Aggregating Local Storage for Scalable Deep Learning I/O

Zhao Zhang*, Lei Huang*, J. Gregory Pauloski‡, Ian T. Foster¶
*Texas Advanced Computing Center
Email: zzhang, huang@tacc.utexas.edu
‡University of Texas at Austin
Email: jgpauloski@utexas.edu
¶University of Chicago & Argonne National Laboratory
Email: foster@uchicago.edu

***Abstract*—Deep learning applications introduce heavy I/O loads on computer systems. The inherently long-running, highly concurrent, and random file accesses can easily saturate traditional shared file systems and negatively impact other users. We investigate here a solution to these problems based on leveraging local storage and the interconnect to serve training datasets at scale. We present FanStore, a user-level transient object store that provides low-latency and scalable POSIX file access by integrating the function interception technique and various metadata/data placement strategies. On a single node, FanStore provides performance similar to that of the XFS journaling file system. On many nodes, our experiments with real applications show that FanStore achieves over 90% scaling efficiency.**

I. Introduction

Deep learning (DL) is an emerging and increasingly popular application paradigm, and we see increasing use of supercomputer systems for model training [1–4]. Distributed DL training often involves large datasets and introduces heavy I/O workload. Taking image classification as an example, the ImageNet-1k dataset [5] contains 1.3 million small files (from bytes to megabytes) that are spread across 2,002 directories. Training a ResNet-50 [6] neural network usually runs for 90 epochs, which means every file will be accessed 90 times. The total 117 million file accesses are distributed across the training procedure. Depending on the scale, the concurrency of file access can be in the order of *O(N)*, where N is the number of processors, e.g., CPUs and GPUs.

A common way to build DL applications starts with datasets on a shared file system. Training programs take directories as categories and files as training data. In this straightforward way, users can easily scale DL training on more nodes with larger datasets by simply switching the data paths. However, as the dataset grows, the I/O traffic of thousands of directories and millions of files can easily saturate the shared file system due to the high access frequency, concurrency, and the sustained workload. In a typical cluster setting, such an I/O workload causes other users to experience degraded file system performance or even unresponsiveness. A second way is to encapsulate the larger number of small files in an optimized data format, such as Tensorflow's TFRecord [7]. Using the customized data format can dramatically reduce the metadata workload, but the data workload remains the same and the limited bandwidth between file system and compute nodes is a performance bottleneck. Additionally, users have to make code changes to read data from the corresponding format. Another technical workaround, copying the complete dataset to the local disks of each compute node, is only a viable solution when each local disk is large enough.

In this paper, we present FanStore, a runtime shared file system to enable efficient and scalable distributed DL training. It leverages the local storage space and interconnect to enhance the I/O capacity of existing computer clusters and supercomputers. FanStore is designed based on the findings of a profile study on distributed DL I/O behavior (Please find details in §II). FanStore exposes a POSIX file access interface with relaxed multi-read single-write consistency, so users do not have to make intrusive code changes to take advantage of the optimized I/O performance. FanStore employs the function interception technique to enable the POSIX file access interface in user space. With little overhead, I/O performance through FanStore is close to the hardware limit. FanStore preserves the global view of a dataset by broadcasting metadata and distributing file data, with remote file access as a round-trip MPI [8] message. By combining the relaxed consistency and various techniques, FanStore can scale distributed DL training to 512 nodes with over 90% efficiency.

This paper makes the following contributions:

- A distributed DL training I/O profile study, with identified opportunities for file system consistency relaxing and I/O optimizations.
- The investigation of various techniques that make high utilization of existing hardware with improved overall training performance.
- The verification of the effectiveness of FanStore's design using three real applications that cover the convolutional, recurrent, and generative adversarial network architectures.
- The open source implementation of FanStore, accessible at https://github.com/TACC/fanstore.

The rest of this paper is as follows: We review related work in §III. §IV discusses the FanStore design and implementation. Performance measurements with real applications are presented and discussed in §V. We conclude in §VI.

