Lecture: TritonSort

Adam Belay <abelay@mit.edu>

Logistics

- Course website schedule + papers updated
- Please sign up for two papers to present. Sign up sheet is posted on Piazza. Due this Wednesday
- Lab 1 will be assigned on Wednesday
- Office hours will be held Friday from 1PM-3PM
  - Get help with Lab 1
  - Meet to discuss projects
  - Meet to discuss paper presentations

Balanced systems

- Goal: Raw performance?
  - Yes, but not the primary focus
  - Prior systems achieved performance through scale
  - But wasteful: e.g., 94% disk and 33% CPU left idle
- Instead, goal is **balance**
  - Drive all resource usage as close as possible to 100%
  - Choose best hardware configuration
  - Architect software to use hardware as well as possible
- Benefits of balance
  - Makes the system significantly cheaper
  - Or handles significantly bigger problems with the same infrastructure

Why distributed sorting?

- Key primitive for large-scale data processing
- Used by Dryad, Hadoop, and Spark
- Exercises compute, network, and storage
- TritonSort focuses only on sorting
  - Missing data replication, node failure handling, and generalized computational model
  - But provides enough to study balanced compute

Problem formulation

- 100-byte tuples (10-byte key, 90-byte value)
  - Realistic? Values are often this small but not always uniformly sized (see Section 6.1).
- Goal: Sort 10s-100s TB of random tuples
  - Must transform input data into ordered set of output files, also stored on stable storage
  - Concatenation of output files must be fully sorted data
- TritonSort won the GreySort competition in 2014

Choosing balanced hardware

- Not easy: Can't buy a processor, for example, with exactly 13 cores
- TritonSort was constrained by cost
  - Infiniband provided more bandwidth but expensive
  - Flash provided more IOPS but expensive
  - Used 10GbE and rotating disks instead
  - Disk **seeks** expensive, software must avoid at all costs
  - But must still use disks, RAM is 70x more expensive
- Today: GreySort winners use Flash, 100GbE, and even RAM; all have become more affordable

Hardware pricing

Storage			
Type	Capacity	R/W throughput	Price
7.2k-RPM	500 GB	90-100 MBps	$200
15k-RPM	150 GB	150 MBps	$290
SSD	64 GB	250 MBps	$450

Network	
Type	Cost/port
1 Gbps Ethernet	$33
10 Gbps Ethernet	$480

Server	
Type	Cost
8 disks, 8 CPU cores	$5,050
8 disks, 16 CPU cores	$5,450
16 disks, 16 CPU cores	$7,550

