DeepClone: Lightweight State Replication of Deep Learning Models for Data Parallel Training

Bogdan Nicolae, Justin M Wozniak, Matthieu Dorier, Franck Cappello

▶ **To cite this version:**

Bogdan Nicolae, Justin M Wozniak, Matthieu Dorier, Franck Cappello. DeepClone: Lightweight State Replication of Deep Learning Models for Data Parallel Training. CLUSTER'20: The 2020 IEEE International Conference on Cluster Computing, Sep 2020, Kobe, Japan. hal-02914545

HAL Id: hal-02914545
https://hal.science/hal-02914545

Submitted on 11 Aug 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DeepClone: Lightweight State Replication of Deep Learning Models for Data Parallel Training

Bogdan Nicolae*, Justin M. Wozniak*, Matthieu Dorier*, Franck Cappello*
*Argonne National Laboratory, USA
Email: {bnicolae,woz,mdorier,cappello}@anl.gov

***Abstract*—Training modern deep neural network (DNN) models involves complex workflows triggered by model exploration, sensitivity analysis, explainability, etc. A key primitive in this context is the ability to clone a model training instance, i.e. "fork" the training process in a potentially different direction, which enables comparisons of different evolution paths using variations of training data and model parameters. However, in a quest improve the training throughput, a mix of data parallel, model parallel, pipeline parallel and layer-wise parallel approaches are making the problem of cloning highly complex. In this paper, we explore the problem of efficient cloning under such circumstances. To this end, we leverage several properties of data-parallel training and layer-wise parallelism to design DeepClone, a cloning approach based on augmenting the execution graph to gain direct access to tensors, which are then sharded and reconstructed asynchronously in order to minimize runtime overhead, standby duration, readiness duration. Compared with state-of-art approaches, DeepClone shows orders of magnitude improvement for several classes of DNN models.**

***Index Terms*—deep learning; data-parallel training; layer-wise parallelism; model cloning; state replication**

I. Introduction

Deep learning applications are rapidly gaining traction both in industry and scientific computing. A key driver for this trend has been the unprecedented accumulation of big data, which exposes plentiful learning opportunities thanks to its massive size and variety. Unsurprisingly, there has been significant interest to adopt deep learning at very large scale on supercomputing infrastructures in a wide range of scientific areas: fusion energy science, computational fluid dynamics, lattice quantum chromodynamics, virtual drug response prediction, cancer research, etc.

Initially, scientific applications have gradually adopted deep learning more or less in an ad-hoc fashion: searching for the best deep neural network (DNN) model configuration and hyper-parameters through trial-and-error, studying the tolerance to outliers by training with and without certain datasets, etc. Often, the lack of *explainability*, i.e., being able to understand why a DNN model learned certain patterns and what correlations can be made between these patterns and the training datasets was overlooked if the results were satisfactory. While this is acceptable for some industrial applications (e.g., a misclassification of a picture as a dog instead of cat is mostly harmless), scientific applications are often mission-critical and therefore require more rigorous understanding and confidence in the results. However, with increasing complexity of the DNN models and the explosion of the training datasets, ad-hoc methods are not sustainable and limit the applicability of deep learning.

In a quest to solve this challenge, several more systematic approaches are beginning to emerge: guided *model exploration* where configurations and hyper-parameters are automatically identified [1], *sensitivity analysis* [2] that automatically generates perturbations of the training data and the model states to find under what circumstances the model loses accuracy and/or what parts/layers of the DNN model and/or training samples are the most influential the learning process, *ensemble deep learning* [3] that improves the predictive performance of a single model by training multiple models and combining their predictions. All these approaches rely on a common pattern: the need to run many training instances, often by "forking" an initial training instance that has progressed up to a point into many parallel alternatives where slight variations are introduced. We refer to this pattern as *cloning*.

Cloning involves the notion of replicating the DNN model state such that three objectives are simultaneously addressed: (1) introduce as little runtime overhead as possible on the initial training instance; (2) minimize the amount of time necessary to construct the replicated training instance (to avoid wasting core hours; (3) continue training on the replicated instance as early as possible (to finish the work as early as possible).

Checkpoint-restart is one approach to achieve cloning in a straightforward fashion: the DNN model state is checkpointed by the initial training instance and then used by another fresh training instance to restart from it. Despite a large class of optimizations that were proposed for checkpoint-restart, especially in the context of high performance computing (HPC), such an approach is inefficient for the purpose of cloning for two reasons. First, checkpointing is treated separately from restart, which means a suspend-checkpoint-resume-restart cycle is needed to implement cloning. This misses an opportunity to overlap and co-optimize the steps of the cycle. Second, checkpointing is optimized to persist the critical state to durable storage (e.g., a parallel file system), which has limited I/O bandwidth and may cause bottlenecks. In the case of cloning, the critical state is immediately needed for restart, which means the durable storage becomes an unnecessary middle man that introduces extra overhead.

Furthermore, with increasing complexity and sizes of DNN

U.S. Government work not protected by U.S. copyright

models and training data, a mix of data parallel, model parallel, pipeline parallel and layer-wise parallel approaches are emerging to speed-up the training process. In this context, a training instance is not a single process anymore, but an entire group of tightly coupled processes that are distributed across many devices and/or compute nodes of large scale HPC infrastructures. Such groups of processes collaboratively work on a shared DNN model state, exhibiting specific properties and access patterns. Under such circumstances, cloning becomes a highly challenging problem.

To address this challenge, we propose DeepClone, a novel cloning framework specifically designed to bridge the gap between modern approaches for training DNN models (in particular data parallelism and layer-wise parallelism) and HPC infrastructures. We summarize our contributions as follows:

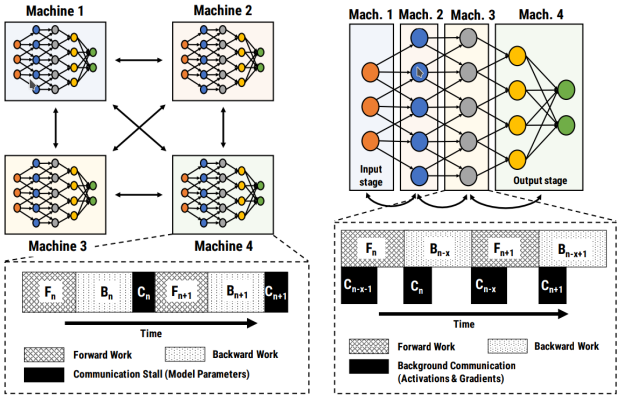
- We introduce several key design principles that underline the main idea of our proposal: low-overhead, zero-copy transfer of the DNN model state, large tensor sharding and reconstruction, asynchronous data transfers overlapped with the back-propagation through augmentation of the execution graph (Section IV).
- We discuss several considerations and algorithms and provide an efficient implementation of the design principles to build DeepClone, the research prototype that demonstrates our proposal (Sections IV-B and V).
- We evaluate our approach in a series of experiments that involve two classes of deep learning applications with several DNN model configurations. Compared with state-of-art approaches, our proposal shows significantly better scalability and orders of magnitude less overhead (Section VI).

