DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications

Hariharan Devarajan[1], Huihuo Zheng[2], Anthony Kougkas[1], Xian-He Sun[1], and Venkatram Vishwanath[2]

hdevarajan@hawk.iit.edu, huihuo.zheng@anl.gov, akougkas@iit.edu, sun@iit.edu, venkat@anl.gov
[1]Illinois Institute of Technology, Chicago
[2]Argonne National Laboratory

***Abstract*—Deep learning has been shown as a successful method for various tasks, and its popularity results in numerous open-source deep learning software tools. Deep learning has been applied to a broad spectrum of scientific domains such as cosmology, particle physics, computer vision, fusion, and astrophysics. Scientists have performed a great deal of work to optimize the computational performance of deep learning frameworks. However, the same cannot be said for I/O performance. As deep learning algorithms rely on big-data volume and variety to effectively train neural networks accurately, I/O is a significant bottleneck on large-scale distributed deep learning training. This study aims to provide a detailed investigation of the I/O behavior of various scientific deep learning workloads running on the Theta supercomputer at Argonne Leadership Computing Facility. In this paper, we present DLIO, a novel representative benchmark suite built based on the I/O profiling of the selected workloads. DLIO can be utilized to accurately emulate the I/O behavior of modern scientific deep learning applications. Using DLIO, application developers and system software solution architects can identify potential I/O bottlenecks in their applications and guide optimizations to boost the I/O performance leading to lower training times by up to 6.7x.**

***Index Terms*—deep learning; scientific applications; representative; benchmark; data intensive; I/O; characterization; Tensorflow; data pipeline;**

I. Introduction

In the past decade, deep learning (DL) has been applied to a wide range of applications [1] with tremendous success. These include image recognition [2], natural language processing [3], autonomous driving [4], as well as physical science domains such as cosmology [5], materials science [6], [7], and biology [8], [9]. Using DL methods in scientific applications is beneficial in two meaningful ways: a) accelerate time-to-results by minimizing the simulation cost, b) extract patterns out of large datasets where heuristics cannot. DL methods achieve this by iteratively adjust the weights within the network to minimize a loss function. At each training step, the application reads a mini-batch of data, computes the gradient of the loss function, and then updates the network's weights using stochastic gradient descent. Many new AI hardware (e.g., GPU, TPU, Cerebras, etc.) have been designed and deployed to accelerate the computation during the training. However, as the size and complexity of the data sets grow rapidly, DL training becomes increasingly read-intensive with I/O being a potential bottleneck [10]. On the other hand, more and more studies on scientific DL application are conducted on supercomputers through a distributed training framework to reduce the training time-to-solution [11]. Therefore, characterizing the I/O behavior [12] of DL workloads in high-performance computing (HPC) environments is crucial to address any potential bottlenecks in I/O and to provide useful insights to guide I/O optimizations.

Data is the core of all deep learning methods. As the data volume explodes, efficient data ingestion and processing becomes a significant challenge. To address this, the cloud community has developed several comprehensive DL frameworks such as TensorFlow [13], Pytorch [14], and Caffe [15] that encapsulate methods to tune different data access parameters. These tuning parameters are tailored to heterogeneous cloud environments [16], including: different data sources and data representations (e.g., textual formats or custom data formats such as TFRecord); mechanisms for adjusting worker assignment to read, process, and feed data into a distributed training process; and finally, hardware-specific optimizations [17], (e.g., RDMA, GPU-Direct, NVLink etc.,) to enable efficient data movement within the application. These optimizations improve data access significantly in these environments.

Studying the behavior of DL applications allows developers to fine tune their training pipelines. Benchmarks suites have traditionally been used to drive insights and reason about the expected performance of applications. In this study, we highlight three major hurdles in developing such benchmarks. First, existing DL benchmarks have been focusing on characterizing the computational capabilities of DL frameworks [18] but do not address their data management competency. Second, this compute-centric thinking has led to a lack of a standard methodology to quantify the benefits of existing data access optimizations implemented by DL frameworks for efficient data ingestion in scientific workflows. Third, I/O research and optimization [19]–[22] for DL applications in HPC requires the adoption of mini-applications [23] that encapsulate the data access and processing characteristics of complex scientific DL workflows. Hence, the existence of mini-applications will enable fast prototyping and testing of novel and innovative solutions. Hence, a benchmark suite that can encapsulate the data access behavior of various scientific DL applications is crucial.

In this work, we present DLIO, an I/O benchmark for scientific DL applications. DLIO aims to accurately characterize the behavior of scientific DL applications and guide data-centric

978-1-7281-9586-5/21/$31.00 ©2021 IEEE
DOI 10.1109/CCGrid51090.2021.00018

optimizations on modern HPC systems. To build this benchmark, we first characterize the behavior of modern scientific DL applications currently running on production supercomputers at Argonne Leadership Computing Facility (ALCF). Our approach captures a wide variety of application behaviors, from different scientific domains, informed by several active projects such as Argonne Data Science Program (ADSP), Aurora Early Science Program (ESP), and DOE's Exascale Computing Project (ECP). In order to acquire a holistic view of how data is accessed in DL applications, we utilized both high-level and low-level I/O profiling tools. DLIO incorporates the observed I/O behavior in these applications and provides tunable mechanisms to test and adjust different I/O access optimizations. Our benchmark suite is validated by statistically comparing the generated I/O behaviors with the applications. DLIO uses mini-applications to emulate several DL application behaviors. Lastly, DLIO provides a highly tunable data-generation toolkit that can be used to project the behavior of DL applications at scale. The contributions of this work are:

1) A comprehensive study of the I/O behavior of eight scientific DL applications on a production supercomputer (III).
2) The design and implementation of a modular and flexible I/O benchmark for scientific DL applications (IV).
3) An illustration of how DLIO can guide software optimizations to boost application's I/O performance (V).

II. Related Work

HPC benchmarks for DL have concentrated on measuring the machine's computing power. There are several challenges in the DL domain such as system heterogeneity, the variety of deep learning workloads, the stochastic nature of approaches, and the difficulty in designing simple, yet comprehensive, measurements. Researchers have attempted to highlight these challenges by incorporating different machines [1], [24] or DL algorithms [25], [26]. All of these benchmarks focus solely on capturing the computation aspect of DL workloads on HPC systems. However, this work aims to capture the I/O behavior for many scientific DL workloads so as to propel innovations and designs. Scientists have [27], [28] characterized the I/O behavior of deep learning application's I/O performance over parallel file systems running in HPC infrastructure. However, those studies were limited to single node and Imagenet Benchmark evaluations and characterizations. Our study aims to provide a deeper dive into various scientific DL applications in HPC and build a representative benchmark which can further research and development.

