2021 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-7281-9666-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/Cluster48925.2021.00097

MONARCH: Hierarchical Storage Management for Deep Learning Frameworks

Marco Dantas, Diogo Leitão, Cláudia Correia, Ricardo Macedo, Weijia Xu†, João Paulo
INESC TEC & University of Minho †*Texas Advanced Computing Center*

***Abstract*—Due to convenience and usability, many deep learning (DL) jobs resort to the available shared parallel file system (PFS) for storing and accessing training data when running in HPC environments. Under such a scenario, however, where multiple I/O-intensive applications operate concurrently, the PFS can quickly get saturated with simultaneous storage requests and become a critical performance bottleneck, leading to throughput variability and performance loss.**

We present MONARCH, a framework-agnostic middleware for hierarchical storage management. This solution leverages the existing storage tiers present at modern supercomputers (*e.g.,* compute node's local storage, PFS) to improve DL training performance and alleviate the current I/O pressure of the shared PFS.

We validate the applicability of our approach by developing and integrating an early prototype with the TensorFlow DL framework. Results show that MONARCH can reduce I/O operations submitted to the shared PFS by up to 45%, decreasing training time by 24% and 12%, for I/O-intensive models, namely LeNet and AlexNet.

***Index Terms*—High-Performance Computing, I/O Optimization, Storage Tiering, Deep Learning**

I. INTRODUCTION

High-performance computing (HPC) infrastructures are increasingly popular to support computational demanding deep learning (DL) workloads. Such workloads are typically backed by large-scale datasets that range from few GiB to several TiB in size and are made of multiple small-sized files. For example, Open Images [1] has around 9 million images and ImageNet-22k [2] has approximately 14 million images. During training time, data samples are repeatedly read from storage in random order to achieve accurate and unbiased models. However, this random access pattern, combined with the long-lived and recurrent access to millions of small-sized files, can easily overload HPC's shared parallel file system (PFS) (*e.g.,* Lustre [3], BeeGFS [4], GPFS [5]) with the exceptional amount of both data and metadata requests, leading to high throughput variability and performance loss [6]–[10].

DL frameworks (*e.g.,* TensorFlow [11], PyTorch [12], MXNet [13]) are aware of this performance bottleneck and follow various approaches to suppress them. First, optimized data formats, such as TensorFlow's TFRecords [14], MXNet's RecordIO [15], and HDF5 [16] pack several small-sized files into a single, larger one, reducing the number of files being accessed and, consequently, the number of metadata operations required during the training phase. Second, DL frameworks provide an optimized data loading pipeline that enables I/O optimizations, such as caching, prefetching, and parallel I/O.

Nevertheless, many datasets tend to go beyond the memory capacity of HPC compute nodes, and DL frameworks fail to properly utilize the nodes' local storage mediums. Thus, most of the employed optimizations become subpar [17]–[19]. Indeed, in modern supercomputers, such as Frontera [20] and ABCI [21], compute nodes are equipped with fast local storage mediums (*e.g.,* SSD, NVMe) that can be used to totally or partially cache DL datasets, consequently reducing the pressure on the PFS and speeding up DL training.

However, the decision to resort to these local devices must, in most cases, be made manually by users when submitting DL jobs. To prevent this manual effort, previous work allows users to cache full datasets at the compute nodes' local disks [18], [19]. The main drawback is that these optimizations are only applicable under scenarios where the entire dataset can be cached at local devices, while datasets that do not meet this condition must still be entirely stored at and accessed from the PFS.

Therefore, it is imperative to identify novel solutions for taking advantage of compute nodes' local storage capabilities to reduce the pressure of DL jobs at the PFS and, consequently, to provide better and more sustained training performance. Furthermore, these solutions should: *i)* support datasets with variable sizes that may or may not be cached entirely on the compute node's (*i.e.,* local storage and memory); *ii)* be transparent to users while not requiring any manual intervention or changing how DL workloads are deployed; and *iii)* be portable and applicable across different DL frameworks.

