DStore: A Lightweight Scalable Learning Model Repository with Fine-Grained Tensor-Level Access

Meghana Madhyastha
mmadhya1@jh.edu
Johns Hopkins University
Baltimore, Maryland, USA

Robert Underwood
runderwood@anl.gov
Argonne Nat. Laboratory
Lemont, Illinois, USA

Randal Burns
randal@cs.jhu.edu
Johns Hopkins University
Baltimore, Maryland, USA

Bogdan Nicolae
bnicolae@anl.gov
Argonne Nat. Laboratory
Lemont, Illinois, USA

ABSTRACT

The ability to share and reuse deep learning (DL) models is a key driver that facilitates the rapid adoption of artificial intelligence (AI) in both industrial and scientific applications. However, state-of-the-art approaches to store and access DL models efficiently at scale lag behind. Most often, DL models are serialized by using various formats (e.g., HDF5, SavedModel) and stored as files on POSIX file systems. While simple and portable, such an approach exhibits high serialization and I/O overheads, especially under concurrency. Additionally, the emergence of advanced AI techniques (transfer learning, sensitivity analysis, explainability, etc.) introduces the need for fine-grained access to tensors to facilitate the extraction and reuse of individual or subsets of tensors. Such patterns are underserved by state-of-the-art approaches. Requiring tensors to be read in bulk incurs suboptimal performance, scales poorly, and/or overutilizes network bandwidth. In this paper we propose a lightweight, distributed, RDMA-enabled learning model repository that addresses these challenges. Specifically we introduce several ideas: compact architecture graph representation with stable hashing and client-side metadata caching, scalable load balancing on multiple providers, RDMA-optimized data staging, and direct access to raw tensor data. We evaluate our proposal in extensive experiments that involve different access patterns using learning models of diverse shapes and sizes. Our evaluations show a significant improvement (between 2 and 30× over a variety of state-of-the-art model storage approaches while scaling to half the Cooley cluster at the Argonne Leadership Computing Facility.

CCS CONCEPTS

• **Computing methodologies** → *Search methodologies*; *Distributed computing methodologies*; • **Information systems** → **Parallel and distributed DBMSs**.

KEYWORDS

DL model repository, fine-grained tensor storage and access, benchmarking

ACM Reference Format:
Meghana Madhyastha, Robert Underwood, Randal Burns, and Bogdan Nicolae. 2023. DStore: A Lightweight Scalable Learning Model Repository with Fine-Grained Tensor-Level Access. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3577193.3593730

ICS '23, June 21–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0056-9/23/06.
https://doi.org/10.1145/3577193.3593730

1 INTRODUCTION

Deep learning (DL) models are widely used both in industry and in scientific computing: speech and vision [8], fusion energy science [17], computational fluid dynamics [18], cancer research [9], and pandemics [34].

In this context, the unprecedented accumulation of big data and the development of computational capabilities expose plentiful learning opportunities thanks to the data's massive size and variety [19, 27]. In a quest to take advantage of the complex patterns in the data, DL models are becoming complex themselves from many perspectives [39]: size (number of parameters), depth (number of layers/tensors), and structure (layers with multiple inputs/outputs interconnected using directed graphs that feature divergent branches, fork-join, etc.). Under such circumstances, model training is time-consuming and resource-intensive, which is why artificial intelligence (AI) workflows try to avoid retraining DL models from scratch as much as possible and instead serialize them to a repository, in anticipation of opportunities to reuse them later. However, by interacting with the repository frequently and under concurrency, the repository becomes a bottleneck in AI workflows. Thus, the problem of how to design scalable, high-performance repositories for DL models is important but challenging for several reasons.

High-frequency access to the repository under concurrency at scale: Storing and loading DL models to/from a repository are frequent operations that may cause an I/O bottleneck for AI workflows. For example, unexpected interruptions prompt the need to checkpoint DL models more frequently than an epoch boundary, potentially at an individual iteration [26]. The study of sensitivity analysis, which considers the robustness of inference to training data, involves numerous comparisons with variations of DL models checkpointed during the training with different subsets of the data [42]. Ensemble learning [43] also relies on training and storing a large number of alternative DL models, which are used later to improve the accuracy and/or generality of the inferences. Furthermore, AI workflows typically run on high-performance computing (HPC) platforms at scale, leveraging multiple compute nodes and graphics processing units (GPUs) simultaneously. In this context, the store and load operations are issued to the model repository with both high frequency and concurrency.

Fine-grained access to the tensors of the DL model: In a majority of applications where DL models can be reused, the original context is slightly different, prompting the need for adjustments after adopting *transfer learning* [36]. This typically involves multiple

steps: (1) load the content of reusable layers from a related model from the repository into the new DL model; (2) freeze the reused layers and train the rest of the layers with additional data; and (3) store the new DL model in the repository. In this case, loading the whole DL model from the repository and extracting individual tensors from the local copy are inefficient. Instead, the repository should provide the capability to directly access tensors at fine granularity. Furthermore, transfer learning is not an occasional pattern. For example, network architecture search (NAS) is a notoriously difficult AI workflow that automatically generates, trains, and evaluates a large number of related DL model variations [9]. NAS can scale to a large number of workers and has been shown to benefit from transfer learning [22], because the DL models are related. Thus, fine-grained access to the tensors is often needed together with high-frequency access under concurrency.

Limitations of state-of-the-art: Open repositories for expert- or user-designed DL models are typically implemented as web-enabled services that enable users to upload, classify, curate, and search for DL models (e.g., Tensorflow Hub, Caffe's Model Zoo, DLHub [21]). In this case, the emphasis is on providing rich features and ease of use, rather than high performance and scalability. Even on HPC platforms, typical approaches to store and load DL models involve significant overheads: the tensors in the composition of the DL models are serialized into custom formats (e.g., HDF5 [37]) or SavedModel [7]), which are then written to a parallel file system (PFS) acting as a repository shared by the compute nodes. These overheads have multiple causes, both related to the serialization (bloated metadata, too many files) and due to the PFS itself becoming a bottleneck (limited I/O bandwidth under concurrency and poor support for small, noncontiguous I/O operations).

Contributions: We focus on the design on a distributed DL model repository that delivers high performance and scalability for high-frequency, concurrent load/store operations that may involve fine-grained access to individual tensors. In order to overcome the limitations of state-of-the-art approaches, our key idea is to decompose the DL models into lightweight collections of tensors and their associated metadata, which are organized using novel data structures, indexing, and caching techniques specifically optimized for direct manipulation using distributed, RDMA-enabled key-value stores. We summarize our contributions as follows.

First, we propose a series of design principles: compact representation of the DL model architecture, stable hashing of layer configurations with client-side metadata caching, low-overhead memory management based on preallocated and pre-pinned buffers, and tensor storage layout optimized for bulk transfers (Section 3). Second, we implement these design principles and techniques as a research prototype that makes use of state-of-the-art technologies to interface with popular AI runtimes (such as TensorFlow) in order to obtain direct access to the tensors, as well as state-of-the-art communication libraries that provide RDMA-enabled RPC abstractions suited for the development of services on HPC systems (Section 5). Third, we evaluate our proposal in a series of extensive experiments conducted on the Cooley HPC system at the Argonne Leadership Computing Facility. We focus on the performance and scalability of loading and storing a variety of DL models (different number of layers, uniform vs. variable sizes of layers, different total size, etc.) both individually and under concurrency. The results show significant improvement over several state-of-the-art approaches that leverage either a parallel file system or in-memory storage (Section 6).

