I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning

Fahim Chowdhury
Florida State University
fchowdhu@cs.fsu.edu

Yue Zhu
Florida State University
yzhu@cs.fsu.edu

Todd Heer
Lawrence Livermore Nat'l Lab
heer2@llnl.gov

Saul Paredes
Florida State University
paredes@cs.fsu.edu

Adam Moody
Lawrence Livermore Nat'l Lab
moody20@llnl.gov

Robin Goldstone
Lawrence Livermore Nat'l Lab
goldstone1@llnl.gov

Kathryn Mohror
Lawrence Livermore Nat'l Lab
kathryn@llnl.gov

Weikuan Yu
Florida State University
yuw@cs.fsu.edu

ABSTRACT

Parallel File Systems (PFSs) are frequently deployed on leadership High Performance Computing (HPC) systems to ensure efficient I/O, persistent storage and scalable performance. Emerging Deep Learning (DL) applications incur new I/O and storage requirements to HPC systems with batched input of small random files. This mandates PFSs to have commensurate features that can meet the needs of DL applications. BeeGFS is a recently emerging PFS that has grabbed the attention of the research and industry world because of its performance, scalability and ease of use. While emphasizing a systematic performance analysis of BeeGFS, in this paper, we present the architectural and system features of BeeGFS, and perform an experimental evaluation using cutting-edge I/O, Metadata and DL application benchmarks. Particularly, we have utilized AlexNet and ResNet-50 models for the classification of ImageNet dataset using the Livermore Big Artificial Neural Network Toolkit (LBANN), and ImageNet data reader pipeline atop TensorFlow and Horovod. Through extensive performance characterization of BeeGFS, our study provides a useful documentation on how to leverage BeeGFS for the emerging DL applications.

1 INTRODUCTION

PFS is one of the most essential building blocks of the persistent storage stack in large-scale HPC infrastructure. It provides fast global access to large volumes of data and ensures data persistence through a high number of distributed storage devices. While there are many well-known PFSs, like Lustre [13], PanFS [16], Parallel Virtual File System (PVFS) [24], OrangeFS [15], etc., BeeGFS [4] is quickly emerging into the HPC community mainly because of its performance, scalability, and ease of use. For instance, it supports multiple metadata servers for managing the metadata in a per-file or per-directory basis. Its frequent development to adapt to cutting-edge technologies makes it more promising [1, 3, 5, 9]. Even though there have been lots of research contributions on the evaluation of different PFSs [31, 34, 36, 42–47], not many studies have been performed on BeeGFS. Hence, there is a need for a systematic I/O characterization and performance analysis of BeeGFS for both system practitioners and application users who would like to explore its use on leadership computers.

Meanwhile, the ability of DL applications has been quickly recognized by the HPC community. Researchers are leveraging the current HPC facilities with high computation capabilities for distributed DL training. When conducting training on an HPC cluster, PFS is commonly used for storing the large volume of datasets. The DL frameworks, such as TensorFlow [19], Caffe2 [6], MXNet [20], LBANN [41], etc., invoke file read requests to PFS and form the mini-batches of data required for successful training. Contrasting from the traditional well-structured HPC I/O pattern (e.g., checkpoint/restart, multi-dimensional I/O access), the DL training phase gives rise to highly random small file accesses.

The random file access pattern in DL training is mainly required by the use of Stochastic Gradient Descent (SGD) [2] model optimizer. The SGD model optimizer requires mini-batches being iteratively organized in a randomized order. This is important for accelerating the model's convergence speed and decreasing the noise learned from the input sequence. The randomly shuffled input requirement imposes significant pressure to PFSs which are typically designed and optimized for large sequential I/O. To exploit the best performance out of big HPC systems, DL training frameworks, like TensorFlow and LBANN, offer built-in input pipeline for better I/O parallelism. However, popular image datasets such as ImageNet [23] often have millions of images each ranging from 100KB to 200KB. It is important to evaluate the performance of an emerging PFS to examine its suitability to meet such I/O requirements of DL applications.

In this work, we firstly conduct the evaluation on BeeGFS using IOR [11] and MDTest [14] for a systematic I/O characterization. We stress the system by increasing the number of client nodes and processes per client. Then, we check the resource management efficiency with different stripe patterns and examine the significant metadata operations. To analyze the I/O pressure of DL training workloads, we leverage the real DL applications developed atop LBANN, TensorFlow and Horovod [40]. In the tests, we use an I/O tracing tool Darshan [8] to profile the file access pattern when training AlexNet [30] and ResNet-50 [27] using LBANN with ImageNet over BeeGFS. Later, we develop and use a data reader pipeline for importing data through TensorFlow's Dataset API [18] to perform further experiments.

In summary, we have made several contributions as listed below.

- We examine the composition of input files in large DL datasets and the resulting I/O patterns in DL applications, and discuss their implication to the underlying PFSs on HPC systems.
- We perform an extensive set of experiments for characterizing the impact of data read/write operations on BeeGFS.
- We thoroughly analyze the metadata performance and make recommendations on organizing data for better throughput.
- We report our observations in the context of DL application workloads and discuss the suitability of BeeGFS in handling those.

2 BACKGROUND

In this section, we briefly discuss the architecture of BeeGFS and the functionalities of its basic building blocks. Then, we introduce data pipeline in the DL frameworks used for the experiments. Finally, we discuss the challenges posed by DL applications while importing data from PFS.

2.1 BeeGFS Software Architecture

As shown in Fig. 1, BeeGFS is mainly composed of four system components.

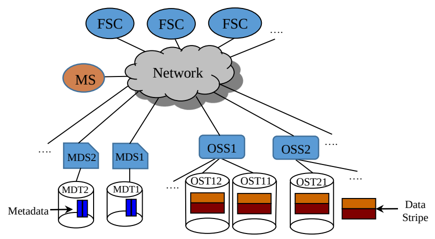


Figure 1: A sketch of BeeGFS architecture

2.1.1 Management Server (MS). There can be exactly one MS in a BeeGFS configuration. It keeps track of the connectivity information and makes sure all the services and targets can find each other during the initial setup. It maintains a list of information, e.g., network status, storage capacity, etc., on all the components of the file system.

2.1.2 Metadata Server (MDS). MDS provides a multi-threaded service that manages information about the object data. These metadata information contain the access permission, directory information, and directory ownership, and the location of user file contents on storage target. One MDS can have exactly one Metadata Target (MDT) where one metadata entry is created per file.

BeeGFS supports **Scalable Metadata Management**. Each MDS is assigned to operate an exclusive portion of the entire file system namespace. BeeGFS maintains a global tree for keeping the information of MDSs. Each node of this tree represents a file or a directory. A node corresponding to a directory contains the information about the MDSs that hold its subdirectories. Hence, even though there can be an unlimited number of MDSs in the system, the directory structure can be traversed efficiently. This design decision for metadata management can facilitate the scalability of metadata operations, e.g., file stat, open and close to read/write, file creation, etc., (see Section 4.4).

2.1.3 Object Storage Server (OSS). Each OSS hosts the file contents in one or many Object Storage Targets (OST). OST is generally a RAID-set with any POSIX compliant and Linux distribution supported file system (xfs, ext4, zfs, etc.) on top. Each OSS manages the striping of data and strives to maximize bandwidth via parallelization while maintaining data consistency.

2.1.4 File System Client (FSC). FSC is the kernel module which facilitates the usage of BeeGFS from the hosts that have installed BeeGFS clients. It includes a service, `beegfs-client`, which loads and exposes the client kernel module for the users.