II. DL I/O Profile

We next review how distributed DL works in a data parallel manner and its I/O behavior. Then we summarize the I/O frequency, concurrency, and consistency. For the ease of understanding, we will use the ResNet-50 training case with the ImageNet-1k dataset implemented with Keras [9], TensorFlow [7], and Horovod [10]. Keras provides a concise programming interface with back end support of TensorFlow and other frameworks, while Horovod works at the communication layer of TensorFlow and enables distributed training. The summarized I/O pattern applies to the three example applications discussed in this paper and other widely used DL frameworks (e.g., PyTorch and Caffe) and applications.

A. I/O in Data Parallel Distributed DL

Data parallelism is a commonly used approach distributed DL training in which the data of each mini-batch are scattered to processors while the model (parameters) is replicated. In the beginning of training, the program will traverse the metadata in the training and validation directories to calculate the number of files, then determine the number of iterations in each epoch and in total. From the first iteration, each node will concurrently read a mini-batch of training files. The size of the mini-batch is a user specified parameter, and the mini-batch size is critical to final convergence: mini-batch sizes larger than published numbers are divergence prone during training. The training process then carries out forward computation along the neural network, and computes the loss. Then, each node will use the computed loss to calculate the gradients with regard to the parameters, which is referred to as a back-propagation algorithm. Since each node has a different set of training items, the derived loss and respective gradients are different across nodes. Usually, the training process calls the Allreduce collective communication primitive to compute the sum (then mean) of the gradients before applying the stochastic gradient descent (SGD) method to update parameters. One iteration finishes after all parameters are updated. If the end of the current iteration overlaps with the end of an epoch, the training process may validate the model on the validation dataset and checkpoint the model to file system. The complete training process iterates until all epochs finish.

B. Global Dataset View

With the global dataset view, every compute node sees the same directory structure and file contents. Maintaining a global view of the training dataset is critical for convergence in distributed DL training. Another way to store a large dataset in local storage across multiple nodes is to let each node store an exclusive subset, which results in the partitioned dataset view. Figure 1 shows the last 30 epochs of the 90-epoch ResNet-50 training on the ImageNet-1k dataset with both views. The experiment runs on 16 Nvidia GTX 1080 Ti GPUs with batch size of 512. The partitioned dataset view loses ∼4% of validation accuracy, which is unacceptable.

C. Metadata Access

At a high level, the metadata access of distributed DL has high volume and concurrency. Metadata are accessed in two places, once at the beginning of the training process, where the program gathers information on the training and validation dataset, and again during each iteration, where the program launches multiple threads per process to read files. For example, each Keras process uses four I/O threads by default.

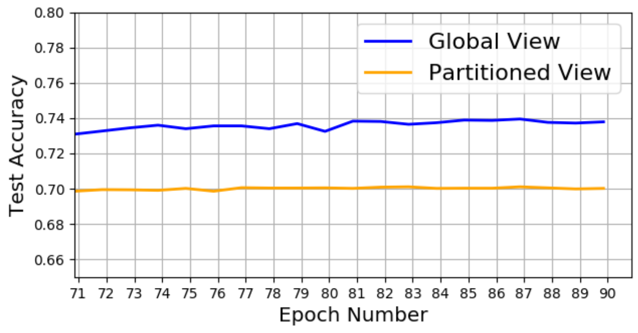


Fig. 1: Validation Accuracy of ResNet-50 on ImageNet-1k Dataset with Global and Partitioned View. Please note this is not a standard ResNet-50 benchmark run with baseline validation accuracy of 74.9%. It is a less fine-tuned run to show the training divergence with partitioned dataset view.

In the ImageNet-1k dataset, each process accesses the metadata of 2,002 directories and 1.3 million files at the beginning of training. On a cluster of N GPUs, usually running one process per GPU, there will be $4N$ simultaneous *readdir()* or *stat()* operations. The highly concurrent metadata access of large volume can easily saturate the metadata server in a traditional shared file system such as Lustre [11] and GPFS [12].

