2018 IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

# Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems

Yue Zhu† Fahim Chowdhury† Huansong Fu† Adam Moody‡ Kathryn Mohror‡ Kento Sato‡ Weikuan Yu†
†Florida State University ‡Lawrence Livermore National Laboratory
{yzhu,fchowdhu,fu,yuw}@cs.fsu.edu {moody20,kathryn,sato5}@llnl.gov

***Abstract*—Deep neural networks have recently gained tremendous interest due to their capabilities in a wide variety of application areas such as computer vision and speech recognition. Thus it is important to exploit the unprecedented power of leadership High-Performance Computing (HPC) systems for greater potential of deep learning. While much attention has been paid to leverage the latest processors and accelerators, I/O support also needs to keep up with the growth of computing power for deep neural networks. In this research, we introduce an entropy-aware I/O framework called DeepIO for large-scale deep learning on HPC systems. Its overarching goal is to coordinate the use of memory, communication, and I/O resources for efficient training of datasets. DeepIO features an I/O pipeline that utilizes several novel optimizations: RDMA (Remote Direct Memory Access)-assisted in-situ shuffling, input pipelining, and entropy-aware opportunistic ordering. In addition, we design a portable storage interface to support efficient I/O on any underlying storage system. We have implemented DeepIO as a prototype for the popular TensorFlow framework and evaluated it on a variety of different storage systems. Our evaluation shows that DeepIO delivers significantly better performance than existing memory-based storage systems.**

## I. Introduction

Artificial intelligence (AI) and machine learning (ML) applications have gained wide-spread prominence, particularly because of the employment of powerful neural networks in various application domains such as computer vision, image classification [33], and natural language processing.

While neural networks with small datasets such as NORB [35], Caltech-101/256 [20], [24], CIFAR-10/100 [32], and MNIST [34] can be solved efficiently using a small group of computation tasks, many shortcomings of small datasets for training have been recognized in prior works. It has been observed that small datasets fail to reflect real-world variations and reduce the generalization of other datasets, as each object can produce a wide range of images based on different poses, positions, and scales [39], [45]. To meet the need of generalization, neural networks (e.g, GoogleNet [44], ResNet [26]) with larger filters and more weights are adopting deeper layerswith larger datasets (LabelMe [40], ImageNet [19]), because large datasets can prevent deep neural network (DNN) from overfitting during training by utilizing more iterations to update an increased number of weight parameters [43].

While there have been many existing efforts to enable deep neural networks to leverage the powerful CPU and GPU processors from leadership high-performance computing (HPC) systems, large-scale deep learning with larger datasets requires efficient I/O support from the underlying file and storage systems. For example, deep learning frameworks such as TensorFlow [18] and Caffe [28] need to read datasets from backend storage systems during training. In TensorFlow, datasets are fetched from different platforms (e.g., HDFS [41], POSIX-like file systems). Besides POSIX-like file system, Caffe also supports other storage systems such as LMDB/LevelDB [6]. In these deep learning frameworks, in order to achieve a high level of accuracy in the training model, datasets often have to be read from the backend storage multiple times in a random order. The randomization process, often called *shuffle*, is important to avoid bias and update parameters of the training model efficiently.

This I/O pattern has led to inefficiency in reading large datasets from backend storage for many DNN training frameworks. For some clusters that have node-local storage devices (e.g., SSD), the dataset size is limited to the size of the storage devices. Furthermore, on a GPU cluster, to meet the training speed, the read bandwidth of devices is expected to be relatively high. An alternative approach for the clusters without node-local storage devices is to leverage traditional parallel file systems (e.g, Lustre [11], BeeGFS [4]). However, the read bandwidth of a parallel file system depends highly on the size of the dataset. Small datasets can easily be "cached" locally by parallel file systems for multiple reads whereas large datasets cannot fit in the file system cache. For example, our experiments show that on Cab [5] cluster at Lawrence Livermore National Laboratory (LLNL), which uses Lustre as the backend file system, TensorFlow's reading bandwidth for a dataset of 1 TB is less than half of that for a dataset of 32 GB.

In this research, we propose an efficient I/O framework for large-scale deep learning on HPC systems. Our main objective is to coordinate the use of memory, communication, and I/O resources for efficient training. To this end, we design and implement an entropy-aware I/O pipeline for TensorFlow. In addition, to overcome the performance impedance of TensorFlow dataset API, we design a portable storage interface so that efficient I/O for deep learning can be enabled across a wide variety of underlying file and storage systems.

Specifically, we make the following contributions:

- We design DeepIO, an I/O framework for training deep neural networks.

2375-0227/18/$31.00 ©2018 IEEE
DOI 10.1109/MASCOTS.2018.00023



IEEE computer society

- We implement DeepIO as a prototype for TensorFlow featuring novel optimizations: RDMA-assisted in-situ shuffling, input pipelining, and entropy-aware opportunistic ordering.
- To overcome the overhead of the TensorFlow dataset API, we develop a portable API for TensorFlow to leverage DeepIO on different storage systems.
- We conduct a systematic set of evaluation tests that show DeepIO improves the I/O bandwidth by at least 6.12 times and 1.17 times, respectively, compared with a memory-based BeeGFS and Octopus [38], an RDMA-based distributed persistent memory file system.

## II. Background & Motivation

In this section, we review the background of using large datasets in DNN and discuss the motivation for DeepIO.

### A. DNN with Large Datasets

When training deep neural networks, large datasets are commonly used since they represent diverse real-world scenarios. On HPC systems, the datasets can be placed on node-local devices or parallel file systems such as Lustre [11] and BeeGFS [4]. For example, Catalyst [8] at LLNL is equipped with 800 GB node-local SSDs on every compute node. However, the SSD devices with fast read speed and large capacity are quite expensive and not available on all clusters. Therefore, parallel file systems are a feasible choice for users on HPC systems without node-local storage devices.

When dealing with datasets on parallel file systems, the sizes of the datasets have a high impact on the reading speed. We use BeeGFS as an example to illustrate this. If the size of a dataset is relatively small, it can be cached in the memory of BeeGFS clients or Object Storage Servers (OSSs). In this way, after the first epoch of a DNN training, the dataset will always be read from "cache" instead of being fetched from physical storage devices. However, if the size of a dataset is large and cannot be cached, some data must be fetched from physical storage devices, since all images must be read in each training epoch. The slow reading speed can significantly prolong the training time.

To demonstrate the impact of dataset size on parallel file system performance, we use IOR [10] to measure the maximum read bandwidth (N-to-N sequential read) of BeeGFS with 16 clients on our in-house cluster (the system configuration is described in Section IV-A). In our examination, every node reads 512 MB from BeeGFS (8 GB in total) for measuring the impact of small datasets, and 10 GB (160 GB in total) for measuring the impact of large datasets. The aggregated read bandwidth for small datasets is 7411.98 MB/s, and 4662.49 MB/s for large datasets. As stated previously, the bandwidth of reading small dataset is much greater than reading large dataset since the small dataset can benefit from the OSS's and BeeGFS clients' caches.

### B. DNN Training Algorithms

There are several algorithms for optimizing training parameters of deep neural networks (DNN). Gradient descent is one of the most popular algorithms. It uses prediction error to update the parameters of models to reduce the error in the next round. Batch gradient descent, stochastic gradient descent, and mini-batch gradient descent are several variants of gradient descent algorithms [1]. The difference among these three is how frequently the parameters are updated: after processing every element of a dataset (stochastic gradient descent), after processing the entire dataset (batch gradient descent), or after processing a few elements (mini-batch gradient descent). The mini-batch gradient, often referred to as SGD (Stochastic Gradient Descent) [3], is more commonly used because it requires less memory and leads to faster convergence speed. However, using SGD as the optimizer of a model requires the sequence of input elements being randomly shuffled. This is to avoid the model being biased by the noise of the input order.

