PerfEstimator: A Generic and Extensible Performance Estimator for Data Parallel DNN Training

Chengru Yang*, Zhehao Li*, Chaoyi Ruan*, Guanbin Xu*, Cheng Li*, Ruichuan Chen†, Feng Yan‡
*University of Science and Technology of China, Hefei, China
†Nokia Bell Labs, Stuttgart, Germany ‡University of Nevada, Reno, USA

2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence) | 978-1-6654-4563-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/CLOUDINTELLIGENCE52565.2021.00012

***Abstract*—Understanding the performance of data parallel DNN training at large-scale is crucial for supporting efficient DNN cloud deployment as well as facilitating the design and optimization of scalable DNN systems. Existing works adopt analytical modeling, which may fall short in capturing the system behaviors resulting from the fast evolving DNN systems and constantly proposed optimizations. In this paper, we present `PerfEstimator`, a generic and extensible estimator for accurate performance estimation of large-scale data parallel DNN training. `PerfEstimator` is driven by three major components, namely, an extensible attributed graph based performance model, a computation and synchronization profiling and simulating tool for obtaining runtime time costs on a single machine, and a computation-synchronization pipeline builder to derive the scaling factors. Our evaluation highlights that `PerfEstimator` can accurately predict the performance of data parallel DNN training jobs with a prediction error of 0.2-11%.**

I. Introduction

Deep neural networks (DNN) have been widely applied in many scenarios, such as image processing, speech recognition, recommendation systems, and search engines [4], [9]. As large datasets and high-dimensional DNN models have quickly exceeded the storage and computing capabilities of even the most powerful single machine, distributing the DNN training jobs across a cluster of machines has become the *de facto* solution for scaling the DNN systems [5]. Due to resource elasticity, deployment of cloud servers for DNN training is becoming extensively favorable.

There are a few different parallelization strategies. For instance, the *Data Parallelism* strategy partitions the whole dataset and makes each training node collectively trains the globally shared model by consuming its own data partition [7]. The *Model Parallelism* strategy splits a large model (often cannot fit into a single device memory) and distributes model partitions among training nodes [1], while *Hybrid Parallelism* combines the aforementioned two strategies [26]. Among these strategies, due to simplicity, the *Data Parallelism* strategy is widely adopted by mainstream DNN systems such as TensorFlow, MXNet, PyTorch, etc.

To cope with the unprecedented increase in the DNN model complexity and datasets, the deployment of training jobs has been pushed to extra large scale. For instance, researchers at Sony use up to 2176 Tesla V100 GPUs to train ResNet50 [15]. However, many recent studies suggest that data parallel DNN training faces a significant scalability problem [13], resulting in resource inefficiency and long model convergence time. To mitigate this problem, various optimization solutions have been proposed to accelerate data parallel DNN training. They all concentrate on addressing three problems, namely, improving local computation resource utilization (e.g., operator fusion [7]), reducing gradient synchronization overhead (e.g., gradient compression [27]), and overlapping computation-synchronization pipeline (e.g., message scheduling and partitioning [17]).

Though the above optimization designs are shown to outperform their non-optimized counterparts, they are often tested at relatively small scale. This makes it difficult to discover their scalability problems, whose symptoms only surface in large-scale deployments [23]. The common practice of understanding the performance implications of various optimizations at large scale is through large scale deployments. However, this trial-and-error approach is incredibly expensive and thus unfavorable on the cloud. Given the high complexity of integrating optimizations into DNN systems [17] and the high cloud deployment cost at large scale, it is more economical and promising to first predict its performance implications without the actual large-scale implementation and deployment. These performance hints can offer *early-feedback* to practitioners to significantly shorten the development cycle and cost.

To offer competitive DNN training infrastructure as a service, it is also the major cloud service providers' intent to offer such performance tools to facilitate efficient and easy-to-use cloud DNN training interface. Furthermore, these tools can help cloud service providers achieve more efficient resource management (e.g., the trial-and-error approach challenges resource provisioning) and enable advanced features (e.g., Neural Architecture Search [30] where hundreds to thousands training jobs with different scalability are executed thus trial-and-error performance tuning is infeasible).

