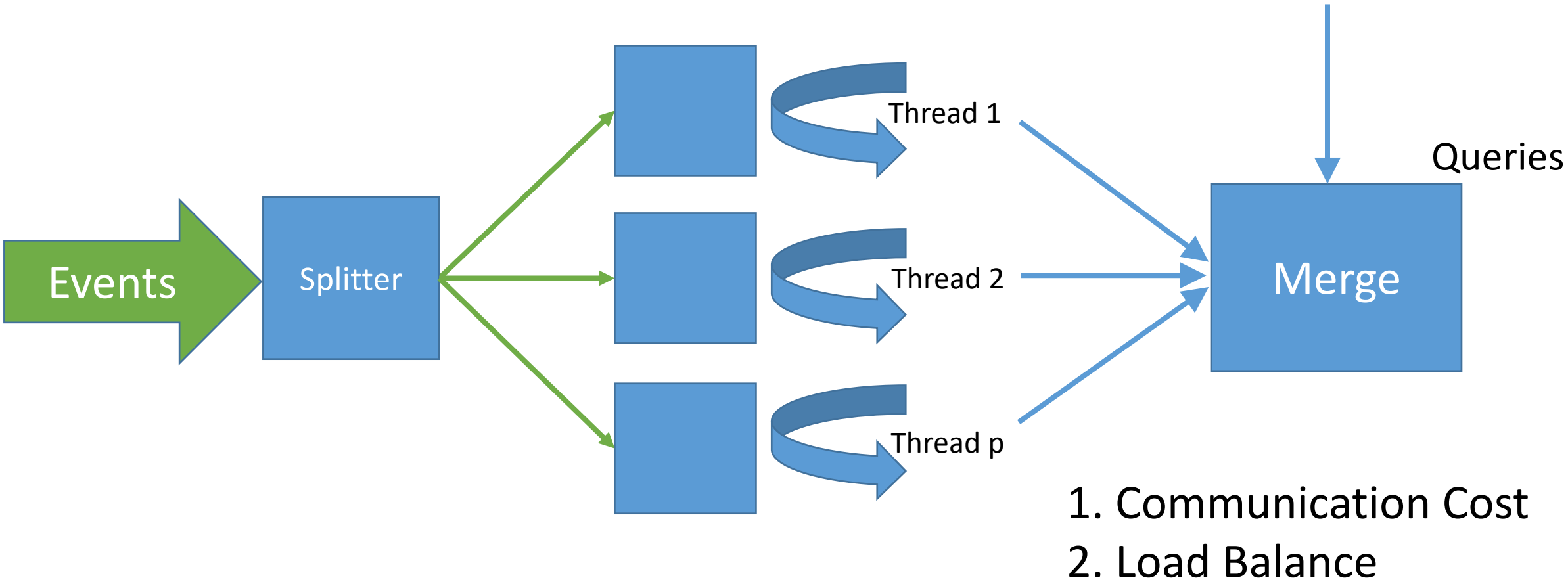
Other Aggregates

- Sliding Window Aggregates
  - Basic Counting
  - SUM
  - Heavy Hitters
- Count-Min Sketch
- Further details in: “Parallel Streaming Frequency-Based Aggregates”, Tangwongsan, Tirthapura, Wu, ACM SPAA 2014

Overview

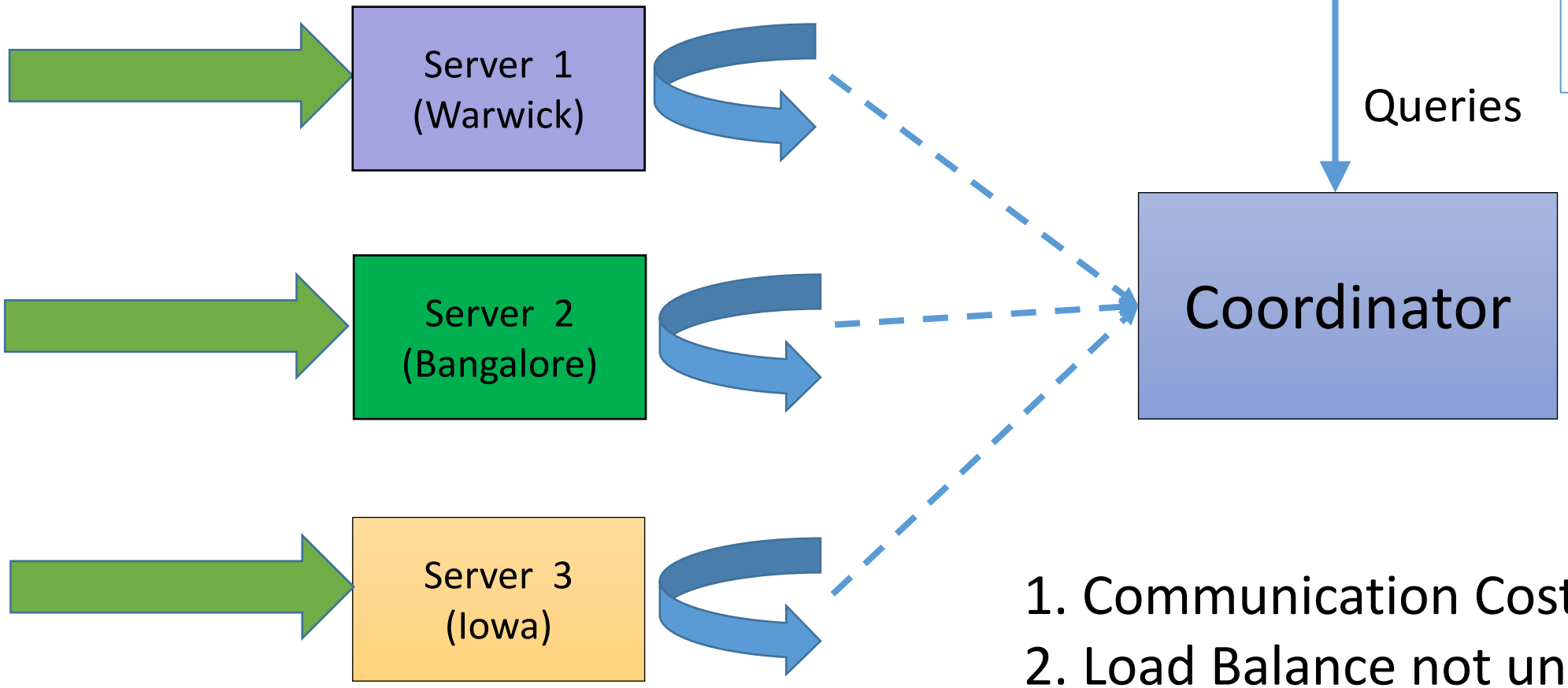
- Parallel Streaming in Shared Memory
- Aggregation Tasks
  - Random Sampling
  - Frequency Estimation (Heavy-Hitter Identification)
- Comparison with Distributed Streaming

