Towards a Flexible and High-Fidelity Approach to Distributed DNN Training Emulation

Banruo Liu*
Tsinghua University

Mubarak Adetunji Ojewale
KAUST

Yuhan Ding
Tsinghua University

Marco Canini
KAUST

ABSTRACT

We propose NeuronaBox, a flexible, user-friendly, and high-fidelity approach to emulate DNN training workloads. We argue that to accurately observe performance, it is possible to execute the training workload on a subset of real nodes and emulate the networked execution environment and the collective communication operations. Initial results from a proof-of-concept implementation show that NeuronaBox replicates the behavior of actual systems with high accuracy, with an error margin of less than 1% between the emulated measurements and the real system.

CCS CONCEPTS

• **Networks** → **Network experimentation**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Distributed Deep Learning Training, Machine Learning Systems, DNN Training Emulation

ACM Reference Format:
Banruo Liu, Mubarak Adetunji Ojewale, Yuhan Ding, and Marco Canini. 2024. Towards a Flexible and High-Fidelity Approach to Distributed DNN Training Emulation. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '24), September 4–5, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678015.3680478

*Work done primarily while author was interning at KAUST.

APSys '24, September 4–5, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-1105-3/24/09.
https://doi.org/10.1145/3678015.3680478

1 INTRODUCTION

Modern DNN training clusters are remarkable engineering feats that more closely resemble high-performance specialized computing environments – and the high costs that these entail – than their mainstream counterparts in commodity cloud computing data centers. Optimizing resource utilization and overall efficiency is paramount to maximizing the performance of training workloads and minimizing associated costs. Therefore, it is highly desirable to explore the large space of potential design considerations, performance optimizations, and configuration tunings, and ideally, to do so without incurring the time, energy, and monetary costs of profiling training workloads at scale on actual hardware.

Conducting in-depth "what if" analyses is essential to making informed decisions and beneficial for various scenarios. For instance, an ML engineer may want to explore for a given model the impact of a particular parallelization strategy[1] on the training time and resource utilization. But it is not practical to profile the training workload on thousands of HW accelerators (GPUs, TPUs, etc.) for each possible strategy and different configurations. Similarly, a researcher may want to quantify the benefits *at scale* of a new optimization technique to improve LLM training efficiency. Also, in this case, it is hardly feasible to run systematic experiments on a large cluster for each possible configuration.

Recent work has shown the potential of simulation and analytical methods to gain insights about DNN training behavior [1, 2, 22–24, 32]. However, these approaches suffer from at least one of three limitations: 1) they require significant effort to transform the actual workloads into an input model for the simulator, 2) they require explicit models of parallelization strategies and incorporating new ones entails non-trivial development of new simulation models, and, 3) the fidelity of their results is limited by how faithful the underlying analytical models of compute and communication are, which are notoriously difficult to get right at scale [16].

This work pioneers and advocates the use of emulation to aid in the analysis and experimentation of distributed

[1]Possible strategies include data parallelism [33], tensor parallelism [26], pipeline parallelism [8], fully sharded data parallelism [20, 35] among others.

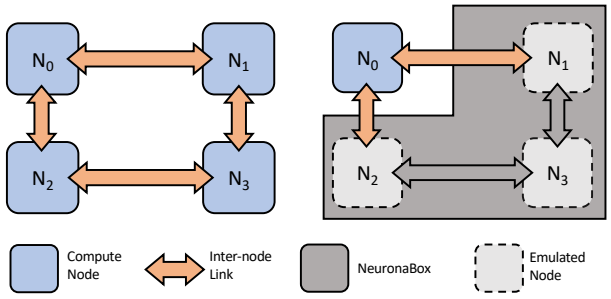

Figure 1: A training job running in a 4-node cluster (left) is emulated by executing a single real node (N_0) wrapped by NeuronaBox, which emulates the environment (right).

DNN training workloads. What we mean by emulation is illustrated in Fig. 1. In a nutshell, we propose to isolate a node subset (denoted as $\mathcal{N}$) of a distributed training job and emulate the networked execution environment (denoted as $\mathcal{E}$) from the perspective of the nodes in $\mathcal{N}$. We elect to view the network as a natural boundary between the real and emulated environments since communication between nodes in distributed training jobs typically occurs through a collective communication library (e.g., NCCL [18]) that both isolates the training scripts from dealing with all the unnecessary details of the underlying network and demarcates clear points for inter-process synchronization. We refer to our approach as NeuronaBox.

