DYAD: Locality-aware Data Management for accelerating Deep Learning Training

Hariharan Devarajan[1], Ian Lumsden[2], Chen Wang[1], Konstantia Georgouli[1], Tom Scogland[1], Jae-Seung Yeom[1], and Michela Taufer[2]

hariharandev1@llnl.gov, ilumsden@vols.utk.edu, wang116@llnl.gov, georgouli1@llnl.gov, scogland1@llnl.gov yeom2@llnl.gov, mtaufer@utk.edu

[1]Lawrence Livermore National Laboratory

[2]University of Tennessee, Knoxville

***Abstract*—Deep Learning (DL) is increasingly applied across various fields to solve complex scientific challenges in modern high-performance computing (HPC) systems that are beyond the reach of traditional algorithms. Training DL models for scientific applications involves processing multi-terabyte datasets in each epoch. The data access behavior during DL training exposes optimization opportunities to cache these datasets in near-compute storage accelerators in HPC systems, enhancing I/O throughput. However, current middleware solutions employ near-compute storage accelerators primarily as exclusive caches, which limits the effectiveness of cache access locality. To address this problem, we introduce DYAD, a system designed to maximize sample locality in the cache, thereby significantly increasing I/O throughput in HPC systems.**

DYAD optimizes I/O for DL training based on three key features. First, DYAD boosts inter-node access speeds by using a DL-optimized streaming RPC with RDMA protocol, achieving a 1.25x performance gain over state-of-the-art solutions. This is possible because DL training datasets typically exhibit a normal distribution of sample and file sizes, allowing DYAD to pre-allocate buffers effectively for transfers. Second, DYAD further enhances inter-node access by coordinating data movement, which mitigates network congestion and increases throughput for inter-node accesses by up to 8.78x. Last, DYAD uses smart metadata caching that outperforms traditional global metadata access methods by several orders of magnitude in terms of lookup throughput for inter-epoch data sharing in DL training. We demonstrate how DYAD accelerates large-scale DL training on a high-end HPC cluster with 512 GPUs by up to 10.82x faster epochs compared to UnifyFS by performing locality-aware caching on near-compute storage accelerators.

***Index Terms*—caching, deep learning, I/O performance, middleware, HPC, sample sharing, RDMA-enabled.**

I. Introduction

Deep Learning (DL) training is a core part of many high-performance computing (HPC) workloads in scientific domains such as cosmology [1, 2], materials science [3, 4], and biology [5, 6]. DL training's efficiency depends on matching DL computation on the GPU with the I/O throughput achieved to read the dataset. DL training has three unique features that differentiate it from traditional scientific workloads [7]. First, DL training depends on a large number of dataset samples to achieve the generalization of DL models for solving new scientific problems [4]. Second, DL training iterates over this large dataset once per epoch to allow DL models to converge on common solutions for a given dataset [8, 9]. Finally, DL training shuffles this dataset between epochs to reduce dataset order bias while training DL models [10, 11]. DL training's significant dependence on data creates I/O-bound execution, where the HPC storage system cannot keep up with the demands of DL training on GPUs. Researchers have leveraged these features to increase DL training efficiency and reduce costs, introducing a range of technologies [12] aimed at optimizing data access.

Distributed caching is a popular mechanism to optimize data access by temporarily storing the DL dataset in near-compute storage accelerators within HPC systems. Some caching solutions [13, 14] use a fixed data placement and an exclusive cache policy to store the training dataset in near-compute storage accelerators. Other solutions [15–17] place the sample close to its first load and then utilize a global metadata service to locate the sample later. Additionally, NoPFS [12] uses fixed data placement and an inclusive cache policy to allow multiple copies of the same samples on different nodes within the compute cluster. These solutions reduce the training cost by increasing I/O throughput within these HPC DL workloads by using near-compute storage accelerators over traditional parallel file systems.

Existing caching solutions targeting the improvement of DL training on HPC systems suffer from four specific limitations. First, some solutions use dynamic buffer allocation to move data between nodes, which lowers the network's potential throughput due to increased allocation costs. TensorFlow Data Service [18] addresses this limitation with pre-allocated buffers, and Horovod [19] uses statically allocated buffers. Second, some solutions employ static and exclusive caching policies, leading to a high percentage of off-node data transfer calls, which diminishes the caching system's bandwidth. This is because DL training often requires fetching data from remote nodes. Solutions like NoPFS (Non-Exclusive Policy for File Systems) and adaptive caching mechanisms like Alluxio [20] attempt to address this issue. Third, some solutions lack coordination in data movement, resulting in multiple I/O calls to the same dataset. Since DL datassets might contain multiple samples per file, this increases network congestion and reduces data transfer performance. Solutions like the Unified Data Access Layer (UDAL) with

coordinated prefetching address this issue [21]. Finally, some solutions fail to cache sample discovery operations from other processes on the same node, leading to higher metadata costs during training. Solutions like metadata caching in Alluxio and the Distributed Metadata Service (DMS) attempt to mitigate this issue. These limitations highlight the need for locality-aware caching solutions that can efficiently cache both data and metadata to accelerate DL training beyond the existing solutions that are generally tailored to target one or two of these limitations, but not all.

Given these limitations, we present DYAD, a cohesive system designed to address all four limitations by leveraging locality-aware caching to accelerate large-scale DL workloads on HPC systems. DYAD uses preallocated buffers with a novel streaming Remote Procedural Call (RPC) and Remote Direct Memory Access (RDMA) protocol, similar to how TensorFlow Data Service and Horovod improve throughput with static allocation strategies. DYAD employs locality-aware caching to reduce off-node data transfers, akin to the approaches in NoPFS and Alluxio, which ensure data is available locally. Additionally, DYAD reduces redundant I/O calls through data movement coordination, similar to coordinated data prefetching and orchestration in solutions like UDAL, thereby reducing network congestion and improving data transfer performance. Furthermore, DYAD enhances metadata management through its hierarchical sample discovery mechanism with node-local metadata caching, addressing high metadata costs. This mechanism is comparable to Alluxio's metadata caching and the Distributed Metadata Service, which minimizes metadata-related overhead and enhances training efficiency. With these features, DYAD can accelerate DL training epochs for Unet3D [22]and Massively Parallel Multiscale Machine-Learned Modeling Infrastructure (MuMMI) [6, 23] up to 10.82 $\times$ compared to state-of-the-art solutions. The main contributions of this work are:

- We *introduce streaming RPC with RDMA protocol* for efficient inter-node data transfer for sharing data during DL training;
- We *establish locality-aware caching* to maximize DL training by maximizing node-local access and minimizing off-node data transfers;
- We *create data coordination mechanism* that reduces redundant data movement traffic and maximizes the throughput of DL training;
- We *develop hierarchical sample discovery mechanism* to lookup samples efficiently by using metadata caching; and
- We *accelerate large-scale DL workloads* on the Corona cluster with 512 GPUs for multi-terabyte dataset by up to two orders of magnitude better performance.

II. Background and Related Work

Background. We examine the unique behaviors of DL training pipelines and their potential optimizations, highlighting the synergy between understanding these behaviors and enhancing training efficiency through innovative software solutions.

The efficiency of DL model training depends on the performance of its input pipeline in reading datasets from the file system. This input pipeline exhibits the following five primary behaviors [7, 14, 24]. First, the input pipeline is executed by task-driven runtime on the file system. Second, the pipeline reads the entire dataset on each training epoch. Third, data-parallel training shards the dataset among multiple GPUs. Fourth, samples are randomly selected to increase the model generalization. Finally, the AI datasets contain one or multiple samples per file. These behaviors differentiate DL training from traditional HPC workloads such as simulations and high-performance data analytics.

Caching software can leverage these behaviors to optimize the input pipeline for DL training by providing the following five features. First, the software must efficiently provide highly concurrent access to the dataset. Second, the software needs to improve the cache locality of samples to progressively enhance the input pipeline across multiple epochs. Third, the software must avoid data staging costs by dynamically loading the dataset as accessed. Fourth, the software must provide high bandwidth access for random sample access. Finally, the software must consider the dataset's sample distribution to reduce metadata and I/O costs on the file system. Using these behaviors within system software is critical to maximizing I/O throughput for large-scale DL training.

Related Work. State-of-the-art solutions enhance the input pipeline in DL training using exclusive cache policies with static and dynamic sample placements. A technical comparison of these solutions reveals distinct approaches and inherent limitations, as outlined in Table I and the following discussion.