2 RELATED WORK

DL model checkpointing: Popular runtimes such as TensorFlow and PyTorch serialize the tensors of DL models into custom formats (e.g., HDF5 [37] or SavedModel [7]), which are then written as files to a parallel file system (PFS). This process incurs high serialization and I/O overheads. Optimized checkpointing approaches exist for data-parallel training [29]. They take advantage of multiple identical model replicas to parallelize the writes to different shards. Other checkpointing efforts such as [26] focus on determining the optimal checkpointing interval through systematic online profiling of the overhead. FlameStore [1] is an effort that aims to reduce the serialization overheads by directly capturing tensors and storing them into configurable providers (in-memory, local file system). Such approaches either suffer from I/O performance and scalability bottlenecks due to relying on a parallel file system or are not optimized for fine-grained access to subsets of tensors.

Web-enabled DL model repositories: These repositories are optimized for collaborative rapid application development. They enable users to upload, classify, curate, and search for DL models. Prominent examples include open repositories such as Tensorflow Hub [6], PyTorch Hub [5], and Caffe's Model Zoo [3]. Some efforts, such as DLHub [21], target scientific applications specifically. For web-enabled repositories, the emphasis is on providing rich features and ease of use, rather than high performance and scalability. One notable feature is versioning of DL models. For example, Data Version Control [12] relies on Git's version control capabilities to store metadata references to large binary and textual objects that represent training data and DL model checkpoints. However, the serialization method and actual storage of the objects are outside of the purview of the repository, which it delegates to either web-based data transfers to/from remote servers or parallel file systems that are not optimized for the storage of many small, scattered tensors that compose a DL model.

Key-value and object stores: Hash tables, key-value stores, and remote dictionaries [13] are fundamental building blocks to store both ephemeral and durable data at fine granularity. They feature a multitude of backends (in-memory storage, flash and persistent memory, file systems) and advanced features such as multi-versioning [28]. In the same spirit, object storage systems such as DAOS [24] and RADOS [40] also aim to alleviate the limitations of the aging POSIX semantics employed by parallel file systems such as Lustre [2] and GPFS [38]. Some optimizations such as leveraging foreknowledge about the read access pattern of training samples can be used to accelerate AI workloads [23]. However, such building blocks cannot be leveraged out of the box to manipulate DL models, because it is nontrivial to serialize and consistently store models as a collection of tensors mapped to key-value pairs. Additionally, the key-value pairs cannot be simply scattered across a large number of providers, because the overhead of assembling a DL model from a large number individual tensors would involve large communication overheads due to the need to contact a large number of providers concurrently.

Remote data access: Web-enabled services often expose RESTful APIs that enable clients to access data on servers. However, such approaches involve expensive encoding of parameters and states into text representations (JSON, XML) to maximize portability at the expense of performance/latency. RDMA is a well-established backbone of communication libraries with low-latency, high-throughput message-passing requirements. It is used extensively in HPC data centers and storage systems [44]. In this context, MPI proposes the notion of single-sided communication [16] that involves memory windows declared by each rank as the target for put/get operations. Unfortunately, such a model is not flexible enough for client-server use cases, since it involves collective operations (initialize memory windows, synchronization based on fences, etc.). Recently, HPC-oriented communication libraries based on RPCs are gaining increasing traction [33], thanks to higher flexibility compared with MPI. Such approaches can be used as a backbone for DL model repositories.

To the best of our knowledge, this is the first work that explores the problem of building a DL model repository suitable for HPC systems that addresses the needs of AI workflows interacting with the repository frequently under concurrency while enabling fine-grained tensor-level access.

3 SYSTEM DESIGN AND DESCRIPTION

Compact representation of the DL model architecture graph: DL model architectures have evolved from simple sequential arrangements of the layers to complex arrangements that involve, for example, branching, repetitive structures, and fork-join patterns. In the most general form, these arrangements can be expressed as graphs. In practice, layers are defined in a recursive fashion [14], forming a graph of sublayers down to leaf layers that are actually composed of tensors (that hold relevant parameters such as weights and biases). Thus, to reconstruct a previously stored DL model, we need the following elements: (1) the graph of the leaf-layers, (2) mapping of the leaf layers to tensors, and (3) the content of the tensors. State-of-the-art solutions [1, 37] capture the graph using a standardized format (e.g., JSON), which is supported by popular AI libraries (e.g., Keras [14]). Then, based on a label given to each leaf layer, the tensors can be serialized independently as key-value pairs using a standard format, such as HDF5 [37]. Metadata formats such as JSON describe each layer using comprehensive and verbose information: name of the layer's class, shape, label (a descriptive string), relationship to other leaf layers (provides output to or needs input from), and so forth. Such solutions, while human-readable and user friendly, may incur large metadata overheads during store/load operations, especially for DL models with a large number of small tensors. Consequently, our first contribution is to introduce a compact representation of the graph: as we traverse the recursive definition of the architecture bottom up, we extract for each leaf layer the list of other leaf layers whose input is a direct dependency (henceforth referred to as input dependencies). Then, we assign a unique ID (whose representation fits in a small number of bits) to each leaf layer and obtain a compact list of its input dependencies based on their unique IDs. By focusing only on the leaf layers and using small unique ID references, we both reduce the metadata size and make the architecture easier to parse.

Stable hashing of layer metadata that retains the features of the DL model architecture: Obtaining unique IDs for the leaf layers of the DL model architecture is challenging because layers of identical type and with identical shape can be reused in different parts of the graph. Also, we cannot simply rely on the labels given to the layers because they are assigned in an inconsistent fashion. For example, during the recursive layer definition, Keras [14] keeps an internal counter that increases every time the same type of layer is encountered, then automatically labels it using a string that combines the layer class name and the counter value. Later, if the DL model architecture is instantiated in a different way or multiple DL model architectures are instantiated in the same way but in a different order, the labels of the same leaf layers will not match. To overcome this challenge, we propose a stable hashing technique inspired by Merkle trees [25], which is detailed in Section 4. The key idea is to hash the structurally significant attributes of the layer together with hashes describing the ancestry of its input dependencies. This enables unique IDs regardless of how many repetitions of layers or entire substructures exist in the architecture. Note that we abuse the term unique ID with this approach, because there is a theoretical chance of hashing collisions. By using a cryptographically secure hash function, however, this chance is negligible in practice.

Client-side cached mapping of leaf layer and unique IDs: To reconstruct the DL model architecture from a compact graph representation, we need to maintain a mapping between the unique IDs and the properties of their leaf layers, such that we can create new layer class instances (or reuse existing ones) as needed. In this context, large numbers of DL model architectures can be reused either fully (e.g., checkpointing the same model) or partially (e.g., transfer learning). This applies to many of the scenarios mentioned in Section 1. Therefore, we leverage this observation in order to maintain a repository of cached layer metadata and their mapping to unique IDs. For performance considerations, these mappings are cached locally on the clients. When a new DL model needs to be stored, only the new layers (different shape or different predecessor ancestry) need to be hashed to obtain a unique ID, while the rest can be directly reused. Similarly, when a DL model needs to be loaded, the cache can be used to directly instantiate leaf layers from unique IDs. Since we do not overwrite existing metadata, cache synchronization is not a concern, thus enabling a lightweight and highly scalable implementation.

Distributed key-value store providers: To achieve scalability, we store both the compact architecture graph and the collection of tensors as key-value pairs on a set of distributed providers, which can be either co-located with the clients on the same set of compute nodes or hosted by a separate set of dedicated nodes. Two challenges arise in this context: (1) how to enable the clients to find out what providers to contact in order to recover the architecture and/or content of the DL model and (2) how to distribute the requests of concurrent clients to different providers in order to achieve load balancing and scalability. We propose to solve both problems simultaneously by leveraging stable hashing: we require the clients to assign unique names to the DL model instances. Based on the unique names, we obtain unique model IDs, which can be mapped to a deterministic set of providers (e.g., unique model instance ID modulo the number of providers) that are responsible for them.