D. Data Access

The file data access of distributed DL is highly concurrent and persists through the whole training process. The individual file size ranges from a few bytes to a few mega bytes. Modern DL frameworks such as Keras and Caffe support asynchronous I/O, where the I/O overlaps with computation for faster training speed, often referred to as data prefetch. Assuming a cluster of N GPUs, the mini-batch size is specified proportionally to GPU count to maintain high utilization of the hardware. In the ResNet-50 example, we use a mini-batch size of $64N$. Thus the data access is in the form of $4N$ concurrent threads reading $64N$ files for each iteration. When a file is read, it is read sequentially and completely. There is no random read that starts from an arbitrary offset nor partial read from a file. Each ResNet-50 iteration runs for ∼300 ms. If the I/O performance cannot keep up with the pace of computation, there will be wasted hardware cycles. This data access pattern persists until the final iteration. Since it is not rare to use tens, hundreds, or even thousands of GPUs or CPUs for distributed DL training, it is critical for the file system to keep up with the computational hardware in a scalable manner.

Besides accessing the training and validation dataset, distributed DL may also write to the file system. The master process can periodically write the model to the file system as a checkpoint. In applications such as generative adversarial networks, the training program may output the generated synthetic data to the file system for human examination. In all of these case, the write operations are writing to new files without overwriting file or concurrently writing to the same file. (Although the checkpoint can be overwritten, it is common practice to write to a file labeled with epoch number.) Unless the training program starts from the last checkpoint, these written files are not read again by the training program.

The I/O behavior of distributed DL training shows a multi-read single-write consistency pattern. An input directory or file can be accessed by multiple processes/threads simultaneously, while each output file is written exclusively by a single process/thread with no further access. Thus it is free of read-after-write or write-after-write hazards.

III. Related Work

The problem of massively concurrent access to many small files has been studied extensively in the HPC community.

One type of optimization focuses on distributed metadata server design [13–15]. GIGA+ [13] use a dynamic metadata server scaling design to deal with incremental file count growth, while ZHT [14] and GlusterFS [15] propose a static zero-hop hash table for scalable metadata management. Both metadata server designs can achieve decent scaling performance for massively concurrent file I/O. However, since metadata are spread across servers, directory access with *readdir()* has to communicate with all servers to gather information. Thus both designs will result in slow directory access for distributed DL training, given the large metadata volume and highly concurrent access.

A second optimization technique seeks to relax file system consistency. For example, HDFS [16] restricts file writing to appending, while AMFS implements multi-read single-write consistency [17] to provide highly efficient and scalable I/O support for workflow applications on supercomputers. In the context of distributed DL training, further relaxing file system consistency can achieve better performance, as the output files are rarely read by the training program.

To preserve the POSIX file access interface, users usually have to mount these file systems through FUSE [18] in user space. The overhead through FUSE is not trivial [19]. In our work, we use the system call interception technique to remedy this performance issue.

IV. Design and Implementation

We next discuss the FanStore architecture and implementation. In general, FanStore is concerned only with enabling read accesses to data; it leaves write operations to the shared file system, as typically only one training process writes checkpoints or sample output (as discussed in §II).

A. Design Requirements

FanStore, at a high level, uses local storage space and interconnect to reduce the I/O traffic between compute nodes and shared file system. As discussed in §II, the requirements of FanStore are:

- a global namespace,
- high volume metadata access with high concurrency,
- high volume data access with high concurrency,
- a POSIX file access interface in user space,

To address these requirements, FanStore exploits different data placement and caching strategies for metadata and file data. It exposes the POSIX file access interface via the function interception technique which works in user space without significant loss of performance. It also implements a general lossless compression algorithm to support data compression for all data types.

B. Architecture

FanStore has two major components: the data preparation tool and the FanStore daemon. Users use the data preparation tool to package the dataset on the shared file system so that FanStore daemons can load the dataset efficiently into the local storage space. The FanStore daemon maintains runtime information on system status, and manages two types of data: metadata and file data. FanStore can use local storage, e.g., RAM, RAM disk, SSD, and HDD, as the storage backend. Figure 2 provides an overview of the FanStore architecture.

FanStore places metadata in RAM and file data in storage backend by default, and maintains several data structures (e.g., open file counter table) for high-throughput file access. Worker threads in each FanStore daemon handle file system requests intercepted from the DL training program. These worker threads manipulate the metadata stored locally and retrieve file data either from local storage or remote node via interconnect.