To overcome the aforementioned challenges, we propose MONARCH, a framework-agnostic middleware for hierarchical HPC storage management, while providing the following contributions:

- An **experimental study** demonstrating the performance impact of running DL jobs under the Lustre PFS and the compute nodes' local storage. Results show that local storage can significantly accelerate training performance and reduce throughput and I/O variability.
- **MONARCH**, a novel storage middleware that mediates I/O requests between DL frameworks and HPC storage resources. It provides a new tiering mechanism that leverages from persistent storage resources (*e.g.,* local disks) available at compute nodes to fully or partially cache DL datasets and to alleviate the I/O pressure at the PFS. Also, MONARCH follows a decoupled design and provides a simple interface that enables straightforward integration with existing DL frameworks.

978-1-7281-9666-4/21/$31.00 ©2021 IEEE

DOI 10.1109/Cluster48925.2021.00097
Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on July 01,2024 at 19:55:26 UTC from IEEE Xplore. Restrictions apply.

- An **early prototype of MONARCH** and its integration with TensorFlow, which only required adding 6 lines of code. MONARCH is available at **https://github.com/dsrhaslab/monarch**.
- An **experimental evaluation** of our prototype with different models and dataset sizes. Results show that MONARCH can reduce I/O operations submitted to the shared file system by up to 45%, decreasing DL training time by 24% and 12% for I/O-intensive models, namely LeNet and AlexNet.

II. MOTIVATION

We conducted a preliminary experimental evaluation to understand and demonstrate the performance impact of running DL jobs from datasets stored in different storage mediums. Namely, we considered the following testing scenarios.

- **Vanilla-lustre**. Dataset samples are served solely from the remote storage backend *i.e.,* Lustre file system.
- **Vanilla-local**. Dataset samples are served solely from the compute node's local storage through the XFS file system.
- **Vanilla-caching**. During the first training epoch, dataset samples are served from Lustre and written to local storage, while for the remainder epochs, samples are read from the local disk.

Experimental setup. Experiments were conducted on a compute node of the Frontera supercomputer. The compute node is equipped with two 16-core Intel Xeon E5-2620 processors, four Nvidia Quadro RTX 5000, 128 GiB of RAM, and a single 240 GiB SSD with an accessible 119 GiB partition. Software-wise, it uses CentOS 7.8 with the Linux kernel v3.10. We limited memory usage to 68 GiB to simulate a scenario where the dataset could not fit entirely in memory.

Dataset, models, and DL framework. We used a truncated version of the ImageNet-1k dataset [22], that includes 900k images (100 GiB), enabling the dataset to fit entirely on the local device. To speed up training process, the dataset was converted into TFRecords. Experiments included two I/O-bound models, namely LeNet [23] and AlexNet [24], and a compute-bound model, namely ResNet-50 [25]. Due to its popularity and support for the caching optimization discussed in the *vanilla-caching* setup [26], we used the TensorFlow DL framework (v.2.3.2) with I/O parallelism, prefetching and parallel preprocessing optimizations enabled. For all runs, TensorFlow used all four GPUs available in the compute node.

Methodology. We measured the elapsed time and resource usage (*i.e.,* CPU, GPU, memory) of all experiments throughout three training epochs. The results of each experiment concern the average and standard deviation of 7 runs.

A. Results Analysis

Training time. Figure 1 depicts the average training time for each epoch under all described scenarios. When compared to *vanilla-lustre*, the *vanilla-local* setup significantly reduces overall training time for LeNet and AlexNet models. Specifically, under LeNet, the combined execution time (three epochs) decreases from 1205 to 650 seconds, while for AlexNet it decreases from 1193 to 976 seconds, representing a 46% and 18% decrease, respectively. The same is not observed for ResNet-50, as it is a compute-bound model in this experimental setup and imposes less I/O demand [27].

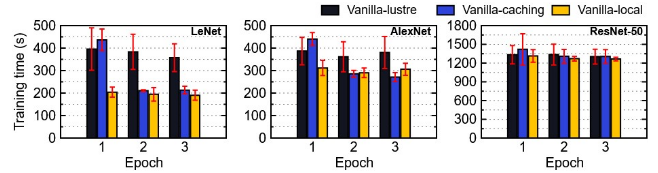

Fig. 1. Average training time and standard deviation for the *vanilla-lustre*, *vanilla-local*, and *vanilla-caching* setups when using LeNet, AlexNet, and ResNet-50 models. Results are detailed per training epoch.

Regardless of the DL model, we observed high performance variability under the *vanilla-lustre* setup, since Lustre is concurrently accessed by other jobs executing in the Frontera supercomputer. This motivates our claim that reducing the load on shared storage is key for having sustained and predictable performance for DL workloads.