Existing works on performance implications of DNN systems adopt analytical modeling based performance estimation [18], [21], [29]. While these approaches are lightweight and can capture some characteristics of the DNN system backbones, they often fall short in capturing the behaviors of more advanced system features, especially the various optimizations proposed for the fast evolving DNN systems.

Chengru Yang and Zhehao Li are co-first authors. {hibiki, richardhall, rcy, xugb}@mail.ustc.edu.cn, chengli7@ustc.edu.cn, ruichuan.chen@gmail.com, fyan@unr.edu. Cheng Li is the corresponding author.

978-1-6654-4563-4/21/$31.00 ©2021 IEEE
DOI 10.1109/CloudIntelligence52565.2021.00012

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 28,2021 at 06:54:58 UTC from IEEE Xplore. Restrictions apply.

To overcome the limitations of the analytical modeling based performance estimation, in this paper, we propose a generic and extensible performance estimator called PerfEstimator to address the following two major problems. First, to capture the training workflow and to be extensible to incorporate new optimization features, we build PerfModel, a performance model based on an attributed graph, where each vertex represents an operator (e.g., forward propagation and allreduce) and edges represent the ordering constraints between these operators, while the attributes on vertices indicate the time cost. This model is extensible, since one can easily add new operator-related optimizations into the underlying graph as well as the proper ordering constraints.

Next, to democratize large-scale estimation, we build ProfSim, a single-machine scale-simulation framework. We leverage the intra-job predictability [28], i.e., the time cost of local training operators are often constant even when the cluster expands, to profile the time cost of those scale-independent operators. For synchronization, given the cost of synchronizing gradients is difficult to profile without actual system deployments due to the cost varies with respect to different synchronization methods, cluster sizes, etc. To address this challenge, we adopt the simulation method [12] from the HPC community to obtain synchronization cost. PerfEstimator then uses the profiled and simulated time cost from ProfSim to populate the graph built by PerfModel for a given DNN model. Finally, PerfEstimator constructs the computation-synchronization pipeline by taking into account the ordering constraints plus time cost, and derives the overlapping ratio between computation and synchronization, which can be directly translated into scaling efficiency.

We prototype PerfEstimator and evaluate it with two widely used DNN models: VGG13 and ResNet50. Experiments primarily focus on testing PerfEstimator's effectiveness on original training flows at this early stage. Results indicate that PerfEstimator can accurately predict the performance of data parallel DNN training jobs with an impressive error of 0.2-10.9%. A large-scale validation study shows that we are able to perform the performance estimation with up to 128 nodes and the results are consistent with the training trend reported by literature. Finally, the gradient compression use case study validates PerfEstimator's extensibility for various optimizations.

II. Background and Related Work

A. Data parallel DNN training

In a typical data parallel training, a group of training nodes consume disjoint data partitions and collectively train a globally shared DNN model through multiple iterations. In an iteration, each node takes a batch of data samples and performs a local training computation (including both forward and backward propagation phase) to produce the DNN model updates (a.k.a gradients). Following that, there is a gradient synchronization phase, in which locally produced gradients are exchanged and aggregated among all training nodes to compute the new model parameters, which then will be fed into the next iteration. This training process is repeated until the DNN model converges to a state where its loss is below a certain threshold or the maximum number of iterations have been performed.

The gradient synchronization can be done asynchronously to eliminate the negative impacts of a distributed barrier at a penalty of possibly not converging [16], [24]. Therefore, due to its simplicity and convergence guarantee, the strongest consistency model called Bulk Synchronous Parallel (BSP) has become the common practice [25]. Following this, we also align our work to BSP.

