MilliSort: an Experiment in Flash Bursts

Yilong Li
with Seo Jin Park, Collin Lee, John Ousterhout

Introduction: Flash Burst

- **Today's large-scale datacenter applications run from seconds to hours**
- **Flash burst: a new style of datacenter computation**
  - Very short lifetime (e.g., < 10 ms)
  - Harness hundreds or thousands of servers
  - Enable data-intensive real-time analytics
- **Understand requirements of a general-purpose infrastructure for executing and managing flash burst application**
  - Create an example application (i.e., MilliSort) to learn about flash burst
- **Lessons learned:**
  - Possible to organize 1000s of servers to perform non-trivial computation in <10 ms
  - Group communication operations are critical to performance
  - Full bisection bandwidth is necessary for best performance of shuffle

Introduction: MilliSort

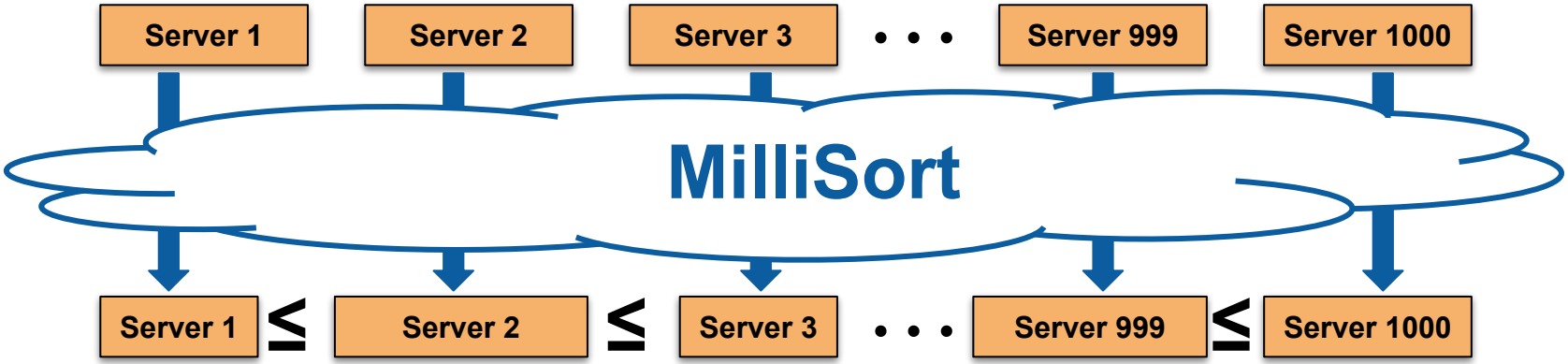
- **MilliSort: sort as many small records as possible within 1 ms, using any number of servers available in a datacenter**
- **Why sorting?**
  - Intensive and unpredictable communication
  - Interesting algorithm
  - Useful building block in distributed computation
- **Early results**
  - Sort 4.6 million 100-byte records using 700 servers (106x speedup) in 1 ms
  - # servers harnessed & data per server increase linearly with time budget
  - # records sortable increases quadratically with time budget

Outline

- **Millisort Overview**
- **Implementation & Cost Estimator**
- **Measurements**

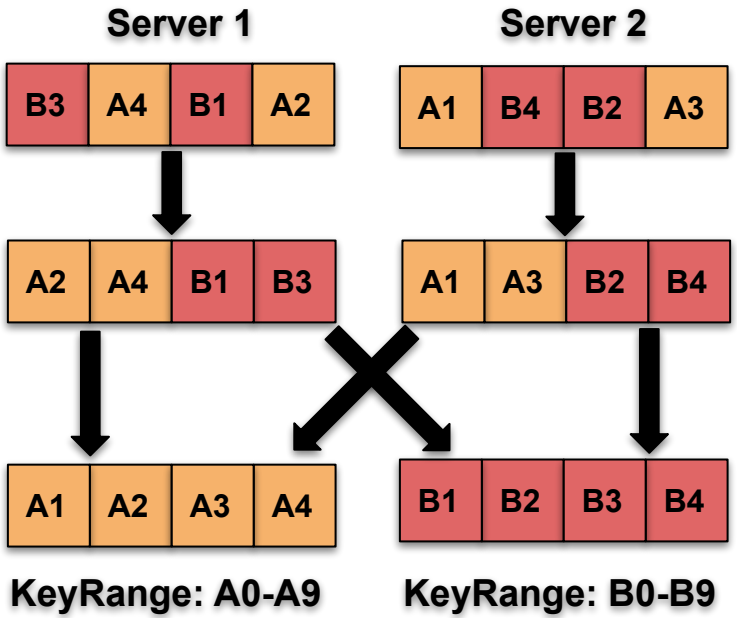
The MilliSort Challenge

- **How many small records can you sort in 1 ms using unlimited number of servers available in a datacenter?**
  - 100-byte records (10-byte keys and 90-byte values)
  - Input data already distributed evenly among servers in DRAM
  - Result data *not* required to be distributed evenly across servers



Background: Distributed Bucket Sort

- **Most distributed sorting algorithms are a form of bucket sort**
  - Local sort: each server sorts its initial data
  - Partitioning bucket boundaries: determine the key range each server stores after sorting
  - Shuffle data: each server transmits its records to the targets
- **Advantages:**
  - Optimize network bandwidth usage
  - Simple to implement



MilliSort: A Strawman Partition Scheme

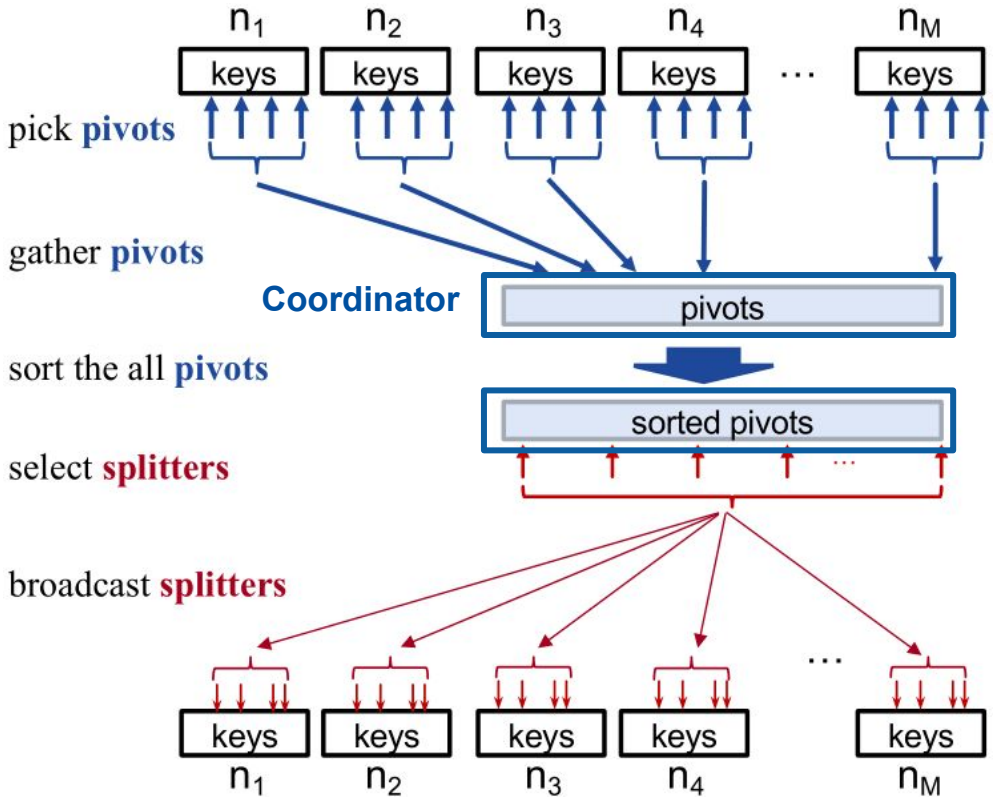
Terminology:

- Pivot: key chosen to divide local records
- Splitter: pivot chosen to be final bucket boundary

Parameters:

- **M**: number of machines
- **P**: number of pivots per server

If P = M, the skew factor of the final data bucket is at most 2.



MilliSort: A Strawman Partition Scheme

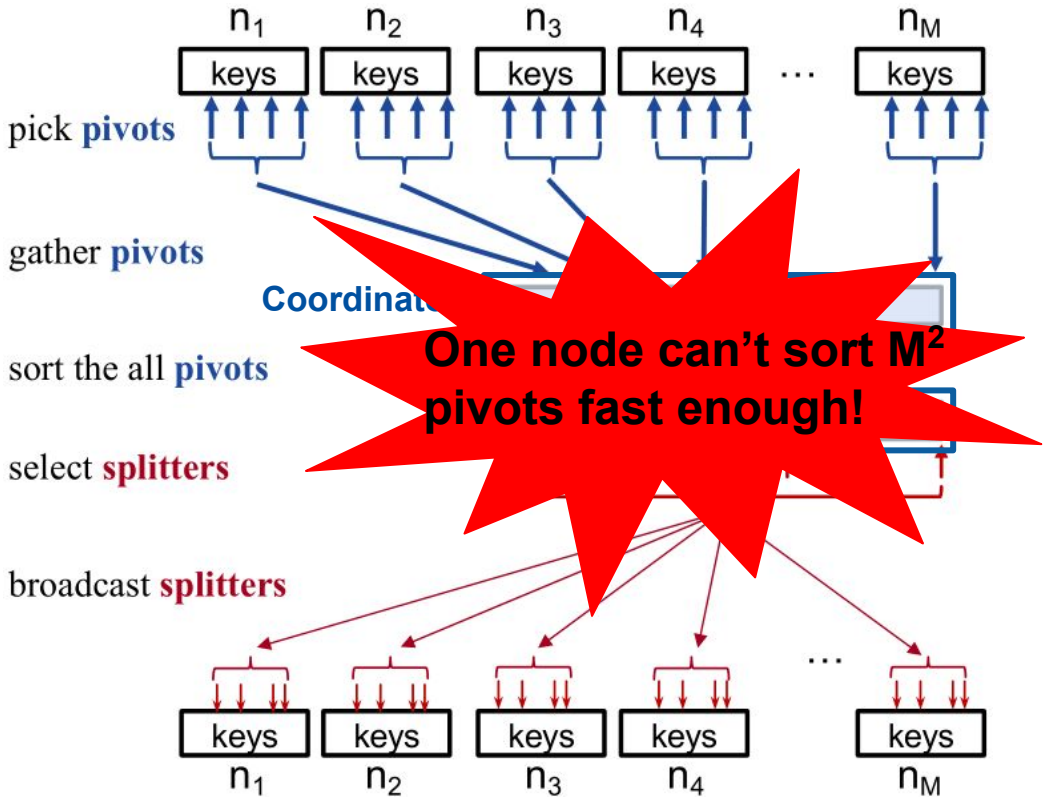
Terminology:

- Pivot: key chosen to divide local records
- Splitter: pivot chosen to be final bucket boundary

Parameters:

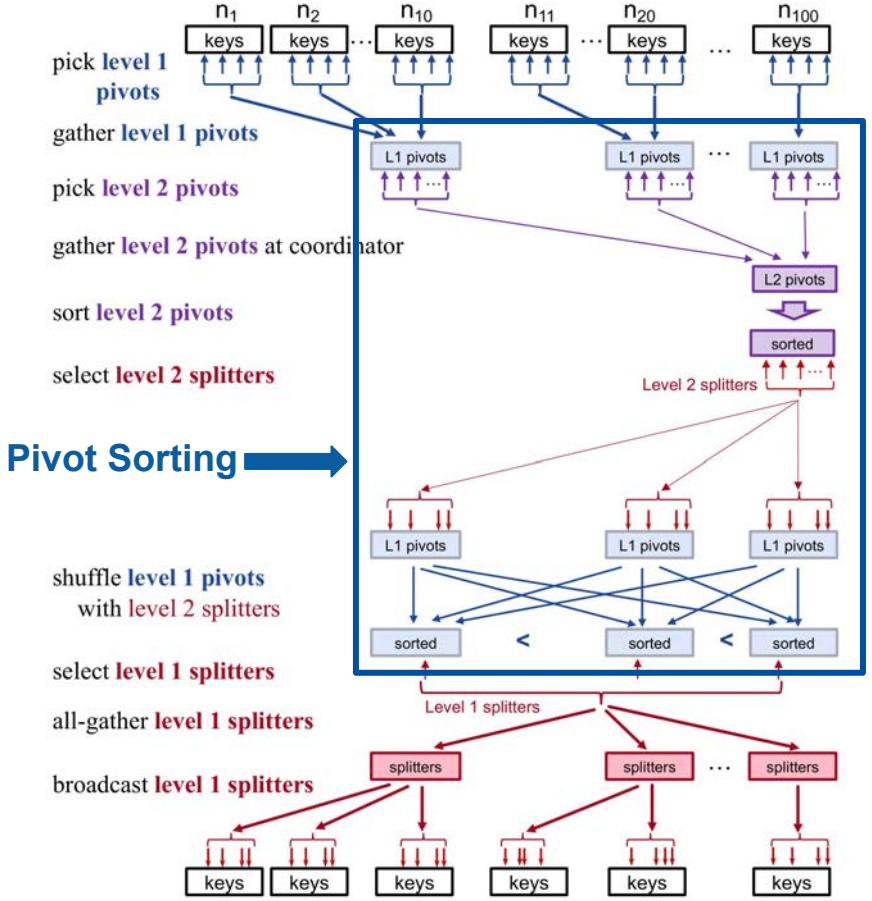
- **M**: number of machines
- **P**: number of pivots per server

If P = M, the skew factor of the final data bucket is at most 2.



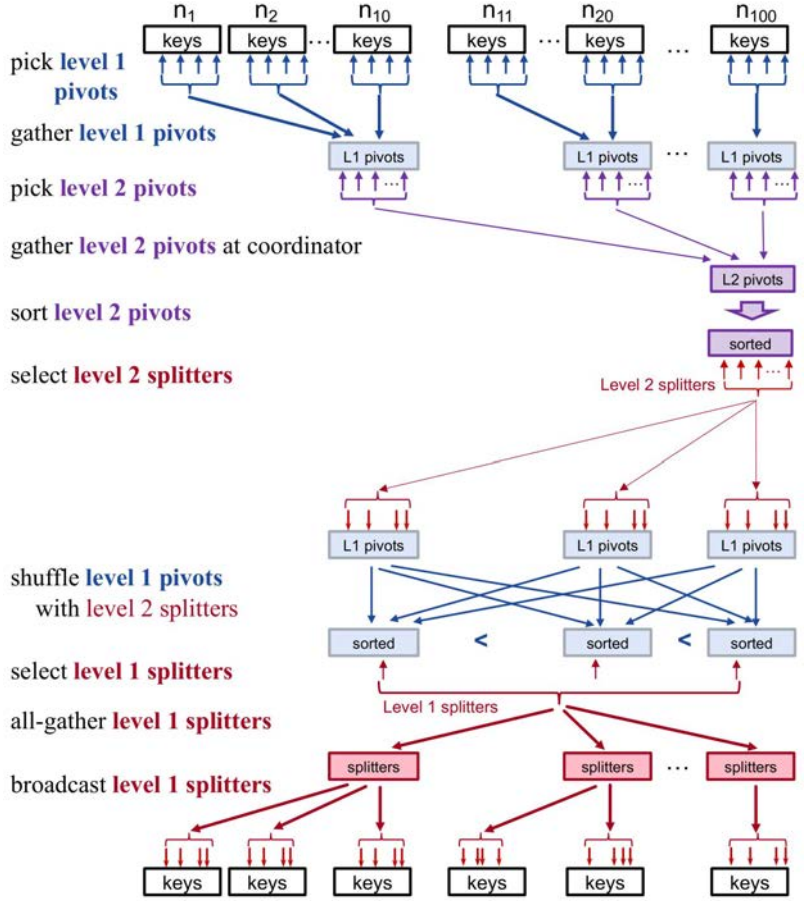
Recursive Partitioning

- **Key idea: sorting pivots is just a smaller version of MilliSort**
  - Apply distributed bucket sort again
  - Use a smaller set of servers
- **Recursive reduction:**
  - Assign one pivot sorter every **R** servers (# pivot sorters = M/R)
  - L2 pivot: chosen to divide L1 pivots on pivot servers
  - L2 splitter: L2 pivot chosen to be L1 pivot bucket boundary



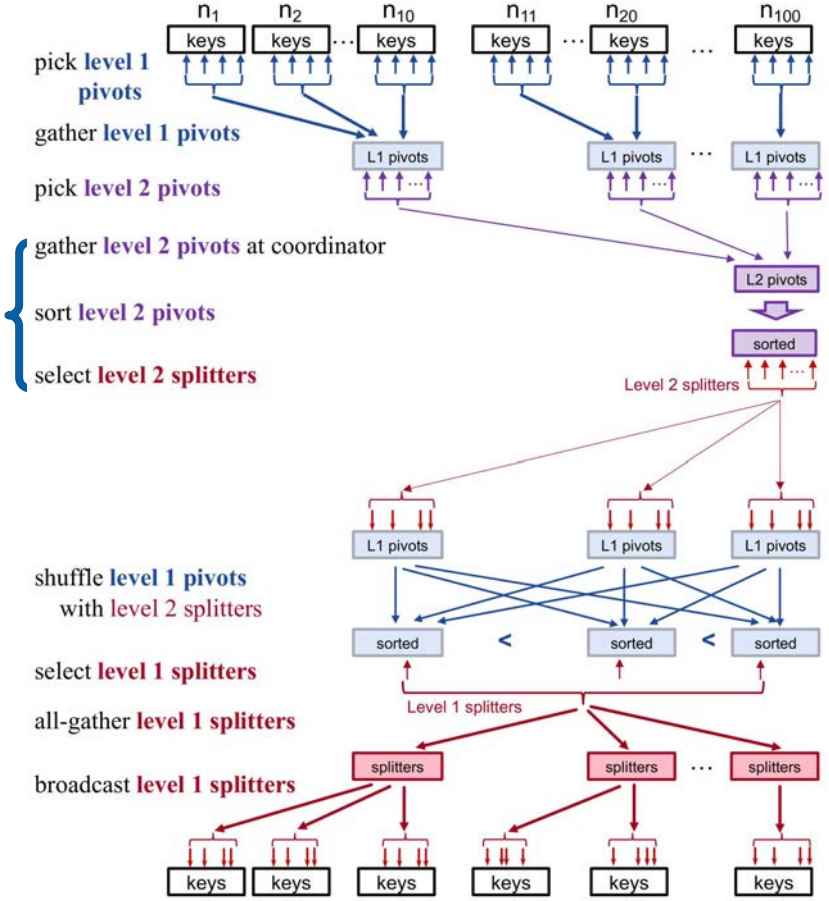
Recursive Partitioning

- **Key idea: sorting pivots is just a smaller version of MilliSort**
  - Apply distributed bucket sort again
  - Use a smaller set of servers
- **Recursive reduction:**
  - Assign one pivot sorter every **R** servers (# pivot sorters = M/R)
  - L2 pivot: chosen to divide L1 pivots on pivot servers
  - L2 splitter: L2 pivot chosen to be L1 pivot bucket boundary