With TensorFlow's caching mechanism (*i.e., vanilla-caching*), data samples are cached at the local SSD during the first training epoch, reducing training time for the LeNet and AlexNet models by 288 seconds and 135 seconds, representing a 24% and 11% decrease when compared to *vanilla-lustre*.

However, a slight increase in training time is visible for the first epoch when using the *vanilla-caching*, going from 396 seconds (*vanilla-lustre*) to 437 seconds. This is due to the extra data copying that must be done between Lustre and the local file system. On the other hand, a performance boost is noticeable for the second and third epochs, reaching similar performance to the *vanilla-local* setup. Namely, training time is decreased by up to 43% and 25% (total elapsed time of 424 and 557 seconds) for LeNet and AlexNet, respectively. For the same reason, performance variability is more noticeable during the first epoch when Lustre is being accessed.

Resource usage. Under *vanilla-lustre*, LeNet has an average CPU and GPU utilization of 30% and 22%, respectively, increasing to 57% and 39% when configured with the *vanilla-local* setup. For *vanilla-caching*, CPU usage increases to 37% and GPU usage to 28%. For the AlexNet model, the *vanilla-lustre* scenario shows 31% of CPU usage and 58% of GPU usage. For *vanilla-local*, these values rise to 42% and 72%, respectively. With *vanilla-caching*, CPU usage is 34% and GPU usage is 63%. These results highlight that, with better storage performance, compute nodes' CPU and GPU resources are used more efficiently by I/O-bound models. As ResNet-50 is a compute-bound model, for all setups, CPU and GPU usages remain at 10% and 90%, respectively.

Finally, memory usage remains approximately the same for all models and testing scenarios, averaging at 10 GiB.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on July 01,2024 at 19:55:26 UTC from IEEE Xplore. Restrictions apply.

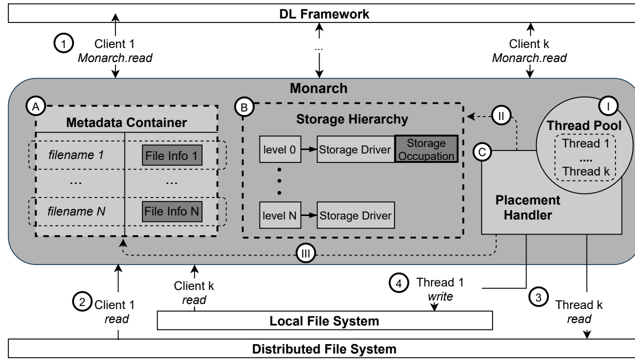

Fig. 2. MONARCH's simplified architectural design.

Summary. Results show that serving the dataset from local storage improves training time and performance predictability for I/O-bound models, while reducing data and metadata pressure on the Lustre file system. TensorFlow's caching mechanism transparently copies the dataset to local storage without manual input from users. However, the current implementation of this mechanism is only applicable when the full dataset fits on the local disk, which is not the case for many scenarios (*e.g.,* the original ImageNet-1k dataset has 140GiB).

III. MONARCH

MONARCH is a framework-agnostic storage tiering middleware that leverages existing storage mediums of HPC infrastructures to accelerate DL training performance. MONARCH sits between the DL framework and a hierarchy of storage backends, including local file systems mounted on the compute node's local storage and distributed storage such as the Lustre PFS. Its primary purpose is to move data samples from the shared and performance-expensive remote storage to the compute node's local device to reduce the DL training time as well as the I/O pressure imposed over the PFS.

To ensure transparency to users, and to not conflict with I/O-oriented optimizations already implemented at the DL framework (*e.g.,* data shuffling, caching and prefetching, I/O parallelism), MONARCH resides at the POSIX layer, thus not impacting the internal operation model of the targeted framework. It achieves this by exposing a simple API composed by a custom `read` operation that replaces the traditional POSIX `read` system call used by DL frameworks. Such an API enables straightforward integration with different DL frameworks, as it requires minimal code changes, maintaining MONARCH's design generic and portable.

A. Design

Figure 2 depicts MONARCH's architecture. It is organized in three main modules, namely the *storage hierarchy*, *placement handler*, and *metadata container*.

