Scaling Parallel Algorithms to Massive Datasets using Multi-SSD Machines

Algorithm Engineering

Jinha Kim

Outline

- Overview of SSD and NVMeSSDs
- Multi-SSD Setting and Lines I/O
- Historical Background
- Distribution (Algorithmic Primitive)
- Random Shuffle Primitive
- Empirical Results
- Strengths and Weaknesses

Scaling Parallel Algorithms to Massive Datasets using Multi-SSD Machines

Haohong Li
University of Maryland
College Park, MD, USA
lih@umd.edu

Jamshed Khan
University of Maryland
College Park, MD, USA
jamshed@umd.edu

Laxman Dhulipala
University of Maryland
College Park, MD, USA
laxman@umd.edu

Abstract

It is now possible in principle to build relatively inexpensive multicore servers equipped with dozens of terabytes, to even petabytes of local NVMe SSD storage. This raises a natural question of what can be done with such machines, and how many parallel storage devices are required to quickly compute over data that is much larger than main memory? Can we design algorithms for such machines that achieve nearly in-memory performance while gracefully scaling to datasets that are much larger than main memory?

In this paper we describe our first results with designing parallel algorithms for an inexpensive multicore workstation equipped with a 30TB NVMe SSD array composed of 30 SSDs; the entire machine, including the processor and all storage, can be purchased for a total cost of about ten thousand dollars. We microbenchmark our multi-SSD machine, measure the performance of different I/O patterns, and describe scaling bottlenecks for multi-SSD machines and how to overcome them by leveraging the modern asynchronous I/O stack. We then describe scalable I/O primitives for the multi-SSD setting that can be used to concisely express several I/O-efficient parallel algorithms, including samplesort, random shuffle, and basic sequence primitives.

Our experimental results reveal that there can be very little overhead to well-designed multi-SSD algorithms relative to their in-memory counterparts. For example, our multi-SSD samplesort matches the performance of one of the fastest in-memory samplesorts for inputs that fit in memory, and gracefully scales to sorting one trillion 8-byte integers in about an hour while using under two hundred gigabytes of DRAM.

ACM Reference Format:
Haohong Li, Jamshed Khan, and Laxman Dhulipala. 2025. Scaling Parallel Algorithms to Massive Datasets using Multi-SSD Machines. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28–August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages.
https://doi.org/10.1145/3694906.3743308

1 Introduction

Leveraging multicore machines to rapidly solve very large problems in-memory is the de-facto approach today across a broad range of areas, including database systems [22, 37, 49], optimization [6, 28], graph processing [16, 17, 45], bioinformatics [4, 33], information retrieval [44], among many other important data-driven areas. However, in-memory approaches are fundamentally limited to solving problems that fit within the internal memory (DRAM) of the machine, and even very costly multicore servers today cannot equip more than about 20 of terabytes of DRAM.

Over the last decade, we have witnessed a dramatic improvement in NVMe SSD technology in terms of its cost ($/GB), capacity, and bandwidth. Today, modern NVMe devices can be purchased for as little as a dozen cents per GB, while offering read/write bandwidth of at least several GB/s *per-device*. Recent generations of high core-count chips and motherboards designed for these chips have also seen a rapid increase in the number of *PCIe lanes* supported (each PCIe lane offers between 2–4GB/s of bandwidth depending on the PCIe generation). The increase in PCIe lanes has made it possible in principle to equip a multicore machine with *dozens of NVMe SSDs* that provide an aggregate bandwidth of 100 GB/s or more. Unlike a terabyte of DRAM, a terabyte of NVMe SSD requires only 5% of the power, representing a significant cost and environmental savings, especially when viewed over the lifetime of a server.