RDMA-enabled distributed tensor grouping and metadata consolidation: Given the huge number of tensors, it is not feasible to simply distribute each tensor as a separate key-value to a different provider, because this will force the clients to issue a large number of tiny put/get requests to a large number of providers, which results in degraded I/O performance. Hence, we propose to consolidate the tensors of a store/load request into a single provider. However, simply aggregating all tensors as a single block of contiguous data that is then transferred between clients and providers also incurs a large overhead. Furthermore, after a store request, we need to retain efficient fine-grained access to the tensors for future read requests. To address these issues, we leverage bulk RDMA operations: the client assembles a set of (offset, size) segments corresponding to the in-memory tensors, while the provider allocates memory and pulls these segments locally. This results in a single transfer operation that collects the tensors as they are scattered in memory directly. Next, the client stores the compact architecture graph of the DL model together with the segments corresponding to the tensors on the same provider. This metadata consolidation step allows the segments to be reused later during read requests, which reduces the setup overhead for RDMA bulk operations. Specifically, the client exposes preallocated memory regions into which the provider directly pushes the content of the tensors.

Preallocated and pre-pinned staging buffers to accelerate RDMA transfers: Even if we leverage bulk RMDA operations to enable low-latency, high-throughput communication between the clients and the providers, other overheads can negatively impact the overall performance of load and store operations. In particular, the memory regions involved in bulk RDMA transfers between a client and a provider need to be pinned by the operating system. This operation is needed in order to make sure the corresponding physical addresses of memory pages remain fixed and can be directly accessed by the network interfaces. However, pinning a large number of memory pages is an expensive operation. Therefore, we propose to use a fixed staging buffer on both the clients and the providers, which is used for all bulk RDMA-operations and is preallocated and pre-pinned on initialization of the client and provider, respectively. Additionally, continuously allocating and deallocating memory on demand to store tensors are expensive, and preallocating on initialization removes this overhead. These staging buffers are nontrivial to manage, however, especially under concurrency: the same client may issue concurrent load/store requests to different providers, and the same provider may serve concurrent load/store requests from different clients. To address this issue, we introduce a custom allocation/deallocation strategy on the staging buffer specifically optimized for concurrency.

Native tensor operators to obtain direct access to the tensor content: High-level AI libraries are often implemented in high-level languages (e.g., Python) and do not have direct access to the data structures of the underlying low-level implementation. Therefore, there are restrictions in terms of when and how it is possible to access such low-level data structures. For example, the TensorFlow implementation of Keras requires a separate execution graph context in order to extract the value of tensors into separate copies (e.g., NumPy arrays). Consequently, extracting the raw tensors of DL models may incur high overheads. These overheads are often present in state-of-the-art DL model serialization approaches. For example, DL model serialization into the HDF5 [37] format involves such expensive copies of tensors into NumPy arrays, which are then written through the HDF5 Python bindings into a file. The reverse process applies in the case of reading DL models. We address this issue by tapping directly into the memory allocated for the tensors of the DL model, by defining native low-level tensor operators that have direct access to the raw pointers.

4 ZOOM ON STABLE HASHING

As mentioned in Section 3, we propose an algorithm inspired by Merkle trees [25] to enable stable hashing of the layer metadata that retains the features of the DL model architecture. This algorithm is needed in order to uniquely identify the leaf layers in the compact architecture graph of the DL model, independently of preassigned labels and tolerant to repetitions of identical layers types and and/or substructures. It is listed in Algorithm 1.

Algorithm 1: Assign Unique IDs

Input : Architecture of model M
Output: A map of all leaf-layer $\rightarrow$ *unique_id*

Function $LeafLayerHash(l, L)$**:**
  $H \leftarrow Hash(l.config)$
  foreach $l_i \in l.inputs$ **do**
    if $l_i \notin L$ **then**
      $L \leftarrow LeafLayerHash(l_i, L)$
    $H = Hash(H, L[l_i])$
  $L[l] \leftarrow H$
  return L
$L \leftarrow \emptyset$
foreach $l \in$ *outputs of* M **do**
  $L \leftarrow LeafLayerHash(l, L)$
return L

Specifically, starting from the output layers of DL model M, we hash each of the attributes affecting the model structure of each leaf layer l (layer class name, shape, etc. captured by l.get_config) with the hashes of its direct dependencies ($l_i \in l.inputs$) and store the resulting hash into a memoization table L. For any direct dependency l_i that was not hashed yet, the process repeats recursively. Thanks to the memoization table L, each leaf layer is hashed only once.

The complexity of this algorithm is $O(|V| + |E|)$, where $|V|$ is the number of leaf layers and $|E|$ is the number of input edges in the architecture graph.

5 IMPLEMENTATION

We implemented a research prototype based on the design principles introduced in Section 3. Specifically, the repository follows a client-server architecture: the clients issue store/load DL model requests, while the providers collaborate to serve these requests. An overview is depicted in Figure 1.

The interactions between the clients and the providers leverage bulk RDMA transfer operations, which are optimized by using HPC-oriented RPCs, as provided by Mochi [33]. Specifically, we leverage

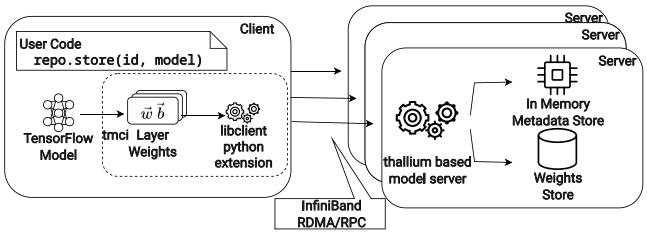


Figure 1: System overview: The repository exposes a high-level Python API that integrates seamlessly with DL runtimes (e.g., TensorFlow). Underneath, the clients access the tensors of the DL model directly (using native C++ execution graph operators) and transfer them using HPC-oriented RPCs to/from a set of distributed providers, responsible for storing the compact architecture graph and the tensor content at fine granularity in a scalable fashion.

the following components: Mercury, which provides low-level RPC abstractions with bulk RDMA support for a variety of network interfaces and transport protocols; Argobots, which enables user-level threads for high performance concurrency control; Margo, which combines Mercury and Argobots to expose high-level abstractions for building providers that can serve RPCs concurrently; and Thallium, which integrates the rest of the components as a single package that exposes concise C++ abstractions with serialization support.

The store primitive is depicted in Figure 2. On initialization, both the clients and the providers initialize their staging buffers and pre-pin the host memory. When a store operation is issued at Python level, the tensors are copied from the GPU to the staging buffer. This involves two aspects: (1) obtaining fine-grained access to the raw tensors using native C++ low-level tensor operators and (2) reserving space in the staging buffer for the tensor copies (potentially under concurrency with other load/store requests). The latter is implemented by using the polymorphic memory resource std::pmr::synchronized_pool_resource abstraction for dynamic memory management as provided by the C++17 standard. It exposes a pooled allocator design based on a power-of-two big bag of pages. Once the tensors are available in the staging buffer, we compute the compact architecture graph representation using our stable hashing approach, which is subject to client-side caching. At this point, the client is ready to initiate RDMA bulk operations to send the tensors to the provider, which stores them in a key-value pair collection. This is subject to reusing the provider-side staging buffer, which is implemented in a fashion similar to that for the client-side staging buffer. From here, the key-value pair collection is stored by the providers into configurable buckets with different persistence properties. The only remaining operation is sending the compact architecture graph to the provider, to be stored together with the DL model, which happens in a separate remote procedure call.