There are a family of gradient synchronization strategies to implement the synchronous gradient synchronization phase. One representative in this family is Parameter Server (PS) [11], [14], where training nodes are assigned either server or worker role. All parameter servers together keep a global copy of the parameters of the DNN model, while each worker maintains a local copy of the model parameters and performs neural network computation against its data shard to update its local parameters. Additionally, collective communication is an important component of distributed deep learning, which is boosted by a series of state-of-the-art technologies in High Performance Computing [3], such as reduce, gather, scatter, broadcast, allreduce, allgather, all-to-all, reduce-scatter, etc. These technologies, such as Horovod [20] and NCCL [2], which target low latency and high bandwidth, giving distributed deep learning great improvement in performance.

B. Existing performance modeling work

Understanding performance of data parallel DNN training at large-scale is crucial for not only advising the hardware configurations for deploying DNN training jobs on the cloud, but also guiding the design and implementation of scalable DNN systems as well as various optimizations.

Recently, there are many related work that focus their interests on the performance modeling of data parallel DNN training. For example, Feng Yan *et al.* have proposed a performance model for estimating the scalability of distributed DNN training [29], primarily targeting Parameter Server. This model is further used to power a scalability optimizer that determines the optimal distributed system configuration that minimizes training time. Pelao [18] is another analytical performance model for the lean consumption of resources during the training of DNNs. It builds a mathematical analysis to compute the time cost of layer-wised computation, and embraces a simple communication model to predict the performance of synchronizing gradients via three strategies other than Parameter Server. Most recently, Shi *et al.* have combined both analytical and experimental analysis to understand the performance gaps among four state-of-the-art deep learning frameworks [21]. DNNMem [8] is an accurate estimation tool for GPU memory consumption of DNN models to reduce out-of-memory failures by also leveraging the iterativeness of DNN computation.

PerfEstimator is a generic, framework-independent and extensible performance estimator for large-scale data

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 28,2021 at 06:54:58 UTC from IEEE Xplore. Restrictions apply.

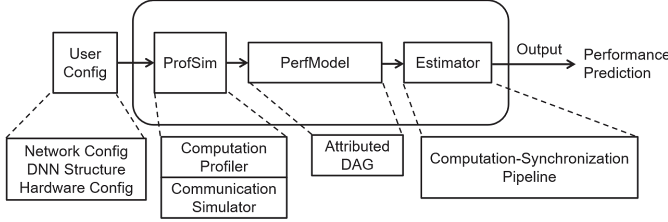

Fig. 1. The architecture of `PerfEstimator`.

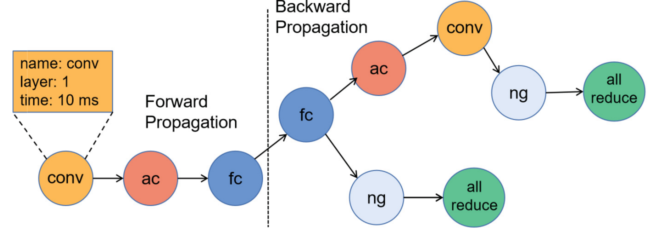

Fig. 2. An illustration of ADAG of a typical simple 3-layer CNN (conv: convolution layer, ac: activation layer, fc: fully connected layer, ng: negotiation).

parallel DNN training. Our solution significantly differs from all the above mentioned existing work as we do not rely on any analytical model to capture the system behaviors, which may not be scalable and extensible to handle the evolution of current DNN systems and various newly proposed optimizations. In more details, we have the following nice properties: 1) we leverage the intra-job predictability of DNN training [28], where the computation cost is often stable w.r.t the cluster size increases. Therefore, instead of proposing a mathematical performance analysis for predicting computation cost, we take advantage of profiling tools to obtain the time cost for all DNN operators for avoiding accuracy loss; 2) we are neither targeting a particular deep learning frameworks nor a specific synchronization strategy. Instead, we rely on network simulator to predict network behaviors, which enables to specify existing or new synchronization strategy; and 3) we allow to extend our performance model to incorporate new optimization features for exercising new design choices.

III. Design of PerfEstimator

To achieve accurate performance prediction for large-scale data parallel training, we build `PerfEstimator` with the following key system components: 1) `PerfModel`, a graph-based performance model that captures the training workflow and is extensible to incorporate new optimization features; and 2) `ProfSim`, a single-machine scale-simulation framework for profiling the time cost of training-related operators and simulating the gradient synchronization cost by taking into account the synchronization strategies and cluster size; and 3) `Estimator` combines the graph model and collected statistics to estimate the actual training performance without real deployments.