For static placement techniques, state-of-the-art caching solutions, such as NoPFS [12], TensorFlow Data Service [18], and LMDBIO [25], leverage predetermined sample ordering within DL training to determine the frequency of accesses. This information is used to preload samples into storage accelerators to maximize the impact of cache hits on DL training. However, this approach requires pre-staging samples into hierarchical storage. It also omits inter-node communication for sample sharing, consequently diminishing the input pipeline's I/O throughput during DL training. In contrast, DYAD operates independently of sample ordering knowledge and implements locality-aware caching of samples.

For dynamic placement techniques, state-of-the-art solutions dynamically load data from parallel file systems and cache the data into storage accelerators to enhance the I/O throughput of the input pipeline. This bandwidth improvement is largely due to the employment of faster storage devices and network links for sharing data across multiple processes. Examples of these solutions include HVAC [13], UnifyFS [15, 26], DeepIO [14], Quiver [27], Alluxio [20], FanStore [28], and DataSpaces [16, 17]. However, these solutions do not fully deploy the bandwidth potential of storage accelerators. They rely on an exclusive cache design, which increases inter-node communication significantly slower than node-local accesses. Moreover, current solutions lack coordinated access, resulting in possible redundant I/O operations for multiple samples located in the

Table I: Difference in features between state-of-the-art solutions and DYAD

Features	HVAC	UnifyFS	NoPFS	DataSpaces	DYAD
Remote Data Access	RPC	Mercury RPC with RDMA	MPI	Mercury RPC with RDMA	Streaming RPC with RDMA
Metadata Management	Uniform Hash	Service with Broadcast	Static Placement	Distributed Metadata	Hierarchical KVS
Call Aggregation	None	Yes (Local Service)	None	Yes (Local Service)	Hybrid (Local FS + KVS)
Inter-Epoch Read Caching	Single Copy	Single Copy	Static Placement	Single Copy	Locality-aware Caching
Optimization Goal	DL-training	Write-heavy	DL-training	Sample Sharing	Write-once-read-many
Node-Local Data Access	Service	Service	Local FS	Service	Local FS
Access Granularity	Sample	Sample	Sample	Sample	File (enables coordination)

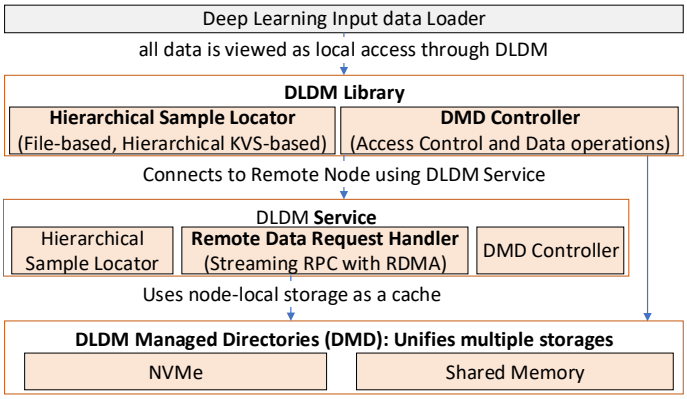


Figure 1: The DL training data loader uses DYAD as a cache transparently to consume samples on local and remote DMD.

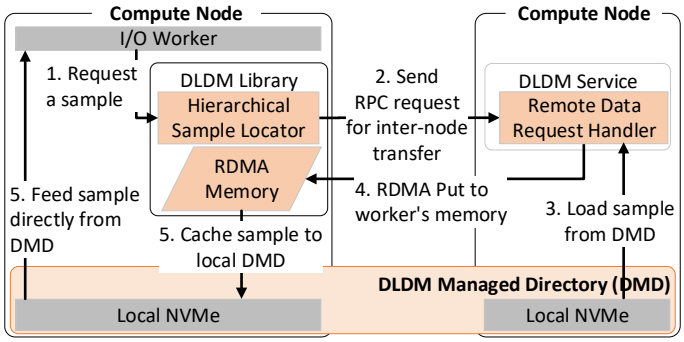


Figure 2: The I/O worker uses DYAD to perform sample discovery from either local or remote DMD seamlessly.

same file. DYAD addresses these issues by implementing locality-aware caching, which duplicates cache entries to boost the rate of node-local device accesses. Furthermore, DYAD enhances access efficiency to the same file by coordinating I/O operations to minimize network congestion within the cluster.

III. Locality-aware Data Management for DL

DYAD is a locality-aware cache designed to share data efficiently between I/O workers of distributed DL training. DYAD operates on write-once-read-many (WORM) data semantics to accelerate deep learning and High Performance Data Analytics (HPDA) workloads in large-scale HPC systems. DYAD has two core objectives toward the goal of a locality-aware cache for DL training. First, DYAD leverages node-local accelerators to enhance I/O bandwidth for deep learning training. This efficiency is achieved through effective network communication based on streaming RPC with RDMA protocol (Section III.B), and data movement strategies based on file-level granularity, file-lock-based synchronization, and passive coordination (Section III.C). Second, it performs sample discovery using multiple node-local resources to isolate metadata operations and improve the throughput of DL training (Section III.D).

A. DYAD Primary Components

At a high level, DYAD consists of three primary components: (a) the DYAD Managed Directory, (b) the DYAD Library, and (c) the DYAD Service (Figure 1). The DYAD Managed Directory (DMD) component manages near-computer storage accelerators such as node-local burst buffers by providing a unified namespace for DL training to cache their datasets. This unified global namespace allows applications to view several DMDs across multiple nodes as a unified distributed cache. The DYAD Library enables DL workloads' access to the DYAD Managed Directory using data loader defined for PyTorch and TensorFlow. For constructing the input pipeline, DYAD defines a data loader interface for PyTorch. This is because we cannot transparently intercept I/O calls from frameworks like PyTorch and NVIDIA Dali due to the limitations associated with preloading libraries (via LD_PRELOAD) in DL training. Once the DYAD Library is integrated into a DL application, the application can retrieve samples from the DYAD Managed Directory located on either the same node or a remote one. Access to samples on remote nodes involves communication with the respective node's DYAD Service. A DYAD Service is a multi-threaded program that is deployed as a process per node to efficiently shares samples from its Managed Directory (DMD) with other nodes through a novel Streaming RPC with RDMA protocol. DYAD uses a Hierarchical Sample Locator (HSL) for sample discovery. Initially, the DYAD Library searches the local DMD for the sample. If the sample is not local, the DYAD Library contacts its local DYAD Service to determine its location. Should another process on the node have previously retrieved the sample, the local DYAD Service provides its location. Otherwise, the DYAD Service consults the Global Sample Locator (GSL) to identify the sample's destination. The GSL is a distributed metadata store hosted on every node's DYAD Service. This hierarchical approach to sample discovery allows DYAD to enhance metadata scalability across the cluster.

Figure 2 shows an example of an I/O worker utilizing DYAD to load a sample. In the figure, DL training uses a data loader to build an input pipeline for reading the dataset. This pipeline is managed by independent I/O workers and is initiated by the GPU process. These workers read the data and then supply it to the GPU process. The I/O worker requests assistance from the DYAD Library to read a sample (Step 1). Using the HSL, the library identifies the sample's location within DYAD. If the sample is available in the local DYAD Managed Directory (DMD), it is directly delivered to the I/O

worker. Otherwise, the DYAD Library asks the remote DYAD Service to send the sample to the local DMD (Step 2). Upon receiving the request, the remote DYAD Service caches a sample from the local DMD (Step 3) and transfers the sample to the I/O worker's memory through RDMA Put (Step 4). Once the sample becomes accessible to the worker, it is locally stored in the DMD for subsequent use (Step 5). A pointer to the sample is also provided to the I/O worker for consumption.

B. Streaming RPC with RDMA Protocol

DYAD enhances network bandwidth for inter-node data transfers, a crucial aspect of DL training where data movement efficiency can significantly impact overall performance. Traditional inter-node transfers [13, 29] typically rely on a two-sided RPC protocol. However, state-of-the-art solutions such as DataSpaces [16], HCL [30], and UnifyFS [15] demonstrate the potential of combining RPC with RDMA protocol to speed up inter-node data access. These solutions usually dynamically create and register buffers for RDMA transfer. This dynamic process, while flexible, introduces a considerable overhead that can limit scalability to peak network bandwidth.