Notably, in this approach, the nodes in $\mathcal{N}$ run unmodified training scripts, DNN frameworks and libraries. In particular, the communication is handled by the actual collective communication library over the network fabric. Meanwhile, the emulation environment $\mathcal{E}$ executes on dedicated hardware resources. The requirements for the emulation environment are modest: it doesn't require HW accelerators, it can run on a single CPU-based node, and it requires network bandwidth to match the available aggregate bandwidth of nodes in $\mathcal{N}$.

The key benefit of this approach is that it allows us to faithfully execute on real hardware a portion of the training workload, which executes without overheads from instrumentation (since there is none) nor profiling $\mathcal{N}$ in controlled conditions. Therefore, we can observe the actual behavior of the training job, including the HW utilization metrics and collective communication patterns that are critical in analyzing the performance of distributed training workloads.

We wish to stress that our objective is to enable performance analysis and optimization of distributed training workloads. Implications on model quality are out of scope.

Thus, in this work, we initiate the study of these core research questions: 1) *What aspects of the workload must $\mathcal{E}$ emulate?* 2) *How can this approach maintain high fidelity while retaining wide applicability?*

With this short paper, we aim to provide an initial exploration of the feasibility and potential of this approach, and to solicit feedback from the community on the soundness of our approach and the prospects for future research.

2 PROPOSED APPROACH

We aim to enable any subset $\mathcal{N}$ of nodes in a distributed DNN training job to execute the workload as if it were running on the entire set of nodes and resources. We propose to achieve this goal by emulating the interactions between $\mathcal{N}$ and its networked environment $\mathcal{E}$, which, in a sense, can be viewed as a virtualization of the remaining job's nodes. We argue that, under certain assumptions (detailed below), by observing the performance of $\mathcal{N}$, we can analyze and extrapolate the behavior for an entire job with high fidelity.

In our design, we adhere to two driving principles:

1) Ease of use. Users should be able to use NeuronaBox without modifying their existing code.

2) Flexibility and independence of parallelization strategies. NeuronaBox should target a level of abstraction that is independent of the specific parallelization strategy. NeuronaBox should be flexible to seamlessly adapt to changes in parallelization strategies, including new ones that may emerge in the future.

Workflow and architecture. Fig. 2 depicts an overview of our approach. The high-level workflow of NeuronaBox is as follows. First, the user provides the training script, the job configuration (e.g., world size, nodes in $\mathcal{N}$, HW resources, etc.), and optionally a set of what-if conditions for experimentation (an example is given later). Second, NeuronaBox initializes the emulation environment by synthesizing the network topology and instantiating a communication model that calculates delay times for collective operations within the emulated environment. Third, the training script is launched (e.g., via torchrun). Meanwhile, desired performance metrics like iteration time and resource utilization are gathered in $\mathcal{N}$. Traces of collective communication (e.g., NCCL traces) can also be collected.

Assumptions. We assume that nodes have uniform hardware and network configuration. In practice, it is common to execute distributed training jobs on homogeneous clusters [11, 14, 29, 31, 36]. We assume that the model fits entirely within $\mathcal{N}$. This assumption is not restrictive, as it is common to use model or tensor parallelism within a node or a shard [9, 26] . We expect that these assumptions yield a sort of symmetry in the workload distribution across the nodes, which allows us to treat the nodes in $\mathcal{N}$ as a representative sample of the entire nodes. We discuss how to extend our arguments to a non-uniform scenario in § 2.3. We assume that the network bandwidth of $\mathcal{E}$ matches the aggregate bandwidth of nodes in $\mathcal{N}$.

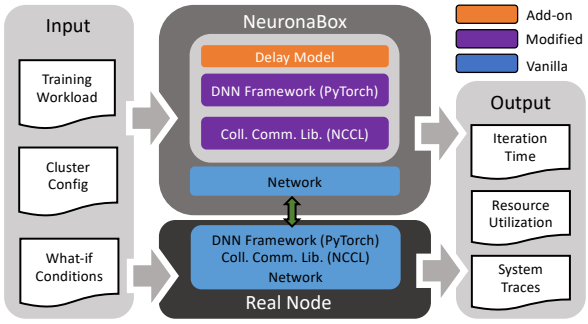

Figure 2: Overall workflow and architecture of NeuronaBox.