The buckets responsible for storing collections of tensors can be either KV stores such as RocksDB [31] (for which persistency is guaranteed by the underlying backing file that is assumed to survive the AI workflow) or lightweight in-memory hash tables. The latter are particularly useful for AI workflows that do not require

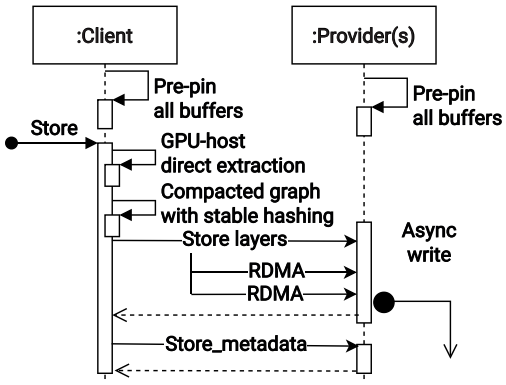


Figure 2: Store primitive sequence diagram

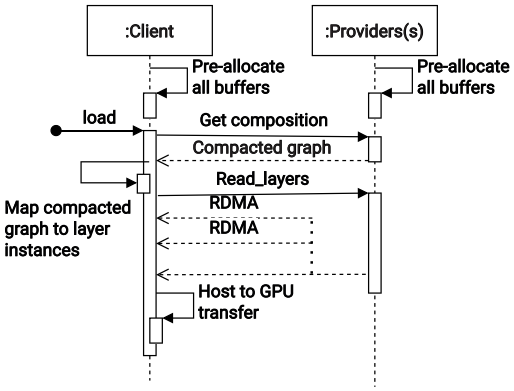


Figure 3: Load primitive sequence diagram

persistency (campaigns that reuse a large number of DL models during runtime but need only a small subset after completion) or can afford to run the providers as a permanent service, in which case the in-memory hash tables can be protected by means of replication and/or erasure coding.

The load primitive is depicted in Figure 3. Similarly, both the clients and the providers initialize their staging buffers and pre-pin the host memory. Next, the client issues a dedicated remote procedure call to obtain the compact graph representation. Then it uses the client-side cache to map the unique IDs of the leaf layers to actual layer instances (which are initialized on the fly if necessary). Next it reserves space on the staging buffer (using the same approach as in the case of the store primitive) and initiates transfer of the chosen tensors. Note that this may involve fine-grained access to specific tensors. In its turn, the provider replies with a bulk RDMA transfer of the tensor content. This is highly optimized, especially when using an in-memory hash table bucket, since the segments of the RDMA bulk transfer are precached on the staging buffer. Once the client receives the tensors into its own staging buffer, it can initiate the copy of the tensor content from the staging buffer to the GPU memory.

Our implementation provides high-level bindings for TensorFlow (Python-level) and is intended to be a user-friendly alternative to the standard save and load primitives that enable serialization and de-serialization of DL models to/from the HDF5 and SavedModel

formats mentioned in Section 2. With the exception of the client-side cache that retains the mapping between unique IDs and layer instances, most other aspects are high-performance C++ implementations that live in a client library, which is interfaced with the Python-level primitives. The providers are fully implemented in C++. Thanks to this isolation, our implementation is generic and can be easily ported to other AI runtimes, notably PyTorch.

6 EXPERIMENTAL EVALUATION

6.1 Setup

We perform our experiments on Cooley, an Argonne Leadership Computing Facility (ALCF) HPC system comprising 126 compute nodes. Each node is equipped with two Intel (Haswell) 2.4 GHz E5-2520 CPUs and an Nvidia Tesla (Kepler) K80 accelerator. The K80 accelerator is a dual-GPU design that exposes 2 Tesla GK210 GPUs across a PLX switch over a PCI Express connection to the card (max 15.6 GB/s). Each node features 372 GB DDR4 RAM on the host side and 24 GB of high bandwidth memory split evenly between the two GPUs. The nodes are interconnected using a 4xFDR InfiniBand network (max 7 GB/s). The same network provides access to a Lustre parallel file system (10 PB) capable of ≈210 GB/s in peak bandwidth across 56 object storage targets. Since our experiments do not involve GPU computations and rather focus on I/O throughputs, this setup is representative of data centers with newer GPU accelerators as well.

In terms of software ecosystem, the AI runtime we installed is TensorFlow 2.9. The bulk RDMA-enabled RPC runtime is based on the following stack: Mercury 2.1.0 (using OFI verbs provider in libfabric), Argobots 1.1, Margo 0.9.10, and Thallium 0.10.0 [33].

6.2 Compared Approaches

We compare our proposal with state-of-the-art approaches and with an alternative implementation that includes only a subset of our design principles (used as a reference in ablation studies to highlight the impact of individual contributions.

HDF5-PFS: This approach stores and loads DL models as HDF5 [37] files. It is part of the standard TensorFlow distribution and implemented in Keras. The store primitive first copies the content of the tensors into NumPy arrays (by launching a separate TensorFlow execution context), then writes the arrays into an HDF5 file using the HDF5 Python bindings. The load primitive loads the arrays from the HDF5 file, then copies their content into the tensors managed by TensorFlow. We chose a minimalist setup that only stores the model weights in the HDF5 file, which is the only mode that supports fine-grained read access to the tensors (by specifying a set of layer labels to be loaded) and is typically employed for performance reasons. On HPC systems, AI workflows typically use a parallel file system to store the resulting HDF5 files. In our case, this is a Lustre deployment. Overall, this configuration is a typical baseline used in production on an HPC system.

SavedModel-PFS: this approach is similar to HDF5–PFS in that it serializes and deserializes the DL models as files on the Lustre parallel file system. However, there are notable differences due to a different SavedModel format. Specifically, the internal representation of the DL model is decomposed into an entire set of files that hold additional metadata about the DL model (DL model architecture, state of the optimizer, mathematical descriptions of custom operators). To ensure a fair comparison, we store only the model architecture and disable the rest of the metadata. Just as in the case of HDF5–PFS, the overhead of explicit copies between tensors and intermediate arrays is also present. Unlike HDF5–PFS, however, fine-grained access to the tensors is not available. Some AI workflows prefer this format because of the additional information it captures about the DL models, which makes it another representative baseline.

FlameStore [1]: This state-of-the-art approach adopts a client-provider model similar to our approach. Like our approach, it has a mechanism to capture raw pointers to tensors and uses RDMA to facilitate the communication between the clients and the providers. The tensors are stored in an in-memory hash table that delivers the highest performance and scalability. Unlike our proposal, however, it does not feature metadata optimizations (it stores the DL model architecture in full using a JSON format), does not support fine-grained read access to the tensors, and does not include optimized caching and memory management.

DStore-Opt: This approach implements all our proposed design principles described in Section 5 and runs with all optimizations active. As in the case of FlameStore, the buckets used to store the tensors are our own version of in-memory hash tables optimized for concurrent access and subject to the proposed design principles. DStore–Opt showcases the best peformance and scalability that can be achieved compared with the state of the art.

DStore-OnDemand: This approach is identical to DStore–Opt except for deactivating the preallocated and pre-pinned staging buffer used for memory management. Instead, it simply performs on-demand allocations/deallocations and pinning of the memory for RDMA operations. Its purpose is to enable us to isolate and study the impact of our custom memory management technique.

6.3 Methodology

We evaluate the performance and scalability of the approaches described in Section 6.2 for a variety of DL model architectures that feature large vs. small overall DL model size, large vs. small number of leaf-layers, and various distributions of the tensor sizes (layers of similar size vs. layers of different sizes). Thanks to this approach, we cover a large spectrum of the diversity of the DL models encountered in real-world AI applications.

Our goal is to show that DStore can efficiently (1) store and transfer both small and large models; (2) handle both simple and complex metadata (metadata complexity is proportional to the number of leaf-layers); and (3) adapt to the heterogeneity of the leaf-layers (mix of small and large tensors).

Unless otherwise mentioned, we co-locate providers and clients launched with the compute job. This is the most commonly encountered scenario. Note that if the HPC system allows it, the providers can also be run as a permanent service on a separate set of nodes. For brevity, we do not explore this configuration.

Synthetic workloads: We designed a DL model architecture generator that takes three input parameters: (1) total size of the DL model (in bytes); (2) number of leaf-layers; and (3) maximum difference between the layer sizes (in bytes).

