HVAC: Removing I/O Bottleneck for Large-Scale Deep Learning Applications

Awais Khan[1], Arnab K. Paul[2], Christopher Zimmer[1], Sarp Oral[1], Sajal Dash[1], Scott Atchley[1], Feiyi Wang[1]
[1]*Oak Ridge National Laboratory, TN, USA,* [2]*BITS Pilani, K K Birla Goa Campus, India*
{khana, zimmercj, oralhs, dashs, atchleyes, fwang2}@ornl.gov, arnabp@goa.bits-pilani.ac.in

***Abstract*—Scientific communities are increasingly adopting deep learning (DL) models in their applications to accelerate scientific discovery processes. However, with rapid growth in the computing capabilities of HPC supercomputers, large-scale DL applications have to spend a significant portion of training time performing I/O to a parallel storage system. Previous research works have investigated optimization techniques such as prefetching and caching. Unfortunately, there exist non-trivial challenges to adopting the existing solutions on HPC supercomputers for large-scale DL training applications, which include non-performance and/or failures at extreme scale, lack of portability and generality in design, complex deployment methodology, and being limited to a specific application or dataset. To address these challenges, we propose High-Velocity AI Cache (HVAC), a distributed read-cache layer that targets and fully exploits the node-local storage or near node-local storage technology. HVAC seamlessly accelerates read I/O by aggregating node-local or near node-local storage, avoiding metadata lookups and file locking while preserving portability in the application code. We deploy and evaluate HVAC on 1,024 nodes (with over 6000 NVIDIA V100 GPUS) of the Summit supercomputer. In particular, we evaluate the scalability, efficiency, accuracy, and load distribution of HVAC compared to GPFS and XFS-on-NVMe. With four different DL applications, we observe an average 25% performance improvement atop GPFS and 9% drop against XFS-on-NVMe, which scale linearly and are considered the performance upper bound. We envision HVAC as an important caching library for upcoming HPC supercomputers such as Frontier.**

***Index Terms*—High-Performance Computing (HPC), Deep Learning, Caching and I/O Optimizations**

I. Introduction

Deep Learning (DL) is an emerging technology gaining dominance to solve critical problems and predicting trends in several domains, including computer vision [21], [20], speech recognition [59], [45], natural language processing [48], [26], scientific [43], [55], [24] and climate science [25], [29], [40]. To train an efficient deep neural network (DNN) with high accuracy, large volumes of input datasets and high-speed compute accelerators are required [43], [18], [55], [57]. Thus, training a DNN is becoming an increasingly important workload on HPC supercomputers, such as Summit [23], [28]. However, to efficiently run and scale DL applications in HPC environments to leverage state-of-the-art HPC supercomputers remains a challenge. Figure 1 shows three key factors involved in training a DNN: i) I/O provides the data and labels for training to each node, ii) computation to execute the DNN, and iii) communication, to synchronize updates across nodes. Much of the prior work have focused on runtime and algorithmic enhancements to optimize the computation and communication [18], [43]. Despite these enhancements, however, DL frameworks on HPC supercomputers still suffer from scalability limitations, particularly with respect to data I/O [37], [16], [46], [41].

On large-scale supercomputer systems, the shared parallel file system (PFS) such as GPFS [3] and Lustre [44] store the extremely large DL datasets. DL training makes use of multiple epochs, where each epoch reads the entire dataset in a random shuffled order. The HPC I/O subsystems are not built to deal with the manner DL frameworks read data at scale and thus, are easily saturated with a large number of concurrent and random accesses on small files. For instance, ImageNet-1K [6] – a popular image dataset for computer vision tasks, contains more than 1.28 million files in $1,000$ categories. Similarly, the Open Images [10] dataset contains approximately 9 million images. This I/O problem will become even more severe with forthcoming generation Exascale supercomputer, which will deliver 10^{18} Flop/s of scalable computing capability [4]. All that computing capability will be for naught if the I/O software stack cannot meet the needs of DL applications running at scale — leaving DL applications starve while waiting to read data from PFS.

Therefore, a number of performance studies tend to explore the scalability of DL applications workloads by exploiting the node-local or near-node-local storage on HPC supercomputers. Of particular interest and closely related to our work is the subject of scaling DL application training on different HPC supercomputer platforms [51], [52], [18], [55], [17], [43]. Several efforts such as, BGFS BeeOND [1], and local read-only cache (LROC) on IBM Spectrum Scale (GPFS) [3], have been done to create optimized file systems for HPC applications. These file systems are intended for both write and read access mechanisms for applications. However, for DL applications which are primarily read-intensive, these solutions incur a huge metadata overhead to support writes. Lustre persistent client caching (LPCC) [44] with read-only mode exploits node-local storage. However, the cache size and performance is limited to a single node-local NVMe. A few of the existing studies focus on building prefetching and caching solutions for large datasets to improve I/O performance of DL applications, such as [18], [55], [43], [58], [32], [53], [38], [26], [40]. We discuss the limitations of existing studies in section II-D.

Unfortunately, there exist several non-trivial challenges to adopting the above prefetching and caching solutions on HPC supercomputers for large-scale DL training applications. First, several aforementioned studies [57], [26], [55] show scaling DL applications with small-scale testbed or in simulation environments [40] and do not scale well on real large-scale HPC supercomputers, e.g., over 1000 nodes. Second, the majority of the existing studies [18], [43], [55], [32], [53] require non-trivial modifications of the entire input pipeline architecture and are intrusive to the DL application code, which is contrary to the goal of this study, i.e., portability. Third, choices like Memcached [9] and LMDB [34] lack support for POSIX interface and are not suitable choice for HPC supercomputers. Fourth, a few of the existing works [38], [53], [18] require managing additional metadata backend storage, which again acts as bottleneck in case of large-scale small file I/Os. Last but not least, a fraction of previous works have been tailored to meet a particular application, dataset, and/or architectural need and lack generality in their design.

Therefore, to address the aforementioned challenges, we propose High-Velocity AI Cache (HVAC), a transparent read-only caching layer, for large-scale HPC supercomputers to improve DL applications training performance when using large datasets (i.e., terabytes to petabytes). The key design idea behind HVAC is to scale to thousands of compute nodes on HPC supercomputer such as Summit and Frontier, and to remove I/O bottleneck for large-scale DL applications without the need to modify application or file systems, and free of additional metadata bookkeeping overhead.

Our key contributions in this paper are:

- We present the design and implementation of HVAC, a scalable lightweight read-only cache system for HPC supercomputers with node-local storage on compute nodes, and near node-local storage, to cache large datasets from shared parallel file systems.
- We use distributed hashing to determine the cached location for data requests without compromising training accuracy and avoid the metadata bottleneck with the goal of significantly improving the random read performance compared to traditional parallel file systems.
- We provide portability and support for standard POSIX open, read, and close file operations by intercepting I/Os via LD_PRELOAD. So, that DL applications or underlying parallel file systems do not require modifications to adopt HVAC.
- We present a series of extensive experiments ran on Summit supercomputer using four different DL applications and models in order to study the scalability, training performance and accuracy for the proposed approach. We compare multiple HVAC variants against large-scale high-bandwidth GPFS and XFS-on-NVMe. The experimental results show that HVAC can fully scale to 1,024 nodes on Summit with an average 25% reduction in training time compared to GPFS. Whereas, HVAC shows only 9% performance overhead compared to upper I/O bound, i.e., XFS-on-NVMe.

