TorchFS: A Distributed Filesystem for ML Workloads

Daniel Pereira
University of Minho
pg55928@alunos.uminho.pt

Duarte Ribeiro
University of Minho
pg55938@alunos.uminho.pt

Filipe Pereira
University of Minho
pg55941@alunos.uminho.pt

Abstract

As deep learning and machine learning advance, the need for efficient data storage and retrieval has become crucial due to the exponential growth of training datasets. We introduce TorchFS, a distributed file system tailored for storing and retrieving data used in training deep learning models. Designed for integration with the PyTorch framework, TorchFS prioritizes performance and resilience to meet the demands of modern AI workflows.

Keywords: PyTorch, Distributed Storage, File System Optimization, Deep Learning

1. Introduction

With the rapid evolution of machine learning (ML) and more specifically deep learning (DL), the demand for a filesystem that can handle the intense data ingestion requirements of ML training pipelines has become paramount. As the volume of training data continues to grow exponentially, efficient data storage and retrieval are crucial. Traditional distributed file-systems like Ceph, HDFS, and Lustre [1], [2], [3] offer reliable throughput and durability but are not optimized for the predominantly sequential, read-heavy access patterns typical of ML data workflows.

In this paper, we introduce TorchFS, a distributed file-system that addresses this gap by providing a file-system designed specifically for ML workloads. TorchFS plugs directly into the PyTorch ecosystem, exposing a familiar POSIX-style interface while retooling the backend for training-centric performance and resilience.

Our design centers on three key ideas:

1. **Epoch aware caching & pre-fetching** — utilizes training-step hints to keep frequently accessed training data on the host device, reducing latency and minimizing data transfer overhead.
2. **Horizontal scalability** — object-striped data servers that add bandwidth and capacity linearly as you scale out.
3. **Metadata/data separation** — decoupling namespace operations from bulk I/O (inspired by GFS, GPFS and BeeGFS [4], [5], [6]), so lookups never contend with tensor streaming.

This design ensures that TorchFS can handle the unique demands of DL workloads, providing a robust and efficient solution for data storage and retrieval.

2. Background and Challenges

Traditional file systems are designed to handle a wide variety of workloads, including random access patterns, small file sizes, and a mix of read and write operations. However, ML workloads often involve large datasets that are accessed in a more predictable manner, with the bulk of the load coming from reading training files. This means we can tailor our file-system to better suit these workloads using optimizations.

2.1. PyTorch Access Patterns

To know how to optimize our file-system to suit PyTorch workloads, we first needed to understand how PyTorch accesses data. Therefore, we have to first profile the its data access patterns. To do so, we used a passthrough implementation of FUSE [7] that logged the accesses made to files during training. We evaluated the access patterns with

two different datasets: resnet and cosmoflow, with 1 and 4 GPUs each. Resnet is a composed of a few large files (80MB each), while cosmoflow is composed of many small files (64KB each). These four configurations let us evaluate the access patterns of PyTorch in different scenarios.

After evaluation, we found the PyTorch access pattern to be as follows:

1. Epoch Starts
2. The GPU opens a few files, and reads a few chunks[1] of data from each file immediately (presumably as a pre-fetch cache)
3. The GPUs then start reading from the first file sequentially, in a first-come-first-serve manner, until the end of the file is reached.
   3.1. When the file is a chunk long, each GPU uses a whole file by themselves
4. When a file is fully read, it is closed. A new one (that is after the first few that were opened at the start) is then opened and its first chunks read, and the next file in the order is fully read.
5. When all files are fully read, the epoch ends.
6. After each epoch, a small number of files are opened and read sequentially, presumably for validation purposes.
7. A new epoch starts and the process repeats.