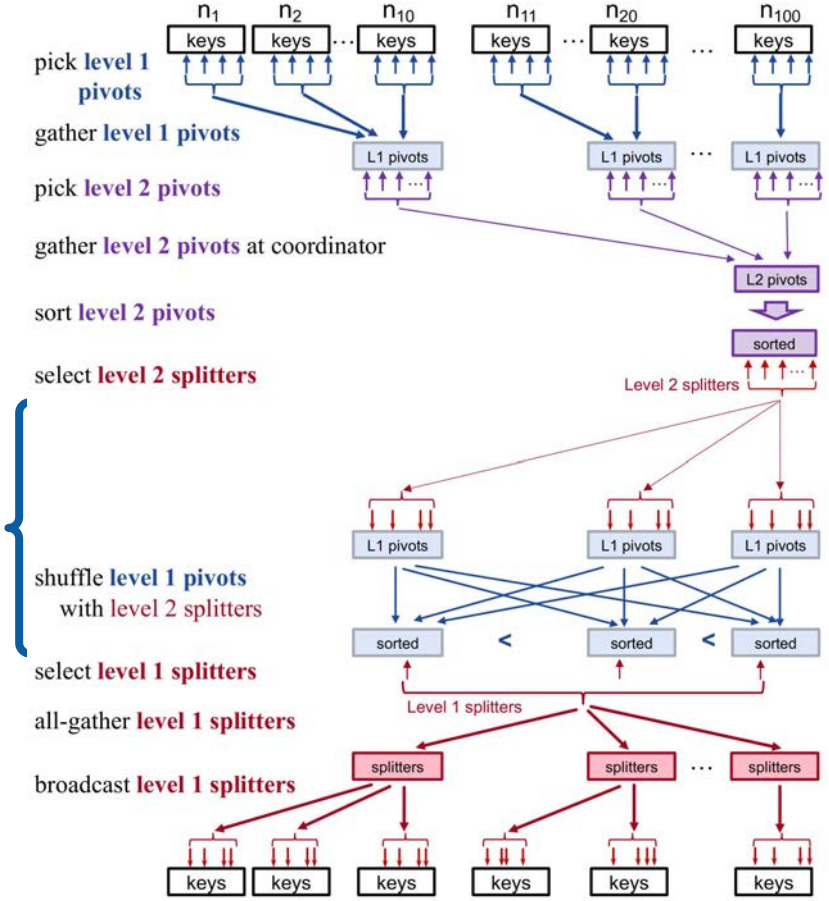
Recursive Partitioning

- **Key idea: sorting pivots is just a smaller version of MilliSort**
  - Apply distributed bucket sort again
  - Use a smaller set of servers
- **Recursive reduction:**
  - Assign one pivot sorter every **R** servers (# pivot sorters = M/R)
  - L2 pivot: chosen to divide L1 pivots on pivot servers
  - L2 splitter: L2 pivot chosen to be L1 pivot bucket boundary



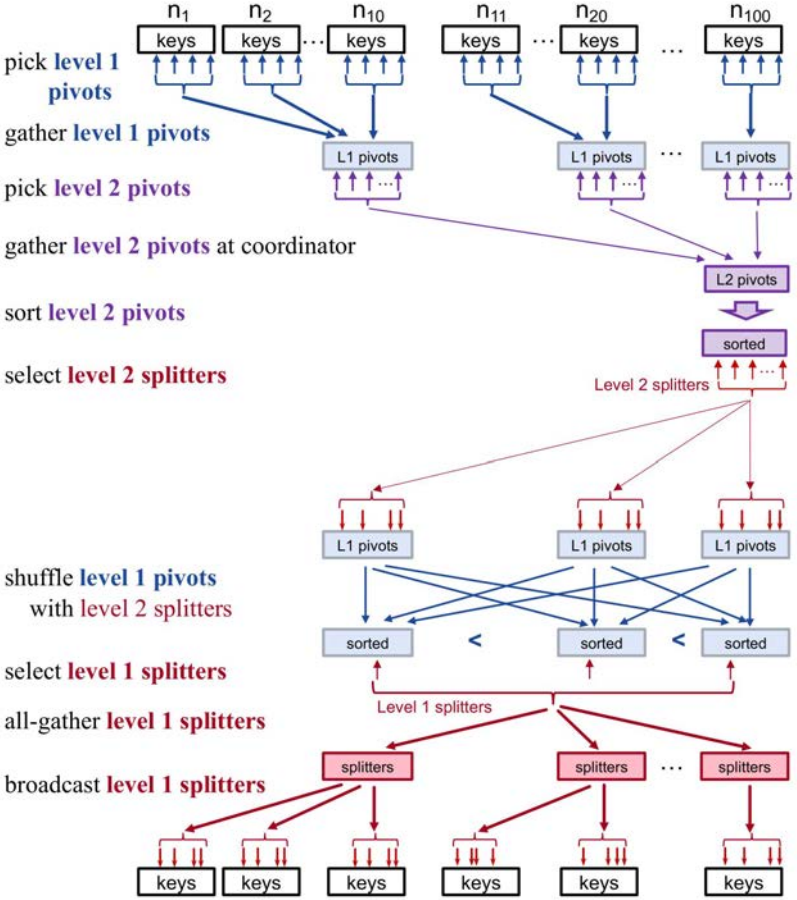
Recursive Partitioning

- **Key idea: sorting pivots is just a smaller version of MilliSort**
  - Apply distributed bucket sort again
  - Use a smaller set of servers
- **Recursive reduction:**
  - Assign one pivot sorter every **R** servers (# pivot sorters = M/R)
  - L2 pivot: chosen to divide L1 pivots on pivot servers
  - L2 splitter: L2 pivot chosen to be L1 pivot bucket boundary



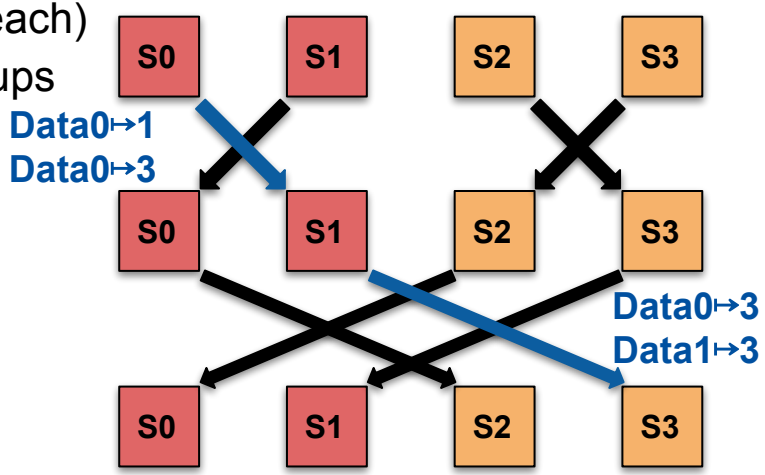
Recursive Partitioning

- **Key idea: sorting pivots is just a smaller version of MilliSort**
  - Apply distributed bucket sort again
  - Use a smaller set of servers
- **Recursive reduction:**
  - Assign one pivot sorter every **R** servers (# pivot sorters = M/R)
  - L2 pivot: chosen to divide L1 pivots on pivot servers
  - L2 splitter: L2 pivot chosen to be L1 pivot bucket boundary



Large-Scale Shuffle: Per-Message Cost Matters

- **As data is divided over more servers:**
  - Each node must send more messages
  - Each message gets smaller
  - Fixed per-message SW overhead limits performance eventually
- **Reduce # messages to send per server with 2-level shuffle**
  - Divide servers into √**M** groups (√**M** servers each)
  - Shuffle within groups & shuffle between groups
  - Each server sends fewer messages
  - Double network bandwidth usage



Outline

- Millisort Overview
- **Implementation & Cost Estimator**
- Measurements

Implementation

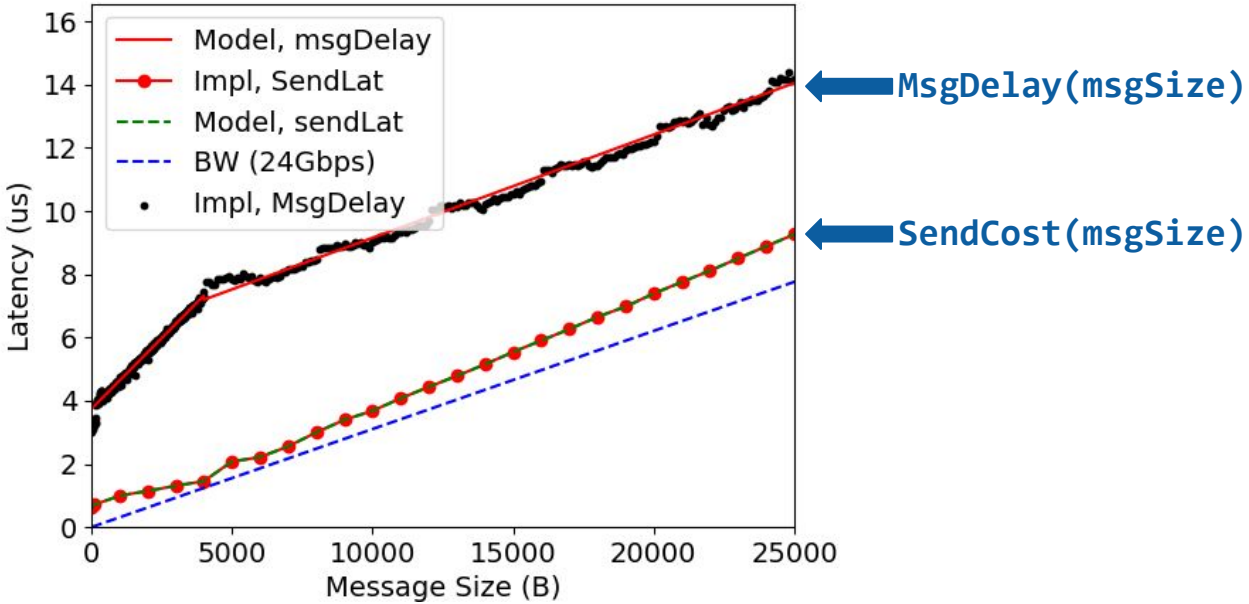
- **Prototype implementation using network transport infrastructure from RAMCloud**
  - Kernel bypass with DPDK or Infiniband Verbs: 5 µs RTT, 25 Gbps throughput
  - Arachne for user-level thread and core management
  - ~1500 lines of C++ code for group communication operations
  - ~3000 lines of C++ code for Millisort application
- **Limitations due to RAMCloud's dispatch model**
  - Require all outgoing/incoming messages to pass through a single dispatch thread
  - Single dispatch thread w/o batch send: ~1.6 million messages/second