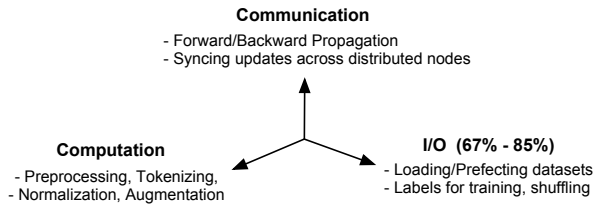


Fig. 1: Distributed DL comprises of the three components. Note that, DL applications running at large-scale training environments spend 67-85% of their execution time performing I/O to a PFS as reported in several recent works [37], [16], [55], [42].

II. Background and Motivation

A. Distributed Deep Learning

Deep learning on a single node has two steps: forward computation and backward propagation. A data parallel approach to distributed DL involves replicating the DL model and distributing the training dataset on all the nodes. The I/O in a data parallel approach is complex and typically consists of multiple stages as shown in Figure 1. First, I/O, the most expensive step in distributed DL training [37], [16], [18], is responsible for providing data and labels to each training epoch and iteration [18], [56], [43]. Second, computation which involves dataset preprocessing in some of the image-based models, such as augmentation operations including padding, scaling, rotations, resizing, and distortion [55], [56]. Third, communication which refers to transferring newly calculated model gradients to all the involved GPUs for iterative model updates, collations, forward/backward propagation, and syncing updates across the distributed nodes [55], [56]. This data parallel approach is most favored for scaling DL on thousands of nodes, such as on a large scale HPC supercomputers [52]. Notably, backward propagation for distributed DL training firstly computes the gradients on each node, then runs an MPI all-reduce operation so that average gradients for each weight are distributed to all nodes, and finally updates the weights in each nodes. The gradient distribution takes place after each epoch.

B. Deep Learning: Access Patterns and I/O Characteristics

Next, we discuss common access patterns and I/O characteristics of DL applications. For simplicity, we show the data access pattern of DL training application with a small dataset containing 16 files, as shown in Figure 2. Before starting DL training, a batch size is chosen, which signifies the number of files that will be read together by a compute unit, such as a GPU. Therefore reading the total dataset of 16 files in batches divided into four iterations. Note that, an epoch constitutes that all files in the training dataset are read once. The model convergence can take hundreds to thousands of epochs. Before each epoch, the DL training framework shuffles all the files in the training dataset to generate a random read order. This randomization is necessary to attain high prediction accuracy and avoids overfitting in the trained model.

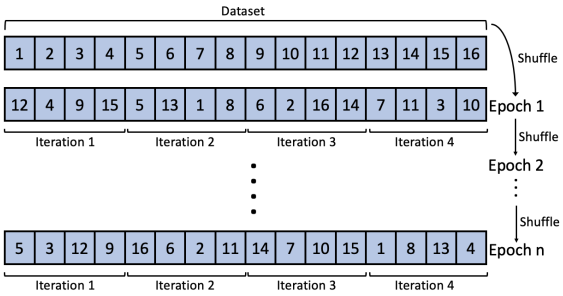


Fig. 2: The data access pattern of a training dataset of 16 files. The numbers represents file IDs. The batch size is four. Files are shuffled and accessed randomly after each epoch.

The DL training step for a dataset therefore exhibits the following file-access characteristics.

- A DL training task only works on one dataset, and performs **read-only access to the dataset** in one epoch.
- All files in a dataset are **read once in an epoch, and will be traversed again** in subsequent epochs.
- Between epochs, files are shuffled in different random orders to avoid model overfitting. However, the **random order of file accesses can be varied** as it does not affect the model accuracy.

The data access patterns in a DL training task therefore provides the following I/O characteristics.

- Within a DL training job, there is a **high degree of shareability in I/O**. As each DL training task makes multiple passes (epochs) over the same input training data, there is a clear benefit to caching the data for use in subsequent epochs.
- Shareability of I/O makes DL jobs cache friendly. However, the random access pattern in each epoch makes it **cache-adverserial if the data does not fit in cache**.
- The exact sequence of random order of data does not affect the accuracy in an epoch. Therefore, **I/O is substitutable** which makes the partially cached dataset equally cache-friendly.

C. MDTest on Summit Supercomputer

To motivate the need of HVAC, we run MDTEST benchmark on Summit supercomputer. The specifications of the Summit supercomputer is described in Section IV-A1. MDTest [8] is an MPI-based application to test and evaluate metadata performance of PFS. DL jobs can train on a dataset that can have either small files or big files, both raise two different kinds of PFS performance issues. Note that, both file type I/Os follow a transaction comprising of `<open-read-close>` operations. Therefore, we use 32KB to demonstrate small files and 8MB to mimic large files. The gap in performance between the parallel file system (GPFS) and XFS-on-NVMe, i.e., node-local storage is shown in Figures 3 and 4.

The results of the 32KB read measurements in Figure 3 show that at smaller file sizes the PFS metadata performance

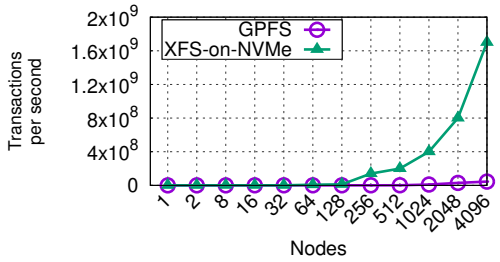


Fig. 3: Transactions per second for 32KB random file open-read-close operations on Summit comparing GPFS and XFS-on-NVMe.

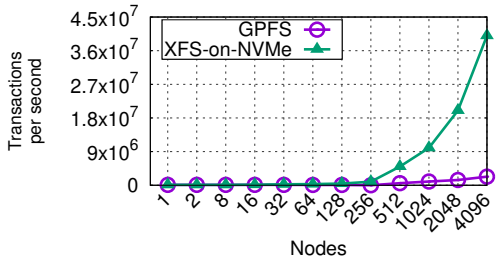


Fig. 4: Transactions per second for 8MB random file open-read-close operations on Summit comparing GPFS and XFS-on-NVMe.

is an impediment to large-scale DL training jobs. In contrast, at 8MB file accesses in Figure 4 the problem reduces the burden on metadata and shifts to bandwidth constraints. The aggregate read performance of the node-local storage, i.e., NVMe SSDs at 4,096 nodes in Summit is 22.5 TB/s compared with 2.5 TB/s of Summit's GPFS.

When a DL training job runs on a large-scale HPC supercomputer, the model is trained using hundreds or more GPUs, each requesting small batches of data creating potentially millions of requests to the PFS. This access pattern is very challenging for PFS. When a process opens a file on the PFS, a request is made to a metadata server to verify the global state of the file and the locations of the objects holding the data for the file. On behalf of the requester, the metadata server requests a lock or token to access the data to ensure it is not in a modified state. The read request then generates new requests to the series of servers holding the stripes of data needed for the request. Upon receipt of the data, the application closes the file and begins to process the data. Reading each file from the PFS creates a cascade of operations. On leadership class storage supercomputers, there are tens of metadata servers and a few hundreds of data servers. This infrastructure works well when file metadata operations are low and access requests are large, but when millions or hundreds of millions of file open requests are required in a short amount of time, such as for a DL training job, the low count of metadata resources struggles to operate on the request in a timely manner.

D. Limitations of Existing Systems

Optimizations in PFS: BGFS BeeOND [1] is used to aggregate the performance and capacity of internal SSDs in compute nodes for the duration of a compute job to

provide a transient namespace. Lustre persistent client caching (LPCC) [44] provides a read-write cache on the local SSD of a single client; and a read-only cache over the SSDs of multiple clients. Solutions such as LROC/ILM on IBM Spectrum Scale (GPFS) [3] or UnifyCR [13] enable applications to use node-local storage as burst-buffers for caching writes. These optimizations are intended to create full-fledged write and read access mechanisms for applications. HVAC differs from these strategies by not explicitly supporting writes, which eliminates a significant overhead in the ways of metadata checks, locking, and data search.