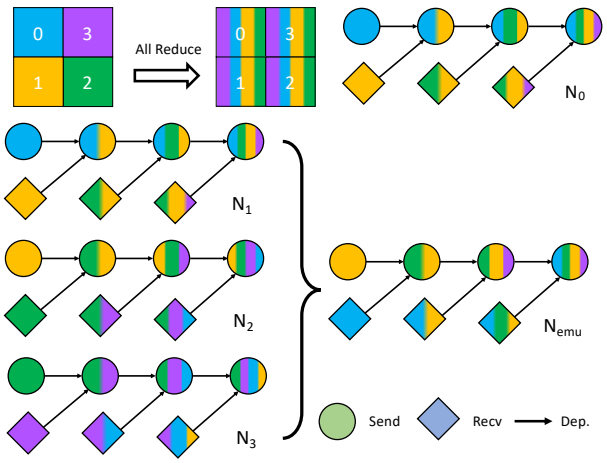

Figure 3: An example DAG for four-node ring all-reduce. The upper left squares show the net result of all-reduce, where color-coded data from different nodes are reduced and then gathered at each node. The upper right figure shows the dependency DAG for $\mathcal{N}$ (N_0). The lower figure shows how we merge the dependency DAG of N_1, N_2, N_3 into $\mathcal{E}$. The cross-node dependencies from $send(x)$ to $recv(x)$ are not shown for clarity sake. We only show the initial 4 steps of all-reduce for simplicity.

Further, we assume that the collective communication layer is the only point of interaction between $\mathcal{N}$ and $\mathcal{E}$. This assumption is reasonable, as the collective communication layer is the primary interface between the computation and the network stack in distributed training jobs. Finally, note that we are free to modify the DNN framework and collective communication libraries within the emulator. That is how we are able to implement NeuronaBox.

Scalability. While collective communication is a natural layer to target in our work, the astute reader may now wonder how scalable this approach is. Scalability is traditionally a challenge in network emulators [12, 30], as emulating a large number of nodes could overload the emulator. Our key insight is that we are only interested in the interaction between $\mathcal{N}$ and the outside world. And so, the actual communication between the emulated nodes can be skipped. Instead, only the delay resulting from these communication operations needs to be incorporated into the emulation. As a result, the number of connections as well as the amount of data transfer for NeuronaBox is the same as that of $\mathcal{N}$. This observation allows NeuronaBox to potentially scale to a large number of nodes. A complete exploration of the scalability of NeuronaBox is left for future work.

In the remainder of this section, we detail the design and implementation of our proposed approach, NeuronaBox. Since we aim to demonstrate feasibility through a proof-of-concept, we focus on a single real node, denoted by N_0. That is, $\mathcal{N} = \{N_0\}$.

2.1 Initialization

Before we describe how NeuronaBox behaves during training, we first need to initialize the emulator. This requires setting up the environment, including the network topology and the communication model.

Topology detection. This step involves establishing the connection between any pair of nodes and finding the optimal data paths between all node pairs. It is normally bootstrapped by the collective communication library itself. During this process, each node builds up its own local topology graph (how HW accelerators are connected via NVLink, PCIe, and NICs), then exchanges its local graph and, together with other nodes, builds the global graph. In NeuronaBox, the emulator fakes the local graph of emulated nodes based on the job configuration input. Then, it emulates multiple endpoints (the virtualized job nodes) so that nodes in $\mathcal{N}$ can communicate with them (directly via RDMA).

Delay calculation. Based on the global topology, NeuronaBox calculates the communication and computation delays within the emulated environment. In our design, we provide an interface so that the calculation itself is done within a user-defined add-on plugin. For example, the delay in a ring all-reduce call can be estimated simply by using the classical all-reduce delay model [28] or by using a packet-level simulator. In the future, we seek to leverage the network simulation components of DNN simulators [1, 2, 32].

2.2 Emulation in a Uniform Scenario

In general, a collective operation (e.g., ring all-reduce) can be split into a number of messages with dependencies. The

emulator must send/receive messages in a way that takes into account both the dependency and the internal protocol of the collective operations library (in our case, NCCL). We first describe how we treat a single collective operation and then generalize to multiple asynchronous collective operations.