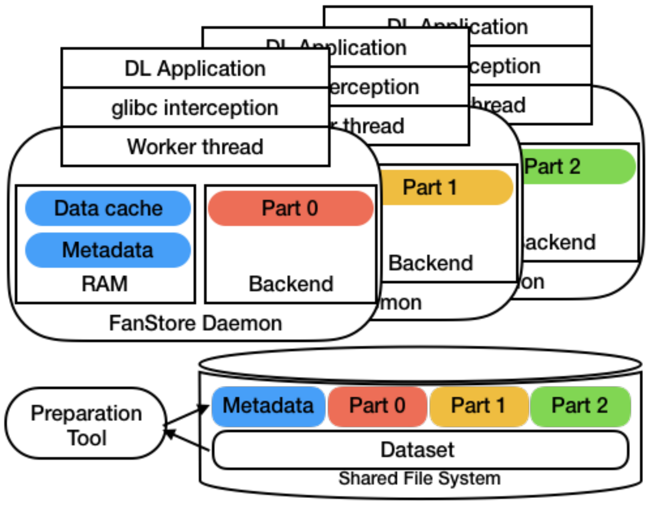


Fig. 2: FanStore architecture overview

C. Data Preparation

FanStore requires a data preparation step before training, where a user will have to pass into the data preparation tool a list of all files involved. Large datasets originally stored in the shared file system are then reorganized into partitions. Each partition contains an exclusive subset of the files. Table I shows the data layout in a partition. Each partition starts with an integer (four bytes) of the file count, followed by a 256 byte long file name, a 144 byte long stat structure as the file's metadata, and the data size after compression. Then the actual data are appended. The rest of the files are organized continuously.

Upon loading, FanStore traverses each partition to dump the actual data into local storage backend and builds an index of file path and storage place, which includes both the node id and the data offset. Such a design dramatically reduces the metadata count compared to the case of storing the files on the shared file system.

When using FanStore, the original relative file path is prefixed with a predefined mount point. For example, the path *img_train* on the shared file system will be available as */tmp/fs_user_id/img_train*. The internal structure remains unchanged, and all nodes share the same view of the namespace of training and validation datasets.

D. Metadata Management

FanStore keeps metadata in a hashtable in RAM. Each entry has the file path as the key and the metadata record as the value. In addition to the standard POSIX information, each metadata record maintains the file location. All the metadata of input files are replicated across nodes to maintain the global view of the dataset. In each FanStore process, the file metadata of a directory is preprocessed and cached in a hash table to allow *readdir()* to return immediately.

E. Data management

We have observed two access patterns for training and validation datasets: the training dataset is usually larger than the validation dataset, and each training process can randomly access files in the training dataset in each iteration. In contrast, a subset of the validation dataset is read by each process during validation, and this is usually done at the end of each epoch. Based on this observation, FanStore allows users to specify a directory so that all files in this directory will be replicated across all nodes. This replication can improve validation performance due to higher locality hit rate.

As discussed in §II-D, an input file is read completely in sequential order. Thus, FanStore stores each input file as a byte array without a block abstraction or striping.

Upon receiving a file open request, the FanStore worker thread checks metadata to determine its availability and location. If the file exists in local storage, the thread pulls the file to memory and then returns its content. If the file exists on a remote node, the thread communicates with that node to retrieve its content. If the file does not exist, the thread returns an error code. The communication in FanStore is implemented by using MPI *send()* and *recv()*.

Each file in the training dataset has a uniform probability of being accessed, and each file access is independent of other file access operations. Thus, a conventional cache will not perform well: if it holds 20% of the dataset, then the cache hit rate will be 20% in expectation in each iteration. FanStore therefore implements a simpler caching mechanism: a file is cached in memory only until the file descriptor is released. We intend to use as little RAM space as possible, given that the training process can be memory hungry. Occasionally, multiple training processes on the same node can access the same file simultaneously. Closing the file descriptor or evicting the file from cache can result in a stale state in other process. FanStore maintains a file counter table in memory with the file path as the key and the number of processes that are currently accessing it as the value. When a file is accessed, the corresponding counter increases by one. Upon the release of a file descriptor, the corresponding counter decreases by one. If the counter is zero, the file content is evicted from cache.