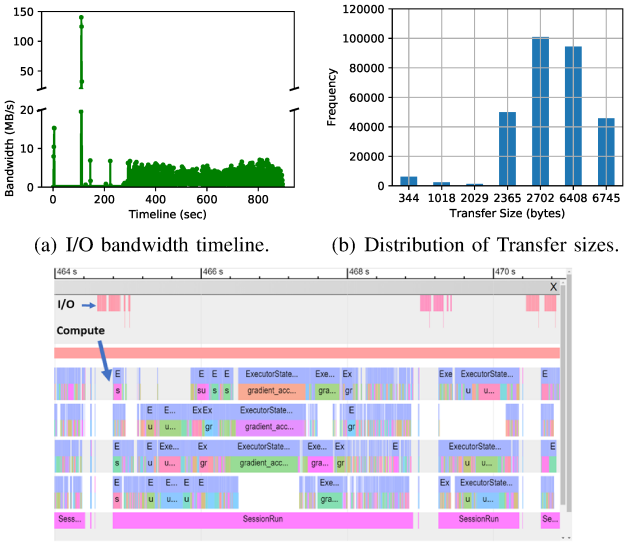
DL in cloud environments inspires more and more interests from both academia and industry; hence, a series of benchmarks have been proposed. Fathom [25], BenchNN [29], DeepBench [30], and MLPerf [31] provides multiple deep learning workloads and models implemented with TensorFlow. DNNMark [32] is GPU benchmark suites that consists of a collection of deep neural network primitives. All of these benchmarks target cloud platforms whereas scientific workloads are typically run on supercomputing platforms. Additionally, unlike this work, their focus is to express the computation requirements of DL workloads but not the I/O requirements. Scientists have proposed I/O frameworks [33], [34] for training deep neural networks by enabling RDMA-assisted in-situ shuffling, input pipelining and entropy-aware opportunistic ordering. These frameworks are benchmarked against the TensorFlow dataset API, and a portable API for TensorFlow is developed to leverage DeepIO on different storage systems. However, our work focuses on characterizing and optimizing existing TensorFlow applications by building a representative benchmark targeting scientific DL applications.

III. Methodology

In this section, we aim to understand the I/O behavior in scientific DL applications. We explore a collection of scientific deep learning workloads currently running at the Argonne Leadership Computing Facility (ALCF). These workloads are selected from various projects, such as Argonne Data Science Program (ADSP), Aurora Early Science Program (ESP), and Exascale Computing Projects (ECP). The science domains represented by the workloads include physics [5], [35], cosmology [36], materials science [6], and biology [8], [9]. Many of the workloads are in active development targeting upcoming exascale supercomputers. One of the long term goals for this study is to identify any existing I/O bottlenecks in these workloads on current production machines and suggest I/O optimizations for current applications and provide a road map for future systems. We profile the I/O behavior of eight DL applications on Theta, the current production leadership-class supercomputer at ALCF. We utilize the profilers provided by the DL framework, such as the TensorFlow profiler as well as low-level I/O profiler such as Darshan, to study the I/O behavior of applications. These profilers are accompanied with analysis tools. However, to get a holistic view of the application, we developed our own Python-based analysis tool, VaniDL [37], for post-processing the information obtained from profiling tools and generating high level I/O summary.

A. I/O behavior of scientific Deep learning applications

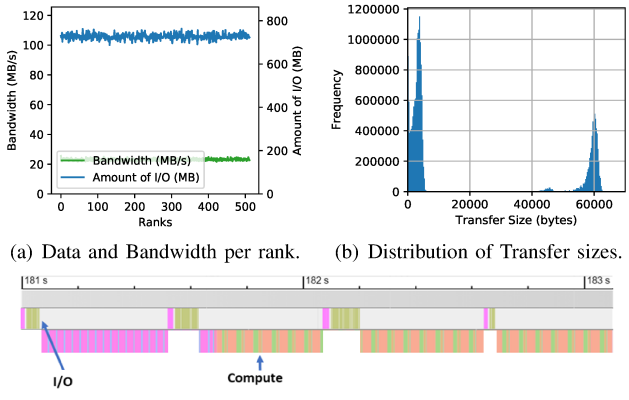
Hardware: We run the applications on Theta [38]. Theta consists of 4392 compute nodes and 864 Aries routers interconnected with a dragonfly network. Each router hosts four compute nodes, each contains 64 2nd generation Intel Xeon Phi^{TM} processors, code name Knights Landing (KNL). Each node is equipped with 192 GB of DDR4 and 16 GB of MCDRAM. In all the studies presented here, we set two hyper-threads per core for a total of 128 threads per node, and four MPI processes per node. The datasets are stored in the HDD-based Lustre file system of size 10 PB with 56 OSTs. We set the Lustre stripe size to be 1 MB and stripe count to be 48. The peak read performance the Lustre filesystem is 240 GB/s.
Applications: We target distributed DL workloads. These include Neutrino and Cosmic Tagging with UNet [5], Distributed Flood Filling Networks (FFN) for shape recognition in brain tissue [6], Deep Learning Climate Segmentation [39], CosmoFlow for learning universe at scale [36], Cancer Distributed Learning Environment



(a) I/O bandwidth timeline. (b) Distribution of Transfer sizes.

(c) Compute and I/O timeline.

Fig. 1. I/O behavior of Cosmic Tagger: Figure a) shows the application achieved an aggregate bandwidth of 11 MB/s. Figure b) shows that most requests were made between 2 KB and 6 KB due to the small chunk size set in the dataset. Figure c) shows 10% of the I/O is hidden behind the compute.



(a) Data and Bandwidth per rank. (b) Distribution of Transfer sizes.

(c) Compute and I/O timeline.

Fig. 2. I/O behavior of DFFN: Figure a) shows the equal distribution of I/O between ranks of the application and a bandwidth per rank of 22 MB/s. Figure b) shows that small accesses (i.e., 100 Bytes) are from the metadata file and large accesses (i.e., 64 KB) are from the data file. Figure c) showcases the distinct I/O and computation phases.

(CANDLE) for cancer research [8], Fusion Recurrent Neural Net for representation learning in plasma science [9], Learning To Hamiltonian Monte Carlo (L2HMC) [35], and TensorFlow CNN Benchmarks [40]. These applications are implemented in TensorFlow and use Horovod for data parallel training.

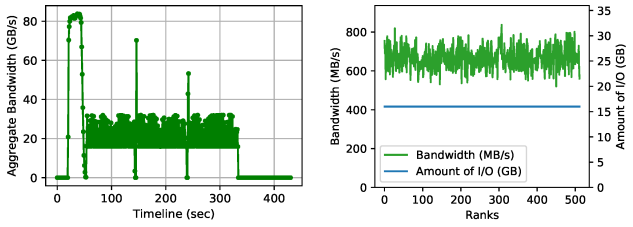
Tools: We use Darshan (with extended tracing) as our low-level I/O profiling tool along with the TensorFlow profiler. Additionally, we process the profiling results using our custom analytic tool, VaniDL [41] to integrate the low-level Darshan logs with high-level TensorFlow profiler logs and generate a holistic I/O information of the application, such as I/O access pattern, transfer size distributions for all the I/O operations, I/O access timeline, etc. All the results described below are the outcome of the analysis produced by VaniDL. In order to see how much overlap there is between compute and I/O, we align the I/O timeline generated from Darshan profiling results and the TensorFlow timeline. We use anchors (e.g., timestamp to a file) placed within code to align the timelines.

1) Neutrino and Cosmic Tagging with UNet [5]: Cosmic Tagger is a convolutional network for separating cosmic pixels, background pixels, and neutrino pixels in a neutrino dataset. In our benchmark, the application reads 33 GB of the dataset stored in HDF5 format using the larcv3 [42] library. The dataset contains 43000 samples, each of which contains three images of size 1280×2048. The samples are stored in an HDF5 dataset as sparse data of average size 40 KB with a standard deviation of 10 KB. The application is run for 150 steps on a single epoch. At each step, each process reads 32 images and perform pre-processing with twelve initial filters.