### C. Challenges from Large-Scale Deep Learning

With the growing size of DNN datasets, the training time has been increasing as well. As described in the previous section, SGD is one of the most popular algorithms for deep learning. SGD allows weights and coefficients to be updated more efficiently, by processing training samples in batches instead of individually. To overcome the performance challenges caused by large datasets, strategies such as distributed training with large mini-batches are typically employed. Different mini-batch sizes have been observed in practical training.

The batch size for a mini-batch is critical in terms of training speed and accuracy. A small batch size leads to less computation in one iteration but can be more easily affected by noises during the training process. A larger batch size reduces the number of iterations per epoch, but it may cause the training model to be less likely to converge. Although very large batch size may have less competitive performance [30], many recent studies have shown that cleverly enabling large minibatch helps both training performance and training accuracy [31], [37], [23], [47]. In addition, when training with SGD, it requires the training dataset to be shuffled randomly before each training epoch. This prevents possible overfitting of the neural networks from inputting samples in a known order. It remains a challenge on how to efficiently generate large mini-batches for distributed DNN models while maintaining the randomness of input datasets for accuracy assurance.

Deep learning frameworks such as TensorFlow and Caffe support multiple file formats such as batched binary files and raw images. When training with raw images, a massive amount of small random reads are issued to parallel file systems to offer full randomization while organizing mini-batches. Therefore, simply enabling a large mini-batch on raw images causes relatively low performance due to the random small reads. When training with batched binary files, TensorFlow can perform sequential reads, but the randomness of its input datasets is highly dependent on the size of shuffling buffer in the training framework, which is discussed in Section II-D.

Furthermore, using large mini-batches in DNN on HPC systems requires a good match of performance between computation and I/O during the training. As mentioned previously,

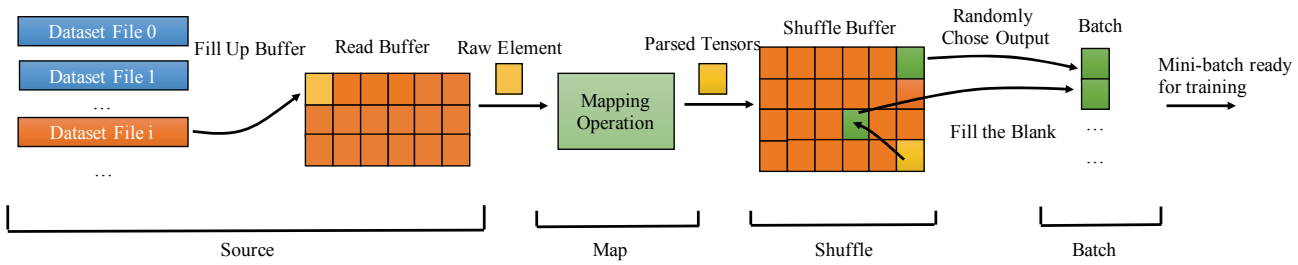



Fig. 1: Data flow in TensorFlow Dataset API.

some large datasets have to be stored on parallel file systems due to the HPC systems' limitation. Although prefetching from parallel file systems can be enabled to overlap the loading with computation time, the read performance is still not able to match the fast computation speed, especially with powerful training devices, such as GPUs. For example, Goyal et al. [23] report that the training time of one iteration is as low as 0.26 second with 11K images per mini-batch on 44 nodes. To match such a speed, the estimated mini-batch producing speed needs to be greater than 6 GB/s. And if the size of the dataset is relatively large, it becomes difficult to deliver the estimated reading speed when reading dataset through a training framework from parallel file systems, since the training framework can add additional overheads. Therefore, there is a genuine need for a specialized, high-performance I/O solution that can construct highly-randomized large mini-batches for the DNN training.

### D. Data Flow in TensorFlow Dataset API

As data fetching and randomization procedures are important for deep learning, we describe how mini-batches are prepared for DNN training in TensorFlow.

TensorFlow provides `tf.data` API to enable data importing [14]. The `tf.data` API simplifies importing data from files in various formats (e.g., text files, raw images, zip files), ensures randomization of the files, and transforms them into batches. The `tf.data` API introduces several stages (e.g., *Source*, *Map*, *Shuffle*, *Batch*) for importing a dataset. Normally mini-batches are generated in the following order: *Source* → *Map* → *Shuffle* → *Repeat* → *Batch*. Note that the *Shuffle* step is optional and a TensorFlow application can opt to read the dataset elements in a specified order.

Fig. 1 shows these basic stages of reading files in TensorFlow (excluding the *Repeat* stage). First, the *Source* stage retrieves elements from files and stores them in a read-in data buffer. Then, the *Map* stage transforms the elements, e.g., decodes raw values into a three-dimensional pixel value tensor. The *Shuffle* stage inserts the new elements in the shuffle buffer, typically appending them to the end of the buffer. Then the output of the *Shuffle* stage is randomly chosen from the shuffle buffer for the *Batch* stage. The elements at the tail of the shuffle buffer are moved to the indices of the randomly chosen elements to fill the resulting holes. Finally, at the *Batch* stage, TensorFlow accumulates *N* elements from the previous stage (*Shuffle* in our example) to a mini-batch, and copies them into a contiguous memory allocation. *N* is a parameter specified by the user. Once a mini-batch is ready, it is used by the rest of the TensorFlow training model.

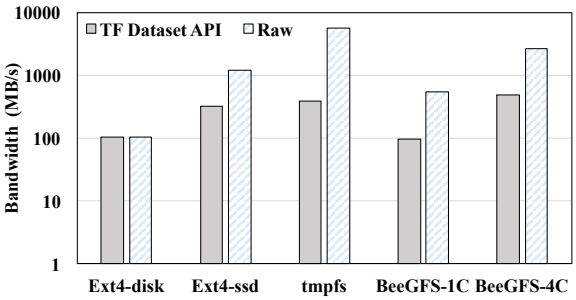



Fig. 2: Loading speed of TensorFlow Dataset API.

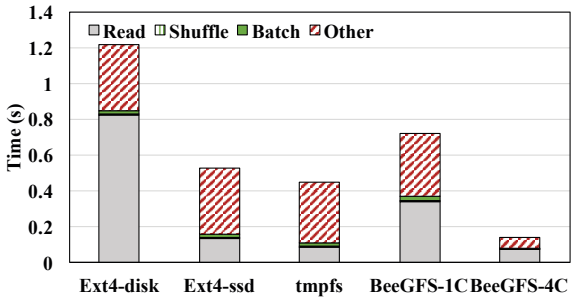



Fig. 3: Time breakdown of TensorFlow Dataset API.

We have used one of the `tf.data` API (i.e., `tf.data.FixedLengthRecordDataset`) to examine the I/O performance of TensorFlow Dataset API with the Cifar10 training dataset, whose total size is 146.48 MB and fixed element size is 3 KB. We measure the fixed length dataset read speed with *Source* stage, *Shuffle*, and *Batch* stage (batch_size = 128) on disk (*Ext4-disk*), SSD (*Ext4-ssd*), memory (*tmpfs*), 1 BeeGFS client (*BeeGFS-1C*), and 4 BeeGFS clients (*BeeGFS-4C*). We compare the performance of the API with the raw performance of these file systems (denoted as *Raw* in Fig. 2). As shown in Fig. 2, the performance on the Ext4-disk system is similar to that of the `tf.data` API because the slow disk speed overshadows any differences. However, the performance of TensorFlow Dataset API on Ext4-ssd and tmpfs are much lower than their raw performance, which is caused by the overheads introduced by the TensorFlow API.