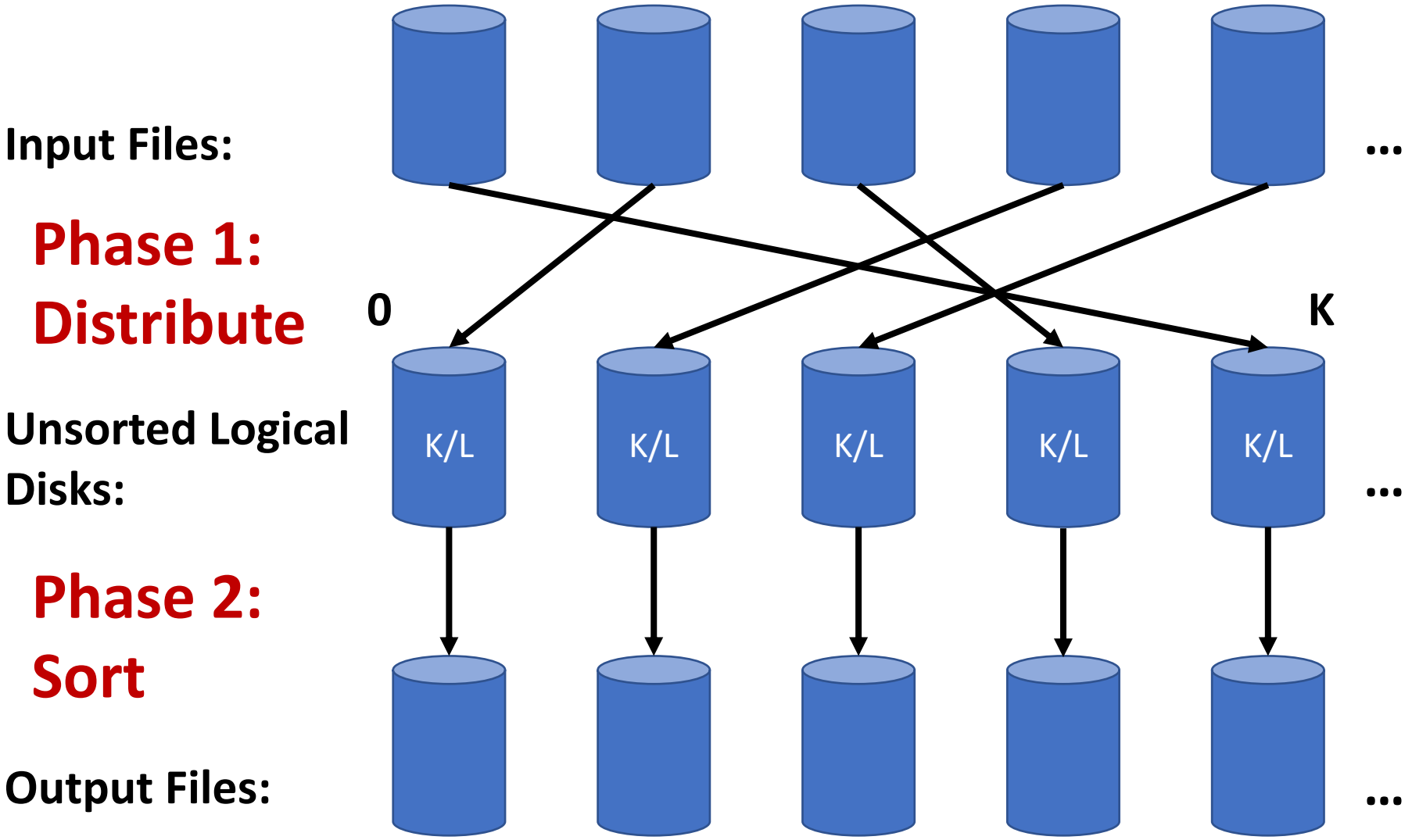
- Each 7200RPM drive provides ~1Gbps
- TritonSort used 16 7,200RPM disks + 8 CPU cores, 24GB RAM (increased from 12GB)
- Can all disks be exposed over network at same time?

Q: Why is provisioning the right hardware configuration difficult?

E.g., Why did they not know at first, they needed 24GB RAM?

Q: How could we decouple the hardware configuration from the workload?