Distributed Caching for DL Jobs:

- FanStore [58] places metadata and file data in RAM and local disks, respectively. One or more worker threads within each FanStore process handle file system requests intercepted from the DL training process. However, FanStore works on an internal hash map technique which limits resiliency on large-scale HPC supercomputers.
- Quiver [32] is designed for a shared GPU cluster where Kubernetes manages scheduling of DL jobs on a specific virtual machine. However, Quiver needs data preprocessing, works only on static data and needs a Quiver specific API, which makes it not transparent and usable for large-scale HPC DL applications.
- DIESEL [53] is data-aware storage and caching co-designed supercomputer that supports both data writing and data reading for DL jobs. DIESEL however requires restructuring of the dataset, additional storage servers, and provides a FUSE interface that routes I/O operations from inside the Linux kernel back out to a user process which incurs a performance penalty. Also, each client caches the metadata eliminating the need for metadata servers, but the metadata can consume significant memory if there are a large number of files.
- NoPFS [18] solves the I/O challenges of loading large-scale DL datasets from PFS on HPC supercomputers and proposed a near optimal prefetching and caching by analyzing DL application access patterns. It highly depends on the regression, access patterns and history data. Thus, accuracy of prefetching can be compromised if DL application shows different access patterns. Also, it requires an additional metadata store to track access patterns and samples. Further, it requires some changes in data loader of DL application to use NoPFS.

In-Memory Stores: The choices like Memcached [9] and LMDB [34] provide scalable I/O for datasets and are used to improve DL workflows. Memcached is an in-memory, key-value store that uses a distributed hash table to store keys up to 250 bytes and value objects up to 1 MB, and places the burden on the application to manage mapping larger files onto multiple key-value pairs. LMDB is memory-mapped DB and relies on mmap to perform in-memory data access. Both Memcached and LMDB have non-POSIX interface that require application-level modifications. Atop, mmap interface is not suitable to complex DL applications access patterns, such as strided access of batches of data [43].

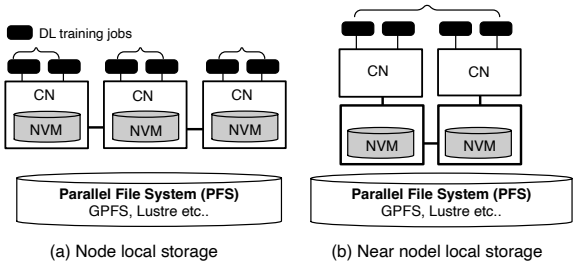


Fig. 5: The target architecture for the proposed system. (a) depicts Summit's node-local storage on compute nodes, whereas (b) reflects the near node-local storage.

Although, the current work shares some basic design considerations with the existing works. However, most of the existing studies are limited to various perspectives. For instance, running DL applications at a large scale has not been studied well enough, e.g., 1024 nodes. Other limitations include non-trivial modifications of the entire input pipeline and metadata bottlenecks. Thus, we design HVAC to be a highly scalable software library for large-scale HPC supercomputers. Our key contributions lie in the real deployment of HVAC library on the Summit supercomputer at 1,024 nodes scale, without requiring any modifications to the application and without the need for any additional metadata storage backends.

III. HVAC: Design and Implementation

In this section, we present our key design goals and target architecture followed by design and implementation of the proposed caching system.

A. Design Goals

Our key design goals include:

- **Scalability**: The key design goal is to enable DL application scalability on large-scale HPC supercomputers. The training cost of DNN-based applications is very high for two reasons. First, DNNs are getting more complicated due to increased depth and parameters. Second, the training samples are getting much larger, i.e., terabytes of training data, which wields significant pressure on IO subsystems as the entire data is re-loaded in random order on every iteration to achieve higher accuracy and convergence. Therefore, in this regard, HVAC must scale to hundreds of application threads, and petabytes of data.
- **Portability**: There exist several existing studies to improvise the training time of DL applications. However, those studies require non-trivial modifications to either input pipeline to the application code or underlying PFS. Our goal is simple yet effective: keeping HVAC lightweight, generic and portable, requiring no modifications to complex scientific DL applications or underlying file system. Further, HVAC can seamlessly benefit with any read optimizations applied to underlying PFS.

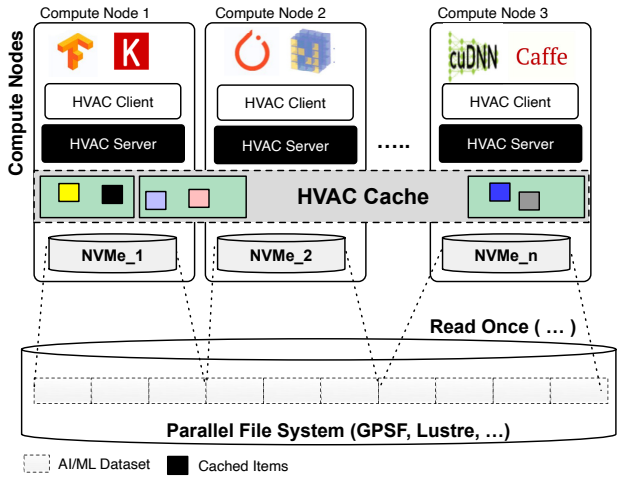


Fig. 6: An overview of HVAC architecture for node-local storage on compute nodes of Summit supercomputer.

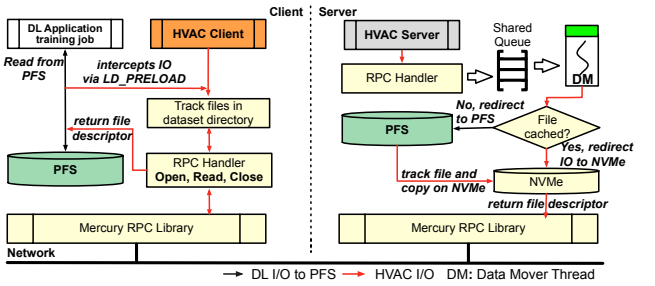


Fig. 7: HVAC I/O flow on compute node. Note that, ML/DL application is unmodified. HVAC client and server instances can be co-located on same node or different nodes based on the underlying target architecture. Further, multiple HVAC server instances can be executed on a single node.

- **No Metadata Bottleneck**: Another key bottleneck in large-scale PFS such as GPFS and Lustre is often attributed to metadata I/Os. Therefore, we aim to eliminate such metadata bottlenecks by employing a robust metadata-less server architecture and enable algorithmic location determination from the application to improve performance.

B. Target Architecture

In recent years, it has become common to deploy large-scale systems with additional storage, often flash-based as shown in Figure 5, for handling high-bandwidth workloads such as check-pointing [31]. These storage systems have a variety of deployment models such as stand-alone racks in the case of DataWarp, compute node-local storage as in Summit and Sierra, and rack-local storage such as the upcoming "Rabbit" architecture in El Capitan [7]. Therefore, HVAC caching system is designed to be robust and easily deployable on large-scale HPC supercomputers with different deployment models.

C. System Overview

HVAC is a simple, lightweight and transparent client-server library intended to accelerate I/O accesses for DL applications that utilize read-only data with a high re-read rate. An architectural overview of HVAC caching system is shown in Figure 6. The HVAC consists of two main components: HVAC client library and HVAC server process. When a training job is allocated over a set of compute nodes on HPC supercomputer such as Summit, an instance of HVAC server is constructed on the fly across these compute nodes, using the node-local storage, or other fast storage devices. For instance, on Summit supercomputer, the `alloc_flags` *"hvac"* option in job submission script initiates both the NVMe device on the compute nodes in the allocation and spawns the HVAC server process. At startup the server process establishes the Mercury RPC and bulk communication components and generates a directory of known server processes. On the node-local storage, the HVAC server is designed to have the same temporary life cycle as a batch-submitted job, while it uses node-local fast storage to improve application read performance.