Storage hierarchy. The *storage hierarchy* module organizes and manages the storage tiers (or levels) that will be used to read and store data samples for DL training. Tiers are organized hierarchically, and the system designer defines their order. For instance, in this paper, storage tiers are organized by descending order in terms of performance, but could be organized through other criteria (*e.g.,* storage quota). Each tier is represented by a *storage driver*, which is an object that abstracts the I/O logic performed under a given storage backend (*i.e.,* local file system, PFS). In addition, this object contains a set of properties that allow governing the current state of that backend, including storage path (*i.e.,* file system mount point) and available storage quota (*storage ocuppation*). This abstraction adds the possibility to support different storage backends, promoting modularity and extensibility.

Except for the last one, all levels start without any files and are read-write backends. The last level is reserved for the PFS (*e.g.,* Lustre), which holds the full dataset and acts as a read-only data source.

Placement handler. The *placement handler* module holds the logic to select the storage tier where a given file should be placed. The data placement process occurs at runtime and follows the user-defined *placement policy* specified at the *storage hierarchy* module. Specifically, given a *storage hierarchy* of size N, the algorithm starts the data placement in descending order, writing the data samples first to level 0 until reaching its capacity, moving then to the remainder levels, until all levels that hold local *storage drivers* are filled ($[0, N-2]$). This module provides a dedicated *thread pool* to fetch and store data samples from the lowest storage tier (*e.g.,* PFS) to other levels, enabling DL frameworks' requests to be served in parallel. Contrary to related work [18], [19], our strategy allows supporting storage tiering when the dataset does not fit entirely on local storage. Moreover, due to the original workflow logic of the DL framework, no evictions are made at any level of the *storage hierarchy*. As the DL training data access pattern is random (*i.e.,* data shuffling), all files consider the same probability of being access within each epoch. Using a cache replacement policy would increase the operations between storage tiers, accentuating I/O trashing effects and the strain placed on the PFS. Our experiments demonstrate that this *placement policy* works well in the evaluated scenarios.

Regarding the timing when data placement is done, there are two main options: *i)* training files are read from the PFS and placed in the corresponding storage levels before executing the training phase; or *ii)* files are placed in the correct storage level during the first epoch of the training phase while being requested by the DL framework. We opted for the second approach to prevent any delay in the training execution time. With this approach, data samples are fetched and placed at the different tiers by following the order of I/O requests issued by the DL framework. Also, this strategy requires the same number of operations to the PFS backend as the first one, thus not adding additional I/O pressure on the shared file system.

In more detail, when a file is requested for the first time by the DL framework, the corresponding POSIX read request is intercepted by MONARCH, and the file content is read from the last tier level. Then, the content returned by the request

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on July 01,2024 at 19:55:26 UTC from IEEE Xplore. Restrictions apply.

is forwarded to the DL framework and, at the same time, in background, it is written to the appropriate storage level following the data placement algorithm previously discussed.

When using large file formats (*e.g.,* TFRecords), the DL framework may issue read requests for obtaining only a minor portion of the file's content. In this scenario, MONARCH replies to the DL framework with the requested file content, but, in background, it reads the full content of the file from the last tier level and writes it to the corresponding storage level. This is a meaningful optimization that allows subsequent requests to the same file to be served from a top-level tier instead of the PFS.

Metadata container. The *metadata container* module manages a virtual namespace of the overall *storage hierarchy*, storing general information about each file (given by *file info*), including its size, name, and current location (*i.e.,* storage tier). Similar to the main workflow of HPC jobs, the metadata container follows an ephemeral storage model. Specifically, the namespace, alongside other structures of the remainder modules, is populated at the beginning of the DL training phase, continuously updated during runtime, and removed at the end of the job's execution.

B. MONARCH*'s Operation Flow*

We now describe the main operation flow of a read request made by a DL framework to MONARCH, as illustrated in Figure 2. For illustration purposes, let us consider a dataset composed of large-sized files (*e.g.,* TFRecords). Before execution, the system designer specifies the main MONARCH configuration, defining the storage tiers that should be considered. For example, MONARCH is tuned with two storage tiers — level 0 respects to the compute node's local file system that is backed by the local SSD drive, while level 1 points to the dataset location at the shared PFS (*e.g.,* Lustre).

At startup time, before any request is submitted, the DL framework initializes a MONARCH instance. Here, MONARCH initializes the *metadata container* module by traversing the directory where the dataset resides (level 1) and builds the *file info* for each file.

