Scaling Parallel Algorithms to Massive Datasets using Multi-SSD Machines

H. Li, J. Khan, L. Dhulipala — SPAA 2025

Presented by: *Andrii Smutchak*

Problem and Background

Problem and setting

- In-memory parallel computing is the default for large problems, but is capped by DRAM ($\lesssim 20\,\text{TB}$, very expensive).
- NVMe SSDs: $\approx \$0.10/\text{GB}$ vs. $\approx \$1.50/\text{GB}$ DRAM; several GB/s per device.
- 128 PCIe lanes $\Rightarrow$ dozens of SSDs, $\geq 100\,\text{GB/s}$ aggregate.
- **Question:** can a multi-SSD machine reach near in-memory performance for $n \gg M$?

CPU	64 cores (TR PRO 3995WX)
DRAM	512 GB DDR4
SSDs	$30 \times 1\,\text{TB}$ PCIe 4.0
SSD:DRAM	60 : 1
Cost	$\approx \$10\text{k}$

Contributions

1. Microbenchmark study of a 30-SSD machine: what limits aggregate bandwidth.
2. Parallel I/O primitives on `io_uring`; the central one is **distribution**.
3. Sequence algorithms built on them: **samplesort**, **random shuffle**, **map**, **reduce**.
4. Evaluation vs. in-memory baselines (ParlayLib) up to 2^{40} keys (8 TiB).
5. Open-source library `PLAID`.

External-memory model and PDM

External-memory (EM) model [Aggarwal–Vitter '88]

- Internal memory M, block size B; cost = number of block transfers.
- Scan: $\Theta(n/B)$. Sort: $\Theta\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$.

Parallel Disk Model (PDM) [Vitter–Shriver '92]

- D disks; in each *synchronous* I/O round one block per disk.
- Disk striping turns D disks into one disk with block DB:
  fan-out drops from M/B to $M/(DB)$ — costly even for modest D.

Work–span: work W (total operations), span S (critical path).

Model used in the paper

- Shared memory: P processors, memory M, analysed by work/span.
- External memory: size E over D identical SSDs, block B.
- Transfers are **asynchronous** — no I/O rounds (differs from PDM).
- Input: D equal files, one per SSD; output distributed the same way.
- With NVMe, in-memory work and memory traffic can dominate, not only I/O count.

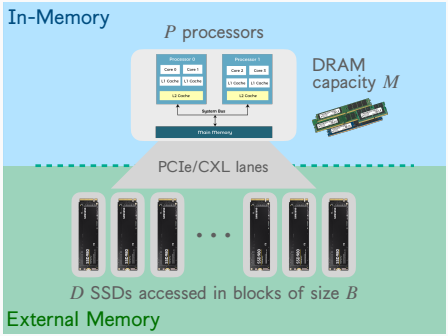


Fig. 1, Li et al.

Machine and Microbenchmarks

Hardware and microbenchmarks

- 4 PCIe root complexes, each capped at **42 GB/s**: assign SSDs round-robin ⇒ near-linear scaling to ≈ 135 GB/s.
- **Direct I/O** essential: buffered reaches 22.6 GB/s (4 jobs).
- **Large blocks** (4 MiB): peak bandwidth at lowest CPU cost.
- XFS within 11% of raw device.
- **DRAM bandwidth 146 GB/s** is the real ceiling.

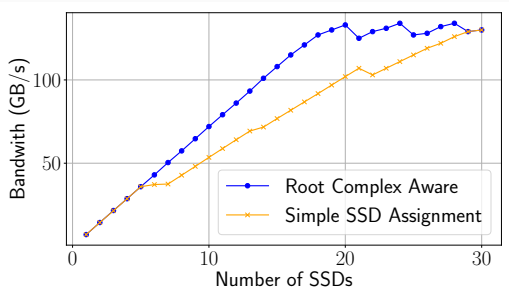
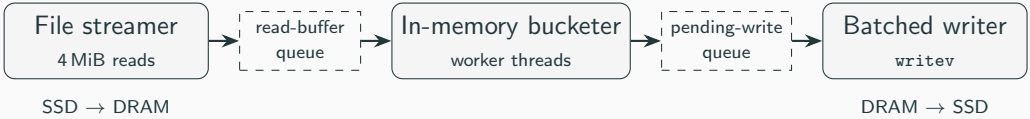