When HVAC server instances are constructed, every HVAC server instance spawns a dedicated data-mover thread, which manages a shared `FIFO` queue to track and manage the forwarded file I/O operations from HVAC client. The HVAC server daemon operates independently and does not communicate with other server processes on remote compute nodes, therefore being effectively unaware of each other. An HVAC server daemon's purpose is to process forwarded file system operations from HVAC clients and to retrieve data from PFS and cache it to node-local storage. The server internally builds an aggregate cache layer atop node-local fast storage to accelerate the DL application read performance as shown in Figure 6. On the other hand, the HVAC client consists of an interception interface that catches relevant file system calls to PFS and redirects to the respective HVAC server. The environment variable `HVAC_DATASET_DIR` points to the parent directory of the dataset to be cached within the allocation.

HVAC uses the Mercury communication library [49] for supporting RPC and bulk data transfers on Summit over the Infiniband network and in future will also support Slingshot 11 on the Frontier supercomputer [5].

D. I/O Flow

HVAC works as a traditional cache, where files are first attempted to be accessed from the cache, and only upon a cache miss the file is retrieved from the PFS. This step also aids in overcoming the challenges of having to pre-copy data to the node-local storage when sharding [55], [56] and prefetching [18] techniques are used to avoid the PFS. We consider prefetching an important addition to HVAC as our future work. As shown in Figure 7, the large-scale DL applications running on compute nodes make use of distributed DL frameworks, such as, Caffe [2], PyTorch [11], and TensorFlow [12]. These DL applications access the read-only datasets stored on PFS such as GPFS and Lustre. Any DL application that uses HVAC

must first preload the client interposition library that intercepts file system operations such as `open, read, and close`. This is a commonly observed pattern in DL training. Note that, the file system calls are intercepted via `LD_PRELOAD` and do not require any modifications to existing DL applications or underlying parallel file system.

We demonstrate I/O flow in HVAC with two file read scenarios.

First Read: ❶ A DL application initiates a read request to DL dataset directory on PFS as shown in Figure 7. ❷ HVAC client intercepts any incoming file I/O `<open, read and close>` and starts tracking in the dataset directory. ❸ The RPC handler of HVAC client redirects the requested file I/O to respective HVAC server whether local or remote to HVAC client. This redirection is done via hashing, which we describe in next subsection III-E. HVAC client and server internally manage RPC handlers, responsible to send and receive the messages over the network. ❹ When HVAC server receives a request, the RPC handler en-queues the forwarded file I/O to share FIFO queue. ❺ The data-mover thread checks if the file is already cached on node-local storage. As, this is usecase for the first read of a file, ❻ the data-mover thread tracks and copies the file to node-local storage via `fs::copy(src, dst)` method from PFS. ❼ The returned file descriptor or stream is used to track the read offset and length, and used in RPCs to the servers for initiation of the bulk transfers. ❽ Close calls are then intercepted and out of band RPCs are generated to the owning server process to signal the tear down of the file being accessed. Note that, it is highly likely that a single file can be requested by multiple HVAC clients simultaneously, therefore we use mutex lock on shared queue to guarantee consistency and to avoid repeated copying of file to node-local storage.

Future Cached Reads: On the contrary to the first read, when a read I/O is received for an already cached file on the HVAC server as shown in Figure 7, ❻ the data mover thread redirects the I/O to read file from node-local storage bypassing PFS and ❼ returns the file descriptor or stream to the corresponding HVAC client, which then provides the file descriptor to the DL application. The step ❽ is performed same as above.

Note that, the life cycle of the dataset in the cache is coupled with the life cycle of the job on HPC supercomputer. When the job is finished, the cached datasets is purged from the node-local storage.

E. Hash-based I/O Redirection

HVAC client aggressively uses hashing to distribute I/Os internally and locate the contents cached at HVAC servers. This obviates the need for a metadata store or in-memory database to store cached file metadata. In HVAC, file cache locations are determined using the file path and job node allocation. This combination of information is first used to algorithmically determine the location of the cache within the allocation that "homes" the file. The client then contacts the resident server process to retrieve the data. Note that, each file is managed and cached by one HVAC server. This hashing technique is common in file systems such as GekkoFS and CephFS [47]. Though CephFS uses CRUSH which is an intelligent pseudo-random object placement algorithm. This hash-based server determination not only relieves us from i) complicated location metadata management for datasets with billions of files, and ii) broadcasting requests to find file to hundreds of servers but also aids in balancing load-across the compute nodes. In current work, HVAC caches data at file granularity. However, to ensure an even load-distribution among HVAC servers for datasets with highly skewed file sizes, segment-level caching [17] can be implemented. HFetch [17] proposed segment-level caching to accelerate scientific workflows, which requires managing additional file-to-segment layout mappings.

F. Intercepting Read I/Os

For the initial prototype, HVAC is used to profile the read calls from the DL frameworks like PyTorch and Horovod, to understand how the data loaders within the frameworks access the files. HVAC is built using an `LD_PRELOAD` mechanism for intercepting I/O related function calls. The `LD_PRELOAD` mechanism avoids the necessity of forcing applications to modify their code bases to support HVAC. This is important for large-scale HPC supercomputer DL application runs, where having to modify large applications to use HVAC presents a huge burden on the application developers. With the DL data loading frameworks, traditionally written in python, it was initially unclear if this would be a viable strategy. Our initial tests on ResNet50 using PyTorch and Horovod show that the framework use POSIX `<open, read, and close>` calls to access the underlying file system. The profile from the DL application - ResNet50 show the pattern of open, a single 16MB read, and a close per file and files were all read in prior to each iteration of the framework. This pattern is successfully intercepted by HVAC.

G. Cache Eviction and Replacement

The DL training job reads a dataset repeatedly, so caching the dataset in the compute node-local storage increases the read performance significantly. HVAC partitions the complete dataset among the compute nodes. However, in cases where dataset is larger than the aggregate capacity of all node-local storages of compute nodes, cache eviction and replacement can be performed. Currently, HVAC is designed to perform eviction and replacement randomly and various cache-eviction and replacement policies can be considered. However, HVAC is designed to run at scale, i.e., 1,024 compute nodes. Therefore, in real scientific usecases, we do not observe any dataset which can outgrow the aggregate capacity of node-local storages of the compute nodes.

H. Future Work

The initial prototype for HVAC simplifies the design space to enable focus on the important underlying primitives of the library. However, there are several underlying design

choices that need to be investigated to improve performance, scalability, and reliability. The current implementation of the home location calculation assumes that a file is only resident in a single location within the allocation. This can lead to one-to-many lookup issues that may imbalance the access and more importantly, if the node-local NVMe fails, lead to a failed training run. With over 7PB of node-local NVMe storage on Summit, for many datasets it is reasonable to enable data replication within the allocation. This would allow for local grouping within the allocation to distribute accesses and improve performance and enable the calculation of fail-over locations in the case of a failed HVAC server.

IV. Evaluation

In this section, we investigate and evaluate the scalability of the proposed HVAC caching system.

A. Experimental Setup

1) Testbed: We used Summit supercomputer for large-scale evaluation of HVAC. Summit [39] is based upon IBM AC922 system and deployed at the Oak Ridge Leadership Computing Facility (OLCF). It consists of 4,608 compute nodes. The description of compute node is shown in Table I. Summit is connected to Alpine, a 250 PB IBM Spectrum Scale (GPFS) file system. Summit can access Alpine at 2.5 TB/s in aggregate under a large, sequential write I/O access pattern. Alpine is a center-wide file system and is directly accessed by all other OLCF resources.