When scaling out distributed DL training, there will be more aggregated local storage space, though the dataset may not fit in a single node. In this case, FanStore allows users to specify a replication factor of N, so that each node can host N different partitions.

F. POSIX Interface

Users on clusters usually do not have root privilege making it infeasible to mount FanStore as a kernel module. Exposing POSIX file access interface through FUSE is a viable solution. However, FUSE introduces non-trivial overhead as the system call crosses the user-kernel boundary [20]. Such overhead results in significant slowdown in DL training.

To eliminate the performance overhead while preserving user-space usability, FanStore implements the POSIX interface using the function interception method [21]. I/O operations from applications eventually call the low level functions such as *open(), close(), stat(), read(), write()* in the GNU C Library (glibc). The function interception method replaces the first several instructions of the low level functions in glibc and forces them to jump into a user space library where FanStore logic is implemented. In this way, all I/O related function calls stay in user space. Our current implementation supports x86_64 and POWER9 architecture.

V. Experiments

We use application experiments to evaluate the effectiveness of the FanStore design and implementation. We present here results of those experiments.

A. Hardware and Software Stack

We use two clusters for all experiments reported here. The first cluster, Maverick2 (here, **GPU Cluster**) has 24 nodes, each with one Intel Xeon E5-2620 CPU, four Nvidia 1080 Ti GPUs, and a 60 GB local SSD. The nodes are connected by a Mellanox FDR Infiniband interconnect with up to 56Gbps bandwidth and a sub-micro second latency. The second cluster, Stampede2 (here, **CPU Cluster**) has 512 nodes, each with two Intel Xeon Platinum 8160 processors and 144 GB local SSD, connected by a 100Gb/sec Intel Omni-Path (OPA) network with a fat tree topology.

On the GPU cluster, DL frameworks use CUDA 9.0, CUDNN 7.0, NCCL 2.1.4. Both clusters run CentOS 7.4, Intel MPI 17.0.3, TensorFlow 1.8.0, TensorLayer [22] 1.9.1, Keras [9] 2.2.2, and Horovod [10] 0.13.4.

TABLE I: Data layout in a partition

field	num_files	file_name	stat	misc	data	file_name	stat
byte_range	0 - 3	4 - 259	260 - 403	404 - 411	412 - 411+data.size	...	...

TABLE II: Characteristics of application datasets

App	Dataset Name	# files	# dirs	total_size	file_size
ResNet-50	ImageNet	1.3 million	2002	140 GB	KB–MB
SRGAN	EM	0.6 million	6	500 GB	MB
FRNN	RS	0.6 million	1	1.7 TB	KB

B. Applications

We use three real world applications to evaluate FanStore performance: ResNet-50 with ImageNet-1k, super resolution generative adversarial network (SRGAN [23]) with a private electron microscopy dataset [24], and fusion recurrent neural network (FRNN [25]) with a reactor status dataset. Table II summarizes the associated file counts and data sizes.

C. Data Preparation Cost

We first measure the time consumed by data preparation when working with each of the three application datasets. Preparing the ImageNet-1k, SRGAN, and RCNN datasets takes 13, 11, and 14 minutes, respectively, on a single Intel Xeon E5-2680 CPU node. These data preparation costs are incurred only once, and are trivial compared to the subsequent training times of hours or days.

D. Single Node Performance

We next compare FanStore performance against that of alternative storage options and techniques on a single node. Figure 3 compares the read performance of FanStore and TFRecord on a single compute node. In both cases, data is stored in the local SSD. Across the three datasets, FanStore reads 4–6× faster than TFRecord.