Cost Estimator Overview

- **Goal: quickly explore a wide range of configurations**
  - Try larger system scales than are possible with the implementation
  - Find optimal configuration from the configuration space (e.g., # servers, # pivot sorters, # levels of partitioning, etc.)
  - Vary basic technology parameters (e.g., network speed, latency, software overhead, etc.)
- **Cost estimator implementation**
  - ~900 lines of Python code
  - Simulate MilliSort algorithm at a high level (i.e., a series of group comm. ops)
  - Estimate broadcast, gather, and all-gather cost using a message cost model
  - Shuffle cost is modeled differently, as a function of the shuffle message size

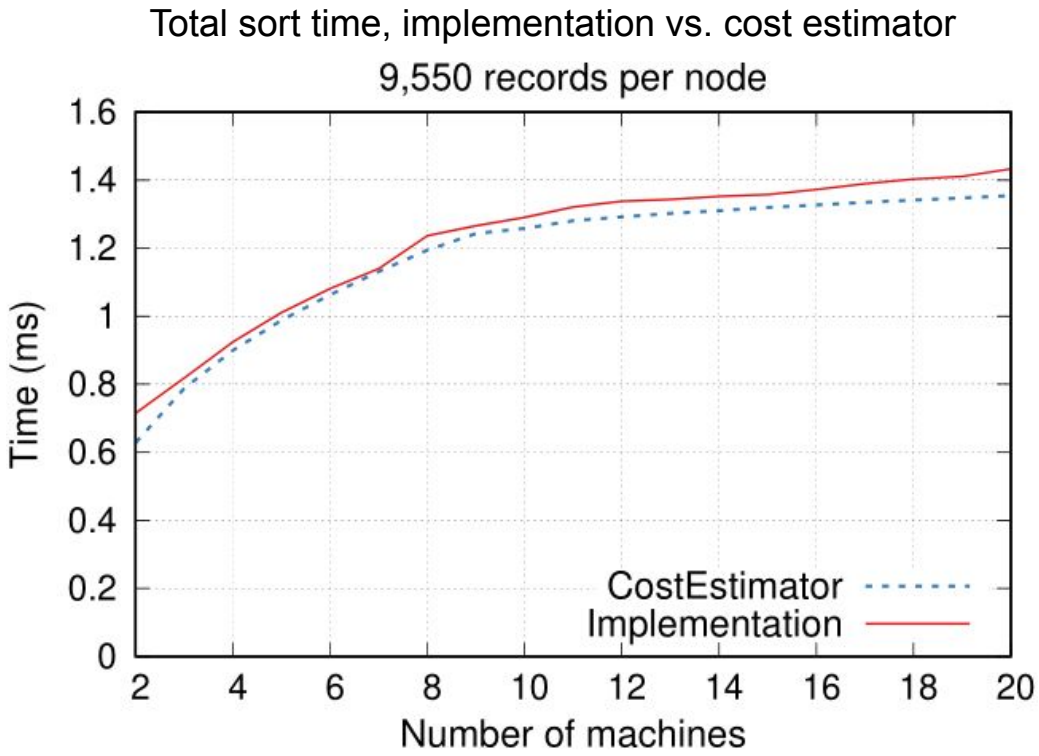
Model Calibration: Message Cost Model

- **Message cost modeled as a function of message size**
  - `MsgDelay(msgSize)`: one-way delay (incl. SW overhead)
  - `{Send, Recv}Cost(msgSize)`: marginal cost to send/receive a message
- **Works well for broadcast, gather, and all-gather operations**



Model Calibration: End-to-End

Cost estimator predicts total sort time quite accurately at small scale.



Outline

- Millisort Overview
- Implementation & Cost Estimator
- **Measurements**

How many records can you sort in 1 ms?

- **Cost estimator parameters**
  - 4 cores / node, 2 threads / core
  - 40 Gbps full bisection network
- **Within 1 ms, MilliSort can sort 4.6 million records using 700 servers**
  - ~600 ns CPU time (phys. core) to sort one record

How many records can you sort in 1 ms?

- **Cost estimator parameters**
  - 4 cores / node, 2 threads / core
  - 40 Gbps full bisection network
- **Within 1 ms, MilliSort can sort 4.6 million records using 700 servers**
  - ~600 ns CPU time (phys. core) to sort one record
- **Within 10 ms, MilliSort can sort 1 billion records using 13600 servers**

	1 ms	10 ms	Δ
Total records	4.6 million	1 billion	225X
# records/server	6600	76000	11.5X
# servers	700	13600	19.4X

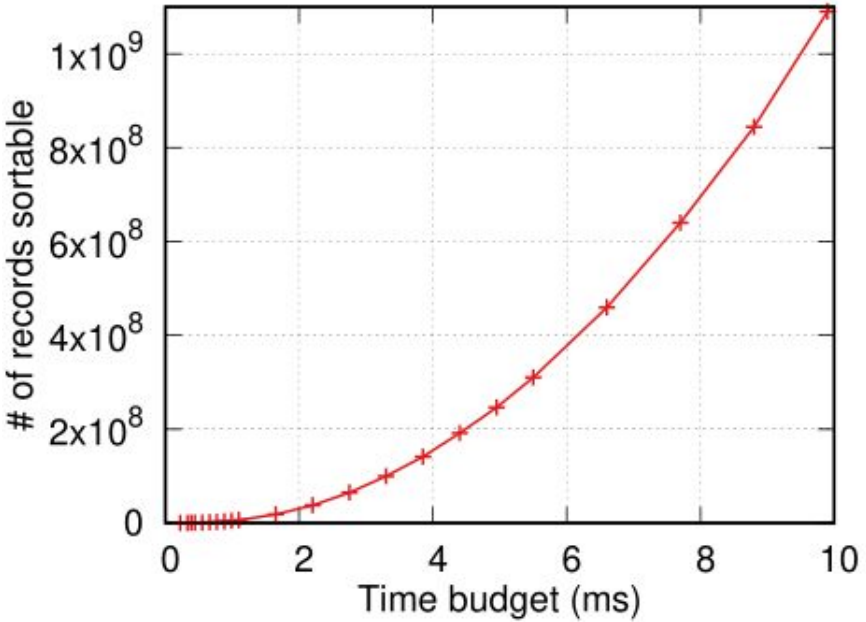
How many records can you sort in 1 ms?

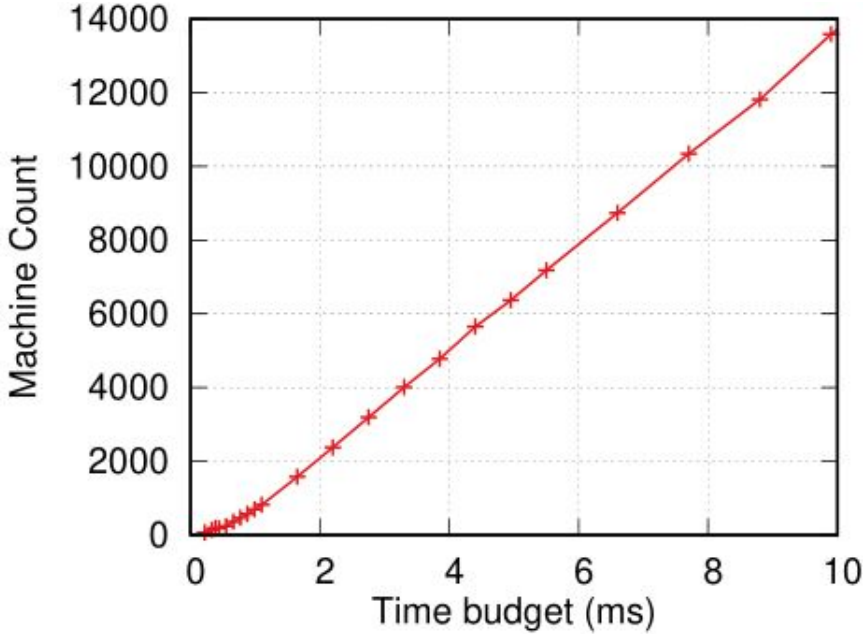
- **Cost estimator parameters**
  - 4 cores / node, 2 threads / core
  - 40 Gbps full bisection network
- **Within 1 ms, MilliSort can sort 4.6 million records using 700 servers**
  - ~600 ns CPU time (phys. core) to sort one record
- **Within 10 ms, MilliSort can sort 1 billion records using 13600 servers**

	1 ms	10 ms	Δ
Total records	4.6 million	1 billion	225X
# records/server	6600	76000	11.5X
# servers	700	13600	19.4X

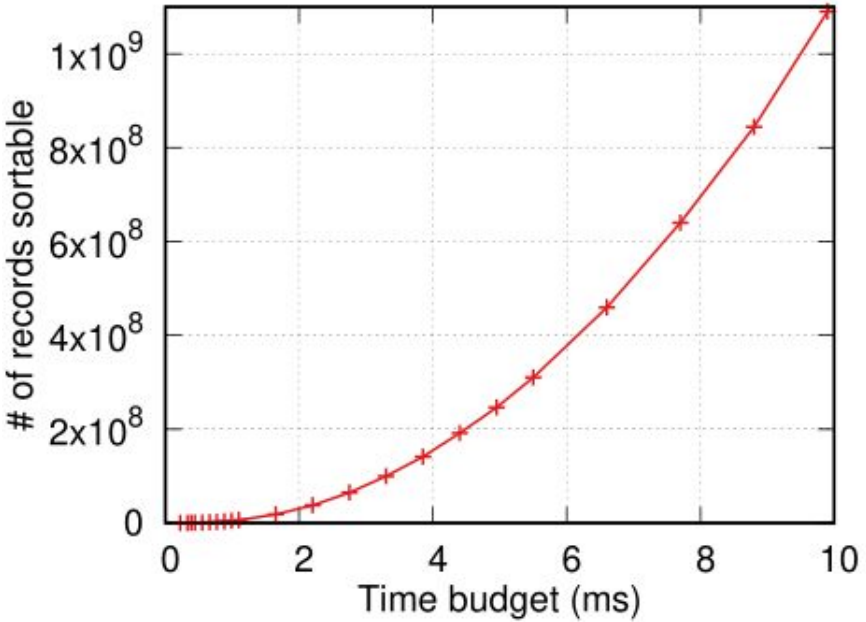
>100X since shuffle becomes more efficient