Moreover, with BeeGFS-1C, the `tf.data` API reports 46.8% of the raw sequential read bandwidth. With BeeGFS-4C, `tf.data` achieves 52.31% of the aggregate sequential read performance. This is because the same amount of data is read concurrently in both cases and when four clients are used, there are less `tf.data` function calls for each client. One reason that `tf.data` API's mini-batch generation speed is much lower than the file systems' raw performance is that the `tf.data` API introduces several memory copies in its data path. In one epoch, the dataset has to be copied to a read buffer for improving the read performance, and an additional memory copy is also necessary when batching elements for next layers.

We show the breakdown of time spent in each stage in Fig. 3. In the `tf.data` API, very little time is spent performing the actual functionality of each stage, and is instead spent on reading and other overheads. One overhead is due to the fact that the core of `tf.data` API and TensorFlow is implemented in C++. While TensorFlow provides wrappers to execute C++ code in python, the cost of executing the wrappers to invoke the C++ code is not trivial when the total execution time is short. Furthermore, the `tf.data` API acquires iterators to repetitively generate input data for training. Dispatching and scheduling a thread from the threads pool for the iterators also incur non-negligible overheads.

## III. Design

The design of DeepIO incorporates several techniques to improve the I/O performance of distributed DNN training frameworks. To remove the overhead of reading datasets from backend storage, we design DeepIO to be an ephemeral, in-memory storage system that is co-located with a distributed DNN training application. By retaining datasets in memory, we are able to employ additional techniques to improve performance including RDMA data transfers, overlapping I/O operations with training iterations, relaxing order of records retrieval, and the DeepIO API for retrieving records.

In Fig. 4 we show the comparison of data flow between the original TensorFlow and DeepIO with TensorFlow. Fig. 4(a) shows the original TensorFlow data flow. Here, in each epoch, every TensorFlow worker reads elements from a shard of the dataset located on the backend storage using the TensorFlow Dataset API. The element is a training image, which can be a raw image (e.g., .JPG file), binary data (e.g, pixel value of an image), etc. The element means the pixel value of an image in the following sections if it is not specified. Then, the elements are organized into mini-batches via the `tf.data` API.

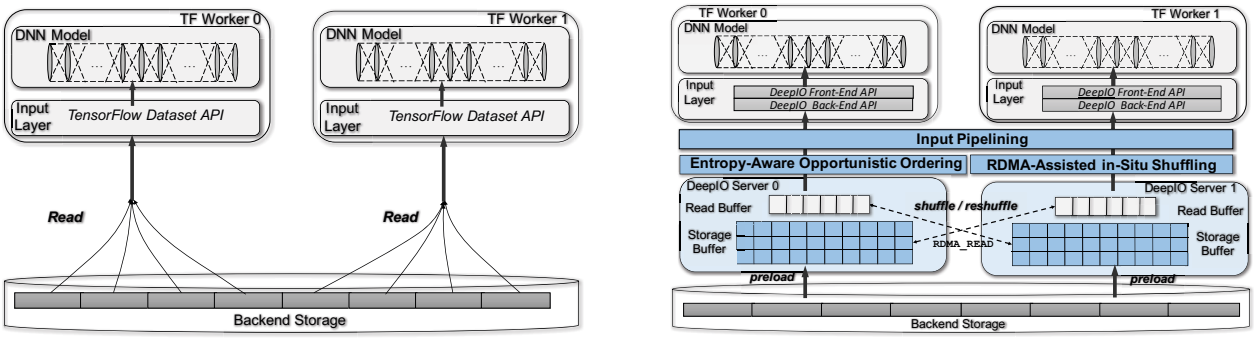
With DeepIO, as shown in Fig. 4(b), dataset elements are loaded from the backend storage into the memory of DeepIO servers that run on each compute node. TensorFlow workers access the elements using the DeepIO API. The DeepIO servers employ several optimizations to return elements to the workers with high performance: *RDMA-assisted in-situ shuffling*, *input pipelining to hide I/O latency*, and *entropy-aware opportunistic ordering*. In the RDMA-assisted in-situ shuffling, datasets are buffered in each server's local memory and exposed to other DeepIO servers for `RDMA_READ` operations. Note that, since the dataset is now retained in memory, elements can be easily re-shuffled between the participating nodes for the next epoch without reloading dataset from back-end storage. The input pipelining reduces the I/O waiting time of workers by overlapping training with mini-batch generation. For the entropy-aware opportunistic ordering, we observe that the order of generated mini-batches is not important as long as it is randomized rather than delivered in particular order. Our approach is to relax the strict ordering requirements to improve I/O performance while still maintaining the level of randomness required by the training algorithm. To estimate the level of randomness, DeepIO leverages the notion of *cross-entropy* in the shuffling procedure. Cross-entropy is a measure of how one probability distribution diverges from a second expected probability distribution [16]. We use cross-entropy as a measure of the difference between our relaxed ordering and the fully-shuffled scheme on the probabilities of occurrence for a sequence of data elements.

To incorporate DeepIO into TensorFlow, we also introduce a portable API which also enables any backend storage system to be used for loading datasets with high performance. The detail of the API is discussed in Section III-D.

### A. RDMA-Assisted In-Situ Shuffling

DeepIO stores the training dataset in the memory of the distributed compute nodes and leverages RDMA for mini-batch generations. As shown in Fig. 5, both the storage buffer and the read buffer on DeepIO servers are exposed for RDMA data transfer. The storage buffer is used to store the dataset in memory, and the read buffer is used both as the destination of dataset shuffling and the shared memory communication conduit between a training worker and a DeepIO server for updating status of memory blocks. This is in contrast to the TensorFlow where workers sequentially read elements of a batched dataset and randomly choose elements for the shuffle buffer. In our approach, the shuffle operation occurs in-situ with the placement of the elements into the read buffer. Every block in the storage buffer and read buffer contains an element whose size is known to DeepIO.

*1) Memory-Resident Data Buffering:* When the DeepIO servers are launched, they establish RDMA connections between each other. The storage buffer on every node is allocated and exposed to `RDMA_READ`. If the whole dataset can fit in the memory available to DeepIO servers, each node will initialize an equal number of blocks that in total can hold all elements of the dataset. If not, the number of blocks initialized by the servers will be contingent upon a configurable user-defined ratio (e.g., 0.25) of the total dataset. The remaining portion of the dataset left on storage will be read and processed in a pipelined fashion, described later. Each DeepIO server will read a different partition of the dataset from backend storage into the storage buffer. Because the memory block size is equal to the element size, the element ID equals to the block ID when a dataset is all in memory. Therefore, any DeepIO server can




(a) Original TensorFlow.

(b) TensorFlow with DeepIO.

Fig. 4: Data flow of reading dataset for TensorFlow.

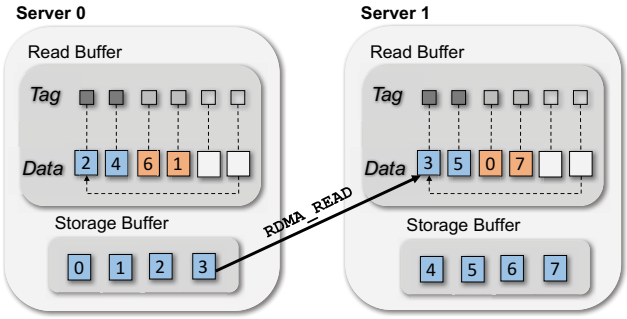



Fig. 5: RDMA-assisted in-situ shuffling.

locate a desired element in the storage buffer by simply using the block ID.