Attribute	Description
Supercomputer	Summit
CPU	2 x IBM POWER9 22Cores 3.07GHz
GPU	6 x NVIDIA Tesla Volta (V100)
Memory Capacity	512 GB DDR4
Node-local Storage	1.6 TB Samsung NVMe SSD with XFS
Network Interconnect Family	Dual-rail Mellanox EDR Infiniband

TABLE I: The compute node specification of Summit.

2) Benchmark and Applications: We use a mix of deep neural networks and scientific deep learning applications from MLPerf-HPC benchmark [19] to evaluate the scalability of the proposed caching framework.

- **DNNs:** We use two DNNs for our experiments namely, ResNet50 and TResNet_M. We used ImageNet21K dataset to train the models with PyTorch and Horovod for distributed training. ResNet50 is a large network with 228 layers and 25.6M parameters.
- **CosmoFlow [36]:** is a highly scalable deep learning application from MLPerf HPC v0.5 benchmark suite [19]. It involves training a 3D convolutional neural network on N-body cosmology simulation data to predict physical parameters of the universe. We use cosmoUniverse dataset to train CosmoFlow application. It uses cosmoflow model and includes more than 51K parameters.
- **DeepCAM [33]:** is a PyTorch implementation for the climate segmentation benchmark, based on the Exascale Deep Learning for Climate Analytics. DeepCAM is also part of MLPerf HPC v0.5 benchmark suite and was awarded the ACM Gordon Bell Prize in 2018 [33], [19]. DeepCAM is trained on images with 768 × 1152 pixels and 16 channels, which is substantially larger than standard vision datasets like ImageNet, where the average image is 469 × 387 pixels with at most 3 or 4 channels.

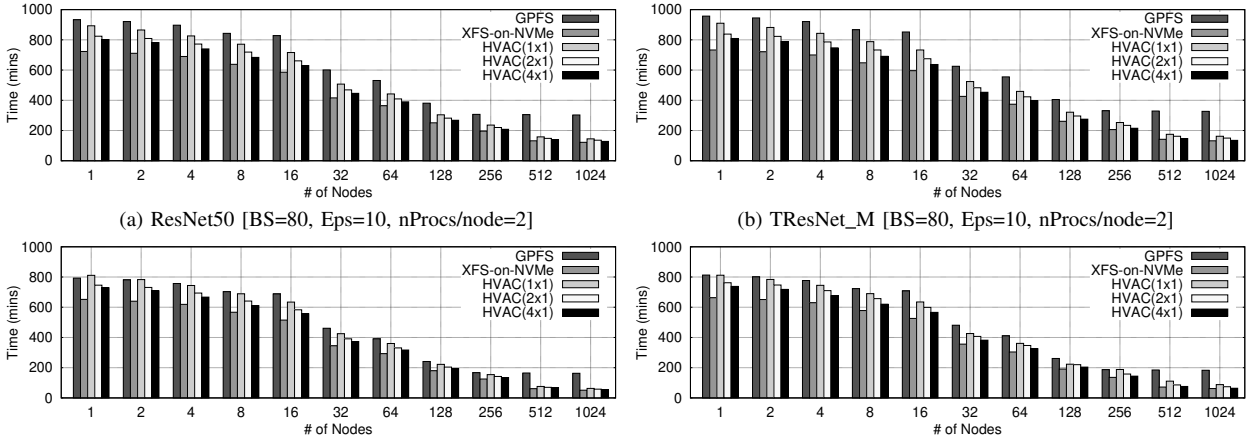
3) Datasets: We use two different large-scale datasets for our experiments. The first dataset we used is the ImageNet21K [6]. This dataset contains 11,221 classes, where the training set has 11,797,632 data points and the test set has 561,052 data points. The average sample size in dataset is approximately 163KB (total dataset is 1.1TB). The second dataset is cosmoUniverse containing preprocessed TFRecord files generated from simulations runs by the ExaLearn group at NERSC [36]. There are 524,288 samples for training and 65,536 samples for validation (total dataset is 1.3TB). All the datasets are stored on IBM Spectrum Scale GPFS. Note that, all the datasets we used are large enough to introduce significant I/O challenges in scale-out environments, which expose GPFS performance limitations. All experiments were repeated three times, unless otherwise noted, and we report an average with a 95% confidence interval.

We compare our approach with the following systems:

- **GPFS:** is the large-scale IBM Spectrum Scale shared PFS hosting the complete dataset. Each epoch and iteration touches the file system to read the dataset.
- **XFS-on-NVMe:** An ideally optimized scenario, where the complete dataset is staged to compute node-local NVMes formatted with XFS file system prior to application run and no GPFS is involved. This is upper I/O bound for all of our experiments.
- **HVAC (i x 1):** The proposed caching system with ***i*** instance(s) running on each compute node. The dataset is read from GPFS only in the first epoch and for rest of training epochs, the dataset is accessed from HVAC cache. We run multiple HVAC servers on a single compute node to show its flexibility, portability and ease of deployment. Further, such multiple instances also demonstrate a multi-threaded version of HVAC and to show performance improvement.

B. Effect of Scaling the Number of Compute Nodes

First, we show the effect of scaling the number of compute nodes on training performance in terms of training time (Figure 8 (a) - (d)). We compare the training performance of each of the four DL applications and scale the training job from a single node to 1,024 compute nodes of Summit and measure the training time of two concurrently running DL training jobs per node using the entirety of training dataset for each application. For Figure 8(a) and (b), we observe a notable reduction in training time (averaging 20%) by HVAC compared to GPFS with small number of nodes, i.e., 32 nodes. However, when we scale the number of nodes incrementally from 32 to 1,024, the high-bandwidth GPFS is insufficient as the bottleneck does not lie in bandwidth, but instead small metadata I/Os becomes a bottleneck for huge number of small files. Additionally, we also see that in Figure 8(a) and (b),



(a) ResNet50 [BS=80, Eps=10, nProcs/node=2]

(b) TResNet_M [BS=80, Eps=10, nProcs/node=2]

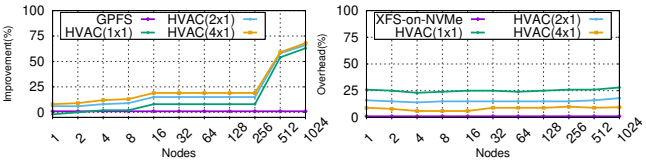
(c) CosmoFlow [BS=80, Eps=10, nProcs/node=2]

(d) DeepCAM [BS=80, Eps=10, nProcs/node=2]

Fig. 8: Among the four DNN models explored in our study, we see that HVAC outperforms the GPFS with scaling the number of nodes. BS, Eps, and nProcs/node denote batch size, epochs and number of application processes per node. Y-axis shows training time in minutes.

GPFS training time is much higher than (c) and (d). As, the dataset used for cosmoflow and deepcam contains larger file sizes compared to (a) and (b), thus stressing the metadata servers on GPFS. Surprisingly, we notice that, at 1,024 nodes, GPFS performance saturates and show a little higher training time compared to the peak performance of GPFS gained at 450 nodes. On the contrary, all HVAC variant results imply that the proposed caching scales fairly-well and provide consistent improvement in training time, which is critical to DL application performance. While, all HVAC variants show large performance improvement compared to GPFS at scale, and for larger datasets, there is still a gap between its performance and XFS-on-NVMe performance. All the results in Figure 8 show that HVAC fails to meet the XFS-on-NVMe training time. With HVAC(4x1), we see an average of 9% lower performance compared to upper I/O bound, i.e., XFS-on-NVMe,. This lower performance against XFS-on-NVMe is attributed to several factors, i) implementation overhead incurred by HVAC for cache loading/allocations, ii) accessing files from remote compute nodes, and iii) performance drop in first epoch, for copying files from GPFS to node-local storage.