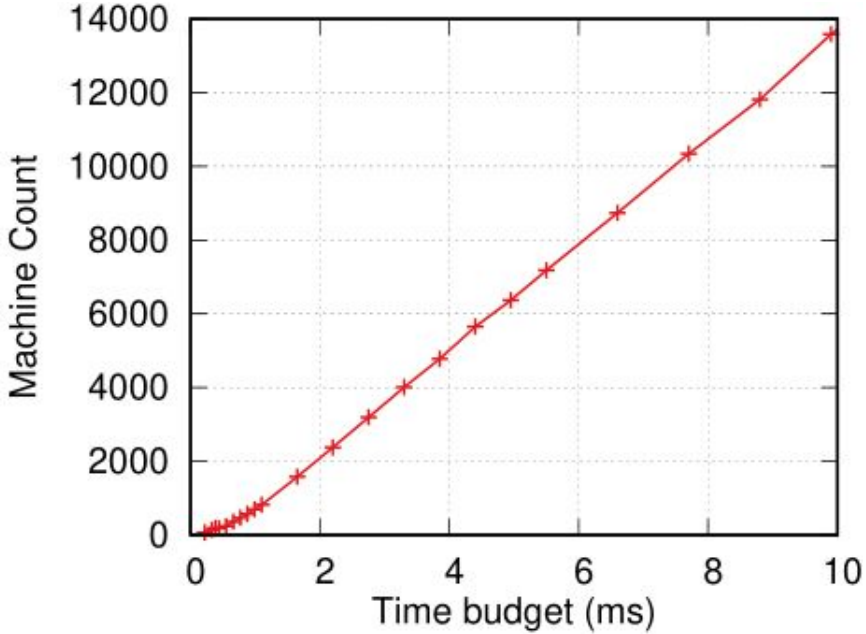
Impact of Time Budget on MilliSort





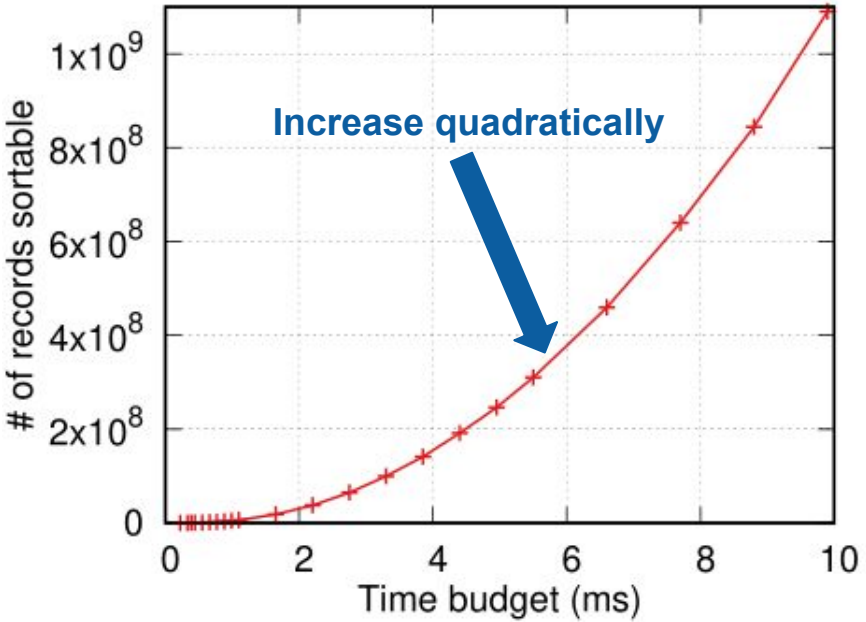
Impact of Time Budget on MilliSort

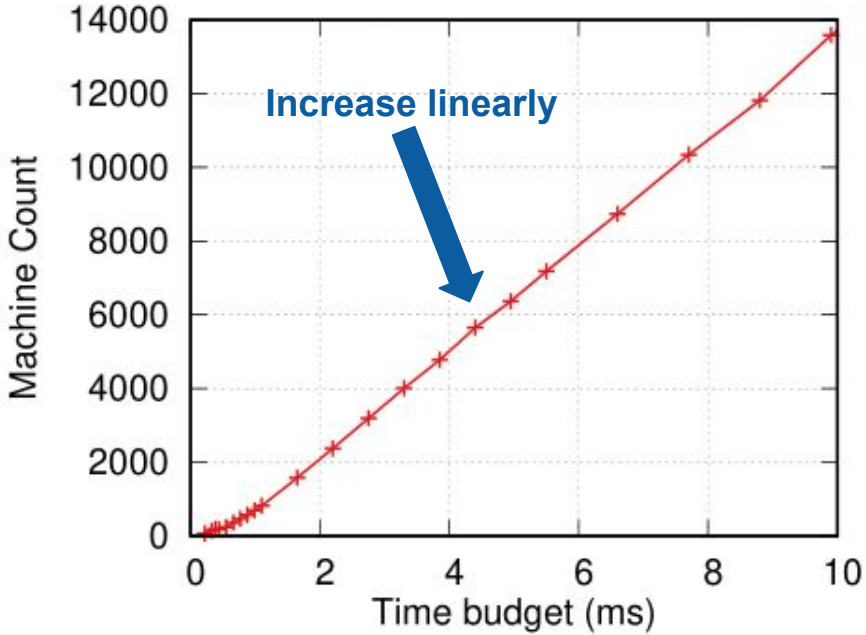




MilliSort can coordinate 1000s of machines for sorting in a few milliseconds

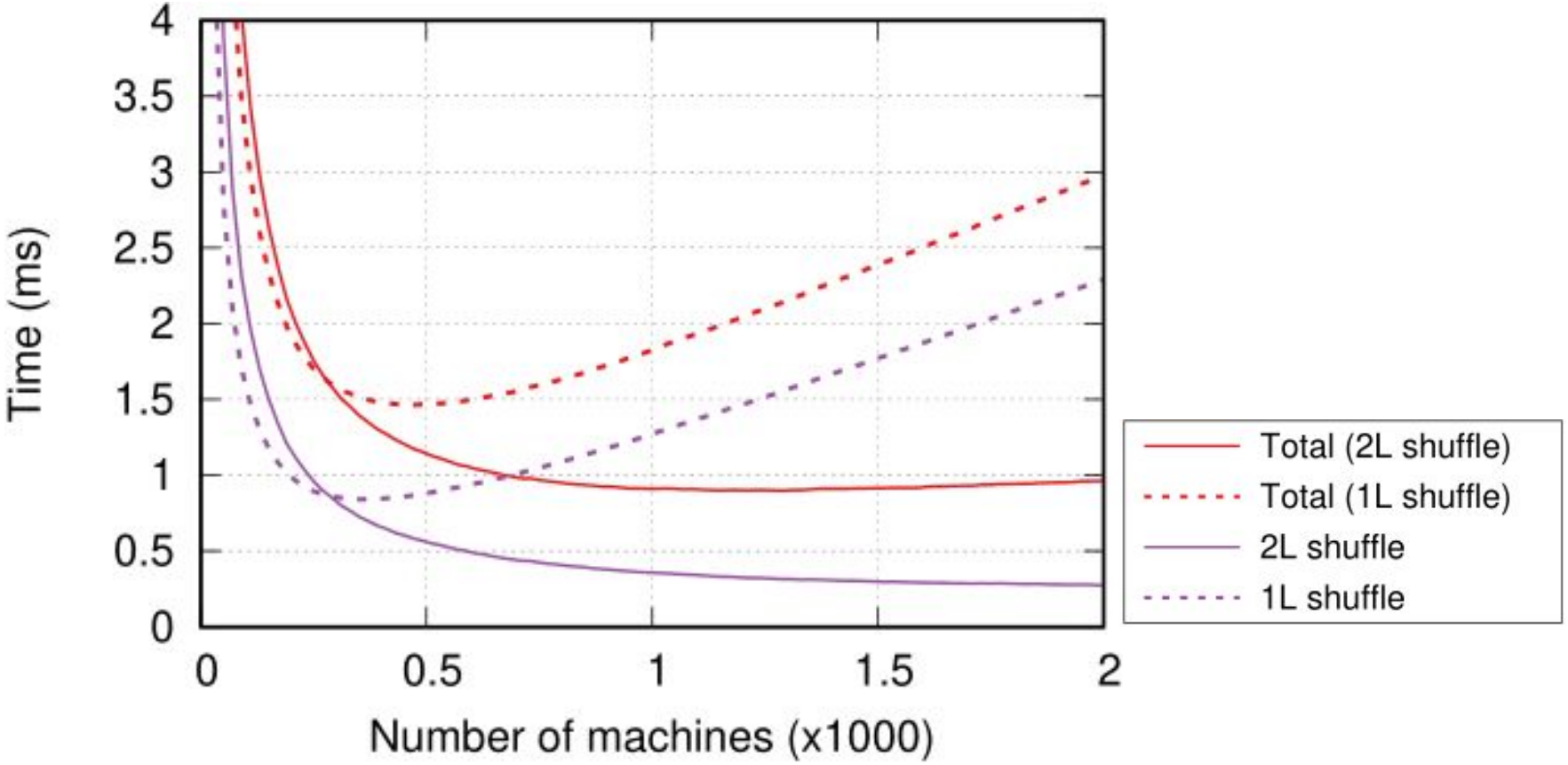