*2) RDMA-Assisted Shuffling:* To generate mini-batches, DeepIO servers first create a random list of element IDs using the same seed which is shared by broadcast at the beginning of each epoch. Each DeepIO server generates the same list of random IDs according to the same seed and then is assigned a unique partition of the IDs list. After that, each server will read its assigned elements from the storage buffer using `RDMA_READ` or local memory read, depending on the location of the element. The destination of an `RDMA_READ` or a local memory read is the local read buffer. The structure of the read buffer shown in Fig. 5 is designed to facilitate both data shuffling and pipelining. The read buffer contains *data* and an associated *tag* for each data element. The tag indicates which elements have been used by the training workers and which still need to be processed. The purpose of the tags is to indicate the status of elements to further avoid the need of copying the mini-batch elements to a separate buffer. The blue and the orange blocks in Fig. 5 are corresponding to the different element status. The blue blocks mean that the reading is finished and the data is ready to be used. The orange blocks imply that these blocks have been assigned to incoming elements.

An advantage of our approach is that it can remove the need for additional memory copies of the mini-batch elements, depending on the behavior of the training framework. For example, in Caffe, the memory data layer allows reading data directly from memory without memory copy [7], so the corresponding memory will not be released as long as the data is still in it. Therefore, the RDMA-assisted in-situ shuffling with zero-copy can be enabled when generating mini-batches. However, TensorFlow requires a specialized output tensor buffer in a TensorFlow operation [13]. The tensor buffer will be released as soon as the tensor is not needed anymore. In this case, an additional memory copy cannot be avoided for re-organizing the data into the output tensor buffer. Then, the generated mini-batches are ready for input to the DNN model, as shown in Fig. 4.

### *B. Input Pipelining*

To overlap disk I/O when the storage buffer is not able to hold the entire dataset, DeepIO forms a pipeline of mini-batches as shown in Fig. 6. To illustrate, we introduce two pipeline processes in our multi-level pipeline scheme: a *hybrid backend-memory pipeline* and an *in-memory pipeline*. The hybrid backend-memory pipeline is for overlapping the training iterations when the size of the storage buffer of DeepIO server is insufficient to hold the entire dataset and some elements must be retrieved from backend storage. The in-memory pipeline, indicated in Fig. 6 by the boxes with the dashed line, is a part of the hybrid backend-memory pipeline. It reads elements from the storage buffers of all participating DeepIO servers and batches them for workers.

When the entire dataset resides in memory, the in-memory pipeline moves elements from local and remote storage buffers in a pipelined manner, overlapping training with mini-batch preparation. During the in-memory pipeline, once the DeepIO server detects the completion of an element read (`RDMA_READ` or local memory copy), it updates the corresponding tag to the *read completed* state. After the data of the block is used or copied out by the training worker, the tag of the block is updated to *empty* state to mark it as available for a new element. The read buffer is viewed as a ring by the pipeline. The pipeline progresses across the buffer to the next available block to be replaced and wraps around from the tail of the buffer to the head when the end is reached.

Hybrid backend-memory pipelining is designed to reduce the overhead of constructing mini-batches when datasets can-

not be fully uploaded to the storage buffers. To allow the uploading time to be overlapped with the training time, we use a double buffering scheme. The read buffer is divided into two equal-sized buffers. In one buffer, elements are continually read in from backend storage in a sequential manner. Concurrently, the data in the other storage buffer partition is used for shuffling and mini-batch generation. Because the random elements chosen for the mini-batches come only from the in-memory buffers and not from storage in this mode, the size of the storage buffer may affect the randomization level of the mini-batches. We discuss this impact in the next section.

### C. Entropy-Aware Opportunistic Ordering

There are two modes to select elements for mini-batches in DeepIO, *ordered* and *entropy-aware opportunistic ordering*. In the ordered mode, the order of element retrieval is based on the requests submitted by the client in the case that they opt out of the shuffling step. However, in some training jobs, e.g., when using SGD for optimization, the input training elements are not required to be in a meaningful order. Therefore, the order of input elements is not important as long as the input order is randomized. Using this knowledge, we introduce the entropy-aware opportunistic ordering mode. Here, workers of training frameworks, also referred to as clients, are not aware of the input order and wait for the data prepared by the DeepIO servers.

In ordered access mode, input requests are a list of element IDs, which enables DeepIO servers to process a batch of elements all at once and significantly reduce the overheads of inter-process communication. There are two "cursors" to guarantee the access order following the requested order. The DeepIO server uses one cursor to indicate the first free data block, and workers (clients) use another cursor to indicate the next read position. The cursor used by the server moves forward only after the block is assigned to an incoming element. The issued read follows the submitted ID list. The cursor used by the client moves forward when the tag for that block marked as ready by the server. Therefore, the elements are guaranteed to be processed in order since the data blocks are assigned based on the input order by the DeepIO server. However, this strict ordering results in a massive number of small random reads from backend storage to the storage buffer when the entire dataset cannot fit in the memory, which leads to relatively low read bandwidth.

With entropy-aware opportunistic ordering, DeepIO servers independently determine which elements will be taken in next mini-batches. The algorithm for determining the elements for mini-batches in opportunistic ordering is designed to avoid excessive inter-process communication using a seed broadcasting method, and it avoids a large number of small random reads from backend storage by utilizing only the elements that are loaded into the in-memory storage buffers. The DeepIO servers receive element count and size of storage buffer for shuffling before training starts. Then before each epoch, seeds are broadcast to all servers which are used to generate random lists of memory block IDs. Because the seed used to generate

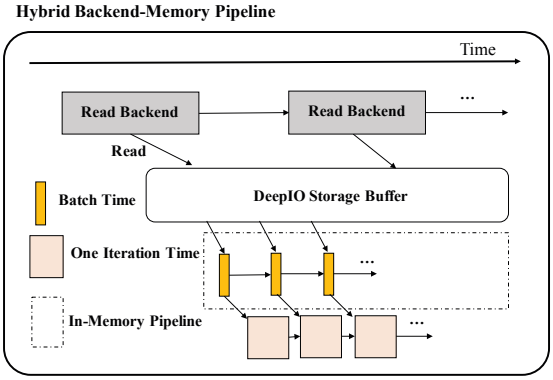


Fig. 6: Pipeline of importing dataset with opportunistic order.

the list is the same on all servers, the random list is identical across servers. Each DeepIO server uses a pre-assigned portion of the random list of memory block IDs for the mini-batch generation. Then, similar to the ordered access mode, each server issues element read requests according to the order of the IDs until the read buffer is filled up. By using input pipelining in DeepIO, the element read requests are overlapped with the training of the workers.

In DNN training process, randomization is ensured by shuffling input elements. The input order generation is similar to the events of randomly choosing elements from a dataset. According to the Information Theory, an unlikely event is more informative than a likely event [17]. Similarly, when training with shuffled input order, a higher randomized order is more informative than a non-randomized order. For example, if an input order of each epoch is fixed, i.e., the probability of the appearance of the input order is 1, the training model actually learns the noise of the elements' order instead of the elements themselves.

We leverage cross-entropy to help estimate the randomization level (RL) of an input sequence. The cross-entropy $H(P,Q)$ is

$$H(P,Q) = -\sum_{i} P(i) log_2 (Q(i)), \tag{1}$$

where $P(i)$ and $Q(i)$ indicate the probabilities of the occurrence of $i$-th event.