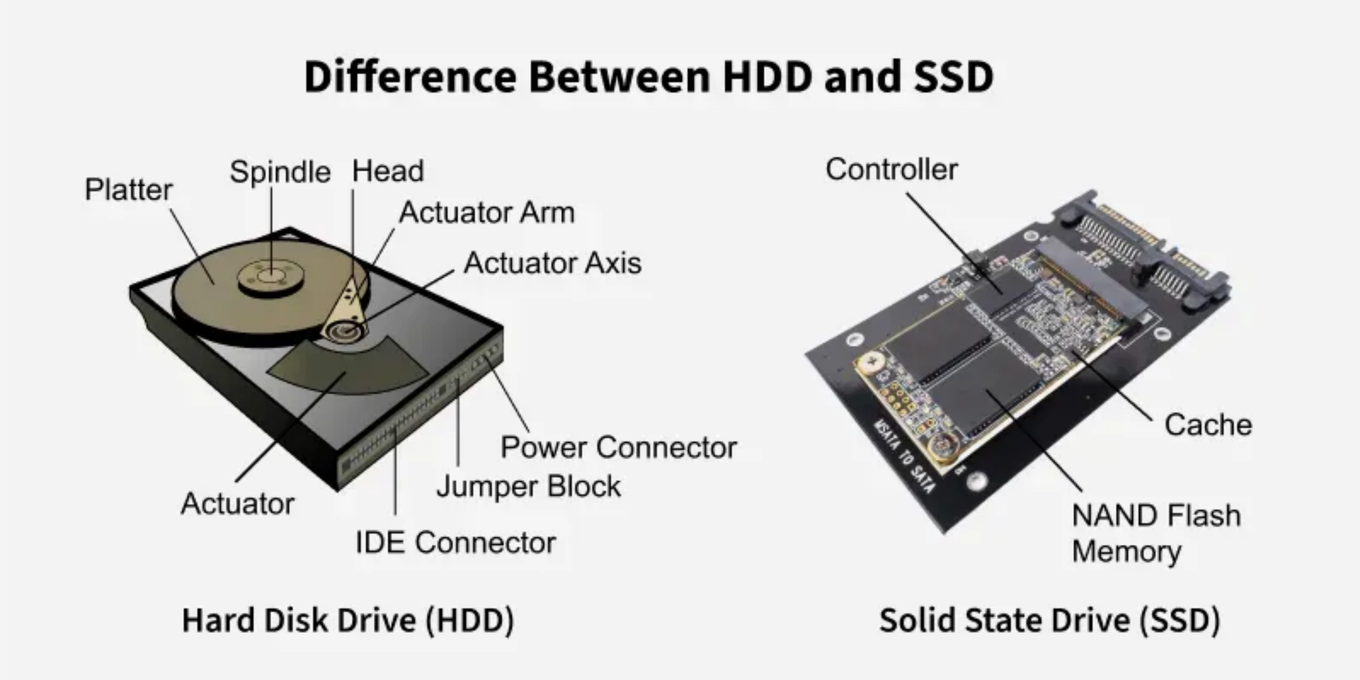
In this paper we are interested in studying the capabilities of multicore machines equipped with a large amount of NVMe SSDs; throughout the paper we refer to this type of machine as a ***multi-SSD*** machine. For example, what kind of problems can be solved efficiently on a multi-SSD machine, and can we engineer scalable algorithms that obtain nearly in-memory performance despite being run on datasets much larger than main memory? Since only a few prior reports on multi-SSD machines and algorithms have been published in the literature, primarily focusing on database systems [25–27] using fewer SSDs, we describe our design of a multi-SSD machine in detail. Our benchmarks reveal that many of the same challenges faced by external-memory algorithms designed for the single-disk or the single-SSD settings also hold in the multi-SSD setting. For example, it is still important to design algorithms with good locality, as algorithms that are not explicitly optimized to utilize large SSD block reads (at least 4KiB, and ideally larger) are unlikely to make effective use of the multi-SSD bandwidth.

Some of the challenges are specific to the fact that we have multiple disks. For example, algorithms must take care to load balance the reads and writes across the different SSDs—in theory this problem admits an elegant solution using page striping [34, 42, 50]. However, page striping requires increasing the effective block size by a factor of D (the number of disks), which can be costly even for modest values of D. Furthermore, since the aggregate bandwidth from the SSDs is high, the algorithm must be carefully optimized to ensure that the work done by the processors in-memory does not become the bottleneck. In particular, in-memory computation should be work-efficient and have low-depth, excessive copying should be

SPAA '25, July 28–August 1, 2025, Portland, OR, USA

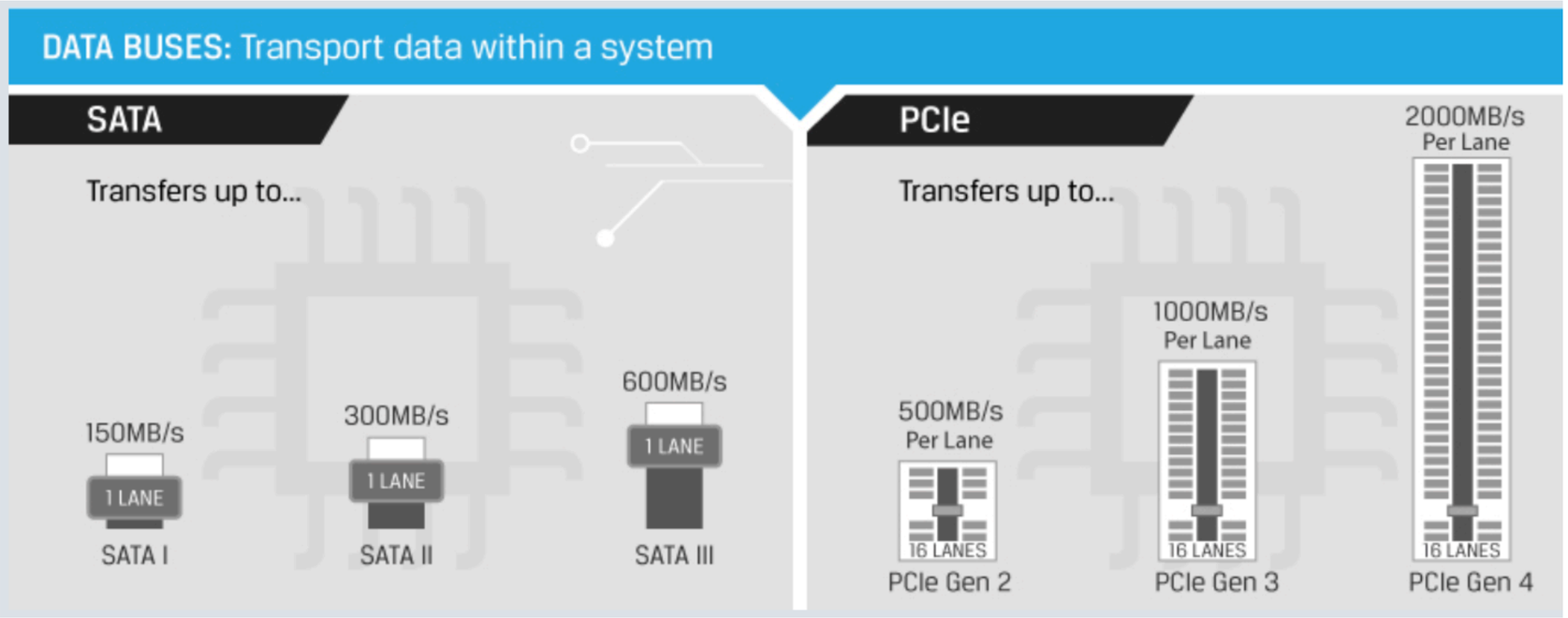
ACM ISBN 979-8-4007-1258-6/25/07.
https://doi.org/10.1145/3694906.3743308