To address these limits, DYAD implements two key strategies in its inter-node data transfer design: the use of preallocated buffers and the adoption of streaming RPC. First, DYAD uses preallocated buffers, a common practice in benchmarks like `perftest` and `ucx-perftest`, to enhance network bandwidth. While preallocating buffers introduces additional memory management costs, especially for large data transfers, DYAD addresses these with a unique, decoupled approach. This approach combines Streaming RPC and RDMA atomic calls for efficient management of large data transfers. Since most DL datasets contain uniformly distributed samples [1, 22, 31–34], DYAD's strategy ensures that it never exceeds preallocated buffers, therefore, maximizing network bandwidth for DL training. For cases with varying sample size distribution, DYAD's data transfer protocol is equipped to handle data transfer using multiple RDMA calls. Additionally, DYAD's preallocated buffers enable a configurable and constant memory footprint on the CPU memory for data transfers. Second, DYAD opts for streaming RPC over the traditional two-sided RPC to introduce a more efficient way of handling notifications and managing the I/O worker's memory through a lighter control plane. This streaming RPC mechanism is crucial for efficient inter-node data transfers with preallocated buffers. DYAD uses lightweight notifications to signal buffer transfers and effectively manage data movement during network congestion.

To ensure an efficient and streamlined data transfer process between remote services and local storage systems, DYAD's streaming RPC employs the RDMA protocol. This protocol facilitates the movement of data sizes that may exceed the sizes of predefined buffers by utilizing a ring buffer approach for RDMA transactions. Figure 3 provides a straightforward illustration of this protocol's operational flow, offering an intuitive understanding of its functions. The DYAD Library initially preallocates and registers a buffer for RDMA operations (Step 1). This buffer consists of a contiguous memory block,

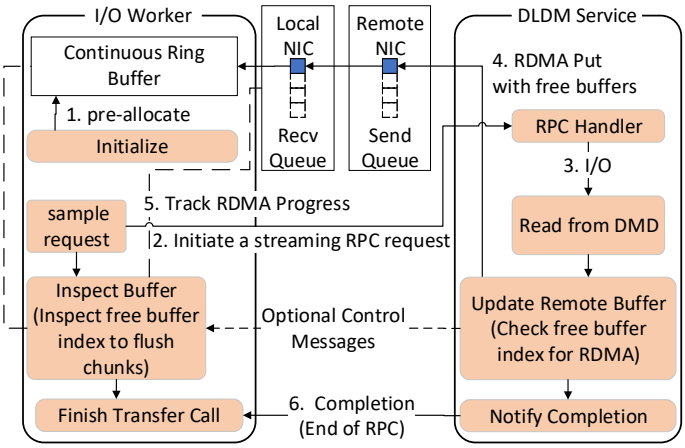


Figure 3: The Streaming RPC enables lightweight control messages between I/O worker and DYAD Service. The DYAD Service moves data through RDMA Put while the I/O worker tracks RDMA progress and flushes data into local DMD.

beginning with a 96-bit header: the first 64 bits for the transfer size and the next 32 bits for identifying the next free buffer position. At the start, the transfer size is set to -1, and the free buffer position is set to the last index of the buffer. Streaming RPC is initiated at the destination DYAD Service when a sample is read from a remote DMD. For the initial communication between the I/O worker and the DYAD Service, an RDMA memory pointer exchange occurs to establish a connection for RDMA data transfers (Step 2). A pre-existing RDMA connection is used for subsequent interactions, enhancing data transfer efficiency throughout the application's runtime. Following the connection setup, the DYAD Service retrieves the sample from its local DMD (Step 3). If the sample size is within the limit of the preallocated buffer, it is directly transferred to the I/O worker's memory using an RDMA Put (Step 4). However, if the sample size exceeds the buffer limit, the protocol initially employs a control plane RDMA Put to update the first 96 bytes of the buffer, indicating the total transfer size and resetting the free buffer position to zero. This is followed by the transfer of the data in 1MB chunks (Steps 4 and 5). The I/O worker monitors the transfer progress via its local NIC, sequentially storing the data on its local DMD and updating the free buffer position in RDMA memory. This DYAD transfer process ensures that the I/O worker can adequately manage the incoming data rate, balancing the intensive remote transfers with local buffered I/O operations. If the I/O worker falls behind in the transfer process of the incoming data, this is detected by the DYAD Service through an unchanged free buffer position, prompting the service to poll the memory for available buffer space before resuming data transfers. By doing so, DYAD manages data transfers independently, leveraging the efficiency of RDMA's one-sided operations. The completion of the data transfer is signaled by the DYAD Service to the I/O worker (Step 6), marking the conclusion of the operation.

C. Data Movement Coordination

In the training of DL models, it is common for I/O workers to independently access samples from a dataset, employing a technique known as embarrassing parallelism [35]. This approach allows for multiple sample requests to be processed simultaneously, with these requests being directed toward both local and remote DMDs. DL models are designed to access a unique set of samples during a single training epoch. However, given that DL training datasets can comprise thousands of samples per file, as noted in several studies [23, 32, 33], this can lead to numerous I/O requests targeting the same file concurrently. To address multiple I/O requests efficiently, DYAD uses file-level data transfers, optimizing sample locality for subsequent accesses and efficiently using network bandwidth for data transfers between nodes, particularly since DL dataset samples typically are less than 512 KB in size [23, 32, 34]. These characteristics of DL datasets, combined with DYAD's approach to inter-node file-level data transfer, underscore the need for coordinated data movement.

DYAD uses three strategic approaches to streamline data movement coordination for efficient deep learning (DL) training: file-level granularity, file-lock-based synchronization, and passive coordination. These strategies collectively enable DYAD to manage data movement more efficiently, reducing waiting times and optimizing the use of network and storage resources during DL training. Adopting a *file-level granularity* approach, DYAD coordinates access to complete files rather than targeting individual samples within those files. This strategy capitalizes on cache locality to enhance access speed and effectively increases network bandwidth for data transfers. When confronted with multiple simultaneous requests for access to the same remote file, DYAD implements a file-lock-based synchronization mechanism on DMD. This mechanism ensures that only a single process can perform the file transfer at a time when concurrent requests occur. Subsequent processes must wait until the transfer concludes, at which point they can access the required sample directly from the local DMD, thereby minimizing unnecessary data movements. Contrary to methods that delay data access until a pool of requests is sorted and prioritized [36–39], DYAD adopts a *passive coordination* strategy. This strategy initiates data transfers immediately upon the first request by an I/O worker to access a file on the PFS, bypassing active coordination among processes. Subsequent access requests are either delayed until the current transfer is complete or diverted to a locally cached copy on the DMD, making the data access process more efficient.

Figure 4 illustrates three instances matching the three fundamental movements and coordination strategies that form the backbone of any data movement operation in DYAD. In the figure, the term W denotes an I/O worker accessing a sample within a file. The instance on the left (Case 1) shows a *sequential I/O worker access on the same node.* Two I/O workers, *W1* and *W2*, are on the same node, accessing data in sequence. *W1* goes first, pulling data from PFS into the local DMD. *W2*, arriving afterward, bypasses the PFS, directly

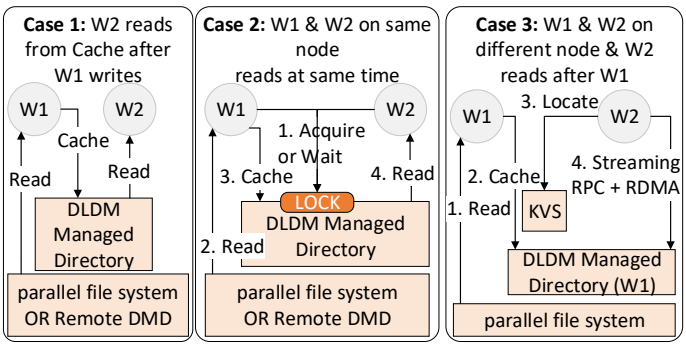


Figure 4: DYAD coordinates I/O workers fetching the same file by using a file-lock-based mechanism to reduce redundant I/O calls to PFS and inter-node data transfer.

accessing the data from the DMD, which now contains the file cached by *W1*. This setup allows *W2* to retrieve the sample immediately from the DMS without further coordination, thanks to *W1*'s prior caching, demonstrating an efficient data access strategy within the same node. The instance in the middle (Case 2) shows a *simultaneous I/O worker access on the same node.* The two I/O workers, *W1* and *W2*, from the same node attempt to access data from the same file at the same time. They both try to acquire a file lock to read from the PFS. *W1* obtains the lock first, enabling it to fetch and cache the file in the local DMD. After completing the transfer and caching the file, *W1* releases the lock. *W2*, having waited for *W1* to finish, then accesses the now locally cached data directly from the DMD. This process allows DYAD to minimize unnecessary I/O operations and data transfers, enhancing efficiency in data access within the node. The instance on the right (Case 3) shows a *sequential I/O worker access across different nodes.* The two I/O workers, *W1* and *W2*, operate from different nodes. *W1* is the first to access the data, reading and caching a file in its local DMD. It then asynchronously informs the local HSL. If *W1* completes this notification before *W2* starts its operation, DYAD initiates an inter-node data transfer to replicate the file in *W2*'s local DMD. As a result, *W2* can directly access the cached sample from its own DMD, leveraging remote DMD reuse and streamlining the data access process across nodes. These strategies are not only effective individually for managing data within and across nodes but can also be integrated and used in concert, enhancing each other in various combinations. This integration is essential for DL training operations' scalability, performance, and flexibility.