Fig. 3a: topology-aware vs. naive placement

The Distribution Primitive

Distribution: definition and pipeline

- Input: n elements and `getbucket`: $e \mapsto [r]$, $r \ll n$. Goal: group elements by bucket; each bucket becomes a file on some SSD.
- This is the MapReduce shuffle; distribution plus in-memory processing of each bucket expresses MapReduce/MPC algorithms.



Linux asynchronous I/O: `io_uring`

- Two rings shared between user and kernel: **submission queue** and **completion queue**.
- Many requests submitted with one system call; `libaio` needs one per request.
- Completions reaped without a syscall (`io_uring_peek_cqe`) or by blocking (`io_uring_wait_cqe`).
- **Direct I/O** (`O_DIRECT`): bypasses the page cache; buffers and offsets must be aligned (typically 512 B).
- `writev`/`readv`: one request covering many buffers.

Axboe '19; Didona et al., SYSTOR '22

File streamer

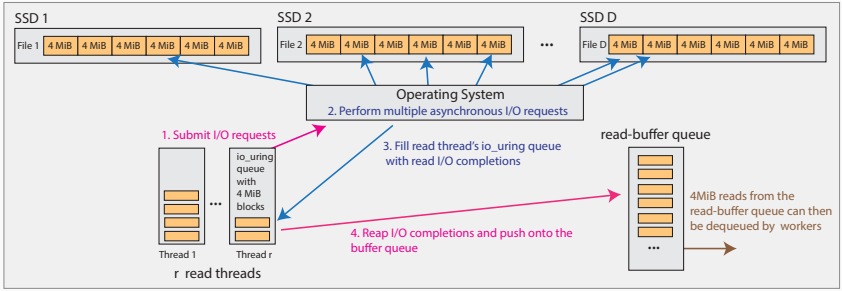


Fig. 5, Li et al.

- r reader threads; files split among them, round-robin within a thread.
- Submit up to `max_outstanding`; block for one completion, then drain all with `peek`.
- Under 512 B alignment, 2 reader threads: ≈ 60 GB/s; with mimalloc ≈ 100, jemalloc ≈ 20 GB/s.
- 10 reader threads: ≈ 120 GB/s at higher CPU cost; $r = 2$ used for CPU-bound runs.

In-memory bucketer

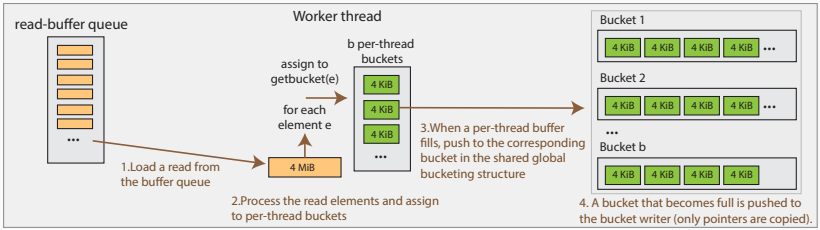


Fig. 6, Li et al.

- Naive: per-thread 4 MiB buffer per bucket: $128 \times 1024 \times 4\,\text{MiB} = \mathbf{512\,GiB}$.
- **Level 1:** thread-local 4 KiB aligned buffers per bucket.
- **Level 2:** full buffer moved *by pointer* to the bucket's mutex-protected queue.
- Queue above threshold $\Rightarrow$ batch pushed to the write queue. No copies.