Figure 1 plots the operation count for each dataset. We discovered that the access pattern varies across operations:

- *readdir* and *create* are used a fixed (and small) amount of times
- *open* and *close* are proportional to the number of files in the dataset (and the number of epochs)
- *write* is tied mostly to epoch count, but also to the number of files, as a large file count causes a lot of writes when training finishes
- *getattr* is tied both to the file size and file count, as it was called for every open, close and also every two reads
- *read* is tied to the file size (or count if the file is smaller than a chunk)

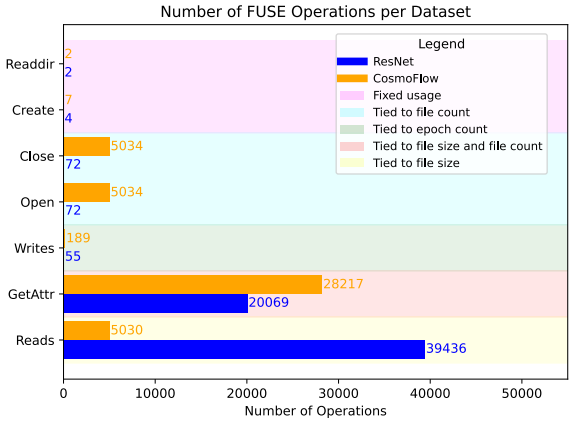

Figure 1: Metadata and I/O operations comparison. Each color indicates how each call is related to the training characteristics.

2.1.1. Insights

Our workload analysis revealed two key characteristics of PyTorch training on large datasets. First, file accesses follow a highly predictable, sequential pattern in which a small set of files is opened, read in order, and then the next file in the sequence is processed. Second, at the start of each epoch, the same initial subset of files is accessed repeatedly. These observations guided the design of an optimized file-system that exploits sequential access and temporal locality—enabling aggressive pre-fetching and caching strategies that significantly reduce I/O latency and improve GPU utilization without impacting generality.

3. The TorchFS design

TorchFS targets ML workloads with three design goals: scalability, performance, and availability. Its decoupled architecture separates metadata management from data storage, allowing each to scale independently. A client component presents a POSIX interface that is compatible with existing ML frameworks; a metadata cluster—composed of multiple metadata nodes (MDN)—maintains the file-system namespace and maps logical paths to physical locations; and a storage cluster—composed of storage nodes (SN)—handles reads, writes, and deletions without routing each request through the metadata layer. By isolating

[1]Chunks are about 130KB in size

namespace operations from data-path traffic, TorchFS avoids metadata bottlenecks and lets SNs scale horizontally with bandwidth and capacity demands.

To optimize performance for sequential, read-heavy training patterns, TorchFS employs aggressive caching and prefetching on the client side. Based on the predictable order in which training jobs consume dataset files, the system fetches upcoming files, ensuring data is locally available and masking network latency. This approach maintains high GPU utilization without imposing extra complexity on the ML pipeline.

High availability is achieved through two complementary mechanisms. The MDN cluster runs Raft to guarantee consistent, fault-tolerant namespace operations, surviving metadata-node failures without downtime. Data chunks are protected via erasure coding across SNs, delivering configurable fault tolerance while keeping storage overhead low. In combination, these strategies allow TorchFS to tolerate multiple simultaneous failures, preserve data integrity, and maintain uninterrupted access for distributed training.

4. Client Operations

TorchFS uses FUSE [7] to provide a POSIX interface, routing *open*/*read*/*write*/*close* calls to its metadata and storage layers. Each operation follows a sequence that ensures consistency, caching, and, if needed, erasure-code encoding/decoding.

When a client issues open on a file that isn't already open locally, TorchFS first contacts the metadata cluster to fetch two crucial pieces of information: (1) the file's attributes (size, permissions, timestamps, etc.), and (2) the location of each erasure-coded chunk. This metadata is then stored temporarily for the duration of the *open* on the client side. Once the chunk-location map is available, the client initiates parallel fetches of all fragments from their respective storage servers. As each fragment arrives, TorchFS performs erasure-code decoding in memory, reconstructs the

