PREPRINT - The final publication is available at *https://ieeexplore.ieee.org/document/9556043*

Streamlining distributed Deep Learning I/O with ad hoc file systems

Frederic Schimmelpfennig*, Marc-André Vef*, Reza Salkhordeh*, Alberto Miranda†, Ramon Nou†, André Brinkmann*

* *Johannes Gutenberg University Mainz, Mainz, Germany*
† *Barcelona Supercomputing Center (BSC), Barcelona, Spain*
{frschimm, vef, rsalkhor, brinkman}@uni-mainz.de, {alberto.miranda, ramon.nou}@bsc.es

***Abstract*—With evolving techniques to parallelize Deep Learning (DL) and the growing amount of training data and model complexity, High-Performance Computing (HPC) has become increasingly important for machine learning engineers. Although many compute clusters already use learning accelerators or GPUs, HPC storage systems are not suitable for the I/O requirements of DL workflows. Therefore, users typically copy the whole training data to the worker nodes or distribute partitions. Because DL depends on randomized input data, prior work stated that partitioning impacts DL accuracy. Their solutions focused mainly on training I/O performance on a high-speed network but did not cover the data stage-in process, for example.**

We show in this paper that, in practice, (unbiased) partitioning is not harmful for distributed DL accuracy. Nevertheless, manual partitioning can be error prone and inefficient. Typically, data must be unpacked and shuffled before it is distributed to nodes. We propose a solution that features both: efficient stage-in and fast access to a global namespace to prevent biases. Our architecture is based around an ad hoc storage system relying on a high-speed interconnect allowing an efficient stage-in of DL data sets into a single global namespace. Our proposed solution does not limit access to parts of the data set or relies on data duplication, also relieving the HPC storage system. We obtain high I/O performance during training and ensure minimal interference with communication of the learning workers. The optimizations are transparent to DL applications and their accuracy is not affected by our architecture.

***Index Terms*—Deep Learning, HPC, file system, data parallel**

I. Introduction

Deep Learning (DL) techniques are used by an increasing range of scientific disciplines and are responsible for many innovations in both industry and research. The fundamental technologies and mechanisms in DL and other data-driven applications rely on ever-increasing huge data sets, raising the technical demands on systems that run such applications. The European Centre for Medium-Range Weather Forecasts (ECMWF) reported a 45% growth for over 100 petabytes of data already in 2014 [1]. In climate and earth sciences, for example, prediction models rely on training of terabytes of data [2]–[5]. Processing such huge amounts of data is therefore one of the major challenges of these fields.

DL techniques employ encapsulated, dedicated solutions like NVIDIA DGX [6] to perform model training. They offer high processing capabilities on a single machine, combined with medium-sized local storage with up to 16 NVMe drives. However, if the requirements of DL applications exceed their capabilities, more scalable infrastructures have to be used. Because of this reason, *High-Performance Computing* (HPC) has become a cornerstone for these applications in recent years. For example, the training duration for classifiers of the popular ImageNet [7] data set reduced from hours [8] to 11 minutes [9] down to 2 minutes and below [10], [11] within only 3 years. This is mainly due to the availability of highly distributed setups using more than 3,000 Tesla V100 GPUs and high-speed interconnects.

HPC storage systems optimize mostly toward sequential access patterns on large files. DL applications, however, have different access patterns [12], which parallel file systems (PFS), e.g., GPFS [13] or Lustre [14], in HPC environments cannot efficiently service. Therefore, for such DL applications, it is common to store local data set copies on all compute nodes.

Depending on the technique and size of the data, we observed that copying data sets into the compute nodes before they are processed (called *stage-in*) can severely impair the performance of the PFS, affecting all users of the file system. One common solution used in practice is to split the data set into several partitions [2], [15], [16], also called *data shards*, and then distribute them across all nodes. This technique becomes a necessity when the data set is larger than the available node-local storage space. Because shards partition the data set, some previous work was concerned about the training accuracy when using shards or did not mention them as a valid technique [17]–[20], developing methods to use the network interconnect to access training data elements from other compute nodes. However, we found that shards do not categorically cause a penalty for training accuracy if the data is staged into the nodes without biases. This seems to be a common misconception in the systems field, and, to the best of our knowledge, we are the first to investigate the impact on training accuracy when using shards.

Nevertheless, using shards without biases can be challenging to implement in practice. Therefore, an efficient and flexible stage-in process must be considered as an integral part of the overall workflow. Yet, prior work focused mainly on read performance during the training process, requiring extensive application modifications or customized solutions for

a single use case that cannot be generalized [18], [21], [22].

In this paper, we propose a workflow using distributed *ad hoc file systems* providing a shared global namespace across all training nodes and hence unbiased access to all training data. We have used *GekkoFS* [23] as an exemplary ad hoc file system that leverages on the high-speed network interconnect for fast inter-node communication. The file system runs in user space on a per-job basis and does not require administrative support to operate. It can be transparently used by existing DL frameworks without any modifications as well as by custom solutions. Our experiments show that our approach can achieve similar performances compared to having full data set copies available at node-local storage. Moreover, we demonstrate efficient and scalable stage-in techniques minimizing the PFS load and the impact on other users.

Therefore, our contributions are as follows: 1) We show the impact on DL training accuracy using shards in biased and unbiased cases; 2) We present how ad hoc file systems can streamline distributed DL workflows without any modification to any component of the DL workflow; 3) We evaluate both data set stage-in and training I/O using fast network fabrics, including a scalability and interference analysis of various DL configurations.

First, we present the background of distributed DL, its I/O challenges, and the related work in Sections II and III. Section IV demonstrates the usage of ad hoc storage systems in DL workflows, and Section V evaluates our proposed solution. Section VI concludes this paper.

II. Background

This section provides a general introduction to the (distributed) deep learning (DL) topic. Next, we discuss the DL frameworks and tools which are relevant in the context of this paper. Finally, we present the typical I/O behavior of DL applications and explain data set *shards*.

a) Distributed Deep Learning: Deep Learning is the process of optimizing connected parameters of a deeply layered model to obtain predictions of increasing accuracy. Typically, a large amount of training samples are repeatedly propagated in forward and backward passes through the model. This allows making corrections of the parameters based on the correctness of the model's prediction when compared to before known, correct labels. Common model optimizers are based on Stochastic Gradient Descend (SGD) [24] methods and, therefore, rely on random selection of the input data [25].

Computing the gradients and applying them to the model is most efficient when using GPUs or other special accelerator hardware, e.g., TPU [26]. To ensure efficient usage of the hardware's parallel capabilities, multiple training samples are used within a single training step, known as the *batch size*. Within a *training epoch*, each sample is applied once to the model. In most cases, multiple training epochs are run to reach converging accuracy levels. Configuring DL models to achieve high accuracy levels requires repeatably adjusting so-called *hyper parameters* which modify the learning rate, general optimizer settings, layer widths, and layer depths, among many others. Hence, finding the appropriate configurations for a given model often results in a trial and error loop. Moreover, various techniques are applied to further increase the model accuracy, such as data augmentation describing the usage of slightly on-the-fly modified training data (e.g., shifted or rotated images), which improve the generalization of the predictions. Therefore, because of the sheer amount of available options to adjust a training model for each individual goal, it is important to reduce both the training time and the necessary resources for each run.

There are two main approaches for distributing a DL application across multiple GPUs or nodes: via model parallelism or data parallelism [27]. In the former, the model itself is split into parts which are trained independently. Since this process can be highly specialized, data parallelism is usually used in practice. For data parallelism, each worker, e.g., the GPU, has a copy of the model. Each model is trained in parallel with different data, and it is synchronized between each step.

b) Frameworks and Tools: DL frameworks like Tensorflow [28], Caffee [29], or PyTorch [30] aim to ease the implementation of DL applications and abstract a wide range of accelerator hardware for user-friendly access. They provide interfaces for implementing backward and forward passes and for gradient descend operations.