When we have to use our hybrid backend-memory pipelining approach due to limited memory size, the randomization level is affected since not all elements are available in the in-memory storage buffer for organizing mini-batches at every training iteration. Therefore, the possibility of an input sequence of a hybrid backend-memory pipeline is

$$P = \frac{1}{\prod_{1}^{N_r} \left( C_{N_f - r \times N_c}^{N_c} \times N_{mem}! \right)}, \tag{2}$$

where $N_{mem}$ is the number of memory blocks on all compute nodes, $N_f$ is the number of files of a dataset, $N_c$ is the number of files that can be uploaded in $N_{mem}$ memory blocks, $N_r$ is the number of rounds needed for $N_f$ files to be uploaded to the

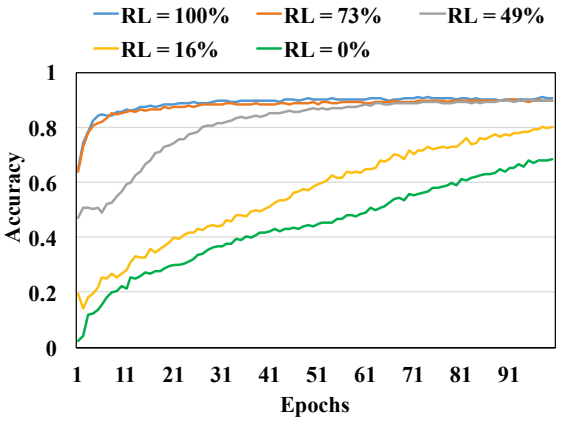



Fig. 7: Validation Accuracy with Different Randomization Level on AlexNet.

$N_{mem}$ blocks, and $r$ indicates the $r$-th file uploading round in an epoch. Then, $C^{N_c}_{N_f - r \times N_c}$ in Equation 2 implies the number of possible file combinations in memory, and $N_{mem}!$ is the number of all possible input sequence for one file uploading round. In addition, the possibility of a fully shuffled input sequence is

$$Q = \frac{1}{N_{images}!}, \tag{3}$$

where $N_{images}$ is the image count of a dataset.

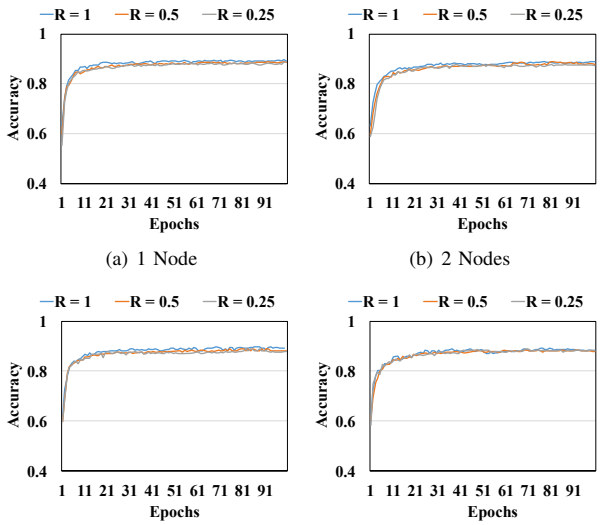
Equation 2 and 3, however, are hard to calculate if a dataset has a large number of files and elements. To evade the potential calculation problem, we calculate the appearance probability of each element when building the input file sequence with different modes. We leverage $P(i,r)$ and $Q(i)$, shown in Equation 4, as the possibility of $i$-th element in a pipelined and a fully shuffled sequence, respectively.

$$\begin{aligned} P(i,r) &= \frac{N_c}{N_f - N_c \times r} \times \frac{1}{N_{mem} - i\%N_{mem}}, \\ Q(i) &= \frac{1}{N_{images} - i}, \end{aligned} \tag{4}$$

where $r$ is the dataset file uploading round ID, $\frac{N_c}{N_f - N_c \times r}$ in $P(i,r)$ implies the chance of selected files on memory blocks in pipelined sequence with shuffling without replacement, and $\frac{1}{N_{mem} - i\%N_{mem}}$ and $\frac{1}{N_{images} - i}$ are the possibility of randomly choosing elements without replacement from memory blocks. Then, $P(i,r)$ and $Q(i)$ from Equation 4 can be applied to Equation 1 to calculate the cross-entropy between any pipelined and fully shuffled sequences. Therefore, the randomization level (RL) is

$$RL = \frac{H}{H_{fully}}, \tag{5}$$

where $H_{fully}$ indicates the cross-entropy between two fully shuffled sequences, and $H$ is the cross-entropy between the input sequence and a fully shuffled sequence. When $RL = 100\%$, it means that the input sequence is fully shuffled. When




(a) 1 Node (b) 2 Nodes

(c) 4 Nodes (d) 8 Nodes

Fig. 8: Accuracy validation for entropy-aware pipelining.

$RL = 0\%$, it implies that the input sequence is in a constant order.

To show the randomization level effect on the validation accuracy, we have trained AlexNet [33] with Cal101 dataset on a single machine, where the AlexNet is a convolutional neural network with eight layers and the Cal101 is a dataset with 101 categories and 40 to 800 images per category. In the experiments, we divide the dataset into training (85%) and validation (15%) portions. The $RL$ in Fig. 7 indicates the randomization level, and $RL = 100\%$ and $RL = 0\%$ mean fully shuffled and constant order, respectively. To emulate other randomization levels, we read $N$ images in a constant order. $RL = 73\%, 49\%$, and 16% imply that 4, 16, and 256 images are concatenated in a constant sequence, which means that every 4, 16, and 256 images are treated as an independent element in shuffling, respectively. As shown in Fig. 7, the best validation accuracy comes with full shuffling, and the accuracy decreases with the randomization level decreasing.

To demonstrate that our pipeline does not affect the randomization level, we have trained the AlexNet similarly to the previous experiment but on multiple nodes, as shown in Fig. 8. In Fig. 8, $R$ indicates the ratio of the shuffling memory size to the size of the entire training dataset. For example, when $R = 0.25$, it means that the memory size used to store the dataset for one round of random read is 25% of the entire dataset. When $R = 1$, it means that the entire dataset is resided on the memory indicating no uploading pipeline. The randomization level of $R = 0.5$ and 0.25 are 98.54% and 96.96% respectively. Therefore, in these cases, the randomization of generated mini-batches could deliver almost the same validation accuracy as shown in Fig. 8. Although the size of mini-batches changes with node counts, we can still keep high training accuracy by carefully adjusting training parameters.

### D. DeepIO API

We design a generic API for integrating DeepIO into TensorFlow. We provide a frontend and backend API to support loading dataset to TensorFlow. Our goal in developing this API is to enable datasets to be read easily from different storage systems using our DeepIO framework. Additionally, using the DeepIO API avoids the redundant memory copies and thread scheduling overheads presenting in the TensorFlow `tf.data` API.

The frontend API is used directly by a training worker:

1) `generate_seed(epoch_count, seed_file);`
   Generate a seed list whose length is `epoch_count` and store in a `seed_file`.
2) `index_array = shuffle_rand(epoch_id, worker_id, seed_file);`
   Get a randomized index array based on a provided seed from `seed_file` for a specific worker with `worker_id` at the certain epoch.
3) `mini_batch = deepIO_batch(index_array, batch_size, element_size, count_per_read, dataset_filename_list);`
   Read and produce a mini-batch with `batch_size` of elements based on the previously generated `index_array`. `element_size` is the fixed length of each element (i.e., image), `count_per_read` is the element count when reading from backend storage used in the backend API, and `dataset_filename_list` contains the list of the dataset file name. Note that this is a TensorFlow operation.

The backend API (inside the `deepIO_batch()` for incorporating different storage systems):

1) `deepIO_inner_read(index, read_size, count_per_read, out_tensor, &fs_read);`
   This function works inside the previous `deepIO_batch` TensorFlow operation. `index` indicates the element IDs, `read_size` and `count_per_read` indicate the read size to underlying storage systems, `out_tensor` is the destination of output tensor, and `fs_read` is the read function for reading images from a specified storage system.
2) `deepIO_inner_init(&fs_init, argv);`
   `fs_init` is the initialization function of the backend system. Same as `deepIO_inner_read()`, this function also resides inside the `deepIO_batch`.