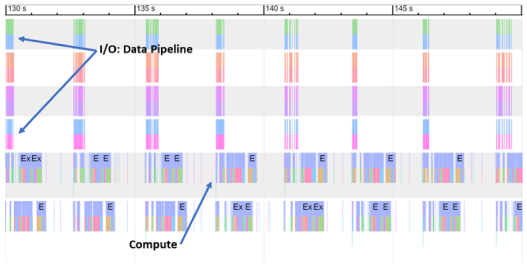
Figure 1 shows the I/O profile of the application's behavior. The application spent 227 seconds on I/O, which is 22% of the overall time. It reads the whole dataset sequentially with an average aggregate bandwidth of 11 MB/s. The HDF5 file is opened at the beginning of the training, which triggers Lustre's prefetching at the beginning of the file. This results in the initial high bandwidth seen in Figure 1(a). After this, each process continues reading data for each step of the training with an average bandwidth of 4 MB/s. This relatively low bandwidth observed can be explained by the transfer-size distribution of the application. The HDF5 file is chunked with 2 KB chunk size. This chunking creates a small I/O unit (2KB and 6KB on the figure) on the parallel file system which, as known by previous studies [43], resulting in a sub-optimal performance. Finally, Figure 1(c) shows the compute and I/O timeline obtained by merging Darshan tracing with TensorFlow profiler tracing. It is found that the application uses a single thread for I/O and multiple threads for computation. At each step, the I/O occurs before the computation starts. Part of the I/O overlaps with the compute. Out of 227 seconds of I/O, 23 seconds was hidden behind compute. This accounts for 10% of the overall I/O time.

2) Distributed Flood Filling Networks (DFFN) [6]: DFFN is a recurrent 3D convolutional network for segmenting complex and large shapes of neurons from a brain tissue's raw image. In our benchmark, the application reads a dataset of 2.28 GB from a HDF5 file. The I/O is performed using the h5py python library. The data is stored in two files: one contains real data and the other contains metadata associated with the dataset (e.g., size of samples, location of samples within the dataset, etc.). The training dataset consists of 18678 samples, each of size 32×32×32. The samples are read by the application with 4096 fields of view. The application runs for 400 steps in one epoch. At each step, each process reads a batch of 32 images.

Figure 2 shows the behavior of the DFFN application. Every process in the application reads the 1/3rd of the dataset (i.e., overall I/O of 363 GB) randomly in 67.858 seconds (i.e., 18 % of the overall time). The dataset was read by the

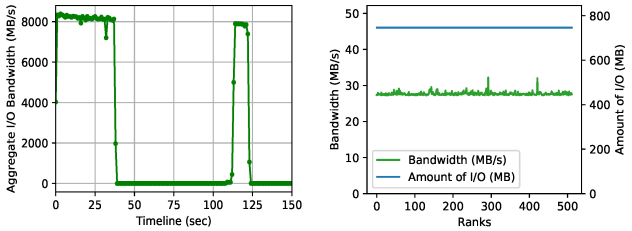


(a) I/O bandwidth timeline. (b) Data and Bandwidth per rank.

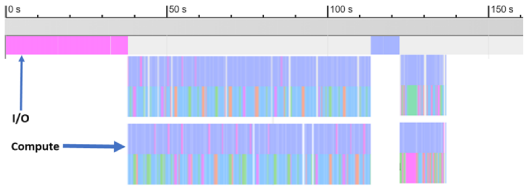


(c) Compute and I/O timeline.

Fig. 3. I/O Behavior of CosmoFlow: Figure a) shows that the application achieves a bandwidth of 36 GB/s. Figure b) shows the average bandwidth achieved per rank within the application is 750 MB/s. Figure c) shows that I/O and compute completely overlap within the application.



(a) I/O bandwidth timeline. (b) Data and Bandwidth per rank.



(c) Compute and I/O timeline.

Fig. 4. I/O Behavior of CANDLE: Figure a) shows the aggregate bandwidth achieved of 8 GB/s for the applications. Figure b) depicts the distribution of I/O (i.e., 700 MB) and the achieved bandwidth (28 MB/s) across ranks. Figure c) shows the merged timeline of I/O and compute, which shows that I/O and compute do not overlap in the application.

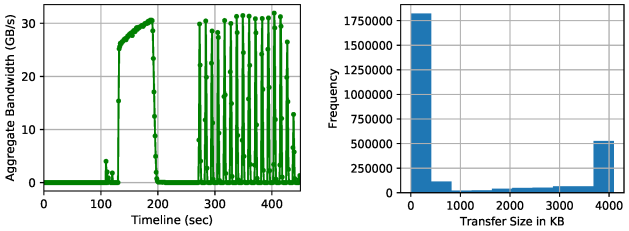
application (512 ranks) with an aggregate bandwidth of 11 GB/s [Figure 2(a)]. The application reads this dataset with the distribution of the transfer size centers at two places [Figure 2(b)], one at about 64 KB (for reading images), the other at about 100B (for reading metadata). A chunk size of 64KB is used in the data file. That is why the transfer size for reading images is about 64 KB. This dataset is accessed randomly due to shuffling and then a batch of images is read. Finally, Figure 2(c), shows the merged timeline of compute and I/O within the application. In this case, we do not see overlap between I/O and compute.

3) Cosmoflow Benchmark [36]: CosmoFlow is a 3D convolutional neural network model for studying the features in the distribution of dark matter. The application reads a dataset of size 2 TB, which consists of 1024 TFRecord files. The dataset is accessed using TensorFlow's `tf.data` APIs. Each TFRecord file consists of 262,144 samples, each of size 128×128×128×4. The application runs for four epochs with 256000 steps. The batch size is one. That is, each process reads one image from the dataset at each step.

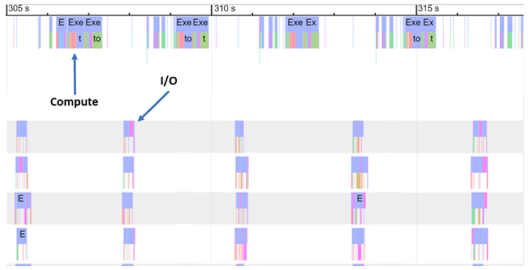
Figure 3 shows the behavior of the CosmoFlow application. The application ran for 431 seconds, out of which 12% (i.e., 51 seconds) was spent on I/O. During the whole benchmark, the application read 8 TB of data with a bandwidth of 36 GB/s [Figure 3(a)]. Each process of the distributed TensorFlow training reads eight files completely with a bandwidth of 756 MB/s [Figure 3(b)]. The transfer size for each request is 256 KB, which is the default in TFDataset APIs. Figure 3(c) shows the merged timeline of I/O and compute within CosmoFlow. This request size on Lustre (with stripe size of 1 MB), results in higher bandwidth for next consecutive calls with higher bandwidth. Initially, as there is no model computation, TensorFlow data pipeline performs I/O frequently. Once the model computation starts, the count of requests per second reduces. This results in a lower aggregate bandwidth as shown in the figure. The application uses eight threads per process for computation (only two are shown in the figure) and four threads for the I/O data pipeline. The I/O data pipeline consists of two parts: I/O and data preprocessing. Figure 3(c) shows the data pipeline takes 23% of the overall time and preprocessing takes 11% of the time. Finally, the application completely overlaps its I/O with compute.

4) Cancer Distributed Learning Environment (CANDLE) [8]: CANDLE is a 1D convolutional network for classifying RNA-seq gene expression profiles into normal or tumor tissue categories. The application reads 700 MB dataset stored in CSV format using the Pandas framework. The dataset is divided into train and test dataset with a total of 1120 records. Each record contains 65536 columns of 32 bit floating-point numbers. The application run over 60 epochs with a batch size of 20 records.