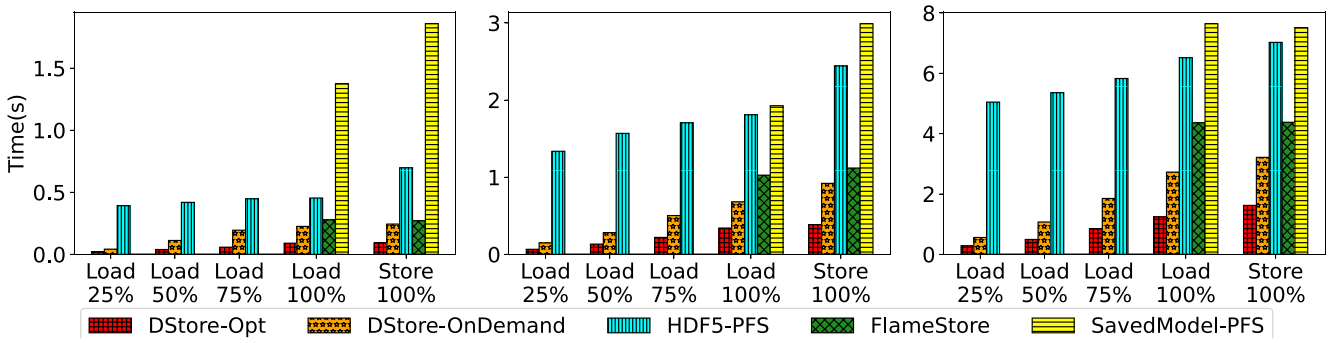


Figure 4: Variable model size (256 MB, 1 GB, and 4 GB): single-client partial load (fine-grained tensor reads) and store (writes) of DL models (10 layers of uniform size). FlameStore and SavedModel do not support partial load. Lower is better. Our approach is $\approx 5\times$ to $21\times$ faster.

The generator is inspired by network architecture search approaches, notably DeepHyper [9], which also generate DL models that satisfy a set of constraints. In our case, however, the goal is to control the aspects that are the most relevant in the context of DL model storage: complexity of the architecture and metadata, total size of data being transferred, and contiguity of the data transfers (many small tensors scattered at different memory locations are more difficult to transfer than a few large contiguous tensor, even if they add up to the same size). Using this approach, we can evaluate the performance and scalability of the compared approaches for a diverse set of scenarios that are applicable to a broad set of AI workflows. To enable a fair comparison, we generate a fixed set of DL models in the beginning of the experiment, and then we reuse the exact same DL models for all compared approaches.

Real-life DL models: To complement the synthetic DL models, whose purpose is to cover a broad range of architectures, we also use real-life DL models in our evaluations. To this end, we rely on the Nvidia NGC Catalog [4] as a provider of reference implementations. Specifically, we consider VGG19 [35], a 549 MB convolutional neural network used for image classification and BERT–large [11], a 1.22 GB transformer language model. In the case of BERT–large, we consider a scenario that appends a classifier head consisting of several large, dense layers bringing the total model size to 3 GB. Such adaptations are frequently used to build complex classifiers that accept natural language as input [14, 30, 32]. For example, they can be used in sentiment analysis (angry, impatient, calm, etc.) or to infer entities based on their descriptions. In this case, the components of the DL model are typically reusable, which justifies the need for fine-grained access to extract them.

Metrics: we instantiate each DL model with random weights (the tensor content does not affect the serialization overheads and/or data transfer throughput/latency). Then we measure the duration of the store and load operations respectively for each of the compared approaches. This measurement is the end-to-end overhead perceived by the Python-level AI workflow. For experiments that involve concurrent load and store operations, we report the average duration. In addition to the end-to-end overhead, some of our experiments also provide a breakdown of the cost of the low-level operations performed during load and store. In this case, the discussions of the results provide more details about the metrics.

6.4 Results: Different Model Size

First we study the end-to-end performance of a single client served by a single provider. Both the client and the provider are deployed on a different compute node.

We consider how each of the compared approaches performs for a simple synthetic DNN model that uses 10 dense layers of equal size and a variable total size of 256 MB, 1 GB, and 4 GB. Such DL models types exhibit minimal metadata but large tensors. They are common in applications powered by CANDLE [41] (Cancer Distributed Learning Environment), which aims to combine the power of exascale computing with deep learning to address a series of loosely connected problems in cancer research. For each configuration, we first store the DL model using one client, then read it back using a different client. The clients are deployed on separate compute nodes to avoid any node-local caching effects. The results are depicted in Figure 4.

The duration of the store operation scales roughly linearly with the model size, which is expected given the the small number of large, contiguous tensors (a favorable I/O pattern for parallel file systems). We note the large overhead of pinning the host memory for RDMA transfers: thanks to the preallocated and pre-pinned staging buffer, DStore–Opt is consistently 2x faster than DStore–OnDemand. We explore this further in Section 6.5. Also interesting is the impact of the other design principles. Compared with FlameStore, which also uses an in-memory key-value store, DStore–OnDemand is up to 25% faster, despite minimal metadata needed to represent a simple model of only 10 layers. Compared with SavedModel–PFS, our approach is up to two orders of magnitude faster, especially for small DL models. HDF5–PFS is significantly faster than SavedModel–PFS, which can be attributed to the fact that the HDF5 format is better optimized for Lustre. However, it still lags far behind our proposal, being up to 30x slower.

In the case of the load operations, we vary the total size of the layers that are read back from 25% up to 100% (full model). This simulates various degrees of fine-grained access to the tensors, as discussed in Section 1. The duration of the load operations follows a trend similar to the case of the store operations, especially when we read back all tensors. Note that FlameStore and SavedModel do not support fine-grained access to the tensors; hence they are depicted only in the 100% case. HDF5–PFS has high overhead for fine-grained

access: despite loading only 25% of the tensors, the overhead is almost the same, especially for small sizes. On the other hand, our approach has much better scalability, since the duration of load grows proportionally to the total size of the tensors being read back.

6.5 Results: Low-Level Operation Breakdown

Next, we focus on a breakdown of the overheads of the low-level operations that are involved in the workflow diagrams of the store and load primitives (Figures 2 and 3) described in Section 5. The goal is to study the impact of each low-level operation in the end-to-end measurements discussed in the preceding sections.

The overhead of obtaining a compact architecture graph using stable hashing is negligible (below one millisecond) and involves sizes in the order of a few kilobytes. This demonstrates the high performance of our metadata management. Compared with the overheads of the rest of the low-level operations, it can be ignored. Therefore, for the rest of this section, we will study the latter.

Specifically, we compare DStore–Opt with DStore–OnDemand in Figure 6. We consider the following low-level operations: on-demand allocations (measure the duration of regular host-side malloc calls in the case of DStore–OnDemand and custom allocations on the staging buffer in the case of DStore–Opt), GPU-host transfers (measure the duration of copying the tensors from the GPU memory to the regular host memory in the case of DStore–OnDemand and to the staging buffer in the case of DStore–Opt), RDMA pinning (measures the duration of pinning the host memory in the case of DStore–OnDemand), and RDMA transfers (measure the duration of RDMA transfers for both approaches).

As expected, the GPU-host and RDMA transfer duration is consistent between the two approaches for both load and store operations. However, DStore–OnDemand exhibits significant overheads for both on-demand allocations and RDMA pinning, while the corresponding overheads are negligible in the case of DStore–Opt. This demonstrates the importance of our custom memory management.

6.6 Results: Different Number of Layers

Our second set of experiments considers the end-to-end performance of a single client that is served by a single provider, in a setup similar to that described in Section 6.4.