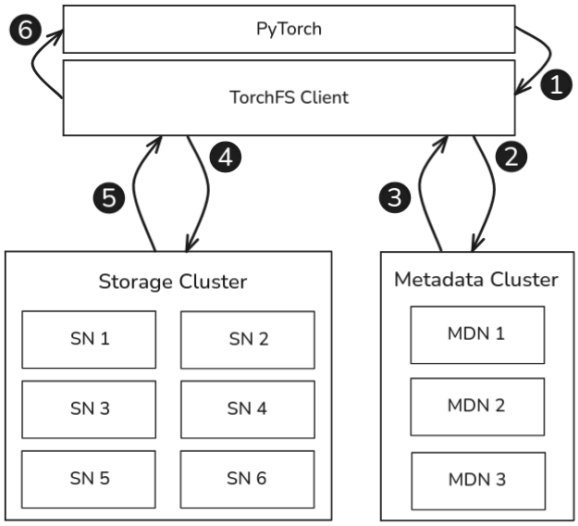

Figure 2: TorchFS Architecture Overview: the TorchFS client intercepts PyTorch file operations (1), routes metadata requests to the metadata cluster (2-3), and accesses data directly from storage nodes (4-5). Results are returned to PyTorch (6). Data access bypasses metadata nodes when metadata is known by the client to prevent contention.

complete file, and writes it to a local-disk cache. From this point forward, any subsequent read or write operations on that file are satisfied directly from the client's cached metadata or the locally stored file image, thereby minimizing remote round trips and reducing latency.

All user-level I/O operations proceed against the local copy: reads consult the in-memory metadata and the on-disk file image. When close is invoked, TorchFS checks whether any write-through activity has occurred during the file's open session. If so, it re-encodes the modified file into its constituent erasure coded chunks and immediately flushes each chunk back to its assigned SN - again in parallel to maximize aggregate throughput. Once every chunk has been successfully written, the local cache entry can be safely evicted or marked as clean, ensuring that subsequent open calls will retrieve the new data from the cluster.

TorchFS adopts an append-only update model inspired by AFS's whole-file writeback semantics [8]. In AFS, clients cache an entire file locally and buffer all modifications until *close*, at which point the file is atomically replaced on

the server. Similarly, in TorchFS, any modification during an open session triggers full-file re-encoding into erasure-coded fragments and their subsequent distribution across storage nodes upon *close*. Although this approach imposes additional work for small writes—requiring full-file encoding and network transfer—it fits our predominantly read-oriented workload. By deferring all writes until close and handling them as atomic replacements, TorchFS preserves simple consistency semantics while ensuring that occasional write bursts do not degrade overall system efficiency.

4.1. Erasure Coding

TorchFS leverages Reed–Solomon erasure coding to strike a balance between fault tolerance and storage efficiency for large ML datasets. Incoming data blocks are split into k data shards and encoded into m parity shards, yielding k + m fragments that are distributed across distinct storage nodes. On a write, TorchFS applies this encoding, dispersing each of the k + m shards so that any k shards suffice to reconstruct the original block.

Reads normally retrieve k data shards (minimizing CPU work). If any data shards are unavailable, TorchFS incrementally fetches parity shards until k total shards have been collected; fewer than k shards renders recovery impossible. By tolerating up to m simultaneous node failures and imposing a storage overhead of only (k + m)/k, far below naive replication, TorchFS optimizes for predominantly read-heavy workloads, accepting that encoding and decoding costs grow with m.

4.2. Caching and Prefetching

We implement a FIFO cache augmented by aggressive prefetching. On the first cache miss for a given file that was open for a read operation, the client immediately enqueues the next set of predicted files for prefetch, exploiting the workload's highly predictable, sequential access pattern. As incoming entries arrive, the oldest cache entries are evicted in FIFO order. This strategy guarantees that forthcoming data is loaded before it is requested, while imposing minimal overhead on cache management.

Prefetching is the key enabler of our system's scalability. In one experiment, a cache covering 25% of the dataset achieved a hit rate of approximately 86.9% when prefetching was enabled, compared to only 4.3 % without prefetching. By pulling data ahead of consumption, prefetching transforms a modestly sized cache into a highly effective buffer, dramatically reducing remote fetches and sustaining high GPU utilization.

5. Distributed Metadata

TorchFS implements a centralized metadata architecture in which each MDN maintains a complete replica of the entire file-system namespace. The metadata cluster relies on the Raft consensus algorithm [9] to enforce a total ordering of metadata updates and to guarantee availability in the presence of MDN failures. Client applications issue all metadata requests to the Raft leader, ensuring that concurrent operations on different replicas cannot introduce inconsistencies or race conditions.