Here, we briefly describe the implementation of DeepIO API. First, to avoid the additional communication channel to be built for broadcasting the randomized image index array, DeepIO stores a seed list for universally generating the array on every node by `generate_seed()`. Then each worker can use the seed to produce a shuffled index array and fetch its portion of the index array based on the worker ID in `shuffle_rand()`. Before using the `deepIO_batch()` to build mini-batches from different storage systems, the additional effort for indicating the read and the initialization function of the storage system is needed. The mini-batches that are the output of `deepIO_batch()` will be ready for the next training iteration. The TensorFlow operation `deepIO_batch()` is written, registered, and run in the same way as the original TensorFlow operations [13], and there is no additional change to the TensorFlow source code. In addition, to reduce the potential overheads involved in the TensorFlow `tf.data` API, we directly enable loading mini-batches from DeepIO API instead of implementing DeepIO as a general file system platform [12] for TensorFlow.

## IV. Experimental Evaluation

In this section, we evaluate the performance of DeepIO implementation. We explore the contributions of the use of DeepIO API, pipelining, and entropy-aware opportunistic ordering.

### A. Experimental Setup

We conduct our experiments on Innovation, an in-house cluster at Florida State University. Each node is equipped with 10 dual-socket Intel Xeon(R) CPU E5-2650 cores, 64 GB memory, and a 1 TB Seagate ST91000640NS SATA disk. Some of the nodes are equipped with a 400 GB Intel DC P3700 NVMe SSD. All nodes are connected through an FDR InfiniBand interconnect with the ConnectX-3 NIC.

Because there is no other open source solution similar to DeepIO for addressing the potential I/O problems in DNN training, we evaluate our DeepIO against two existing storage systems.

**BeeGFS**: We use BeeGFS as an example of a conventional parallel file system. For a fair comparison, we set up BeeGFS over memory (tmpfs) for storage. We have two Metadata Servers and six Object Storage Servers. Each Metadata Server has one Metadata Target of 30 GiB size, and each Object Storage Server maintains two RAID'ed Object Storage Targets of 60 GiB each. These sizes are selected according to the need of the dataset we were using. For the striping pattern of BeeGFS, we use default RAID0 type with the chunk size of 512K and 4 desired storage targets.

**Octopus**: Although there are a few memory-based storage systems, such as Crail [42], NVFS [27], and Alluxio [36], we use Octopus [38] as the RDMA- and memory-based comparison target of DeepIO. As the state-of-the-art RDMA based memory file system solution, Octopus already shows that it can outperform the other options listed above. Since DeepIO leverages memory of allocated compute nodes as data storage, we also emulate the same scenario when using Octopus on compute nodes for a fair comparison. Therefore, Octopus's servers and clients are collocated on the same nodes (i.e., one server and one client on one node).

To measure the read performance of BeeGFS, Octopus, and DeepIO, we generate a dummy dataset with random numbers to represent the pixel values of images. As randomization is needed for reading training dataset, we leverage fully randomized read across all files. Therefore, the read pattern in tests is fully randomized if there is no additional notification. The reported results are the average of 10 tests.

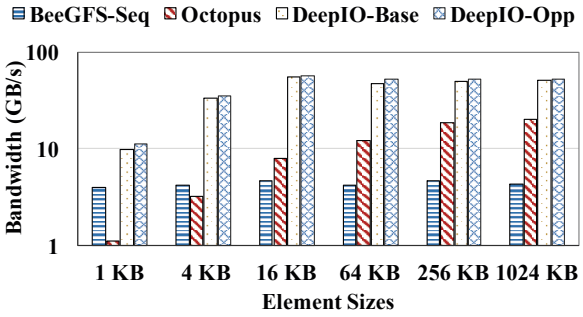



Fig. 9: Aggregate read bandwidth for different element sizes.

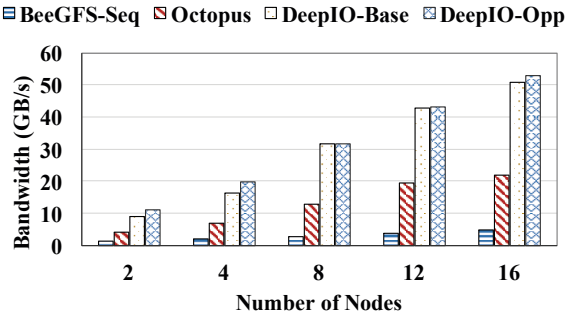



Fig. 10: Aggregate read bandwidth with different node count.

### B. Overall Bandwidth

Here we evaluate the bandwidth achieved by DeepIO using different element sizes, node counts, and numbers of clients.

*1) Read Bandwidth of Different Read Sizes:* Fig. 9 shows the concurrent random read bandwidth with varying image sizes (element sizes) on 16 nodes for BeeGFS, Octopus, and DeepIO. In our experiments, the total read size is 160 GB, with 10 GB per node. The element size varies from 1 KB to 1 MB. Although running on memory, the read bandwidth of BeeGFS is still limited by its complexity. Our experimental results show that the fully randomized read bandwidth on BeeGFS (not shown in Fig. 9) is at least 2× lower than the aggregate N-to-N sequential read bandwidth (N processes read N files simultaneously). So we leverage the aggregate N-to-N sequential read bandwidth of BeeGFS as the baseline, which is denoted as *BeeGFS-Seq* in Fig. 9. The fully shuffled read bandwidth of Octopus is implied as *Octopus* in Fig. 9. For DeepIO, we have two test scenarios as shown in Fig. 9, where *DeepIO-Base* indicates that the returned elements follow the input request submission order, and *DeepIO-Opp* indicates that the returned elements follow the opportunistic order.

In Fig. 9, when the element size is small, BeeGFS-Seq performs better than Octopus. This is because Octopus clients read elements in a fully randomized order instead of sequentially. Moreover, comparing Octopus and DeepIO's performance, the Octopus's read performance is much lower than DeepIO when the element size is small. This is because clients of Octopus have to consult their servers to know the address of the requested data before data operations. When the element size is small, more requests for checking data addresses with Octopus's servers are triggered by the clients, leading to relatively lower performance. In fact, because the training models only read datasets instead of overwriting them, it is not necessary to check the address of data with Octopus' server for every read request. Additionally, because servers and clients are collocated on the same nodes, both in and out data transfers consume the RDMA bandwidth. In contrast, DeepIO shows better read performance for all, especially on small element sizes, since no redundant operations are performed while reading elements from memory. Therefore, Octopus as a general-purpose file system cannot match the performance of DeepIO.

Overall, DeepIO-Base and DeepIO-Opp outperform BeeGFS-Seq by at least 1.43× and 1.82×, and up to 11.14× and 11.54×; DeepIO-Base and DeepIO-Opp surpass Octopus by at least 1.5× and 1.59×, and up to 9.34× and 9.86×. When the element size is small (e.g., 1 KB), the performance of DeepIO-Opp is 16.1% higher than DeepIO-Base; but when the element size is getting larger, the performance difference between the two becomes less, as less inter-process communications are triggered in reading.

*2) Read Bandwidth at Scale:* We also evaluate the read bandwidth using a larger number of nodes. As Fig. 9 reveals, the read performance does not increase much when the element size is greater than 256 KB. We use 256 KB as the element size for the scalability tests. The results for scalability tests are shown in Fig. 10. The aggregate read performances of all test cases increase with the number of nodes. Both DeepIO scenarios have consistently better performance than the other two systems, mainly because there are fewer overheads introduced in a read operation. In overall, DeepIO-Base and DeepIO-Opp outperform BeeGFS-Seq by at least 6.12× and 7.81×, respectively; they also exceed Octopus by at least 1.17× and 1.21×, respectively.