At execution time, upon a read request made to a file X by the DL framework with the intent of reading a small portion of the file, MONARCH intercepts the request and verifies at the *metadata container* module which storage level where file X is placed/stored (①). Having this information, the read request is forwarded to the corresponding *storage driver* (*i.e.,* level 1) and submitted to the respective storage backend (②). After completing the request, the retrieved content is served back to the DL framework.

At the same time, MONARCH utilizes one of the *thread pool's* background threads (Ⓘ) to perform the data placement of X. First, the *placement handler* (Ⓒ) determines the level where X should be placed. It does so by hierarchically traversing the storage tiers (Ⓘⓘ), searching for the first one that has enough space to store the fetched content. In this case, level 0 is picked. At that time, another request is submitted to the *thread pool* to perform a copy of the file from level 1 to level 0 (operations ③ and ④). As explained previously, this mechanism copies the full content of the file so that subsequent read requests can be served from a faster storage tier. An independent thread performs this to avoid delaying the retrieval of the partial content requested by the DL framework. If the DL framework had requested the full content of the file operation, event ③ would not happen, and MONARCH would copy the content read in ② and place it on level 0 (④). Finally, the file storage level is updated to 0 (ⓘⓘⓘ) and the *storage occupation* of level 0 (Ⓘⓘ) is updated.

Throughout the DL workload execution, files are stored in this fashion until all available storage tiers (except for PFS) run out of storage space or if the first epoch has ended (*i.e.,* the full dataset is already stored at the upper storage levels). After the placement ends, MONARCH redirects the DL framework requests to their corresponding storage level.

C. Implementation

We implemented a MONARCH prototype with 1,500 lines of C++14 code. The *placement handler* module implements a *thread pool* of background threads that copy data samples between *storage tiers*. This *thread pool* was implemented using the C++ Thread Pool Library (CTPL), under 0.0.2 version [28]. We implemented the *metadata container*'s namespace with lookup tables using the Abseil library (v20210324.2) [29]. All MONARCH modules are thread-safe to enable multi-threaded environments.

Integration with TensorFlow. We integrated MONARCH with the TensorFlow framework. This framework was chosen due to its widespread use and adoption by the DL community. The integration of MONARCH with TensorFlow was reasonably straightforward. Namely, the latter already provides several interfaces to interact with different storage backends (*e.g.,* POSIX, HDFS, S3) and allows building custom drivers for other storage backend interfaces. Therefore, we developed a new driver based on the existing POSIX file system one and just replaced the `pread` invocation with our `Monarch.read` operation. The `Monarch.read` operation reads data from MONARCH rather than acessing the default POSIX storage backend. Contrarily to the POSIX `pread`, `Monarch.read` receives the filename as an argument, rather than its file descriptor. This integration only required changing 6 LoC at TensorFlow, originated by the middleware instantiation, driver registration, and the `pread` substitution. This process will be similar for other frameworks.

IV. EXPERIMENTAL EVALUATION

A preliminary evaluation was conducted for the MONARCH prototype to assess two main questions:

- Can MONARCH improve training performance for different DL models and dataset sizes?
- Can MONARCH reduce the I/O pressure on the PFS backend?

Experimental setup and methodology. The experimental setup, models, and methodology used in these experiments

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on July 01,2024 at 19:55:26 UTC from IEEE Xplore. Restrictions apply.

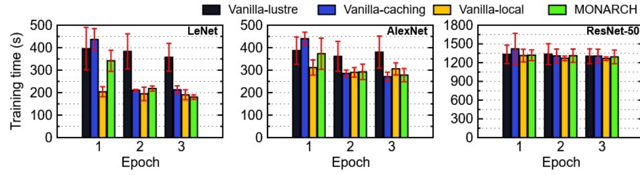

Fig. 3. Average training time and standard deviation for *Vanilla-lustre*, *Vanilla-local*, *Vanilla-caching* and MONARCH setups when using the LeNet, AlexNet and ResNet-50 models with the 100 GiB ImageNet-1k dataset. Results are detailed per training epoch.