2.2 Data Pipeline in DL Frameworks

2.2.1 LBANN Data Pipeline. LBANN [41] is a neural network toolkit under active development at Lawrence Livermore National Laboratory (LLNL). It targets at increasing capability of exploiting the parallelism opportunity offered by large-scale HPC facilities. For model parallelism, it leverages an MPI+Threads framework with node-local thread-level parallelism through Intel BLAS library [10] for distributed processing and communication. For data parallelism, it allows fetching mini-batches in parallel from PFSs or node-local storage. When reading datasets, LBANN has several data readers to parse different dataset formats. These include the readers for raw images, CSV files, etc.

LBANN maintains a map of three data readers for training, validation and testing purposes. First, it instantiates the data readers based on the dataset and populates the map. Then, it loads the file path and the labels in another map. Afterward, LBANN instantiates a model abstraction where it sets all parameters for loading data, e.g., the number of parallel data readers, the list of input layers, number of mini-batches, etc. This model is kept as a member of each data reader module. At the beginning of each epoch, a list of indices is shuffled and input layers are constructed. The information about each file, i.e., file path and label, can be accessed from the map through these indices. When an input layer is constructed, it calls the `fetch_data` function from an input I/O buffer manager. There is another function `fetch_datum` defined in each dataset specific data reader. This function is invoked from `fetch_data` for each file through OpenMP multi-threading to read the mini-batch assigned to a parallel data reader. At the end of the pipeline, `fetch_datum` requests the underlying PFS for reading file one by one.

2.2.2 Importing Data in Distributed TensorFlow. TensorFlow [19] is one of the most popular end-to-end open source platform for machine learning developed by Google. It provides rich API sets for developing deep learning applications. Meanwhile, Uber developed Horovod [40], which is a distributed training framework for TensorFlow, Keras [12], PyTorch [17], and MXNet. It has collective communication support via both MPI and NVIDIA Collective Communication Library (NCCL) for parallel communication.

In order to develop an input pipeline for distributed training, we can use Horovod for initializing MPI and sharding the data according to the communicators' size and rank. Thus, each node in the distributed system gets equal portion of data to read from PFS into memory. On the other hand, TensorFlow's Dataset API `tf.data` [18] can be leveraged as a module for importing data during the training phase. It has the provision of iteratively sharding, batching, and optionally shuffling the dataset. Moreover, it has the support for adding custom file reader function through `map_func` parameter in the `tf.data.Dataset.map` method. In addition, there are some built-in functions in TensorFlow, e.g., `tf.read_file`, `tf.image.decode_image`, etc., for reading and decoding common file formats like image files.

2.3 Challenges of Reading Datasets from PFS

Most of the DL frameworks are equipped with built-in data pipeline for supporting distributed training with different dataset formats. For example, CIFAR10 format, TensorRecord format, and raw image (e.g., .jpg) are all allowed in TensorFlow. Different dataset formats

20 and sizes engender distinct I/O workloads on storage systems. Generally, PFSs are designed for efficiently reading large batched files chen ping rather than importing a massive number of small files in random 21 order. Hence, DL applications can put significant challenges on PFS while importing data for training. Particularly, dataset size and ran- chen ping domness of file access patterns are some critical reasons behind these challenges.

2.3.1 Dataset Size. A training dataset has to be read entirely at every epoch. So, the performance difference between reading large and small datasets becomes more noticeable when reading from PFSs in distributed training on clusters. A small dataset is easily cached by the PFS after the first epoch. Afterward, the small dataset is always read from the cache of PFS, which largely accelerates the dataset reading speed. While reading a huge dataset, read requests to 22 physical backend devices may frequently happen, since the dataset cannot fit entirely in the PFS's cache. These frequent I/O requests to chen ping read all the data from a large dataset at each epoch lead to relatively slower I/O performance than that for smaller datasets [22, 49].

2.3.2 Random File Access. The read pattern in DL frameworks can lead to distinguishable I/O throughput. For instance, randomization of the input sequences is mandatory at the beginning of each epoch to facilitate the training parameters' convergence speed and avoid bias from input order. Although some frameworks (e.g., TensorFlow) 23 support local shuffling after sequentially reading a few elements from a batched file, randomly reading small raw images is a general chen ping practice to ensure the randomization of an input sequence. These massive small random reads impose non-trivial performance loss compared to sequential reads of large batched files [22, 49].

3 SYSTEM CONFIGURATION AND WORKLOADS

In this section, we describe the system configuration and software tools we use for the experimental evaluation.

3.1 BeeGFS Configuration

All the experiments are run on the Catalyst cluster [7] at LLNL which is 150 teraFLOP/s Cray CS300 system with 324 nodes with 48 cores per login and 24 cores per compute node. Each compute node is equipped with two 12-core Intel Xeon E5-2695v2 processors, 24 128 GB (41.5 TB in total) dynamic random access memory (DRAM) and 800 GB of non-volatile memory (NVRAM).

Catalyst is equipped with a BeeGFS-7.1.2 setup with 338 TB total chen ping capacity that consists of both SSD and HDD devices underneath as depicted in Figure 2. It has 12 server hosts connected via QLogic Infiniband QDR interconnect, and 12 object storage server (OSS) and 6 metadata server (MDS) processes equally distributed among 25 the hosts. Each OSS manages two object storage targets (OST) each having 15 TB space on 4 HDDs with ZFS RAID-Z formatting and chen ping 721 GB ZFS intent log (ZIL) on SSDs, and Level 2 Adjustable Replacement Cache (L2ARC) log on DRAM devices for caching. Each MDS handles one MDT having 721 GB space on 4 SSDs with ZFS RAID-Z formatting.

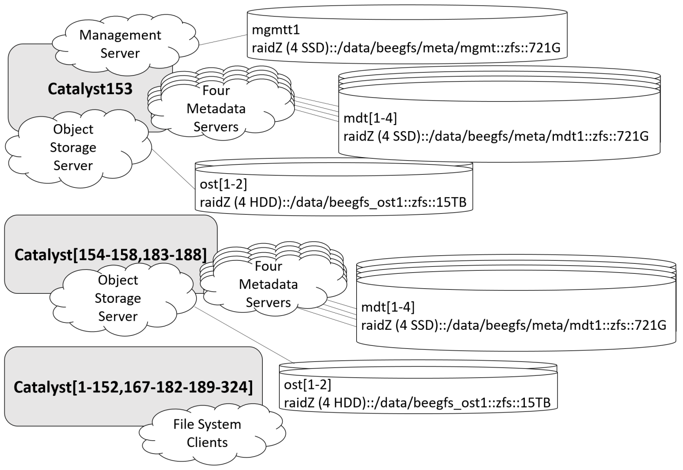

Management Server
mgmtt1
raidZ (4 SSD)::/data/beegfs/meta/mgmt::zfs::721G
Catalyst153
Four Metadata Servers
mdt[1-4]
raidZ (4 SSD)::/data/beegfs/meta/mdt1::zfs::721G
Object Storage Server
ost[1-2]
raidZ (4 HDD)::/data/beegfs_ost1::zfs::15TB
Catalyst[154-158,183-188]
Object Storage Server
Four Metadata Servers
mdt[1-4]
raidZ (4 SSD)::/data/beegfs/meta/mdt1::zfs::721G
Catalyst[1-152,167-182-189-324]
ost[1-2]
raidZ (4 HDD)::/data/beegfs_ost1::zfs::15TB
File System Clients

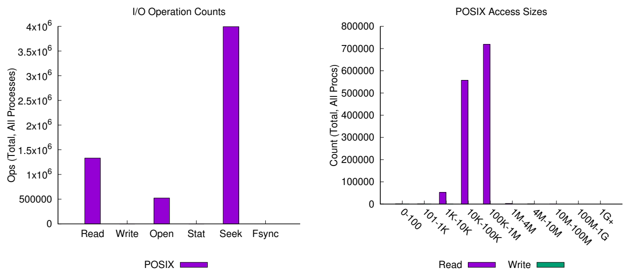
Figure 2: BeeGFS configuration on Catalyst

3.2 Software Tools

We use IOR and MDTest for data and metadata performance analysis respectively. Apart from that, for evaluating the impact of DL workload on BeeGFS, we use AlexNet and ResNet-50 implemented in LBANN that use ImageNet dataset stored in the BeeGFS mount point on Catalyst. Moreover, we develop an input pipeline using TensorFlow and Horovod for importing the same dataset, and perform further experimentation.

