Performance Modeling and Analysis of Distributed Deep Neural Network Training with Parameter Server

Xuan Zhang[1], Jiao Zhang[12*] , Dehui Wei[1], Tian Pan[12], Tao Huang[12]
[1]Beijing University of Posts and Telecommunications, Beijing, China
[2]Purple Mountain Laboratories, Nanjing, China
{x_z, jiaozhang, dehuiwei, pan, htao}@bupt.edu.cn

***Abstract*—With the growth of dataset size and the development of hardware accelerators, the application of deep neural networks (DNN) in various fields has made great breakthroughs. In order to improve the training speed of DNN, distributed training has been widely used. However, the imbalance between computation and communication makes distributed training difficult to achieve maximum efficiency. Therefore there is a need to detect the bottleneck state and verify the effect of some optimization schemes. Testing on a physical cluster incurs additional time and cost overhead. This paper builds a DNN-specific performance model that is used for bottleneck detection and tuning at a low cost. We build this model through detailed analysis and reasonable assumptions. We also focus on fine-grained modeling of scalability and network components, which are key factors affecting performance. Then we verify the performance model with an average error of 5% on testbed and emulator. Finally, we provide use cases of the performance model.**

***Index Terms*—Distributed Training, Performance Modeling, Communication Optimization**

I. Introduction

In recent years, the research on DNN has made exciting progress in many fields, such as natural language processing [1], computer vision [2], recommendation systems [3] and etc. In addition to the more advanced DNN architecture, this progress mainly comes from larger data sets and more advanced hardware acceleration devices (GPU, TPU, FPGA, etc). However, the computing resources on a single device cannot support fast training of DNNs. This results in researchers or enterprises needing a longer time to validate ideas or deploy applications. The unacceptable time cost gives birth to distributed training which improves the training speed by distributing the tasks to multiple nodes.

The synchronous data parallel method that does not affect convergence currently is the most widely used distributed training method, where all nodes have the same DNN model but are trained with a different subset of the dataset. The training results are aggregated and distributed through the network to ensure that the DNN is trained on the complete dataset. Before the aggregation is completed, nodes stop training and wait for the aggregation results to update the DNN parameters. The additional time cost introduced by parameter aggregation is an important factor determining the efficiency of distributed training. Matching between communication and computation is necessary to achieve a near-linear speedup ratio.

*Corresponding author

However, the current computing performance is greater than communication. Taking the GPU as an example, it completes the replacement of new models in a period of about two years to achieve several times the performance improvement (H100 GPU achieves at least 3x overall performance compared to the previous). It takes several times longer to upgrade the performance of network equipment (Ethernet speeds from 10Gbps to 100Gbps take eight years to standardize [4]). Therefore, communication will become the bottleneck of distributed training for a long time in the future.

In order to fully optimize network bottlenecks, it is necessary to detect the current state of network bottlenecks and select appropriate solutions to make corresponding adjustments. This detection and adjustment can be performed on physical machines, but it will bring extra time and cost overhead, which may exceed the benefits of optimization. Therefore, a better approach is to establish a performance model to estimate bottlenecks and validate the benefits of optimization solutions. Furthermore, we can choose the optimal solution that meets the requirements with optimized parameters.

It is challenging to model the performance of such a complex system as distributed training. The difficulty lies in (1) How to use a unified mathematical form to partition and define the process of practical distributed training. (2) How to reflect the impact of scale on overall performance. (3) How to improve accuracy without introducing too much complexity in network modeling. Unfortunately, none of the existing solutions have addressed the aforementioned issues.

In this paper, we propose an advanced distributed training performance model that addresses the aforementioned challenges. We simplify the model by linearizing processes with different characteristics in the pipeline. Then we conduct detailed analysis and modeling of the parts that affect performance. The performance model we ultimately established can effectively predict performance in real-world scenarios.

In summary, we make the following contributions:

- We analyze the characteristics of different processes in distributed training and make reasonable definitions that simplifies the complexity of the model.
- We conduct in-depth modeling for overlapping technology, scalability, and network to reflect their impact and improve the accuracy of predictions.
- We verify the performance model using testbeds and simulations. In addition, we also provide two use cases.

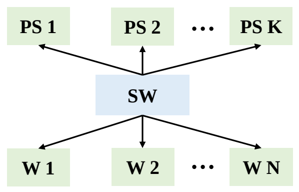


Fig. 1. Parameter server architecture

II. Motivation

A. Application of Overlapping Technology

Due to the imbalance between communication and computation, overlapping techniques are introduced into distributed training. Benefiting from the layerwise structure of DNN, the calculation of each layer in backward propagation only depends on the output of the previous layer. The communication of the gradient does not need to wait until all layers have completed the calculation, but after the current layer completes the calculation. Therefore, computation and communication can be executed in parallel to achieve overlapping and improve training efficiency. This technique has been widely used in popular DNN training frameworks [5].

With the application of the overlapping technology, the overall training time is no longer a simple linear summation relationship of the time spent on each step. We need to model this relationship in the model to accurately reflect actual performance. However, the existing performance model did not consider this point because it was proposed too early, which made it impossible to restore the actual performance.

B. Limitations of the Previous Performance Model

In addition to new technologies, several influencing factors influenced the results more than ever. Previous work on distributed training performance modeling has focused on two aspects: (i) establish a performance model for a specific part of distributed training, which mainly includes computing resources such as multi-process and multi-GPU [6]–[9]. Then, the optimization objective is proposed, and mathematical methods such as convex optimization are used to solve it. (ii) establish an overall model of distributed training to verify and evaluate existing hardware and optimization methods [10]–[12]. The purpose of the evaluation is to quantify the impact of hardware and software on performance.