In the distributed setting, the gradients each worker computes must be synchronized before each training step. This applies to both cases when multiple GPUs are used on a single machine or in multiple node setups. In the above-listed frameworks, however, the support for synchronizing the gradients via all-reduce operations is limited. As a result, frameworks like Uber's Horovod [31] have become a popular option to enable support for multiple GPUs or multiple nodes on top of the DL framework itself. With such a framework, users can adjust how model synchronization via *all-reduce* is performed (e.g., as a ring or hierarchy), or how much gradient values are compressed. Horovod supports NVIDIA NCCL [32] allowing GPUs to use the high-speed network fabric.

c) Deep Learning I/O and data set shards: During training, the required data for the next step is typically loaded by the CPU while the GPU is computing and synchronizing the current step. Since randomness of the training data is required, a HPC system's storage would have to serve small accesses [12] in a randomized pattern. Nevertheless, the underlying HPC storage systems are not designed for such access patterns [33], [34]. Hence, the data set is first copied (staged-in) into node-local storage. Instead of copying the complete data set, it can also be split into *shards* that are distributed across the corresponding nodes. Using shards, however, restricts a node's training sample combinations to a static part of the data set. This might result in an overall accuracy loss as accesses are not completely randomized. Therefore, the encapsulation of samples may lead to gradients not targeting a common optimum.

One example for this problem is a classifier that decides between dog and cat images. Such a classifier could be trained in a data parallel manner using only two workers. Through

insufficient shuffling, the first worker only processes single-colored dogs and cats while the second worker processes multi-colored ones. Before model synchronization, two gradients are produced, pointing at the optimum of classifying either single-colored or multi-colored dogs and cats. During synchronization via all-reduce, the gradients are averaged, but the resulting gradient may not point at the optimum of a general dog and cat classifier. If each worker have had access to the full dataset, minor false optima would be corrected in further epochs. However, in the above example, the bias would be present in each epoch causing a general accuracy loss.

Theoretically, the most extreme case of using shards would be a shard size equal to the node's batch size, resulting in the same gradients in every training step. This is not the case, when using the same batch size with random selection from the whole data set. Moreover, data sets, especially scientific data[1], often come in packages that are sorted by various properties, e.g., label, date, location, metrics or measurement settings. If such packages only follow a basic partitioning, this would lead to strong biases of the shards, also impacting the accuracy. Therefore, the elements must be unpacked and shuffled before the training starts. Due to the many kinds of biases, shuffling the data correctly for using shards can be a cumbersome process. Hence, in general, avoiding shards can be beneficial for most DL pipelines. Figure 1 summarizes a typical training workflow of the initial stage-in phase and the following training steps.

Although DL frameworks offer tools for building up I/O pipelines, they do not support data distribution or accessing training elements in another node. They prefer that all the data is uniformly accessible. TensorFlow's `tf.data.Dataset` objects[2], for instance, allow for various modifications, like randomization, prefetching, and caching. However, their `shard()` operation only selects a subset out of an existing `Dataset` object, limiting its use to distributing data to multiple workers within the same process or node. Via optimized `tfRecord` [3] packages, TensorFlow can reduce the overhead of accessing small files, yet splitting packages is not part of the design. This would, consequently, result in similar limitations when using shards of the raw data set.

III. Related Work

In this section, we present the related work and also discuss other custom deep learning approaches aiming to solve the I/O bottleneck. Further, we present related work regarding methods for model accuracy evaluation.

a) Large-scale deep learning: For DL researchers and developers, the capabilities of the I/O subsystem are essential, especially in a distributed setting. For instance, custom I/O solutions for stage-in and training in Kurth et al. [2] were one of their main contributions when they performed *exascale* level DL on a 3.5 TiB climate science data set using more than

[1]For instance, climate AI research data sets listed at http://mldata.pangeo.io/

[2]https://www.tensorflow.org/api_docs/python/tf/data/Dataset

[3]https://www.tensorflow.org/tutorials/load_data/tfrecord

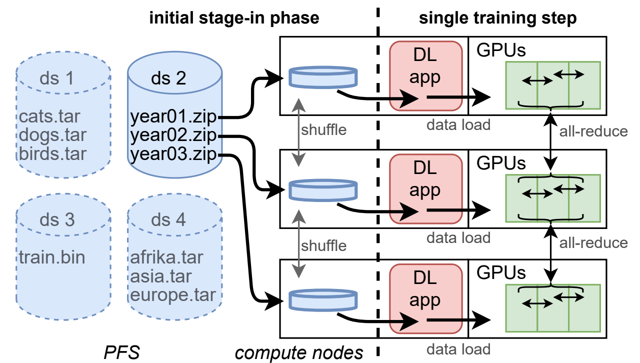


Fig. 1: Classical workflow: Initial stage-in of data set copies or shards followed by node-local reads during training.

25K GPUs. Oyama et al. [16] discussed the challenge staging-in a 10 TiB cosmology data set from the PFS, which quickly became the dominant portion of their training workflow. For both works, the situation was aggravated by the fact that the data sets could not fit into a single node's storage. Their solutions used highly specific routines that read, shuffled, and distributed the data. Nevertheless, their stack is only reusable for similar process pipelines and data layouts, requiring other projects to find another I/O solution. Overall, their work shows that DL can be significantly held back without a dedicated I/O solution, especially in highly distributed scenarios.

b) Deep Learning I/O: The systems community addressed DL I/O challenges in various approaches. Chowdhury et al. [19] evaluated the usage of the BeeGFS file system in DL workflows. BeeGFS offers a global namespace across all compute nodes combining the storage space of all nodes' storage devices. However, managing a kernel-based file system on-demand in a cluster environment can be challenging as it requires administrative support. Another recent study developed a DL-specific I/O layer that uses NVMe-over-fabric to access data from other compute nodes [18]. They provided training-related features, e.g., optimized reading of batches containing random samples. But, their API does not follow the POSIX standard resulting in custom changes to established tools and frameworks. FanStore [17] aims to leverage inter-node I/O for DL training. The authors focus on training performance, featuring additional ideas like transferring compressed data for optimizing throughput. For the training data stage-in, elements need to be assigned to the target nodes, limiting the efficient stage-in process to cases where the PFS's data layout matches exactly how the DL application accesses it. For instance, this is not the case for commonly packed training data.

These works either do not discuss the capabilities of using shards [18], [19], or they assume that using shards results in an accuracy penalty [17], which has not to be the case as we present in this paper. Further, they are focusing on I/O during training, but they did not discuss the advantages of a convenient and efficient stage-in process. Their used storage backends during training were limited to a shared PFS or node-

local SSDs. In this paper, we show the importance of the stage-in process in the overall workflow and also present valuable insights on how shards impact the DL accuracy. Our evaluation is accompanied by a wide range of available storage backends: a PFS, local file systems operating on memory and SSDs, and a burst buffer file system running on memory and SSDs. This allows us to present an in-depth evaluation of various performance bottlenecks when using different DL workloads.

Other work focused on enhancing the I/O capabilities of TensorFlow and PyTorch by adding solutions for accessing samples stored on other compute nodes of the job [21], [22]. This allows for additional features like enhanced shuffling capabilities [21] or more efficient file access [22], but it requires customized tools and frameworks and does not improve the stage-in process. Yang et al. aimed to overcome I/O bottlenecks during training by building training batches primarily out of node-cached elements of previous steps [35]. DIESEL [20] used an outsourced database infrastructure to beat the performance of using the PFS for training at the cost of maintaining a more complex setup.

c) Model accuracy: Our setup follows common workflows for empirically evaluating the accuracy impact of shards using the CIFAR [36] data sets. Google recently investigated how CIFAR training is influenced by several parameters representing the quality of the data set [37]. Meng et al. [25] investigated the effects of local shuffling, where each worker randomly selects data from its local data set, and global shuffling, where workers communicate which data they select, to prevent different workers from using the same samples within a single step. However, prior work did not show the influence of using shards or biased shards.