3.2.1 Interleaved-Or-Random (IOR). We use IOR-2.10.3 to evaluate the I/O bandwidth of BeeGFS. During the tests, we use MPIIO API for generating the N-N and N-1 workloads. We also enable random inter-file access across all process to alleviate client-side caching. In all the experiments, the aggregated test file size is 240 GiB. The default transfer size is 1 MiB if not explicitly mentioned.

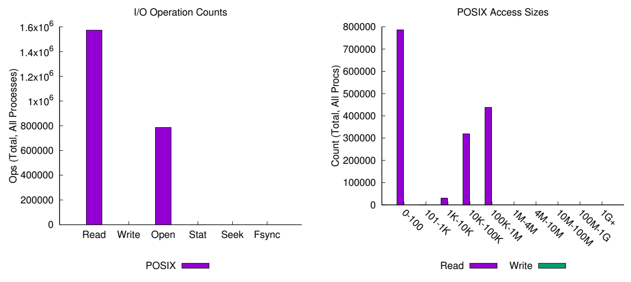
3.2.2 MDTest. We use MDTest-1.9.3 to perform file `create, stat` and `read`. We emulate the effect of flat directory and single-depth hierarchical directory for N-N workload across multiple clients with 8 processes per client while keeping the total number of files to 409600.



(a) Ops vs. I/O Operations (b) Operation Count vs. File Sizes

Figure 3: LBANN ImageNet data reader I/O pattern

3.2.3 LBANN Benchmarks. For evaluating the impact of DL applications on BeeGFS, we train LBANN's AlexNet [30] and ResNet-50 [27] over ImageNet dataset. We set the stripe size to 512 KB and stripe count to 4. In the tests, we increase the number of clients from 4 to 32 for both the models. We run the tests with Darshan-3.1.7. From the Darshan logs, we collect the file access patterns posed by the ImageNet data reader module in LBANN. As shown in Figure 3(a), POSIX file seek operation has the most count in total, because LBANN calls it to measure the file size for initializing the memory buffer to keep the data. Besides, file open and read have reasonable overhead. This pattern demonstrates what happens when an application tries to read from millions of small files from a PFS. Besides, Figure 3(b) depicts that, when accessing the ImageNet dataset by LBANN, most of the POSIX operations on a file are around 100 KB size.



(a) Ops vs. I/O Operations (b) Operation Count vs. File Sizes

Figure 4: TensorFlow ImageNet data reader I/O pattern

3.2.4 TensorFlow Input Pipeline. In the case of TensorFlow experiments, we keep the same BeeGFS stripe pattern as that for LBANN benchmarks and utilize the same tools to perform the measurements. From LBANN results, we observe that training phase does not have much impact on the data import stage in Deep Learning applications. Hence, instead of running the entire training process, we run the input pipeline that we develop for reading the ImageNet dataset using Dataset API [18] in TensorFlow-1.10.0. Moreover, we leverage Horovod-0.16.1 [40] for implementing a distributed TensorFlow input pipeline. As depicted in Figure 4(a), file open and read invokes most of the I/O overhead. On the contrary to LBANN input pipeline, TensorFlow pipeline introduces a large amount of tiny POSIX reads when `tf.read_file` API is used as shown in Figure 4(b). This phenomenon makes the application read less amount of data while the metadata overhead for file reading remains the same.

For developing a simplistic input pipeline, we configure the TensorFlow session with six inter-operation parallelism threads and one intra-operation parallelism thread for leveraging multithreading within TensorFlow. Then we populate two pairs of lists with the file paths and corresponding labels from train.txt and val.txt file in the ImageNet dataset. Later, we create two Tensor dataset instances with the lists of files and corresponding labels. We configure the dataset with a shuffle buffer size of 10, and assign equal shards of data file list and label list across all Horovod ranks. We attach a file reader function with the dataset's `map_func` and set `num_parallel_calls` to 4 for efficient reading. In this function, the `tf.read_file` is used to read each file into memory and `tf.image.decode_image` method is used to decode the file contents.

4 I/O CHARACTERIZATION OF BEEGFS

In this section, we present the evaluation results and perform a step-by-step analysis of BeeGFS I/O and metadata performance. The purpose of these synthetic experiments is to test the different aspects of the PFS by increasing the process number for single node and varying the striping pattern to explore the I/O performance trend. Then, we run scalability tests via increasing client nodes while keeping the other parameters (e.g., number of client processes, stripe size and count) fixed to the best options from the single node tests. Afterward, we discuss the observations from the experiments done by varying transfer sizes for a fixed number of clients, processes per client and striping pattern. Finally, we run some experiments to test the metadata handling performance for critical operations, like file stat, read and create.

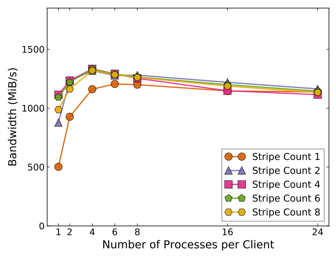
4.1 Single Client Test

The single node BeeGFS tests are performed by varying the number of client processes from 1 to 24 while changing the stripe count from 1 to 8 for stripe size 256 KB, 512 KB, and 1 MB. We choose stripe size 512 KB as the representative to demonstrate the trends in the other two stripe sizes. To better distinguish the performance difference, we select 4, 6, 8 processes per node to rerun the extensive tests by changing the stripe count from 4 to 24 for stripe size 512 KB. In the experiments, we examine file-per-process (N-N) read/write and single-shared-file (N-1) read/write bandwidth.

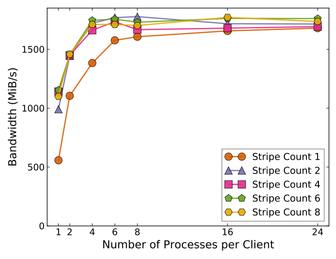
4.1.1 Increasing Number of Processes Per Client. For both N-N read and write, shown in Figure 5(a) and Figure 5(b) respectively, the bandwidth steeply increases when the stripe count increases from 1 to 2 but with increasing stripe counts greater or equal to 2, the bandwidth does not increase much. N-N read bandwidth displays good stress handling from 1 till 4 processes per client, then it decreases slowly with increasing number of processes. Network bandwidth saturation can be the possible reason behind this performance trend. Moreover, the write bandwidth peak (i.e., 1800 MiB/s) is more than that of read bandwidth (i.e., 1333.01 MiB/s). We find this kind of performance measures after incorporating RAID-Z with ZIL. Hence, for single client write-intensive applications, employing RAID-Z with ZIL on SSD devices can be efficient in spite of the existence of HDD OSTs and L2ARC caching for reading as depicted in Section 3.1.