D. Hierarchical Sample Discovery

In distributed DL training, I/O workers often locate specific data samples across various training epochs repeatedly. This process involves identifying the file and its offset (i.e., the specific part where the sample is stored), locating that file, and then reading the sample from the specified offset. Multiple I/O workers can simultaneously look for different samples within the same file. Moreover, the same sample can be accessed by different workers across various training epochs.

To address these challenges, DYAD uses a multi-level hierarchical system to efficiently find samples, which helps prevent bottlenecks when multiple I/O workers access data during the distributed DL training. DYAD's HSL uses three strategic approaches to enhance the process of finding samples. DYAD uses *Lookup Isolation* to streamline the process of locating data samples and avoid duplication of effort among I/O workers. I/O workers prioritize local sources first, reducing redundant searches and balancing the workload among them. A worker starts with the local DMD for immediate access, moves to the local HSL on the DYAD Service for intermediate storage, and finally resorts to the global HSL for a comprehensive search across nodes. The local DMD uses FS metadata for quick searches, the local HSL serves as an efficient, node-specific key-value store, and the global HSL distributes the search process across nodes through its hierarchical tree structure of the key-value stores. DYAD implements *Metadata Caching* to speed up future sample access. As samples are accessed, their metadata is stored at various levels (local DMD, local HSL, and global HSL), making it quicker for I/O workers to find them next time. This caching mechanism also has a hierarchy, with local caches serving immediate needs and global caches providing a backup. When an I/O worker accesses a sample, the metadata is saved in the local DMD for quick retrieval on the same node. If the local DMD is full and thus evicts this data, the search moves to the local HSL as a second-level cache with longer lookup times. Failing to find the metadata locally, the search progresses to the global HSL. Although accessing the local HSL is one order of magnitude slower than the DMD, it is one order of magnitude faster than reaching out to the global HSL (Section IV-E), delivering an optimal balance between retrieval speed and network resource use for DYAD's distributed environment. DYAD enhances data retrieval speeds and resource efficiency through *Lookup Concurrency*, allowing multiple I/O workers to locate samples simultaneously, thus optimizing concurrent access for better throughput. The I/O workers follow the established sequence of *Lookup Isolation*, starting with the local DMD, then the local HSL, and finally, the global HSL. Such a structured approach to parallelism ensures that workers achieve a higher discovery rate of samples, especially from local sources such as DMD, compared to more extensive network searches like those required for the global HSL.

Figure 5 illustrates four distinct use cases for hierarchical sample discovery within DL training using DYAD. In each use case, two workers access samples from the same file but encounter different caching situations. In the figure, "W" denotes the I/O worker accessing a sample within a file. In *Case 1, W1* and *W2* are on the same node accessing the same sample, with *W2* finding it in the local DMD because *W1* had already cached it there. In *Case 2,*, similarly to Case 1, *W1* and *W2* are on the same node. However, the sample initially cached and accessed by *W1* has been removed before *W2*'s search begins. Consequently, *W2*'s attempt to locate the sample using the local DMD fails, leading *W2* to successfully find the sample in the local HSL. In *Case 3, W1* and *W2* are on different nodes.

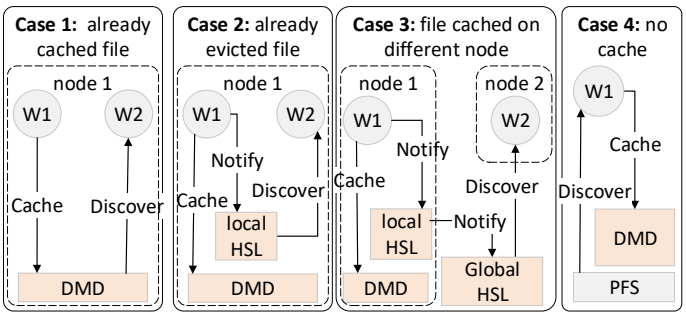


Figure 5: The Hierarchical Sample Locator (HSL) caches the discovery operations from I/O workers in different levels to support metadata scalability for DL training.

W1 had already cached the samples locally, and thus *W2* finds it in the global HSL due to a miss in its local caches. In this case, the global HSL is a tree, and, in the worst case, the file with the sample can be the root node of HSL hierarchical structures. Finally, in *Case 4, W1* reads a sample not yet cached in DYAD. In this case, the DYAD's HSL lookups fail, and the file is located directly on the parallel file system. These cases illustrate DYAD's hierarchical sample discovery process supports diverse and efficient data access in DL training environments. By caching sample discovery operations at various levels, DYAD maximizes metadata throughput across a cluster.

E. Implementation Details

The DYAD software comprises core library, DYAD APIs, DYAD wrappers, and the DYAD Service. The core library, written in C, provides essential external interfaces to facilitate the development of various use cases. Internally, the core library employs the POSIX interface for data transfer to and from the DYAD Managed Directory (DMD). DYAD APIs feature explicit initialization and finalization routines, enhancing the core library's functionality. Configuration of DYAD can be achieved through environment variables and direct API calls during its initialization phase. The wrappers ensure productivity and portability. They are developed in multiple languages: POSIX and STDIO wrappers in C, file stream wrappers in C++, and wrappers for PyTorch Data Loader, Numpy array, and h5py in Python. The DYAD Service, integrated as a module within the Flux Broker manages Streaming RPC interactions with I/O workers, hosts the Hierarchical Sample Locator, and performs cache eviction. DYAD adopts Streaming RPC through Flux Streaming RPC and RDMA calls via Unified Communication X (UCX), leveraging UCX's atomic and one-sided operations. Initial setup of DYAD includes UCX infrastructure initialization and a network jitter-minimizing warmup message using a loopback address. To accelerate RDMA transfers, DYAD utilizes UCX for page-aligned memory allocation. The Hierarchical Sample Locator in the DYAD Service employs the Flux Key-Value Store (KVS) [40], meeting two essential criteria. First, the Flux KVS locally caches lookup operations, facilitating isolation in metadata discovery. Second, its hierarchical structure enables controlled isolation of remote

metadata calls for different workloads. Our experiments utilize a flat topology within the Flux KVS.

IV. Evaluation

We assess the scalability and performance of the DYAD by examining its key technologies, including Streaming RPC with RDMA, data movement coordination, and Hierarchical Sample Locator. We also evaluate DYAD's ability to optimize DL training and its performance compared to leading solutions like DataSpaces and UnifyFS.

A. Methodology

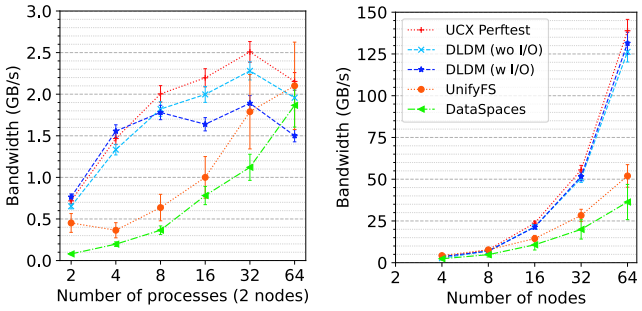
1) Hardware.: We run our experiments on the Corona cluster, comprising 121 nodes with a 48-core AMD Rome processor, 256GB of memory, 1.5 TB NVMe SSD, and eight AMD MI50 GPUs each. We weak scale the DL workloads to 64 nodes.