IV. Ad hoc file systems in Deep Learning

Ad hoc file systems or *burst buffer file systems* are created for a specific use case and within a particular context. They combine node-local storage, e.g., many SSD, into a single global namespace, accessible at a user-defined path. For instance, they can be run within the context of an HPC job and therefore for a temporary lifespan, or they can exist for extended time periods in *campaigns* where multiple applications reuse the data instead of repeatedly accessing the HPC system's PFS. While accessing data within the burst buffer file system, an application is no longer dependent on bandwidth variations of the PFS and, at the same time, cannot interfere with other HPC applications on the storage level. Use cases for such applications include, for example, big data workloads, bulk-synchronous applications, checkpoint/restart, producer-consumer workloads, and DL workloads [38]. Their underlying I/O access patterns can be challenging for PFSs to handle efficiently and include large numbers of metadata operations, non-contiguous and random access patterns, or small I/O requests.

As a result, burst buffer file systems can considerably reduce the overall PFS load and I/O interference with other HPC applications [38]. Moreover, since burst buffer file systems, such as *GekkoFS* [23], *BeeGFS's BeeOND* [39], BurstFS [40], or

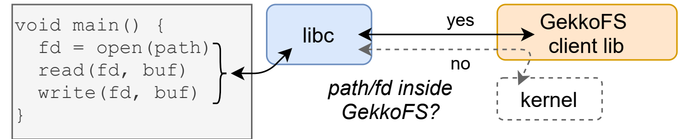


Fig. 2: GekkoFS's client library interception mechanism.

UnifyFS[4], scale linearly with the number of used nodes, they can outperform the bandwidth of traditional PFSs depending on the size and hardware of their storage backend [38]. In this work, we exemplarily use GekkoFS to highlight the benefits of using a burst buffer file system for the DL use case. In the following paragraphs, we introduce some of GekkoFS's internals that are important for the remainder of this paper.

A. GekkoFS

a) Usage and architecture: GekkoFS's main goal is to focus on scalability and consistency, allowing it to scale to an arbitrary number of nodes while still providing the same consistency as traditional POSIX file systems when accessing data files [23]. GekkoFS does not use a *distributed locking manager* (DLM) as it typically results in complex inter-node locking communication that can significantly slow down the file system, especially during metadata communication [41]. As a result, file system operations where the number of affected files is unknown a priori, e.g., for the `ls -l` command, are following an eventual consistency model. Note that this does not affect DL applications that are reading a large number of files since such operations follow the strong consistency model.

GekkoFS runs entirely in user-space and does not require root privileges to launch the two file system components: the server and the client. Before an application can use the GekkoFS client, a file system server process (*GekkoFS daemon*) is started on each node. For all GekkoFS-related data, each daemon uses a storage device that is available at a user-accessible path, e.g., a node-local SSD. Overall, all daemons combine the storage capacities and bandwidths of all used storage devices to form a global shared namespace. When all daemons are running, an application communicates with the client by preloading the GekkoFS interposition client library via `LD_PRELOAD`. The client library intercepts all file system-related system calls and checks if it operates within GekkoFS (see Figure 2). The file system provides a new namespace and, therefore, input data needs to be copied into GekkoFS first – known as the *stage-in* process. Note, for DL applications, a stage-in is already necessary to copy training data to the training nodes as accessing the PFS directly is not a viable option. Further, applications do not need to be modified in any way because GekkoFS is supporting the standard POSIX file system interface.

Figure 3 presents GekkoFS's architecture and how it interacts with a DL application. In general, the GekkoFS ecosystem

[4]https://github.com/LLNL/UnifyFS

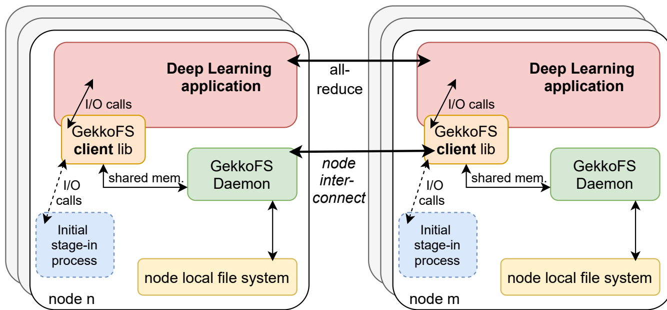


Fig. 3: GekkoFS's architecture with a DL application.

is decoupled, and, thus, daemons can handle each client request independently and do not communicate with other daemons. This minimizes file system complexity as well as network communication and allows GekkoFS to scale linearly in both metadata and data workloads, reaching over 45 million metadata operations per second on a 512 node cluster. An in-depth investigation of GekkoFS's architecture and performance evaluation can be found in [23], [42].

b) I/O distribution: By default, GekkoFS distributes all data and metadata in a pseudo-random fashion across all file system nodes, called *wide-striping*. For each file system operation, clients hash the file's path to determine the responsible daemon, which then independently handles the request. To efficiently move data between file system nodes, it utilizes the Mercury RPC communication library [43] which offers a network abstraction layer to natively use common network fabrics in HPC environments. In the example configuration of Figure 3, the GekkoFS daemons run on the same nodes as the clients use a communication protocol via shared memory. As a result, bulk transfers between clients and daemons can be done via *remote direct memory access* (RDMA) or emulated RDMA and via *cross-memory attach* (CWA) for the remote case and for the local case, respectively.

While metadata requests, e.g., file creation, target a single GekkoFS daemon, I/O is further split into several chunks based on the configured chunk size (512 KiB by default). This avoids the uneven usage of the node-local SSDs and improves overall file system performance as the load of large files is distributed to all daemons. Finally, all GekkoFS I/O operations are synchronous and do not use caching mechanisms beyond the operating system's buffer caches.

c) Data stage-in: Generally, input data and output data are staged into the burst buffer file system from the parallel file system (PFS) before an application is run and vice versa once it is finished. The stage-in process can be started by a simple `cp` command on the command line or by parallel copy applications, such as `dcp` as part of the popular *mpifileutils* package [5]. However, such tools require the data to be available in raw form to distribute all files to the corresponding file system nodes. Yet, DL training data can involve millions of small files (>1 MiB), which can be challenging for a PFS to handle efficiently. Hence, to mitigate the metadata load on the PFS, training data are usually available in compressed form, e.g., *tarballs*, or in specialized formats, e.g., TensorFlow's `tfRecord` containing a sequence of binary records. Training data must therefore be initially copied from the PFS to the training nodes. Because of GekkoFS's distribution, the training data is spread across all nodes, regardless if a data set consists of many small files or fewer large `tfRecord` files, for example. Hence, the expected DL file format can be used without the need to modify the training data whatsoever.

[5]https://github.com/hpc/mpifileutils

Model	Trainable param.	GPU time ratio
ResNet 50 [44]	26M	high
ResNet 18 [44]	12M	medium
WideResNet 28-10 [45]	37M	high
AlexNet [46]	63M	low
trivial net	49K	very low

TABLE I: Models used for Deep Learning experiments.