Figure 1 illustrates the high-level architecture and workflow of `PerfEstimator`. First, the required input configurations of `PerfEstimator` are: 1) neural network structures, 2) network configurations, e.g., the topology of network, the maximum bandwidth and link latency; and 3) device information, e.g., maximum FLOPS speed.

Once these parameters are in-place, `ProfSim` starts computation profiling and synchronization simulation both on a single machine. For computation profiling, `ProfSim` trains the target DNN model for a limited number of iterations to collect time costs of each training-related operator. Regarding synchronization simulation, `ProfSim` launches SimGrid [6], a simulator built in the HPC community, with user-specified network configurations and synchronization strategy. It is worth mentioning that `ProfSim` can be applied to both local clusters and clouds, since the cloud infrastructure has been evolving to become the optimal option for running HPC and AI workloads. At the end, `PerfEstimator` will use the profiled and simulated time cost from `ProfSim` to populate the graph built by `PerfModel` for any given DNN model. Following that, `PerfEstimator` will construct the computation-synchronization pipeline by taking into account the ordering constraints plus time cost, and finally compute the overlapping ratio between computation and synchronization, which can be directly translated into scaling efficiency.

A. `PerfModel` formalization

`PerfModel` models the overall training process of a given job including both local computation (forward and backward propagation) and global synchronization (e.g., allreduce). We first formulate the performance model `PerfModel` as an attributed directed acyclic graph (ADAG), where each vertex represents the operators, e.g., forward propagation and allreduce, and edges represent ordering constraints between these operators, while the attributes on vertices indicate the time cost. This model is extensible, since one can easily add new operator-related optimizations into the underlying graph as well as the proper ordering constraints. Such abstraction will be proven useful and generic to facilitate later analysis and prediction of scalability of training jobs.

Figure 2 shows the ADAG of a simple 3-layer CNN, in which operators at the first and third layer in the backward propagation phase generate two gradients, respectively. As follows, we will introduce the necessary ordering constraints among operators, which cannot be violated during execution and play a key role in determining the degree of parallelism of data parallel DNN training.

First, as the operators in the forward propagation phase are executed sequentially from one layer to another. Thus, we have $O_{fp}^{conv} \rightarrow O_{fp}^{ac} \rightarrow O_{fp}^{fc}$, where O denotes operators, $\rightarrow$ represents happened-before orderings, "fp" stands for forward propagation, while $conv$, ac and fc correspond to operators of the three DNN layers. Upon the completion of the forward phase, the backward propagation phase will start immediately, and all its operators are also executed in a sequence but in a reversed order of the forward phase. Therefore, we have $O_{fp}^{fc} \rightarrow O_{bp}^{fc}$, and $O_{bp}^{fc} \rightarrow O_{bp}^{ac} \rightarrow O_{bp}^{conv}$, where "bp" stands for backward propagation.

In the backward propagation phase, when $conv$ and fc layer produce gradients, the synchronization step will be triggered, including a negotiation operator (ng) and an allreduce operator, where the latter operator is used to globally aggregate

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 28,2021 at 06:54:58 UTC from IEEE Xplore. Restrictions apply.

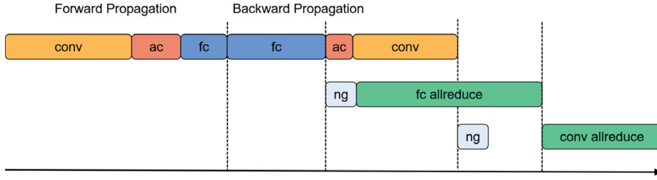

Fig. 3. An illustration of computation-synchronization pipeline, the axis at bottom is timeline.

gradients and broadcast new results, while the former operator is used to coordinate this process. Therefore, we have $O_{bp}^{fc} \rightarrow O_{ng}^{fc} \rightarrow O_{allreduce}^{fc}$, and $O_{bp}^{conv} \rightarrow O_{ng}^{conv} \rightarrow O_{allreduce}^{conv}$.