On startup, every SN contacts the Raft leader to register its presence; likewise, whenever a new file is created, the corresponding SN notifies the leader MDN and registers the new file. Upon receiving a file-creation request, the leader MDN selects an optimal subset of SNs for storing that file—based on current load and network topology—to achieve balanced distribution of both data and I/O traffic.

6. Distributed Storage

The TorchFS storage layer distributes data across multiple storage nodes, each of which operates independently using the underlying file system's page cache and I/O scheduler. All I/O and deletion requests at a storage node invoke direct POSIX calls, while an RPC interface allows clients—after consulting the metadata layer for chunk locations—to perform reads and writes directly against the appropriate node. By relying on the OS's native caching and scheduling, TorchFS avoids custom

storage management logic, simplifying implementation and boosting performance.

Because storage nodes are stateless and register themselves with the metadata cluster, adding capacity requires only spinning up a new nodes and notifying the MDN. Once registered, a storage node instantly begins accepting chunk-level reads, writes, and deletions under the existing namespace. This simplicity minimizes operational overhead and enables seamless horizontal scaling to accommodate growing storage and bandwidth demands.

7. Implementation & Evaluation

TorchFS is implemented in approximately 4,300 lines of modern C++ and leverages several existing libraries to simplify development and maximize performance. We employ FUSE [7] to expose a POSIX-compatible file-system interface, enabling transparent integration with PyTorch. For RPC between clients, MDNs, and SNs, we use bRPC [10]; for consensus and replication within the metadata cluster, we rely on bRAFT [11] with RocksDB [12] as the underlying storage mechanism. All source code is publicly available in our GitHub repository [13].

In this section, we present a concise evaluation of TorchFS's performance and the impact of our optimizations on ML workloads. Our testbed consists of a single-node Ubuntu 22.04.5 LTS instance equipped with a 120 GB NVMe SSD and 4 GB of RAM. We benchmarked TorchFS using the DLIO suite [14] configured with a single NVIDIA H100 GPU. Throughout our experiments, we utilized a system configuration featuring a metadata cluster with three MDNs and a storage cluster with six SN's.

7.1. Caching and Prefetching Effectiveness

Figure 3 illustrates the dramatic impact of prefetching under a 50% cache-capacity constraint on a 4 GB dataset when training ResNet-50 [15]. In both runs, a client mounted TorchFS and executed a standard ResNet-50 training loop, reading data sequentially. In the first run (Without Prefetch), the cache achieved only 12 hits against 250 misses, yielding a negligible hit rate and incurring frequent remote fetches. By contrast, when prefetching was enabled, the cache delivered 275 hits with only 5 misses, corresponding to an effective hit rate exceeding 98%.

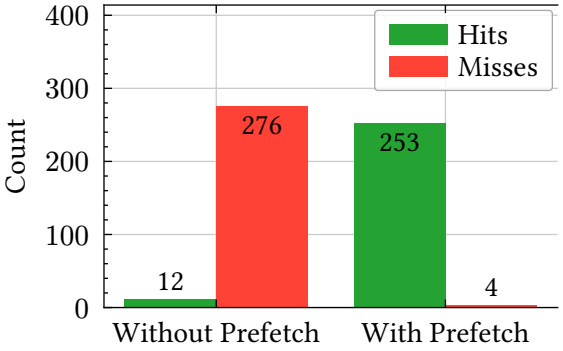

Figure 3: Cache Hits and Misses With and Without Prefetching - 50% Cache Capacity.

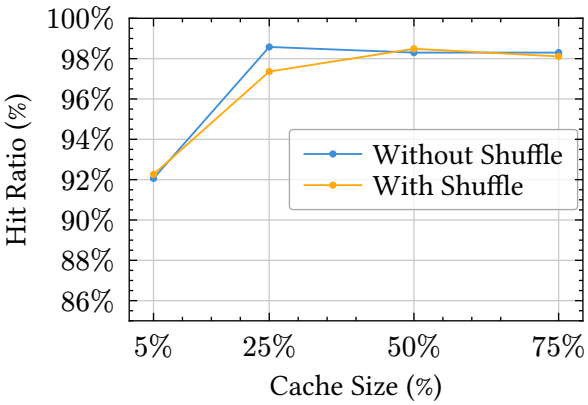

Figure 4: Cache Hit Ratio vs. Cache Size (With prefetching).