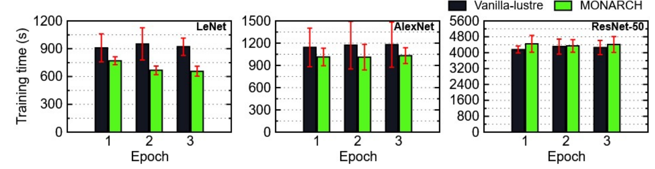

Fig. 4. Average training time and standard deviation for *Vanilla-lustre* and MONARCH setups when using the LeNet, AlexNet and ResNet-50 models with the 200 GiB ImageNet-1k dataset. Results are detailed per training epoch.

are the same as those described in Section II. The only differences are that the MONARCH prototype and a new transformed version of the ImageNet-1k dataset were added to the evaluation. This new dataset contains 3 million images, sizing at approximately 200 GiB, and was used to assess the scenario where data cannot fit entirely on the compute node's local storage and memory.

Configurations. MONARCH's prototype was configured with 6 threads for the *placement handler*'s thread pool and two storage levels for the *storage hierarchy* module. Namely, level 0 corresponds to the compute node's XFS file system mounted on top of a local SSD partition with 115 GiB of available storage space. Level 1 corresponds to the dataset directory stored at the Lustre file system. TensorFlow was used with the same configurations and optimizations discussed in Section II when deployed together with our middleware.

A. Training time

We start by analyzing the scenario where the ImageNet-1k dataset completely fits on the compute node's local disk (*i.e.,* same dataset used in the motivation experiments) and then, we discuss the results for the alternative version of the dataset that can only be partially stored on the local storage medium.

100 GiB ImageNet-1k dataset. Figure 3 shows that, in comparison with *vanilla-lustre*, using MONARCH significantly reduces average training time for LeNet and AlexNet. The LeNet model training time is reduced from 1205 to 811 seconds (33% decrease). As for the AlexNet model, the training time is reduced from 1193 to 1018 seconds (15% decrease). As previously discussed in Section II, the training performance is similar across all setups for the ResNet model.

The average training time for the first epoch of LeNet and AlexNet models is smaller for MONARCH when compared with both *vanilla-lustre* and *vanilla-caching*. However, one would expect this time to be similar across setups since all of them are reading data from the PFS backend. We believe this difference is justified by the placement mechanism implemented by our solution (Section III-A). Namely, when TensorFlow requests a portion of a TFRecord, MONARCH will read the full content of this record and place it at a faster storage tier. Therefore, subsequent portions of the TFRecord will be read from this level and not from the PFS, thus boosting I/O performance and reducing training time.

For the second and third training epochs, MONARCH and the other two setups that leverage local storage gain a considerable performance boost, when compared to *vanilla-lustre*, reducing training time by up to 47% and 23% (total elapsed time of 398 seconds and 570 seconds) for LeNet and AlexNet. Finally, the metadata initialization phase performed at the *metadata container* component takes approximately 13 seconds for the considered dataset.

200 GiB ImageNet-1k dataset. Since this extended dataset does not fit entirely on the local storage medium of compute nodes, only the MONARCH and *Vanilla-lustre* setups present viable solutions for its training (*Vanilla-caching* is not included because it requires the full dataset to fit into the local medium).

As depicted in Figure 4, the computational-bound ResNet model maintains similar performance for both setups. More interestingly, the LeNet model execution decreases from 2842 to 2155 seconds (24% reduction) when using our solution. For AlexNet, the average training time goes from 3567 to 3138 seconds (12% reduction). The improved training performance can be again explained by the decrease in the number of accesses to the Lustre file system, mainly on the second and third training epochs. Note that, since the dataset cannot be stored completely at the local storage medium, there are still I/O operations being issued to Lustre in the second and third epochs, namely approximately 360,000 operations (out of a total of 798,340 operations) at each epoch. Globally, during the full training workload, MONARCH reduces I/O operations to Lustre by an average of 55%. Given the size of the dataset, the metadata initialization phase performed at the *metadata container* component takes approximately 52 seconds.

B. Resource usage

For the 100 GiB dataset experiments, and in comparison with the results obtained in Section II, MONARCH demonstrates the second highest CPU and GPU utilization while only being surpassed by the *vanilla-local* setup. Namely, CPU and GPU usage was approximately 44% and 31% for LeNet model, 37% and 68% for the AlexNet, and 11% and 91% for ResNet.