Parallel Streaming in Distributed Memory



1. Communication Cost
2. Load Balance

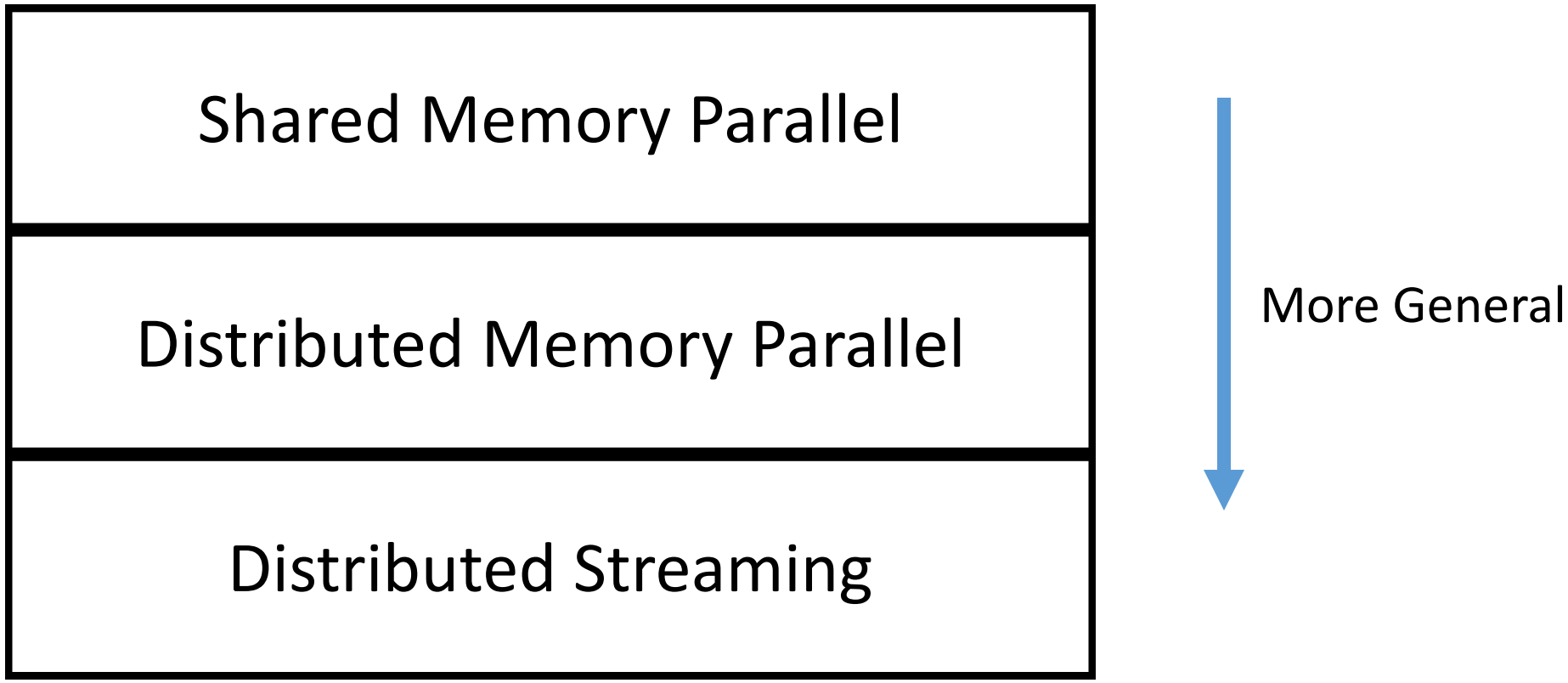
Distributed Streaming



Many works on this Cormode 2013 for a survey

1. Communication Cost Important
2. Load Balance not under control of algorithm

Parallel and Distributed Streaming Models



Streaming, Parallel, Distributed

Sequential Streaming	Single Pass Processing of Data	Bounded Memory	Processing time per item, Query time
Distributed Streaming			Message Complexity
Parallel Streaming			**Parallel Time and Work**

Conclusions

- Other Tasks in Parallel Streaming?
  - Clustering
  - Graph Algorithms
  - More expensive, the better
- Relations between different streaming models
- Parallel and Incremental Algorithms