Figure 4 shows the I/O behavior of the application. The application ran for 290 seconds, out of which 36% was spent on reading the test and train dataset. Each process reads the whole dataset in-memory row-by-row from the CSV file with an aggregate bandwidth of 8 GB/s [Figure 4(a)]. The application reads the entire training dataset to the memory at once and then performs training. This is possible as the size of the dataset size is 700 MB; hence, it can fit easily into memory. As the dataset grows, the current pandas CSV load would fail, and the application would have to do out of core training. Each process reads the whole dataset with a bandwidth of 28 MB/s [Figure 4(b)] with a transfer size of 256 KB (size of each row). Figure 4(c) shows the merged timeline of I/O and compute for the whole application. We can observe that the application uses eight computation threads (6 shown here)

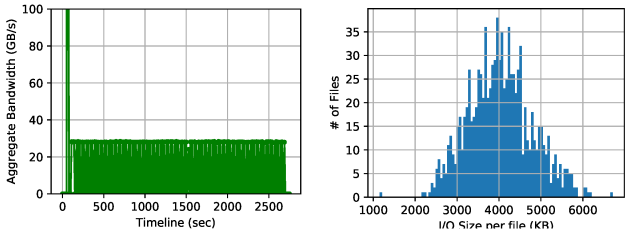


(a) I/O bandwidth timeline. (b) Transfer Size distribution.

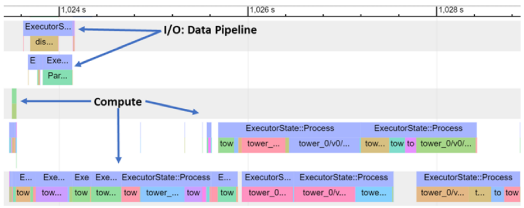


(c) Compute and I/O timeline.

Fig. 5. I/O behavior of FRNN: Figure a) shows the aggregate bandwidth of 28 GB/s. Figure b) shows that distribution of transfer size is highly concentrated near small accesses on metadata files and large accesses near data files. Finally, Figure c) shows the lack of overlap between computation and I/O.



(a) I/O bandwidth timeline. (b) Data distribution across files.



(c) Compute and I/O timeline.

Fig. 6. I/O behavior of Imagenet: Figure a) shows the aggregate bandwidth of 32 GB/s achieved by the application. Figure b) shows the distribution of accesses per file made by the TensorFlow data pipeline. Finally, Figure c) shows the complete overlap of I/O and computation.

with one thread for performing I/O, with no overlap between the computation and I/O.

5) Fusion Recurrent Neural Net (FRNN) [9]: FRNN is a deep learning model for disruption prediction in tokamak fusion plasmas. It accesses data in the Numpy array (NPZ) format using Numpy APIs. The total size of the dataset is 6 GB, which is divided into 2800 signal files. Each file is 2 MB in size and contains 1024 samples, each of size 2 KB. The application uses 1M samples with 100 estimates in a random forest model with a maximum depth of 3. It uses an RBF kernel with three hidden layers and a learning rate of 0.1. Each training step is fed with a batch of 1024 samples per-rank.

Figure 5 shows the I/O behavior of the application. The application runs for 436 seconds, out of which 23% of the time is spent on I/O. Every process reads the dataset for 28 GB across the application with a bandwidth of 28 GB/s. The data is read from the signal files before each training step is executed [Figure 5(a)]. The application makes many requests on the signal shots file, which is the metadata of the signal files. Hence, we observe in Figure 5(b) many small I/O accesses. The large accesses in the Figure (around 4 MB) are from the data files. The Figure 5(c) shows the merged timeline of I/O and computation. It can be observed, the application uses eight threads for computation (one shown here), and multiple threads are used within the app for I/O. Additionally, the data is read before each step of the training, and there is no overlap between computation and I/O.

6) Imagenet Benchmark [40]: It is an image classification benchmark which contains implementations of several popular convolutional models such as resnet50, inception3, vgg16, and alexnet. It accesses a dataset of images in the TFRecord format. The dataset consists of 1024 files with each file containing 1024 sample images of size 256 KB. The images are serialized using protocol buffers. The application feeds images with a batch size of 32. The alexnet model is trained with 100 steps.

Figure 6 shows the I/O behavior of the Imagenet application. The application has a total runtime of 2757 seconds, out of which 15% was spent on performing I/O. The application reads the whole dataset with a bandwidth of 32 GB/s 6(a). The files are read on-demand during the training, and hence the samples per file access are not uniformly distributed among the files 6(b). The files are selected at random but are accessed sequentially. The data transfer size of the request is 256 KB, which is the default for TFDataset APIs. Similar to CosmoFlow, this application also has a higher aggregated bandwidth initially, and then request per second reduces once the model computation starts. Figure 6(c) shows the merged timeline of I/O and computation within the application. The application uses four threads for compute and two threads for data pipeline (i.e., I/O plus preprocessing). The overall data pipeline takes 28% of the overall time of the application. As can be observed, the I/O is completely overlapped with the compute of TensorFlow training.

7) Deep Learning Climate Segmentation Benchmark [39]: The Climate Segment benchmark is based on the Exascale Deep Learning project for Climate Analytics, which comprises multiple deep learning models for different climate data projects such as AR detection, Storm tracking, and Semantic segmentation. The application generates its training dataset in memory based on a stats file. At the end of the training, it creates a JSON file that contains the summary of the model and its tuned hyper-parameters. The application runs for one epoch with 1200 steps and a batch size of one sample. We observe that the application does not perform much I/O. It initially reads a 4 KB HDF5 file over 50 operations of 500 bytes in size. Then the benchmark shows training and, each

rank writes the model parameters to a 4 MB JSON file. The stat file is read randomly during the application's lifetime.

8) L2HMC Algorithm with Neural Network [35]: This application uses the Learning To Hamiltonian Monte Carlo (L2HMC) algorithm to generate gauge configurations for LatticeQCD. L2HMC provides a statistically exact sampler that can quickly converge to the target distribution (fast burn-in), produce uncorrelated samples (fast mixing), efficiently mix between energy levels, and is capable of traversing low-density zones to mix between modes (often difficult for generic HMC). The application generates data synthetically and performs the training. During training, it performs checkpointing of the model for restart purposes. The checkpoint files are written in TensorFlow using STDIO, which uses a single operation to write the files. The application runs over 150 steps with a batch size of 32 records per training step. It checkpoints the model data in every 50 steps. As the application generates data synthetically in memory, the only I/O within the app is during the checkpoint. The checkpoint data is written with STDIO interface. The checkpoint writes four files of several megabytes in size. The write time is less than 1% of the total execution time. This checkpointing is performed by the root process of the distributed TensorFlow training.

IV. Deep Learning I/O Benchmark

The DLIO benchmark [44] is aimed at emulating the behavior of scientific deep learning applications. The benchmark is delivered as an executable that can be configured for various I/O patterns. It uses a modular design to incorporate more data formats, datasets, and configuration parameters. It emulates modern scientific deep learning applications using Benchmark Runner, Data Generator, Format Handler, and I/O Profiler modules. These modules utilize state-of-the-art design patterns to build a transparent and extensible framework. The DLIO benchmark has been designed with the following goals.

1) **Accurate:** DLIO should be an accurate representation of selected scientific deep learning applications. It should incorporate all the I/O behavior seen in various configurations of applications. Additionally, It should act as a mini-application that can precisely mimic the I/O behavior of scientific deep learning applications.
2) **Configurable:** DLIO should be easily configurable for different scenarios required by the user. These include features such as the ratio-of-computation to I/O, available threads for I/O, data operators (e.g., decoding, shuffling, and batching), and mechanism to feed data into training.
3) **Extensible:** DLIO benchmark should allow adding custom data directories and enable easy extensions to the benchmark to incorporate different data formats or data generation algorithms. These changes should not affect the basic benchmark operations.

A. Architecture

Figure 7 shows the high-level design of the DLIO benchmark. The user runs the benchmark executable with command-line arguments (i.e., different I/O configurations).