Single collective operation. Recall that we only need to consider the interaction between $\mathcal{N}$ and $\mathcal{E}$. This means that we can omit the dependencies for communication within $\mathcal{E}$. The workflow of a collective operation can be represented as a directed acyclic graph (DAG) where vertices are send or recv tasks and edges are the data dependencies. Thus, we note that in NeuronaBox, this DAG can be greatly simplified. Fig. 3 shows an example for ring all-reduce.[2] It is worth noting that the DAGs for $\mathcal{N}$ and NeuronaBox are isomorphic. As a result, with every message received from $\mathcal{N}$ to $\mathcal{E}$, we can always determine the correct state in order to generate the next message of the collective operation workflow.

To achieve this, $\mathcal{E}$ maintains a bitmap of the messages that have been sent to or received from $\mathcal{N}$, and it applies the following two actions ($\mathcal{E}$ polls these using background threads) that advance the state of the DAG ensuring synchronization correctness:

1) *Try Send To* $\mathcal{N}$. A message can be sent if and only if all its predecessors have been sent or received in the DAG. If the next unsent message in the bitmap fulfills this condition, we update the bitmap and send the message.

2) *Try Receive From* $\mathcal{N}$. Upon receiving a message from $\mathcal{N}$, $\mathcal{E}$ checks whether it is the expected message for the current operation. If this is the case, the bitmap of the record is updated; otherwise, an error is reported.

Multiple async collective operations. The design of a single collective operation can easily be extended to support multiple operations by assigning a unique ID to each operation and maintaining its information in a logically centralized controller. We record the mapping of operations to their message bitmaps. We ensure fairness between the individual streams through round-robin polling. Note that the method for a single operation is asynchronous by nature, as the functions are polled by background threads. Given that, the synchronization is achieved by busy-waiting.

2.3 Extension to a Non-uniform Scenario

The workload of individual nodes in a training job may not always be balanced. This is the case, for example, when model parallelism fails to achieve a balanced workload distribution or when hardware and topology heterogeneity exists. Consequently, emulating the behavior of an arbitrary node subset ($\mathcal{N}$) may not adequately represent the behavior of the entire workload. To overcome this challenge, we propose classifying each node in the job based on the part of the model it contains, e.g., having different stages in model parallelism. We then ensure that one node in each class is in $\mathcal{N}$. This approach allows us to infer the behavior of the workload by observing the collective behavior of each class of nodes. This approach also solves the heterogeneity of hardware and topology when we classify nodes with different hardware into different classes.

However, we note that this approach requires more resources. Suppose there are m classes; in the basic solution, we need to have one representative real node for each of them. And with t hardware types, we then need tm nodes in $\mathcal{N}$. We conjecture that it may be possible to reduce the number of nodes in $\mathcal{N}$, say to k, by decoupling the emulation between different nodes. Assuming nodes in $\mathcal{N}$ are in different classes, if all the communication is interposed by $\mathcal{E}$, then we can time-multiplex the class-based workload and assign each node $\frac{m}{k}$ amount of load. We leave the exploration of these techniques for future work.

Size	AllreduceE	AllreduceB	AllgatherE	AllgatherB
1KB	435.2μs	418.8μs	282.6μs	276.2μs
4KB	526.5μs	511.0μs	306.5μs	300.2μs
32KB	564.9μs	552.3μs	329.0μs	322.6μs
256KB	1326.0μs	1314.0μs	868.9μs	859.6μs
2MB	7661μs	7655μs	4928μs	4929μs
16MB	59.0ms	58.9ms	37.5ms	37.5ms
128MB	470ms	469ms	298ms	298ms
1GB	3760ms	3760ms	2408ms	2407ms

Table 1: Average run time per call. B for baseline and E for emulator (NeuronaBox). NeuronaBox only incurs at most 4%/2% extra time for all-reduce and all-gather, respectively.

2.4 Proof-of-concept Implementation

Our proof-of-concept implementation entails the development of an end-to-end system using the PyTorch DNN framework and NCCL as the collective communication library, chosen because of their popularity.