SSD versus HDD



- SSDs don’t require mechanical movements unlike HDDs (spinning platters and heads) which adds mechanical seek/rotational latency
- SSDs are also less fragile than HDD and has lower latency and greater throughput

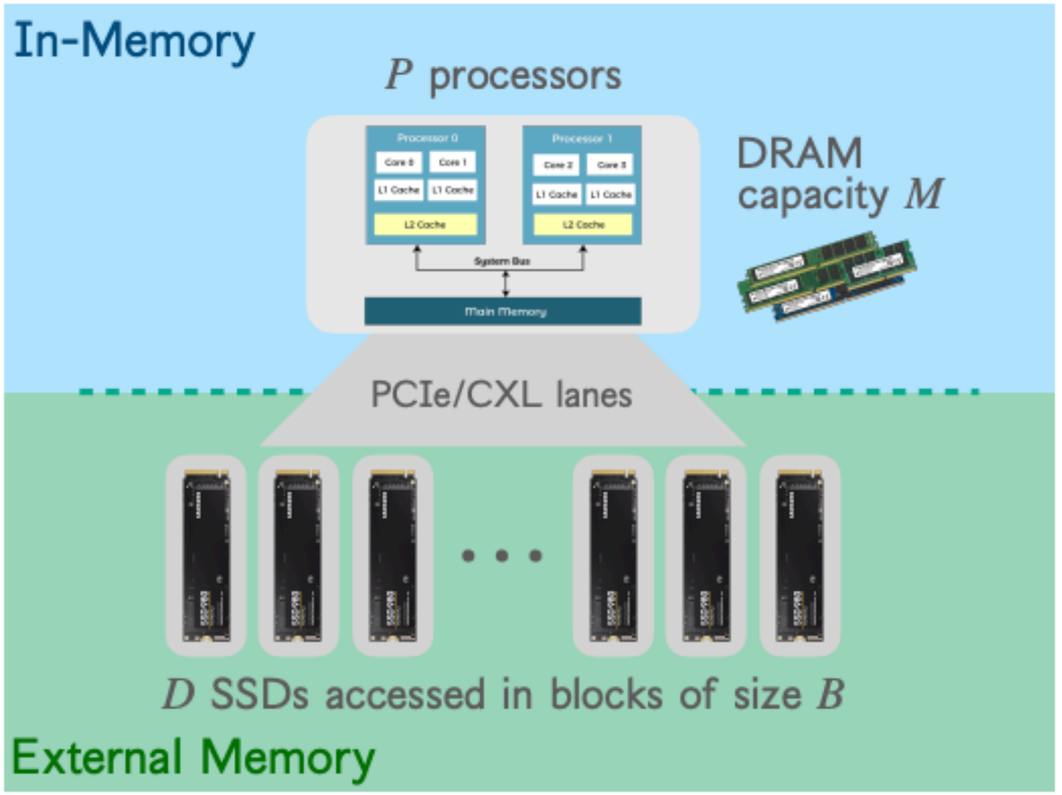
NVMe SSDs



- NVMe SSDs are used as secondary storage (or multi-SSD machine) and are SSDs that utilize PCI express (e.g. GPU use these for fast data transfer)
- NVMe SSDs transfers 25x more data than SATA equivalent