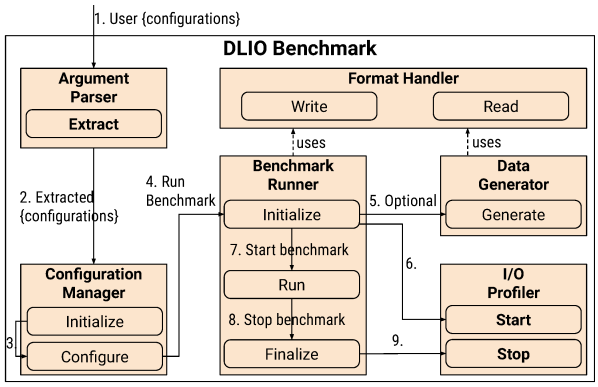


Fig. 7. DLIO high-level design

The arguments are parsed and extracted into configurations for the benchmark. The extracted configurations are passed to the Configuration Manager, which is first initialized with default benchmark values and then updates itself with the incoming configurations. At this stage, incompatible/incorrect configurations would be thrown as error back to the users. Once the configurations are validated and applied, the benchmark runner is invoked. The runner initializes prepared data (if needed) and then starts the profiling session. Once the session has started successfully, the benchmark `Run()` is invoked, which runs the benchmark. In the run phase, we run the benchmark for e epochs. During each epoch, the whole data is read once using n steps. During an epoch, checkpoint operations are performed every c steps as well. Additionally, an inter-step computation is performed to emulate computation and I/O phases by deep learning application. Finally, once the benchmark run finishes, the finalize is called, which stops the profiler, saves its results, and exits the benchmark.

B. Benchmark Configurations

The benchmark configurations are motivated by the I/O behavior observed in Section III. We describe the configurations present in the benchmark below:

1) **Interface:** The benchmark supports popular data format interfaces that were observed in various scientific deep learning applications. These interfaces include TensorFlow data format (i.e., TFRecord), hierarchical data format (i.e., HDF5), textual data format (i.e., CSV), and finally, scientific array format (i.e., NPZ). These data formats are accompanied by their own I/O interface to load and access the data. This is designed using the Format Handler component in the design.
2) **File Access Pattern:** Among the deep learning applications, files are accessed using an independent I/O interface (i.e., POSIX) or using collective I/O interfaces (i.e., MPI-IO). Independent I/O can be further classified into multiple processes reading multiple files or multiples processes reading a single shared file. These two patterns depend on how the dataset is generated and partitioned for the training

process. For collective I/O, we utilize coordinated data access using MPI-I/O collective calls across processes.

3) **Data Access Pattern:** In most cases of scientific deep learning applications, data is read sequentially and consecutively from the dataset. In certain cases, each process might perform data shuffling where the process would first jump to a monotonically increasing random offset and then read a batch of records. These two data access pattern encompasses the behaviors of all observed DL applications.
4) **I/O Types:** There are primarily two I/O behaviors within scientific deep learning applications. First, the data is read before training. Second, for every c steps, the model checkpoint is written in the filesystem. The latter is a straightforward process where five files (two 1KB, one 4KB, one 64KB, and one 4MB file) are written. The former includes either reading the complete dataset into memory or partially on-the-fly. Since this data access is achieved based on the data format used, the Format Handler APIs enable this feature.
5) **Transfer Buffer:** The transfer buffer size is the unit in which I/O is performed within the application. For TFRecord, the transfer buffer size is set during the TensorFlow API call, which is defaulted to 256 KB. For other data formats, this variable depends on the record-size$\times$batch-size.

C. Implementation Details

The DLIO benchmark is implemented in python (version 3.6). The extensible parts of the benchmark, such as Configuration Manager and Format Handler, utilize factory patterns to enable simple interfaces that can be implemented to extend the benchmark functionality. The implementation of DLIO uses existing I/O interfaces of various data formats such as TFRecord, HDF5, CSV, and NPZ. Additionally, we include profiler packages such as Darshan and TensorFlow profiler to enable profiling within the benchmark. The computations within the benchmark are emulated using busy waiting loops, to emulate the overlap of computation and I/O accurately.

V. Evaluations

A. Methodology

Testbed: We ran the scientific deep learning applications on the Theta supercomputer [38]. We run the TensorFlow framework on CPUs with two hyper threads available for a total of 128 threads per node. The datasets are stored in the Lustre file system with stripe size of 1 MB and stripe count of 48. The peak read performance the Lustre filesystem is 240 GB/s.

Software Used: We use Darshan (with extended tracing) and TensorFlow profiler to measure the I/O performance of the benchmark. Additionally, we use the VaniDL analyzer tool. We use TensorFlow 2.2.0 with Horovod 0.19.5 for distributed training. Additionally, we use NumPy version 1.19.1, h5py version 2.10.0, and mpi4py version 3.0.3.

Application	I/O time Difference	# files Difference	Sequential Access Difference	Consecutive Access Difference	Transfer Size Difference		Bandwidth Difference		Overall Similarity
					Median	Average	Median	Average	
Imagenet	0.18	0.07	-0.01	-0.01	0.00	0.00	-0.65	-0.53	0.96
Cosmic Tagger	-0.05	0.00	0.15	0.13	-0.38	0.31	0.32	0.42	0.97
Cosmoflow	-0.10	-0.34	0.37	0.27	0.00	0.71	0.00	0.26	0.95
FFN	-0.07	0.00	0.03	0.09	-0.29	-0.40	0.00	-0.10	0.94
CANDEL	-0.19	0.00	-0.06	-0.06	0.00	-0.40	-0.07	-0.68	0.97
FRNN	-0.15	0.02	-0.13	-0.13	-0.37	-0.02	0.03	-0.08	0.94

Fig. 8. DLIO Similarity with Real Applications: shows a cosine similarity. The factors (normalized) used for similarity are overall I/O time (in seconds), data read (in bytes), transfer size distribution, achieved bandwidth per operation (in MB/s), percentage of sequential and consecutive accesses, and the number of files read. The figure shows a correlation of 94% for all apps.

B. Benchmark Verification

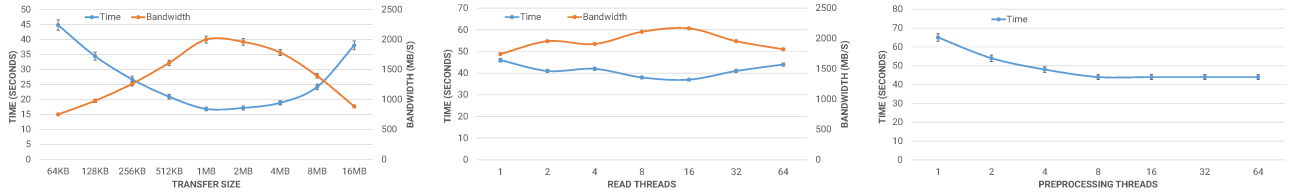
In this section, we test if our designed benchmark can represent the real application's I/O behavior. To achieve this, we run our benchmark with a workload similar to their application counterpart (as observed in Section III) and calculate the similarity between the two runs (i.e., DLIO benchmark and real application). To calculate the similarity, we use the cosine similarity metric $S = \frac{\bar{A}\cdot\bar{B}}{||\bar{A}||||\bar{B}||}$, where $\bar{A}_i = \frac{A_i}{max(A_i,B_i)}$, $\bar{B}_i = \frac{B_i}{max(A_i,B_i)}$. The parameters for calculating similarity include overall I/O time (in seconds), amount of data written (in bytes), transfer size distribution (such as min, max, mean, and median), achieved bandwidth per operation (in MB/s), percentage of sequential and consecutive accesses, and the number of files read. For each of the metric, we normalize the data using *max* for each parameter. We run the similarity test on a 128-node configuration and calculate the above metrics across all the processes.