In N-1 read and write tests, as depicted in Figure 5(c) and Figure 5(d) respectively, the bandwidth almost doubles when the stripe count is increased from 1 to 2, as it leverages the bandwidth from additional OSTs. Finally, it reaches the peak (i.e., 1271.58 MiB/s) for stripe count 6 and 6 processes per node. The highest N-1 write bandwidth (i.e., 1166.24 MiB/s) is reached for stripe count 6 and 1 process per node. So, N-1 write almost always suffers from additional processes. Nevertheless, Both read and write bandwidths remain reasonable for stripe count 4 and upwards.

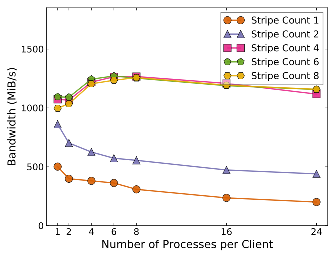
4.1.2 Increasing Number of OSTs. We analyze the results described in Section 4.1.1 and select 4, 6 and 8 processes per node for further investigation. We increase the number of stripe counts from 4 to



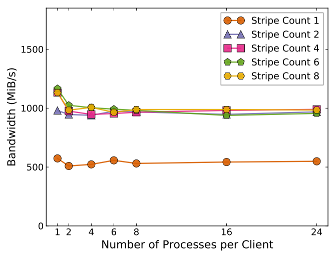
(a) N-N Read



(b) N-N Write

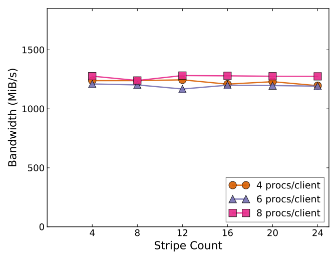


(c) N-1 Read

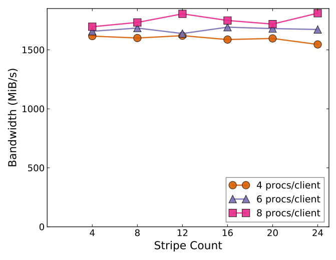


(d) N-1 Write

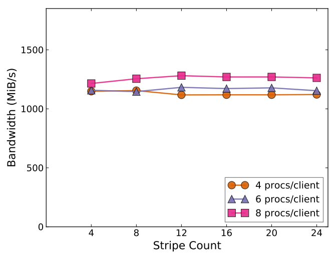
Figure 5: I/O bandwidth for varying number of processes per node



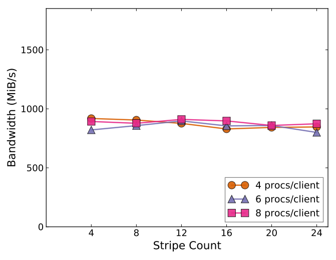
(a) N-N Read



(b) N-N Write



(c) N-1 Read



(d) N-1 Write

Figure 6: I/O bandwidth for varying stripe count

24 while keeping the stripe size fixed at 512 KB. We run the single node BeeGFS test in order to perceive the behavior in the simplest case first. Then, we strain the client to analyze the upper bound and best cases. As depicted in Figure 6, the variability of the N-N or N-1 read/write bandwidth is not much distinct while changing stripe count or the number of active OSTs per operation.

For N-N read, the bandwidth increases with the increasing process count per node, which is relatable to Figure 5(a) plot. As shown in Figure 6(a), N-N read bandwidth reaches the peak (i.e., 1281.89 MiB/s) for stripe count 12 and 8 processes per node. We observe that the bandwidth can benefit more from the underlying hardware layout when the stripe count is equal to the total number of OSSs on Catalyst. For N-1 read, as shown in Figure 6(c), the bandwidth is higher for more processes per node. As stated in Figure 6(d), the bandwidth is good for stripe count 12 when compared with the other values for the same workload. Although the highest bandwidth (i.e., 918.13 MiB/s) is obtained for 4 processes per node and stripe count 4, it is not much higher than the peaks we get for stripe count 12 with 6 processes per node (i.e., 897.54 MiB/s) and 8 processes per node (i.e., 910.48 MiB/s). On an average, according to Figure 6(b) and Figure 6(d), N-1 write bandwidth is almost half of that for N-N write. All in all, the bandwidth change for varying stripe count is negligible.

36 chen ping

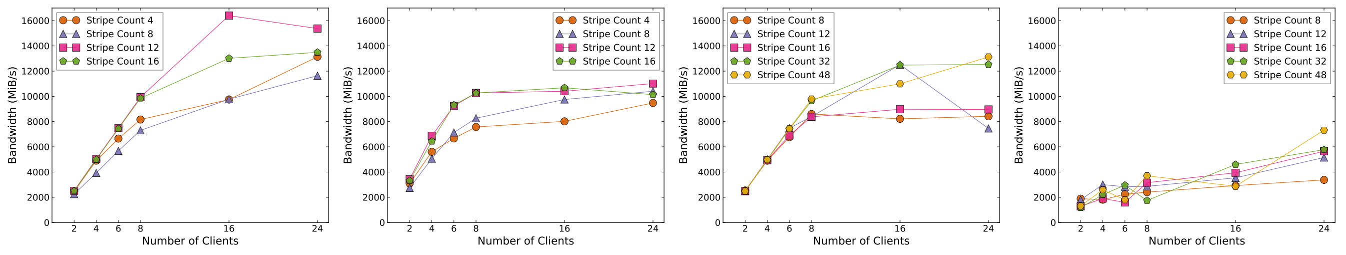
4.2 Scalability

From the insights we get from the experiments discussed in Sec- .1, we extend the research to analyze the scalability behavior of eeGFS by increasing the number of client nodes from 2 to 24. We select 4 and 8 process count per node as candidates for further study instead of performing exhaustive experimentation. We increase the stripe count from 4 to 16 and 8 to 48 for N-N and N-1 workloads respectively. We examine with more stripe count for N-1 workloads than that for N-N, because N-1 workloads are supposed to put more resource contention in case of increasing stripe count. It allows us to create more pressure on the system. The experiments with 8 processes per node generally performed better than others according to the discussion in Section 4.1 and plots depicted in Figure 6. Hence, we take the results for 8 processes per client and keep the stripe size at 512 KB.

37 chen ping

For N-N read, as plotted in Figure 7(a), BeeGFS shows excellent scalability, as the bandwidth increases with the increasing num- f clients, almost linearly till 8 clients. The best bandwidth tained when the stripe count is 12, reaching the peak (i.e., 16262.32 MiB/s) for 16 clients. As stated in Figure 7(b), N-N write bandwidth shows good scalability like N-N read. It reaches the culmination (i.e., 11085.77 MiB/s) for 24 clients and stripe count of 12.

