



CoTS: A Scalable Framework for Parallelizing Frequency Counting over Data Streams

Sudipto Das Shyam Antony Divyakant Agrawal Amr El Abbadi
Department of Computer Science, University of California Santa Barbara, CA 93106, USA

DSL

Supported by NSF
Grants IIS-0744539
and CNS-0423336

Data Stream Processing

Data tuples streaming in - Real-time processing requirements

Frequent Elements and Top-k Queries

- **Frequent Elements query:** Return all the elements whose frequency of occurrence is above a certain threshold
- **Top-k query:** returns the k elements with the highest frequency
- **Frequency counting** forms the basis for both these queries

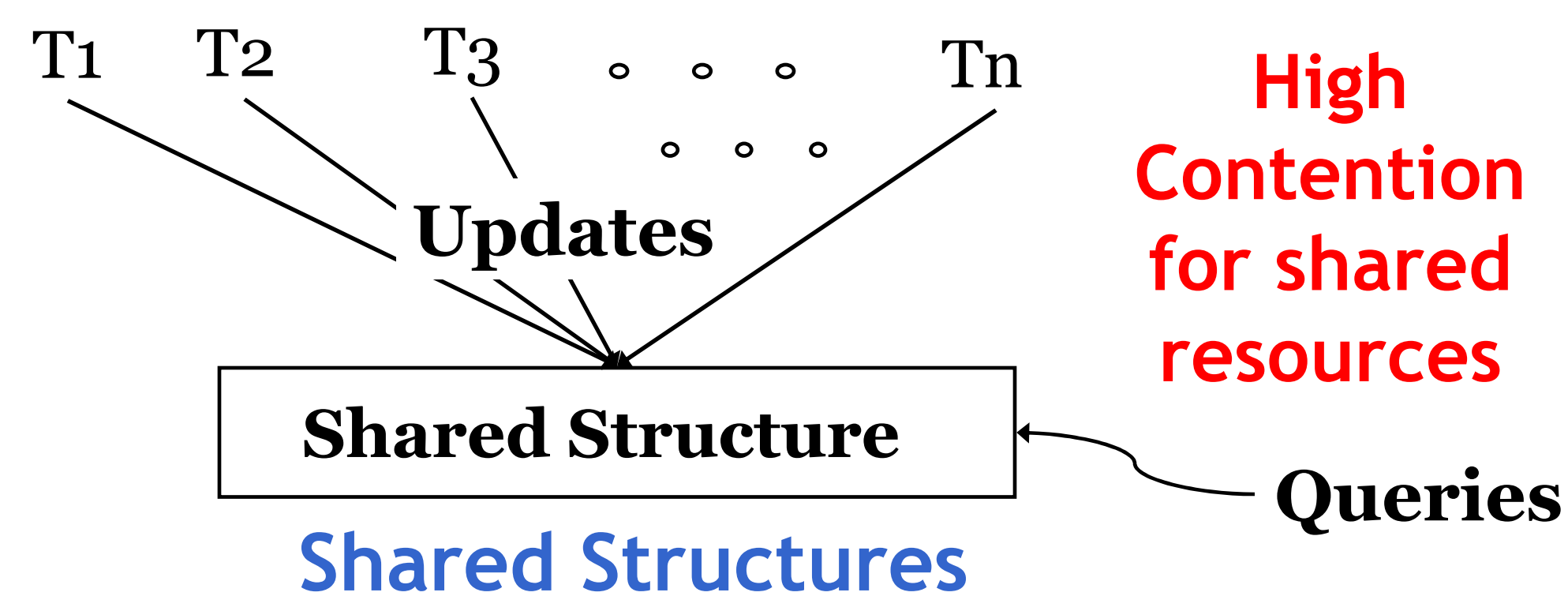
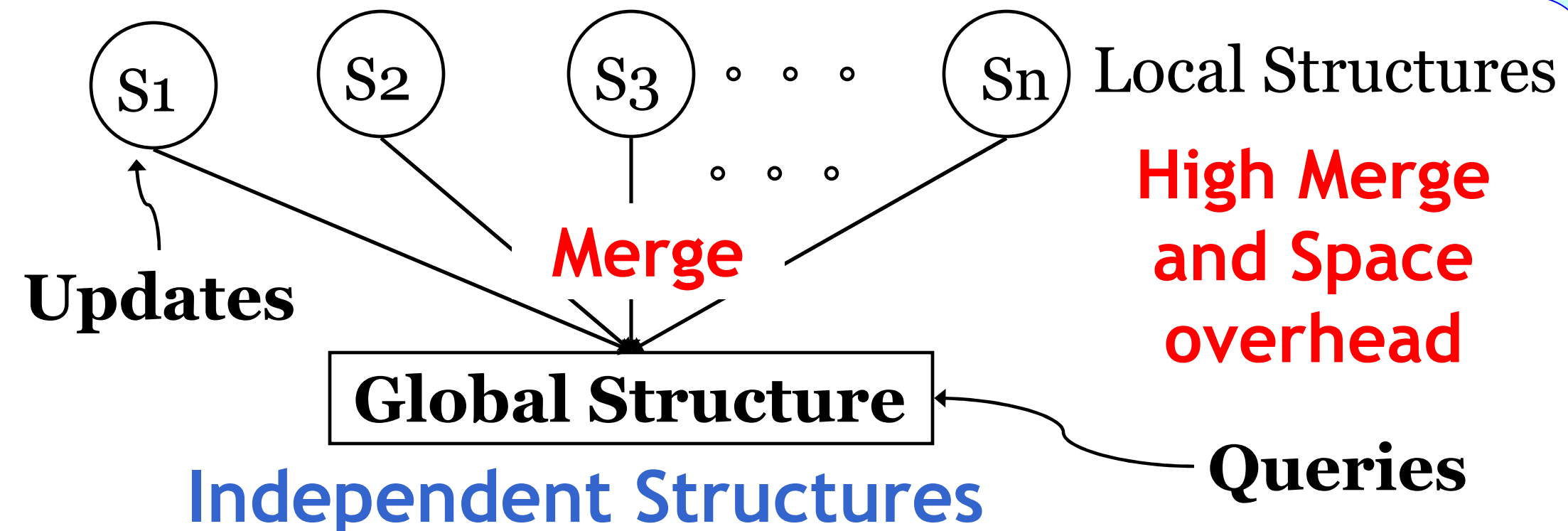
Applications of Frequency Counting