2) Software.: We run the DL workloads using Python 3.9.12, OpenMPI 4.1.2, and UCX 1.15.0. The Python packages included Torch 2.2.1, h5py 3.10.0, and Numpy 1.26.0. This software setup enables multi-processing and distributed training with MPI, alongside UCX communication for the PyTorch workload. To trace Python functions and I/O calls, we utilized DFTracer version 1.0.2.

3) Workloads.: We use micro-benchmarks and DL workloads in PyTorch, including UNet3D [22] (available on GitHub [41]) and MuMMI DL Training [6, 23], to evaluate DYAD's performance. DYAD was integrated with these workloads using the PyTorch data loader interface [42]. We measure metrics such as data transfer bandwidth, throughput, and execution time across tests, with results replicated ten times to assess variability. We compare DYAD with Lustre [43], UnifyFS [15], and DataSpaces [16, 17] on HPC cluster. These solutions are architecturally closest to DYAD and utilize state-of-the-art network protocols to share data efficiently within HPC workloads. We acknowledge that these solutions were not originally designed for DL workloads. However, as seen in our results, UnifyFS optimizes DL workloads by 3×. DYAD outperforms UnifyFS and DataSpaces due to improved data locality and metadata optimizations performed for DL workloads. DataSpaces is tested with 16 processes per node as it has scalability issues beyond that for 64 nodes. For Dyad, UnifyFS, and DataSpaces, we allocate one core for the Server for internal tests and we use optimal Server parameters to run real-world AI workloads. Attempts to use HVAC and NoPFS are hindered by compatibility issues with our AI workloads. Additionally, deploying BeeGFS and Alluxio on our Corona cluster required superuser privileges, which are unavailable for users.

B. Scalability of inter-node sample access

To measure the efficiency of our Streaming RPC with RDMA protocol for inter-node data movement, we conduct tests varying from 2 to 64 nodes, assessing the scalability of data transfers. These tests ranged from using a single process per node up to 32 processes per node and selecting the best configuration from the two-node setup to apply to multi-node scenarios. Each process performs sixteen 1 MB transfers between nodes to measure bandwidth. Figures 6(a) and 6(b) show the scalability of our protocol; the x-axis indicates the scale of the test, and the y-axis indicates bandwidth in GB/s. We examine our protocol's network bandwidth under two conditions: with I/O operations (DYAD with I/O) and without I/O operations (DYAD without I/O). DYAD with I/O mirrors the total bandwidth for sample movement, while DYAD without I/O focuses exclusively on the network's performance in sample transfer.

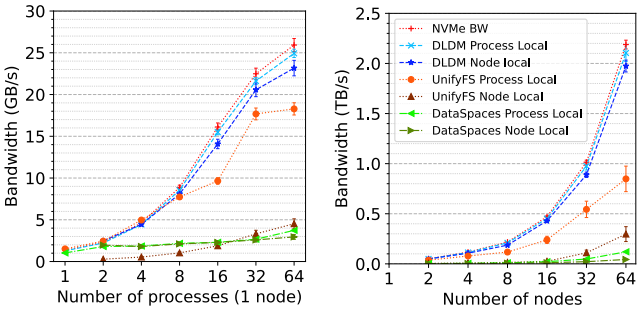


(a) Two-node scaling test. (b) Multi-node scaling test.

Figure 6: The Streaming RPC with RDMA protocol is bound by the I/O at small-scale and by the network at large-scale.

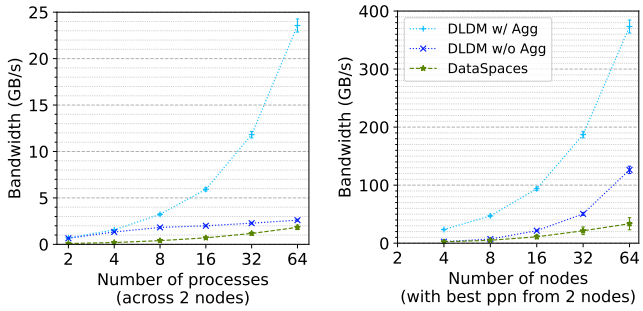
In Figures 6(a) (two-node test), we observe that the peak bandwidth for `ucx-perftest` reaches 2.5 GB/s with 32 processes. UnifyFS approaches the network limit but experiences significant overhead with lower process counts (4.2 times slower with four processes) compared to `ucx-perftest` due to dynamic buffer allocation, which impacts performance. DataSpaces is between 1.6 and 9.5 times slower than `ucx-perftest`, as it creates dynamic connections and allocates buffers, lowering its data transfer speed. For DYAD with I/O, bandwidth peaks at 1.9 GB/s, matching the maximum performance of a single process on node-local NVMe for 1 MB requests. DYAD without I/O achieves similar scalability to `ucx-perftest`, reaching a peak bandwidth of 2.3 GB/s. The higher bandwidth with DYAD without I/O is due to using Streaming RPC for initiating transfers, followed by RDMA Put operations for data movement. Adding memory allocation operations reduces the bandwidth by up to 8x; therefore, within DYAD, we preallocate the buffers and maximize network bandwidth.

In Figure 6(b) (multi-node test), we observed that the bandwidth for `ucx-perftest` scales with the number of nodes due to the UCX capacity for moving larger volumes of data across the network as the number of nodes increases. Both UnifyFS and DataSpaces demonstrate nearly linear scaling with the number of nodes. However, they also exhibit performance overheads due to memory allocation and the establishment of dynamic connections, resulting in UnifyFS and DataSpaces being up to 2.5 times and 3.6 times slower, respectively, than `ucx-perftest`. Unlike in the two-node test, both configurations of DYAD (with and without I/O) closely match the scaling performance of `ucx-perftest` because the aggregate bandwidth of node-local storage increases lin-



(a) One node scaling test. (b) Multi-node scaling test.

Figure 7: DYAD's lightweight node-local HSL enables close-to-device bandwidth for accessing samples during DL training.



(a) Two node scaling test. (b) Multi-node scaling test.

Figure 8: Coordination accesses for samples on the same file can improve network bandwidth by up to a 3-fold increase.

early with the number of nodes. Using a round-robin approach, the inter-node data transfer protocol of DYAD reaches up to 135 GB/s in bandwidth for data transfers across 64 nodes. Performance at the small scale is limited by the I/O bandwidth of node-local storage, whereas at the large scale, the limiting factor shifts to network bandwidth, which scales sub-linearly compared to the linear scalability of node-local storage.

Findings 1: *The Streaming RPC with RDMA protocol exhibits scalability in inter-node data transfers, with performance being I/O-bound at smaller scales and network-bound at larger scales.*

C. Scalability of node-local sample access

To measure the scalability of node-local accesses, we conducted tests ranging from a single node to 64 nodes. In the case of a single node, the tests varied from one to 64 processes per node. For the multi-node tests, we scale the number of nodes from the best scenario identified in the single-node tests. Each process made 16 requests, each 1 MB in size, to assess the bandwidth. Figures 7(a) and 7(b) show the results of the single-node and multi-node tests, with the x-axis indicating the scale of the test and the y-axis indicating the bandwidth in GB/s. We include a baseline performance of our NVMe device for comparison against Process Local Access (when a process reads a sample that it previously accessed in the last epoch) and Node Local Access (when a process reads a sample that another process on the same node accessed).

In Figure 7(a) (one-node test), we observe that the peak bandwidth for node-local NVMe with OS caching is 25 GB/s for 64 processes. We do not scale beyond 64 processes as the maximum number of cores available on the node is 48. Process local accesses for UnifyFS closely follow the trend of NVMe up to 8 processes per node. After that, the cost of bookkeeping for appending to the log-structured storage [15] reduces its overall bandwidth by 40% for 64 processes per node. For DataSpaces, all accesses go through the mercury infrastructure [17], making it 6.9x slower than NVMe. For DYAD, both Process Local Access and Node Local Access closely follow the trends of the NVMe device as the sample discovery is done using local-DMD with a peak bandwidth of 23GB/s. This demonstrates that DYAD has minimal overhead for node-local sample accesses.

The multi-node test in Figure 7(b) also exhibits similar trends to that of the one-node test. The exclusive nature of node-local NVMe, the bandwidth linearly scales up with the number of nodes with a bandwidth of 2.2 TB/s for 64 nodes. This bandwidth includes the OS caching effect, which is a realistic case for most node-local accesses on modern supercomputers with large memories larger than 256GB. UnifyFS and DataSpaces are 2.5x and 50.1x slower than NVMe accesses. Similar to the two-node test, DYAD's Process Local Access and Node Local Access closely follow NVMe's bandwidth trend, with 2.0 TB/s bandwidth for 64 nodes.