Unlike all the above ordering constraints, there are some operators that can run in parallel, e.g., the gradient synchronization of a layer and the backward propagation of the preceding layer (i.e., $O_{ng}^{fc} \| \, O_{bp}^{ac}$ and $O_{allreduce}^{fc} \| \, O_{bp}^{ac}$).

B. Profiling and simulation

Then, we annotate vertices of the above graph model with "time cost" attributes. First, for computation-related operators, `PerfEstimator` takes full advantage of existing built-in profilers in mainstream DNN systems to measure their costs throughout a limited number of training iterations on a single machine, for a given DNN model. Second, with regard to the synchronization cost involving network communication, we adopt SimGrid [6], a simulation framework for MPI applications, which works with collective communication primitives that are widely used in data parallel training. The major changes we introduce to SimGrid are the real codes of synchronization strategies, and network configurations of the target deployments (e.g., on the cloud). Then, the codes are actually executed by SimGrid and the predicted latencies will be reported when simulation is done.

C. Computation-synchronization pipeline

Finally, as the key step, `PerfEstimator` will project all operators in the ADAG built by `PerfModel` into a conceptional timeline, which corresponds to the pipeline of computation and synchronization of a single iteration of distributed DNN training. Figure 3 shows the computation-synchronization pipeline projected from the ADAG in Figure 2, where each box represents one operator, and the box length indicates the time cost. This pipeline begins with $conv$ of the forward propagation phase and ends with $allreduce$ for synchronizing gradients produced by $conv$ of the backward propagation phase. The ordering constraints defined in the ADAG are all preserved in this pipeline. However, negotiation and allreduce operators can be executed in parallel with some backward propagation operators. Note that there is a gap between the negotiation and allreduce operator of the gradient produced by $conv$, since we assume the underlying network device has a single port and message sends are serialized.

To predict performance, we define **scaling factor** (α) as:

$$\alpha = \frac{O_{bp}^{l1}.end_ts - O_{fp}^{l1}.start_ts}{O_{allreduce}^{l1}.end_ts - O_{fp}^{l1}.start_ts},$$

where $l1$ stands for the first NN layer, while $start_ts$ and end_ts are timestamps. This **scaling factor** describes the overlapping ratio of computation and synchronization. The value higher, the scalability of the corresponding setup better. Following this, we can estimate the performance p_n of a data parallel DNN training job across n training nodes as $p_n = p_1 \times n \times \alpha$, where p_1 stands for the single-node training performance.

TABLE I
STATISTICS OF TRAINED MODELS.

Name	total size	max gradient	#gradient
VGG13 [22]	507.54MB	392MB	26
ResNet50 [10]	97.46MB	9MB	155

IV. PRELIMINARY EVALUATION

In this section we present the preliminary evaluation of `PerfEstimator`, which only demonstrates its efficiency on original training flows with no optimization features. We leave the extensibility validation in future work. First, we evaluate the accuracy of `PerfEstimator` by comparing its results against testbed measurements of actual deployment. Second, we perform a large-scale validation study for `PerfEstimator` by checking whether it can predict correct performance trends when the training scale is very large.

A. Experimental setup

We run `ProfSim` on a single server with 512 GB DRAM, two 16-core Intel(R) E5-2620 v4 processors and 1 NVIDIA Geforce GTX 1080 Ti GPU. The time cost of local training operators and synchronization operators are profiled by the built-in profiler of MXNet and the SimGrid simulation framework(3.18), respectively. We only specify MPI_allreduce for synchronizing gradients in SimGrid, and leave the examination of other strategies, such as NCCL, as our future work.