- Network Monitoring: detecting rogue users consuming higher share of network bandwidth
- Click stream analysis for fraud detection and mining

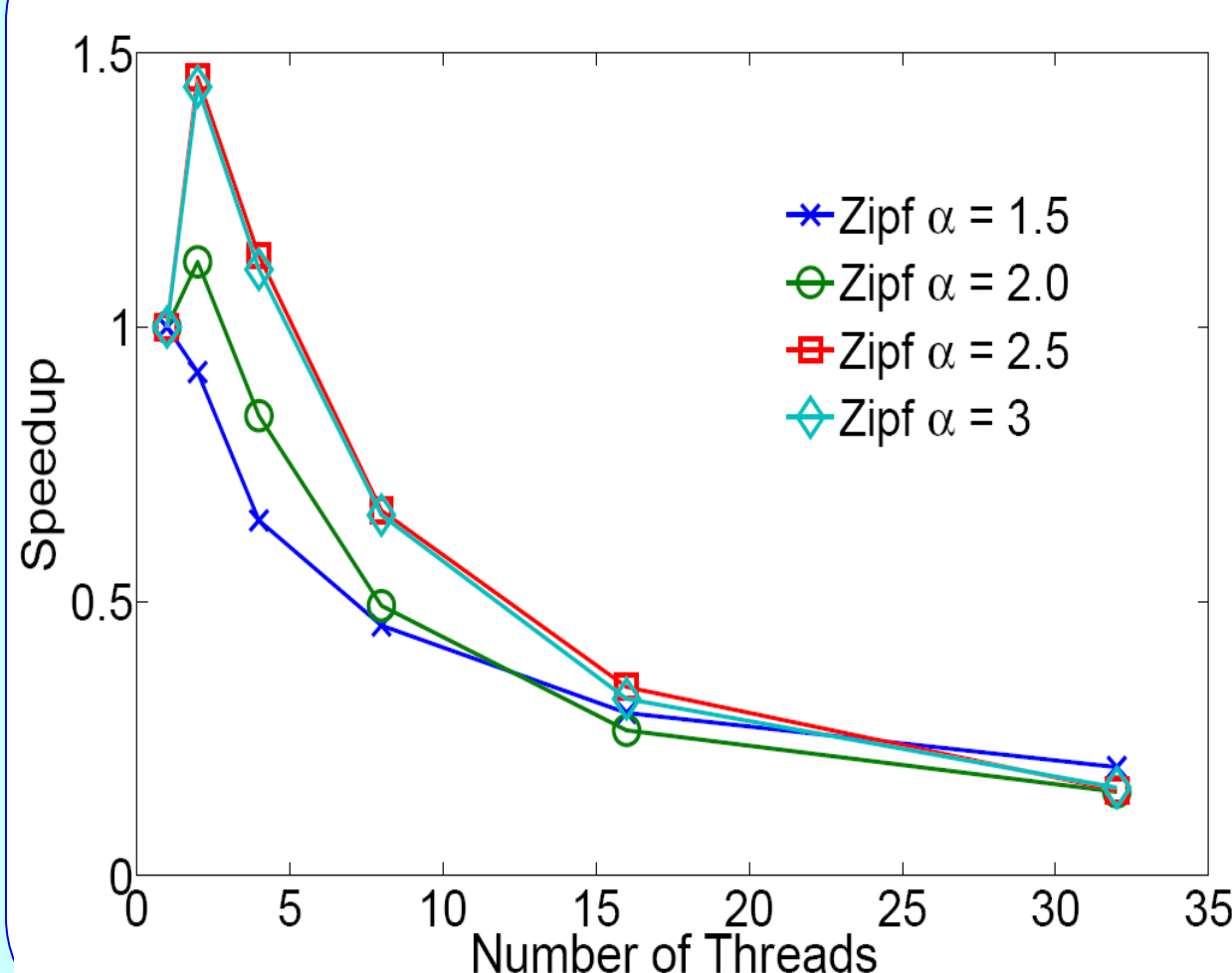
Need for Parallelism

- **Free lunch is over** - No automatic performance boost through increasing processor clock speeds
- Efficient parallel designs needed to effectively utilize the inherent parallelism
- Present state of the art is sequential processing of the stream
- **Intra-operator parallelism** (parallelizing single operator to effectively utilize the multiple cores) would be helpful