II. Background and Problem Formulation

Deep learning (DL) algorithms are a class of machine learning algorithms that are based on complex neural networks with a large number of layers (hence called deep). They have have been successfully applied in a wide range of tasks: image recognition, machine translation, forecasting [4].

DL algorithms primarily use an optimizer such as gradient descent to update the weights: first, the answer to an input is obtained in a forward pass over all layers. Then, in a backward pass, the difference (gradients) between the predicted and actual result ("ground truth") is used to update the weights layer by layer in reverse order. This is repeated iteratively for a large number of training samples until the DNN model has converged. To speed up the training process mini-batches are a common optimization: multiple training samples are used in the forward pass and the resulting average is used for back-propagation.

Gradient descent is computationally expensive. The explosion of available training data and the need to solve more complex problems have led to the introduction of deeper structures with more layers (e.g., complex residual networks that can be built with 1000+ layers, such as ResNet [5]). Therefore, gradient descent is more expensive to run not only because it needs to process more batches, but also because each batch is more expensive to process. To solve this problem, distributed DL algorithms have been developed, capable of scaling horizontally on multiple devices (e.g., multiple GPUs) and/or compute nodes.



(a) Data parallelism: DNN model is replicated, local gradients are averaged.

(b) Pipeline parallelism: DNN model is partitioned and distributed as stages (full layers).

Fig. 1: Data parallelism vs. pipeline parallelism (adapted from [6])

The most widely used technique is *synchronous data-parallel* training. It creates replicas of the DNN model on multiple workers, each of which is placed on a different device and/or compute node. We denote such workers as *ranks*, which is the terminology typically used in high performance computing (HPC). The idea is to train each replica in parallel with a different mini-batch, which can be done in an embarrassingly parallel fashion during the forward pass on all ranks. Then, during back-propagation, the weights are not updated based on the local gradients, but using global average gradients computed across all ranks using all-reduce operations. This effectively results in all ranks learning the same pattern, to which each individual rank has contributed. The process is illustrated in Figure 1a.

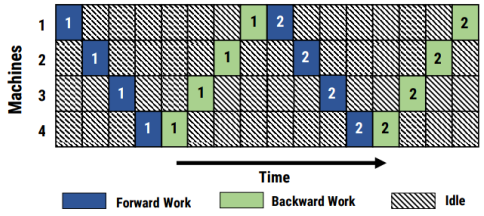


Fig. 2: Model parallelism: DNN model is partitioned and distributed

Model parallelism [7] is another complementary approach (Figure 2). It works by partitioning the DNN model across multiple ranks, each of which is running on a different device and/or compute node. This solves the problem of large DNN models that do to fit in the memory of a rank, but requires data transfers between operations and disallows parallelism within an operation.

Pipeline parallelism [6] combines model parallelism with data parallelism by partitioning the DNN model into stages, each of which comprises one or more layers that are distributed across the ranks (Figure 1b). These stages form a pipeline that can process different batches at different times, both during the forward pass and the back-propagation.

DL algorithms take advantage of multi-core and hybrid architectures (e.g., CPUs + GPUs) to parallelize the gradient computation and weight updates. Specifically, once a rank has finished computing the local gradients for a layer, it immediately proceeds to compute the local gradients of the previous layer. At the same time, it waits for all other ranks to finish computing their local gradients for the same layer, then updates the weights (based on the average gradients obtained using all-reduce in the case of data-parallelism). This is called layer-wise parallelism. An example is depicted in Figure 3 as a DAG (directed acyclic graph): the local gradient of each layer is a dependency for both the previous layer and the rest of the operations (all-reduce and weight updates; for now the reader can ignore shard extraction, which will be explained later). Once the local gradients are computed, both paths in the DAG can be executed in parallel. Over time, several runtimes that implement such ideas have become popular, such as Tensorflow [8], Caffe [9] and Torch [10].

The combination of synchronous data-parallel training and layer-wise parallelism has proven especially popular and many deep learning approaches have introduced support for them: Distributed Tensorflow, Distributed Torch, etc. Some of these runtimes can use MPI as the underlying communication layer that provides an optimized all-reduce implementation, which is a natural fit for supercomputing architectures. A particular implementation, *Horovod* [11], has gained significant traction in production because it can leverage MPI to take advantage of an optimized all-reduce implementation for high-end networking infrastructures, while integrating seamlessly with the Python ecosystem and the high-level machine learning libraries (such as *Keras* [12]) that emphasize ease of use and convenience.

In this context, we formulate the problem of cloning as follows: given N ranks that perform data-parallel training (thus holding each a full replica of the DNN model) and assuming that each rank applies layer-wise parallelism, how can we "fork" the initial N ranks into another set of identical cloned ranks such that both groups can continue training (potentially in a different direction), starting from a given training step, independently of each other.

The challenge in this context is to optimize three objectives simultaneously: (1) minimize the runtime overhead that the initial ranks have to suffer due to transferring the DNN model state to the cloned ranks; (2) minimize the time it takes the clones to receive the DNN model state (to avoid wasting core-hours in an unproductive state where they need to wait without being able to start working); (3) minimize the delay between the moment when the initial ranks and the clones are beginning the next training step (in other words, start the clones as soon as possible so they can finish their work faster).

The combination of layer-wise parallelism and data parallelism introduces significant complexity, but also opens new opportunities for cloning, which we will discuss next. To keep the discussion easy to follow, for the rest of this paper we will focus only on this combination. However, it is important to remember that our proposal can be easily adapted to include pipeline parallelism as well. In this case, the problem of cloning can be formulated as "forking" individual data-parallel stages rather than whole DNN model replicas.

III. Related work

Live migration: is a popular approach to transfer the memory and local storage of a virtual machine (VM) to a standby replica. If the original instance is kept alive, then this is equivalent to cloning. In this context, techniques such as *pre-copy* [13], [14] and *post-copy* [15] can be used. Specifically for the problem of cloning VM images, many alternative approaches are available: copy-on-write (e.g., QCOW2), fork-consistent replication systems based on log-structuring [16], mirroring of I/O operations (both asynchronously [17] and synchronously [18]). Most of these approaches assume a single source and destination for the live migration, with relatively few approaches [19], [20] considering the migration of entire groups of VMs. Related to live migration of VMs is also the notion of container [21] and IoT function migration [22]. Although possible to use as cloning mechanisms, such system-level approaches would incur a high overhead in the context of DNN cloning, because they transfer a much larger state than needed.

Checkpointing: is a complementary direction to cloning, especially if the intermediate states need to be persisted for later reuse. In this regard, *multi-level checkpointing*, as adopted by frameworks such as SCR [23] and FTI [24], is a popular approach that leverages complementary strategies adapted for HPC storage hierarchies. VELOC [25], [4] takes this approach further by introducing asynchronous techniques to apply such complementary strategies in the background. When the checkpoints of different processes have similar content, techniques such as [26], [27] can be applied to complement multi-level checkpointing. However, redundancy is detected on-the-fly, which is an unnecessary overhead in our context (in which the DNN model replicas are known to be identical).