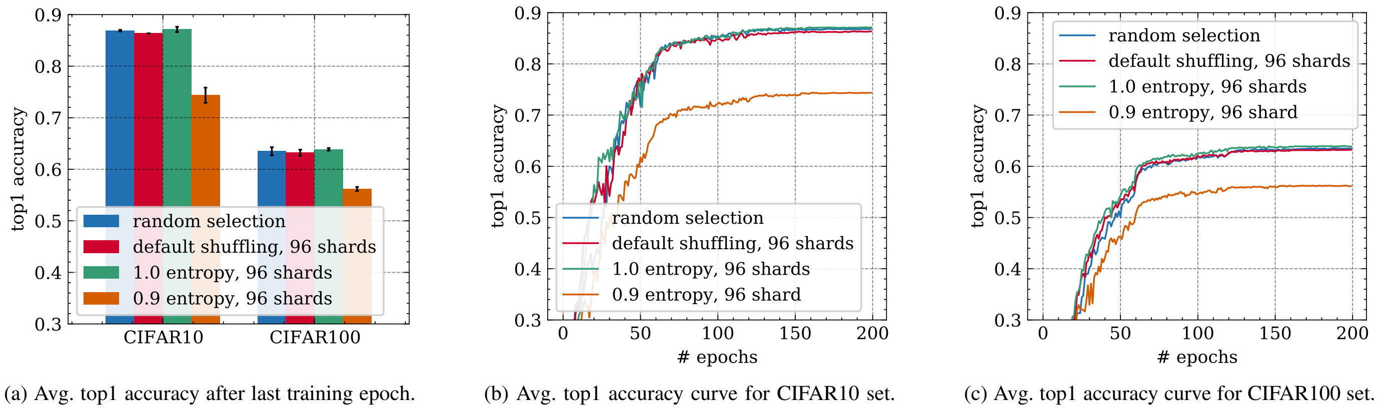
V. Evaluation

This section evaluates our proposed GekkoFS-driven distributed Deep Learning (DL) workflow. We divided our experiments into four sub-tasks: 1) investigate the effect of splitting data sets into shards for both biased and unbiased cases; 2) evaluate GekkoFS's raw I/O performance in a distributed namespace for memory and node-local storage via synthetic benchmarks; 3) compare the training step times of our solution with classical approaches on node-local storage when using real-world training setups, providing a detailed analysis of various configurations, overall scalability, and interference with the distributed training application; and 4) evaluate the required time for staging training sets into different node-local storage media and GekkoFS, and discuss stage-in bottlenecks and the overall metadata performance.

A. Experimental setup

We performed our experiments at the MOGON II supercomputer at the Johannes Gutenberg University Mainz in Germany[6]. MOGON II consists of 1876 nodes divided into 822 nodes using Intel Broadwell processors, and 1046 nodes using Xeon Gold 6130 Intel Skylake processors with a memory capacity of up to 512 GiB. The cluster uses a 100 Gbit/s Intel Omni-Path interconnect to establish a fat-tree between all compute nodes, providing a 7.5 PiB storage backend that is managed by several Lustre file systems. Experiments involving the PFS used a Lustre file system with 4 object storage servers (OSSs), 28 object storage targets (OSTs), and 1 metadata service (MDS) with a total of about 2.2 PiB of available storage. For node-local storage, each Skylake node provides an Intel SATA SSD DC S3700 Series with 400 GiB of available storage. The SSD is XFS formatted and can be used within a compute job as scratch space. We used the Skylake nodes in experiments that did not involve distributed learning. Moreover, MOGON II provides 30 separate nodes

[6]Mogon II https://hpc-en.uni-mainz.de/, May 2021



(a) Avg. top1 accuracy after last training epoch.

(b) Avg. top1 accuracy curve for CIFAR10 set.

(c) Avg. top1 accuracy curve for CIFAR100 set.

Fig. 4: Distributed training accuracy on the CIFAR10/100 datasets with a WideResNet 28-10 model (bs28 per worker).

using Intel Xeon CPU E5-2650 v4 processors and 6 Nvidia Geforce 1080 Ti GPUs each, which we used for the distributed learning experiments. They use Nvidia CUDA v11.2.2 and the NCCL library v2.8.4. They offer up to 128 GiB of memory, and they are connected via a 50 Gbit/s (4X FDR) Infiniband interconnect. A Micron 5100 Pro node-local SSD (XFS formatted) is available within a compute job.

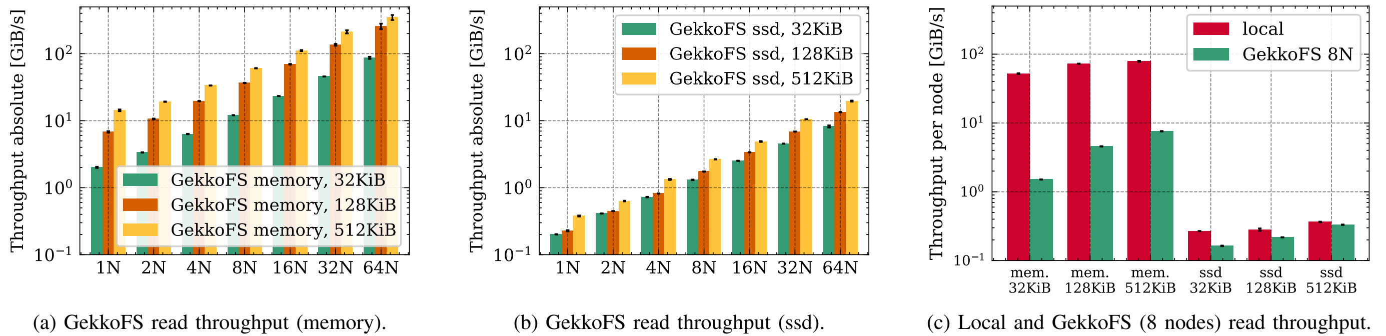
We used TensorFlow (v2.4.1) and Horovod (v0.21.3) to run real-world DL workloads. We focused on the training of classification problems that require large training data and hold a significant practical relevance. Overall, classification training is a good fit for distributed training as it typically tends to scales well using data parallel approaches. Table I lists the used models and their number of training parameters, i.e., the model size. These models are widely used in production systems and allow for a comprehensive evaluation of various kinds of bottlenecks. Note that more training parameters directly results in a larger gradient to optimize the model along with a higher data volume in the corresponding *all-reduce* operations. Nevertheless, more training parameters do not necessarily increase the runtime for each step. For each model, hyper parameters, such as the learning rate or optimizer settings, were configured for fastest convergence of accuracy. Finally, data augmentation was done as part of the models on the GPU, and all indicated batch sizes are per GPU worker. In general, our workflow is not limited to TensorFlow, and comparable results can also be achieved with PyTorch, for instance.

For each experiment, all SSD contents are removed, and all operating system caches are flushed before each experiment. For experiments that used the SSD, we filled more than 80% of memory with random data to limit the available buffer caches. All experiments that operated in memory used a *tmpfs* file system and disabled swap space.

B. Accuracy impact of staging-in data set shards

For measuring the effect of using shards, we ensured that each of the six training processes per node also used its dedicated data pipeline. Using 16 nodes with 6 GPUs each, this allows measuring the impact of up to 96 shards because each process can either use a copy of the data set or a dedicated shard. To show the impact of a biased stage-in using shards, we artificially constructed shards with a parameter representing the class entropy of the elements. We define entropy as the percentage of classes, a single worker can choose from, during the complete training process. Thus, an entropy lower than 1.0 introduces an artificial bias. Nevertheless, the complete set of workers still process all available classes and samples. In other words, an entropy of 1.0 is the ideal case whereas, for an entropy of 0.9, each shard only contains elements of 90% of the classes. Besides class labels, entropy could also be measured by using properties such as contrast or color variations. However, the class-based approach is comprehensive and straight-forward to implement. As described in Section II, distributing real data sets can easily be affected by biases because of many possible and not immediately obvious irregularities. Similar to previous work [37], we chose the popular CIFAR data sets together with a capable WideResNet 28-10 model. For maximizing the accuracy, we also adjusted the hyper parameters as in [45], and Horovod distributed SGD (Stochastic Gradient Descent) was using gradients with 32-bit-sized floating-point numbers. The CIFAR10 and CIFAR100 data sets are relatively small, containing 50k training (+10k testing) elements of 10 and 100 classes, respectively, with a size of $\sim$160MiB in total. For such small workloads, shards or dedicated stage-in solutions are not relevant for training in practice because a full copy of a few hundred megabytes is easy to maintain. However, the effect of unbiased shards is expected to be more pronounced for fewer training samples due to a fewer overall selection variety.

Figure 4a shows the *top1* accuracy after 200 epochs of distributed training with 96 workers using the WideResNet 28-10. Top1 represents the most conventional value for accuracy in DL, describing the probability of exactly predicting the expected result. Notice that the general accuracy is lower for CIFAR100 due to the more intricate learning process. In the random selection case, no shards are used and each worker can access the entire data set. When using 96 shards, that is, each learning process uses a fixed and randomly selected subset



(a) GekkoFS read throughput (memory).