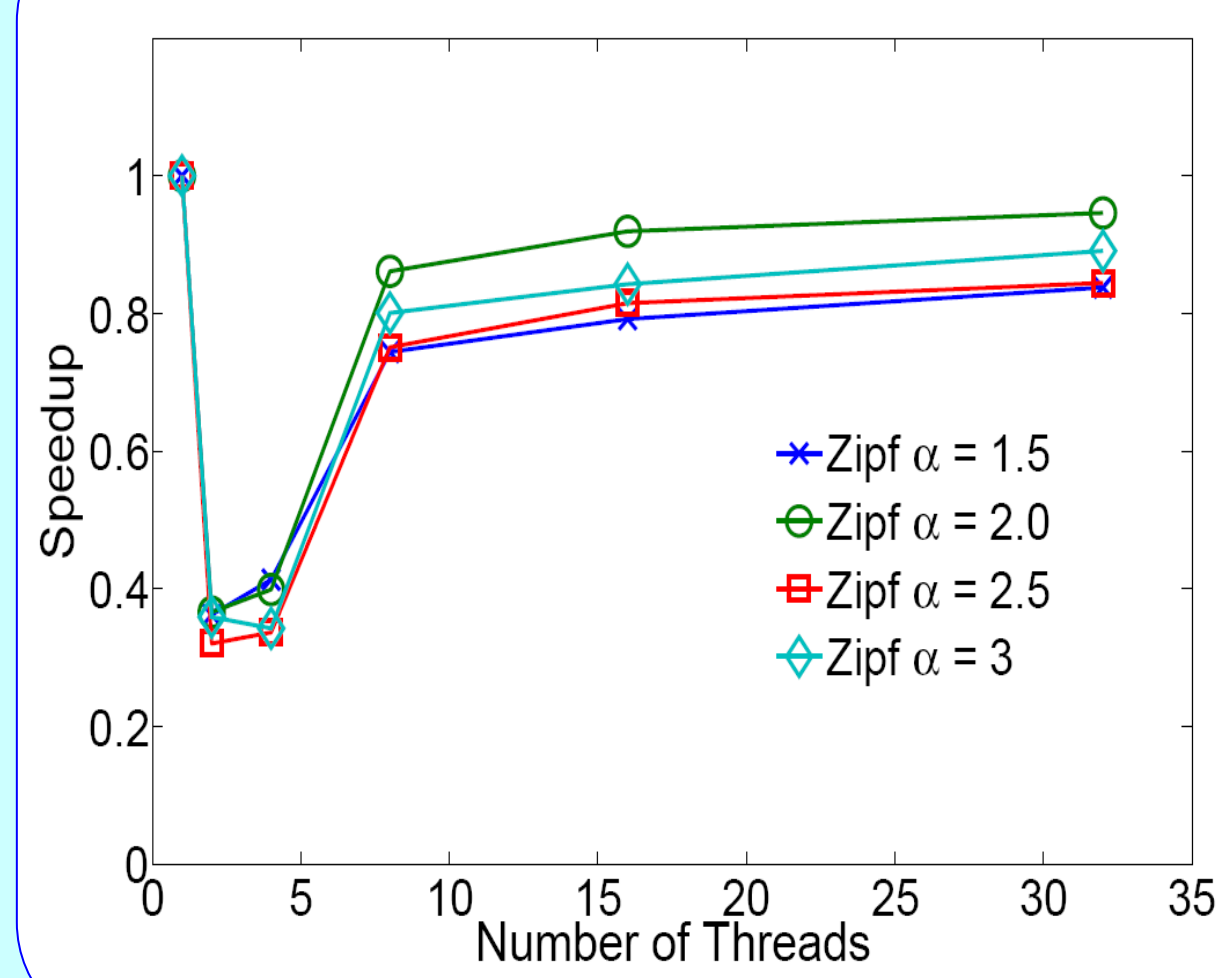
Naïve Parallelization



Independent



Shared



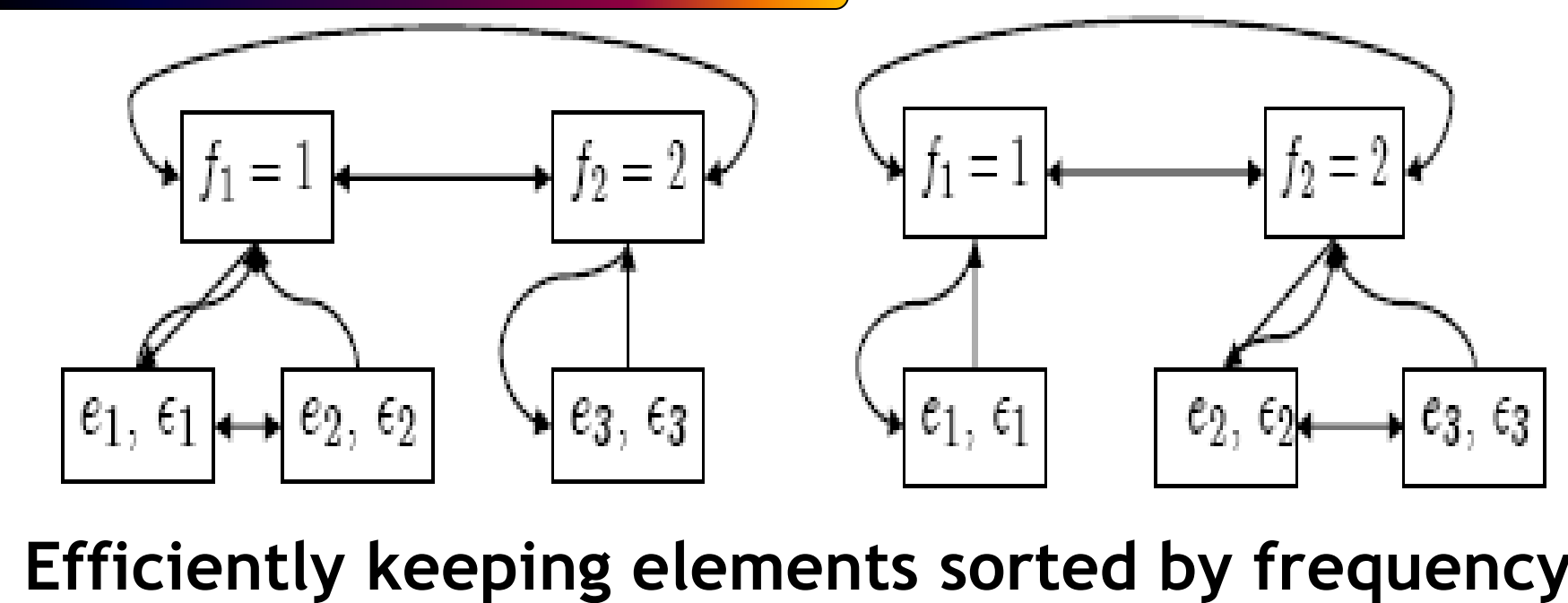
- Synthetic Zipfian distribution with parameter α
- Smaller the value of α , smaller is the skew
- 5 million elements in the stream
- Experiments on an Intel Quad core 2.4GHz processor, implementation in C++. Details in [Das 2008]