Figure 8 shows the results. We see the DLIO benchmark, in all cases, achieves over 90% similarity in I/O behavior. The difference is given by $\frac{Baseline-DLIO}{max(Baseline,DLIO)}$. This similarity validates that the DLIO benchmark is an accurate representation of the real applications. The loss of 3-6% similarity is because all applications have a distribution of transfer request sizes, which is represented as a median request size within the benchmark.

C. Optimizations using DLIO Benchmark

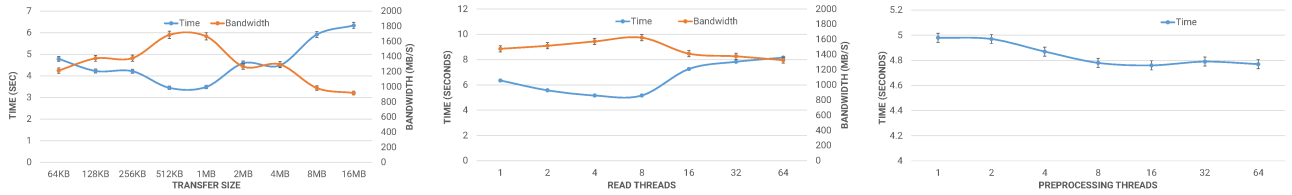
In the last sub-section, we showcased how the DLIO can accurately represent the real application's I/O behavior. We can use this to exhaustively test optimizations on the DLIO benchmark, which can be later transferred to the real application. Based on the underlying configuration of the application, we identify several opportunities for optimization within the workload. We explore the values for each optimization to understand their quantitative effect on the workload. We run the optimizations test on an eight-node configuration and calculate the metrics across all the processes.

1) TFRecord Workloads: Imagenet and Cosmoflow benchmark utilize `tf.data` APIs to read TFRecord data from the file system. The TensorFlow data pipeline provides three optimizations to improve the performance of the data pipeline, namely: a) *Transfer Buffer Size*: the granularity of I/O (in bytes) to be performed on each request. b) *Read Parallelism*: the number of threads per process to be used to perform a



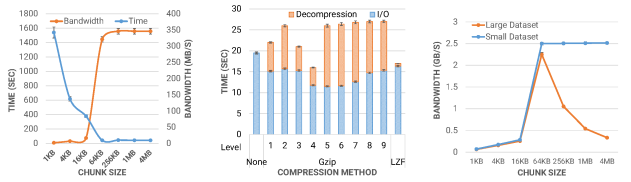
(a) Transfer size (b) Read Parallelism (c) Preprocessing Parallelism

Fig. 9. Imagenet Optimizations: shows the impact of changing transfer size, read threads, and preprocessing parallelism on the Imagenet Benchmark. Configurations: Each image is 256 KB in size with a batch size of 512 images. Figure a) showcases the optimal bandwidth of 2 GB per sec at a transfer size of 1 MB (aligned to Lustre stripe size). Figures b) and c) show a read and preprocessing parallelism is best at eight threads.



(a) Transfer size (b) Read Parallelism (c) Preprocessing Parallelism

Fig. 10. Cosmoflow Optimizations: shows the impact of changing transfer size, read threads, and preprocessing parallelism on the Cosmoflow Benchmark. Configurations: Each image is 128 KB in size with a batch size of 512 images. Figure a) showcases the optimal bandwidth of 1.8 GB/s at a transfer size of 1 MB (aligned to Lustre stripe size). Figures b) and c) show a read and preprocessing parallelism is best at eight threads.



(a) Chunk Size (b) Data Compression (c) Chunk Size (1 node)

Fig. 11. Neutrino and Cosmic Tagging with UNet Optimizations: Configuration: Each image is 50 KB in size with a batch size of 32 images. Figure a), we see an increase in I/O bandwidth at 64 KB chunk size. Figure b), we see the performance is best with gzip at compression level 4. Figure c) shows on a single node that increasing chunk size for large dataset which cannot fit in memory degrades performance after ideal chunk size.

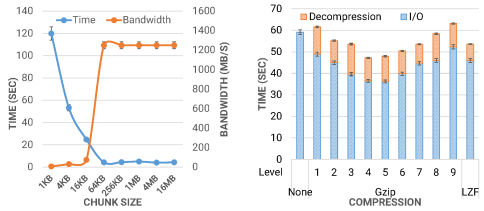
parallel reading. Finally, c) *Preprocessing Parallelism*: the number of threads per process to utilize for parallelizing data operations such as decoding and transformations. Note, in real-applications the model computations might utilize certain number of threads. We should set the I/O threads while considering exiting computation threads. We run the DLIO configurations of Imagenet and Cosmoflow with each of these parameters, calculate the overall time (in seconds), and bandwidth achieved per operation. We present the results in Figures 9 and 10, where the x-axis represents the values of the metric tested, the y-axis shows time elapsed in seconds, and the y2-axis shows the bandwidth achieved in MB/s.

In Figures 9(a) and 10(a), we observe the achieved bandwidth increases as we increase the the data transfer size. This bandwidth increase is because the Lustre file system can serve bigger requests with higher bandwidth than multiple smaller requests. This behavior can be seen for both the applications where 1 MB of transfer size achieves the best bandwidth of around 2 GB/s for both applications, resulting in the lowest I/O times. This results into a performance improvement of 1.25x and 1.32x for Imagenet and Cosmoflow over the default value of 256 KB which is set within TensorFlow Data Pipeline. However, we observe that the improvement trends vary slightly of the two applications where Imagenet has a much steeper rise and fall than Cosmoflow. This is because the record size and batch size match the 1 MB transfer buffer size precisely for Imagenet as opposed to Cosmoflow, where it's a multiple of several Megabytes. The transfer size should be set to be a multiple of record-size, batch-size, and the Lustre stripe unit. This will enable the applications to get a batch of images through one or more stripes and enable efficient reading from the PFS.

In sub-figures 9(b) and 10(b), we see that as we increase the number of threads for reading files per process, the time taken to read decreases and then starts increasing again. This is because increasing the number of reading threads initially increases parallelism but later introduces metadata interference on the Lustre filesystem, which starts hurting the performance. In both cases, we see read parallelism of 8 threads gives the ideal read performance of 2 GB/s read bandwidth. This would result in a performance improvement of 1.2x for both applications over the default value of one thread.

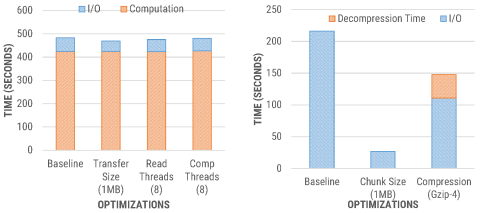
In sub-figures 9(c) and 10(c), we see that as we increase the number of threads to parallelize data preprocessing, the time taken by the application decreases. After a point in both applications (specifically eight threads), the read time becomes constant. This is because, at eight threads per process, the preprocessing task is parallelized and is fast enough for an efficient data pipeline. This results in a performance gain of 66% and 4% in Imagenet and Cosmoflow over default value of 1 thread.

2) HDF5 Workloads: The Neutrino and Cosmic Tagging with UNet and Distributed Flood Filling Networks applications utilize the HDF5 interface to perform I/O. This data format



(a) Chunk Size (b) Data Compression

Fig. 12. Distributed Flood Filling Networks Optimizations: Configuration: Each image is 32 KB in size with a batch size of one image. Figure a), we see an increase in I/O bandwidth for 128 KB chunk size. Figure b), we see the best performance achieved with gzip at compression level 4.