Impact of Time Budget on MilliSort



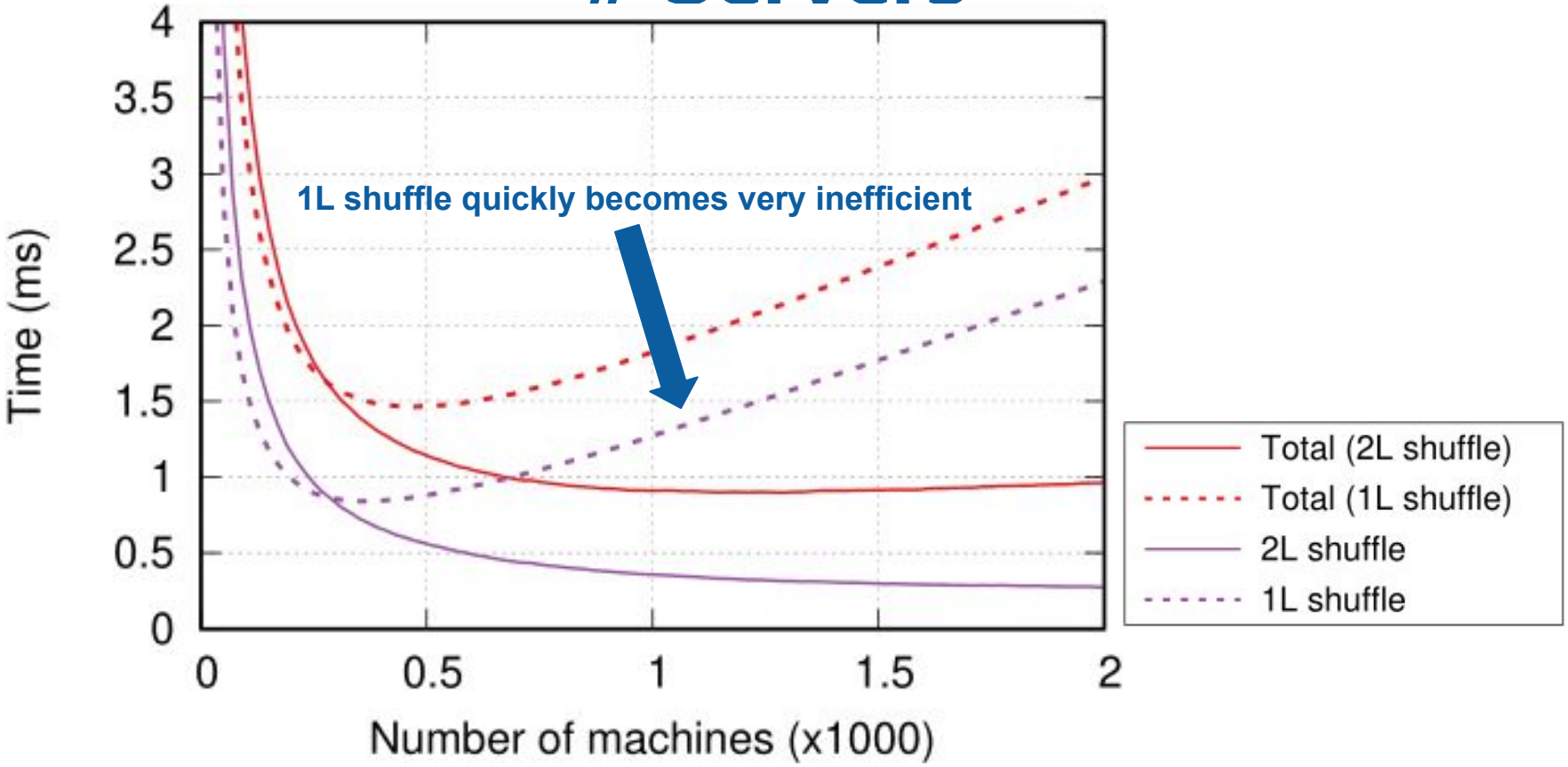


MilliSort can coordinate 1000s of machines for sorting in a few milliseconds

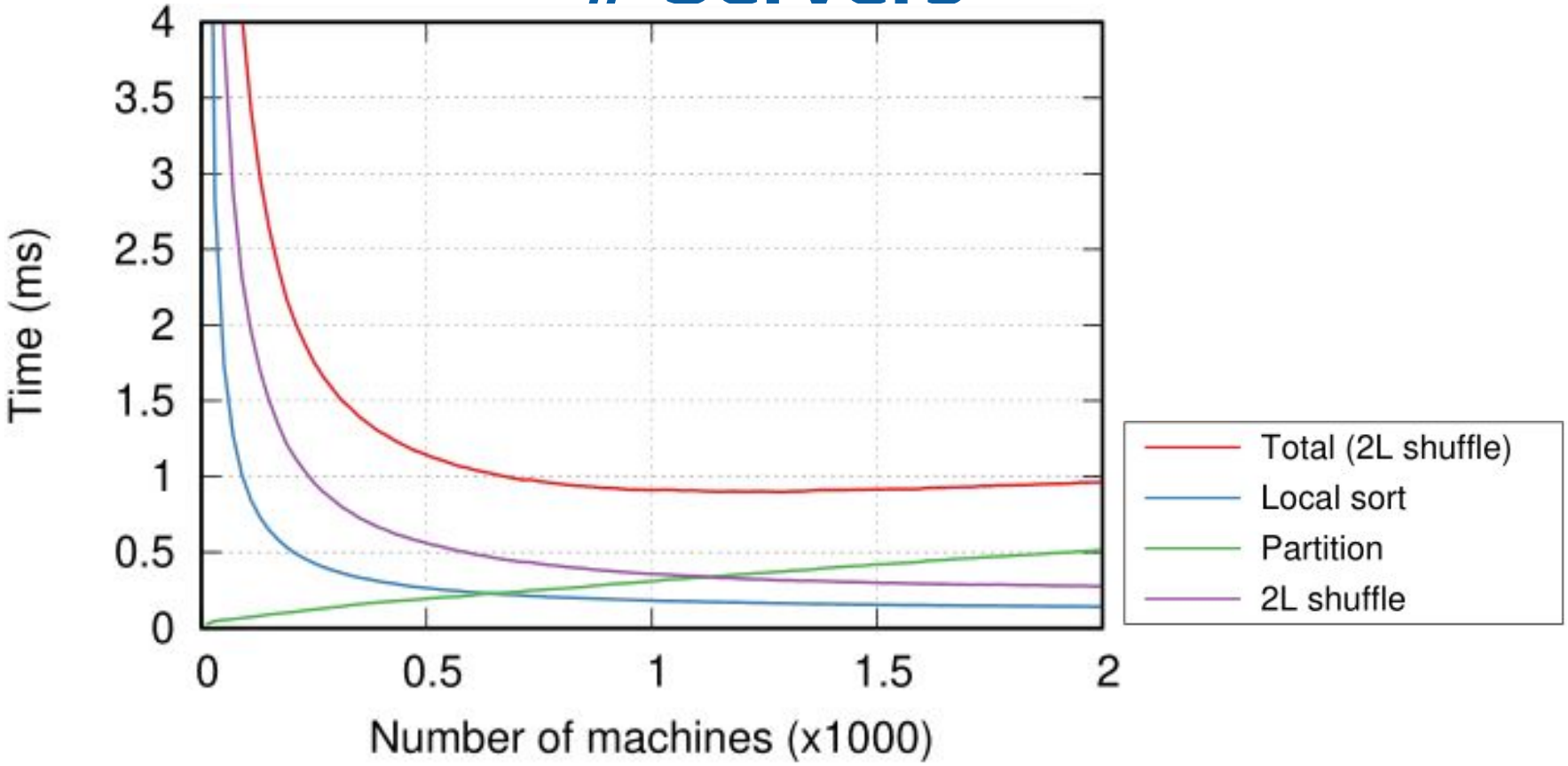
Time to Sort 4.6 Million Records, Vary # Servers



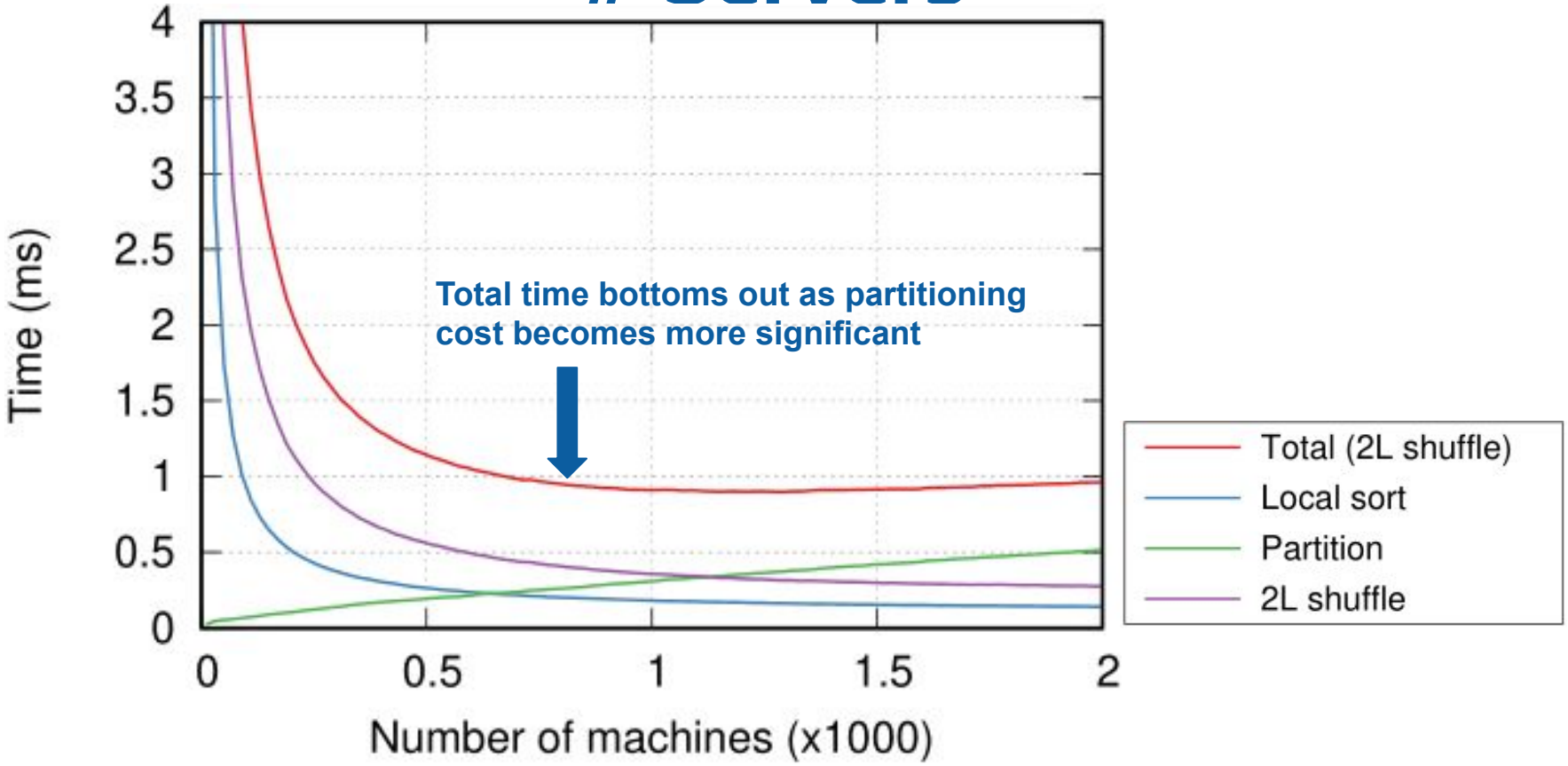
Time to Sort 4.6 Million Records, Vary # Servers