The key difference is that we keep the total size of the DL models fixed at 4 GB but vary the number of dense layers of equal size from 100 to 1000. DL models with many layers have much larger metadata and trigger more difficult I/O patterns for approaches that rely on serializing the DL models as files, since it involves many noncontiguous tensors allocated in scattered memory locations that need to be read/written. These models are common in applications that use complex residual networks that can be built with 1000+ layers, such as modern large graph neural network models [20].

Similar to the case of variable model size, we first store the DL model using one client, then read it back using a different client deployed on a different compute node to avoid caching effects. The results are depicted in Figure 5.

As can be observed, the duration of the store operation challenges the SavedModel–PFS approach the most, since it suffers from large metadata overheads, large overheads to collect the scattered tensors, and large I/O overheads due to writing the data and metadata into a large number of files on the parallel file system. This effect is particularly pronounced as the number of layers increases, which is an expected future trend of DL models. By comparison, HDF5–PFS has smaller overheads and scales better. We note, however, that unlike SavedModel–PFS, it does not store the DL model architecture. On the other hand, FlameStore performs much better transfer of many scattered, noncontiguous tensors thanks to its bulk RDMA-enabled I/O. However, it suffers from high metadata overheads compared with DStore–OnDemand. The difference grows slightly higher with increasing number of layers, which demonstrates the effectiveness of our compact architecture graph. Furthermore, the advantages of using a preallocated and prepinned staging buffer are even more pronounced in the case of a large number of layers. In this case, DStore–Opt is more than 2x faster than DStore–OnDemand and 3x faster than FlameStore. As in the case of variable DL model sizes, DStore–Opt is orders of magnitude faster than SavedModel–PFS and up to 10× faster than HDF5–PFS, despite the fact that it also saves the DL model architecture.

The load operations involve proportionally more layers compared with the variable DL model size scenario. An interesting effect is observed when comparing HDF5–PFS with SavedModel–PFS. In the case of SavedModel–PFS, the load throughput is almost double the store throughput, yet the opposite is true for HDF5–PFS, in which case the write throughput is slightly higher. This effect happens because SavedModel–PFS uses a large number of files while HDF5–PFS uses only one, allowing the operating system to buffer writes better. However, just as in the case of variable DL model size, the load operations are much faster and exhibit better scalability for the other three approaches. FlameStore behaves virtually identically for load operations and store operations, which again demonstrates the effectiveness of our compact metadata and staging buffer. For fine-grained tensor access, DStore–Opt is again more than 2x faster than DStore–OnDemand and orders of magnitude faster than HDF5–PFS, which exhibits poor scalability when less than the full DL model is being loaded.

6.7 Results: Different Layer Sizes

Our third set of experiments uses the same setup as described previously, but this time we keep the number of layers and the DL model size fixed, while changing the distribution of the layers sizes. Such an imbalance between the layer sizes is frequently encountered in practice, hence the need to study this aspect separately.

To this end, we configure the generator to randomly create different tensor sizes that can deviate up to 30% from the size used for uniform tensors. We fix the number of layers to 100 and the total DL model size to 4 GB.

As can be observed in Figure 7, the advantage of DStore–Opt against all other approaches remains evident: it is still up to an order of magnitude faster than HDF5–PFS, especially for partial loading of DL models that involve fine-grained reads of the tensors. Furthermore, DStore–Opt is at least 2× faster than FlameStore and an order of magnitude faster than SavedModel. These results are consistent with Figure 5, where we also fix 100 layers and 4 GB total size, but with uniform layer sizes.

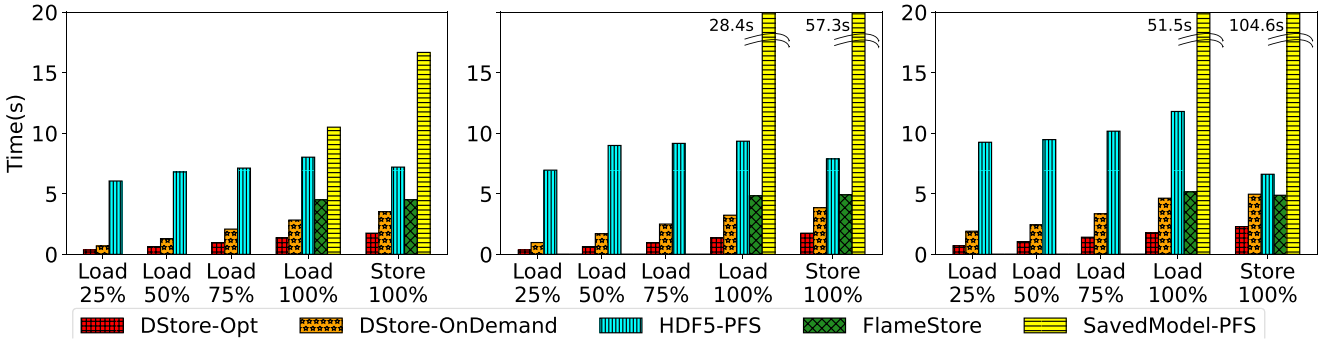


Figure 5: Variable number of layers (100, 500, 1000): single-client partial load (fine-grained tensor reads) and store (writes) of DL models (uniform layer size, 4 GB total size). FlameStore and SavedModel do not support partial loads. Lower is better. Our approach is 2x–45x faster.

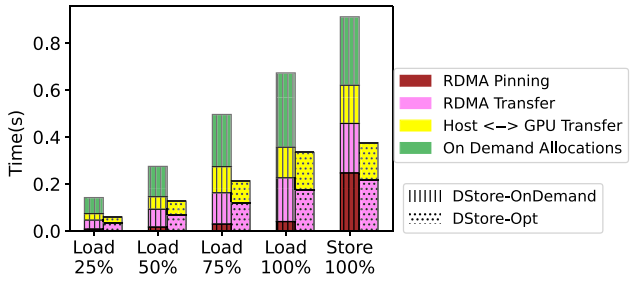


Figure 6: Breakdown of the duration of low-Level operations in DStore–Opt vs DStore–OnDemand when loading/storing a DL model (10 uniform layers, 1 GB total size)

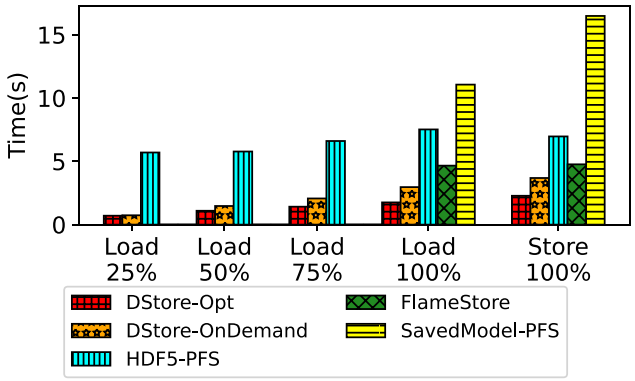
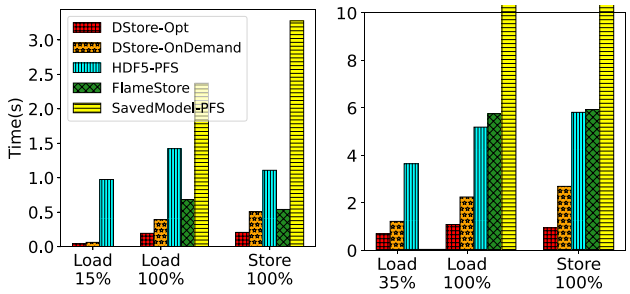


Figure 7: Variable layer sizes: 100 layers, 4 GB total size. FlameStore and SavedModel do not support partial load. Lower is better. Our approach retains high performance despite layer size imbalance (compare with 100 layers of uniform sizes in Figure 5).