DNN model checkpointing: The problem of checkpointing DNN models efficiently is beginning to emerge in deep learning, where most efforts so far focus on efficient access of training batches [28], [29], [30], [31]. TensorFlow checkpoints model to files in its SavedModel format,[1] or in HDF5 files through Keras.[2] These file-based methods, while simple and adapted for single-node training, are becoming a bottleneck when scaling data-parallel training to a large number of compute nodes. Complex ML workflows that train many models in parallel need the ability to fork models. Yet to our knowledge, all existing production ML workflows rely on checkpoints that are stored persistently into a repository, such as AI Hub[3], TF

[1] https://www.tensorflow.org/guide/saved_model

[2] https://www.tensorflow.org/guide/keras/save_and_serialize

[3] https://cloud.google.com/ai-hub

Hub.[4], or DLHub [32]. Although repositories can be designed to be scalable, the interactions with them incur high I/O overhead.

DNN model cloning: Cloning a model is a basic operation provided by Keras,[5] however this operation is meant for use in different scenarios: to create a copy of an in-memory DNN model (e.g., to enable faster inference) or to replicate a part of a DNN model (e.g., to create a more complex model based on repetitive patterns). Compared with checkpoint-restart, such approaches do not employ persistency techniques and therefore eliminate the overhead of using durable storage as a middle man. However, they are subject to a similar cycle: the training needs to be suspended, the DNN model is cloned, then both the original and cloned instance are resumed. As explained in Section I), this missed opportunities for overlapping and co-optimization. Furthermore, they lack awareness of the distributed nature of the training process.

Relationship to previous work: Previously, we developed *DeepFreeze* [33], an asynchronous checkpointing approach for deep learning models that specifically targets data-parallel training. In this context, we introduced principles such as tensor sharding and augmentation of the execution graph for asynchronous tensor access. In this work we target the problem of cloning, which introduces related principles but with important differences: (1) we complement tensor sharding with the notion of simultaneous reconstruction; (2) we solve a different problem facilitated by the augmentation of the execution graph: zero-copy transfer of tensor shards over the network (as opposed to in-memory copies required for asynchronous checkpointing); (3) we co-optimize such principles for different goals and metrics (in particular, standby and readiness duration).

To summarize, state-of-art techniques that can be used to achieve cloning for data-parallel DNN training are either based on checkpointing (and therefore lack optimizations), or were optimized for other scenarios (cloning of processes, VM instances and images, containers, etc.) and therefore do not take advantage of the specific properties of DNN data-parallel training. To our best knowledge, *we are the first to explore this problem.*

IV. System design

A. Design principles

a) ***Low-overhead, zero-copy transfer of the model state****:* Modern machine learning frameworks are composed of a complex stack of low-level and high-level libraries. Most users never interact with the low-level libraries. For example, *Keras* is a popular high-level Python frontend to Tensorflow. However, this also makes it difficult to access tensors directly, often involving high serialization overheads due to large memory copies (e.g., conversion to *numpy* arrays). Such overheads keep adding up: creating a checkpoint involves serializing high-level Python data structures into files (e.g., using HDF5), which need to be written to a shared storage service such as a parallel file system. Creating a group of cloned processes from a checkpoint incurs the same chain of overheads in revers order: reading the checkpoint (concurrently, which introduces I/O bottlenecks at scale), extracting the model state as *numpy* arrays from it, then initializing the tensors based on these arrays. Even if shortcuts are possible (e.g., send the numpy arrays directly between the original and cloned processes), significant serialization overheads still remain due to the conversion from low-level to high-level tensor representations. To mitigate this issue, we propose to avoid serialization altogether, by passing the raw, low-level tensors directly to high-performance, zero-copy communication libraries (e.g. RDMA, MPI, etc.). Naturally, this raises technical issues regarding how to gain efficient access to the low-level representation of the tensors, as well as correctness considerations, since the tensors need to remain immutable for the entire duration of the direct transfers.

b) ***Large tensor sharding and reconstruction****:* An important goal of cloning is to minimize the runtime overhead it has on an ongoing data-parallel training. Therefore, the more data each initial rank needs to transfer to the clones, the higher the runtime overhead. On the other hand, data-parallel training approaches replicate the model state on each rank. Therefore, it is not necessary for any of the initial ranks to transfer the full model state to the clones. Instead, each tensor can be sliced into a number of shards equal to the number of ranks. Then, each initial rank can send a different shard to the corresponding cloned rank. At a later time, each cloned rank can collect the missing shards from the other cloned ranks. Using this approach, the initial ranks spend less time sending data (which reduces runtime overhead), but the cloned ranks need extra work to recover the full DNN model state. Note that sharding does not imply the need to create additional memory copies: since we have direct access to the raw tensor data, slicing can be achieved simply by adding an offset to a pointer. Also note that we decided to slice each tensor individually rather than the model state as a whole. Although the latter may reduce the number of required data transfers for the same size (which is usually more efficient), it also limits further optimizations, which will be discussed next. Therefore, we have chosen the former sharding approach. In this case, we need to handle the situation of small tensors, for which sharding and reconstruction does not pay off because sending the full tensor is almost as cheap and sending the shard. For the purpose of this work we adopt a simple solution: we simply send the full tensor if it is smaller than a predefined threshold. We also note further possible optimizations, such as aggregating multiple small tensors in a single message.

c) ***Asynchronous transfers by augmentation of the execution graph****:* We leverage the observation that modern DNN frameworks like Tensorflow take advantage of layer-wise parallelism during back-propagation to update the weights of a layer in parallel with the gradient calculations and averaging (which is done using all-reduce operations) for the lower layers. Since layers are typically represented as tensors (multi-

[4]https://www.tensorflow.org/hub

[5]https://www.tensorflow.org/api_docs/python/tf/keras/models/clone_model

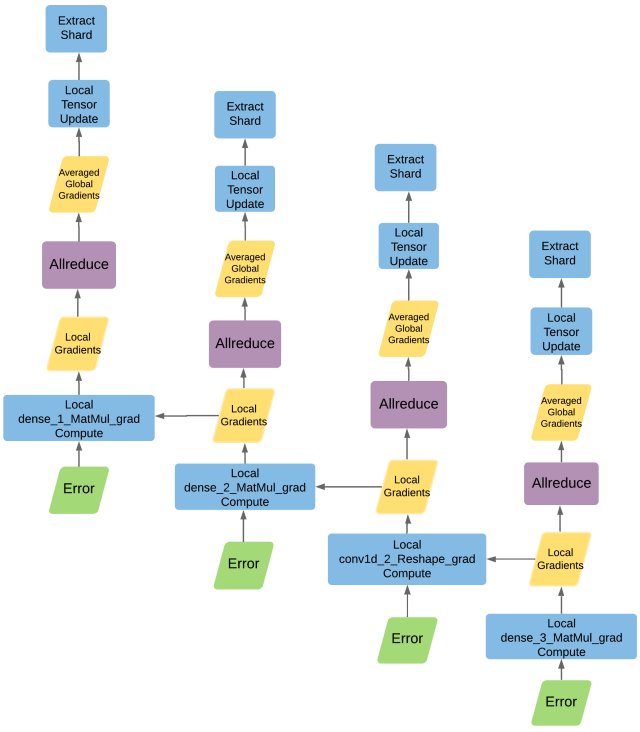


Fig. 3: Example execution graph augmentation: for each layer, the local gradient calculation for the lower layers can proceed in parallel with all-reduce and sharding