For prediction references, as shown in Table I, we train two widely-used DNN models, VGG13 and ResNet50, with the ImageNet dataset [19], over a cluster of 12 physical nodes, each of which has the identical configuration as above but smaller DRAM (64GB). They are connected via 10Gbps Ethernet. Each node runs CentOS7.6, CUDA 10.1, cuDNN 7.5.1, OpenMPI 3.1.2, MXNet 1.6.0, and Horovod 0.19.4. Though not replicating experiments with clouds like AWS, we align the configurations of our local machines to those of the EC2 *p3.2xlarge* instance, e.g., 10Gbps network, 1GPU per machine, for performance references.

B. Accuracy validation of `PerfEstimator`

1) Validation with VGG13: VGG13 [22] contains 10 convolution layers and 3 fully connected layers. One of the important features of VGG13 is that the communication overhead of VGG13 in distributed training is high because it requires large size gradient to be transmitted in the network. Therefore, the communication operators play an important role in the operator execution pipeline of VGG13.

First, we profile the computation cost of each computation operator on a single GPU using MXNet [7] profiler and Chrome tracing tool, setting batch size to be 32. Then, we use the SimGrid simulation framework [6] to get the predicted time cost of allreduce operators of VGG13.

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 28,2021 at 06:54:58 UTC from IEEE Xplore. Restrictions apply.

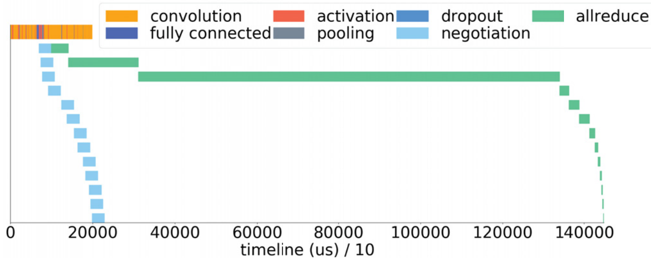

Fig. 4. Computation-synchronization pipeline of VGG13 over 12 nodes.

TABLE II
REAL VS. PREDICTED ITERATION TIME OF VGG13

#node	real(s)	Our predicted(s)	Paleo predicted(s)	Our error	Paleo error
4	1.307	1.283	1.317	-1.84%	**0.77%**
8	1.498	1.466	1.908	**-2.14%**	27.37%
12	1.562	1.534	2.500	**-1.79%**	60.05%

Table IV shows the comparison between simulated time cost and real time cost of allreduce operators when the cluster size is 12. The errors between real and simulated allreduce duration range from 2% to 10%. We think the network fluctuation in real time, and the overhead of SimGrid may be the reasons that lead to this network simulation error. Allreduce simulation for gradients with small message may suffer more from these reasons than large size gradients. For large size messages, the simulator performs well, e.g., the allreduce simulation result of gradient fc0 weight has error around 5%. Considering this gradient occupies more than 70% of the total size of this model, error of total transmission time prediction is mainly impacted by errors of allreduce simulation of large gradients.

With the above results, based on `PerfModel`, we visualize the computation-synchronization pipeline of VGG13, drawn in the same way with Figure 3, as Figure 4 shows. One thing to note here is that we also take the operator switch time into account, which is set to be 10us based on real profiled results.

From the pipeline, we can easily get the predicted iteration time and scalability factor using formula proposed in section III. First, we compare in Table II the real iteration time against the predicted iteration time from 4 to 12 nodes, with batch size being 32. Results show `PerfEstimator` achieves less than 3% prediction error for VGG13. Here, we also compare our results with Paleo [18]. With a four-node deployment, the predicted iteration time costs by both systems are almost equally accurate. However, when the cluster size increases, the prediction accuracy of Paleo keeps decreasing, while the one of `PerfEstimator` remains roughly unchanged. This is because that Paleo totally relies on theoretical analysis, and fails to accurately capture network behaviors at large scale. Furthermore, to validate the correctness of scalability factor α, we compare the real training performance with the predicted training performance using formula in section III, and predictions with α achieve less than 3% error with results shown in table III.

Finally, Table IV compares the simulated time costs of allreduce operators against those of real tests w.r.t different gradient sizes. The simulated results look similar to the real numbers with prediction errors ranging from 2.51 to 9.41%. This says that the accurate simulation of gradient synchronization leads to the accurate overall performance prediction.