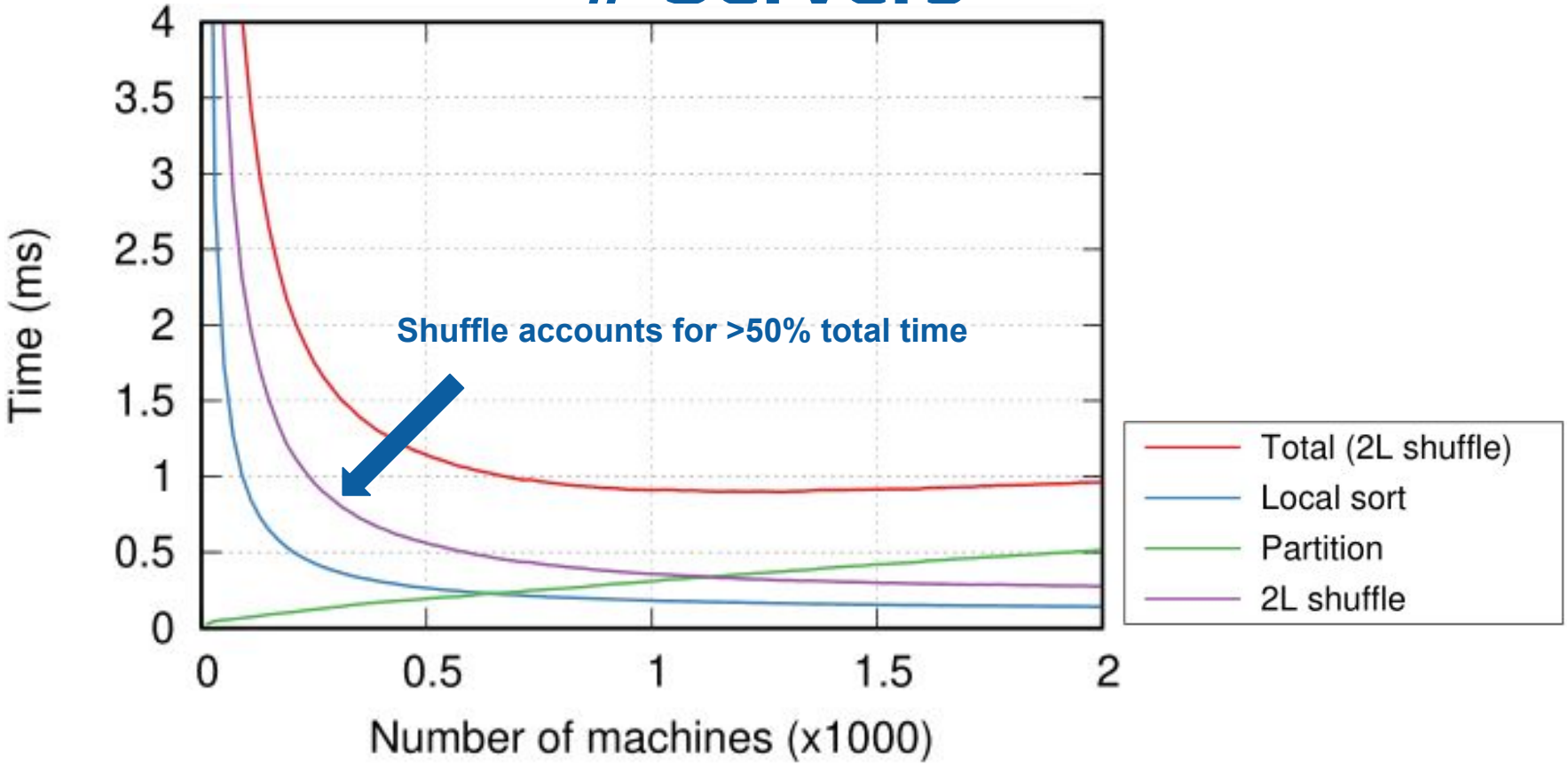
Time to Sort 4.6 Million Records, Vary # Servers



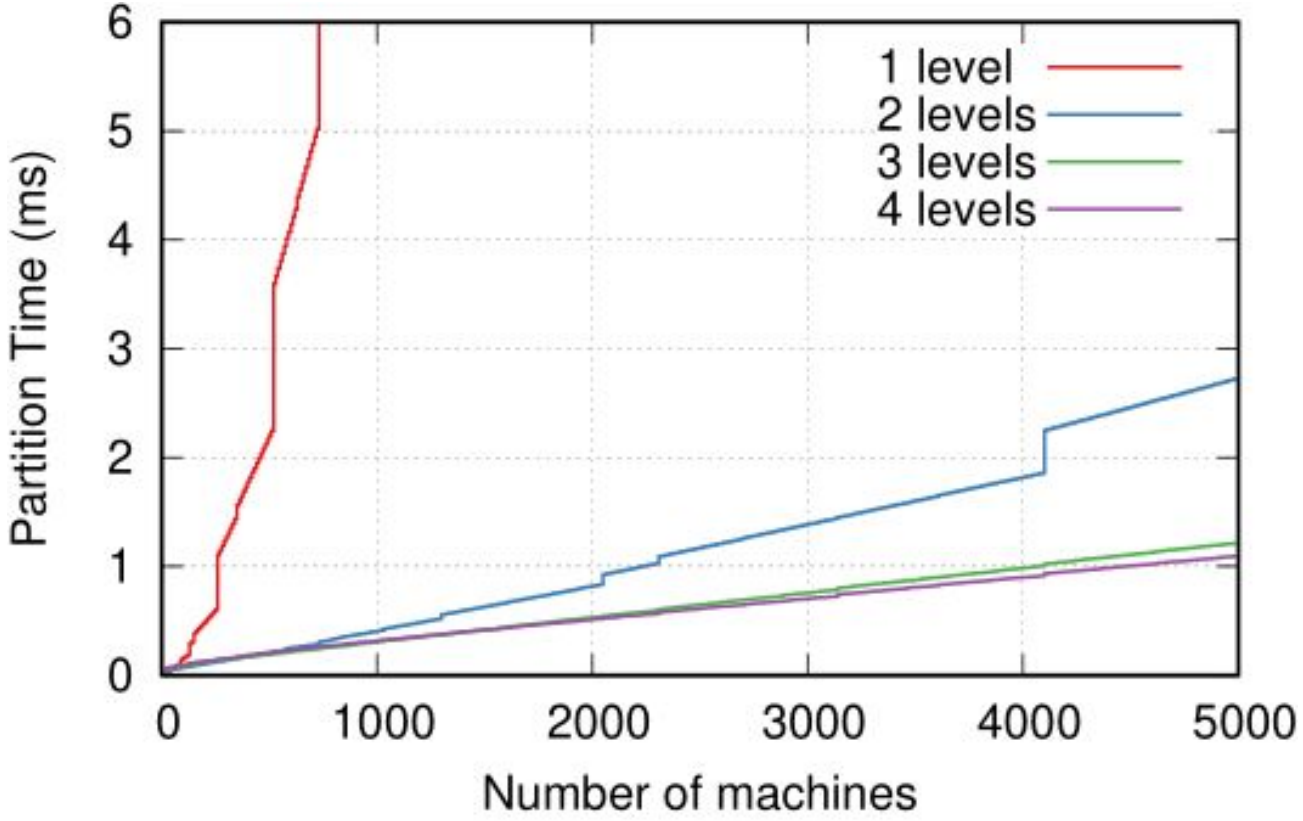
Time to Sort 4.6 Million Records, Vary # Servers