Findings 2: *The locality-aware caching mechanism of DYAD facilitates nearly device-level bandwidth performance for node-local sample access, enabling it to scale efficiently to a bandwidth of 2.0 TB/s across multiple nodes.*

D. Impact of data movement coordination

To evaluate the impact of data movement coordination on inter-node access, we conducted scaling tests similar to those described in Section IV-B, executing 16 requests of 1 MB each across a range from two to 64 nodes. Each node reads a distinct sample from the same file. Figures 8(a) and 8(b) present the results for the two-node and multi-node tests, respectively. The x-axis indicates the scale of the test, while the y-axis indicates the bandwidth in GB/s. Having previously compared our protocol's bandwidth with `ucx-perftest`, we now focus on comparing our protocol with and without aggregation implementation (Section III-C). UnifyFS was excluded from this test as it does not support aggregation.

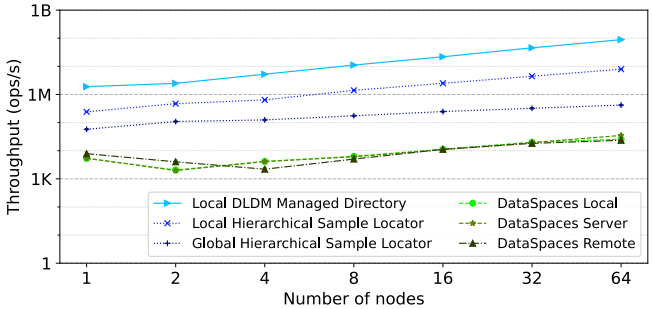
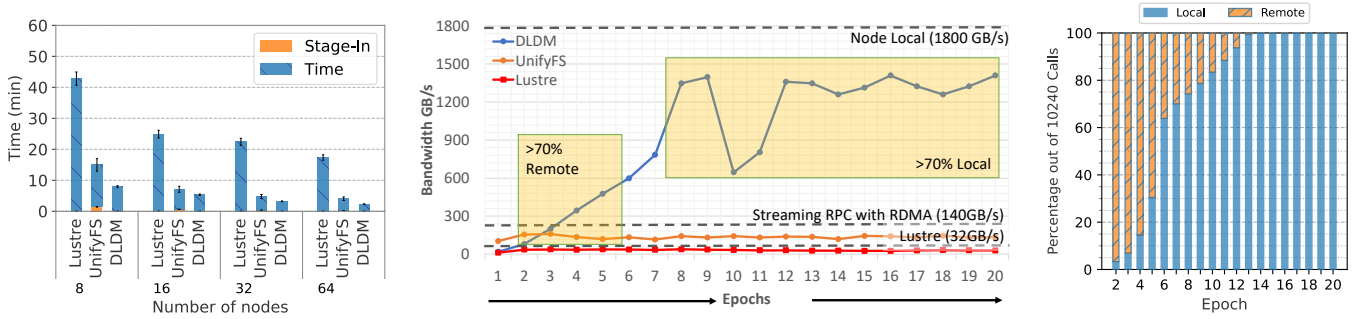


Figure 9: Isolating accesses through HSL improves sample discovery throughput by up to a 272-fold increase.



(a) Scaling Test. (b) Bandwidth per Epoch for 512 GPUs (64 nodes). (c) Local vs Remote for 512 GPUs.

Figure 10: DYAD improves the epoch time of Unet3D by up to 10.82x due to locality-aware caching as compared to UnifyFS.

In the two-node test depicted in Figure 8(a), DataSpaces sees a performance improvement of 5% when sending duplicate requests, benefiting mainly from aggregating sample lookup costs. For DYAD, both metadata and data aggregation occur. We note that the scalability with aggregation approaches linearity, attributed to a single process undertaking the data transfer while others access the sample from local DMD, as detailed in Section III-C. This adjustment boosts our inter-node bandwidth in the test scenario to 23 GB/s with aggregation compared to 2.1 GB/s without aggregation. The two-node test results, shown in Figure 8(b), demonstrate a similar trend. With aggregation, the test's perceived bandwidth increases to 380 GB/s, versus 128 GB/s without aggregation.

Findings 3: *Coordinating access to multiple samples within the same file significantly enhances the perceived inter-node bandwidth of the workload, achieving a 2.9-fold increase..*

E. Impact of hierarchical sample discovery

To evaluate the impact of hierarchical sample discovery, we conduct sample discovery tasks wherein each application process initiates 16 file lookups distributed across a scale from one to 64 nodes, each running 64 processes. This test involves storing the metadata for files across various levels of the Hierarchical Sample Locator (HSL) for DYAD's local Distributed Local Directory Manager (DLDM) managed directory, local HSL, and global HSL, in comparison to DataSpaces' local, server, and remote discovery mechanisms. The aim is to exclusively utilize the desired HSL level for DYAD. The scalability and performance metrics of each HSL level are calculated and analyzed. Figure 9 displays the results of the hierarchical sample discovery test, where the x-axis represents the scale of the test and the y-axis shows the throughput measured in operations per second. Our solution is only compared with DataSpaces, as it is not feasible to separately measure the metadata-related costs within UnifyFS.

For the local DMD test illustrated in Figure 9, we observe that DYAD's throughput of operations scales linearly with the number of nodes. Since lookup operations are isolated within each node, access scales efficiently by adding multiple nodes. Consequently, the throughput of HSL via local DMD reaches 90 million operations per second. This performance of local DMD is primarily dictated by the cost of file system metadata access on node-local NVMe storage. DataSpaces, in contrast, contacts its node-local server to locate the sample, rendering DYAD 3.5K× faster than DataSpaces. Transitioning from local DMD to local HSL, as shown in Figure 9, we notice that DYAD's throughput of operations maintains linear scalability across multiple nodes. The lookup costs remain isolated to each node, facilitated by the interaction between the locator process and the local DYAD HSL. Thus, the throughput of HSL through the local DYAD Service reaches approximately 8 million operations per second. Local DYAD Service's HSL performance is limited by inter-process communication within the Flux KVS for node-local accesses, occurring exclusively when the DMD cache is evicted. For DataSpaces, all node-local discoveries are executed using their node-local Service. In this scenario where DYAD has a sample in DMD, our solution is 3,400 times faster than DataSpaces. In contrast, when DYAD has evicted its sample from DMD, it is 308 times faster than DataSpaces. Last, in the global HSL test depicted in Figure 9, we observe that DYAD's operations throughput similarly scales linearly across multiple nodes. In this scenario, the lookup cost is external to the node and is thus bound by the network latency within the cluster. Specifically, the throughput for remote sample discovery using global HSL is 330,000 operations per second. Global access within DataSpaces involves a broadcast operation across all servers to locate data. This process yields DYAD's global HSL between 10 to 55 times faster than DataSpaces.

Findings 4: *HSL empowers DYAD to isolate sample discovery, leading to orders of magnitude higher in performance as DL training accesses samples multiple times across several hundred epochs.*

F. Real DL Training workloads

We compare DYAD with UnifyFS and DataSpaces in large-scale DL training workloads, Unet3D and MuMMI. For Unet3D, DYAD and UnifyFS are evaluated in a scenario ideal for UnifyFS, involving rapid metadata lookup and data transfer with a single sample per file. For MuMMI, DYAD is compared with DataSpaces, suited for efficiently accessing samples from large files. DYAD loads the dataset directly from the PFS in the first epoch and caches it in a distributed cache. In contrast, UnifyFS and DataSpaces stage the dataset into their services before starting DL training. The dataset

used in DL training is based on the actual dataset used by scientists in their DL workload.

1) Unet3D: is a network for volumetric segmentation in medical imaging, using a 20-layer deep convolutional network to learn from 10,240 sparsely annotated images (140 MB each, totaling 1.36 TB in NPZ format) [44]. Utilizing PyTorch for data parallel training, the model processes batches of four images across six threads per GPU over twenty epochs without checkpointing, scaling from 8 to 64 nodes with eight GPUs each. We analyze performance metrics like runtime and bandwidth per epoch for Luster, UnifyFS, and DYAD as well as the per-epoch distribution of remote vs. local calls for DYAD only using the DLIO Profiler [45].