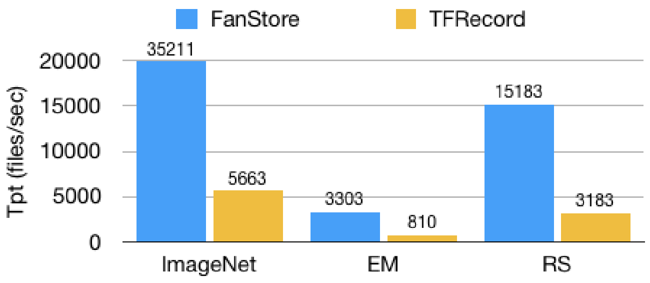


Fig. 3: Read throughput: FanStore vs. TFRecord

Figure 4 shows ResNet-50, SRGAN, and FRNN performance with data stored in FanStore, SSD through XFS, SSD-fuse through bindfs [26], and Lustre (SFS). We report application performance in items per second (items/sec). ResNet-50 achieves a sustained throughput with FanStore of 544 files/s, which is 5.3% faster than on SSD (due to directory metadata caching) and 2.0× faster than SFS. On the other hand, SRGAN shows identical performance across all options. This is due to the significant amount of computation performed in each iteration. Even with the fastest storage in this case, the sustained throughput is only 49 items/s for SRGAN. Similarly, FRNN performs almost identically across the four storage options.

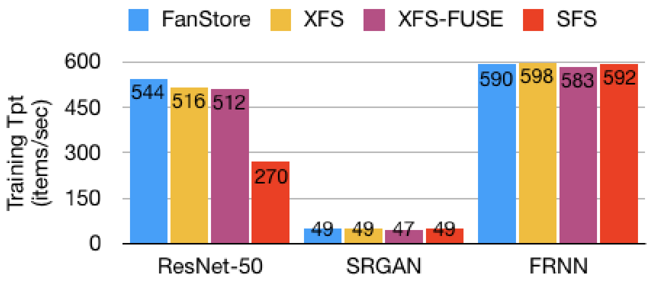


Fig. 4: Training throughput (Items/sec) with data stored on different hardware and software

E. Multi-node Performance

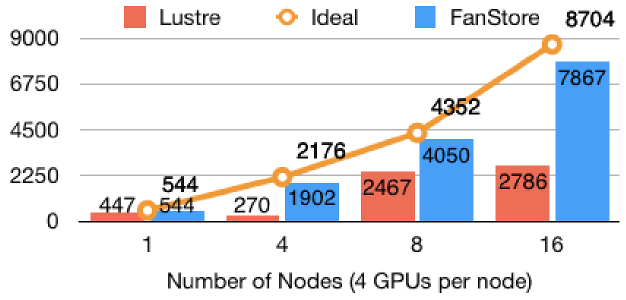
In this experiment, we measure the real application performance across scales on the GPU cluster or CPU cluster to verify the effectiveness of FanStore's design of scalability. If not otherwise specified, each file has only one copy across compute nodes in subsequent experiments. The performance is measured in items/sec.

Figure 5 presents ResNet-50 performance across scales on the GPU and CPU cluster. With the dataset stored in Lustre, the training performance does not scale with the GPU count. At the scale of 64 GPUs, the scaling efficiency with Lustre is only 32.0%. On the other hand, with FanStore, the four-node training runs 76.1% faster than Lustre, and the sustained scaling efficiency is almost 100% on 64 GPUs compared to that on 16 GPUs. The scaling efficiency with that of one node as baseline is 90.4%. On 512 compute nodes in the CPU cluster, the scaling efficiency is 92.2%.

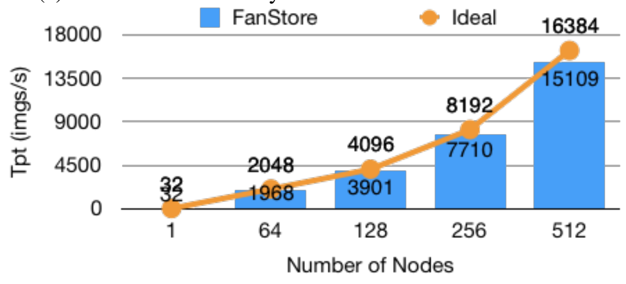
In reality, the shared file system can not scale in a linear fashion, and the performance can fluctuate depending on the workload [27]. In contrast, FanStore's performance relies only on the interconnect and local storage. It is less prone to be affected by I/O from other jobs on the same cluster.

Figure 6 and Figure 7 show the scalability of SRGAN on the GPU cluster and FRNN on the CPU cluster, respectively. SRGAN scales with 97.9% efficiency from one node to 16 nodes (64 GPUs). The close to linear scaling performance attributes to the higher computation requirement in SRGAN compared to that of ResNet-50. FRNN shows 93.3% efficiency on 64 nodes compared to the baseline on one node.