Batched writer and other primitives

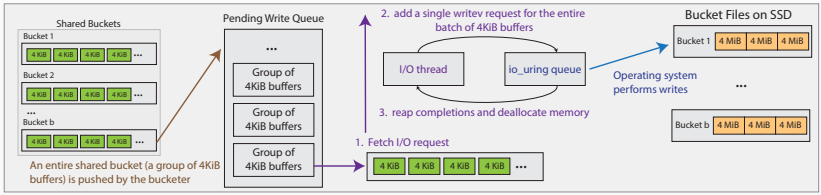


Fig. 7, Li et al.

- Each writer thread owns an `io_uring`; one `writev` covers up to $1024 \times 4\,\text{KiB}$ buffers.
- **File mapper:** load a file, apply a function (typically sequential); random start delay avoids lock-step I/O.
- **File writer:** writes at arbitrary offsets (out-of-order arrival). **Random reader:** small aligned reads.

Algorithms and Analysis

Multi-SSD samplesort

1. **Sampling.** Random reader (512 B/4 KiB reads); oversample, keep $b-1$ pivots.
   Paper: $O(b \log b/\epsilon)$ samples $\Rightarrow$ buckets $\leq (1+\epsilon)n/b$ [Axtmann et al. '15].
   This bound uses *overpartitioning*; plain samplesort needs $O(b \log b/\epsilon^2)$.
2. **Distribution** with `getbucket` = binary search over pivots.
3. **Per-bucket sort** via file mapper; sequential sort per bucket, P buckets in parallel.

Tuning b

- Bucket size $s = 128$ MiB so that DRAM $\geq cPs$: P buckets fit plus I/O buffering.
- Trade-off: more buckets $\Rightarrow$ cheaper sorting, costlier search; pivots should fit in L1.

Skew: key appearing f times among pivots $\rightarrow$ hash table H, gets $f-1$ buckets; its copies go to one of them uniformly at random.

Shuffle, map, reduce

- **Random shuffle** [Sanders '98]: distribute each element to a uniformly random bucket, then permute each bucket in memory. Same pipeline as samplesort; no binary search, no sort.
- **Map:** file streamer $\rightarrow$ apply f $\rightarrow$ file writer (offset-preserving).
- **Reduce:** each worker reduces streamed blocks into a local value; final reduce over P values.

	samplesort	shuffle	map	reduce
Source lines	142	< 50	43	30

Analysis: stated results

I/O count. $O(n/B)$ total; ≈ 4 block transfers per B elements (read input, write buckets, read buckets, write output).

Why one distribution pass suffices. Buffers for $b = n/s$ buckets must fit in memory; roughly $n \leq M^2/B$. With SSD:DRAM ratios of 15–60:1 this always holds, so the $\log_{M/B}$ factor of the EM sorting bound is $O(1)$.

Memory-bandwidth bound on distribution. Per element, at least 5 DRAM operations:

$$\underbrace{1}_{\text{SSD}\to\text{DRAM}} + \underbrace{1}_{\text{read}} + \underbrace{1}_{\text{read before write}} + \underbrace{1}_{\text{write-back}} + \underbrace{1}_{\text{DRAM}\to\text{SSD}} \quad \Rightarrow \quad \frac{146}{5} \approx \mathbf{29.2\,GB/s}.$$

Same accounting: map 4 vs. 3 operations in memory; reduce 2 vs. 1.