TABLE III
REAL VS. PREDICTED SCALING PERFORMANCE OF VGG13

#(node)	real #(batch/s)	α	predicted #(batch/s)	error
1	5.040	1	Ø	Ø
4	3.060	0.155	3.118	+1.87%
8	5.340	0.135	5.457	+2.18%
12	7.684	0.129	7.823	+1.83%

TABLE IV
COMPARISON BETWEEN REAL AND SIMULATED ALLREDUCE DURATION OF VGG13, NUMBER OF NODES = 12.

gradient	size(KB)	real(ms)	simulated (ms)	error
conv0 weight	6.75	0.332	0.350	+5.42%
conv0 bias	0.25	0.289	0.307	+6.23%
conv1 weight	144	1.953	1.904	-2.51%
conv4 bias	1	0.457	0.414	-9.41%
fc0 weight	401408	974.067	1026.463	+5.38%

TABLE V
REAL VS. PREDICTED ITERATION TIME OF RESNET50.

#(node)	real(s)	predicted(s)	error
4	0.329	0.365	+10.94%
8	0.439	0.440	+0.23%
12	0.529	0.511	-3.40%

TABLE VI
REAL VS. PREDICTED SCALING PERFORMANCE OF RESNET50.

#(node)	real #(batch/s)	α	predicted #(batch/s)	error
1	6.262	1	Ø	Ø
4	12.158	0.438	10.959	-9.86%
8	18.223	0.363	18.182	-0.23%
12	22.684	0.313	23.483	+3.52%

2) Validation with ResNet50: ResNet50[1] [10] contains 53 convolution layers, 53 batch normalization layers and 1 fully connected layers. The gradients to be transmitted during training of ResNet50 are much smaller than VGG13's, but the number of gradients to be transmitted is much larger than VGG13's. We repeat the same progress as what we have done with VGG13 to get the performance prediction of ResNet50 using Mxnet [7] and SimGird [6]. Part of the simulation time, ResNet50 prediction result and scalability factor α's validation are listed in Table V, and Table VI, respectively. The error of iteration time prediction (Table V) and the error of scalability factor prediction (Table VI) are both below 11%, which verifies that `PerfEstimator` is accurate. Prediction of ResNet50 shows to be more unstable than VGG13. This is mainly caused by the large number of small gradients of ResNet50. While the simulated communication time cost of these small gradients remains almost the same, the real time cost of these gradients shows to be unstable and deviates from the theoretical value because of network fluctuation, which is reflected by the instability of prediction error.

[1]Here, among all variants, we use ResNet50v1.

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 28,2021 at 06:54:58 UTC from IEEE Xplore. Restrictions apply.

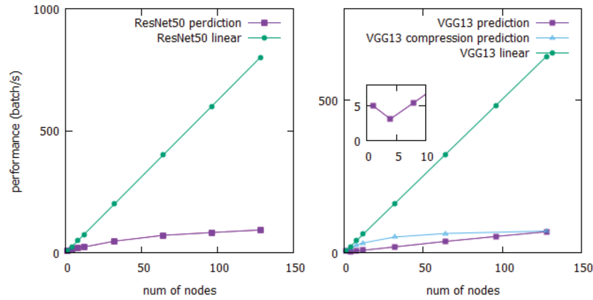

Fig. 5. Performance prediction of VGG13 and ResNet50 on large clusters. Linear line is the best performance in theory that all optimizations would target.

C. Performance study at large-scale

We use `PerfEstimator` to predict training speed for VGG13 and ResNet50 at large scales with 32, 64, 96, 128 nodes, via the single-machine simulation. As shown in Figure 5, the prediction by `PerfEstimator` correctly reflects the performance trend of distributed VGG13 and ResNet50. It is worth mentioning that in the prediction, we find the training performance of VGG13 has a small drop when the number of nodes is 4, which matches the real situation. This observation indicates that `PerfEstimator` can be applied in practice to guide the configuration of distributed DNN systems.