Space Saving [Metwally 2006]

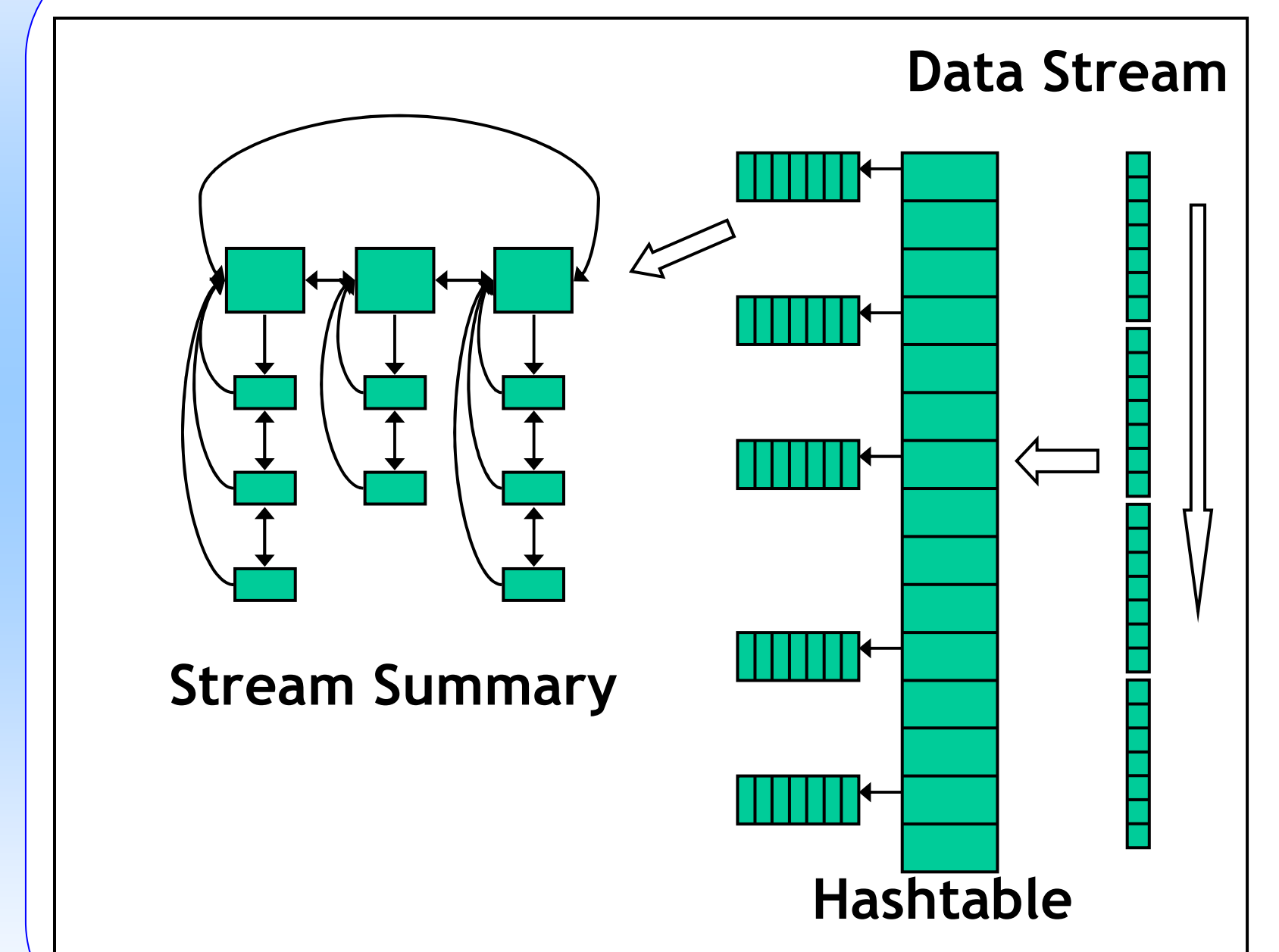
Algorithm

- For a user specified error bound ϵ , monitors only $1/\epsilon$ stream elements, and guarantees ϵ -approximate answers
- Overwrites the minimum frequency element to limit the space
- Benchmarks [Cormode 2008] show that *Space Saving* outperforms similar techniques such as *Lossy Counting*