Overhead Analysis: To clearly demonstrate performance improvement in training time against GPFS, we present the performance improvement normalized with GPFS in Figure 9(a). Notably, for aggregate training time of 10 epochs, the improvement factors lies around 7-25% till we reach 256 nodes. However, there is high performance gain, when we scale to 512 and 1,024 nodes, the average performance improvement is over 50% for all HVAC variants. This shows HVAC's strong scaling property. Next, we show HVAC training time overhead normalized with XFS-on-NVMe in Figure 9(b). We observe that HVAC(1x1) shows higher overhead averaging around 25% compared to other HVAC variants, i.e., HVAC(2x1) 14% and HVAC(4x1) with 9%. We see that this performance overhead is consistent as we scale the number of nodes. Therefore, we attribute it to HVAC's implementation overhead.



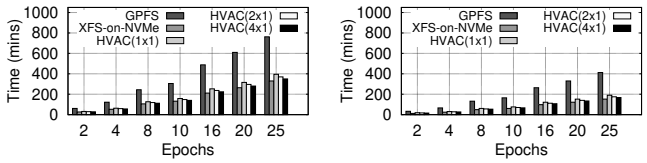
(a) Normalized with GPFS. (b) Normalized with NVMe-SSD.

Fig. 9: HVAC performance gain and drop normalized with GPFS and node-local NVMe-SSDs.

C. *Effect of Scaling the Number of Epochs*

Next, we study the effect of varying number of epochs on training performance in terms of training time (Figure 10 (a) and (b)). We perform experiments with ResNet50 and CosmoFlow DL applications on 512 compute nodes. Note that, with smaller epochs at 2 and 4, the performance of all variants is comparable and not much training time overhead has been noted. However, the DL application model fails to converge at such small number of epochs. Therefore, epochs plays a key role in DL application models. For both models, we scale the epochs linearly and see that the results confirm our findings from Figure 8. As expected, HVAC shows a strong scaling efficiency with increasing number of epochs.

Per Epoch Analysis: It is critical to show and analyze the per epoch analysis of GPFS, HVAC variants and XFS-on-NVMe. Figure 11 shows different epoch experimental results. The epoch-1 refers to the first training epoch, whereas, R_epoch denotes best random epoch (excluding the first epoch) and avg_epoch shows average epoch time contributed in training by each variant. As expected, all HVAC variants compared to GPFS show nearly same training time for the epoch-1. It is because in the first epoch, every HVAC servers reads the file from GPFS and then caches it. In HVAC, other nodes can only benefit from cache, if any of the nodes in the job has previously cached the file. Therefore, in the first epoch, nearly each server touches GPFS. However, subsequent



(a) ResNet50 [BS=80, nNodes=512] (b) CosmoFlow [BS=4, nNodes=512]

Fig. 10: Effect of increasing Epochs on training time. nNodes denote number of compute nodes.

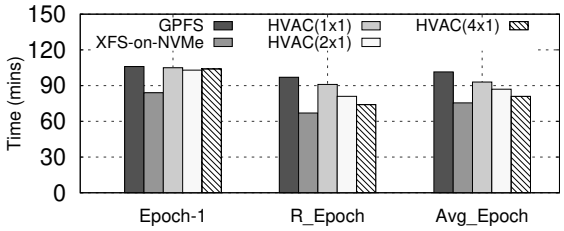
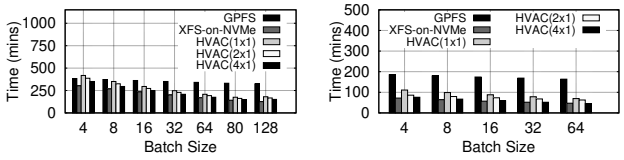


Fig. 11: Training performance with different epochs. Epoch 1, R and Avg refer to first training epoch, random epoch (excluding first epoch), and average epoch training time. [BS=4, Eps=10, nNodes=512]

random epoch shows performance improvement compared to GPFS and we see that, once the dataset is fully cached, the training time has reduced to a factor 3x per epoch by HVAC(4x1) compared to GPFS. This is highly desirable for DL models, when number of epochs grow beyond 1000 to achieve 99% model accuracy. Further, with average epoch, we see a consistent and similar trend in all HVAC variants and XFS-on-NVMe as observed in Figure 8. Our future work will investigate utilizing prefetching techniques to pre-populate the HVAC cache and reduce the performance overhead of epoch-1.

D. Effect of Increasing the Batch Size

In DL training applications, batch size is another key factor to effect training performance. Therefore, we show impact of batch size on training time (Figure 12 (a) and (b)). For this experiment, we use TResnet_M and DeepCAM DL application. For GPFS, we see that the training time of TResnet_M over the course of 80 epochs with increasing batch size from 4 to 128, there is slight improvement in performance of about 2-4% in training time. This performance improvement is derived from loading dataset in bigger batches, thus reducing round-trips to GPFS. However, we observe the similar trend by other approaches, i.e., HVAC variants and XFS-on-NVMe. Thus, we conclude that larger batch size may not yield substantial impact on DL training performed on GPFS, HVAC, and XFS-on-NVMe as commonly expected in previous results. We claim that, with all the previous experimental results, HVAC is highly scalable, robust, and adaptable. Any optimizations applied to GPFS can be inherently seen and applied to HVAC without any modifications to application or HVAC system.



(a) TResNet_M [Eps=80, nNodes=512] (b) DeepCAM [Eps=10, nNodes=512]

Fig. 12: Impact of increasing batch size. nNodes denote number of compute nodes.

E. Effect of Cache Size on Training Time

HVAC is a caching system and cache size can highly impact the performance. Therefore, we also study the impact of different cache sizes on HVAC performance and to spotlight the implementation overhead for accessing dataset from remote compute nodes. Figure 13 show the training time with respect to different cache sizes. For this experiment, we manually control the datasets resident on local and remote compute nodes. We treat the compute node hosting the training job as local and all other compute nodes as remote. We observe a negligible performance difference with multiple cache size configurations. HVAC internally uses high-performance Mercury library for remote communications and bulk data transfers over the Infiniband network. Therefore, the remote node communication overhead does not have high impact on HVAC performance at scale.

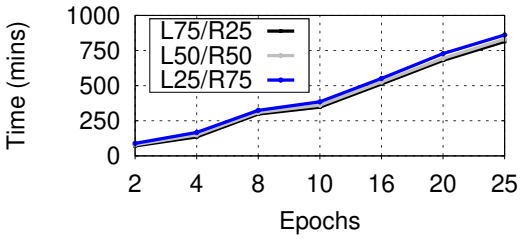
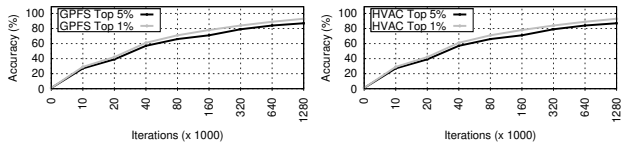


Fig. 13: Impact of cache size on HVAC(1x1). L% and R% denotes percentage of dataset cached on local and remote nodes. [BS=80, nNodes=512]

F. Training to Accuracy with HVAC

The most critical outcome of a DL training is the model training accuracy. To demonstrate this, we train ResNet50 on ImageNet-21K dataset to validate accuracy of GPFS and HVAC and show accuracy comparison in Figure 14. We observe that HVAC does not change the shuffling and randomness of DL training I/O at any time during training. The file lookup via hashing in HVAC does not affect the learning algorithm to compromise the randomness and avoids model over-fitting. Figure 14 shows that both GPFS and HVAC achieve Top 5% and Top 1% accuracy at nearly same iterations during model training. Moreover, in other words, we can say that HVAC achieves top 1% and 5% accuracy much earlier than GPFS due to its lesser training time. The existing sharding techniques tend to impact the accuracy, if the dataset is not fully shard on node-local storage.