.

dimensional arrays), this creates an opportunity to access the tensors immediately after they have been updated, even if the back-propagation has not finished yet. Thus, we propose the idea of augmenting the execution graph with additional tensor operations that initiate a non-blocking data transfer immediately after a tensor has been updated (with or without sharding as needed). An example of how this works is depicted in Figure 3, where each computation of the local gradients activates the previous layer, while in parallel advancing towards the weight updates and data transfers. Only after the back-propagation has finished, it is necessary to wait in case the data transfers have not completed yet. Note that this approach does not raise safety issues during zero-copy transfers, because once updated, a tensor is guaranteed to stay immutable until the next back-propagation (which cannot proceed until all data transfers have finished). Using this approach, we solve two problems simultaneously: first, we further reduce the runtime overhead. Second, we start the cloning as early as possible, overlapping it with the back-propagation. This in turn enables the cloned ranks to start as early as possible, which satisfies our objectives.

B. Zoom on sharding and reconstruction

A key aspect of our proposal is the idea of reducing the data transfer overhead on the initial ranks by sharding and reconstructing the tensors on the cloned ranks. To zoom on this process, we assume the DNN model is composed of an array of tensors t, each of which is characterized by an id, pointer to its content ($data$) and $size$ of $data$. For an initial $rank$, we denote $clone[rank]$ as its cloned counterpart. Similarly, $initial[rank]$ is the counterpart of a cloned $rank$. On the sender side, sharding is a straight-forward approach listed in Algorithm 1.

Algorithm 1 Each tensor update in the execution graph is immediately followed by SEND_TENSOR

```
1:  procedure SEND_TENSOR(t)
2:      if t.size > THRESHOLD then                ▷ needs sharding
3:          s ← t.size/no_ranks
4:          i ← rank * s
5:          async–send(t.id, t.data + i, s) to clone[rank]
6:      else
7:          async–send(t.id, t.data, t.size) to clone[rank]
8:      end if
9:      n + +
10:     if n == no_tensors then
11:         wait–all(async send operations)
12:     end if
13: end procedure
```

However, the same does not hold for the receiver side: the cloned ranks cannot expect to receive the tensor shards in a predefined order, which can happen for various reasons: non-determinism in the execution graph due to parallel directions progressing at different rates on the initial ranks, optimizations in the communication libraries (e.g., aggregation and/or re-ordering of smaller messages), or messages may simply arrive to the cloned ranks in a different order than they were sent.

Some message passing standards (e.g. MPI) enforce strict ordering, but this is still not enough to solve the aforementioned challenge. Therefore, the cloned ranks have no way of knowing in which order the tensors are "completed" (defined by the moment when all cloned ranks received their shard). Without knowing the order of completions, it is not possible to use optimized collective operations such as *all-gather* to reconstruct the tensors in parallel with receiving the shards, because most implementations require all ranks to call all-gather in the same order. Alternatively, each cloned rank could broadcast a shard to the rest of the cloned ranks as soon as it has received it. However, *broadcast* is a non-trivial primitive with its own pitfalls. For example, in MPI it is a collective operation, therefore it can be used only if everybody else is listening.

To overcome these issues, we perform the sharding and reconstruction in two stages: first each cloned rank receives all its shards, then the whole group performs a series of pre-defined all-gather operations. This is illustrated in Algorithm 2.

Although not optimal, this approach has several advantages. First, it is compatible with the MPI standard, respecting the strict ordering requirements for collective *all-gather* operations. Therefore, it can take advantage of optimized implementations for a large variety of platforms. Second, the order of the all-gather phase can be pre-defined to match the order in which the tensors are accessed during the forward pass of the next training step, starting with the lower layers first. Using this approach, the all-gather phase can be overlapped with the

Algorithm 2 Each cloned rank runs RECEIVE_TENSORS before continuing the data-parallel training in a potentially different direction

```
1: procedure RECEIVE_TENSORS
2:     n ← 0
3:     while n < no_tensors do
4:         recv(id) from initial[rank]
5:         t ← tensors[id]
6:         if t.size > THRESHOLD then
7:             s ← t.size/no_ranks
8:             i ← rank ∗ s
9:             recv(t.data + i, s) from initial[rank]
10:        else
11:            recv(t.data, t.size) from initial[rank]
12:        end if
13:        n + +
14:    end while
15:    for t ∈ tensors do                ▷ in access order of forward pass
16:        if t.size > THRESHOLD then
17:            all–gather(t.id, t.data)          ▷ performed in place
18:        end if
19:    end for
20: end procedure
```

forward pass, which enables the cloned ranks to start and finish earlier. Due to additional complexity, we did did not explore this particular optimization in this work, but it is important to note it as a future work opportunity.

Also interesting to note is the general nature of the problem we aim to solve in the specific context of tensor sharding and reconstruction: the need to run the same set of collective operations in different order on each rank and have them progress independently of each other as soon as all ranks are ready to contribute is a recurring pattern in parallel deep learning. For example, *all-reduce* operations needed to average the local gradients may suffer from similar out-of-order behaviors. Approaches such as *Horovod* solve this problem by designating a leading rank to which all other ranks report when they finished computing the local gradients for a layer. The leader then informs all other ranks what all-reduce operation to run next. At scale, this may put a high burden on the leading rank, causing it to become a straggler and slow down the entire group. Therefore, a new flavor of collectives that are tolerant to such out-of-order behavior may be a better solution. We note this opportunity for future research.

V. Implementation

We implemented the checkpointing module on top of Tensorflow 2.0, which includes an optimized version of Keras tightly integrated with it. In this context, we aim for two design goals. First, we expose an API that is compatible with the existing callback mechanism in Keras, which enables users to perform minimal changes to their code in order to use our approach, therefore aligning to the overall design goal of Keras, i.e. provide ease of use and convenience at high level. Second, we isolate the modifications necessary to augment the execution graph into Keras, which means our approach works out of the box with an existing binary distribution of Tensorflow. This is a very important aspect, because many vendors adapt Tensorflow for their machines by integrating it with custom low-level libraries (e.g. Intel MKL), making it challenging if not impossible to modify, recompile and fine-tune Tensorflow.

To achieve these goals, we adopt a strategy similar to our previous work [33]. In terms of API, we provide a Python class that extends the Keras Callback interface and overrides the on_batch_begin and on_batch_end methods. This class can be configured to initiate the cloning at a given training step, ignoring all other training steps. When the desired training step is reached, a boolean tensor flag is set to True by on_batch_begin. This will activate the code responsible to run Algorithm 1. Then, after the training step was completed, it resets the boolean tensor flag to False and returns control to the main loop of model.fit. The user simply needs to invoke model.fit with this class added to the list of callbacks.