TritonSort software architecture



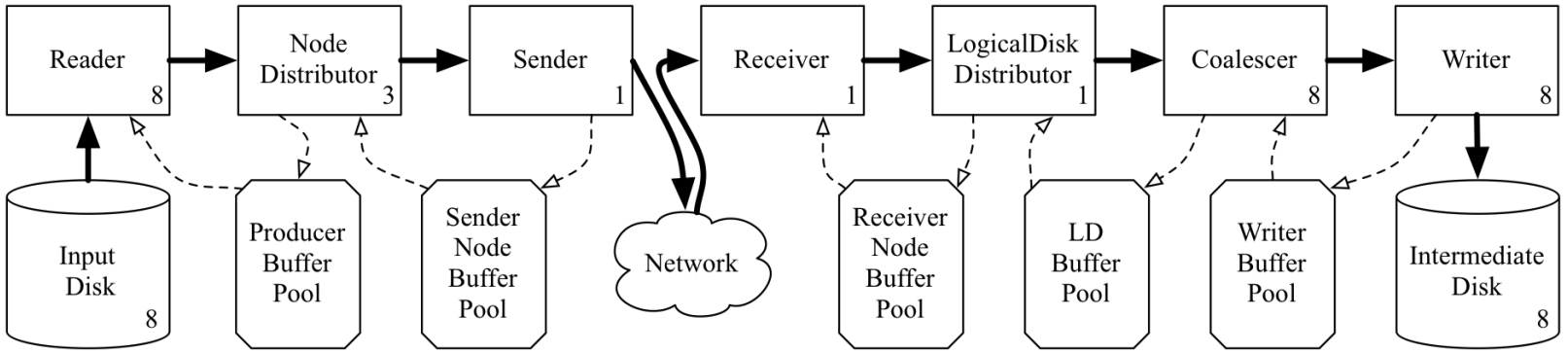
Buffering

- Most of 24 GB per machine is spent on buffering
- TritonSort buffers between every stage. Why?
  - Logically decouples each processing stage
  - Absorbs uneven processing pace over time
  - Storage reads and writes must be large to reduce seeks
- Typically higher latency or unpredictable performance demands more buffering
  - Opportunity: Flash requires significantly less buffering than disk
  - Pitfall: Buffering can increase latency further, demanding yet even more buffering! See BufferBloat.
- TritonSort uses token-based flow control! Enables blocking when out of buffers; can't run out of memory!

Data copying

- TritonSort copies data multiple times
- Conventional wisdom: Zero-copy is faster!
- Why is zero-copy a bad plan for TritonSort?
- Could some copies still be eliminated?

Phase 1: Distributing



- Reader: Reads from input disk (80MB at a time)
- N. Distributor: Partitions tuples, decides where to send each tuple
- Sender: Sends tuples over network
- Receiver: Receives tuples over network
- L. Distributor: Groups together tuples into LDBuffer arrays destined for each logical disk
- Coalescer: Combines LDBuffers into bigger chunks
- Writes: Serializes chunks to disk

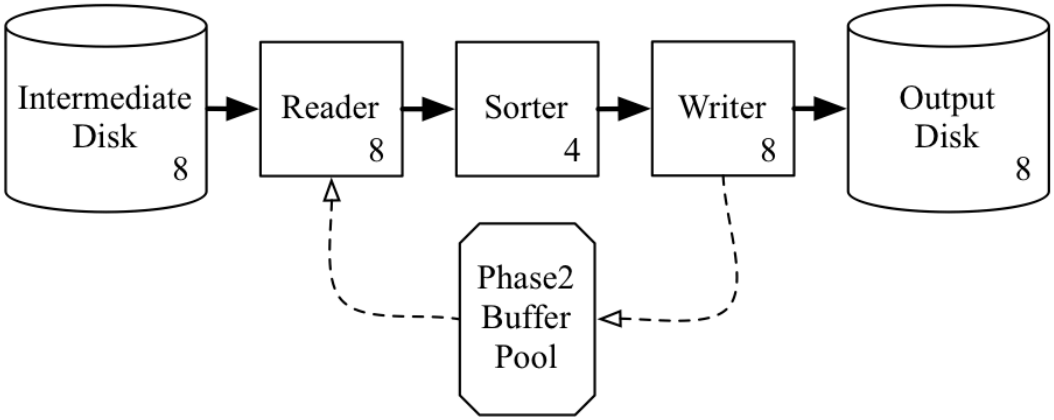
LogicalDisk distributor code

Algorithm 1 The LogicalDiskDistributor stage

```
1:  NodeBuffer ← getNewWork()
2:  {Drain NodeBuffer into the LDBufferArray}
3:  for all tuples t in NodeBuffer do
4:      dst = H(key(t))
5:      LDBufferArray[dst].append(t)
6:      if LDBufferArray[dst].isFull() then
7:          LDTable.insert(LDBufferArray[dst])
8:          LDBufferArray[dst] = getEmptyLDBuffer()
9:      end if
10: end for
11: {Send full LDBufferLists to the Coalescer}
12: for all physical disks d do
13:     while LDTable.sizeOfLongestList(d) ≥ 5MB do
14:         ld ← LDTable.getLongestList(d)
15:         Coalescer.pushNewWork(ld)
16:     end while
17: end for
```

Q: TritonSort did not have a coalescer stage originally. Why didn’t this design work?