(b) GekkoFS read throughput (ssd).

(c) Local and GekkoFS (8 nodes) read throughput.

Fig. 5: Read throughput for synthetic read-only benchmarks and various file sizes.

(containing 1/96 of the data set) for 200 epochs, we measured an accuracy drop of at most 1.5%. This behavior is similar for CIFAR10 and CIFAR100, although we observed rare cases for CIFAR100 where shards result in a higher accuracy. This shows that shards have an overall negligible effect on accuracy. For a balanced class entropy of 1.0, i.e., each shard contains the same amount of samples of each class, the results were indistinguishable from the non-shard accuracy case.

When the entropy was lowered to just 0.9, we observed a considerable drop in top1 accuracy (over 10%). With an entropy of 0.9, shards of CIFAR10 and CIFAR100 contain 9 and 90 different classes per shard, respectively. Figures 4b and 4c show the average top1 accuracy curves for up to 200 epochs for CIFAR10 and CIFAR100, revealing no anomalies other than accuracy and reaching a stable accuracy plateau. It is worth mentioning that an unbiased shard with more training samples, like in ImageNet or in other commercial workflows, would result in even smaller accuracy drops. Moreover, we chose to use a comparatively large batch size, which exaggerates the accuracy effects of shards, because smaller batch sizes allow more randomness in selection and, therefore, result in an even smaller accuracy impact of shards. On the other hand, the impact of entropy does not depend on overall data set size or batch sizes. In summary, even by exaggerating the accuracy effects when shards were used, we only observed a minimal accuracy impact. On the opposite, i.e., shards with biases, the impact on accuracy can become significant. But, performing an unbiased stage-in using shards can be challenging because traits that would cause biases are not necessarily eliminated by balancing the occurrence of classes. Thus, solutions providing global access instead of copies or shards should also focus a convenient stage-in to be usable in practice.

C. Synthetic read benchmarks

During training, DL I/O patterns typically involve many small and read-only accesses. We performed synthetic read benchmarks which emulate similar DL workloads by using the *mdtest*[7] microbenchmark on up to 64 Skylake nodes and 16 processes per node. We used the Skylake nodes as this part of the cluster offers more nodes to fully use the Omni-Path interconnect for GekkoFS. We further ensured that no GekkoFS client re-read its previously written data by shuffling the client processes between the write and read phases.

Figures 5a and 5b present the results for the memory and SSD storage backend, respectively, where each experiment performed full reads on three file size cases: 32 KiB, 128 KiB, and 512 KiB. The y-axis shows the throughput in GiB per second on a logarithmic scale. Each experiment was run at least five times. In general, GekkoFS achieved almost linear scaling in all cases, showing that it can efficiently handle DL workloads involving many small files. Note that for small files, the file system must spend comparatively more time during remote communication than reading from the underlying storage. In other words, small files are network latency-bound because the smaller the file I/O, the less time is spent for the read process per file, resulting in an overall decreased I/O throughput[8]. This effect is especially visible for the memory case, which reveals GekkoFS raw performance without being throttled by much slower SSD storage devices. In general, GekkoFS has shown to linearly scale up to 512 tested nodes in similar workloads [42].

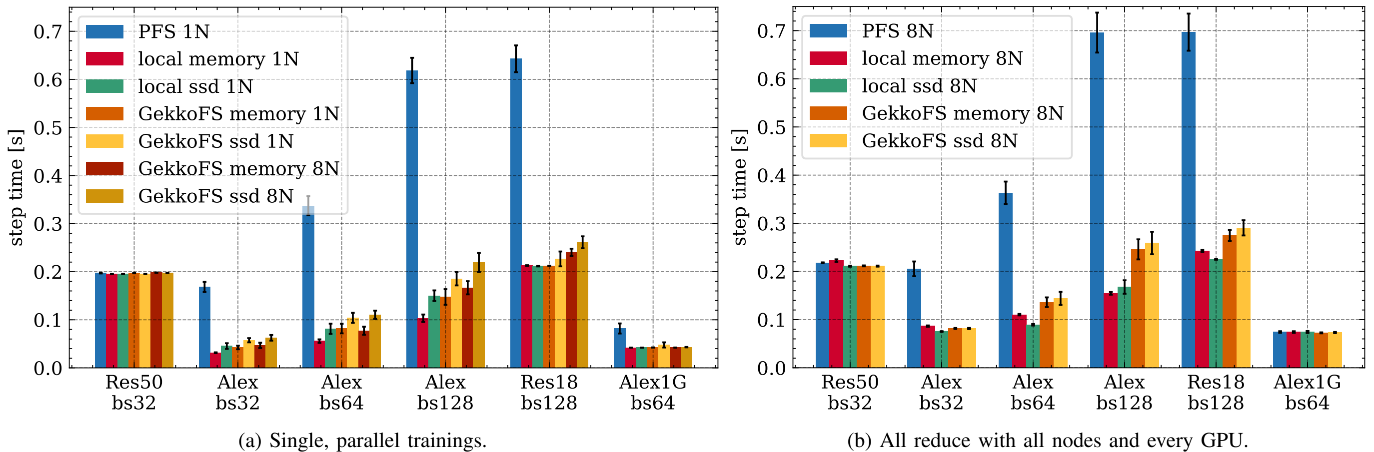
Figure 5c presents GekkoFS's SSD usage efficiency, that is, its achieved throughput on eight nodes compared with the combined throughput when each SSD would be used locally for the same workload. For the memory case and 128 KiB files, GekkoFS is about five times slower than if memory was used locally, showcasing the network cost of a shared GekkoFS environment where all files are distributed across all nodes. However, the throughput gap between the GekkoFS and the local cases gets significantly smaller when an SSD is used. For growing file sizes that are less network-latency sensitive, GekkoFS achieved almost local SSD performance.

D. Distributed TensorFlow training performance

To measure the effect of I/O on TensorFlow's training time, we used the popular ILSVRC2010 ImageNet data set, containing 1.26M images out of 1K classes with an average element size of 109KiB.

[7]https://github.com/hpc/ior

[8]The I/O throughput is connected to the *I/O operations per second* (IOPS). Thus, when I/O throughput decreases, the number of IOPS increases.



(a) Single, parallel trainings.

(b) All reduce with all nodes and every GPU.

Fig. 6: Comparison of avg. training step times for different setups, models and file systems using the ImageNet data set.

Horovod was used with the recommended (fastest) settings where all-reduce is performed in a *ring* mode and with 16 bit-sized floating-point numbers for model synchronization. We measured the step times starting from the fifth step since previous steps showed unstable results. Note, step times are not changing for multiple epochs, and we thus only measure a single epoch. Each data point consists of at least five measurements with their average for the corresponding epochs.

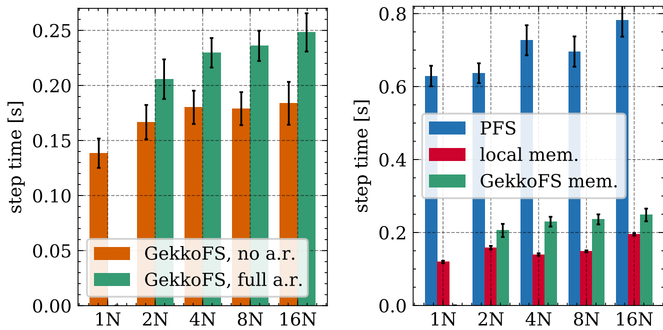
Within TensorFlow, the data pipeline was implemented using `tf.data`. During training, CPU side I/O was performed in parallel with the GPU side computation and the all-reduce operations. Each training process starts six I/O threads and can prefetch data for up to two following steps. The number of I/O threads and the amount of prefetched data were configured to achieve the fastest overall step times. We also ensured that workers do not access the same data twice. For fast all-reduce operations, Horovod was using NCCL on top of fast InfiniBand fabrics.