According to Figure 7(c), N-1 read bandwidth demonstrates good scalability, as the bandwidth usually increases with the increasing number of clients. In this case, stripe count 12 performs better than stripe count 16, but with increasing stripe count (e.g., 32 and 48) the data management overhead is overcome by leveraging the extra amount of aggregate bandwidth provided by the additional storage devices. Hence, the bandwidth reaches the peak (i.e., 12336.39 MiB/s) for stripe count 48 and client count 24. For N-1 write, as depicted in Figure 7(d), the bandwidth increases with more clients usually, reaching the highest value (i.e., 6089.51 MiB/s) for stripe count 48 and client count 24 with 8 processes per client. We observe that 48 is a multiple of 12 and the bandwidth gain for involving 36 extra OSTs is not that much. Besides, N-1 write bandwidth is



(a) N-N Read (b) N-N Write (c) N-1 Read (d) N-1 Write

Figure 7: I/O bandwidth for varying number of clients

38

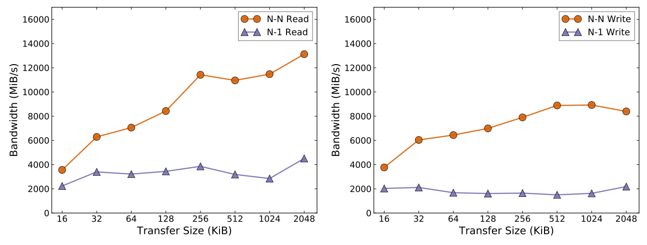
chen ping

almost half of N-N write, but N-1 read performance is not that much less than N-N read.

4.3 Varying Transfer Sizes

39

chen ping



(a) Read (b) Write

Figure 8: I/O bandwidth for varying transfer sizes

To analyze the I/O response of BeeGFS into a more microscopic level, we add this experiment set to evaluate the N-N and N-1 read/write bandwidth for varying transfer sizes. We keep the number of clients fixed at 16 and the number of processes per client constant at 8.

As demonstrated in Figure 8(a), the N-N read bandwidth starts from 3559.33 MiB/s for 16 KiB transfer size and gradually increases to 8428.84 MiB/s for 128 KiB. It reaches a secondary peak for 256 KiB and stays flat till 1024 KiB that finally reaches the peak (i.e., 13125.28 MiB/s) for 2048 KiB transfer size. Again, N-1 read bandwidth remains in the range of 2236.90 MiB/s for 16 KiB and 4511.81 MiB/s for 2048 KiB transfer size.

40

chen ping

According to Figure 8(b), we observe a smooth increment of N-N write bandwidth for varying transfer sizes that saturates at 512 KiB. For instance, the bandwidth is 3768.60 MiB/s for 16 KiB transfer size, which almost doubles to 6038.20 MiB/s for 32 KiB. Then it slowly increases to 6992.61 MiB/s for 128 KiB and reaches the peak at 8924.43 MiB/s for 1024 KiB transfer size. On the other hand, the N-1 write bandwidth decreases from 2113.93 MiB/s for 32 KiB to 1503.34 MiB/s for 512 KiB through 1612.47 MiB/s for 128 KiB transfer size. Then it increases again to 2185.47 MiB/s for 2048 KiB.

41-43

3 notes:

Both read and write bandwidths for N-N workload show better performance trends than that for N-1. This event again recommends avoiding N-1 workload invocation in DL applications. Another notable aspect of these experiments is the demonstration of low bandwidth for lower transfer sizes. This situation creates a plot for us to expect even lower I/O bandwidth for DL I/O workloads, where we often need to deal with thousands of small sized files of around 100 KiB per application process.

BeeGFS I/O Observation 1: *N-N workload is almost always better than N-1.* As discussed in Section 4, all the experimental results demonstrate that N-N workload is always better than N-1 workload on BeeGFS. Although N-1 read bandwidth is sometimes close to N-N read, N-1 write bandwidth is very low in comparison to N-N write bandwidth. On the other hand, the correlation between N-N and N-1 read suggests the presence of a good file read consistency handling mechanism in BeeGFS. For DL applications on HPC systems, we seldom face N-1 read overhead by reading file lists or labels per file, but there is a chance that N-1 write can be invoked by the developers for logging and checkpointing. We want to recommend assigning a single DL application process for writing the log or checkpointing files instead of involving all the processes to avoid N-1 write workload and leverage BeeGFS in an effective manner.

BeeGFS I/O Observation 2: *Increasing stripe count is not always necessary.* It might be a general perception that we need to add more OSTs by increasing the stripe count in BeeGFS to get better throughput. It might not always be considered as a proper decision, as adding up more active OSTs adds communication overhead. Our experiments stated in Section 4.1.2 suggest that we cannot take much benefit from the additional OSTs when we increase the stripe count. From our observations, we find that a stripe count of 4 is reasonable enough to get a good read/write bandwidth while having a cost-effective resource utilization.

4.4 Characterization of Metadata Operations

We collect results for scale-out evaluations using MDTest with 4 to 128 clients having 4, 8 and 16 processes per client. Besides, we vary the BeeGFS stripe count from 4 to 48 for stripe size 512 KB. We run experiments for analyzing the performance of file `stat`, `read`, `create` and `remove`, and tree `create` and `remove` operations. The file `stat`, `read` and `create` operations are the most important ones in the training phase of the DL applications, because, during this phase, the applications check the stat of the files, read them from the underlying PFS and create files for checkpointing. In this case, the measurement of `read` operation actually measures the cost of opening a file, reading 1 B and closing the file, hence it covers the file open and close operations in general.

For flat directory metadata performance evaluation, we perform the operations on files in a single directory. When testing the performance for hierarchical directory structure, we uniformly distribute the files among all the processes involved and keep the files assigned to each task in a dedicated directory under the parent directory. Besides, we analyze the results by operations on a single file shared by all the involved processes. In this section, we discuss the performance of file `stat, read` and `create` for flat directory, hierarchical directory and single shared file by describing the trends shown in Figure 9. We describe the operations per second (ops/sec) performance for each metadata operation presented in logarithmic scale while varying the number of clients with 8 processes per client, 512 KB stripe size, 4 stripes, and 409600 total files.

4.4.1 File Stat Performance. Any file I/O access requires file `stat` operation at least for checking the access level of the respective file. As depicted in Figure 9(a), we observe good scalability for file `stat` operation for flat and hierarchical directory structure, but this operation does not perform as well for the single shared file. For flat directory structure, the performance of file `stat` rises till 16 clients increasing from 5129.80 ops/sec for 4 clients to 11473.31ops/sec for 16 clients, but saturates with increased client count staying in the range of 11000 to 12000 ops/sec. On the other hand, the performance of file `stat` is better for hierarchical directory structure. For instance, it increases from 4546.10 ops/sec for 4 clients to 14052.94 ops/sec for 16 clients and finally reaches to 15648.71 ops/sec for 128 clients. On the contrary, for single shared file, the performance value starts from 2572.91 ops/sec for 4 client nodes and increases to 4289.34 ops/sec for 16 clients. Afterward, it gradually decreases to 1474.24 ops/sec for 128 clients.

4.4.2 File Read Performance. The metadata operations involved in the `read`, i.e., file `open` and `close`, are the ones that are invoked the most during the training phase of DL applications. As presented in Figure 9(b), for flat directory structure, file `read` keeps itself in the range of 2299.60 ops/sec to 2839.67 ops/sec. Similar to the performance of file `stat`, for hierarchical directory structure, file `read` operation displays better performance than the others. For instance, it has impressive performance measures of 4635.74, 13937.09 and 15285.14 ops/sec for 4, 16 and 128 clients respectively. In this case, the single shared file results show good scalability which is slightly better than flat directory structure, as file `read` operation gains 1657.67 ops/sec for 4 clients and grows up to 3222.93 ops/sec for 64 clients and slightly drops to 3121.78 ops/sec for 128 clients. Even though the single shared file workload is effectively handled by BeeGFS with increasing number of clients involved, when designing DL applications to import the dataset from BeeGFS, it is a good idea to avoid single shared file accesses and lessen process contention. We recommend arranging the dataset in a hierarchical directory structure in order to leverage the metadata performance acceleration.