In order to augment the execution graph, we intercepted the apply_gradients method of the the base optimizer class of Keras (keras.optimizer_v2.OptimizerV2). This method is responsible for building the execution graph for the weight updates. We injected an additional "if" node after each tensor update to test whether the boolean tensor flag is set, in which case we run a custom Tensorflow operation that captures the pointer to the tensor and forwards it to Algorithm 1. Using this approach, our proposal is compatible with a majority of optimizers that do not customize the apply_gradients method.

To forward the tensor pointers to our algorithms with minimal overhead, we developed TMCI[6] (Tensor and Model Checkpoint Interface). TMCI is a lightweight library written in C++ and Python that simplifies the interaction between Keras and Tensorflow for the purpose of direct access to the raw content of the tensors. To this end, TMCI's underlying implementation consists of two custom save/load Tensorflow operations written in C++.[7] Like any Tensorflow operation, TMCI's operations can be executed on their own, or inserted into the execution graph. When invoked, they forward their arguments to a specified backend. For the purpose of this work, we developed a custom C++ backed that relies on MPI to provide an implementation for save/load based on Algorithm 1 and Algorithm 2. Eventually, we plan to integrate this backend with the VELOC [25] checkpointing system.

VI. Experimental evaluation

A. Experimental Setup

Our experiments were performed on Argonne's *Theta* supercomputer, a 11.69 petaflops pre-Exascale Cray XC40 system based on the second-generation KNL Intel Xeon Phi 7230 SKU. The system has 4392 nodes, each equipped with 64 core processors (256 hardware threads), 16 GB of high-bandwidth MCDRAM (300-450 GB/s), 192 GB of main memory (DDR4 RAM, 20 GB/s), and a 128 GB SSD (700 MB/s). The

[6]https://xgitlab.cels.anl.gov/sds/tmci

[7]https://www.tensorflow.org/guide/create_op

interconnect topology is based on Dragonfly with a total bisection bandwidth of 7.2 TB/sec.

The system was configured to run in caching mode (MCDRAM caches DRAM at hardware level). Durable storage is provided by the Lustre parallel file system (aggregated bandwidth 250 GB/s), which is mounted using POSIX. The node-local SSDs are accessible as a local ext4 mount point. For the purpose of our experiments, they were disabled.

In terms of deep learning software, we use *Horovod* v.0.19.2 and *Tensorflow* v.2.0, which comes with its own optimized implementation of the *Keras* library. Tensorflow was precompiled with optimized KNL support by taking advantage of Intel's Math Kernel Library (MKL) and Intel's own Python distribution.

B. Methodology

To evaluate our proposal, we run a series of experiments that involve an even set of nodes, each of which is running a single MPI rank that consists of a multi-threaded Tensorflow instance (two intra-threads and 128 inter-threads for a total of 256 units of parallelism, the maximum available on the node). This corresponds to the optimal settings for the Theta pre-Exascale machine according to our previous findings [34].

We divide the ranks into two equal groups corresponding to the initial and cloned ranks, each with its own sub-communicator used by Horovod. Halfway through the training process, the initial ranks start the cloning process, after which both groups continue the training process independently of each other.

Throughout our evaluations, we compare three approaches that can be used to achieve cloning:

PFS-HDF5: This approach realizes cloning by means of checkpointing. Specifically, after half of the training steps are completed, rank 0 (chosen by convention, all ranks hold a full replica of the model state) saves the model using model.save_weights. This is done by registering a callback, which intercepts on_batch_end events and counts until half of the training steps have completed. Then, rank 0 sends a message to rank 1 (chosen as the leader of the standby ranks), which releases a barrier all standby ranks are waiting at. This enables all the standby ranks to proceed reading the checkpoint concurrently and continue training independently of the initial ranks.

Mpi4py-P2P: An alternative to checkpointing is a direct transfer of the model state between the ranks in the on_batch_end callback after half of the training steps were completed. To this end, we implement a peer-to-peer approach where each rank i obtains the model weights using model.get_weights, then sends them to rank $i+1$ using *mpi4py*. Unlike the case of PFS–HDF5, the standby ranks are receiving the model state at the same time the initial ranks are sending it. Furthermore, there is no contention because each rank is interacting with a different rank.

DeepClone: This is our proposed approach that hooks directly into the execution graph to capture the content of the tensors immediately after they were updated based on the average gradients during back-propagation. The tensors are sharded and sent to the cloned ranks in a non-blocking fashion, while the cloned ranks reconstruct them later.

These approaches are compared based on three metrics, corresponding to the three objectives introduced in Section II.

Runtime overhead: This metric evaluates the runtime overhead caused by cloning for the initial group. Specifically, it measures how much slower the training step at which cloning occurs and the following one are compared with the average of the rest of the training steps. It is necessary to measure two training steps, because in the case of PFS–HDF5, rank 0 is checkpointing at the end of the training step while the rest of the initial ranks move on. Therefore, rank 0 lags behind in the next iteration, causing a slowdown for the whole group. In the case of Mpi4py–P2P, all initial ranks are sending the model state at the end of the training step. Therefore, they all contribute equally to the slowdown. In the case of DeepClone, there is no overhead at the end of the training step. However, the extra non-blocking send operations embedded into the execution graph (of all initial ranks), although lightweight, may cause interference. This metric is important because it evaluates the direct impact of cloning on the initial ranks, which ideally should be undisturbed by it. Lower values indicate lower impact.

Standby duration: This metric evaluates the time during which the cloned ranks need to be alive to receive the model state before they can continue the training process. In the case of PFS–HDF5, this is the time needed to read the checkpoint concurrently from the PFS on all cloned ranks. The cloned ranks do not need to be alive while rank 0 is writing the checkpoint. In the case of Mpi4py–P2P, this is the time needed by all cloned ranks to receive the model state. Note that unlike the case of PFS–HDF5, the cloned ranks need to be ready to receive the model state as soon as the initial ranks begin sending it, i.e., immediately after the training step when cloning is initiated. In the case of DeepClone, the cloned ranks need to be ready to receive the tensor shards immediately after the back-propagation has started and can progress after they finished reconstructing the full model state from the tensor shards using all-gather. This metric shows for how long the cloned ranks need to be alive but are unproductive, i.e. they consume core-hours without progressing with the training. It needs to be minimized.

Readiness duration: This metric measures how long it takes from the end of the training step for which cloning is initiated until the cloned ranks are ready to continue the training process. In the case of PFS–HDF5, this is the time it takes rank 0 to checkpoint plus the standby duration. In the case of Mpi4py–P2P, this coincides with the standby duration. In the case of DeepClone, this is the delay due to the asynchronous send operations progressing in parallel with the back-propagation, plus any additional delays due to all-gather operations. This metric is an important complement to the standby duration and also needs to be minimized: the sooner the cloned ranks are ready to continue training, the faster they will complete their work.