Our implementation is able to run a two-node training using a distributed data-parallel strategy and fully sharded data parallelism (FSDP) [20]. In particular, we modify NCCL's instance in $\mathcal{E}$ to emulate the collective operations as per § 2.2. Moreover, in $\mathcal{E}$, we skip the `cudaKernelLaunch` completely so that no GPU computations are involved. We also modify the NCCL proxy so that it sends dummy data in compliance with the internal protocol so that the workload continues to run and $\mathcal{N}$ is not aware of the emulation. In PyTorch, we mainly alter the `autogard` and `c10d` to implement the synchronization that previously relies on a CUDA kernel now compatible with the emulator. We also remove computation

[2]The all-reduce operation performs reduction on data (i.e., sums) across nodes and stores the result in a buffer at every node. We use different color for data in different worker.

in backward pass and model weight update in $\mathcal{E}$, given that those computations are redundant in emulation.

The whole system is about 2000 LoC in CUDA C++ and 50 LoC in Python, exclusive of experiments and tests. We plan to release NeuronaBox as open source.

3 PRELIMINARY EXPERIMENTS

We evaluate our prototype NeuronaBox implementation by (1) running microbenchmarks at the NCCL level to see the pure performance of collective communication, (2) running an end-to-end system in PyTorch to know the accuracy of emulation, as well as measuring CPU utilization to evaluate the overheads of NeuronaBox. We also demonstrate (3) an application scenario of NeuronaBox by performing a "what-if" analysis with latency variations.

Testbed. Our test environment consists of two nodes, each equipped with two 8-core Intel Xeon Silver 4112 CPUs running at 2.60 GHz, 512 GB RAM, and is fitted with a 100 GbE Mellanox ConnectX-5 NIC. In addition, each node contains two NVIDIA V100 GPUs, although only one is used during evaluation. Each node runs Ubuntu 22.04 (Linux kernel 5.15.0), CUDA 12.2, PyTorch 2.2.0a0 and NCCL 2.19.4.

If not otherwise stated, we call the two nodes $\mathcal{N} = \{N_0\}$ and $\mathcal{E} = \{N_{emu}\}$. N_0 always runs the unmodified code. N_{emu} is configured to run either NeuronaBox's modified code(as emulator), or unmodified code (as a baseline).

3.1 Microbenchmark

Setup. First, we assess NeuronaBox's capability to emulate collective communication operations. We devise the benchmark by generating input data tensors of different sizes on GPU and issuing two-node collective operations. We test ncclAllreduce and ncclAllgather. After the warm-up, we measure the time taken over at least 100 repetitions for each call on N_{emu} and report the average. N_0 always runs unmodified code. We compare the result when N_{emu} is running unmodified NCCL (baseline) and NeuronaBox's NCCL (emulator).

Results. Table 1 shows that NeuronaBox only incurs at most 4% overhead; we attribute this to the mutex lock on the controller and bitmap bookkeeping. The overhead diminishes as the size increases; when the size is greater than 2MB, the gap is no more than 1%. Since most data-parallel implementations use buckets to batch all-reduce calls to a larger size (e.g., 25MB in NVIDIA APEX [17]), we believe this NCCL-level overhead of NeuronaBox is acceptable.

3.2 End-to-end Training Emulation

Setup. To evaluate NeuronaBox's ability to accurately emulate end-to-end DNN training, we conducted experiments using three real-world DNN models, including computer vision, natural language processing, and recommendation systems. The models' details are listed in Table 2. We use PyTorch's DistributedDataParallel module for data parallelism. N_0 runs unmodified PyTorch; N_{emu} runs NeuronaBox's PyTorch as an emulator and unmodified PyTorch as a baseline. To further illustrate the flexibility of NeuronaBox, we also train a T5 model using PyTorch's FullyShardedDataParallel module, which mainly uses the all-gather and reduce-scatter collectives. Note that no additional code changes are needed to support FSDP.

Model	Task	Dataset	Parallism
BERT [5]	Question Answering	SQuAD [21]	DP
ResNet152 [6]	Image Classification	ImageNet-1K [25]	DP
DeepLight [4]	CTR Prediction	Criteo 1TB [15]	DP
T5-base [19]	Language Modeling	Wikihow [10]	FSDP

Table 2: Characteristics of benchmark models.

Model	Time-E	Time-B	CPU-E	CPU-B
BERT	629 ± 3.0	628±1.1	12.93%	14.25%
ResNet152	1061±19.8	1063±16.3	12.68%	12.95%
DeepLight	727±15.0	726±13.8	7.52%	7.75%
T5-base	4174±4.4	4175±0.7	10.52%	13.32%

Table 3: End-to-end workload comparison. 'E' and 'B' stand for emulator-enabled (NeuronaBox) and the baseline, respectively. 'Time' stands for the training times in milliseconds; and 'CPU' stands for the percentage of CPU usage in a node.