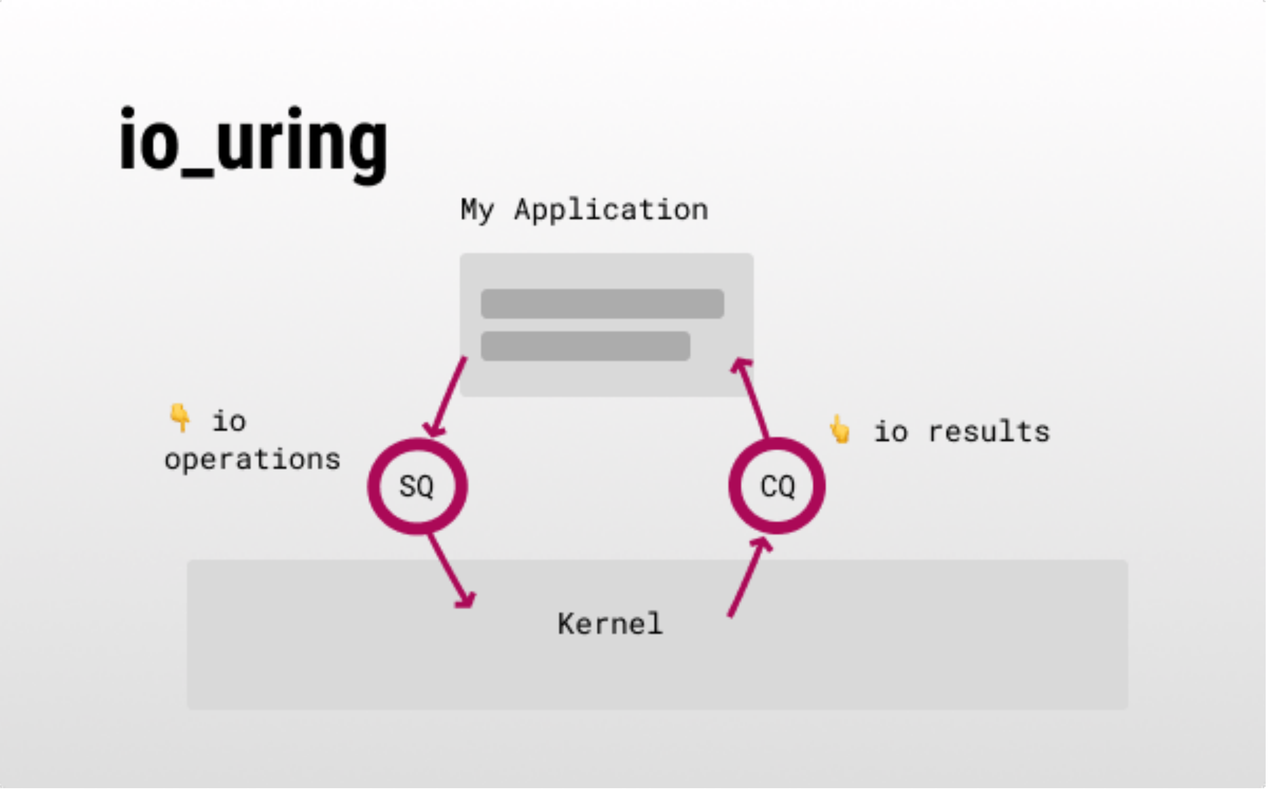
Multi-SSD Setting

- P processors with an internal size memory M
- External memory of size E divided across D SSDs (analyzed using I/O complexity)
- Assume SSDs are homogenous (same bandwidth and capacity)
- Each input is stored as a set of D files containing equal number of elements (one file stored at each SSD)



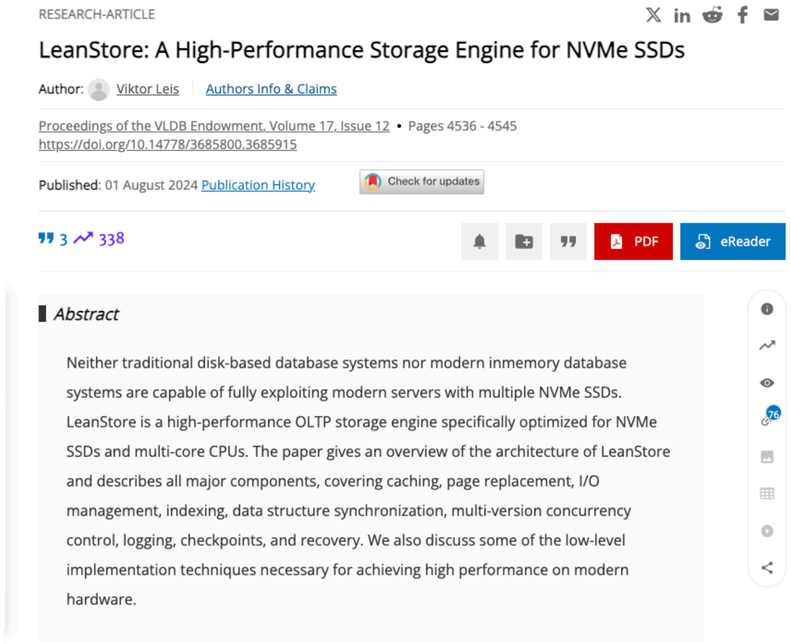
Linux I/O

- io_uring is a asynchronous I/O interface in Linux used for (storage/networking I/O)
- Has submission queue (user submit I/O requests) and a completion queue (can check status of requests) that both user and kernel share
- In comparison to libaio, can submit one system call to notify kernel per batch of I/O requests instead of for every request



Historical Background / Relevant Works

- Lots of existing work on theoretically I/O efficient algorithms
- External-memory software frameworks such as (LEDA-SM, TPIE, stxxl) but are not optimized for modern SSDs
- Promising work for out of memory database systems for multi-SSD setting (Lean Store)



RESEARCH-ARTICLE

LeanStore: A High-Performance Storage Engine for NVMe SSDs

Author: Viktor Leis | Authors Info & Claims

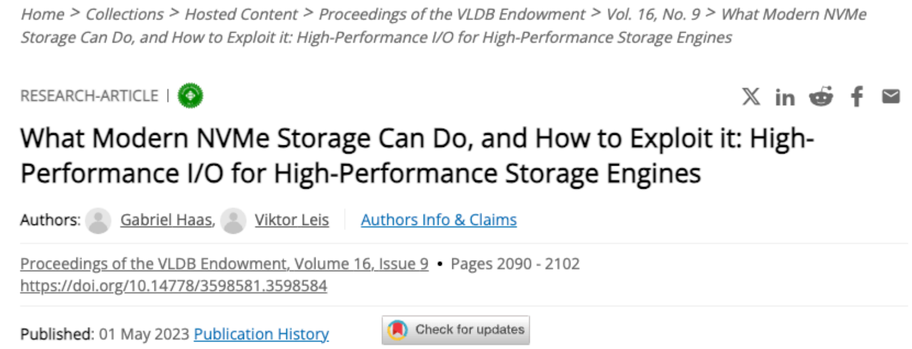
Proceedings of the VLDB Endowment, Volume 17, Issue 12 • Pages 4536 - 4545
https://doi.org/10.14778/3685800.3685915

Published: 01 August 2024 Publication History | Check for updates

3 338 | PDF | eReader

Abstract

Neither traditional disk-based database systems nor modern inmemory database systems are capable of fully exploiting modern servers with multiple NVMe SSDs. LeanStore is a high-performance OLTP storage engine specifically optimized for NVMe SSDs and multi-core CPUs. The paper gives an overview of the architecture of LeanStore and describes all major components, covering caching, page replacement, I/O management, indexing, data structure synchronization, multi-version concurrency control, logging, checkpoints, and recovery. We also discuss some of the low-level implementation techniques necessary for achieving high performance on modern hardware.