We further analyze cache-size sensitivity in Figure 4, comparing hit ratios with and without dataset shuffling under prefetching. For both scenarios, we evaluate four cache capacities (5%, 25%, 50%, and 75%). When the cache holds 5% of the dataset, hit ratios are already above 92%—slightly higher without shuffle (≈92.2%) than with shuffle (≈92.0%)—due to early prefetch of the sequential access stream. Increasing the cache to 25 % raises the hit ratio to nearly 99% without shuffle and to ≈97.7% with shuffle. Beyond 25% coverage, both curves plateau: at 50%, hit ratios reach ≈98.7% (no shuffle) and ≈98.5% (with shuffle), and at 75%, they remain around ≈98.5%. These results demonstrate

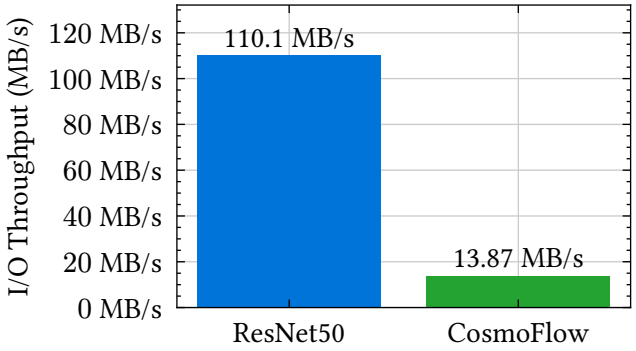

Figure 5: I/O Throughput: ReNet50 vs Cosmoflow for a dataset of 4GB

diminishing returns past 25%: a modest cache suffices to capture the hot working set even when shuffling introduces some randomness. In summary, TorchFS achieves near-optimal caching efficiency with only one-quarter of the dataset resident in local disk, regardless of whether the dataset is read sequentially or shuffled.

These results underscore two key observations. First, even with only half of the dataset resident in the cache, prefetching anticipates future access and preloads the next data blocks, which converts what would otherwise be cache misses into cache hits. Second, by reducing remote I/O to near zero, prefetching keeps GPU utilization at its maximum and minimizes end-to-end training latency. In short, the combination of a modestly sized FIFO cache and aggressive, workload-aware prefetching transforms what would be a cache-starved scenario into one with almost entirely local reads, thereby confirming the effectiveness of our approach under realistic ML training workloads.

7.2. Throughput Evaluation

In this section, we evaluate end-to-end I/O throughput for two 4 GB datasets—ResNet-50 [15] and CosmoFlow [16]—using TorchFS with client-side erasure coding.

Figure 5 compares sustained I/O bandwidth for these 4 GB datasets: ResNet-50 achieves 110 MB/s, whereas CosmoFlow reaches only 13.87 MB/s. ResNet-50's workload comprises a few thousand large image files, so each open/close incurs only a handful of erasure-code operations and yields a low per-byte overhead—most reads hit cache or are satisfied by bulk-fetched chunks, resulting in high throughput.

By contrast, CosmoFlow's 4 GB dataset consists of thousands of small grid-snapshot files. We suspect that the constant erasure-code decoding for so many individual files introduces significant overhead, limiting effective I/O bandwidth. As a result, per-file latency dominates end-to-end performance, driving overall throughput down to 13.87 MB/s—almost an order of magnitude below the ResNet-50 baseline. Consequently, GPU utilization drops from 99.5 % under ResNet-50 to 81.2 % for CosmoFlow, reflecting the inability to fully mask storage latency.

8. Related Work

The intersection of distributed file systems and machine learning workloads has driven significant research as deep learning applications demand increasingly specialized storage solutions. Classical distributed file systems like GFS [4] established the metadata-and-data separation architecture that influences modern systems, demonstrating the effectiveness of decoupling namespace operations from large-scale data transfers. Other traditional systems such as Lustre [3], GPFS [5], and Ceph [1] provide general-purpose, high-performance distributed storage but lack optimizations for ML-specific access patterns. More recent work has targeted those patterns directly: DIESEL [17] uses distributed caching to halve data access time for training; FanStore [18] distributes datasets to local storage under a global namespace via system-call interception; and 3FS [19], [20] leverages NVMe SSDs and RDMA hardware to accelerate AI workloads without relying on traditional caching.