4.4.3 File Creation Performance. File `create` operation is particularly important in case of checkpointing and logging in DL applications. According to Figure 9(c), for the experiments on flat directory structure, ops/sec increases with the increasing number of clients, but the increment is not sufficient to claim that it demonstrates a good scale-out performance trend. In this case, the value is 1312.09 ops/sec for 4 clients and increases up to 1489.28 ops/sec for 128 clients. As usual, for hierarchical directory structure, file `create` operation performance grows better from 4581.17 ops/sec for 4 clients to 13375.17 ops/sec for 16 clients. Then with increasing number of clients, it almost saturates, rendering 15270.43 ops/sec for 128 clients. Single shared file `create` performance is nominally better than flat directory structure. In particular, it displays good resource management along with slight scalability factor as the performance gradually increases from 1695.25 ops/sec for 4 clients to 3499.18 ops/sec for 128 clients. The results on a single file shared by all the involved processes show reasonable trends that demonstrate good process contention handling in BeeGFS as the ops/sec does not change much or generally increases with the increasing number of clients. Similar to file `stat` and `read`, file `create` operation performs the best for hierarchical directory structure.

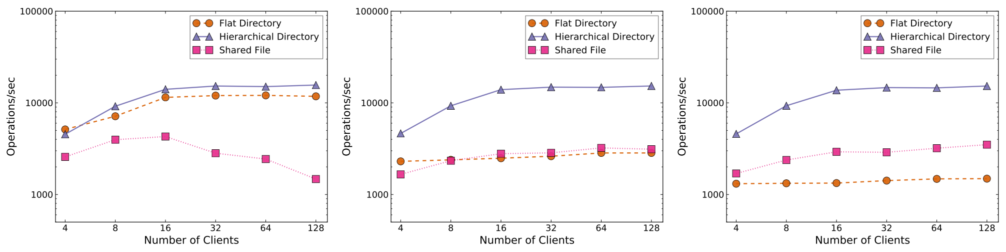
BeeGFS Metadata Observation: *Organizing data in hierarchical structure can be helpful for metadata handling.* The metadata operations' characterization has shown very interesting insight on data arrangement. When we perform tests on hierarchical directory structure, we can see a clear benefit for all metadata operations important for DL workload. In DL application's perspective, we think if the dataset can be arranged hierarchically distributed through different directories in BeeGFS instead of keeping all the files in a single flat directory, the obvious metadata operations like `stat`, `open` and `close` can take advantage of the scalable metadata management design in BeeGFS.

5 PERFORMANCE EVALUATION OF DEEP LEARNING APPLICATIONS

With a view to analyzing the behavior of I/O during the training phase of different DL applications, we consider two convolutional neural network models named AlexNet and ResNet-50 on top of LBANN. Later, we run experiments using a distributed ImageNet input data pipeline developed using TensorFlow and Horovod. We perform these tests using the ImageNet dataset kept on BeeGFS mount point in Catalyst cluster. For these experiments, we choose the stripe size of 512KB and stripe count of 4 for the dataset. We increase the number of nodes involved in running the models from 4 to 32 while keeping the number of processes per node fixed at 8. We collect the average cumulative time per process for file read and metadata operations, and the average data read by each process from Darshan traces. We deduce the total data read and divide it with the total I/O time, i.e., read time + metadata time, to estimate the parallel I/O bandwidth.

5.1 LBANN Benchmark Results

5.1.1 AlexNet. Figure 10(a) shows that the parallel read bandwidth on BeeGFS invoked by AlexNet on LBANN is overall good, but it does not scale efficiently with increasing number of clients. We observe that, although the average cumulative file read time reported by Darshan decreases with an increasing number of nodes, the metadata retrieval time does not drop. The overall data reading bandwidth cannot scale-out with increasing number of clients, because metadata handling becomes the bottleneck. For instance, the read time decreases smoothly from 73.76 seconds to 9.7 seconds when the number of clients increases from 4 to 32. On the other

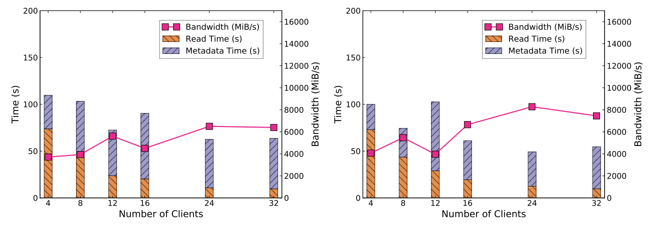


(a) File Stat (b) File Read (c) File Creation

Figure 9: Metadata performance for varying number of clients

47

chen ping



(a) AlexNet (b) ResNet-50

Figure 10: I/O latency and read bandwidth of LBANN Benchmarks

hand, the metadata time ranges between 35.92 and 70.12 seconds for 4 and 16 clients respectively. Hence, The estimated read bandwidth increases slowly from 3710.25 MiB/s to 6500.6 MiB/s when the number of clients is increased from 4 to 24 and lessens a little bit to 6398.67 MiB/s for 32 clients.

5.1.2 ResNet-50. Similar to AlexNet I/O bandwidth, ResNet-50 shows good read bandwidth trend as shown in Figure 10(b). The bandwidth is accordingly governed by the metadata time and keeps itself in a reasonable boundary, while the read time decreases with increasing clients. In particular, the read time decreases from 72.86 to 9.72 seconds when the number of clients increases from 4 to 32. The metadata resides in the range of 27.17 seconds for 4 clients to 73.48 seconds for 12 clients. The bandwidth increases from 4068.09 MiB/s to 8281.12 MiB/s when the number of clients increases from 4 to 24, with a decrement to 3968.28 MiB/s for 12 clients. Again, the bandwidth decreases to 7453.48 MiB/s for 32 clients.

LBANN Observation 1: *BeeGFS can reasonably handle the I/O pattern posed by LBANN.* When we compare the bandwidth results shown in Figure 10 with that in Figure 7(a), we find that the I/O pattern of DL applications' training phase can be handled well by BeeGFS. For N-N read tests, there is one large file per process workload. For DL applications on LBANN, one data reader thread has to read thousands of files per epoch. This phenomenon creates a great deal of metadata overhead, and BeeGFS can feasibly handle this situation. Overall, the bandwidth stays between one-third to half of what we reported in Figure 7(a) for tests with IOR.

48-49
2 notes:

50

chen ping

LBANN Observation 2: *The directory structure of ImageNet dataset assists the performance of BeeGFS.* Files in the ImageNet dataset are by default arranged in a hierarchical directory structure. From Figure 8(a), we can observe that lower transfer sizes have negative impact on the N-N read bandwidth. Although the most popular file size of ImageNet dataset is around 100 KB, as depicted in Figure 3(b), and the metadata overhead is supposed to increase more with increasing clients because of contention, the directory structure of ImageNet helps contain the metadata overhead in a reasonable range. The observations in Section 4.4 suggests a similar trend.