For the 200 GiB dataset experiments, MONARCH is able to increase CPU and GPU efficiency when compared with the *vanilla-lustre* setup. In more detail, for LeNet, *vanilla-lustre* obtains 36% of CPU utilization and 30% of GPU utilization, while MONARCH increases those values to 46% and 38%. For AlexNet, MONARCH increases CPU usage from 31% to 33% and GPU usage from 63% to 69%. For ResNet, both setups

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on July 01,2024 at 19:55:26 UTC from IEEE Xplore. Restrictions apply.

use 9% of CPU and approximately 90% of GPU resources. Finally, memory consumption is identical for all setups and experiments and is approximately on the 10 GiB mark.

C. Summary

To conclude, the previous results show that MONARCH can improve training execution and avoid costly I/O operations to the Lustre PFS. Further, the optimizations discussed in the paper are relevant even for scenarios where the full dataset can only be partially cached at local storage, leveraging an average reduction of 55%, resulting in a decreased training time of 24% and 12%, for LeNet and AlexNet.

V. RELATED WORK

The DL storage bottleneck is currently a relevant and open research issue that has inspired different I/O optimizations.

Data loading and preprocessing. Some proposals improve DL data loading and preprocessing efficiency by resorting to different caching and prefetching algorithms. As examples, DALI [30] supports direct I/O prefetching from storage to GPUs, Pumma et al. [31] optimize Caffe's LMDB I/O subsystem to improve the mapping and caching of training data from storage to memory, while CoorDL [32] provides insights on storage I/O data stalls and mitigates them by providing a new in-memory caching policy.

Although our solution leverages some ideas from these works (*e.g.,* data caching policies, applicability to different frameworks), it is focused on leveraging the local disks of compute nodes to improve training performance and reduce the pressure on the PFS backend. Therefore, it is orthogonal to this work and can even be used in conjunction with it.

Data substitution and staging. Some solutions employ data substitution techniques [33]–[35] where training samples being served to DL frameworks are replaced by others (*e.g.,* cached files) that are faster to access. These techniques are useful for scenarios where several jobs are training models from the same dataset (*i.e.,* shared dataset). Differently, MONARCH optimizations are designed for single-job training scenarios where, to improve accuracy, each file of the dataset must be read once per epoch and in a random fashion. In this scenario, if the cache size is relatively small when compared to the full dataset, data substitution techniques either require accessing the PFS multiple times or may lead to fetching the same files repeatedly at each training epoch, thus potentially impacting training accuracy.

Serizawa and Tatebe [19] use the local disks of compute nodes to fully cache datasets to be trained with the Chainer framework. Furthermore, Fanstore [18] aggregates the local storage of several compute nodes to enable data sharing in distributed DL training environments. Finally, Diesel [36] resorts to local storage mediums and an external distributed key-value store service to cache data and metadata information needed for DL training workloads.

Again, MONARCH is designed for single-node training workloads and provides a simpler solution that avoids the need to allocate additional resources and orchestrating complex data and metadata staging areas, while not assuming that the dataset fits entirely on any of the intermediate storage tiers.

Storage tiering. NoPFS [17] utilizes a performance model to implement data prefetching and hierarchical caching. This solution can leverage different storage devices and hierarchies, while assuming that the training dataset may not fit entirely in each one of these.

MONARCH is focused on the performance impact of storage tiering and outsources data prefetching optimizations to the built-in mechanisms already present in frameworks such as TensorFlow, or provided by external solutions such as DALI. This enables MONARCH to be less intrusive than NoPFS, when being integrated with existing frameworks, and avoids changing the way users specify and deploy their DL scripts.

VI. CONCLUSION AND DISCUSSION

This paper presents MONARCH, a framework-agnostic middleware for deep learning that leverages the hierarchical storage organization present at HPC infrastructures. Preliminary results, resorting to various models and dataset sizes, show that a MONARCH-enabled TensorFlow can speed up DL training and reduce I/O pressure on the shared PFS backend.

Moreover, the work presented in this paper opens the path to several interesting research questions to be pursued.

Additional experiments. Other models, datasets, and DL frameworks should be considered to further assess MONARCH's contributions. Currently, we are integrating our system with PyTorch, which is an important step to validate MONARCH's portability and can additionally be used as the basis for comparison with the NoPFS solution that also targets this framework.