The gap between DStore–Opt and DStore–OnDemand slightly decreases compared with Figure 5 especially for partial loading of DL models. In this case, the staging buffer has slightly higher allocation overheads due to the need to manage different buckets corresponding to different size classes. Nevertheless DStore–Opt still retains a large advantage over DStore–OnDemand in most cases.



(a) VGG19 model. Last layer contains 85% of the weights (500 MB)

(b) BERT-large, used to obtain a complex classifier (3 GB).

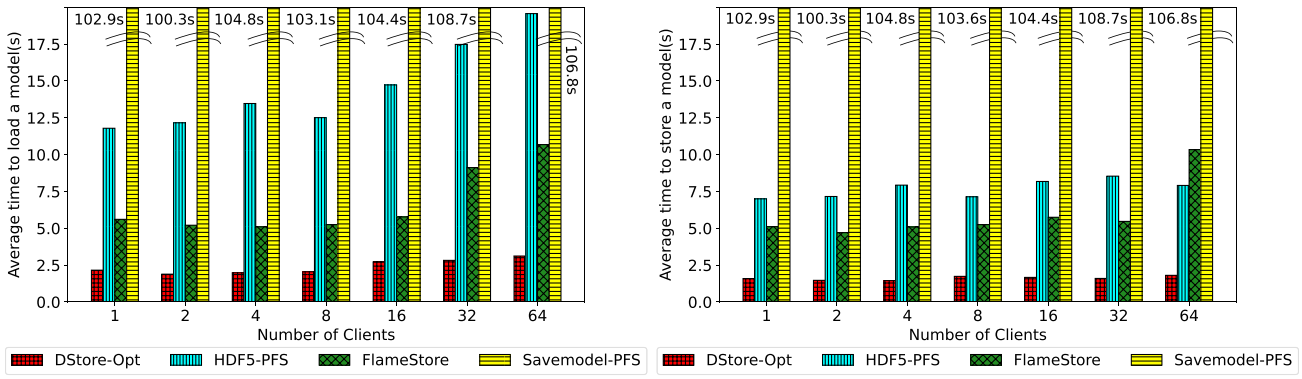
Figure 8: Real-life models: single-client partial load (fine-grained tensor reads) and store (writes). Lower is better. Results are consistent with the synthetic microbenchmarks.

6.8 Results: Real-Life Models

Our fourth set of experiments studies the performance of the load and store operations for two real-life models, VGG19 and a complex classifier based on BERT–large, as described in Section 6.3. We study both full and partial load. In this case, the partial load corresponds to transfer learning that involves fine-grainec accesses of 15% of the weights for VGG19 and 35% of the weights for BERT–large.

As can be observed in Figure 8, similar to the case of the synthetic DL models, the advantage of DStore–Opt against all other approaches is retained for the real-life models. Particularly interesting is the case of partial load for VGG19: in this case, DStore–Opt is two orders of magnitude faster than HDF5–PFS, which underlines the importance of efficient techniques for fine-grained access, especially if the size of the accessed tensors is small. Another interesting observation is that FlameStore performs slightly worse than HDF5–PFS in the case of BERT–large, despite being faster for synthetic DL models and for VGG19. This result shows that increasing DL model complexity poses challenges for the state of the art. On the other hand, DStore–Opt shows much better perfomance levels, increasing the gap not only vs. the state of the art but also vs. DStore–OnDemand.

While we did not have access to massive real-life DL models such as GPT-3 (reported to reach 350 GB [10]), Nvidia Megatron [30](38 GB) and Microsoft's efforts (270 GB) [15]), based on the



(a) Load operations. Lower is better.

(b) Store operations. Lower is better.

Figure 9: Weak scalability: Performance of load and store operations for an increasing number of clients (one per node) synchronized to start simultaneously. Lower is better. DStore–Opt shows perfect scalability and retains its advantage against the other approaches regardless of the number of concurrent clients.

observations above, we believe that the gap between DStore–Opt and the rest of the approaches would significantly increase for such DL models. Nvidia's Megatron model in particular is constructed incrementally via transfer learning [30] from BERT, GPT, and other constituent models, which involves fine-grained reads.

6.9 Results: Scalability of Concurrent Clients

Our last set of experiments focuses on the weak scalability of the load and store operations under concurrency. We deploy an increasingly larger number of clients on separate compute nodes with multiple GPUs, then synchronize the clients to simultaneously start loading or storing their DL models. This experiment underlines the differences between the compared approaches and potential bottlenecks under contention at scale, which, as discussed in Section 1, is increasingly encountered by AI workflows.

In the case of HDF5–PFS and SavedModel–PFS, the parallel file system is subject to concurrent I/O operations, which may lead to I/O bottlenecks. The Lustre parallel file system attached to Cooley has 55 OSTs. Thus, we experiment with up to 64 clients deployed on 64 compute nodes in order to outnumber the OSTs of the parallel file system and trigger I/O bottlenecks. In the case of FlameStore, and DStore–Opt, we co-locate a provider with a client. Furthermore, we enforce a hashing scheme that guarantees that each client will store/load its DL model to/from a remote provider deployed on different compute node. This maximizes the stress on the networking infrastructure and represents a worst-case scenario. The DL model loaded/stored by each client is synthetically generated (4 GB large and includes 10 uniform layers).

The experiment is performed in two stages. First, all clients start at the same time and store their DL model, generating write concurrency to the repository. Then, after all the clients have finished, they start reading (again simultaneously) a different DL model previously stored by a different client (to avoid caching effects), which generates read concurrency to the repository. The results are depicted in Figure 9. As can be observed, in the case of load operations (Figure 9a), FlameStore, HDF5–PFS, and SavedModel–PFS scale well up to 16 concurrent clients, at which point the contention for network bandwidth and, where applicable, the parallel file system drives the average duration higher. On the other hand, thanks to the design principles of our proposal, DStore–Opt shows perfect scalability, which means the gap compared with the other approaches (previously studied for a single client) is only increasing at scale.

A similar trend is visible in the case of concurrent store operations (Figure 9b): FlameStore is bottlenecked beyond 16 concurrent clients, while DStore–Opt continues to scale as it did in the case of load operations. However, in the case of HDF5–PFS and SavedModel–PFS, the I/O bottlenecks of the parallel file system are less pronounced because the OS is flushing the writes asynchronously. Nevertheless, again DStates–Opt retains a speedup of an order of magnitude.

7 CONCLUSIONS

In this work we propose a distributed DL model repository that addresses the needs of HPC AI workflows: high-frequency, concurrent load/store operations with support for fine-grained access to individual tensors. The key novelty of our proposal is a series of design principles (compact architecture graph representation with stable hashing and client-side caching, scalable load balancing on multiple providers, RDMA-optimized data staging, direct access to raw tensor data) that we implemented in the DStore learning model repository, which acts as a drop-in replacement of the standard learning model save/load primitives implemented the Keras library.

Our approach yields considerable speedups over state-of-the-art approaches that store/load learning models as HDF5 files (up to 20×) or use the SavedModel format (up to 45×) on a parallel file system, as well as over approaches such as FlameStore that leverage in-memory storage on the compute nodes (up to 3×). These results apply to both synthetically generated and real-life DL models. Furthermore, weak scalability experiments on up to half the size of the ALCF Cooley HPC system (64 nodes) show perfect scalability for our proposal for an increasing number of nodes, while the gap with respect to the other approaches is widening (up to 60× speedup).

In particular, preallocating and pre-pinning a staging buffer is particularly effective in combination with RDMA and achieves up to 2× speedup compared with the on-demand counterpart.