Home > Collections > Hosted Content > Proceedings of the VLDB Endowment > Vol. 16, No. 9 > What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines

RESEARCH-ARTICLE |

What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines

Authors: Gabriel Haas, Viktor Leis | Authors Info & Claims

Proceedings of the VLDB Endowment, Volume 16, Issue 9 • Pages 2090 - 2102
https://doi.org/10.14778/3598581.3598584

Published: 01 May 2023 Publication History | Check for updates

Showed database management systems like LeanStore could be made more optimal in exploiting NVMe SSDs (currently more throughput focused and less low latency focused)

Multi-SSD specs

CPU

- A 64-core (128-thread) AMD Ryzen Threadripper PRO 3995WX 2.70GHz CPU and 512 GBs of DDR4 RAM
- This chip supports 128 PCIe 4.0 lanes (most of them are available)
- PCIe 4.0 SSD requires 4 PCIe lanes, so attached a total of 124 SSDs (about 128)

SSD

- 31 of SK hynix Platinum P41 NVMe SSDs
- Each has 1TB capacity
- One SSD is dedicated to OS
- Each SSD peak read bandwidth of 7 GB/s and peak write bandwidth of 6 GB/s
- Due to DRAM bandwidth can load up to at most 146GB/s from SSD

Scalable Multi-SSD Algorithmic Primitives

- Primitives to abstract complicated and low level details of programming with many SSDs
- One notable and frequently used primitive is distribution (used for efficient memory sorting algorithms i.e. distribution sort or sample sort)
- Distribution primitive can be divided into *file_streamer, in-memory bucketer, and batched writer.*

File Streamer

- Issues batched reads from a dataset distributed across SSDs and writes completed reads to read-buffer queue (to be popped by worker threads that consume)
- Guarantee that each element is written exactly once to a read-buffer queue
- Create R reader threads and the files are split evenly across reader threads
- Reads issued by a thread is done in a round robin order over its assigned files
- From microbenchmarks, issue reads in 4 MiB chunks (balance of low CPU utilization and high utilization of bandwidth)

File Streamer

- Submit a batch of IO_uring requests
- Tries to send as many until tunable threshold and then block until at least one read is ready
- Thread's io_uring queue is filled
- Pushes the results to the read-buffer queue

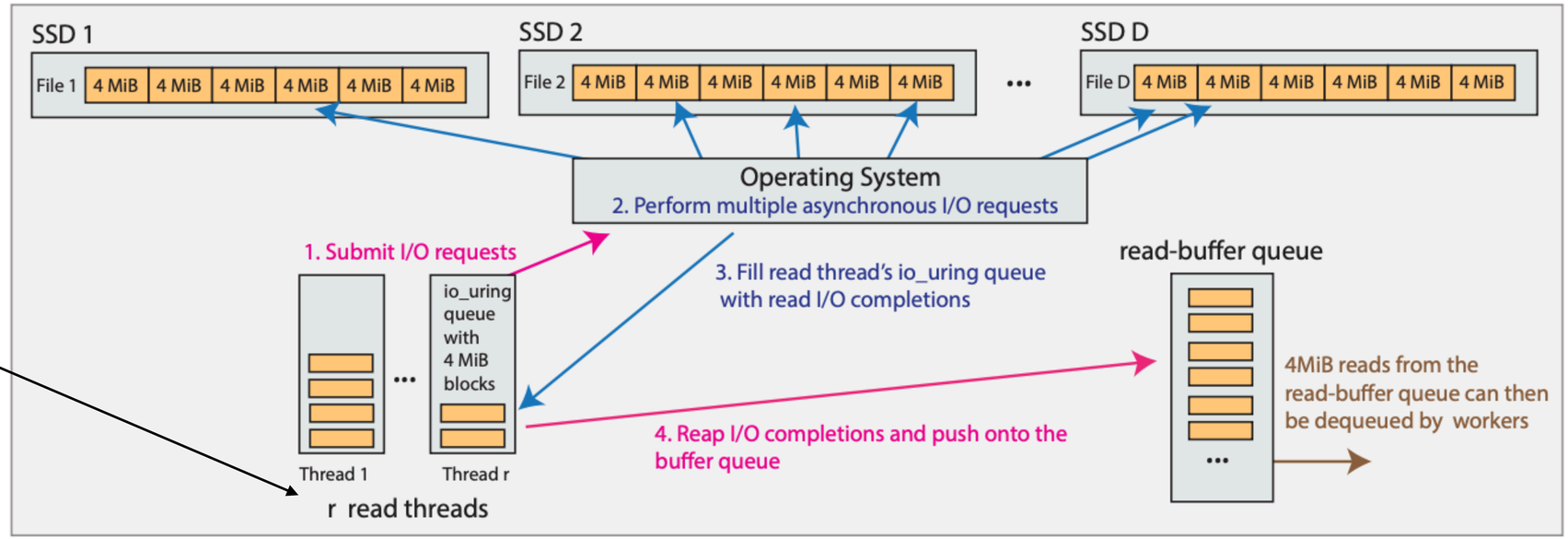
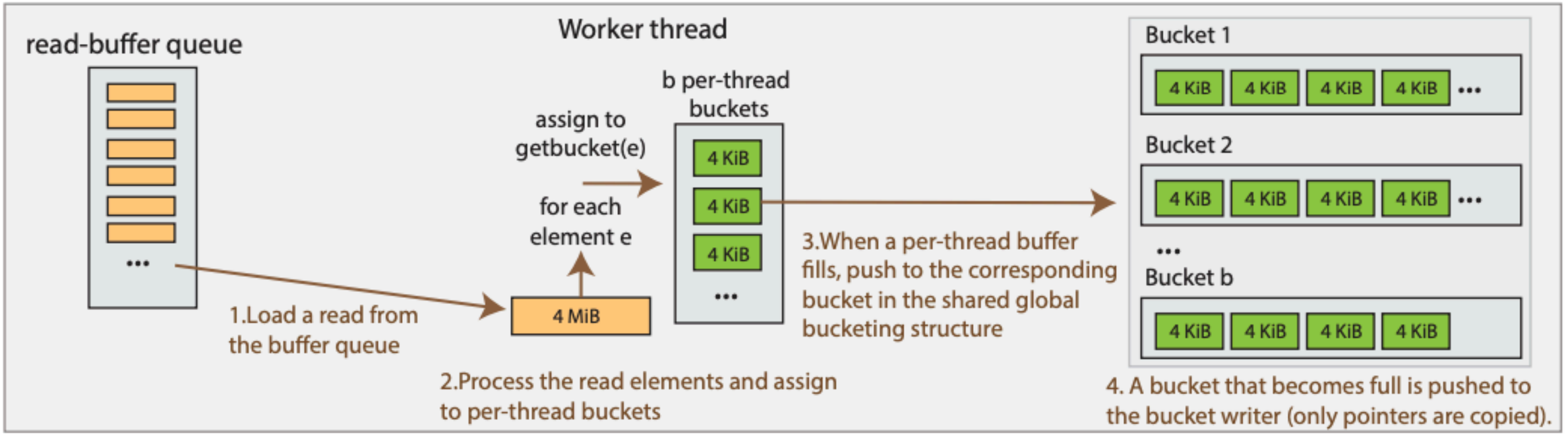