Consider more storage layers. MONARCH is designed and implemented to support different storage devices organized into multiple hierarchy levels. Therefore, it would be attractive to pursue experiments with additional hierarchy levels composed of other storage devices that may be available at supercomputers (*e.g.,* persistent memory or even RAM).

Distributed training. The scope of this paper is targeted towards local DL training. An interesting future research direction would be to expand MONARCH's design to support distributed DL training. This raises new questions regarding data placement and caching that must be addressed as multiple nodes will need access to different data shards of the dataset.

ACKNOWLEDGMENTS

Work financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia within project UTA-EXPL/CA/0075/2019 and PhD grant SFRH/BD/146059/2019 (Ricardo Macedo). This work used Frontera supercomputer which is supported by NSF grant #1818253 at Texas Advanced Computing Center.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on July 01,2024 at 19:55:26 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] "Open Images Dataset," 2017. [Online]. Available: https://github.com/cvdfoundation/open-images-dataset
[2] 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 Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.
[3] P. Schwan, "Lustre: Building a File System for 1000-node Clusters," in *Proceedings of the 2003 Linux Symposium*, vol. 2003, 2003, pp. 380–386.
[4] "BeeGFS." [Online]. Available: https://doc.beegfs.io/latest/overview/overview.html
[5] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *1st USENIX Conference on File and Storage Technologies*. USENIX Association, 2002, pp. 231–244.
[6] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–12.
[7] 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.
[8] J. Han, L. Xu, M. M. Rafique, A. R. Butt, and S.-H. Lim, ""a quantitative study of deep learning training on heterogeneous supercomputers"," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.
[9] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 750–759.
[10] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright, "A year in the life of a parallel file system," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 931–943.
[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 265–283.
[12] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," 2017.
[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015.
[14] "TensorFlow Tutorial: TFRecord and tf.Example," 2021. [Online]. Available: https://www.tensorflow.org/tutorials/load_data/tfrecord
[15] "MXNet Faq: Create a Dataset Using RecordIO," 2021. [Online]. Available: https://mxnet.apache.org/versions/1.8.0/api/faq/recordio
[16] T. H. Group, "Hierarchical data format version 5," 2000-2021. [Online]. Available: https://www.hdfgroup.org/solutions/hdf5/
[17] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning I/O," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2021.
[18] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "FanStore: Enabling Efficient and Scalable I/O for Distributed Deep Learning," 2018.
[19] K. Serizawa and O. Tatebe, "Accelerating machine learning i/o by overlapping data staging and mini-batch generations," in *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, 2019, p. 31–34.
[20] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The evolution of leadership computing at the national science foundation," in *Practice and Experience in Advanced Research Computing*. Association for Computing Machinery, 2020, p. 106–111.
[21] "AI Bridging Cloud Infrastructure." [Online]. Available: https://abci.ai
[22] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.
[23] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.
[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2016, pp. 770–778.
[26] "TensorFlow API: tf.data.Dataset.cache." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset#cache
[27] S. Sarkar, "A Scalable Artificial Intelligence Data Pipeline for Accelerating Time to Insight," 2019, SNIA Storage Developer Conference. [Online]. Available: https://www.snia.org/sites/default/files/SDC/2019/presentations/Machine_Learning/Sarkar_Sanhita_A_Scalable_Artificial_Intelligence_Data_Pipeline_for_Accelerating_Time_to_Insight.pdf
[28] "CTPL." [Online]. Available: https://github.com/vit-vit/CTPL
[29] "Abseil." [Online]. Available: https://abseil.io/
[30] "NVIDIA Data Loading Library." [Online]. Available: https://developer.nvidia.com/dali
[31] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable Deep Learning via I/O Analysis and Optimization," *ACM Transactions on Parallel Computing*, vol. 1, no. 1, pp. 1–34, 2019.
[32] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and Mitigating Data Stalls in DNN Training," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 771—-784, 2021.
[33] D. Choi, A. Passos, C. J. Shallue, and G. E. Dahl, "Faster neural network training with data echoing," 2020.
[34] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, ""entropy-aware i/o pipelining for large-scale deep learning on hpc systems"," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 145–156.
[35] A. V. Kumar and M. Sivathanu, "Quiver: An informed storage cache for deep learning," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, 2020, pp. 283–296.
[36] 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*. Association for Computing Machinery, 2020.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on July 01,2024 at 19:55:26 UTC from IEEE Xplore. Restrictions apply.