51

chen ping

5.2 Tensorflow Benchmark Results

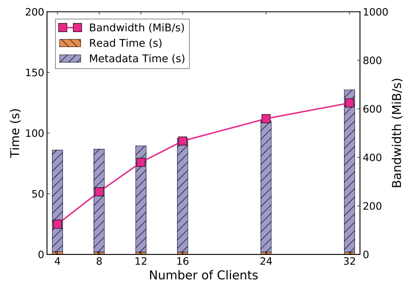


Figure 11: I/O latency and read bandwidth of ImageNet data reader pipeline on TensorFlow

5.2.1 ImageNet Data Reader Pipeline. As depicted in Figure 11, the internal file read optimization in TensorFlow helps lessen the average read time while the metadata overhead remains high. According to the Darshan traces, the total amount of data read is proportional to the total number of processes involved in the application, i.e., less data is read for a lower number of processes. This situation renders the experiment as a weak scaling test. Hence, the bandwidth calculated by dividing the total data by the summation of read and metadata time appears very low in comparison to the bandwidth reported in LBANN benchmarks. For instance, read time slightly decreases from 2.28 seconds for 4 nodes to 1.95 seconds for 32 nodes. On the other hand, the metadata operation latency increases from 83.74 seconds for 4 nodes to 133.71 seconds for 32 nodes. Consequently, the read bandwidths are reported as 124.75, 467.39 and 624.33 MiB/s for 4, 16 and 32 nodes respectively.

Tensorflow Observation 1: *TensorFlow internally tries to optimize file reading by default.* According to the Darshan logs, when the `tf.read_file` method is used in the file reader function mapped through Dataset API's Python wrapper, the application reads small ons of the file rendering the data pipeline with a lot of tiny POSIX I/O accesses as shown in Figure 4(b). Besides, the total amount of data read is increased with more resources, so the bandwidth increases with additional application nodes.

Tensorflow Observation 2: *Metadata handling is a notable bottleneck in TensorFlow data pipeline.* While TensorFlow optimizes file read time by reducing the read access size, the metadata overhead still remains the same, because the files are required to be opened anyway. Hence, metadata handling stays as the main bottleneck in the data import pipeline. Eventually, it hampers the overall throughput of BeeGFS by invoking lots of tiny read requests of less than or equal to 100 B, (see Figure 4(b)).

6 RELATED WORK

There have been many efforts in the HPC community to characterize and analyze different performance metrics of various components in supercomputing infrastructure. Notably, a large body of literature is available on the performance analysis of the PFSs such as Lustre. For instance, Yu et al. perform thorough characterization of data and metadata I/O scalability trends of Lustre atop a Cray XT supercomputer named Jaguar at Oak Ridge National Laboratory, followed by different tuning and optimization techniques on scientific applications to leverage the system [46]. Besides, there has been another case study by Yu et al. on the performance of Lustre over Quadrics and InfiniBand using sequential and parallel I/O, metadata and application benchmarks [45]. While [31, 34, 36, 42–44, 47] demonstrate different techniques to evaluate the I/O performance of PFSs in supercomputing facilities, [32, 35, 39] emphasize on PFS, application and data API tuning and optimization to acquire better I/O throughput on HPC systems. Although many research attempts have been taken for I/O performance analysis of different PFSs, there is a lack of understanding on the characteristics of BeeGFS I/O metadata performance, particularly its capability of handling the workloads posed by DL applications.

In recent times, the emergence of DL in solving real-world problems has given rise to the significance of analyzing the performance of DL applications' workflow. [25, 26, 28, 29, 33] state the results on the evaluation of different DL applications on different HPC systems, but all of these mainly deal with the computation characterization. There is not much work done yet on the I/O profiling and optimization for DL training workload, except [21, 22, 37, 38, 48, 49]. Among them, Zhu et al. have used BeeGFS as one of the baseline PFS for the comparison with the DeepIO implementation [49]. Besides, there are some white papers from ThinkParQ discussing the performance trends of BeeGFS, but those do not evaluate its capability of handling DL applications. Hence, to the best of our knowledge, our paper is the first one to carry out an organized study on the characterization of BeeGFS performance trends and the suitability of this PFS for handling I/O workloads generated by DL applications' training phase.

7 CONCLUSION

We have taken the opportunity to explore different aspects of BeeGFS and mainly focused on its suitability for workloads posed by DL applications and frameworks. We have conducted a microscopic analysis of data and metadata I/O performance on BeeGFS and summarized our observations. In addition, we have leveraged DL applications built on top of LBANN and distributed TensorFlow to evaluate the impact of DL I/O patterns on the performance of BeeGFS. Our study offers a useful document for the HPC users who are perusing BeeGFS, especially for handling workload invoked by DL applications.

In the future, we plan to extend the evaluation with more in-depth analysis on I/O workloads posed by TensorFlow applications. Besides, we will try different types of datasets and explore the TensorFlow Dataset API more intuitively. We also aim at exploring the latest Storage Pool feature in BeeGFS for handling heterogeneous storage stack and features like buddy mirroring for built-in reliability. BeeGFS On Demand (BeeOND) used for managing burst-buffers is another promising attribute for further study. For a dataset like ImageNet, prefetching the entire dataset requires the use of less than 20% memory from each node for a 16 node run on Catalyst. However, there is scope for more research on even larger datasets that cannot be fully prefetched on most of the supercomputing facilities.

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-764078-DRAFT. This work is also supported in part by the National Science Foundation awards 1564647, 1763547 and 1822737. We are thankful to Mr. Nikoli Dryden from the University of Illinois at Urbana-Champaign for sharing his expertise on LBANN.

REFERENCES

[1] Advanced HPC announces its exclusive platinum partnership with BeeGFS. https://www.hpcwire.com/off-the-wire/advanced-hpc-announces-its-exclusive-platinum-partnership-with-beegfs.
[2] An Overview of Gradient Descent Optimization Algorithms. http://ruder.io/optimizing-gradient-descent/index.html#minibatchgradientdescent.
[3] AWI uses BeeGFS. https://www.hpcwire.com/2016/12/22/awi-uses-new-cray-cluster-earth-sciences-bioinformatics.
[4] BeeGFS. https://www.beegfs.io/content.
[5] BeeGFS as Burst Buffer. https://www.hpcwire.com/off-the-wire/beegfs-based-burst-buffer-enables-world-record-hyperscale-data-distribution.
[6] Caffe2. https://caffe2.ai.
[7] Catalyst. https://hpc.llnl.gov/hardware/platforms/catalyst.
[8] Darshan. https://www.mcs.anl.gov/research/projects/darshan.
[9] IC3 selects BeeGFS for climate modeling storage. http://www.hpcnow.com/index.php/es/company-es/news-es2/92-ic3-selects-beegfs-and-hpcnow-for-climate-modelling-storage.
[10] Intel BLAS Library. https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines.
[11] IOR. https://github.com/hpc/ior.
[12] Keras. https://keras.io.