Figure 5: Diagram illustrating the work performed by the read threads in the file streamer.

In-Memory Bucketer

- Work threads perform distribution in-memory (fetch 4 MiB read buffer from shared queue)
- Each element in buffer is applied the user-defined getbucket function
- Don’t want to write each element to corresponding file of its bucket (too many small writes with synchronization costs) so want to buffer (but not too much to consume too memory)
- To address this, have per-thread buffers for each bucket where once one bucket is full it can be asynchronously written to disk

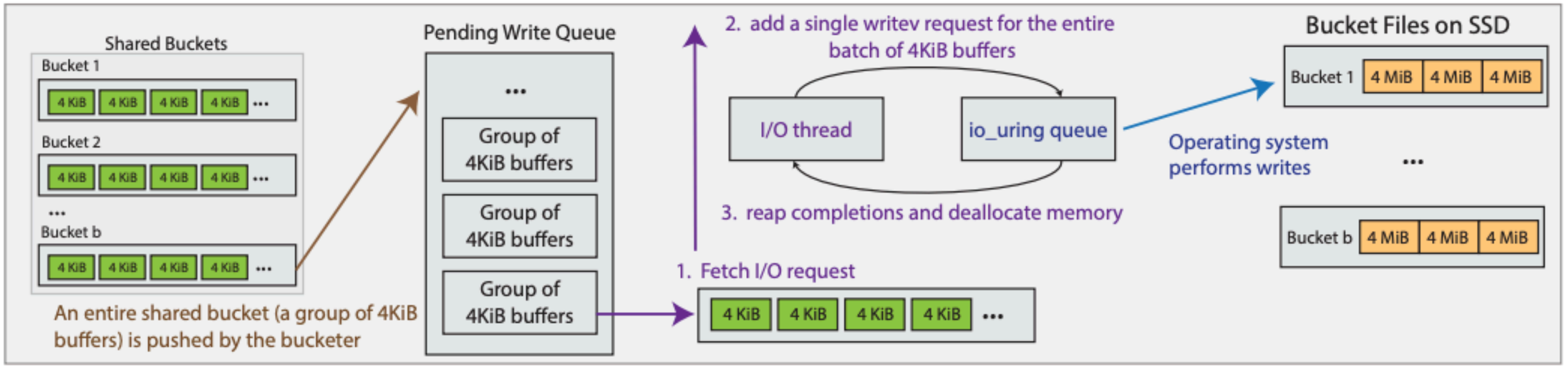
In-Memory Bucketer



- Each worker thread stores writes of an element to the per-thread buffer (4 KiB in size) for the corresponding bucket
- Shared bucket contains I/O request queues per bucket which store directly 4 KiB buffers (do not involving copying each element in the bucket)
- If a bucket request queue is full, submit I/O request queue to a shared write queue

Batched Writer

- Has a shared pending write request queue and set of writer threads that each has a private io_uring ring buffer. It then applies and writes to the bucket file on SSD
- It reaps for completion and deallocates memory



Distribution and Other Primitives

- Distribution primitive includes for an user-specified bucket function a *file_streamer (read files from SSD), in-memory bucketer (assign to buckets in-memory), and batched writer (write files to SSD).*

- Other provided primitives:

  - File mapper: given collection of files for each file, apply user-provided function on each file, write to disk

  - Random reader: when reading from SSDs, reads and writes must be aligned to SSD requirements (usually 512 bytes), designed to be more efficient for smaller reads