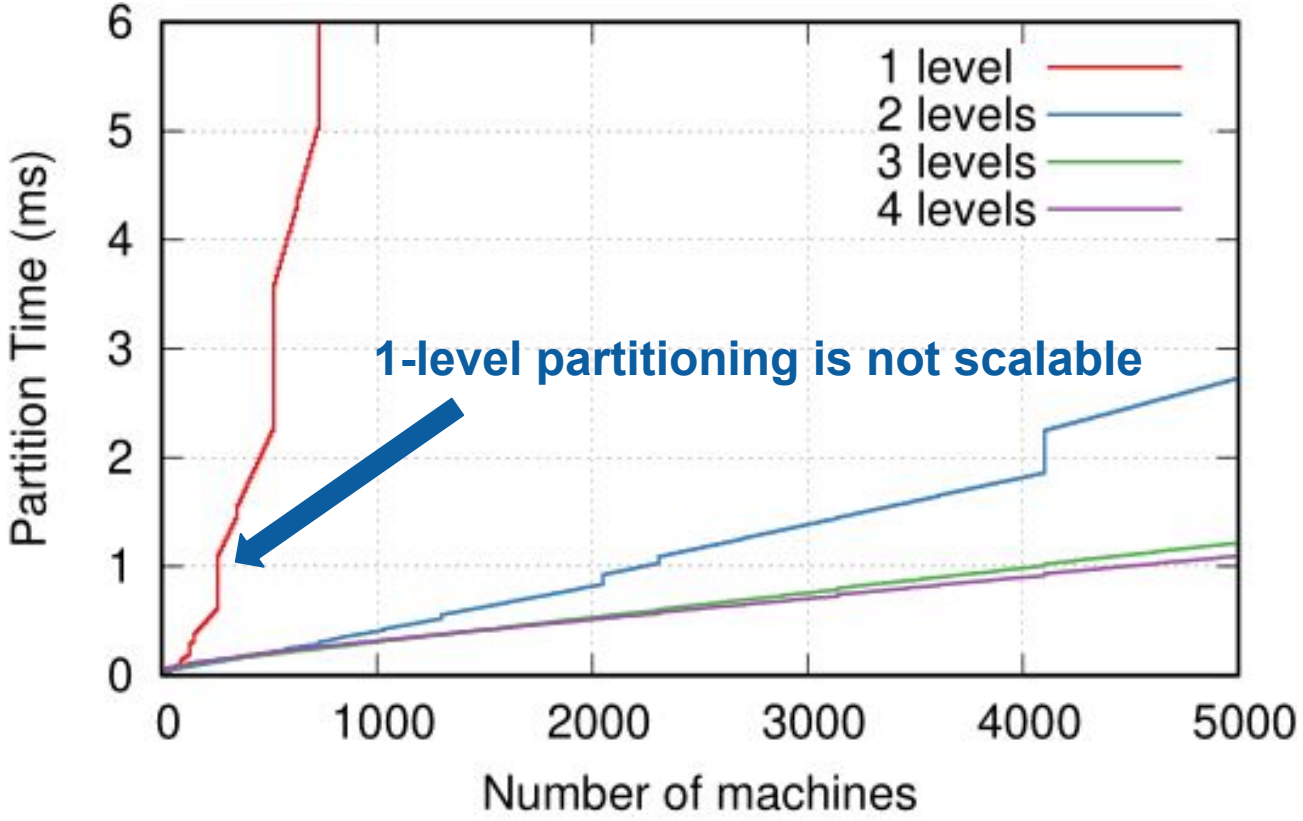
Time to Sort 4.6 Million Records, Vary # Servers



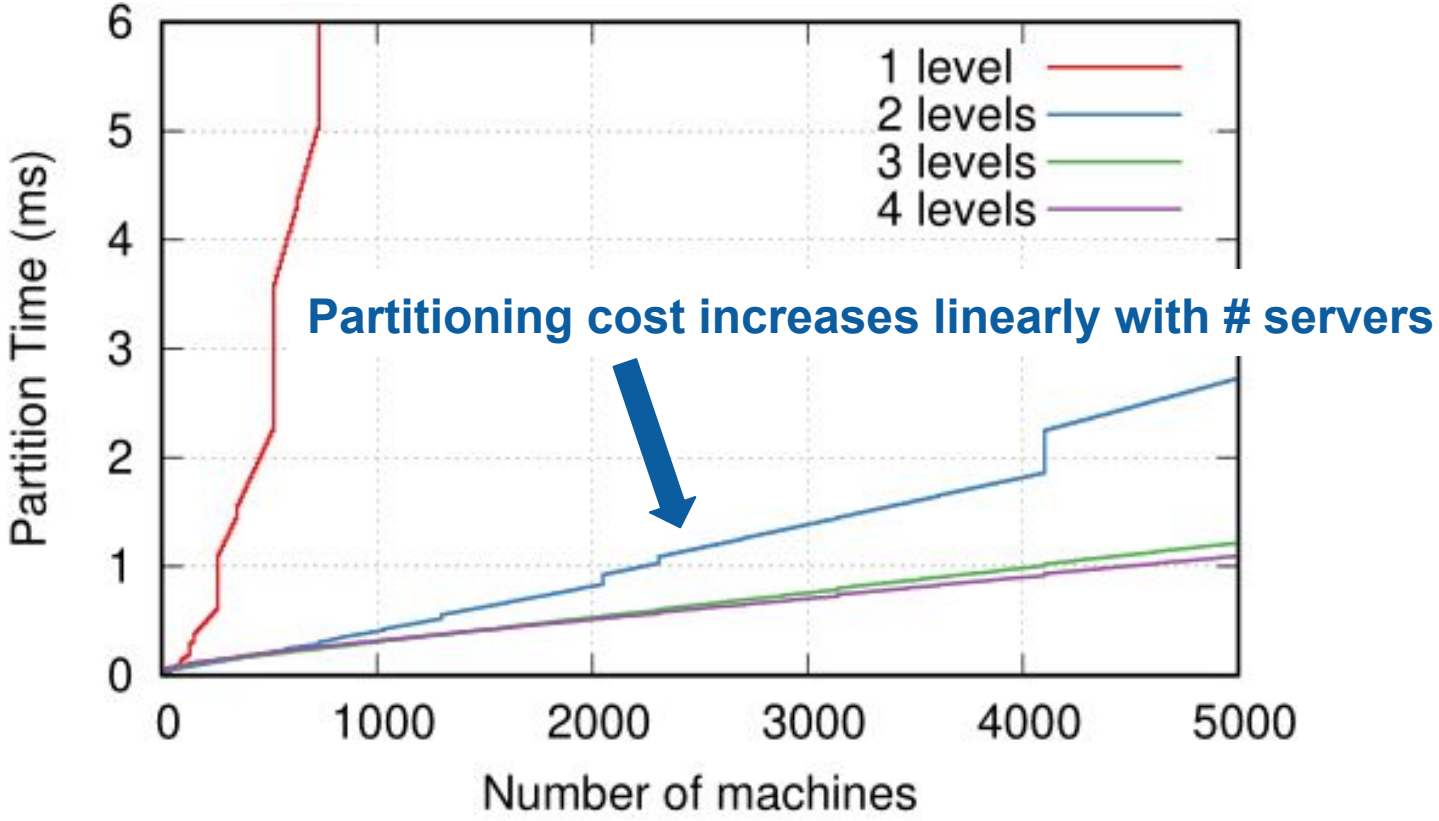
Multi-level Partitioning



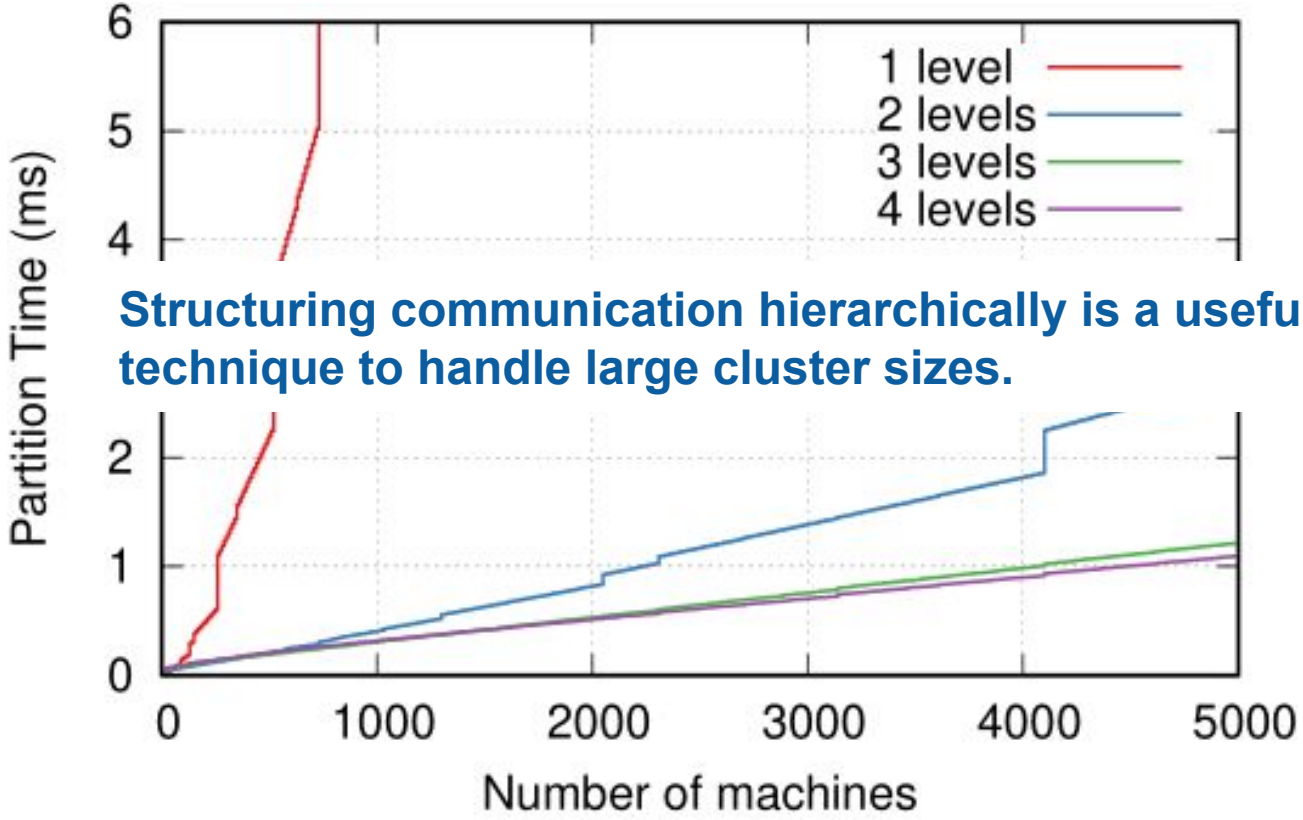
Multi-level Partitioning



Multi-level Partitioning



Multi-level Partitioning



Structuring communication hierarchically is a useful technique to handle large cluster sizes.

Conclusion

- **We developed MilliSort as an experiment to explore the notion of flash burst computation**
- **Flash burst**
  - Possible to harness 1000s of servers working together for a few milliseconds
  - Communication must be structured hierarchically to handle large cluster sizes
  - Full bisection bandwidth is necessary for best performance
- **Low-latency distributed sorting**
  - # records sortable grows quadratically with the time budget
  - Efficient group communication is essential, especially shuffle
  - Easier to scale # machines than per-server data

Questions?