[13] Lustre. http://lustre.org.
[14] MDTest. https://github.com/LLNL/mdtest.
[15] OrangeFS. http://www.orangefs.org.
[16] PanFS. https://www.panasas.com/panfs-architecture/panfs.
[17] PyTorch. https://pytorch.org.
[18] TensorFlow Dataset API. https://www.tensorflow.org/versions/r1.10/api_docs/python/tf/data/Dataset.
[19] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.
[20] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.
[21] Steven W. D. Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. Characterizing Deep-Learning I/O Workloads in TensorFlow. *arXiv e-prints*, page arXiv:1810.03035, October 2018.
58-59
2 notes:
[22] Fahim Chowdhury, Jialin Liu, Quincey Koziol, Thorsten Kurth, Steven Farrell, Suren Byna, Prabhat, and Weikuan Yu. Initial Characterization of I/O in Large-Scale Deep Learning Applications. 11 2018.
[23] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009.
[24] Ibrahim F. Haddad. PVFS: A Parallel Virtual File System for Linux Clusters. *Linux Journal*, 2000:5, 01 2000.
[25] Jiazhen Gu, Huan Liu, Yangfan Zhou, and Xin Wang. DeepProf: Performance Analysis for Deep Learning Applications via Mining GPU Execution Patterns. *arXiv e-prints*, page arXiv:1707.03750, July 2017.
60-61
2 notes:
[26] Mauricio Guignard, Marcelo Schild, Carlos S. Bederián, Nicolás Wolovick, and Augusto J. Vega. Performance Characterization of State-Of-The-Art Deep Learning Workloads on an IBM "Minsky" Platform. In *HICSS*, 2018.
62-63
2 notes:
[27] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.
[28] Norman Jouppi, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Cliff Young, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C Richard Ho, Doug Hogberg, John Hu, and Nan Boden. In-Datacenter Performance Analysis of a Tensor Processing Unit. pages 1–12, 06 2017.
64
chen ping
[29] Yuriy Kochura, Sergii Stirenko, Oleg Alienin, Michail Novotarskiy, and Yuri Gordienko. Performance Analysis of Open Source Machine Learning Frameworks for Various Parameters in Single-Threaded and Multi-Threaded Modes. *arXiv e-prints*, page arXiv:1708.08670, August 2017.
65
chen ping
[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM*, 60(6):84–90, May 2017.
[31] Samuel Lang, Philip H. Carns, Rob Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. 11 2009.
[32] Jianwei Li, Wei keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 39–39, Nov 2003.
[33] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng. Performance Analysis of GPU-Based Convolutional Neural Networks. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 67–76, Aug 2016.
66
chen ping
[34] Glenn K. Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J. Wright. A year in the life of a parallel file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 74:1–74:13, Piscataway, NJ, USA, 2018. IEEE Press.
[35] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling Parallel I/O Performance Through I/O Delegate and Caching System. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 9:1–9:12, Piscataway, NJ, USA, 2008. IEEE Press.
[36] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of Active Storage Strategies for the Lustre Parallel File System. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 28:1–28:10, New York, NY, USA, 2007. ACM.
[37] S. Pumma, M. Si, W. Feng, and P. Balaji. Towards Scalable Deep Learning via I/O Analysis and Optimization. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 223–230, Dec 2017.
67-68
2 notes:
[38] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Parallel I/O Optimizations for Scalable Deep Learning. pages 720–729, 12 2017.
[39] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: adaptive control of distributed applications. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*, pages 172–179, July 1998.
[40] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.
[41] 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*, MLHPC '15, pages 5:1–5:6, New York, NY, USA, 2015. ACM.
[42] F Wang, Q Xin, B Hong, S A Brandt, E Miller, D Long, and T McLarty. File System Workload Analysis for Large Scale Scientific Computing Applications. 1 2004.
[43] Yuan Wang, Yongquan Lu, Chu Qiu, Pengdong Gao, and Jintao Wang. Performance Evaluation of A Infiniband-based Lustre Parallel File System. *Procedia Environmental Sciences*, 11:316âĂŞ321, 12 2011.
[44] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.
[45] Weikuan Yu, R. Noronha, Shuang Liang, and D. K. Panda. Benefits of high speed interconnects to cluster file systems: a case study with Lustre. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8 pp.–, April 2006.
[46] Weikuan Yu, J. S. Vetter, and H. Sarp Oral. Performance characterization and optimization of parallel I/O on the Cray XT. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, April 2008.
[47] T. Zhao, V. March, S. Dong, and S. See. Evaluation of a Performance Model of Lustre File System. In *2010 Fifth Annual ChinaGrid Conference*, pages 191–196, July 2010.
[48] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Multi-Client DeepIO for Large-Scale Deep Learning on HPC Systems.
[49] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems. pages 145–156, 09 2018.

I / O Characterization and Performance Evaluation of BeeGFS for Deep Learning

Chowdhury, Fahim; Heer, Todd; Paredes, Saul; Moody, Adam; Goldstone, Robin; Mohror, Kathryn; Yu, Weikuan

01 chen ping Page 1

3/8/2019 6:44

02 chen ping Page 1

3/8/2019 6:22

03 chen ping Page 1

3/8/2019 6:31

04 chen ping Page 1

3/8/2019 6:31

05 chen ping Page 1

3/8/2019 6:51

06 chen ping Page 1

3/8/2019 6:53

07 chen ping Page 1

3/8/2019 6:53

08 chen ping Page 1

3/8/2019 6:34

09 chen ping Page 1

3/8/2019 6:35

10	chen ping 3/8/2019 6:37	Page 1
11	chen ping 3/8/2019 6:37	Page 1
12	chen ping 3/8/2019 6:39	Page 1
13	chen ping 3/8/2019 7:38	Page 2
14	chen ping 3/8/2019 7:17	Page 2
15	chen ping 3/8/2019 7:44	Page 2
16	chen ping 3/8/2019 7:45	Page 2
17	chen ping 3/8/2019 7:19	Page 2
18	chen ping 3/8/2019 7:50	Page 2
19	chen ping 3/8/2019 7:52	Page 2
20	chen ping 3/8/2019 7:52	Page 3
21	chen ping 3/8/2019 7:52	Page 3

22 chen ping Page 3
3/8/2019 7:55

23 chen ping Page 3
3/8/2019 8:24

24 chen ping Page 3
3/8/2019 8:21

25 chen ping Page 3
3/8/2019 8:22

26 chen ping Page 4
3/8/2019 8:28

27 chen ping Page 4
3/8/2019 8:45

28 chen ping Page 4
3/8/2019 8:45

29 chen ping Page 4
3/8/2019 8:46

30 chen ping Page 4
3/8/2019 8:47

31 chen ping Page 4
4/8/2019 2:04

32 chen ping Page 4
4/8/2019 2:45

33 chen ping Page 4
4/8/2019 2:13

34 chen ping Page 4

3/8/2019 8:37

35 chen ping Page 4

3/8/2019 8:37

36 chen ping Page 5

4/8/2019 2:45

37 chen ping Page 5

4/8/2019 2:38

38 chen ping Page 6

4/8/2019 2:59

39 chen ping Page 6

4/8/2019 3:04

40 chen ping Page 6

4/8/2019 3:16

41 chen ping Page 6

4/8/2019 2:57

42 chen ping Page 6

4/8/2019 2:57

43 chen ping Page 6

4/8/2019 3:16

44 chen ping Page 7

4/8/2019 3:26

45 chen ping Page 7

4/8/2019 3:33

46 chen ping Page 7

4/8/2019 4:34

47 chen ping Page 8

4/8/2019 4:43

48 chen ping Page 8

4/8/2019 4:38

49 chen ping Page 8

4/8/2019 4:39

50 chen ping Page 8

4/8/2019 4:39

51 chen ping Page 8

4/8/2019 4:43

52 chen ping Page 9

4/8/2019 4:54

53 chen ping Page 9

4/8/2019 4:56

54 chen ping Page 9

4/8/2019 5:03

55 chen ping Page 9

4/8/2019 5:06

56 chen ping Page 9

4/8/2019 5:06

57 chen ping Page 9

4/8/2019 5:07

58 chen ping Page 10
4/8/2019 5:16

59 chen ping Page 10
4/8/2019 5:16

60 chen ping Page 10
4/8/2019 5:13

61 chen ping Page 10
4/8/2019 5:13

62 chen ping Page 10
4/8/2019 5:17

63 chen ping Page 10
4/8/2019 5:17

64 chen ping Page 10
4/8/2019 5:13

65 chen ping Page 10
4/8/2019 5:13

66 chen ping Page 10
4/8/2019 5:13

67 chen ping Page 10
4/8/2019 5:17

68 chen ping Page 10
4/8/2019 5:17