Figure 10(a) demonstrates DYAD's runtime improvement for Unet3D, showing a significant performance boost over Lustre and UnifyFS, mainly due to data locality optimizations. DYAD achieves up to 7.5 times faster processing than Lustre and 1.88 times than UnifyFS, with less performance variability due to its use of node-local storage. Figure 10(b) details bandwidth per epoch performance across 512 GPUs, highlighting DYAD's progressive bandwidth increase—starting at Lustre levels, then soaring to 140GB/s with Streaming RPC with RDMA, and finally reaching 1409 GB/s as locality-aware caching makes 70% of calls local. The transition beyond the eighth epoch shows computation costs becoming the dominant factor in training time with DYAD. Compared to UnifyFS, Dyad significantly speeds up the training time by 10.62 for the 20th epoch. As the process progresses and more data becomes cached locally, a stable bandwidth of approximately 1,400 GB/s for subsequent epochs indicates efficient data access and utilization. Figure 10(c) illustrates how local versus remote calls per epoch impact the bandwidth achieved for each epoch in the context of 512 GPUs. Epochs 10 and 11 decrease performance due to remote calls in some batches, which lowers the overall bandwidth.

Findings 5: *DYAD significantly improves the performance and efficiency of Unet3D's data processing for medical image segmentation by optimizing data locality compared to conventional caching solutions.*

2) MuMMI: The Massively Parallel Multiscale Machine-Learned Modeling Infrastructure (MuMMI) is a framework designed for executing multiscale modeling simulations of large molecular systems, integrated through ML techniques [6]. MuMMI has been instrumental in studying the interactions between the plasma membrane and RAS-RAF-14-3-3 protein complexes, among the proteins most frequently mutated in various cancer types. Our workflow, developed in PyTorch, employs a map-style data loader for HDF5 datasets, comprising 600 files with 8k samples and 8,691 elements of size 8 bytes. We adopt the DLIO Benchmark to execute this workload over ten epochs, achieving a simulated computation time of 133 ms. The workload scales up to 64 nodes. We analyze runtime and bandwidth per epoch for Luster, DataSpaces, and DYAD and the per-epoch distribution of remote vs. local calls for DYAD.

Figure 11(a) presents the end-to-end scaling runtime where the Lustre file system significantly reduces the overall runtime from 26 to 12 seconds when scaling from eight to 64 nodes. This figure also demonstrates that DataSpaces' performance is adversely affected compared to Lustre's. The per-epoch bandwidth is detailed in Figure 6(b), indicating DataSpaces' limited performance due to an inter-node data transfer rate of only 11 GB/s for 64 nodes. Moreover, the fact that the dataset dimensions are not a power of 2 negatively impacts metadata scalability in DataSpaces because of the uniform hashing algorithm. This, along with the bandwidth limitation, renders DataSpaces 3.8 to 4.1 times slower than Lustre. Furthermore, DataSpaces struggles to efficiently cache samples for eight nodes, leading to a segmentation fault at the eighth epoch, despite the aggregate memory being four times the dataset size. On the other hand, DYAD is showcased to be 1.8 to 2.4 times faster than Lustre, with its peak per-epoch bandwidth reaching 29 GB/s for the MuMMI dataset, as illustrated in Figures 11(b) and 6(b). Despite a significant I/O bandwidth of 1,762 GB/s on average after the first epoch, the predominantly computational nature of the MuMMI workload means these bandwidth improvements do not translate into substantial performance benefits. However, the availability of most data locally after the first epoch significantly boosts DYAD's performance, a phenomenon highlighted in Figure11(c).

Findings 6: *The DYAD data management system boosts MuMMI's modeling efficiency by outperforming Lustre and DataSpaces, notably by quicker caching of smaller file workloads in the node-local DMD.*

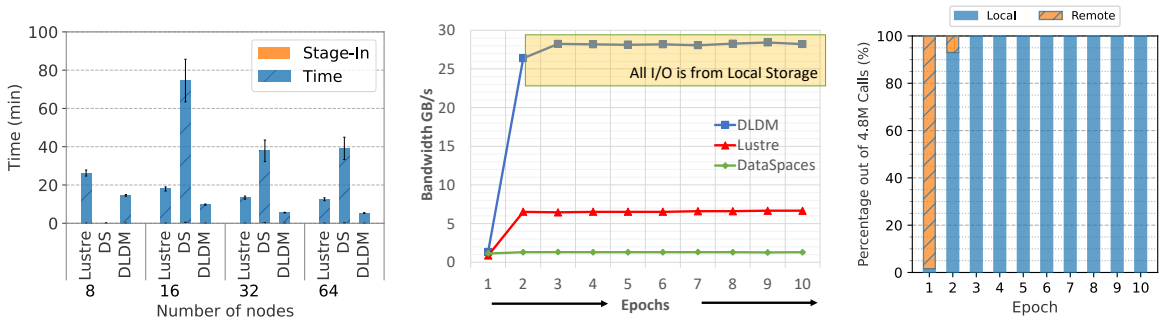
V. Conclusions

DL training involves processing large datasets over many epochs on HPC systems, creating opportunities for speed improvements through strategic data caching. Our solution, DYAD, introduces a locality-aware cache for DL datasets over near-compute storage accelerators. DYAD replaces dynamic memory allocations for data transfer between nodes with a more efficient streaming RPC with RDMA protocol with pre-allocated buffers, thus maximizing network bandwidth. DYAD adopts locality-aware caching with sample redundancy to address the inefficiency of traditional caching methods due to DL's per-epoch full dataset read. DYAD orchestrates data movement to avoid redundant I/O operations and network congestion, boosting sample reading throughput by 8.78x. Last, DYAD uses a hierarchical metadata management approach to drastically reduce lookup times by two orders of magnitude faster discovery. We demonstrate that DYAD enhances training speeds for Unet3D and MuMMI models by up to 10.82$\times$ compared with state-of-the-art solutions on a 512 GPU Corona cluster.

In the future, we will enhance our current design with the capability to move partial samples within the distributed cache. This would require finer metadata management and further reduce data redundancy for communication.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National



(a) Scaling Test. (b) Bandwidth per Epoch for 512 GPUs. (c) Accesses for 512 GPUs.

Figure 11: DYAD's caches the entire dataset during the first epoch and optimizes I/O for future several hundred epochs.

Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 24-SI-005. This research used resources from the Oak Ridge Leadership Computing Facility (OLCF), a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. For computing time, the authors thank the Advanced Scientific Computing Research Leadership Computing Challenge (ALCC) for time on Summit and the Livermore Institutional Grand Challenge for time on Lassen. LLNL-CONF-862383. IL and MT acknowledge the support of NSF #2138811 and #2331152.

References

[1] R. Acciarri, C. Adams, and C. Andreopoulos, "Cosmic Background Removal with Deep Neural Networks in SBND," Apr. 2021, arXiv:2012.01301 [physics]. [Online]. Available: http://arxiv.org/abs/2012.01301

[2] K. H. Scheutwinkel, D. Grün, B. Jones, J. G. Lozano, and V. Knecht, "AI for Cosmology," in *AI for Physics*. CRC Press, 2022, num Pages: 18.

[3] A. Agrawal and A. Choudhary, "Deep materials informatics: Applications of deep learning in materials science," *MRS Communications*, vol. 9, no. 3, pp. 779–792, Sep. 2019.

[4] B. L. DeCost, J. R. Hattrick-Simpers, Z. Trautt, A. G. Kusne, E. Campo, and M. L. Green, "Scientific AI in materials science: a path to a sustainable and scalable paradigm," *Machine Learning: Science and Technology*, vol. 1, no. 3, p. 033001, Jul. 2020, publisher: IOP Publishing. [Online]. Available: https://dx.doi.org/10.1088/2632-2153/ab9a20

[5] X. Wu, V. Taylor, J. M. Wozniak, R. Stevens, T. Brettin, and F. Xia, "Performance, Energy, and Scalability Analysis and Improvement of Parallel Cancer Deep Learning CANDLE Benchmarks," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19. New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3337821.3337905

[6] "Generalizable coordination of large multiscale workflows | Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis." [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3458817.3476210

[7] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath, "DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2021, pp. 81–91.