C. Applications

1) CANDLE NT3: CANDLE [35] (Cancer Distributed Learning Environment) is a project that aims to combine the power of Exascale computing with deep learning to address a series of loosely connected problems in cancer research. Each such problem is driven by a series of benchmarks. One such direction (Pilot 1) aims to predict drug response based on molecular features of tumor cells and drug descriptors. In this context, we study on NT3 [36], which consists of a 1D convolutional network for classifying tissue, expressed as gene sequences, as normal or tumorous. This type of network follows the classic architecture of convolutional models with multiple 1D convolutional layers interleaved with pooling layers followed by final dense layers. The optimizer used by NT3 is SGD (stochastic gradient descent). The training data size for this benchmark is $\approx$ 600 MB, which includes 1120 training samples. NT3 is a representative example of a model with a small number of layers, each of which can grow to huge sizes in the order of GiB.

2) ResNet-50 Family: ResNet is a family of DNN where the layers learn residual functions with reference to the input layers, instead of learning unreferenced functions. This allows ResNet to train extremely deep neural networks with 150+ layers, which was difficult prior to its introduction due to the problem of vanishing gradients [5]. Thanks to this breakthrough, ResNet became a highly popular image classification benchmark. In this paper, we study two variants with 50 and 152 layers, called ResNet-50 and, respectively, ResNet-152. In addition, we also study a related intermediate learning model with 121 layers, DenseNet-121 [37]. All three models have standard implementations shipped with *Keras*. We use the same optimizer as for NT3, i.e., SGD. As training data, we use the ImageNet dataset [38], which is $\approx$ 200 MB large and includes 100,000 samples. The training set of each rank is randomly sampled from the training data. This family of learning models is a representative example of a model with a large number of layers, each of which is small (at most in the order of MiB), which is the exact opposite of NT3 and therefore complements well our study.

D. Results

We use three configurations for the NT3 model by modifying one of the huge layers that can grow in the order of GiB. We refer to these configurations as NT3.A, NT3.B and NT3.C. For the rest of the models, we keep the standard configuration as implemented in *Keras*. The application parameters are listed in Table I.

TABLE I: Model configurations

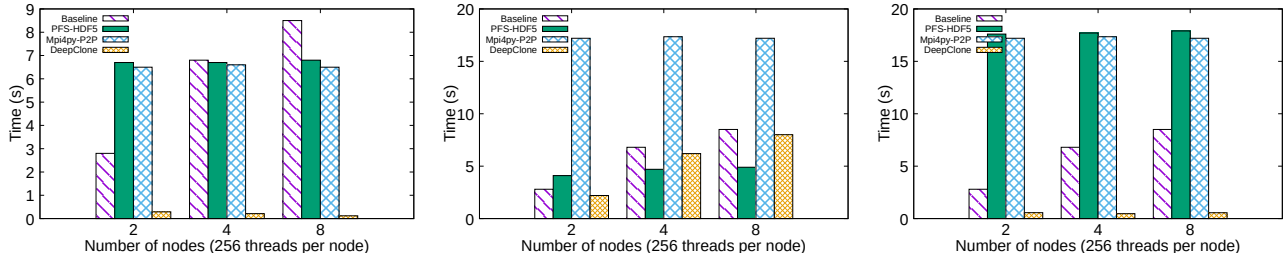
Model	Shorthand	Batch size	Layers	Mutable size
NT3 variant 1	NT3.A	20	10	600 MiB
NT3 variant 2	NT3.B	20	10	1.1 GiB
NT3 variant 3	NT3.C	20	10	1.7 GiB
ResNet-50	R50	32	50	91 MiB
DenseNet-121	D121	32	121	27 MiB
ResNet-152	R152	32	152	223 MiB

For all configurations, we run a total of 20 training steps. The cloning is started by the initial ranks at step 10. Each rank logs the duration of each training step and additional events specific to each of the compared approaches that makes it possible to extract the metrics introduced in Section VI-B. The average duration of each training step under normal circumstances (without cloning) is referred to as the Baseline.

*1) **Scalability for increasing group size**:* Our first series of experiments studies the scalability of the three approaches for an increasing group size. Specifically we experiment with 2, 4, 8 nodes corresponding to 1, 2, 4 initial ranks and an equal number of cloned ranks.

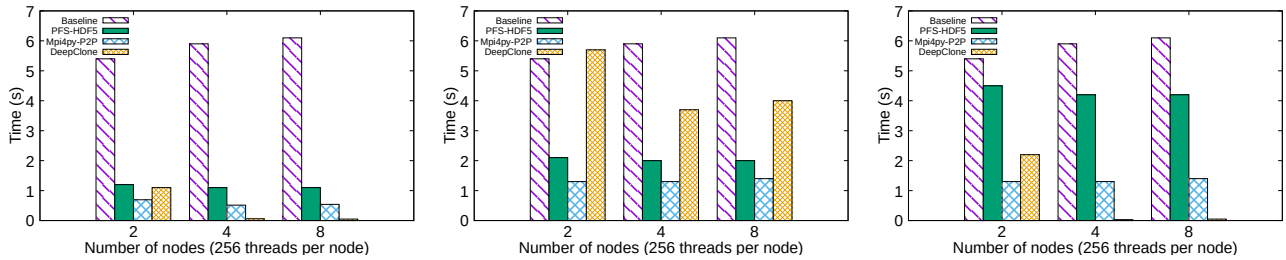
First we study the NT3 benchmark (Figure 4). As can be observed, the runtime overheads for PFS–HDF5 and Mpi4py are close for all configurations. The explanation for this effect is the large serialization overhead incurred by *Keras* to convert tensors from their internal representation into *numpy* arrays. This accounts for the majority of the runtime overhead. A slightly higher overhead is visible for PFS–HDF5, which is expected because writes to HDF5 are usually slower than direct MPI messages. These results are also interesting from a different perspective: despite the fact that PFS–HDF5 only writes a single checkpoint file (and therefore only one rank is lagging behind), a single straggler is enough to slow down the whole group to a comparative level with the case when all ranks experience delays, which is the case of Mpi4py. On the other hand, DeepClone reduces the runtime overhead at least by a factor of 25x, both because it eliminates the serialization overhead (thanks to direct access to the tensors) and because it overlaps the MPI communication with the back-propagation. Also interesting to note is how the runtime overhead decreases for increasing numbers of nodes, which can be attributed to tensor sharding. This underlines the importance of tensor sharding in achieving high scalability, especially when the tensors are large in size.

Regarding standby duration (Figure 4b), large differences can be observed between the three approaches. Surprisingly, in the case of PFS–HDF5, reading the tensors back from a HDF5 file incurs a much lower overhead compared with writing. However, the downside is the fact that all ranks need to read the same file concurrently, which may introduce I/O bottlenecks at scale. This effect is already visible at small scale: the standby duration slightly increases for an increasing number of nodes. This is not the case for Mpi4py, which has a much more stable behavior. However, the standby duration is much higher in this case, which is expected given that the cloned ranks need to be ready to receive as soon as the training step has ended, therefore paying for the runtime overhead too. In the case of DeepClone, the tensors are sent as soon as they are ready during the back-propagation, which results in a standby duration that is at least as long as the back-propagation. For an increasing number of nodes, the scalability of the back-propagation is largely determined by the scalability of the all-reduce operation, as the forward pass is embarrassingly parallel. This effect is also visible for the Baseline, which is higher than the standby duration by a



(a) Runtime overhead (lower is better). (b) Standby duration(lower is better). (c) Readiness duration (lower is better).