### C. Data Importing API for TensorFlow

We deploy DeepIO API and compare the performance of reading dataset through raw DeepIO (i.e. *DeepIO-Raw*) and the proposed DeepIO TensorFlow API (*DeepIO-TF*). Fig. 11 shows the aggregate bandwidth over different node counts with the read size of 256 KB. The performance of importing data through DeepIO-TF delivers around 26.7% of the DeepIO-Raw. This is because the read time with DeepIO-Raw is too short and the added overhead time in DeepIO-TF is close to the pure DeepIO read time.

We further compare our DeepIO API with TensorFlow's Dataset API over several different types of file/storage systems (i.e., Ext4-disk, Ext4-ssd, tmpfs, BeeGFS-1C, BeeGFS-4C mentioned in Section II-D). Table I shows the read bandwidth of using different APIs over different storage systems with two datasets (Cifar10 and a 16 GB dummy dataset). The element size of Cifar10 and the 16 GB dummy dataset are 3 KB and 256 KB, respectively. The read performance is largely different for different backends.

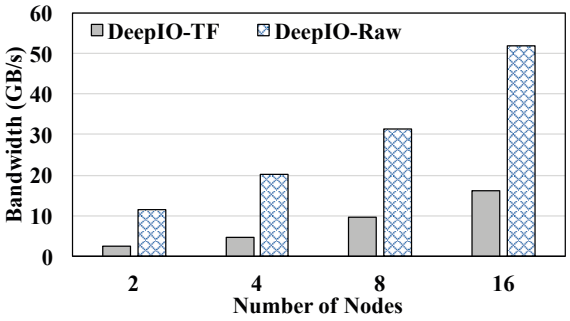
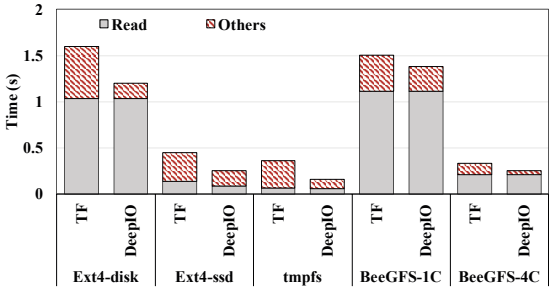


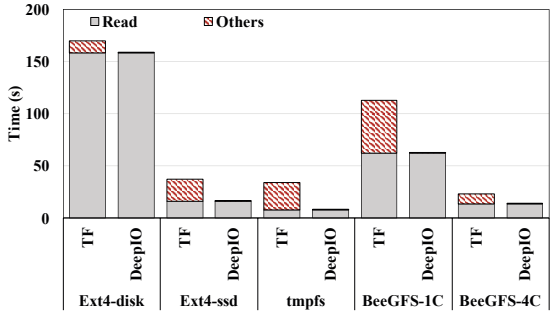

Fig. 11: Read bandwidth of alternative data APIs.

| (MB/s) | TF Dataset API | | DeepIO API | |
|---|---|---|---|---|
| | 3 KB | 256 KB | 3 KB | 256 KB |
| **Ext4-disk** | 105.641 | 95.971 | 122.363 | 103.267 |
| **Ext4-ssd** | 324.927 | 438.183 | 572.116 | 987.103 |
| **tmpfs** | 384.07 | 478.433 | 907.193 | 2034.814 |
| **BeeGFS-1C** | 97.429 | 145.744 | 105.672 | 259.589 |
| **BeeGFS-4C** | 485.739 | 718.606 | 572.978 | 1184.631 |

TABLE I: Images loading speed of different APIs on different storage systems.




(a) Time breakdown of reading Cifar10 (3 KB).




(b) Time breakdown of reading a dummy dataset (256 KB).

Fig. 12: Time breakdown of different loading APIs.

According to Table I, for Cifar10 (3 KB), DeepIO API outperforms TensorFlow Dataset API by 15.8%, 76.1%, 1.36%, 8.46%, and 26.7%, over Ext4-disk, Ext-ssd, tmpfs, BeeGFS-1C, and BeeGFS-4C, respectively. For the dummy dataset, the performance of DeepIO API exceeds TensorFlow Dataset API by 7.63%, 1.25×, 3.25×, 78.11%, and 64.85% for the same five storage systems, respectively. The performance difference between DeepIO API and TensorFlow Dataset API over Ext-disk and BeeGFS-1C is not obvious which is due to the relatively low raw bandwidth of backend storage. In addition, according to Table I, with a larger size of dataset (16 GB) and element size (256 KB), DeepIO delivers higher bandwidth compared to TensorFlow Dataset API. One thing to note is that the performance of BeeGFS-4C is 4× greater than BeeGFS-1C. This is caused by more client side cache involvement. As we don't disable cache effect when reading datasets from BeeGFS, using 4 BeeGFS clients across 4 nodes allows more data to be cached while reading.

To understand overheads in TensorFlow Dataset API and DeepIO API, we investigate the time breakdown of the loading process via those two APIs over the same five storage systems (mentioned above). The results are shown in Fig. 12. The *TF* and *DeepIO* indicate TensorFlow Dataset API and DeepIO API, respectively. We see that the actual read time (gray portion) is similar for both APIs on the same storage devices, but the other time cost (read portion) for DeepIO is much lower than TensorFlow Dataset API because we do not introduce the thread scheduling overheads and redundant memory copies when loading datasets. Furthermore, when dealing with the dummy dataset, since the total I/O time is long, the overheads introduced by DeepIO API are almost negligible, as shown in Fig. 12(b). In contrast, when reading Cifar10, since the total I/O time is short, there is a non-trivial overhead, as shown in Fig. 12(a). In a nutshell, the overhead analysis echoes with the bandwidth results we have in Table I.

### D. Performance of Input Pipelining

Fig. 13 shows the overall time for loading a dataset and training with different training iteration time. In the figure, $R$ indicates that the ratio of the shuffle buffer size to the training dataset size, which has been described in Section III-C.

We use 8 DeepIO servers and 8 dummy training workers on 8 different nodes. Every dummy training worker batches $N$ elements in each iteration but does not do any real training. In our experiments, $N = 128$, the element size is 256 KB, and the dataset size is 8 GB. Therefore, the total size of elements for each iteration over 8 nodes is 256 MB. Every DeepIO server reads 1 GB out of the 8 GB dataset from BeeGFS via the BeeGFS client on the node in every epoch. The aggregate dataset uploading bandwidth over 8 BeeGFS clients is 2526.25 MB/s. To investigate the effect of overlapping training and uploading, we use different time intervals to emulate the training iteration time. Every dummy worker sleeps 100, 50, 10, or 5 ms after acquiring every 128 elements in different test sets.

The results are shown in Fig. 13. When $R = 0$, as every element is directly read from BeeGFS, the worker has to wait until 128 elements are read before a training iteration starts. This significantly prolongs the total execution time. When

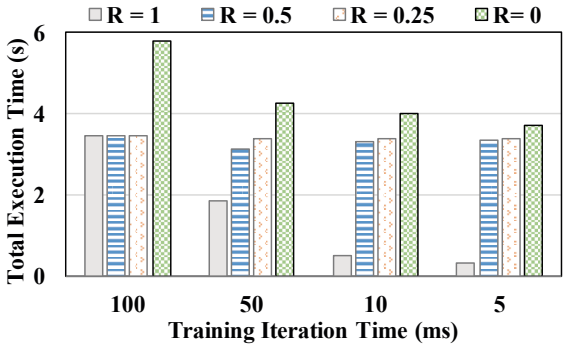



Fig. 13: Total execution time with input pipelining.