a) Step time analysis for the ImageNet data set: Figure 6a visualizes the effects of the I/O subsystem's latency on the training step times when no all-reduce synchronization phase occurs. Disabling the model synchronization phase allows us to investigate only the impact on the DL application when it is reading the training data from various storage systems. We considered workloads ranging from compute-bound to I/O-bound trainings. For example, ResNet50 is highly compute-bound, even for a small batch size of 32, and is not affected by the I/O backend. AlexNet, on the other hand, is I/O dependent, revealing a performance bottleneck for all batch sizes. As expected, reading from a local memory is the fastest option. When using the local SSD, step times increase for the read operations and are comparable with GekkoFS when operating in memory. For I/O-bound workloads, GekkoFS's performance decreased when using the SSD storage backend, but this highly depends on the used training setup (see x-axis). Using the PFS for reading training samples is, as expected, not a suitable option, although we are aware that a PFS is typically not used during training. We decided to include the results for the PFS to highlight the general I/O bottlenecks during training and which training setup is compute-bound. ResNet50 with batch size of 32 (`bs32`) showed a similar compute overhead compared with ResNet18 (`bs128`). In the latter, GekkoFS's experiments showed a minor I/O bottleneck, but its step times remained similar with AlexNet's step times for the same I/O load with a batch size of 128. This shows that I/O timings are generally independent of the compute load for GekkoFS.

For Figure 6b, we enabled Horovod's all-reduce model synchronization, representing the default case when Horovod is working in a fully distributed setting. Thus, step times increased compared to Figure 6a. Generally, step times increased more for the AlexNet than for ResNets due to its larger gradient size. Although I/O was performed in parallel during computation and all-reduce operations, we observed that step times increased more sharply for highly I/O-bound training setups, including when using the PFS. This indicates that for particular levels of I/O performance, prior overlapping parts of I/O and compute can no longer be pipelined efficiently. Overall, enabling model synchronization between workers highlights the compute-bound and I/O-bound training setups by exaggerating the step time for both types. This also reduced the step time difference between the SSD and memory cases.

Therefore, for GekkoFS, the step times for SSD and memory are similar for AlexNet (`bs128`), but the file system achieves similar performance when AlexNet (`bs32`) used local data. Note that GekkoFS does not interfere with the model synchronization (shown later in this section). For verification, both Figures 6a and 6b also include the *Alex1G* training setup which only used one GPU per node, and it is therefore highly compute-bound. We observed no anomalies and measured constant step times across all storage systems, excluding PFS.

b) Scalability of Horovod with GekkoFS: We evaluated the scalability by using up to 16 GPU nodes. Figure 7a shows the results for GekkoFS using memory for the the I/O-bound AlexNet (`bs128`) training. Similar to the above-discussed



(a) All-reduce enabled and disabled compared for GekkoFS using memory.

(b) All-reduce enabled comparison for Gekkofs, local mem. and PFS.

Fig. 7: Step times for distributed AlexNet (bs128) training.

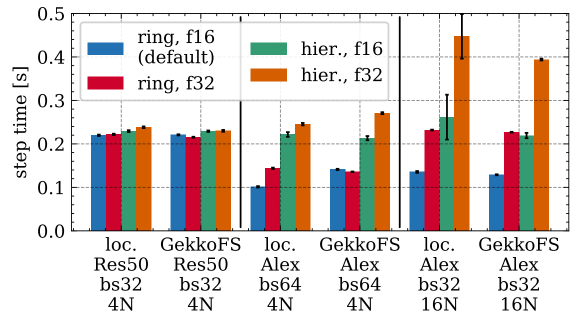


Fig. 9: Step times for various configurations of the all-reduce operation. Node-local storage and GekkoFS use memory.

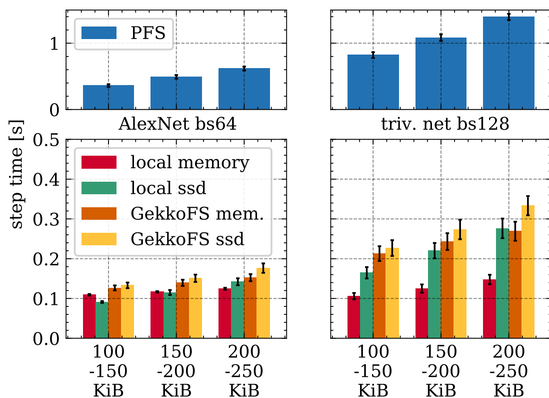
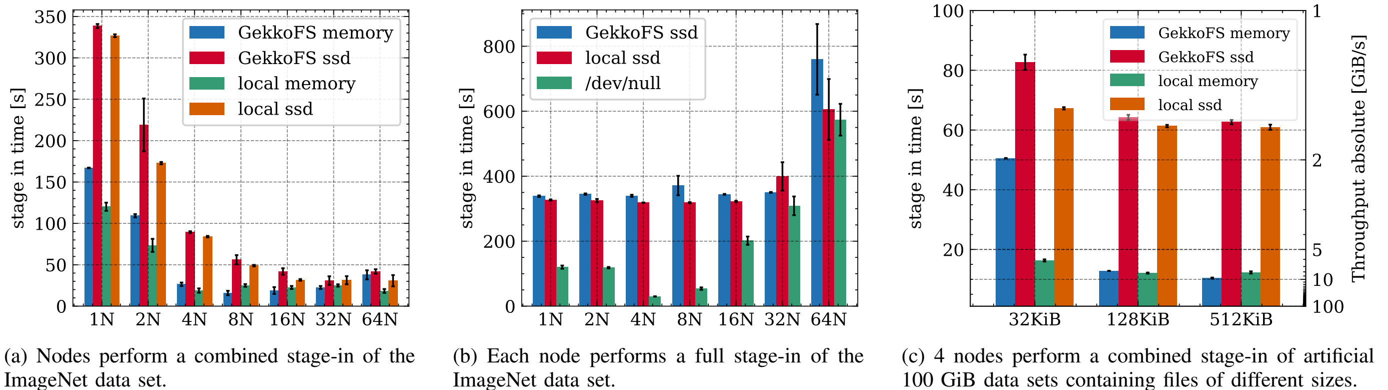


Fig. 8: Training step results after sorting the elements of ImageNet data set to bins of different file size.

results, we disabled model synchronization, i.e., all-reduce operations, (which does not disable parallel processing) to investigate only the step times when training data is read from the storage system (see `GekkoFS, no a.r.`). For this case, the step times converge to about 0.18 seconds. GekkoFS can perform better on fewer nodes as some reads can be served through fast shared memory. Hence, the fewer nodes are used, the more accesses can be served locally (100% of reads are served locally for the `1N` case). When model synchronization was enabled (see `GKFS, full a.r.`), the step times increased for all cases. Figure 7b includes the PFS and local in-memory cases with enabled all-reduce operations, placing GekkoFS's step times close to the local in-memory step times. Further, this shows the scalability of Horovod's model synchronization for the three storage solutions. Interestingly, operating locally in-memory did not show constant step time performance, although a ring all-reduce setting was used. We, therefore, conclude that the I/O performance of GekkoFS converges to around 0.25 seconds per step and remains constant for an increasing number of nodes. Also, the PFS results show increasing step times, indicating that it cannot reliably serve an increasing amount of small accesses.

c) Impact of various access sizes: We present the impact of various access sizes by putting elements of the ImageNet data set into multiple bins. Figure 8 visualizes the step times for 4 nodes and 2 training models: AlexNet (`bs64`) and a trivial net requiring nearly no GPU computation (`bs128`). Thus, the latter case was not compute-bound for any file system. The number of files per step was constant while the data throughput increased for larger element sizes. This is most noticeable for the SSD case which was impacted the most by the higher throughput requirements. Overall, the results show expected step times that are similar to the findings of Section V-C with the PFS performing the worst. Therefore, for smaller access sizes, the I/O throughput decreases while IOPS, i.e., read operations per second, increase, resulting in faster step times. Disabling the model synchronization resulted in similar findings. Finally, we noticed the local in-memory case performing slightly worse in some compute-bound cases than the local SSD. We will further investigate this behavior in future work as we focused mainly on I/O-bound cases.

d) GekkoFS interference during model synchronization: Figure 9 presents the distributed training performance of various models and node counts for different model synchronization settings that performs all-reduce operations. Both node local storage (loc.) and GekkoFS used an in-memory storage. We evaluated the performance by enabling more fine-grained gradient parameters (16→32-bit floating-point numbers for all-reduce) and by replacing the model synchronization from *ring* to *hierarchical*. In general, the runtime for a ring setup does not grow significantly with more nodes compared with a hierarchical setup, and, further, f32 all-reduce gradients result in longer synchronization times. If there are differences between local and GekkoFS accesses that are dependant on the model synchronization type, we can conclude that GekkoFS's inter-node communication may interfere with the distributed training itself. Nevertheless, this was not the case, and we observed similar step times for local and GekkoFS. We also ran similar experiments with data augmentation on the CPU instead of GPU, where we investigated the effect of a significantly higher CPU load. Like the above-shown results, we measured similar step times for both local accesses and GekkoFS.