Fig. 4: CANDLE-NT3: Scalability of cloning for NT3.C (1.7 GiB model size) for increasing number of nodes.



(a) Runtime overhead (lower is better). (b) Standby duration(lower is better). (c) Readiness duration (lower is better).

Fig. 5: ResNet-50: Scalability of cloning for R50 (97 MiB model size) for increasing number of nodes.

constant. Therefore, we can conclude that the standby duration is almost equal to the duration of the back-propagation, which makes the behavior of DeepClone highly predictable.

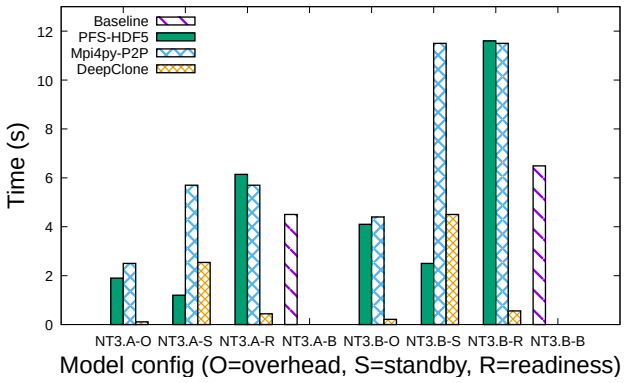
The results above can be well correlated with the readiness duration, depicted in Figure 4c. For PFS–HDF5, the readiness duration includes the time to write the checkpoint and read it back in all cloned ranks, which becomes comparable to the readiness duration of Mpi4py (in which case it overlaps with the standby duration). A slightly higher overhead is visible for PFS–HDF5, for the same reason the runtime overhead is slightly higher: writes to HDF5 are more expensive than direct MPI messages. Recall that the runtime overhead is calculated as the average increase of the two training steps following the training step for which cloning is performed (to account for the effect of stragglers). This is why the readiness duration is not equal to the sum of the runtime overhead and the standby duration. However, it is roughly equal to the double of the runtime overhead plus the standby duration, which makes the readiness duration easy to predict for both PFS–HDF5 and Mpi4py. In the case of DeepClone, we observe a negligible readiness duration that is up to 35x less than in the other two approaches. This underlines the importance of starting the cloning as early as possible. Furthermore, these results also confirm the observation that the standby duration almost overlaps with the back-propagation, which implies that our approach incurs a low all-gather overhead to reconstruct the tensors from shards (as this is the main source of possible delays experienced by the cloned ranks after receiving the tensors).

The results for ResNet–50 are depicted in Figure 5. Unlike the case of NT3, a significant difference is visible for a single initial rank and a single cloned rank (two nodes). In this particular setting, we observe higher overheads for our approach compared with Mpi4py. This outlier case may happen because there is no all-reduce communication happening, therefore embedding a large number of non-blocking send operations of tiny sizes directly into the execution graph has less opportunities to take advantage of synchronization overheads between the layers. To mitigate this case, we will investigate in future work the possibility of embedding a smaller number of send operations that aggregate such small tensors into larger messages.

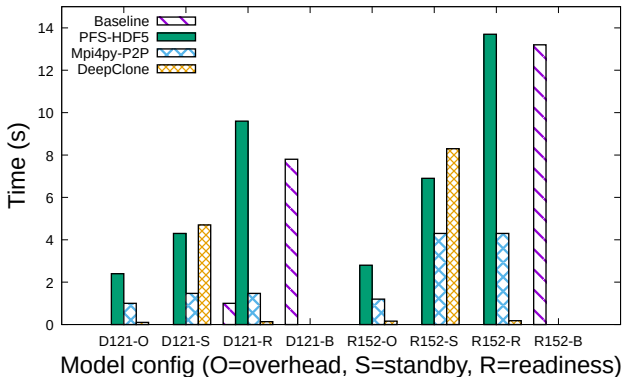
However, as soon as the group size is larger than a single rank, the trend is very similar to the case of NT3. Specifically, both the runtime overhead and the readiness duration of DeepClone are negligible and more than 20x/10x less than PFS–HDF5/Mpi4py. Again, the standby duration of DeepClone almost overlaps with the back-propagation, despite a much higher number of tensors of comparatively much smaller sizes. This underlines the importance of avoiding many all-gather operations for small tensors, which introduces unnecessary overheads.

Another interesting observation can be made when comparing PFS–HDF5 with Mpi4py: unlike the case of NT3, Mpi4py has significantly lower overhead compared with PFS–HDF5, which can be explained by the much lower performance of reading and writing small fields into an HDF5 file compared with sending and receiving small MPI messages.

*2) **Scalability for fixed group size and variable model**:* Our next series of experiments focuses on the scalability of the three approaches for the largest number of nodes (8 nodes, 4 initial ranks, 4 cloned ranks).



(a) CANDLE NT3: runtime overhead, standby duration, readiness duration (lower is better).



(b) ResNet family: runtime overhead, standby duration, readiness duration (lower is better).

Fig. 6: Scalability of cloning for a fixed number of nodes (8) and different model configurations (NT3.C and R50 already depicted previously)

First we study the *NT3* benchmark, depicted in Figure 6a. As expected, with increasing model size, all metrics increase proportionally for all three approaches. Unlike the case of NT3.C, the runtime overhead of PFS–HDF5 is slightly less than for Mpi4py for both NT3.A and NT3.B. However, this does not lead to a lower readiness duration. This is an interesting observation, because it shows that the effect of stragglers is less pronounced when the model is smaller. Also interesting to note is the stability of Mpi4py: when the size of the model doubles, the corresponding runtime overhead, standby duration and readiness duration double too. In the case of DeepClone, we observe a much better scalability: the runtime overhead for NT3.A and NT3.B is between 17x-23x lower compared with PFS–HDF5 and Mpi4py, which grows to 25x for NT3.C. This shows that the gap between our approach and the other two approaches is growing for an increasing model size. A similar trend is visible for the readiness duration: we observe a reduction of 14x-22x compared with PFS–HDF5 and Mpi4py, which grows to 35x in the case of NT3.C. This is also confirmed by the standby duration: as the model size increases, the standby duration as a fraction of the Baseline decreases too.

Next, we study D121 and R152 from the *ResNet* family, as depicted in Figure 6b. Both feature a relatively small size but many more layers and tensors compared with R50. In this case, it is interesting to observe the dramatic increase in runtime overhead for PFS–HDF5 when the number of tensors increases: it is more than double the runtime overhead of Mpi4py, which does not hold for R50, where the runtime overhead was less than double. This confirms the high overhead of reading and writing many small tensors into HDF5 format compared with sending small MPI messages. The HDF5 overhead is also visible when observing the trend exhibited by the standby and readiness duration. On the other hand, DeepClone introduces a much smaller runtime overhead (up to 17x) and readiness duration (up to 75x), confirming that it scales well with an increasing number of layers and tensors, thereby not overburdening the execution graph with many small send operations. In fact, the gap between DeepClone and the other approaches is growing for an increasing number of layers.