On both clusters, FanStore enables highly scalable performance across increasing node counts. The preprocessed dataset has a fixed number of partitions: 48 for the GPU cluster and 512 for the CPU cluster. These files are loaded to local



(a) ResNet-50 scalability with FanStore on GPU Cluster.



(b) ResNet-50 scalability with FanStore on CPU Cluster.

Fig. 5: ResNet-50 scalability when using FanStore for data access, on GPU and CPU clusters.

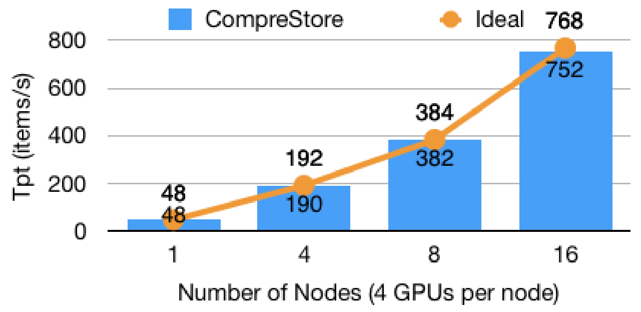


Fig. 6: SRGAN scalability with FanStore on GPU Cluster

storage only at the beginning of the training process. Thus the I/O workload to/from the shared file system remains constant across different scales of training.

Even though SRGAN and FRNN perform similarly with data stored in FanStore and Lustre at small scale, using FanStore dramatically reduces the I/O workload. E.g, the 0.6 million file inputs during the 200-epoch SRGAN training is now served with 48 large file reads and 120 million network round trip MPI messages. Such I/O reduction results in less risks for shared file system performance degradation or unresponsiveness.

F. Discussion

In addition to real applications, we also run a benchmark with varying file sizes of 128 KB, 512 KB, 2 MB, and 8 MB across the scales on the GPU and CPU clusters. The detailed results are not presented due to the limited space in this paper. FanStore's capacity of I/O improvement is way beyond ResNet-50, SRGAN, and FRNN. ImageNet-1k files have an average size of 108 KB. Compared to the benchmark

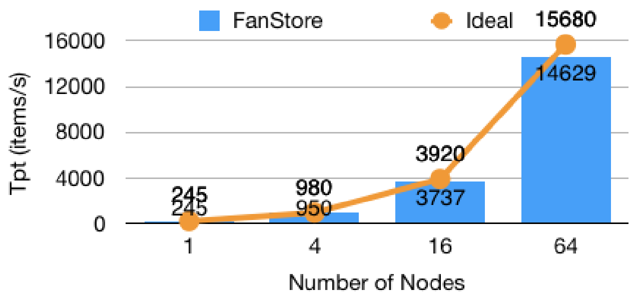


Fig. 7: FRNN scalability with FanStore on CPU Cluster

throughput with 128 KB file on the GPU cluster, ResNet-50 only uses 9.4% (7867 items/sec vs. 84233 items/sec) of the sustained peak throughput of FanStore. ResNet-50 has 50 layers and 1.5 billion single precision floating operations per image indicating FanStore can keep the scaling curve of an application with 10.6× less computation per image as ResNet-50. From the perspective of processors, FanStore can keep the ResNet-50's scalability with 10.6× more powerful hardware.

Up to 512 nodes on the CPU cluster, FanStore does not hit the ceiling of scalability (with over 90% scaling efficiency). This is due to the relaxed I/O consistency and the distributed metadata and data service. FanStore's performance largely relies on the local storage and interconnect. With proper replication setting, FanStore can keep up the scalability with the underlying interconnect.

VI. Conclusion

We have presented FanStore, a transient runtime object store that leverages local storage and interconnect in computer clusters to enable efficient and scalable distributed deep learning training. It dramatically enhances the I/O capacity on existing hardware/software stack. FanStore incorporates various data placement strategies and techniques of function interception, and collective data management. FanStore preserves the global namespace of the dataset and the POSIX file access interface in user space. Real applications show that FanStore achieves read performance that is close to XFS on a single node, and, FanStore scales up to 512 compute nodes with over 90% weak scaling efficiency. The design and implementation of FanStore dramatically enhances the capability of existing hardware and software in supporting the emerging distributed deep learning applications.