We measure metrics of interest detailed below over 300 training iterations and report their averages after 100 warm-up iterations. We measure training time on N_0. For BERT, T5, and ResNet, we report training time per iteration (batch). For DeepLight, we report training time per epoch because it involves model pruning and has variance between iterations. Additionally, we measure the CPU usage on N_{emu} to illustrate the overhead of the emulation.

Results. As shown in Table 3, NeuronaBox is quite accurate in a two-node environment training when compared to the baseline, and achieves an error smaller than 1%. The CPU usage during emulation in all scenarios is modest and smaller than what observed in the baseline case. We attribute that to (1) the efficient and lightweight implementation of NeuronaBox, which keeps the overhead generally low; (2) the removal of computation in the backward pass, eliminating a lot of memory allocations and data movements. And so, the net effect is a drop in CPU usage. This is promising in terms of the potential scalability of NeuronaBox.

3.3 What-if Analysis: Latency

Setup. In this experiment, N_0 runs unmodified code and N_{emu} runs NeuronaBox. We inject additional delay in each

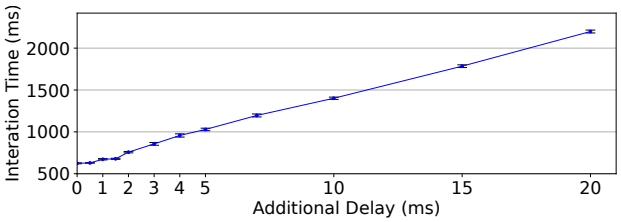

Figure 4: The end-to-end training time per iteration in BERT model (ms) vs the additional delay injected in every all reduce call (ms). The error bar is plotted in black.

all-reduce call in N_{emu} and train a BERT model. We measure the emulated training time per iteration for each delay.

Results. As Fig. 4 shows, the iteration time increases linearly with the delay larger than 2 milliseconds. However, when the delay is small, the overheads reflected in the end-to-end performance do not grow linearly. We consider this "smooth slope" as a result of the computation-communication overlap during the training. The delay injected in each all-reduce call is partly shadowed by the async computation in the backward pass. Such observation implies a possible space of improvement in the training, as there is a gap of 2ms for more communication to happen. This is an example of the kind of insights that can be obtained from using NeuronaBox.

4 RELATED WORK

DDL simulation. A number of simulators have been developed to study the behavior of DNN clusters, including DeepFlow[1], Astra-sim [22, 32], vTrain [2] and several others [7, 23, 24, 27]. These simulators use analytical methods combined with profiling results to make predictions, suffering from limitations mentioned in § 1.

Network emulation. Emulation has been widely adopted in networking research [12, 30]. MimicNet [34] is a machine learning-based network emulator. It exercises a similar idea by dividing the data center into an "observable" cluster ($\mathcal{N}$ in our work) and a black box ($\mathcal{E}$ in our work), and it applies a machine learning model to fit it. However, MimicNet focuses on how to train a model to better approximate the data center network at scale, whereas our work focuses on emulating end-to-end DNN training behaviors.

Goodput prediction. Currently, NeuronaBox only sends *dummy* data to $\mathcal{N}$ during emulation since it only predicts the completion time for each training iteration. However, lossy training optimization techniques like compression and quantization [13, 31] require goodput (accuracy) to be taken into account. To support that, NeuronaBox needs to communicate *meaningful* data to $\mathcal{N}$ without incurring much overhead. One possible solution is to use a proxy model [3] that generates data with a similar distribution to the dataset and intermediate results.

5 CONCLUSION

We proposed a novel approach for estimating time-per-iteration in distributed DNN training, focusing on executing only a part of the model along with collective communication operations. To substantiate our proposal, we designed the NeuronaBox emulator and implement a proof-of-concept system. Through extensive experimentation, we demonstrated in a two-node setup that NeuronaBox achieves high accuracy in predicting training time across a variety of DNN models, with an error margin of less than 1% compared to actual training runs. Finally, we encourage further research in this direction, recognizing that many questions remain to be explored.