$R = 1$ (all data stays in the memory), the overall execution time decreases with the training iteration time, since the performance is not affected by the backend uploading bandwidth. When the time of training iteration equals 100 ms, the total execution time of the first three cases is comparable. This happens because the required minimum backend uploading bandwidth is 2.5 GB/s when the training time is 100 ms, and the aggregate uploading bandwidth (256 MB/100 ms) of 8 BeeGFS clients can meet the requirement. However, when the training time decreases to 50, 10, or 5 ms, the required minimum backend uploading bandwidth increases to 5 GB/s, 25 GB/s, or 50 GB/s, respectively. Therefore, the BeeGFS aggregate bandwidth of 8 nodes cannot allow the uploading to be overlapped by the training shown as $R = 0.5$ and 0.25 in Fig. 13. However, normally in real-world DNN training, the time of one iteration is not as short as those in our experiments. Thus, with appropriate uploading bandwidth, the uploading process can be easily overlapped by the training process.

## V. Related Works

Google Brain team develops TensorFlow [18] and shows its good scalability and training throughput. However, the introduced Dataset API cannot satisfy the need of the full randomization of input dataset among nodes. Caffe [28] as another popular DNN training framework, supports images in raw, HDF5, and LevelDB/LMDB formats. However, in Caffe, reading raw images incurs massive random small reads, shuffling data between HDF5 files is not allowed, and LevelDB/LMDB database format is limited to sequential access. In addition, LBANN [46] uses node-local storage device to store dataset, but not all clusters feature the expensive fast-speed node-local storage devices. Similarly, Weka.IO [15] is a commercial file system that is built over NVMe devices for various types of workloads, including machine learning. However, it needs additional costs and efforts on purchasing and installation on a group of separate storage nodes. In contrast, DeepIO provides an efficient and effective solution to data importing for the DNN training on most HPC systems.

Moreover, many research studies ([29], [27], [25], [38], [42], [9], [2], [36], [15], [22]) have been carried out to exploit Remote Direct Memory Access (RDMA) for improving the communication speed among the compute nodes of clusters and consequently improving the I/O performance of various types of systems, such as remote memory paging [25], key-value stores [29], [21], distributed file systems [27], [38], [42]. In particular, Lu et al. [38] leverage persistent memory with RDMA for developing a file system with high throughput on data I/O and low latency on metadata operations. Stedui et al. [42] propose a fast multi-tiered distributed storage system from ground up for high-performance network and storage hardware to deliver user-level I/O. However, applying the aforementioned RDMA-accelerated middleware directly for DNN training incurs unnecessarily complicated communication process and memory copying, as shown in Section IV.

## VI. Conclusion

The large datasets of DNN training on HPC systems may suffer from the low reading speed due to the limitation of parallel file systems. To better organize mini-batches over HPC systems, we introduce DeepIO for large-scale deep learning with RDMA-assisted in-situ shuffling, input pipelining, and entropy-aware opportunistic ordering. In addition, to implement DeepIO as a prototype over TensorFlow, we implement an alternative data API to allow loading dataset easily from different underlying storage systems. Our experiments show that DeepIO can outperform BeeGFS and Octopus by at least 6.12× and 1.17×, respectively.


### Acknowledgment

This work is performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-750269)[1]. This work is also supported in part by the National Science Foundation awards 1561041, 1564647, and 1744336. We are thankful to Mr. Amit Kumar Nath for his editing comments on the paper.


## References


[1] A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size. https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/.
[2] Accelio. http://www.accelio.org/.
[3] An Overview of Gradient Descent Optimization Algorithms. http://ruder.io/optimizing-gradient-descent/index.html#minibatchgradientdescent.
[4] BeeGFS. https://www.beegfs.io/content/.
[5] Cab. https://computation.llnl.gov/computers/cab.
[6] Caffe Layers. http://caffe.berkeleyvision.org/tutorial/layers.html.
[7] Caffe: Memory Data Layer. http://caffe.berkeleyvision.org/tutorial/layers/memorydata.html.



[1]This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

[8] Catalyst. https://computation.llnl.gov/computers/catalyst.

[9] Ceph over Accelio. https://www.cohortfs.com/ceph-over-accelio.

[10] IOR Benchmark. https://github.com/LLNL/ior.

[11] Lustre File System. http://www.lustre.org.

[12] TensorFlow Adding a Custom Filesystem Plugin. https://www.tensorflow.org/versions/master/extend/add_filesys.

[13] TensorFlow: Adding a New Op. https://www.tensorflow.org/versions/master/extend/adding_an_op.

[14] TensorFlow Importing Data. https://www.tensorflow.org/programmers_guide/datasets.

[15] Weka.IO. https://www.weka.io/.

[16] Wikipedia: Cross Entropy. https://en.wikipedia.org/wiki/Cross_entropy.

[17] Wikipedia Information Theory. https://en.wikipedia.org/wiki/Information_theory.

[18] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A Large-scale Hierarchical Image Database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[20] Li Fei-Fei, Rob Fergus, and Pietro Perona. Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories. *Computer vision and Image understanding*, 106(1):59–70, 2007.

[21] Huansong Fu, Manjunath Gorentla Venkata, Ahana Roy Choudhury, Neena Imam, and Weikuan Yu. High-Performance Key-Value Store on OpenSHMEM. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 559–568. IEEE Press, 2017.

[22] Huansong Fu, Manjunath Gorentla Venkata, Shaeke Salman, Neena Imam, and Weikuan Yu. SHMEMGraph: Efficient and Balanced Graph Processing Using One-sided Communication.

[23] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training Imagenet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[24] Gregory Griffin, Alex Holub, and Pietro Perona. Caltech-256 Object Category Dataset. 2007.

[25] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient Memory Disaggregation with Infiniswap. In *NSDI*, pages 649–667, 2017.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[27] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing*, page 8. ACM, 2016.

[28] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[29] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. Memcached Design on High Performance RDMA Capable Interconnects. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 743–752. IEEE, 2011.

[30] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv preprint arXiv:1609.04836*, 2016.

[31] Alex Krizhevsky. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997*, 2014.

[32] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. 2009.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification With Deep Convolutional Neural Networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[34] Yann LeCun, Corinna Cortes, and Christopher JC Burges. MNIST Handwritten Digit Database. *AT&T Labs [Online]. Available: http://yann.lecun. com/exdb/mnist*, 2, 2010.

[35] Yann LeCun, Fu Jie Huang, and Leon Bottou. Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–104. IEEE, 2004.

[36] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[37] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient Mini-batch Training for Stochastic Optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.

[38] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, 2017.

[39] Nicolas Pinto, David D Cox, and James J DiCarlo. Why is Real-world Visual Object Recognition Hard? *PLoS computational biology*, 4(1):e27, 2008.

[40] Bryan C Russell, Antonio Torralba, Kevin P Murphy, and William T Freeman. LabelMe: A Database and Web-based Tool for Image Annotation. *International journal of computer vision*, 77(1):157–173, 2008.

[41] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[42] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[43] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[44] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going Deeper with Convolutions. Cvpr, 2015.

[45] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the Gap to Human-Level Performance in Face Verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014.

[46] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. LBANN: Livermore Big Artificial Neural Network HPC Toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, page 5. ACM, 2015.

[47] Yang You, Igor Gitman, and Boris Ginsburg. Scaling SGD batch size to 32k for Imagenet Training. *arXiv preprint arXiv:1708.03888*, 2017.