References

[1] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *CoRR*, vol. abs/1802.09941, 2018. [Online]. Available: http://arxiv.org/abs/1802.09941

[2] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. ACM, 2018, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/3225058.3225069

[3] V. Codreanu, D. Podareanu, and V. Saletore, "Scale out for large mini-batch SGD: Residual network training on ImageNet-1K with improved accuracy and reduced time to train," *arXiv preprint arXiv:1711.04291*, 2017.

[4] T. Akiba, S. Suzuki, and K. Fukuda, “Extremely large minibatch SGD: Training ResNet-50 on ImageNet in 15 minutes,” *arXiv preprint arXiv:1711.04325*, 2017.

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2009)*, 2009, pp. 248–255.

[6] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the mpi message passing interface standard,” *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[9] F. Chollet *et al.*, “Keras,” 2015, https://keras.io/.

[10] A. Sergeev and M. D. Balso, “Horovod: fast and easy distributed deep learning in TensorFlow,” *arXiv preprint arXiv:1802.05799*, 2018.

[11] P. Schwan *et al.*, “Lustre: Building a file system for 1000-node clusters,” in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386.

[12] F. B. Schmuck and R. L. Haskin, “Gpfs: A shared-disk file system for large computing clusters.” in *FAST*, vol. 2, no. 19, 2002.

[13] S. Patil and G. Gibson, “Scale and concurrency of GIGA+: file system directories with millions of files,” in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*. USENIX Association, 2011, pp. 13–13.

[14] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, “Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” in *Parallel & distributed processing (IPDPS), 2013 IEEE 27th international symposium on*. IEEE, 2013, pp. 775–787.

[15] A. Davies and A. Orsaria, “Scale out with glusterfs,” *Linux Journal*, vol. 2013, no. 235, p. 1, 2013.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 2010, pp. 1–10.

[17] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. Foster, “Design and analysis of data management in scalable parallel scripting,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC12)*. IEEE Computer Society Press, 2012, p. 85.

[18] M. Szeredi, “Fuse: Filesystem in userspace,” http://fuse.sourceforge.net.

[19] B. K. R. Vangoor, V. Tarasov, and E. Zadok, “To fuse or not to fuse: Performance of user-space file systems.” in *FAST*, 2017, pp. 59–72.

[20] Y. Zhu, T. Wang, K. Mohror, A. Moody, K. Sato, M. Khan, and W. Yu, “Direct-fuse: Removing the middleman for high-performance fuse file system support,” in *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2018, p. 6.

[21] G. Hunt and D. Brubacher, “Detours: Binaryinterception of win32 functions,” in *3rd usenix windows nt symposium*, 1999.

[22] H. Dong, A. Supratak, L. Mai, F. Liu, A. Oehmichen, S. Yu, and Y. Guo, “Tensorlayer: a versatile library for efficient deep learning development,” in *Proceedings of the 2017 ACM on Multimedia Conference*. ACM, 2017, pp. 1201–1204.

[23] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. P. Aitken, A. Tejani, J. Totz, Z. Wang *et al.*, “Photo-realistic single image super-resolution using a generative adversarial network.” in *CVPR*, vol. 2, no. 3, 2017, p. 4.

[24] L. Fang, F. Monroe, S. W. Novak, L. Kirk, C. Schiavon, B. Y. Seungyoon, T. Zhang, M. Wu, K. Kastner, Y. Kubota *et al.*, “Deep learning-based point-scanning super-resolution imaging,” *bioRxiv*, p. 740548, 2019.

[25] J. Kates-Harbeck, A. Svyatkovskiy, and W. Tang, “Predicting disruptive instabilities in controlled fusion plasmas through deep learning,” *Nature*, vol. 568, no. 7753, p. 526, 2019.

[26] “bindfs,” https://bindfs.org/.

[27] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, “Characterizing output bottlenecks in a supercomputer,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 8.