Results and Analysis

I/O primitives

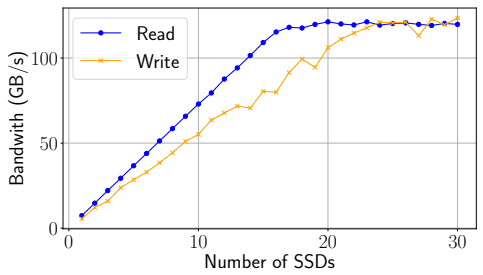


Fig. 8a: file streamer / file writer

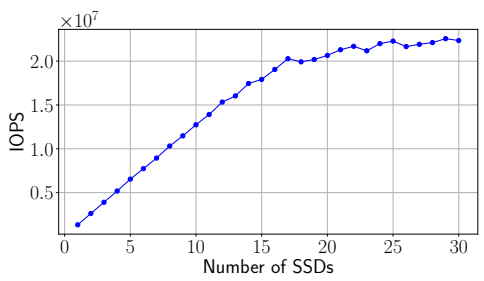


Fig. 8b: random reader IOPS

- Streamer tracks fio ⇒ no visible framework overhead on reads.
- Writer noisier (5–6 GB/s per SSD): sustained writes exhaust the SLC cache.
- ≈ 21M IOPS; sampling needs a few million reads ⇒ negligible.

Distribution vs. the memory-bandwidth bound

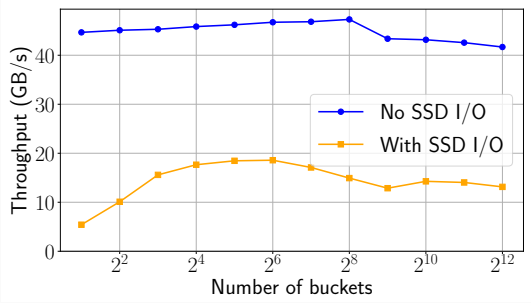


Fig. 9a, 128 GiB input, round-robin `getbucket`

- Without SSD I/O: $\approx$ 42–47 GB/s, roughly flat in b.
- With SSD I/O: peak $\approx$ 18.5 GB/s at 2^5–2^6 buckets; $\approx$ 5.5 at 2^1, $\approx$ 13 at 2^{12}.
- Bound predicts 29.2 GB/s $\Rightarrow$ only $\approx$ 64% reached; the gap is not explained in the paper.
- Possible causes (not tested): few buckets limit write parallelism; many buckets add partially filled buffers and pending state.

Sorting 128 GiB: number of SSDs

	PLSS	1	2	4	8	16	30
Unif 10^9	31.7	192	66.3	33.9	36.3	32.6	32.3
Unif 10^5	29.3	189	69.2	31.6	33.1	28.0	27.5
Unif 10	22.0	191	56.1	31.4	30.6	25.6	23.2
Exp 1	29.2	170	57.1	29.9	31.9	28.7	28.3
Exp 2	29.3	184	63.7	28.8	33.0	30.7	28.3
Zipf 0.6	33.1	200	64.9	35.6	37.1	33.7	33.3
Zipf 1.5	26.7	204	60.7	30.6	30.3	26.4	24.2

Subset of Table 1 (seconds); columns = number of SSDs.

$n = 16 \cdot 10^9$ keys (= 128 GB; caption says GiB), not Table 2's 2^{34} uniform keys.

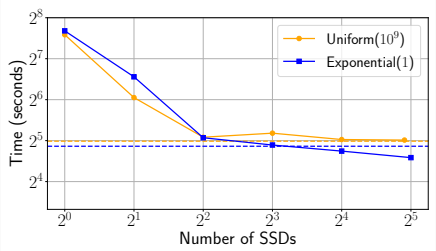
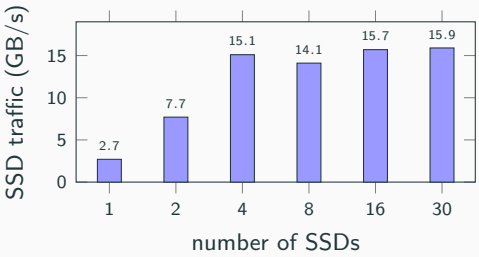


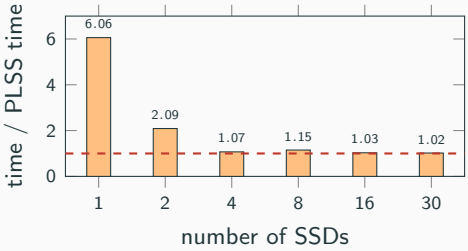
Fig. 9b; dashed: PLSS

- 1 SSD: 5.8–8.7× slower than PLSS. 4 SSDs: within 25% on most inputs, but 1.41–1.43× on Uniform 10^3 and 10.
- 30 SSDs slightly faster than PLSS on 4 of 15 inputs (Unif 10^5, Exp 1, Exp 2, Zipf 1.5).
- High-skew inputs keep improving past 4 SSDs: heavy keys skip binary search; heavy buckets need no sort.