- **Data Structures used:** **Stream Summary** (for maintaining elements sorted by frequency) and **Hash table** (constant time lookup)
- We choose *Space Saving* for parallelization.

Stream Summary



Data Structures

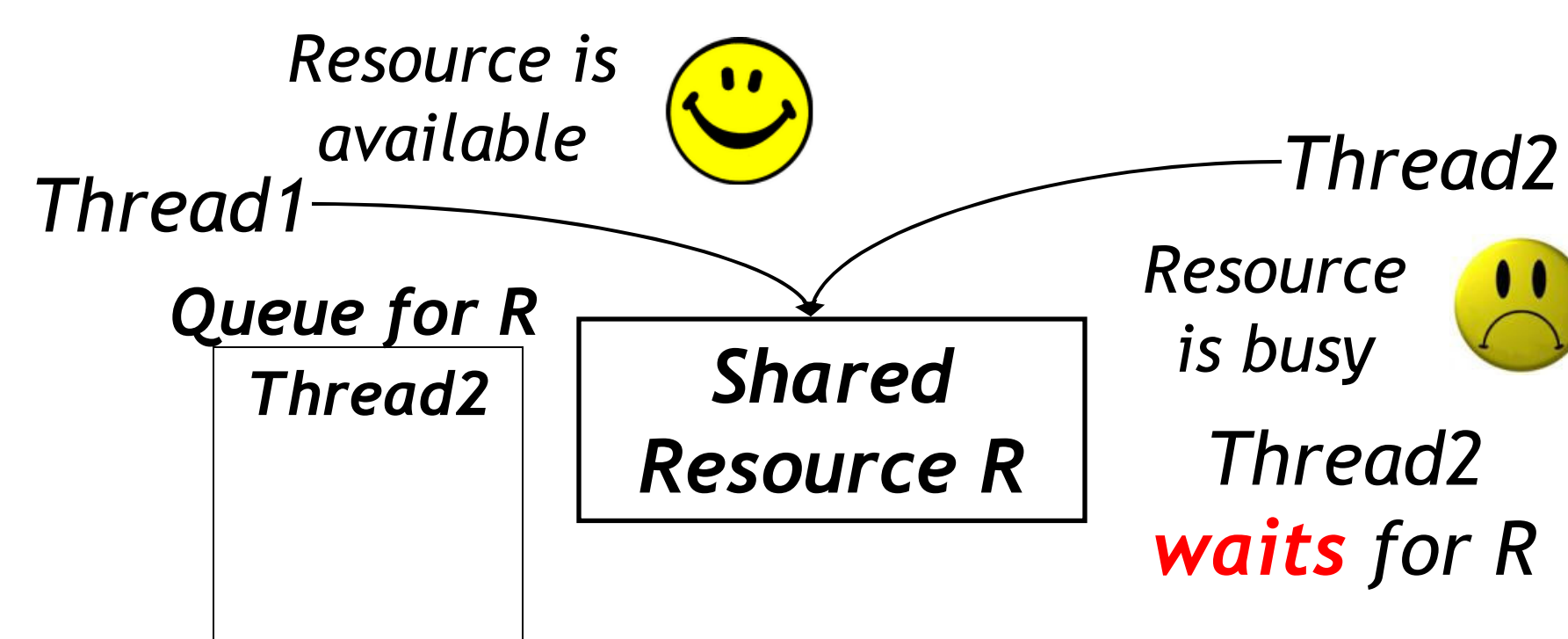


Cooperative Thread Scheduling (CoTS)