9. Future Work

We plan to further reduce read latency by investigating low-level storage optimizations such as SPDK or RDMA for direct, zero-copy I/O between clients and storage nodes. Moreover, we will revisit our decision to perform erasure-coding and decoding on the client side,

exploring offloading these operations to storage nodes or batching strategies to reduce per-file overhead. In parallel, we will revise the metadata cluster design to improve request load balancing—potentially by sharding namespace partitions across multiple MDNs—so that hot directories and files do not concentrate traffic on a single leader.

As distributed ML training continues to proliferate, TorchFS must evolve to support multi-client, data-parallel workflows. In future work, we will extend the client and coordination layers to natively integrate with distributed training frameworks, ensuring that data locality and consistency guarantees hold across GPU clusters.

10. Conclusion

The growth of AI has driven training datasets to unprecedented scales, demanding distributed file-systems tailored to ML workloads. In this work, we presented TorchFS, which combines epoch-aware caching, aggressive prefetching, and metadata-data separation to align with PyTorch's predictable, sequential access patterns. Initial tests show that our prefetching strategy achieves over 98% cache hit rates while using just 25% of the dataset in memory, validating the effectiveness of a lightweight, POSIX-compatible approach without specialized hardware or complex protocols.

Developing TorchFS also highlighted important lessons in distributed storage design: workload characteristics, hardware constraints, and architectural trade-offs must all inform optimization. Although our preliminary evaluation is promising, comprehensive benchmarking across diverse configurations remains necessary to fully characterize TorchFS's benefits. Nevertheless, this project has been an invaluable learning experience, deepening our understanding of the challenges in optimizing storage systems for machine learning.

Bibliography

[1] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, Nov. 2006, pp. 307–320. [Online]. Available: https://ceph.io/assets/pdfs/weil-ceph-osdi06.pdf

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Lake Tahoe, NV, USA, May 2010, pp. 1–10. doi: 10.1109/MSST.2010.5496972.

[3] P. Schwan, "Lustre: Building a File System for 1,000-node Clusters," in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario, Canada, Jul. 2003, pp. 380–386. [Online]. Available: https://www.kernel.org/doc/ols/2003/ols2003-pages-380-386.pdf

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, USA, Oct. 2003, pp. 29–43. doi: 10.1145/945445.945450.

[5] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, USA, Jan. 2002, pp. 231–244. [Online]. Available: https://www.usenix.org/events/fast02/full_papers/schmuck/schmuck.pdf

[6] J. Heichler, "An Introduction to BeeGFS," Kaiserslautern, Germany, Nov. 2014. [Online]. Available: https://www.beegfs.de/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf

[7] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of User-Space File Systems," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 59–72.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, W. E. Howard, and D. S. Rosenthal, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, Mar. 1988, doi: 10.1145/38824.38836.

[9] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, Philadelphia, PA, USA, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf

[10] Apache Software Foundation, "bRPC: A High-Performance RPC Framework." 2024.

[11] Baidu, Inc., "Braft: Baidu's RAFT Implementation in C++." 2024.

[12] Facebook, "RocksDB: A Persistent Key-Value Store for Flash Storage." 2025.

[13] Daniel Pereira, Duarte Ribeiro, Filipe Pereira, "TorchFS." 2025.

[14] Argonne Leadership Computing Facility, "DLIO Benchmark: A Distributed I/O Benchmarking Suite." 2025.

[15] Microsoft, "microsoft/resnet-50." 2025.

[16] ExaLearn Project, NERSC, "CosmoFlow Dataset: N-body Cosmological Simulation Data for Machine Learning." 2019.

[17] L. Wang *et al.*, "Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020, pp. 1–11.

[18] Z. Zhang *et al.*, "FanStore: Enabling efficient and scalable I/O for distributed deep learning," *arXiv preprint arXiv:1809.10799*, 2018.

[19] W. An *et al.*, "Fire-Flyer AI-HPC: A Cost-Effective Software-Hardware Co-Design for Deep Learning," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1–23.

[20] "Fire flyer file system," 2025. [Online]. Available: https://github.com/deepseek-ai/3fs/