Sorting 128 GiB: SSD-bound or compute-bound?



traffic = $4 \times 128\,\text{GB}$ / time (4 SSD passes, $n = 16 \cdot 10^9$)



Uniform 10^9, Table 1; dashed: in-memory PLSS

- **1–2 SSDs:** 6.1× / 2.1× PLSS time; I/O-limited. 1 SSD moves only $\approx 2.7\,\text{GB/s}$, well below one device's bandwidth (paper: contention, tuned for many SSDs).
- ≥ 4 **SSDs:** time within 1.02–1.15× PLSS, and going from 4 to 30 SSDs does not reduce it $\Rightarrow$ in-memory work dominates (paper: sorting is CPU-bound).
- For sorting at this size, 4 SSDs suffice (paper: ≈ 15 for peak bandwidth).

Sorting: scaling with input size

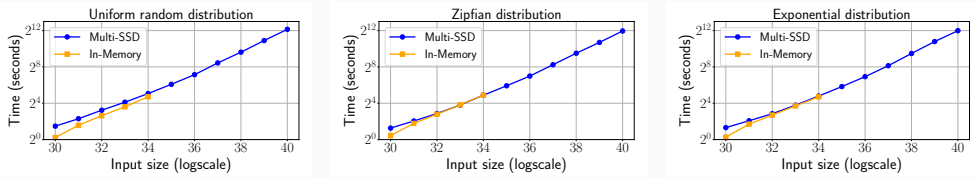


Fig. 10: time vs. $\log_2 n$, $n = 2^{30} \ldots 2^{40}$ 8-byte keys

- In-memory PLSS stops at 2^{34} (not in-place); multi-SSD reaches 2^{40} (8 TiB) in **4509 s** with fixed DRAM footprint.
- Gap to PLSS shrinks with n: 1.27× at 2^{34} (33.6 vs. 26.4 s, Table 2).[1]
- Space: three copies (input, buckets, output) ⇒ 24 TiB of SSD for an 8 TiB sort.

[1] Table 1 reports 31.7 s (PLSS) and 32.3 s (30 SSDs) for $16 \cdot 10^9$ Uniform(10^9) keys; the paper does not reconcile the two tables.

Portability: Google Cloud C3

	C3		Their machine	
$\log_2 n$	ext	in-mem	ext	in-mem
30	5.46	1.17	2.79	1.17
32	16.54	6.14	9.41	6.18
34	66.76	26.22	33.55	26.35
36	270.33	—	141.6	—
38	1018.18	—	787.22	—
39	2036.15	—	1911.81	—
40	—	—	4509.12	—

Subset of Table 2 (seconds)

- C3: 176 threads, 32 local SSDs, only $\approx 13.7\,\text{GB/s}$ read.
- In-memory slightly faster on C3 (more cores); external slower (less I/O).
- Same code: I/O-bound on C3, compute-bound on their machine (paper), without retuning.
- Gap closes at scale: at 2^{39} C3 is only $1.07\times$ slower despite $\approx 10\times$ less I/O bandwidth.