Threads should not “contend”, rather they should “cooperate” for sharing resources

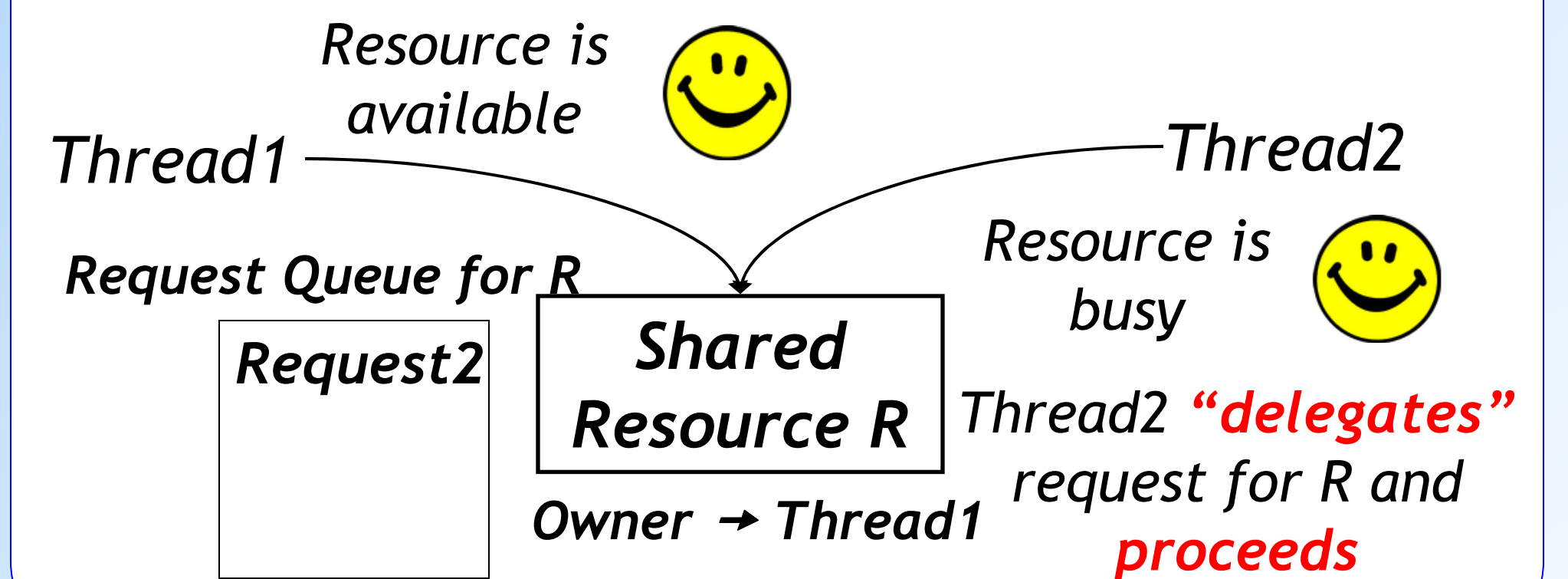
Conventional “contention” based locking

Time wasted “waiting” for shared resources



Proposed “cooperation” based locking

Allow threads to “make progress”



Principle of Request Delegation: If a resource is busy, do not “wait”, rather delegate the request to the thread owning the resource.

Principle of Minimal Existence: Avoid waiting for resources when already acquiring a resource - delegate wherever required.

Request Fulfillment Guarantee: Once a thread has “delegated” a request to another thread, it is neither lost nor left unfulfilled.