(a) GPFS (b) HVAC (1x1)

Fig. 14: Accuracy analysis for ResNet50 model. We observe that HVAC does not compromise randomness of SGD algorithm by introducing hashing-based file lookups. Note that, we observe same accuracy for XFS-on-NVMe but due to space limitation, we do not show it.

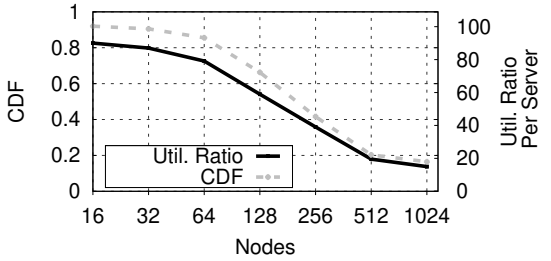


Fig. 15: Per server file distribution ratio with scaling number of compute nodes. The CDF on y-axis denotes Cumulative distribution function and presents the ideal distribution of files across the nodes. HVAC shows relatively balanced distribution across the compute nodes.

G. Fairshare and Load Distribution Analysis

Figure 15 shows the utilization ratio of each server with scaling number of compute nodes in training job. We define utilization ratio as distribution of files across the compute nodes. We observe from Figure 15 that HVAC shows fairly well-balanced file distribution across the compute nodes for ImageNet21K datasets. However, HVAC shows a little deviation from CDF when number of compute nodes are less than 128. This deviation is attributed to random sizes of file in the datasets. Note that, in our future work, topology and failover will also be considered when calculating the location of a given file and balanced distribution of files across the subset of HVAC servers will be an important metric.

V. Related Work

DL techniques are becoming more popular in HPC for solving problems in human health, high-energy physics, material discovery and other scientific areas [50], [15], [16], [54], [52]. There is a spurt of recent works on optimizing DL applications, which we broadly classify into three categories.

Dataset Prefetching and Caching: FanStore [58], DIESEL [53], Hoard [42], and Quiver [32] proposed a caching middle-layer to accelerate DL application performance. Other caching and pipelining techniques include DeepIO [59], GPipe [27], CoorDL [38]. In DeepIO [59], data servers store subsets of the training dataset in an in-memory cache and prioritize reuse of data from the in-memory cache. While this reduces the accesses to the storage system, the mechanism can change the mini-batch sequences and impact the model accuracy. MinIO in CoorDL [38] avoids replacement of cached items between epochs, without affecting the random ordering, so that at least some fraction of data for an epoch is always accessible from the cache. GPipe [27] and Mini-epoch Training (MET) [55] are focused on overlapping communication and data loading to minimize the training time via mini-batching approach. NoPFS [18] the most recent study discussing and solving the I/O challenges of loading large-scale datasets from PFS and proposed a near optimal prefetching and caching approach by analyzing ML/DL application access patterns.

PFS Optimizations: PFS is one of the most essential building blocks of the persistent storage stack in large-scale HPC environments [16], [30]. When conducting training on an HPC cluster, PFS is commonly used for storing the large volume of datasets. The DL frameworks, such as TensorFlow [14], Caffe [2], and Pytorch [11], invoke highly random small read requests to PFS and form mini-batches of the dataset required for successful training. Therefore, many research attempts have been made for I/O performance analysis of different PFSs for their capabilities to handle the workloads posed by DL applications [16], [35], [43], [46], [22]. BGFS BeeOND [1] is used to aggregate the performance and capacity of internal SSDs in compute nodes for the duration of a compute job to provide an elegant way of burst buffering. Lustre persistent client caching (LPCC) [44] provides a read-write cache on the local SSD of a single client; and a read-only cache over the SSDs of multiple clients. The limitations of LPCC with in-node storage cache is limited to the size of a single NVMe and the performance of the single NVMe. HVAC can distribute a dataset much larger than a single NVMe and performance from a client perspective is distributed across multiple NVMes. Other solutions such as LROC/ILM on IBM Spectrum Scale (GPFS) [3] or UnifyCR [13] enable applications to use node-local storage as burst buffers for shared files. CHFS [51] an adhoc parallel file system that utilizes the persistent memory of compute nodes. The design is based entirely on a highly scalable distributed key-value store with consistent hashing. HFetch [17] adopts a segment-level hierarchical caching to optimize scientific workflows, for datasets stored on PFS. These optimizations are intended to create full-fledged write and read access mechanisms for applications.

In-Memory Stores and Memory-mapped IO: In-memory stores choices like Memcached [9] and LMDB [34] provide scalable I/O for datasets and are used to improve DL workflows. Memcached is an in-memory, key-value store that uses a distributed hash table to store keys up to 250 bytes and value objects up to 1 MB, and places the burden on the application to manage mapping larger files onto multiple key-value pairs. LMDB is memory-mapped DB and relies on map to perform in-memory data access. Mmap is a generic Unix system call that maps the layout of a file on the file system to the virtual address space of a process, thus giving an illusion to the process that the entire file is in memory. However, both Memcached and LMDB have non-POSIX interface that requires application modification. Atop, Mmap interface is not suitable to complex ML/DL application access patterns, such

as strided access of batches of data [43].

VI. Conclusion

In this paper, we presented High-Velocity AI Cache (HVAC), which has been demonstrated to remove I/O bottleneck for large-scale DL applications running on leadership scale HPC systems. HVAC exploits the node-local storage on compute nodes and builds an aggregate cache layer atop to accelerate the DL application read performance. We discussed the motivation for building HVAC and the limitations of the current systems based on scalability or application and tool portability. We deploy and evaluate HVAC on 1,024 nodes (with over 6000 NVIDIA V100 GPUS) of the Summit supercomputer. In particular, we evaluate the scalability, efficiency, accuracy, and load distribution of HVAC compared to GPFS and XFS-on-NVMe. With four different DL applications, we observe an average 25% performance improvement atop GPFS and 9% drop against XFS-on-NVMe, which scale linearly and are considered the performance upper bound. We envision HVAC as an important caching library for upcoming supercomputers such as Frontier. Future works include comparison with existing caching and prefetching techniques, investigations of client/server aggregation ratios, data layout options for large files across multiple nodes, and job topology partitioning enabling redundancy for reliability and performance.