Encouraged by these preliminary results, we plan to expand the scope of our experiments to demonstrate the impact of our proposal for real-life AI workflows, notably network architecture search scenarios. Furthermore, we plan to explore several other complementary aspects such as (1) persistency using node-local storage; (2) incremental storage and lineage management for transfer learning; and (3) multilevel collaborative caching between the providers based on exploiting the popularity of tensors and the data/metadata relationships.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

REFERENCES

[1] [n. d.]. FlameStore: A Storage System for Deep Learning Models based on Mochi. https://github.com/mochi-hpc/flamestore.
[2] [n. d.]. Lustre: A parallel file system that supports many requirements of leadership class HPC simulation environments. https://www.lustre.org/.
[3] [n. d.]. Model Zoo. https://caffe.berkeleyvision.org/model_zoo.html.
[4] [n. d.]. NVIDIA NGC Catalog. https://catalog.ngc.nvidia.com/models.
[5] [n. d.]. PyTorch Hub. https://pytorch.org/hub/.
[6] [n. d.]. Tensorflow Hub. https://www.tensorflow.org/hub.
[7] 2022. Tensorflow SavedModel. https://www.tensorflow.org/guide/saved_model.
[8] M. Alam, M.D. Samad, L. Vidyaratne, A. Glandon, and K.M. Iftekharuddin. 2020. Survey on Deep Neural Networks in Speech and Vision Systems. *Neurocomputing* 417 (2020), 302–321.
[9] Prasanna Balaprakash, Romain Egele, Misha Salim, Stefan Wild, Venkatram Vishwanath, Fangfang Xia, Tom Brettin, and Rick Stevens. 2019. Scalable Reinforcement-Learning-Based Neural Architecture Search for Cancer Deep Learning Research. In *SC'19: The 2019 International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, CO, 37:1–37:33.
[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.
[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT'19: The 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186.
[12] Ruslan Kuprieiev et. al. 2022. DVC: Data Version Control - Git for Data & Models.
[13] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. 2016. Using Storage Class Memory Efficiently for an In-Memory Database. In *SYSTOR '16: The 9th ACM International Symposium on Systems and Storage Conference*. Haifa, Israel, Article 21.
[14] Antonio Gulli and Sujit Pal. 2017. *Deep Learning with Keras*. Packt Publishing.
[15] Lexie Hagen. 2021. Make Every Feature Binary: A 135B Parameter Sparse Neural Network for Massively Improved Search Relevance.
[16] Nathan Hjelm. 2016. An Evaluation of the One-Sided Performance in Open MPI. In *EuroMPI'16: The 23rd European MPI Users' Group Meeting*. Association for Computing Machinery, Edinburgh, United Kingdom, 184—-187.
[17] Julian Kates-Harbeck, Alexey Svyatkovskiy, and William Tang. 2019. Predicting disruptive instabilities in controlled fusion plasmas through deep learning. *Nature* 568, 7753 (4 2019).
[18] Dmitrii Kochkov, Jamie A. Smith, Ayya Alieva, Qing Wang, Michael P. Brenner, and Stephan Hoyer. 2021. Machine learning-accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences* 118, 21 (2021), e2101784118.
[19] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper R. R. Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. 2020. The Open Images Dataset V4. *Int. J. Comput. Vis.* 128, 7 (2020), 1956–1981.
[20] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. 2021. Training graph neural networks with 1000 layers. In *International conference on machine learning*. PMLR, 6437–6449.
[21] Zhuozhao Li, Ryan Chard, Logan Ward, Kyle Chard, Tyler J. Skluzacek, Yadu Babuji, Anna Woodard, Steven Tuecke, Ben Blaiszik, Michael J. Franklin, and Ian Foster. 2021. DLHub: Simplifying publication, discovery, and use of machine learning models in science. *J. Parallel and Distrib. Comput.* 147 (2021), 64–76.
[22] Hongyuan Liu, Bogdan Nicolae, Sheng Di, Franck Cappello, and Adwait Jog. 2021. Accelerating DNN Architecture Search at Scale Using Selective Weight Transfer. In *CLUSTER'21: The 2021 IEEE International Conference on Cluster Computing*. Portland, OR.
[23] Jie Liu, Bogdan Nicolae, and Dong Li. 2022. Lobster: Load Balance-Aware I/O for Distributed DNN Training. In *ICPP '22: The 51st International Conference on Parallel Processing*. Bordeaux, France.
[24] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. 2016. DAOS and Friends: A Proposal for an Exascale Storage System. In *SC '16: The 2016 International Conference for High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, UT, 50:1–50:12.
[25] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO '87: Conference on the Theory and Applications of Cryptographic Techniques*. 369–378.
[26] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *FAST'21: 19th USENIX Conference on File and Storage Technologies*. 203–216.
[27] Maryam M. Najafabadi, Flavio Villanustre, Taghi M. Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. 2015. Deep learning applications and challenges in big data analytics. *J. Big Data* 2 (2015), 1.
[28] Bogdan Nicolae. 2022. Scalable Multi-Versioning Ordered Key-Value Stores with Persistent Memory Support. In *IPDPS 2022: The 36th IEEE International Parallel and Distributed Processing Symposium*. Lyon, France, 93–103.
[29] Bogdan Nicolae, Jiali Li, Justin Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *CGrid'20: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. Melbourne, Australia, 172–181.
[30] NVIDIA. 2023. Megatron-LM. NVIDIA Corporation.
[31] Keren Ouaknine, Oran Agra, and Zvika Guz. 2017. Optimization of RocksDB for Redis on Flash. In *ICCDA '17: The 2017 International Conference on Compute and Data Analysis*. Association for Computing Machinery, Lakeland, USA, 155—-161.
[32] Ricardo Ribani and Mauricio Marengoni. 2019. A Survey Transfer Learning for Convolutional Neural Networks. In *2019 32nd SIBGRAPI Conference on Graphics, Patterns and Images Tutorials (SIBGRAPI-T)*. 47–57.
[33] Robert B. Ross, George Amvrosiadis, Philip H. Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Robert W. Robey, Dana Robinson, Bradley W. Settlemyer, Galen M. Shipman, Shane Snyder, Jérome Soumagne, and Qing Zheng. 2020. Mochi: Composing Data Services for High-Performance Computing Environments. *J. Comput. Sci. Technol.* 35, 1 (2020), 121–144.
[34] Connor Shorten, Taghi M. Khoshgoftaar, and Borko Furht. 2021. Deep Learning applications for COVID-19. *J. Big Data* 8, 1 (2021), 1–54.
[35] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[36] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. 2018. A Survey on Deep Transfer Learning. In *ICANN'18: 27th International Conference on Artificial Neural Networks and Machine Learning*, Vol. 11141. Rhodes, Greece, 270–279.
[37] The HDF Group. 1997-2022. *Hierarchical Data Format, version 5*. https://www.hdfgroup.org/HDF5/.
[38] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. 2018. Challenges and Solutions for Tracing Storage Systems: A Case Study with Spectrum Scale. *ACM Trans. Storage* 14, 2, Article 18 (2018).
[39] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. 2022. Machine Learning Model Sizes and the Parameter Gap. https://arxiv.org/abs/2207.02852
[40] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. RADOS: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters. In *PDSW '07: The 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*. Reno, Nevada, 35—-44.
[41] Justin Wozniak, Rajeev Jain, Prasanna Balaprakash, et al. 2018. CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research. *BMC Bioinformatics* 19 (2018).
[42] Justin Wozniak, Hyunseung Yoo, Jamaludin Mohd-Yusof, Bogdan Nicolae, Nicholson Collier, Jonathan Ozik, Thomas Brettin, and Rick Stevens. 2020. High-bypass Learning: Automated Detection of Tumor Cells That Significantly Impact Drug Response. In *MLHPC'20: The 2020 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments*.
[43] Yongquan Yang and Haijun Lv. 2021. Discussion of Ensemble Learning under the Era of Deep Learning. *CoRR* abs/2101.08387 (2021).
[44] Chen Youmin, Lu Youyou, Luo Shengmei, and Shu Jiwu. 2019. Survey on RDMA-Based Distributed Storage Systems. *Journal of Computer Research and Development* 56, 2 (2019), 227.