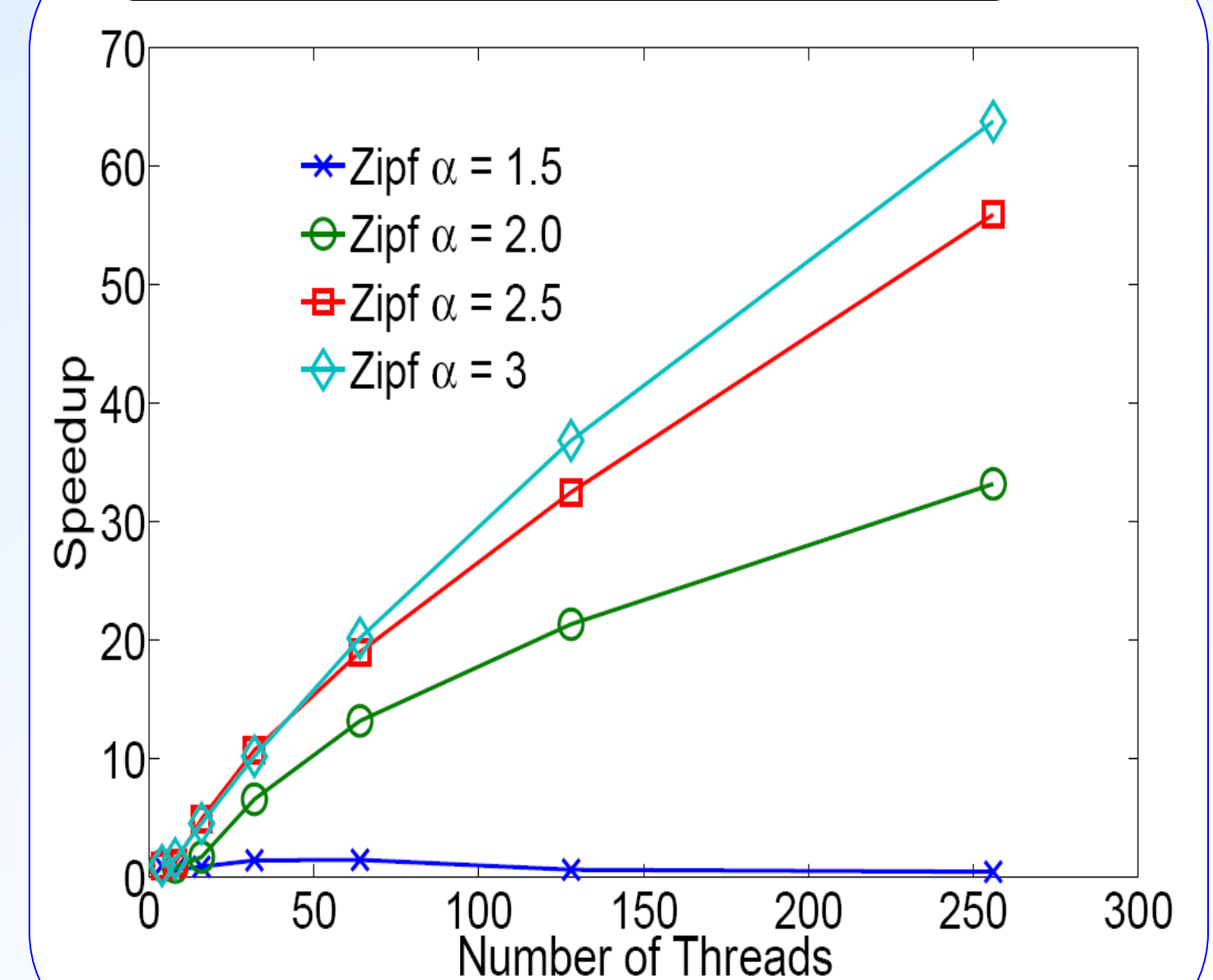
Benefits of Cooperation

Removal of “waits”: Threads do not wait for shared resources. Better utilization of the clock cycles of the multiple cores.

Removal of arbitration overhead: Threads do not contend for shared resources, so no arbitration necessary.

Implementation details in [Das 2008].

Evaluation of CoTS



References

- [Cormode 2008] G. Cormode and M. Hadjieleftherio, “Finding frequent items in data streams,” in *PVLDB* 1(2), pp. 1530-1541, 2008.
- [Das 2008] S. Das, S. Antony, D. Agrawal, and A. El Abbadi, “CoTS: A Scalable Framework for Parallelizing Frequency Counting over Data Streams,” UCSB, Tech. Rep. CS 2008-08, 2008 [Online] Available: http://www.cs.ucsb.edu/research/tech_reports/reports/2008-08.pdf.
- [Manku 2002] G. S. Manku and R. Motwani, “Approximate frequency counts over data streams,” in *VLDB*, 2002, pp. 346-357.
- [Metwally 2006] A. Metwally et al., “An integrated efficient solution for computing frequent and top-k elements in data streams,” *TODS*, 31(3), pp. 1095-1133, 2006.