Other primitives

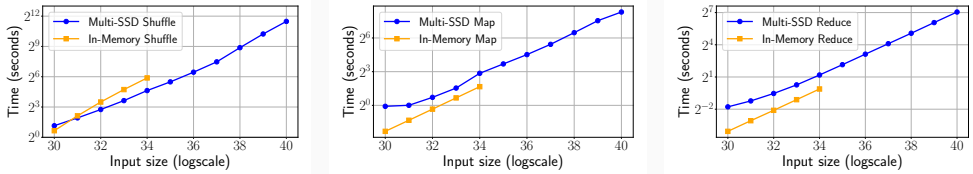


Fig. 11: shuffle, map, reduce; time vs. $\log_2 n$

- Shuffle at 2^{40}: $\approx 2800\,\text{s}$ vs. $4509\,\text{s}$ sort ($1.6\times$). Faster than ParlayLib's in-memory shuffle from 2^{31} on; paper: ParlayLib not tuned for large inputs.
- Map: 1.8–4.7×, reduce: 2.5–4.9× slower than in-memory; largest at 2^{30}. Paper claims "within 3×".
- Operation counting predicts 4/3 (map) and 2 (reduce): both gaps are larger.

Summary: what the paper concludes

Workload / setting	Paper's conclusion
Sorting (CPU-bound)	Negligible slowdown vs. in-memory; scales to 8 TiB with $<$ 200 GB DRAM
Map, reduce (memory-bound)	Within 2–3$\times$ of in-memory: extra DRAM operations per element
Distribution	Memory-bandwidth bound: $\geq$ 5 DRAM ops/element, $\leq$ 29.2 GB/s
Random shuffle	1.6$\times$ faster than sort; faster than in-memory ParlayLib
Few SSDs (1–2)	I/O contention; configuration tuned for many SSDs
Number of SSDs	$\approx$ 15 SSDs reach peak bandwidth; more add capacity and IOPS
Low-bandwidth cloud (C3)	I/O-bound, near-linear scaling; same code works

- Expected to carry over to SSD:DRAM ratios of 15:1–60:1 with similar core counts.

Discussion

Strengths

- Practical, reproducible engineering: documented build, open code, $\approx$ \$10k.
- Useful hardware lessons: PCIe topology, allocator, alignment, `fstrim`.
- Clean, reusable abstraction: distribution + per-bucket processing covers MapReduce-style algorithms in 30–142 lines each.
- Memory-operation counting explains *why* results look as they do.
- Demonstrates trillion-key sorting on a single workstation with under 200 GB of DRAM.

Weaknesses

- **Theory:** model informal; I/O analysis loose (e.g. claims $b = O(n/B^2)$, yet $s = 128\,\text{MiB} \neq B^2$ for $B = 4\,\text{MiB}$).
- **Baselines:** only in-memory ParlayLib; no STXXL or GraySort-style external sort.
- **Insufficient explanations:** distribution reaches $\approx 64\%$ of its bound; *sorting* time grows 2.27–$2.45\times$ per doubling beyond 2^{36} keys — paper attributes this to weaker cores, but $n \log n$ work predicts only $\approx 2.06\times$; map and reduce exceed operation-count predictions.
- **Overstated or inconsistent results:** "4 SSDs within 25%" (up to 43%, Table 1); "map/reduce within $3\times$" (up to $\approx 4.9\times$ at 2^{30}, Fig. 11); PLSS at $\approx 128\,\text{GB}$ is 31.7 s in Table 1 but 26.35 s in Table 2.
- **Narrow evaluation:** 8-byte keys only, no key–value records; one custom single-socket DDR4 machine plus one cloud VM (sorting only); tuning fixed for 30 SSDs.
- **Space:** $3\times$ input size on SSD.

Questions for discussion

1. Should cost models for this setting count memory traffic instead of I/Os? What would such a model look like?
2. Sorting time grows $\approx 2.4\times$ per doubling beyond 2^{36} keys. Pivot search leaving L1, write amplification on fuller drives, or something else? How would you test it?
3. Is kernel bypass (SPDK) worth it if the system is memory-bound anyway?
4. Beyond sequence primitives, which problems (graph algorithms, joins, clustering) can be cast as "distribute, then process each bucket in memory", and which need a different primitive?
5. Sorting writes every byte twice and needs $3\times$ the input in SSD space. Should SSD write volume be an explicit cost in algorithm design, and what would that change?