Phase 2: Sort



- Reader: Reads logical disks from storage
- Sorter: Uses RadixSort to sort each logical disk
- Writer: Writes sorted logical disks to storage

Each logical disk can be sorted in parallel!

Q: In which ways did Linux fail to provide what TritonSort needed?

OS-level bottlenecks

- Threading overhead forced single threads for many processing stages, limiting parallelism
- OS scheduler wasted ~5% of CPU, requiring thread pinning
- Block layer dramatically increased latency when given larger writes, requiring small chunked writes instead
- Poor transport-level congestion control required software to rate-limit data transmission
- All of these shortcomings dramatically increased software complexity and added additional layers
  - Faster, more predictable OSes require less code to use effectively!

Example: Poor thread scaling

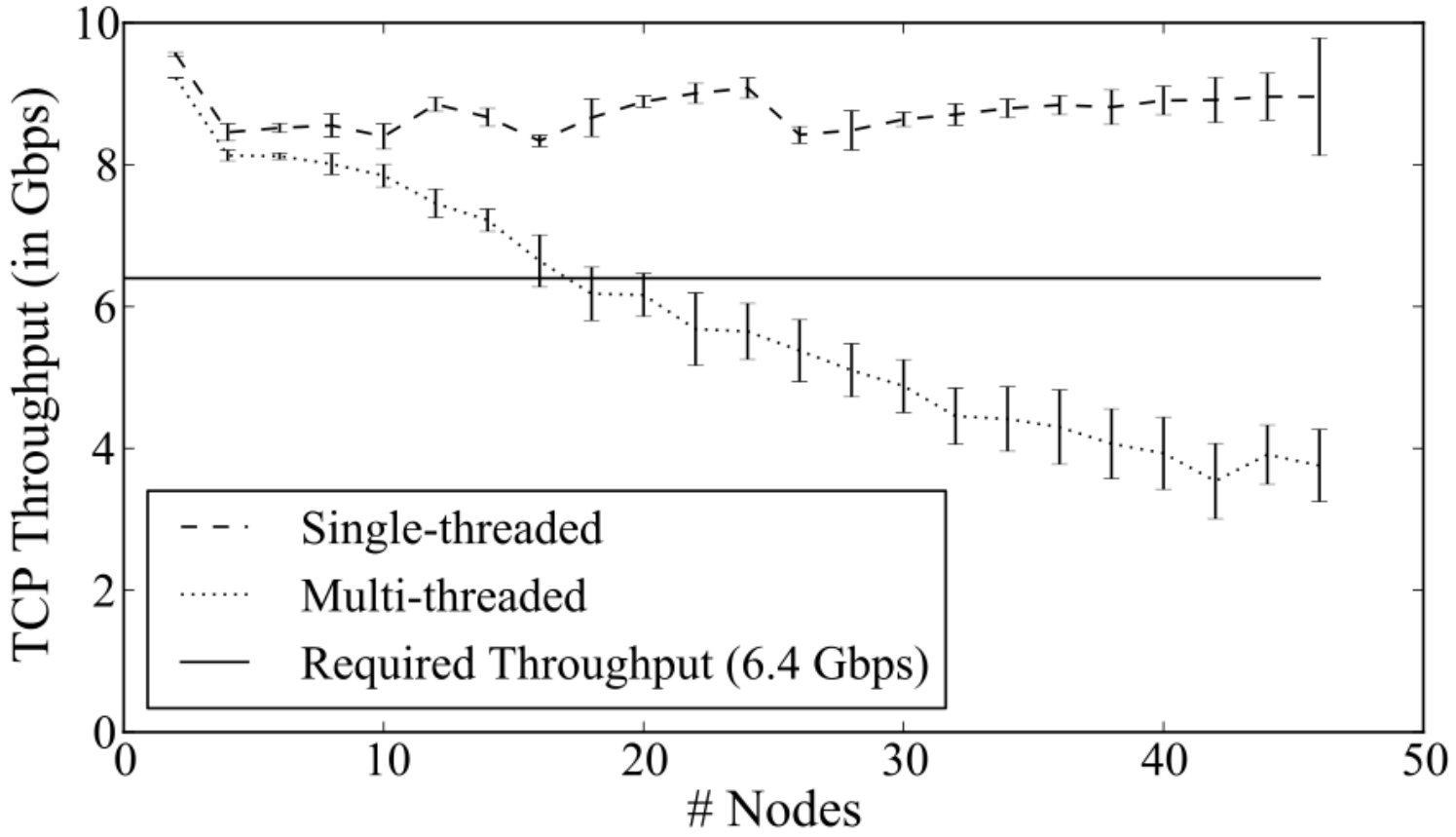
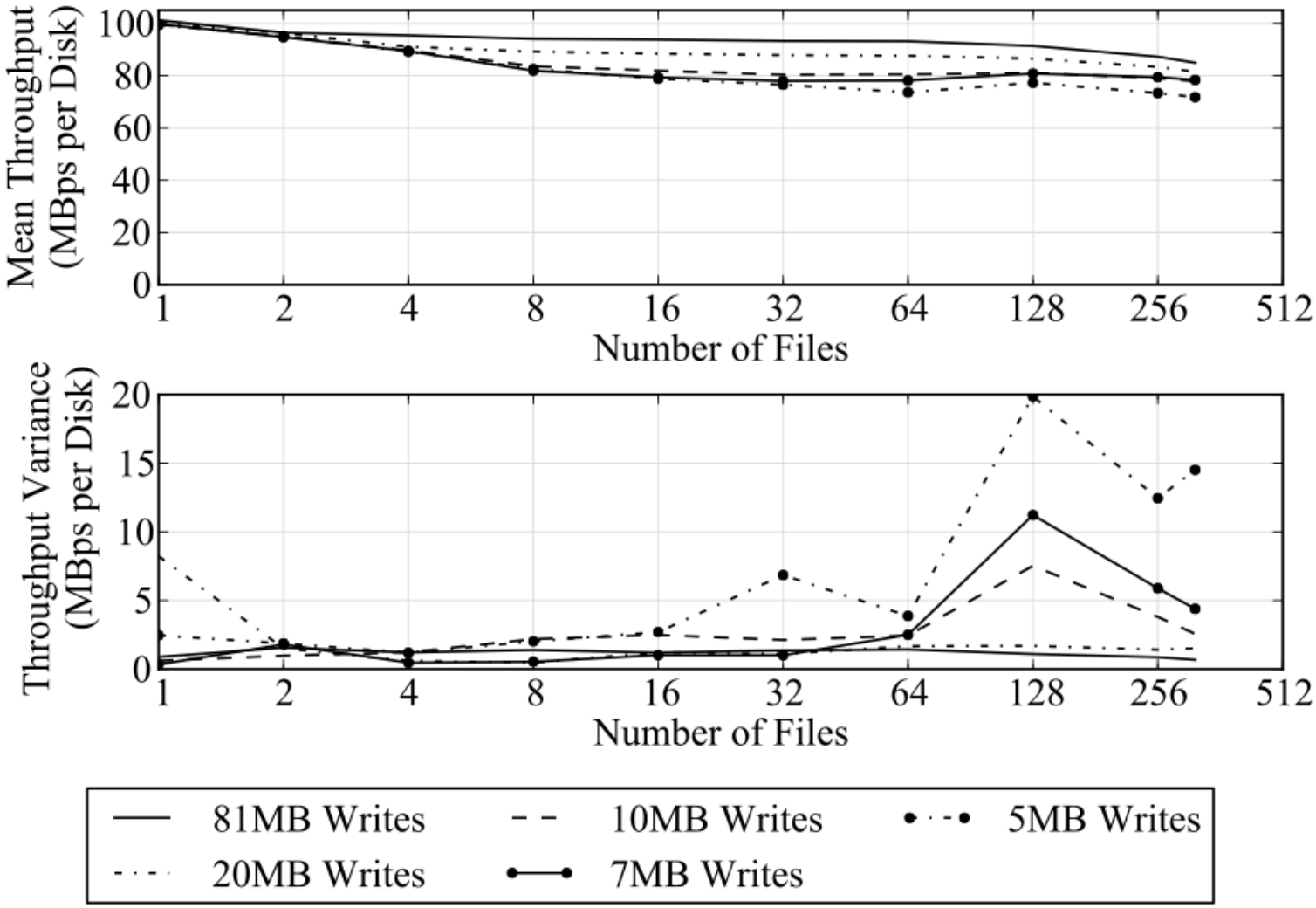