(a) Cosmoflow (b) Cosmic Tagger

Fig. 13. Applying Optimization: We apply the optimization explored from benchmark to actual application. Both applications show a similar effect as the benchmark. Figure a) shows Cosmoflow with a speedup of 1.33x for transfer size of 1 MB, 12% for using 8 read threads, and 8% for 8 preprocessing threads. Figure b) shows Cosmic Tagger with a speedup of 8x by changing chunk size to 1 MB and a speedup of 1.35x for utilizing gzip level 4.

represents a collection of images stored within the dataset of an HDF5 file. The HDF5 file natively supports two data access optimizations: a) *Dataset Chunking:* where a dataset is divided into fixed-sized chunks and I/O is performed on the chunk granularity from the file system, and b) *Data Compression:* the HDF5 library supports several data compression libraries with different compression levels such as GZip, and LZF. We run the DLIO benchmark with different parameters of these two optimizations and showcase the results in Figure 11 and 12. In these figures, the x-axis shows the various configurations of the two optimizations tested, the y-axis shows the time elapsed in seconds, and y2-axis shows the bandwidth achieved in MB/s.

In Figure 11(a) and 12(a), we observe that as we increase the chunk size, the I/O time reduces, after which it becomes constant. For both the applications, we see a steep increase in bandwidth at 64 KB and 32 KB for Cosmic Tagger and FFN, respectively. This is due to the alignment of record length of image and batch size with the chunk size and the stripe size of the Lustre file system. A chunk size of the application should be set equal to the size of the image/record times the batch size. This will enable an efficient reading pattern within the application. Additionally, the Lustre stripe size should be a multiple of the chunk size, to enable aligned I/O and efficient locality caching. This ideal chunk size boosts both applications' performance by almost 96x. Finally Figure 11(c) shows that increasing chunk size for small datasets does not degrade the performance even when shuffling is enabled. This is because the dataset can fit in compute nodes memory. However, for shuffling of samples with large datasets (i.e., datasets that cannot fit in the memory of a compute node) we see a degradation in I/O performance (i.e., performance reduction of 6.7x for 4 MB) after 64 KB chunk size.

In Figure 11(b) and 12(b), we observe that applying compression, in most cases, reduces the overall time elapsed as it reduces the amount of I/O that is performed. However, there is a trade-off between how much compression is applied in both applications and the reduction of I/O. As we can see, a compression level of 4 and 5 achieve the lowest time elapsed for the two applications. This results in a performance gain of 1.35x over no compression for both the applications. If we choose a heavier compression ratio, the decompression time outweighs the benefit gained by reducing I/O, resulting in worse performance than not compressing or lightly compressing. In general, the effectiveness of data compression depends on the distribution of data within the dataset [45], [46] which has to be fine-tuned for each application.

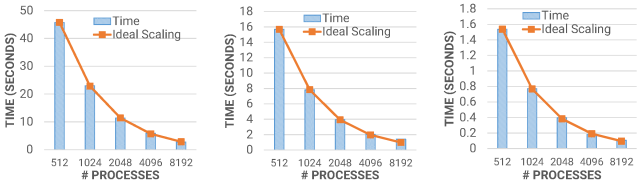
D. Real Application Optimization

On exploring optimizations with DLIO for various applications, We showcase two examples of applying them back to the real application. We choose CosmoFlow as a representative of the TensorFlow Data Pipeline workload and Cosmic Tagger as a representative of the HDF5 workload. We apply the optimizations we explored for both applications directly and observe its impact on I/O performance. Specifically, we apply CosmoFlow with optimization of transfer size of 1 MB (aligned to the Lustre's stripe size) and select read threads and preprocessing to be eight threads. For Cosmic Tagger, we re-align the dataset with a new chunk size of 1 MB and apply the GZip compression of level 4. These values are derived directly from the previous section.

Figure 13 shows the results. The results of the optimization for both applications are similar to the results in the DLIO benchmark. For Cosmoflow, we observe that the transfer size of 1 MB optimizes the I/O time by 1.33x, whereas adding additional read threads and computation threads improve data access by 12% and 7%, respectively. Similarly, for Cosmic Tagger re-aligning the chunk size from 4KB to 1MB gives a speedup of 8x over the I/O access. Additionally, utilizing gzip with compression level 4 can optimize I/O access by 1.35x (consistent with the DLIO benchmark optimizations). These results show the ability of DLIO to accurately represent the scientific DL workloads and a cheap mechanism to test optimization, which can be later transferred to the applications.

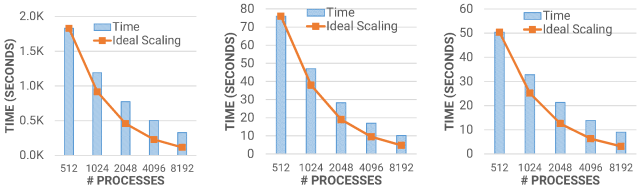
E. DLIO Benchmark Scaling

This sub-section uses the ideal configuration parameters (including the optimizations) from the previous section and scales our benchmark code from 128 nodes to 2048 nodes. We use four processes per node. Hence, we scale our processes from 512 to 8096 processes. In each of the cases, we measure the overall time elapsed in seconds. We can categorize the six applications into one file per process and a single shared file case.



(a) Imagenet (b) Cosmoflow (c) FRNN

Fig. 14. Multiple File applications: We perform strong scaling of applications, we observe in all the sub-figures that, applications divide the data and files equally among the processes and scale linearly in performance.



(a) Cosmic Tagger (b) FFN (c) CANDEL

Fig. 15. Shared File applications: We perform strong scaling of application, we observe in all figures that the applications divide the data and hence the performance is improved. However, the performance is limited by the contention on metadata due uncoordinated data access.

1) One File Per Process: In this category, we have Imagenet Benchmark, Cosmoflow Benchmark, and Fusion Recurrent Neural Net application. In each of these cases, we keep the overall data the same and scale the number of processes (i.e., strong scaling). In each of these applications the files are divided equally among the methods. Hence, when we increase the number of processes, the number of files per process (i.e., the amount of I/O) decreases linearly. Therefore, for all the cases, as shown in Figure 14, we see the time sub-linearly decreases with an increasing number of processes. In multiple files per process case, there is an equal division of files among processes, and hence the overall time will decrease. We see that the applications' I/O scale linearly.

2) Shared File Per Process: In this category, we have Cosmic Tagger Application, Distributed FFN, and CANDEL Benchmark. In all of these cases, the processes access a single shared file and train over a portion of the data. The size of dataset is fixed (i.e., strong scaling). The results are shown in Figure 15. We observe an increase in performance (i.e., decrease in time) on strong scaling the applications. However, the decrease of time is not close to the ideal as seen in multi-file case. This is because when multiple processes access a shared file, the interference of reading the metadata starts dominating the overall I/O time [47]. This is due to lack of any co-ordination between data access. We tested to verify that the metadata contention is only present in larger scales.

VI. Conclusions

The emergence of deep learning (DL) techniques has dramatically improved scientific exploration and discoveries. Scientists have studied the computational aspects of DL applications in scientific domains in great detail. However, the I/O behavior is not well understood yet. As DL applications consume massive amounts of data, understanding the patterns with which they perform I/O is crucial for the overall efficiency of the process. In this work, we characterize different aspects of I/O across several scientific DL applications. We provide a detailed methodology and analysis to describe this behavior. Additionally, we propose a Deep Learning I/O (DLIO) benchmark , which encapsulates all the different aspects of DL applications under one hood. Our results showcase our DLIO's accuracy and how it can be utilized to optimize the I/O behavior of applications by 1.35x through existing optimizations in the TFRecord and HDF5 libraries. Finally, we showcase how the I/O access in scientific formats such as HDF5 do not scale well due to their lack of optimizations in DL space. We envision building a middleware solution that would accelerate I/O for scientific data formats in DL applications.