Multi-SSD Random Shuffle Primitives

- Simple and I/O efficient implementation based on Sanders:
  - Fix a set of buckets
  - Distribution step to distribute elements to bucket randomly (up till in-memory shared bucket is created) Num of buckets is automatically tuned to fit in-memory
  - Randomly permute each bucket in memory (one thread per bucket)
- Other primitives include Multi-SSD sample sort, map, reduce

ELSEVIER

Information Processing Letters
Volume 67, Issue 6, 30 September 1998, Pages 305-309

Random permutations on distributed, external and hierarchical memory

P. Sanders

Show more

+ Add to Mendeley Share Cite

https://doi.org/10.1016/S0020-0190(98)00127-6 Get rights and content

Abstract

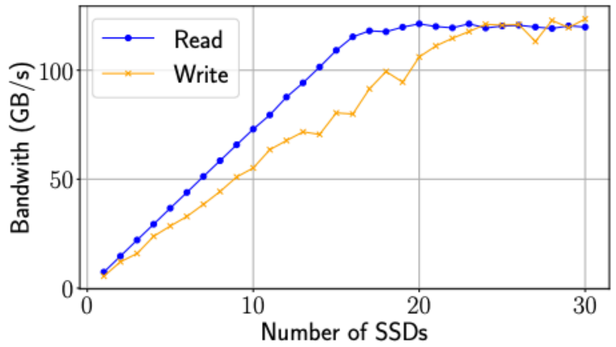
A simple randomized algorithm for generating a uniformly distributed random permutation of size n is investigated. It works in time $O(nP + T_{comm}(nP, P) + T_{prefix}(P))$ on P processors with high probability, where $T_{comm}(k, P)$ is the time for randomly sending or receiving k elements on each processor and $T_{prefix}(P)$ is the time for computing a prefix sum. The algorithm can be directly translated into an optimal external memory algorithm if fast memory for $\sqrt{nB}(1 + o(1)) + O(B)$ elements is available where B is the page size. Due to its simplicity, the same algorithm even outperforms the straightforward method on mainstream workstations if the cache is taken to be the fast memory and the main memory is treated like external memory. The algorithm is almost four times faster on a MIPS R10000 machine.

Empirical Results

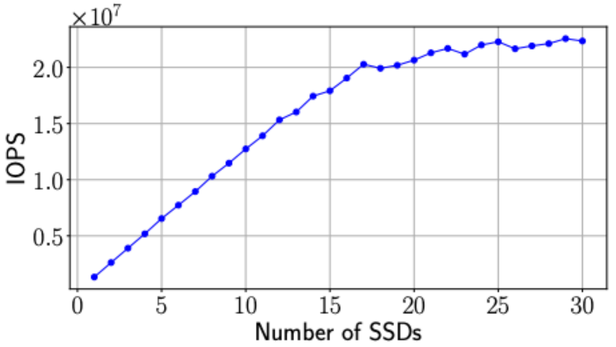
- **Implementation & Code:** Total codebase in C++ using io_uring and ParlayLib (development of efficient thread algorithms) is 3910 lines and sample sort is 142 lines
- **Input for sorting problem:** three standard distributions are used
  - uniform(μ)
  - Exponential distribution
  - Zipfian distribution

Empirical Results

Scaling and throughput, IOPS:

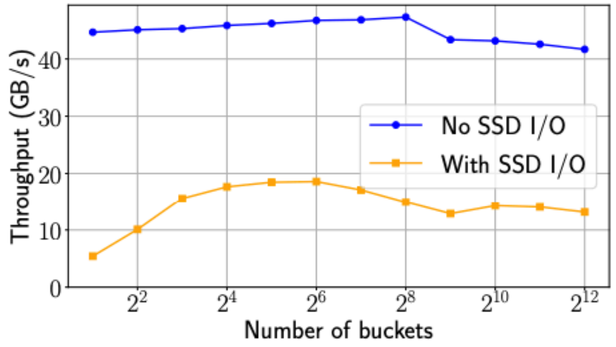


(a) Scaling results for the file streamer and file writer I/O primitives versus number of SSDs.

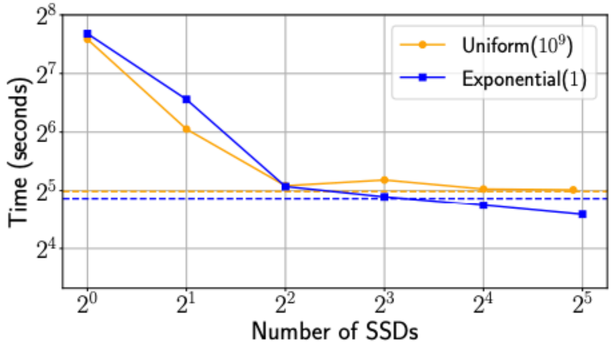


(b) IOPS (I/O operations per second) for the random reader used in the sampling stage of Samplesort.

Figure 8: Scaling results for the file streamer, file writer, and random reader.



(a) Throughput of distribution in GB/s with/without SSD I/O versus the number of buckets used in the distribution.



(b) Scaling of our multi-SSD sorting implementation on 128GiB inputs.