ACKNOWLEDGMENTS

This publication is based upon work supported by the King Abdullah University of Science and Technology (KAUST) Office of Research Administration (ORA) under Award No. ORA-CRG2020-4382.

REFERENCES

[1] Newsha Ardalani, Saptadeep Pal, and Puneet Gupta. 2024. DeepFlow: A Cross-Stack Pathfinding Framework for Distributed AI Systems. *ACM Trans. Des. Autom. Electron. Syst.* 29, 2 (2024).

[2] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. 2023. vTrain: A Simulation Framework for Evaluating Cost-effective and Compute-optimal Large Language Model Training. (2023). arXiv:cs.LG/2312.12391

[3] Cody Coleman, Christopher Yeh, Stephen Mussmann, Baharan Mirzasoleiman, Peter Bailis, Percy Liang, Jure Leskovec, and Matei Zaharia. 2020. Selection via Proxy: Efficient Data Selection for Deep Learning. In *ICLR*.

[4] Wei Deng, Junwei Pan, Tian Zhou, Deguang Kong, Aaron Flores, and Guang Lin. 2021. DeepLight: Deep Lightweight Feature Interactions for Accelerating CTR Predictions in Ad Serving. In *WSDM*.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[7] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. 2017. Deep Learning Scaling is Predictable, Empirically. (2017). arXiv:cs.LG/1712.00409

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*.

[9] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, Joe Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. 2023. Tutel: Adaptive Mixture-of-Experts at Scale. In *MLSys*.

[10] Mahnaz Koupaee and William Yang Wang. 2018. WikiHow: A Large Scale Text Summarization Dataset. (2018). arXiv:cs.CL/1810.09305 https://arxiv.org/abs/1810.09305

[11] Fan Lai, Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, Zeliang Chen, Ellie Wen, Paul Rivera, Jie You, Chun cheng Jason Chen, and Mosharaf Chowdhury. 2023. AdaEmbed: Adaptive Embedding for Large-Scale Recommendation Models. In *OSDI*.

[12] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *HotNets*.

[13] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *ICLR*.

[14] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. (2019). arXiv:cs.CL/1907.11692

[15] Microsoft. 2015. Criteo's 1TB Click Prediction Dataset. (2015). https://docs.microsoft.com/en-us/archive/blogs/machinelearning/now-available-on-azure-ml-criteos-1tb-click-prediction-dataset.

[16] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *SIGCOMM*.

[17] NVIDIA. 2024. A PyTorch Extension: Tools for easy mixed precision and distributed training in Pytorch . (2024). https://github.com/NVIDIA/apex.

[18] NVIDIA. 2024. Collective Communication Library (NCCL). (2024). https://developer.nvidia.com/nccl.

[19] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. (2023). arXiv:cs.LG/1910.10683 https://arxiv.org/abs/1910.10683

[20] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *SC*.

[21] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know What You Don't Know: Unanswerable Questions for SQuAD. In *ACL*.

[22] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *ISPASS*.

[23] Saeed Rashidi, William Won, Sudarshan Srinivasan, Srinivas Sridharan, and Tushar Krishna. 2022. Themis: A Network Bandwidth-Aware Collective Scheduling Policy for Distributed Training of DL Models. In *ISCA*.

[24] Wilfredo J. Robinson M., Flavio Esposito, and Maria A. Zuluaga. 2022. DTS: A Simulator to Estimate the Training Time of Distributed Deep Neural Networks. In *MASCOTS*.

[25] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015).

[26] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. (2020). arXiv:cs.CL/1909.08053

[27] Aleksandar Stanić, Dylan Ashley, Oleg Serikov, Louis Kirsch, Francesco Faccio, Jürgen Schmidhuber, Thomas Hofmann, and Imanol Schlag. 2023. The Languini Kitchen: Enabling Language Modelling Research at Different Scales of Compute. (2023). arXiv:cs.LG/2309.11197

[28] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.* 19, 1 (2005).

[29] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. (2023). arXiv:cs.CL/2302.13971

[30] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. 2002. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*.

[31] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2024. ZeRO++: Extremely Efficient Collective Communication for Large Model Training. In *ICLR*.

[32] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale. In *ISPASS*.

[33] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data* 1, 2 (2015).

[34] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. 2021. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *SIGCOMM*.

[35] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (2023).

[36] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *OSDI*.