Extensibility study. We use gradient compression, a common solution to eliminate communication bottlenecks of data-parallel DNN training, as a use case to study `PerfEstimator`'s extensibility. Here, we profile our internal implementation of TernGrad [27], one of representative gradient compression algorithms, and extend the VGG13's ADAG and pipeline to incorporate such feature. As the light blue curve in Figure 5 shows, the predicted training scaling performance when TernGrad is used is better than the original training process, since it largely reduces the amount of gradients exchanged per iteration basis. However, our simple simulation also tells that the improving factor drops when scale becomes extremely large (validated by our internal deployments), which further suggests that the stand-alone gradient compression is not sufficient and perhaps needs to be combined with other optimizations, e.g., better synchronization strategies other than Ring-allreduce.

V. Conclusion

`PerfEstimator` is a generic framework and extensible performance estimator for large-scale data parallel DNN training. It is driven by an attributed graph based performance model, a computation and synchronization profiling and simulating tool for obtaining runtime time costs on a single machine, and a computation-synchronization pipeline builder. Results show that `PerfEstimator` can accurately predict the performance of data parallel DNN training using popular benchmarks VGG13 and ResNet50. Throughout large-scale checks, it reports expected performance trends.

Acknowledgment

We sincerely thank all anonymous reviewers for their insightful feedback. This work was supported in part by National Nature Science Foundation of China 61802358, "USTC Research Funds of the Double First-Class Initiative" YD2150002006, and National Science Foundation CCF-1756013, IIS-1838024.

References

[1] Mesh tensorflow - model parallelism made easier. https://github.com/tensorflow/mesh, 2020.
[2] Nvidia collective communications library. https://developer.nvidia.com/nccl/, 2020.
[3] Open MPI. https://www.open-mpi.org/, 2020.
[4] Anelia Angelova, Alex Krizhevsky, and Vincent Vanhoucke. Pedestrian detection with a large-field-of-view deep network. In *ICRA 2015*.
[5] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv*, 2018.
[6] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *JPDC 2014*.
[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv*, 2015.
[8] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating gpu memory consumption of deep learning models. In *ESEC/FSE '20*, 2020.
[9] Javier Gonzalez-Dominguez, Ignacio Lopez-Moreno, Pedro J Moreno, and Joaquin Gonzalez-Rodriguez. Frame-by-frame language identification in short utterances using deep neural networks. *Neural Networks 2015*.
[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR 2016*.
[11] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. Flexps: Flexible parallelism control in parameter server architecture. *VLDB 2018*.
[12] Nikhil Jain, Abhinav Bhatele, Sam White, Todd Gamblin, and Laxmikant V Kale. Evaluating hpc networks via simulation of parallel workloads. In *SC'16*.
[13] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *EuroSys 2019*.
[14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI 2014*.
[15] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. Imagenet/resnet-50 training in 224 seconds. *ArXiv*, 2018.
[16] Benoît Patra. Convergence of distributed asynchronous learning vector quantization algorithms. *Journal of Machine Learning Research*, 2011.
[17] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *SOSP 2019*.
[18] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR 2017*.
[19] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *IJCV 2015*.
[20] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv*, 2018.
[21] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In *DASC/PiCom/DataCom/CyberSciTech 2018*.
[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.
[23] Cesar A. Stuardo, Tanakorn Leesatapornwongsa, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, Wei-Chiu Chuang, Shan Lu, and Haryadi S. Gunawi. Scalecheck: A single-machine approach for discovering scalability bugs in large distributed systems. In *FAST 2019*.
[24] Ye Tian, Ying Sun, and Gesualdo Scutari. Asy-sonata: Achieving linear convergence in distributed asynchronous multiagent optimization. In *Allerton 2018*.
[25] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
[26] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *EuroSys 2019*.
[27] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *NIPS 2017*.
[28] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI 2018*.
[29] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *KDD 2015*.
[30] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A deep learning exploratory-training framework. In *OSDI*, 2020.

Authorized licensed use limited to: UNIVERSITY OF NEVADA RENO. Downloaded on September 28,2021 at 06:54:58 UTC from IEEE Xplore. Restrictions apply.