[8] M. Paul, S. Ganguli, and G. K. Dziugaite, "Deep Learning on a Data Diet: Finding Important Examples Early in Training," *Advances in Neural Information Processing Systems*, vol. 34, pp. 20 596–20 607, Dec. 2021. [Online]. Available: https://proceedings.neurips.cc/paper/2021/hash/ac56f8fe9eea3e4a365f29f0f1957c55-Abstract.html?ref=https://githubhelp.com

[9] X. Zhou, C. Chai, G. Li, and J. Sun, "Database Meets Artificial Intelligence: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 3, pp. 1096–1116, Mar. 2022, conference Name: IEEE Transactions on Knowledge and Data Engineering. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9094012

[10] T. T. Nguyen, F. Trahay, J. Domke, A. Drozd, E. Vatai, J. Liao, M. Wahib, and B. Gerofi, "Why Globally Re-shuffle? Revisiting Data Shuffling in Large Scale Deep Learning," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2022, pp. 1085–1096, iSSN: 1530-2075. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9820654

[11] L. Yu, L. Liu, C. Pu, M. E. Gursoy, and S. Truex, "Differentially Private Model Publishing for Deep Learning," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 332–349, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8835283

[12] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning I/O," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 1–15. [Online]. Available: https://dl.acm.org/doi/10.1145/3458817.3476181

[13] A. Khan, A. K. Paul, C. Zimmer, S. Oral, S. Dash, S. Atchley, and F. Wang, "Hvac: Removing I/O Bottleneck for Large-Scale Deep Learning Applications," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2022, pp. 324–335, iSSN: 2168-9253. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9912705

[14] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sep. 2018, pp. 145–156, iSSN: 2375-0227. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8526881

[15] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, and S. Oral, "UnifyFS: A User-level Shared File System for Unified Access to Distributed Local Storage," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2023, pp. 290–300, iSSN: 1530-2075.

[16] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: an interaction and coordination framework for coupled simulation workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: Association for Computing Machinery, Jun. 2010, pp. 25–36. [Online]. Available: https://doi.org/10.1145/1851476.1851481

[17] S. Klasky, M. Wolf, M. Ainsworth, C. Atkins, J. Choi, G. Eisenhauer, B. Geveci, W. Godoy, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, A. B. Maccabe, K. Mehta, G. Ostrouchov, M. Parashar, N. Podhorszki, D. Pugmire, E. Suchyta, L. Wan, and R. Wang, "A View from ORNL: Scientific Data Research Opportunities in the Big Data Age," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, Jul. 2018, pp. 1357–1368, iSSN: 2575-8411. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8416399

[18] A. Audibert, Y. Chen, D. Graur, A. Klimovic, J. Šimša, and C. A. Thekkath, "tf.data service: A Case for Disaggregating ML Input Data Processing," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 358–375. [Online]. Available: https://doi.org/10.1145/3620678.3624666

[19] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," Feb. 2018, arXiv:1802.05799 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1802.05799

[20] H. Li, *Alluxio: A Virtual Distributed File System - ProQuest*. University of California, Berkeley: Pro Quest, 2018. [Online]. Available: https://www.proquest.com/openview/f7300db370f5701bec0b46f6807c35d3/1?pq-origsite=gscholar&cbl=18750

[21] M. Amin and R. M. Rahman, "Universal database access layer to facilitate query," in *Ninth International Conference on Digital Information Management (ICDIM 2014)*, Sep. 2014, pp. 145–150. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6991401?casa_token=q3z030NbSMoAAAAA:wJoYSmCuiH2eo7hp2e-OMC5PK_zbpkeoeR_jLEYxX4o4SqZjKYcyb6Iy48ETchE87RTKRUfR

[22] F. Turk, M. Luy, and N. Barisci, "Comparison of Unet3D Models for Kidney Tumor Segmentation," Jan. 2020. [Online]. Available: https://www.preprints.org/manuscript/202001.0314/v1

[23] J. Y. Moon, F. Di Natale, H. I. Ingolfsson, H. Bhatia, and J. R. Chavez, "Multiscale Machine-Learned Modeling Infrastructure

RAS," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. MuMMI RAS, Sep. 2021.

[24] H. Devarajan and K. Mohror, "Extracting and characterizing I/O behavior of HPC workloads." Heidelberg, Germany: IEEE, Sep. 2022.

[25] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable Deep Learning via I/O Analysis and Optimization," *ACM Transactions on Parallel Computing*, vol. 6, no. 2, pp. 6:1–6:34, Jul. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3331526

[26] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, K. Mohror, D. Ivanov, T. Wang, and C. P. Steffen, "UnifyFS: A Distributed Burst Buffer File System - 0.1.0," Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States), Tech. Rep. UnifyFS, Oct. 2017. [Online]. Available: https://www.osti.gov/biblio/1408515

[27] A. V. Kumar and M. Sivathanu, "Quiver: An Informed Storage Cache for Deep Learning," 2020, pp. 283–296. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/kumar

[28] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "FanStore: Enabling Efficient and Scalable I/O for Distributed Deep Learning," Sep. 2018, arXiv:1809.10799 [cs]. [Online]. Available: http://arxiv.org/abs/1809.10799

[29] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 219–230.

[30] H. Devarajan, A. Kougkas, K. Bateman, and X.-H. Sun, "HCL: Distributing Parallel Data Structures in Extreme Scales," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2020, pp. 248–258, iSSN: 2168-9253. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9229595?casa_token=xaWZMLxIUlwAAAAA:h_tQEUqsyeU_n2xre9sqtALiFirS_4ejTTKPhhSohhJ7ev078vShhCGn-Qaep696lBpPWv_q

[31] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, K. Maschhoff, J. Sewall, N. Kumar, S. Ho, M. F. Ringenburg, P. Prabhat, and V. Lee, "CosmoFlow: Using Deep Learning to Learn the Universe at Scale," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2018, pp. 819–829. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8665771

[32] M. A. Abu, N. H. Indra, A. Abd Rahman, N. Sapiee, and I. Ahmad, "A study on Image Classification based on Deep Learning and Tensorflow," vol. 12, pp. 563–569, Apr. 2019.

[33] S. Ravichandiran, *Getting Started with Google BERT: Build and train state-of-the-art natural language processing models using BERT*. Packt Publishing Ltd, Jan. 2021, google-Books-ID: CvsWEAAAQBAJ.

[34] W. Dong, M. Keceli, R. Vescovi, H. Li, C. Adams, E. Jennings, S. Flender, T. Uram, V. Vishwanath, N. Ferrier, N. Kasthuri, and P. Littlewood, "Scaling Distributed Training of Flood-Filling Networks on HPC Infrastructure for Brain Mapping," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, Nov. 2019, pp. 52–61. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8945106

[35] J.-C. Régin, M. Rezgui, and A. Malapert, "Embarrassingly Parallel Search," in *Principles and Practice of Constraint Programming*, C. Schulte, Ed. Berlin, Heidelberg: Springer, 2013, pp. 596–610.

[36] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, Apr. 2010, pp. 265–278. [Online]. Available: https://doi.org/10.1145/1755913.1755940

[37] M. Kunjir, B. Fain, K. Munagala, and S. Babu, "ROBUS: Fair Cache Allocation for Data-parallel Workloads," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, May 2017, pp. 219–234. [Online]. Available: https://dl.acm.org/doi/10.1145/3035918.3064018

[38] X. Zhang, K. Davis, and S. Jiang, "QoS support for end users of I/O-intensive applications using shared storage systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, Nov. 2011, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/2063384.2063408

[39] F. Tessier, V. Vishwanath, and E. Jeannot, "TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 70–80, iSSN: 2168-9253. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8048918

[40] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A Next-Generation Resource Management Framework for Large HPC Centers," in *2014 43rd International Conference on Parallel Processing Workshops*, Sep. 2014, pp. 9–17.

[41] H. Zheng and k. velusamy, "Deep Learning I/O Workloads for ALCF," Jan. 2024, original-date: 2023-08-22T18:59:25Z. [Online]. Available: https://github.com/zhenghh04/dlio_ml_workloads

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[43] P. Braam, "The Lustre Storage Architecture," *arXiv:1903.01955 [cs]*, Mar. 2019, arXiv: 1903.01955. [Online]. Available: http://arxiv.org/abs/1903.01955

[44] Kidney, "2019 Kidney Tumor Segmentation Challenge," 2019. [Online]. Available: https://kits19.grand-challenge.org/

[45] H. Devarajan, "DLIO Profiler: A multi-level dataflow tracer for capture I/O calls from worklows." Mar. 2024, original-date: 2023-03-29T03:17:54Z. [Online]. Available: https://github.com/hariharan-devarajan/dlio-profiler