(a) Nodes perform a combined stage-in of the ImageNet data set.

(b) Each node performs a full stage-in of the ImageNet data set.

(c) 4 nodes perform a combined stage-in of artificial 100 GiB data sets containing files of different sizes.

Fig. 10: Comparison of stage-in duration's for data sets from PFS to compute nodes.

Model	GekkoFS(4N)	local(4N)	GekkoFS(16N)	local(16N)
Alex (`bs64`)	96ms	95ms	101ms	98ms
Res50 (`bs32`)	16ms	13ms	21ms	21ms

TABLE II: Duration of only the all reduce operation for all GPU's per step (standard derivation around ± 3ms).

We verified that GekkoFS does not interfere with Horovod's model synchronization by using NVIDIA NCCL profiler tools. Due to parallelizing I/O during computation and all-reduce operations, a possible GekkoFS interference could cause higher all-reduce timings. We measured both compute-bound and I/O bound models by using ResNet and AlexNet, respectively. Table II lists the average for each all-reduce operation during the model synchronization. For both 4 nodes and 16 nodes setups, timings of local accesses and GekkoFS accesses are similar, showing no hint for interference.

E. Data set stage-in runtimes

The stage-in experiments were run on MOGON II's Skylake nodes using a custom parallel untar tool. It starts several *untar* threads per node, each reading tar files from the PFS, which are then unpacked into an in-memory queue. A number of *stage-in* threads move the queue elements to the corresponding file system by using standard POSIX write operations. To guarantee high throughput, both thread types are load-balanced dynamically. We used both ImageNet and synthetic data for the stage-in process and compared the runtimes to stage the data into four types of storage subsystems: local memory, local SSD, GekkoFS using memory, and GekkoFS using SSD. The ImageNet data set originally has 1,000 tar files, each containing the elements of a single class. Using the classical approach of simply distributing these tars into shards would result in an extreme training bias. Thus, for non GekkoFS stage-in all training files would need to be shuffled *after* the data set was staged-in. We did not shuffle the data and focused on the throughput aspects of the stage-in process.

Figure 10a shows the runtimes of our stage-in tool for the ImageNet data set. For a single node, the full data set must be copied and unpacked, whereas, for n nodes, each node processes $1/n$ of the data set. Because GekkoFS offers a single global namespace, all nodes can access the complete data set after the stage-in without training bias. This is in contrast to the local file systems which only receive a part of the data set, i.e., a shard. Until 4 nodes, GekkoFS's stage-in throughput is about 10% and 25% slower than the local SSD and memory, respectively. After eight nodes, parallel stage-in is significantly reduced and limited by the PFS.

Figure 10b highlights the PFS limitations by weak-scaling the amount of staged-in data with the number of nodes. As a result, each local node has access to the complete data set. Although this is not necessary for GekkoFS due to its global namespace, we included its results, showing that GekkoFS does not slow down when n data set copies are moved into the file system. Interestingly, for the `/dev/null` case[9] representing the upper bound for Lustre, i.e., data is read from Lustre but not written to local storage, stage-in times decreased at four nodes, indicating file system cache optimizations. Nevertheless, the stage-in runtimes drastically increased after 16 nodes with varying I/O performance at 64 in all cases due to the high load on Lustre. Note that a larger Lustre environments may show a different behavior since Lustre has shown to scale linearly for sequential workloads with more OSSs and OSTs [47]. Figure 10c presents the results for 4 nodes, cooperatively staging-in a synthetic data set of 2,000 tar files containing random data of various file sizes. While GekkoFS was slower for small files (32 KiB) that are network latency-sensitive than the local cases, GekkoFS achieved almost local performance for larger files (128 KiB).

Finally, we compared the runtime to perform `ls -laR` on the whole staged-in ImageNet data set. We include this experiment because it is important for users to know the directory structures and file names when selecting training samples out of the staged-in data. For 64 nodes, XFS on local SSD was the slowest, requiring 18.6 seconds. Within the local memory, listing all files took 5.6 seconds with its metadata

[9]Figure 10b could not run in memory because some nodes do not have sufficient memory space to accommodate the complete data set.

performance depending on unsorted VFS buffers. GekkoFS using 64 nodes was the fastest, requiring only 2.36 seconds. This is because GekkoFS is broadcasting the corresponding `readdir()` operation to all GekkoFS daemons in parallel. Each daemon can then independently return all metadata due to its sorted RocksDB key-value store.

In practice, the ratio between the runtime for stage-in and training depends on the training scenario. Faster training approaches usually use a higher degree of parallelism, and therefore, the distributed stage-in process is even more important. Nevertheless, next to reducing the runtime of DL application, efficient stage-in is also essential to reduce the PFS load. It is worth noting that our used ImageNet data already revealed the limits of our used PFS, but it is comparatively small compared to other data sets, e.g., from the earth or climate sciences which can reach terabytes of data [2], [16]. Therefore, the adoption of our work flow for larger workflows has the broadest impact.

VI. Conclusion and Outlook

The computational requirements of DL applications will likely increase in the future, accompanied by growing demands on the shared I/O storage system of HPC systems. Already today, data sets are copied to node-local storage as storage systems in HPC cannot efficiently handle many small random accesses typically found in DL applications. In this paper, we have first investigated the common misconception under which circumstances splitting data sets into shards negatively affect training accuracy. We have shown that this is not the case for unbiased shards but have also explained its practical challenges. Instead, we have proposed a transparent DL workflow using the GekkoFS ad hoc file system that does not rely on data packing to ensure unbiased data samples due to its shared global namespace. It requires minimal data loading from PFS, and our evaluation has demonstrated that GekkoFS scales linearly for common DL workloads achieving similar performances than if local data set copies were used.

In the future, we plan to extend our workflow in several directions: We will further optimize the network usage of the ad hoc file system, and we will evaluate the effect of NVMe SSDs on the overall workflow. Moreover, we will consider the impact of *storage campaigns* to our workflow where the data is available for extended time periods for several jobs that are working the same data set.

Acknowledgement

This research was conducted using the supercomputer Mogon II and services offered by Johannes Gutenberg University Mainz. The authors gratefully acknowledge the computing time granted on Mogon II.

This work has been supported by the project “Big Data in Atmospheric Physics (BINARY)”, funded by the Carl Zeiss Foundation (grant P2018-02-003), and by the “Adaptive multi-tier intelligent data manager for Exascale (ADMIRE)” project, funded by the European Union’s Horizon 2020 JTI-EuroHPC Research and Innovation Programme (grant 956748).

This work has also been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grants PID2019-107255GB, and the Generalitat de Catalunya under contract 2014–SGR–1051.

References

[1] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth, “Analysis of the ECMWF Storage Landscape,” in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 16-19*, 2015, pp. 15–27.

[2] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. H. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, “Exascale Deep Learning for Climate Analytics,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), Dallas, TX, USA, November 11-16*, 2018, pp. 51:1–51:12.