VII. Conclusions

In this paper, we studied the problem of efficient cloning of data-parallel training instances that are subject to layer-wise parallelism. This is an important pattern in a wide range of deep learning workflows: model exploration, sensitivity analysis, ensemble searches. To this end, we proposed DeepClone, a cloning framework that introduces innovation at several levels: low-overhead, zero-copy transfer of the DNN model state, large tensor sharding and reconstruction, asynchronous data transfers overlapped with the back-propagation through augmentation of the execution graph. We demonstrated the benefits of DeepClone for several deep learning applications and model configurations that cover a wide spectrum: large and small tensors, few and many layers, large and small mutable model state, etc. Under such circumstances, our approach shows much better performance and scalability, reducing runtime overhead and readiness duration by two orders of magnitude. This underlines the importance of addressing cloning as a dedicated pattern, rather than relying on other approaches like checkpoint-restart to achieve the same result. Encouraged by these results, we plan to explore in future work several aspects we left as open questions, such as the problem of co-optimizing a set of collective communication patterns that may be called out-of-order on different ranks, as well as how cloning can complement checkpointing approaches.

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 and Argonne National Laboratory, under Contract LDRD-1007397.

REFERENCES

[1] P. Balaprakash, R. Egele, M. Salim, S. M. Wild, V. Vishwanath, F. Xia, T. Brettin, and R. Stevens, "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*, 2019, pp. 37:1–37:33.

[2] H. Shu and H. Zhu, "Sensitivity analysis of deep neural networks," in *AAAI'19: The 33rd AAAI Conference of Artificial Intelligence*, 2019, pp. 4943–4950.

[3] O. Sagi and L. Rokach, "Ensemble learning: A survey," *WIREs Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1249, 2018.

[4] S.-M. Tseng, B. Nicolae, G. Bosilca, E. Jeannot, and F. Cappello, "Towards portable online prediction of network utilization using MPI-level monitoring," in *EuroPar'19 : 25th International European Conference on Parallel and Distributed Systems*, Goettingen, Germany, 2019, pp. 1–14.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR'16: 2016 IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, USA, 2016, pp. 770–778.

[6] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *SOSP '19: The 27th ACM Symposium on Operating Systems Principles*, Huntsville, Canada, 2019, p. 1–15.

[7] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS'12: The 25th International Conference on Neural Information Processing Systems*, Lake Tahoe, USA, 2012, p. 1223–1231.

[8] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[9] Y. Jia, E. Shelhamer, J. Donahue *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *ICM'14: The 22Nd ACM International Conference on Multimedia*, Orlando, USA, 2014, pp. 675–678.

[10] "Torch: A scientific computing framework for luajit," http://torch.ch/.

[11] A. Sergeev and M. D. Balso, "Meet Horovod: Uber's open source distributed deep learning framework for tensorflow," https://eng.uber.com/horovod.

[12] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[13] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI'05: Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*, Boston, USA, 2005, pp. 273–286.

[14] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *ATEC '05: Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, USA, 2005, pp. 1–25.

[15] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 14–26, 2009.

[16] J. G. Hansen and E. Jul, "Scalable virtual machine storage using local disks," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 71–79, December 2010.

[17] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, San Diego, USA, 2007, pp. 169–179.

[18] K. Haselhorst, M. Schmidt, R. Schwarzkopf, N. Fallenbeck, and B. Freisleben, "Efficient storage synchronization for live migration in cloud infrastructures," in *PDP '11: Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Ayia Napa, Cyprus, 2011, pp. 511–518.

[19] U. Deshpande, X. Wang, and K. Gopalan, "Live gang migration of virtual machines," in *HPDC '11: Proceedings of the 20th International Symposium on High Performance Distributed Computing*, San Jose, USA, 2011, pp. 135–146.

[20] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu, "Vmflock: virtual machine co-migration for the cloud," in *HPDC '11: Proceedings of the 20th International Symposium on High Performance Distributed Computing*, San Jose, USA, 2011, pp. 159–170.

[21] R. Stoyanov and M. J. Kollingbaum, "Efficient live migration of linux containers," in *ISC'18: The 2018 International Conference on High Performance Computing: Workshops*, Frankfurt, Germany, 2018, pp. 184–193.

[22] P. Karhula, J. Janak, and H. Schulzrinne, "Checkpointing and migration of iot edge functions," in *EdgeSys '19: The 2nd International Workshop on Edge Systems, Analytics and Networking*, Dresden, Germany, 2019, p. 60–65.

[23] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC '10: The 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, USA, 2010, pp. 1:1–1:11.

[24] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *SC '11: The 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, USA, 2011, pp. 32:1–32:32.

[25] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards high performance adaptive asynchronous checkpointing at large scale," in *IPDPS'19: The 2019 IEEE International Parallel and Distributed Processing Symposium*, Rio de Janeiro, Brazil, 2019, pp. 911–920.

[26] B. Nicolae, "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal," in *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, Boston, USA, 2013, pp. 19–28.

[27] ——, "Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead," in *IPDPS '15: 29th IEEE International Parallel and Distributed Processing Symposium*, Hyderabad, India, 2015, pp. 1023–1032.

[28] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient user-level storage disaggregation for deep learning," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–12.

[29] S. Pumma, M. Si, W.-c. Feng, and P. Balaji, "Parallel I/O optimizations for scalable deep learning," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2017, pp. 720–729.

[30] 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," *arXiv preprint arXiv:1809.10799*, 2018.

[31] 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.

[32] R. Chard, L. Ward, Z. Li, Y. Babuji, A. Woodard, S. Tuecke, K. Chard, B. Blaiszik, and I. Foster, "Publishing and serving machine learning models with dlhub," in *PEARC '19: Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, Chicago, USA, 2019.

[33] B. Nicolae, J. Li, J. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "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, 2020.

[34] J. Li, B. Nicolae, J. Wozniak, and G. Bosilca, "Understanding scalability and fine-grain parallelism of synchronous data parallel training," in *5th Workshop on Machine Learning in HPC Environments (in conjunction with SC19)*, 2019.

[35] J. Wozniak, R. Jain, P. Balaprakash *et al.*, "CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, no. 19, 2018.

[36] "CANDLE Benchmarks," https://github.com/ECP-CANDLE/Benchmarks.

[37] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR'17: 2017 IEEE Conference on Computer Vision and Pattern Recognition*, Honolulu, USA, 2017, pp. 2261–2269.

[38] J. Deng, W. Dong, R. Socher *et al.*, "ImageNet: A large-scale hierarchical image database," in *CVPR'09: Conference on Computer Vision and Pattern Recognition*, Miami, USA, 2009, pp. 248–255.