Acknowledgment

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] BeeOND (BeeGFS On Demand). http://www.beegfs.io/wiki/BeeOND.
[2] Caffe: A deep learning framework. https://caffe.berkeleyvision.org/.
[3] Encryption and a local read-only cache (LROC) device. https://www.ibm.com/docs/en/spectrum-scale/5.0.4?topic=encryption-local-read-only-cache-lroc-device.
[4] ExaIO: Access and Manage Storage of Data Efficiently and at Scale on Exascale Systems - Exascale Computing Project. https://www.exascaleproject.org/highlight/exaio-access-and-manage-storage-of-data-efficiently-and-at-scale-on-exascale-systems. (Accessed on 05/04/2022).
[5] Frontier supercomputer. https://www.olcf.ornl.gov/frontier/.
[6] ImageNet: An image database. https://www.image-net.org/.
[7] Livermore's El Capitan Supercomputer to Debut HPE 'Rabbit' Near Node Local Storage. https://www.hpcwire.com/2021/02/18/livermores-el-capitan-supercomputer-hpe-rabbit-storage-nodes/. (Accessed on 05/10/2022).
[8] MDTest. https://wiki.lustre.org/MDTest.
[9] Memcached. https://memcached.org/.
[10] Open Images Dataset. https://storage.googleapis.com/openimages/web/index.html.
[11] PyTorch. https://pytorch.org/.
[12] TensorFlow: An end-to-end open source machine learning platform. https://www.tensorflow.org/.
[13] UNIFY-CR. https://github.com/LLNL/UnifyFS.
[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.
[15] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–205, 2017.
[16] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu. I/o characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.
[17] H. Devarajan, A. Kougkas, and X.-H. Sun. Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 62–72, 2020.
[18] 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.
[19] S. Farrell, M. Emani, J. Balma, L. Drescher, A. Drozd, A. Fink, G. Fox, D. Kanter, T. Kurth, P. Mattson, D. Mu, A. Ruhela, K. Sato, K. Shirahata, T. Tabaru, A. Tsaris, J. Balewski, B. Cumming, T. Danjo, J. Domke, T. Fukai, N. Fukumoto, T. Fukushi, B. Gerofi, T. Honda, T. Imamura, A. Kasagi, K. Kawakami, S. Kudo, A. Kuroda, M. Martinasso, S. Matsuoka, H. Mendonça, K. Minami, P. Ram, T. Sawada, M. Shankar, T. S. John, A. Tabuchi, V. Vishwanath, M. Wahib, M. Yamazaki, and J. Yin. Mlperf hpc: A holistic benchmark suite for scientific machine learning on hpc systems, 2021.
[20] J. Fombellida, I. Martín-Rubio, S. Torres-Alegre, and D. Andina. Tackling business intelligence with bioinspired deep learning. *Neural Computing and Applications*, pages 1–8, 2018.
[21] V. Gupta-Cledat, L. Remis, and C. R. Strong. Addressing the dark side of vision research: Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.
[22] P. Hamandawana, A. Khan, J. Kim, and T.-S. Chung. Accelerating ml/dl applications with hierarchical caching on deduplication storage clusters. *IEEE Transactions on Big Data*, pages 1–1, 2021.
[23] J. Han, L. Xu, M. M. Rafique, A. R. Butt, and S.-H. Lim. A quantitative study of deep learning training on heterogeneous supercomputers. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019.
[24] D. Harnie, A. E. Vapirev, J. K. Wegner, A. Gedich, M. Steijaert, R. Wuyts, and W. De Meuter. Scaling machine learning for target prediction in drug discovery using apache spark. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '15, page 871–879. IEEE Press, 2015.
[25] M. S. Hossain and G. Muhammad. Environment classification for urban big data using deep learning. *IEEE Communications Magazine*, 56(11):44–50, 2018.
[26] Z. Hu, J. Tang, Z. Wang, K. Zhang, L. Zhang, and Q. Sun. Deep learning for image-based cancer detection and diagnosis- a survey. *Pattern Recognition*, 83:134–149, 2018.
[27] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
[28] S. A. Jacobs, B. Van Essen, D. Hysom, J.-S. Yeom, T. Moon, R. Anirudh, J. J. Thiagaranjan, S. Liu, P.-T. Bremer, J. Gaffney, T. Benson, P. Robin-

son, L. Peterson, and B. Spears. Parallelizing training of deep generative models on massive scientific datasets, 2019.

[29] B. Juanals and J.-L. Minel. Categorizing air quality information flow on twitter using deep learning tools. In *International Conference on Computational Collective Intelligence*, pages 109–118. Springer, 2018.

[30] A. Khan, T. Kim, H. Byun, and Y. Kim. Scispace: A scientific collaboration workspace for geo-distributed hpc data centers. *Future Generation Computer Systems*, 101:398–409, 2019.

[31] A. Khan, H. Sim, S. S. Vazhkudai, A. R. Butt, and Y. Kim. An analysis of system balance and architectural trends based on top500 supercomputers. In *The International Conference on High Performance Computing in Asia-Pacific Region*, pages 11–22, 2021.

[32] A. V. Kumar and M. Sivathanu. Quiver: An informed storage cache for deep learning. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 283–296, 2020.

[33] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[34] N. Laanait, M. A. Matheson, S. Somnath, and J. Yin. Stemdl. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2019.

[35] S.-H. Lim, S. R. Young, and R. M. Patton. An analysis of image storage systems for scalable training of deep neural networks.

[36] 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, Prabhat, and V. Lee. Cosmoflow: Using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[37] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram. Analyzing and mitigating data stalls in DNN training. *CoRR*, abs/2007.06775, 2020.

[38] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram. Analyzing and mitigating data stalls in dnn training, 2020.

[39] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, and V. V. Larrea. End-to-end i/o portfolio for the summit supercomputing ecosystem. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[40] S. Partee, M. S. Ellis, A. Rigazzi, S. D. Bachman, G. Marques, A. E. Shao, and B. Robbins. Using machine learning at scale in hpc simulations with smartsim: An application to ocean climate modeling. *ArXiv*, abs/2104.09355, 2021.

[41] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari. Revisiting i/o behavior in large-scale storage systems: The expected and the unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[42] C. Pinto, Y. Gkoufas, A. Reale, S. Seelam, and S. Eliuk. Hoard: A distributed data caching system to accelerate deep learning training on the cloud. *arXiv preprint arXiv:1812.00669*, 2018.

[43] S. Pumma, M. Si, W.-C. Feng, and P. Balaji. Scalable deep learning via i/o analysis and optimization. *ACM Trans. Parallel Comput.*, 6(2), jul 2019.

[44] Y. Qian, X. Li, S. Ihara, A. Dilger, C. Thomaz, S. Wang, W. Cheng, C. Li, L. Zeng, F. Wang, et al. Lpcc: Hierarchical persistent client caching for lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[45] T. Ridnik, E. Ben-Baruch, A. Noy, and L. Zelnik-Manor. Imagenet-21k pretraining for the masses, 2021.

[46] F. Rodrigo Duro, J. Garcia Blas, F. Isaila, J. Carretero, J. M. Wozniak, and R. Ross. Experimental evaluation of a flexible i/o architecture for accelerating workflow engines in ultrascale environments. *Parallel Computing*, 61:52–67, 2017. Special Issue on 2015 Workshop on Data Intensive Scalable Computing Systems (DISCS-2015).

[47] H. Sim, A. Khan, S. S. Vazhkudai, S.-H. Lim, A. R. Butt, and Y. Kim. An integrated indexing and search service for distributed file systems. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2375–2391, 2020.

[48] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 909–923, 2019.

[49] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8. IEEE, 2013.

[50] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

[51] O. Tatebe, K. Obata, K. Hiraga, and H. Ohtsuji. Chfs: Parallel consistent hashing file system for node-local persistent memory. In *International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia2022, page 115–124, New York, NY, USA, 2022. Association for Computing Machinery.

[52] R. Vasudev. Why Distributed Deep Learning is Going to Gain Paramount Importance? Why should you care about HPC for Deep Learning. https://towardsdatascience.com/why-distributed-deep-learning-is-going-to-gain-paramount-importance-bd2d83517483.

[53] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, and Q. Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[54] Y. Wang, G.-Y. Wei, and D. Brooks. A systematic methodology for analysis of deep learning hardware and software platforms. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 30–43, 2020.

[55] C. Xu, S. Bhattacharya, M. Foltin, S. Byna, and P. Faraboschi. Data-aware storage tiering for deep learning. In *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW)*, pages 23–28, Los Alamitos, CA, USA, nov 2021. IEEE Computer Society.

[56] C.-C. Yang and G. Cong. Accelerating data loading in deep neural network training. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 235–245. IEEE, 2019.

[57] 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, 2018.

[58] Z. Zhang, L. Huang, J. G. Pauloski, and I. T. Foster. Efficient i/o for neural network training with compressed data. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 409–418. IEEE, 2020.

[59] 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)*, pages 145–156. IEEE, 2018.