[3] S. Rasp, M. Pritchard, and P. Gentine, “Deep learning to represent subgrid processes in climate models,” *Proceedings of the National Academy of Sciences*, vol. 115, no. 39, pp. 9684–9689, 2018.

[4] M. Reichstein, G. Camps-Valls, B. Stevens, M. Jung, J. Denzler, N. Carvalhais, and Prabhat, “Deep learning and process understanding for data-driven Earth system science,” *Nature*, vol. 566, no. 7743, pp. 195–204, 2019.

[5] P. D. Dueben and P. Bauer, “Challenges and design choices for global weather and climate models based on machine learning,” *Geoscientific Model Development*, vol. 11, no. 10, pp. 3999–4009, 2018.

[6] NVIDIA DGX Systems. https://www.nvidia.com/en-us/data-center/dgx-systems/.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” in *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), 20-25 June, Miami, Florida, USA*, 2009, pp. 248–255.

[8] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour,” *CoRR*, vol. abs/1706.02677, 2017.

[9] Y. You, Z. Zhang, C. Hsieh, J. Demmel, and K. Keutzer, “ImageNet Training in Minutes,” in *Proceedings of the 47th International Conference on Parallel Processing (ICPP), Eugene, OR, USA, August 13-16*, 2018, pp. 1:1–1:10.

[10] H. Mikami, H. Suganuma, P. U.-Chupala, Y. Tanaka, and Y. Kageyama, “ImageNet/ResNet-50 Training in 224 Seconds,” *CoRR*, vol. abs/1811.05233, 2018. [Online]. Available: http://arxiv.org/abs/1811.05233

[11] P. Sun, W. Feng, R. Han, S. Yan, and Y. Wen, “Optimizing Network Performance for Distributed DNN Training on GPU Clusters: ImageNet/AlexNet Training in 1.5 Minutes,” *CoRR*, vol. abs/1902.06855, 2019.

[12] S. W. D. Chien, A. Podobas, I. B. Peng, and S. Markidis, “tf-Darshan: Understanding Fine-grained I/O Performance in Machine Learning Workloads,” in *IEEE International Conference on Cluster Computing (CLUSTER), Kobe, Japan, September 14-17*, 2020, pp. 359–370.

[13] F. B. Schmuck and R. L. Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters,” in *Proceedings of the Conference on File and Storage Technologies (FAST), January 28-30, Monterey, California, USA*, 2002, pp. 231–244.

[14] P. Braam, “The Lustre Storage Architecture,” *CoRR*, vol. abs/1903.01955, 2005.

[15] S. A. Jacobs, J. Gaffney, T. Benson, P. B. Robinson, L. Peterson, B. K. Spears, B. V. Essen, D. Hysom, J. Yeom, T. Moon, R. Anirudh, J. J. Thiagarajan, S. Liu, and P. Bremer, “Parallelizing Training of Deep Generative Models on Massive Scientific Datasets,” in *IEEE International Conference on Cluster Computing (CLUSTER), Albuquerque, NM, USA, September 23-26*, 2019, pp. 1–10.

[16] Y. Oyama, N. Maruyama, N. Dryden, E. McCarthy, P. Harrington, J. Balewski, S. Matsuoka, P. Nugent, and B. V. Essen, “The Case for Strong Scaling in Deep Learning: Training Large 3D CNNs With Hybrid Parallelism,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1641–1652, 2021.

[17] Z. Zhang, L. Huang, J. G. Pauloski, and I. T. Foster, “Efficient I/O for Neural Network Training with Compressed Data,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, May 18-22*, 2020, pp. 409–418.

[18] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient User-Level Storage Disaggregation for Deep Learning," in *2019 IEEE International Conference on Cluster Computing (CLUSTER), Albuquerque, NM, USA, September 23-26*, 2019, pp. 1–12.

[19] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning," in *Proceedings of the 48th International Conference on Parallel Processing*. Association for Computing Machinery, 2019.

[20] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, and Q. Luo, "DIESEL: A Dataset-Based Distributed Storage and Caching System for Large-Scale Deep Learning Training," in *49th International Conference on Parallel Processing (ICPP), Edmonton, AB, Canada, August 17-20*, 2020, pp. 20:1–20:11.

[21] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems," in *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Milwaukee, WI, USA, September 25-28*, 2018, pp. 145–156.

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

[23] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - A Temporary Distributed File System for HPC Applications," in *IEEE International Conference on Cluster Computing (CLUSTER), Belfast, UK, September 10-13, 2018*.

[24] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *19th International Conference on Computational Statistics (COMPSTAT), Paris, France, August 22-27 - Keynote, Invited and Contributed Papers*, 2010, pp. 177–186.

[25] Q. Meng, W. Chen, Y. Wang, Z. Ma, and T. Liu, "Convergence Analysis of Distributed Stochastic Gradient Descent with Shuffling," *Neurocomputing*, vol. 337, pp. 46–57, 2019.

[26] Cloud TPU. https://cloud.google.com/tpu.

[27] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 65:1–65:43, 2019.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA, USA, November 2-4*, 2016, pp. 265–283.

[29] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the ACM International Conference on Multimedia (MM), Orlando, FL, USA, November 03 - 07*, 2014, pp. 675–678.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS), Vancouver, BC, Canada, December 8-14*, 2019, pp. 8024–8035.

[31] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *CoRR*, vol. abs/1802.05799, 2018.

[32] NVIDIA NCCL. https://developer.nvidia.com/nccl.

[33] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari, "Revisiting I/O behavior in large-scale storage systems: the expected and the unexpected," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, Colorado, USA, November 17-19*, 2019, pp. 65:1–65:13.

[34] S. W. D. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. Laure, "Characterizing Deep-Learning I/O Workloads in TensorFlow," in *3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW-DISCS@SC, Dallas, TX, USA, November 12*, 2018, pp. 54–63.

[35] C. Yang and G. Cong, "Accelerating Data Loading in Deep Neural Network Training," in *26th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), Hyderabad, India, December 17-20*, 2019, pp. 235–245.

[36] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[37] Y. Bahri, E. Dyer, J. Kaplan, J. Lee, and U. Sharma, "Explaining Neural Scaling Laws," *CoRR*, vol. abs/2102.06701, 2021.

[38] A. Brinkmann, K. Mohror, W. Yu, P. H. Carns, T. Cortes, S. Klasky, A. Miranda, F. Pfreundt, R. B. Ross, and M.-A. Vef, "Ad Hoc File Systems for High-Performance Computing," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 4–26, 2020.

[39] F. Herold, S. Breuner, and J. Heichler, "An Introduction to BeeGFS," https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014, accessed on May, 14, 2021.

[40] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An Ephemeral Burst-Buffer File System for Scientific Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, J. West and C. M. Pancake, Eds. IEEE Computer Society, 2016, pp. 807–818.

[41] M.-A. Vef, V. Tarasov, D. Hildebrand, and A. Brinkmann, "Challenges and Solutions for Tracing Storage Systems: A Case Study with Spectrum Scale," *ACM Transactions on Storage*, vol. 14, no. 2, pp. 18:1–18:24, 2018.

[42] M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - A Temporary Burst Buffer File System for HPC Applications," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 72–91, 2020.

[43] J. Soumagne, D. Kimpe, J. A. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. B. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *IEEE International Conference on Cluster Computing (CLUSTER), Indianapolis, IN, USA, September 23-27*, 2013, pp. 1–8.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[45] S. Zagoruyko and N. Komodakis, "Wide Residual Networks," in *Proceedings of the British Machine Vision Conference (BMVC), York, UK, September 19-22*, 2016.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1106–1114.

[47] S. Oral, D. A. Dillow, D. Fuller, J. Hill, D. Leverman, S. S. Vazhkudai, F. Wang, Y. Kim, J. Rogers, J. Simmons *et al.*, "OLCF's 1 TB/s, Next-Generation Lustre File System," in *Proceedings of Cray User Group Conference (CUG)*, 2013, pp. 1–12.