Acknowledgment

We would like to thank the application developers including Sam Foreman, Rafael Vescovi, Corey Adams, Murali Emani, Kyle Gerard Felker, Xingfu Wu, and Murat Keceli for providing and running the workloads on Theta. This work used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility under Contract DE-AC02-06CH11357 and is supported in part by National Science Foundation under NSF, OCI-1835764 and NSF, CSR-1814872.

References

[1] Z. Jiang, W. Gao, L. Wang, X. Xiong, Y. Zhang, X. Wen, C. Luo, H. Ye, X. Lu, Y. Zhang *et al.*, "HPC AI500: a benchmark suite for HPC AI systems," in *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 2018, pp. 10–22.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *IEEE Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[4] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[5] C. J. Adams, "Neutrino and Cosmic Tagging with UNet," 2015. [Online]. Available: https://github.com/coreyjadams/CosmicTagger

[6] W. Dong, M. Keceli, R. Vescovi, H. Li, C. Adams, E. Jennings, S. Flender, T. Uram, V. Vishwanath, N. Ferrier *et al.*, "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)*. IEEE, 2019, pp. 52–61.

[7] G. B. Goh, N. O. Hodas, and A. Vishnu, "Deep learning for computational chemistry," *Journal of computational chemistry*, vol. 38, no. 16, pp. 1291–1307, 2017.

[8] 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*, 2019, pp. 1–11.

[9] G. Dong, K. G. Felker, A. Svyatkovskiy, W. Tang, and J. Kates-Harbeck, "Fully Convolutional Spatio-Temporal Models for Representation Learning in Plasma Science," *arXiv preprint arXiv:2007.10468*, 2020.

[10] S. W. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. Laure, "Characterizing deep-learning I/O workloads in TensorFlow," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE, 2018, pp. 54–63.

[11] Y. Kwon and M. Rhu, "Beyond the memory wall: A case for memory-centric hpc system for deep learning," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 148–161.

[12] H. Devarajan, A. Kougkas, P. Challa, and X.-H. Sun, "Vidya: Performing Code-Block IO Characterization for Data Access Optimization," in *25TH IEEE International conference on High Performance Computing, data, and analytics*, no. 2640-0316, 2018, pp. 255–264.

[13] 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)*, 2016, pp. 265–283.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[15] J. Gu, Y. Liu, Y. Gao, and M. Zhu, "OpenCL caffe: Accelerating and enabling a cross platform machine learning framework," in *Proceedings of the 4th International Workshop on OpenCL*, 2016, pp. 1–5.

[16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[17] V. Leon, S. Mouselinos, K. Koliogeorgi, S. Xydis, D. Soudris, and K. Pekmestzi, "A TensorFlow Extension Framework for Optimized Generation of Hardware CNN Inference Engines," *Technologies*, vol. 8, no. 1, p. 6, 2020.

[18] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, "Benchmarking machine learning methods for performance modeling of scientific applications," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2018, pp. 33–44.

[19] 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)*. IEEE, 2020, pp. 62–72.

[20] ——, "HReplica: A Dynamic Data Replication Engine with Adaptive Compression for Multi-Tiered Storage," in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 256–265.

[21] A. Kougkas, H. Devarajan, and X.-H. Sun, "I/O acceleration via multi-tiered data buffering and prefetching," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 92–120, 2020.

[22] ——, "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*, 2018, pp. 219–230.

[23] O. B. Messer, E. D'Azevedo, J. Hill, W. Joubert, M. Berrill, and C. Zimmer, "MiniApps derived from production HPC applications using multiple programing models," *The International Journal of High Performance Computing Applications*, vol. 32, no. 4, pp. 582–593, 2018.

[24] J.-H. Tao, Z.-D. Du, Q. Guo, H.-Y. Lan, L. Zhang, S.-Y. Zhou, L.-J. Xu, C. Liu, H.-F. Liu, S. Tang *et al.*, "Bench ip: Benchmarking intelligence processors," *Journal of Computer Science and Technology*, vol. 33, no. 1, pp. 1–23, 2018.

[25] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, "Fathom: Reference workloads for modern deep learning methods," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[26] W. Gao, F. Tang, L. Wang, J. Zhan, C. Lan, C. Luo, Y. Huang, C. Zheng, J. Dai, Z. Cao *et al.*, "AIBench: an industry standard internet service AI benchmark suite," *arXiv preprint arXiv:1908.08998*, 2019.

[27] 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*, 2019, pp. 1–10.

[28] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable Deep Learning via I/O Analysis and Optimization," *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 2, pp. 1–34, 2019.

[29] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, "BenchNN: On the broad potential application scope of hardware neural network accelerators," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2012, pp. 36–45.

[30] S. Narang and G. Diamos, "DeepBench," 2016.

[31] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang *et al.*, "MLPerf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, pp. 8–16, 2020.

[32] S. Dong and D. Kaeli, "Dnnmark: A deep neural network benchmark suite for gpus," in *Proceedings of the General Purpose GPUs*, 2017, pp. 63–72.

[33] 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)*. IEEE, 2018, pp. 145–156.

[34] K. Serizawa and O. Tatebe, "Accelerating Machine Learning I/O by Overlapping Data Staging and Mini-batch Generations," in *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, 2019, pp. 31–34.

[35] D. Levy, M. D. Hoffman, and J. Sohl-Dickstein, "Generalizing Hamiltonian Monte Carlo with Neural Networks," *arXiv preprint arXiv:1711.09268*, 2017.

[36] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook *et al.*, "CosmoFlow: Using deep learning to learn the universe at scale," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 819–829.

[37] H. Devarajan, "VaniDL: DL analyzer tool," 2020.

[38] ALCF, "Theta Machine Overview," 2020. [Online]. Available: https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview

[39] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica *et al.*, "Exascale deep learning for climate analytics," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 649–660.

[40] Tensorflow, "TensorFlow benchmarks," 2018. [Online]. Available: https://github.com/tensorflow/benchmarks

[41] H. Devarajan and H. Zheng, "VaniDL Analyzer for Deep Learning Workloads," 2020. [Online]. Available: https://github.com/hariharan-devarajan/vanidl

[42] DeepLearnPhysics, "LArCV (Version 3)," 2020. [Online]. Available: https://github.com/DeepLearnPhysics/larcv3

[43] S. Saini, J. Rappleye, J. Chang, D. Barker, P. Mehrotra, and R. Biswas, "I/O performance characterization of Lustre and NASA applications on Pleiades," in *2012 19th International Conference on High Performance Computing*. IEEE, 2012, pp. 1–10.

[44] H. Devarajan, "DLIO: Scientific Deep Learning I/O Benchmark," 2020. [Online]. Available: https://github.com/hariharan-devarajan/dlio_benchmark

[45] H. Devarajan, A. Kougkas, L. Logan, and X.-H. Sun, "HCompress: Hierarchical Data Compression for Multi-Tiered Storage Environments," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 557–566.

[46] H. Devarajan, A. Kougkas, and X.-H. Sun, "Ares: An Intelligent, Adaptive, and Flexible Data Compression Framework," in *2019 IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid 2019)*, 2019.

[47] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead, "Harmonia: An interference-aware dynamic I/O scheduler for shared non-volatile burst buffers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 290–301.