Figure 9: Comparing the scalability of single-threaded and multi-threaded Receiver implementations

Storage optimizations

- Disk seeks are expensive, larger reads and writes reduce them, with diminishing returns
- Logical disks are 80MB, too large to fully buffer before writing
- Forced to intermix writes to different logical disks
- Two possible designs
  - Use log-structure to append each write
    - Increases seeks during read
  - Use fallocate() to reserve blocks for each logical disk
    - Increases seeks during write

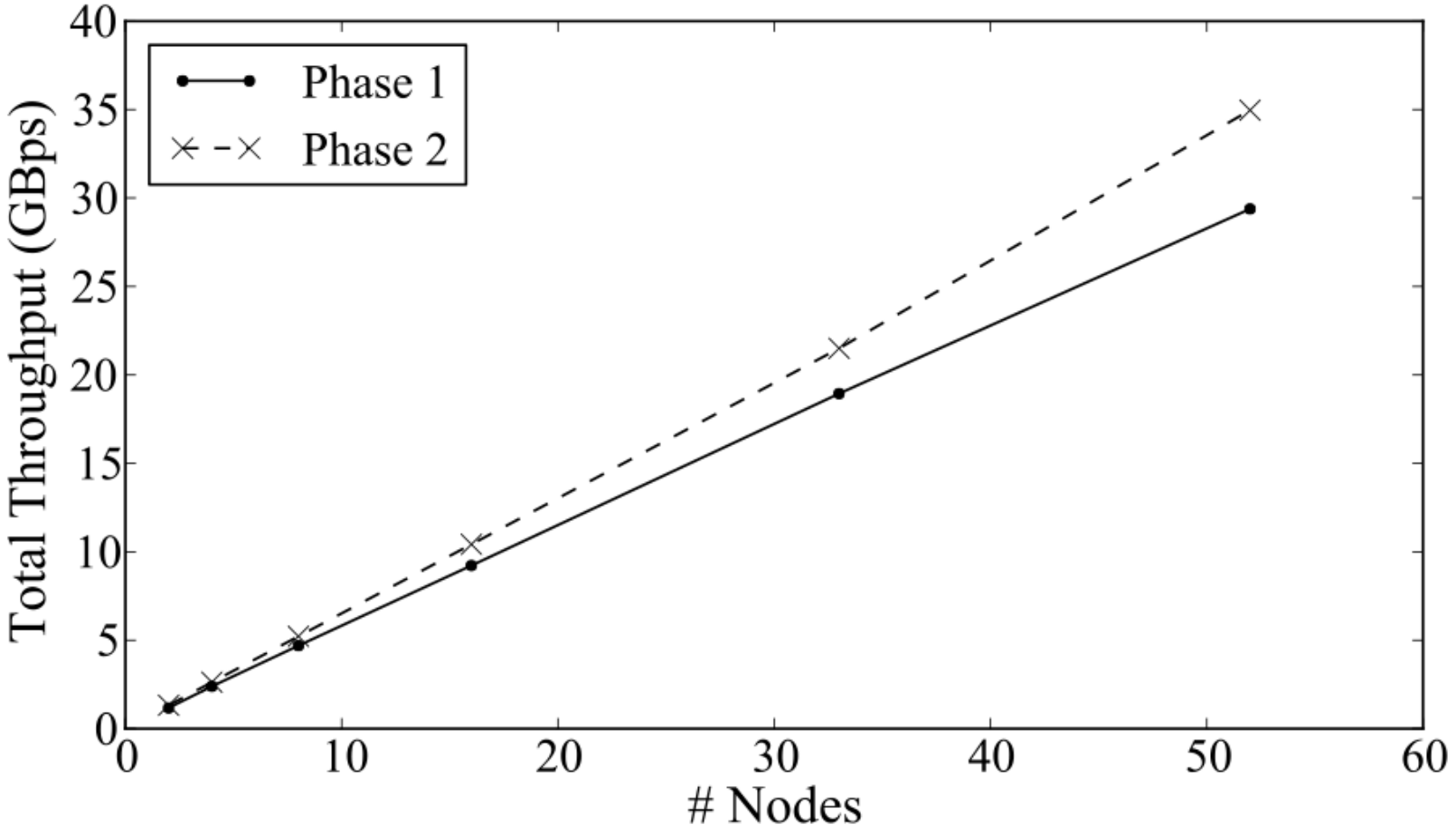
Disk throughput vs. write size



Evaluation

- DEMSort: 100TB on 159 nodes, 564 GB / min
- TritonSort: 100TB on 52 nodes, 916 GB / min
- 159 nodes -> 52 nodes, how?
- TritonSort is 6x more efficient
- Rate per server:
  - DEMSort: 3.5 GB /min
  - TritonSort: 17 GB / min

Scalability



Imbalanced storage

Intermediate Disk Speed (RPM)	Logical Disks Per Physical Disk	Phase 1 Throughput (MBps)	Phase 1 Bottleneck Stage	Average Write Size (MB)
7200	315	69.81	Writer	12.6
7200	158	77.89	Writer	14.0
15000	158	79.73	LogicalDiskDistributor	5.02

- Doubling disk speed has almost no effect
- LogicalDiskDistributor (CPU) becomes new bottleneck

Imbalanced memory

RAM Per Node (GB)	Phase 1 Throughput (MBps)	Average Write Size (MB)
24	73.53	12.43
48	76.45	19.21

- Doubling memory has almost no effect
- Diminishing returns in performance from larger buffers (seeks mostly amortized already)

Q: How much faster could TritonSort be?

i.e., how much faster if it were more balanced and/or better optimized?

Open challenges

- More general-purpose sorting
  - Real keys and values can vary in size
  - Need a scheme to partition key-space evenly by size
- Automatic performance tuning
  - Size of each buffer, number of buffers of each type, number of workers per stage, all picked manually
  - Can we tune these parameters automatically
  - Online algorithm needed to account for CPU interference
- Incorporating SSDs
  - Could we buffer less? Would we be as worried about seeks?

Q: How would you design TritonSort differently in 2020?

Conclusion

- Balanced systems use resources efficiently
  - Enables less cost or more processing power
- Picking the right hardware configuration is hard
  - Challenge: balance shifts with workload and implementation
- Implementing software for balance is hard too
  - Challenge: Must design around peculiarities of hardware
  - Completely new design often needed after hardware changes
- Today's OSes are not designed for balance
  - Overheads make it hard to fully use resources