Empirical Results

Figure 9

		PLSS	Ours-1	Ours-2	Ours-4	Ours-8	Ours-16	Ours-30
Uniform	10^9	31.7	192	66.3	33.9	36.3	32.6	32.3
	10^7	29.7	207	78.3	32.3	33.6	29.9	30.3
	10^5	29.3	189	69.2	31.6	33.1	28.0	27.5
	10^3	22.6	194	56.7	31.8	32.1	28.3	28.2
	10	22.0	191	56.1	31.4	30.6	25.6	23.2
Exponential	1	29.2	169.8	57.1	29.9	31.9	28.7	28.3
	2	29.3	183.7	63.7	28.8	33.0	30.7	28.3
	5	26.1	182.6	66.8	28.7	32.8	28.6	28.4
	7	25.2	196.8	60.3	28.8	32.3	28.4	28.1
	10	24.5	184.5	66.1	31.0	32.0	31.2	28.1
Zipfian	0.6	33.1	200	64.9	35.6	37.1	33.7	33.3
	0.8	31.8	207	66.5	35.0	37.6	32.5	32.3
	1	27.4	205	78.3	33.8	36.4	32.5	32.2
	1.2	25.1	203	79.8	31.6	31.7	27.3	25.8
	1.5	26.7	204	60.7	30.6	30.3	26.4	24.2

- Comparing multi-SSD sample sort with optimized in-memory sample sort (PLSS) from Parlay lib
- Ours-X indicates using X SSDs. Fastest input running time for each is underlined
- There are n = 16 * 10^9 keys (128 GiB of data), parameters for each distribution on the left

Empirical Results

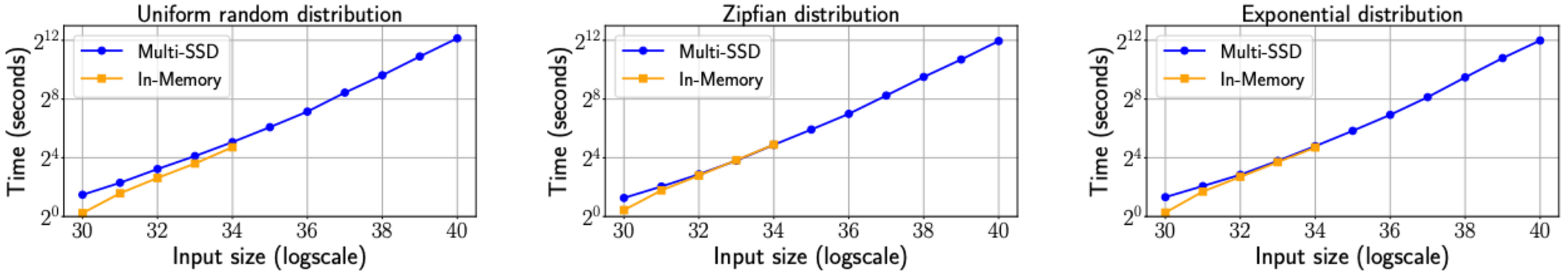


Figure 10: Plots showing running time in seconds vs. input size in log-scale for sorting different distributions (uniform random, Zipfian, and exponential from left to right). Our inputs have size between 2^{30} to 2^{40} keys each. Since our keys are 8-byte integers, the actual size of the input is up to $8 \times 2^{40} = 8$TiB.

- The in-memory plot stops once it reaches the largest input that can fit into main memory
- The Multi-SSD plot gracefully scales beyond

Empirical Results

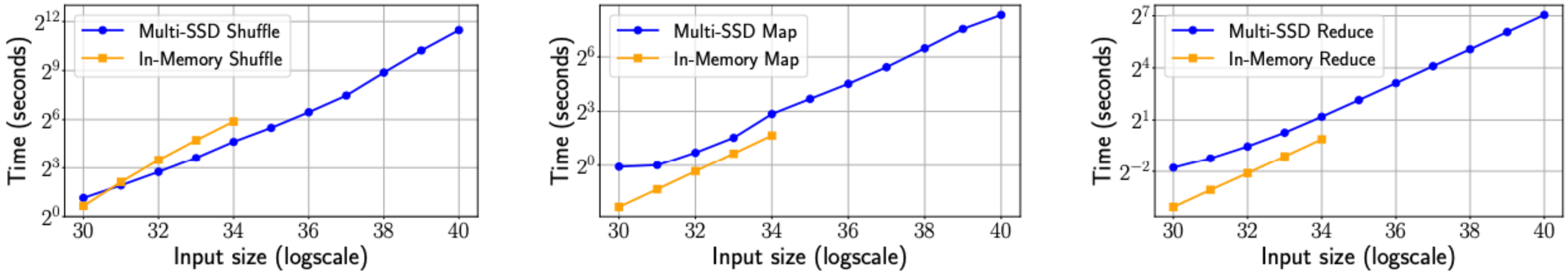


Figure 11: Plots showing running time in seconds vs. input size in log-scale for different sequence primitives. Our inputs have size between 2^{30} to 2^{40} keys each. Since our keys are 8-byte integers, the actual size of the input is up to $8 \times 2^{40} = 8$TiB.

- Can continue scaling fundamental parallel primitives on sequences to datasets that are significantly larger
- multi-SSD implementations can be as fast as existing in-memory implementations

Possible Directions, Strengths, Weaknesses

- Could we extend to multi-socket multi-SSD implementations where per socket there are multiple SSDs?
- Could this integrate with GPUs for running massive data that cannot fit in CPU / GPU RAM. GPUs can also PCIe to read data from SSDs?
- Strength in designing multi-SSD primitives and algorithm implementations such as sample sort that reaches nearly in-memory performance and scales empirically well with number of SSDs
- Provides a thorough and one of first experiments of parallel algorithms within multi-SSD setting
- Relies on direct I/O not buffered I/O which bypasses page caching (there might be settings in which buffered I/O might be necessary and hurts performance on bandwidth)