The modeling of communication in the above two aspects of works is usually a simple linear model. However, communication has become an important factor affecting performance and requires more in-depth modeling to reflect its impact. In addition, they did not take into account the short board effect that becomes more and more obvious as the scale increases. This needs to be considered and added to the modeling under the current rapidly expanding scale of distributed training.

III. Overall Modeling

In this section, we first establish the basic model. Then, we model the overlapping techniques and scalability. **Table I** summarizes some symbols used in modeling.

TABLE I
SUMMARY OF MAIN NOTATIONS

Name	Description
N	Number of workers
K	Number of parameter servers
B	Link bandwidth
M	The number of iterations required for an epoch
L	Total layers of neural network
G	Aggregation granularity
T	Total train time of an epoch
t_X	The time taken to complete the X process
τ_X	The completion time of the X process
PN	The number of packets sent for aggregation
pks	Data packet size
S_f	Switch forwarding speed

A. Basic Model Establishment

An iteration of distributed training typically involves the following steps:

Data Loading (IO): Nodes load data into memory and preprocess data to prepare for input.

Forward Propagation (FP): The processed data is used as input to calculate the output of each layer along the DNN layer-wise structure.

Backward Propagation (BP): The loss is calculated by the loss function using the final output of DNN. And the gradient error of each layer can be obtained through the chain rule.

Gradient Aggregation (GA): The gradient error from nodes is aggregated and then returned to each node.

Updating Parameter (UP): The nodes update the parameters of the DNN with the aggregated gradient error.

The above iteration continues multiple times until all data is trained once, which is called an epoch. A training session lasts for multiple epochs. There are two main communication architectures for gradient aggregation, parameter servers(PS) [13] and ring all-reduce [14]. We mainly study the former.

In the PS as shown in Fig. 1, nodes that specialize in aggregating gradients are called parameter servers, and others participating in training are called workers. After calculating, workers send the gradient error to the parameter servers, which aggregate gradients and broadcast them back to all nodes.

Based on the characteristics of parameter aggregation under PS, we divide parameter aggregation into three processes and make the following definitions.

Definition 1. [Forward Communication (FC)] *The workers send gradient errors to parameter servers.*

Definition 2. [Aggregation Gradient (AG)] *The process of aggregating all gradients by parameter servers.*

Definition 3. [Backward Communication (BC)] *The process of sending back all gradients to the worker.*

This is because workers do not wait for the current iteration of all workers to complete and continue to perform the next iteration. So the difference in time spent by each process causes that the time at which each iteration starts to be cumulatively affected by the previous. Therefore, we cannot use the gradient update completion time to divide the time for synchronization.

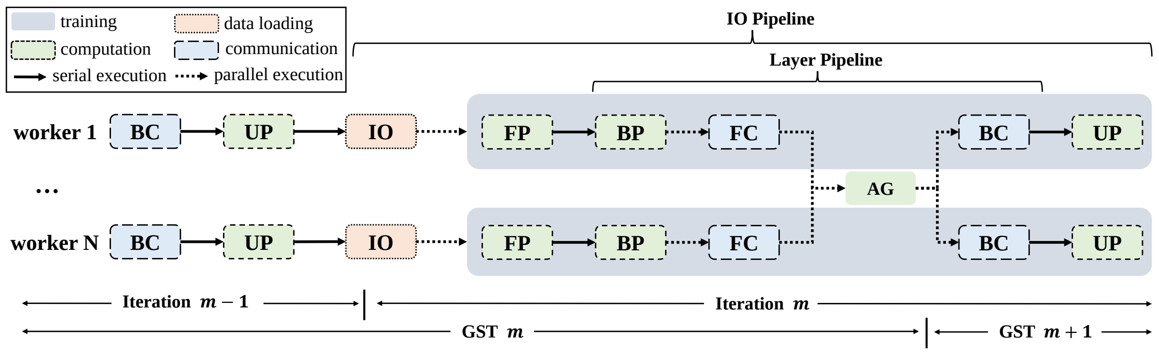


Fig. 2. Overview of Distributed Training process structure with overlapping optimization

Through the above definitions, we find that the process status of all workers is consistent when the AG process is completed. The only impact is from the time of the last time backward communication and parameter update. So we make the following definition.

Definition 4.[Global Synchronization Time(GST)] *Due to the difference in the time spent by workers in each process, the workers may be different progress of different processes at the same time. However, when workers aggregate the gradient to completion, the process state of workers gradually approaches until the process state of all workers reaches a completely consistent unity at the moment when aggregating all gradients. We define the time taken from inconsistency to consistency as the global synchronization time (GST).*

According to Definition 4, we can get the GST.

$$GST(m,n) = \begin{cases} t_{io}^{m,n} + t_{fp}^{m,n} + t_{bp}^{m,n} + t_{fc}^{m,n} + t_{ag}^{m.n} & m=1 \\ t_{bc}^{m-1,n} + t_{up}^{m-1,n} + t_{io}^{m,n} + t_{fp}^{m,n} + t_{bp}^{m,n} & \\ + t_{fc}^{m,n} + t_{ag}^{m,n} & else \end{cases} \tag{1}$$

We obtain an epoch time without considering any optimizations, which limit the differences in time spent by different workers to one iteration.

$$\begin{aligned} T &= \max_n \left\{ \sum_{m=1}^{M} t_{iter}^{m,n} \right\} \\ &= \max_n \left\{ \sum_{m=1}^{M} GST\left(m,n\right) + t_{bc}^{M,n} + t_{up}^{M,n} \right\} \\ &= \sum_{m=1}^{M} \max_n \left\{ GST\left(m,n\right) \right\} + \max_n \left\{ t_{bc}^{M,n} + t_{up}^{M,n} \right\} \end{aligned} \tag{2}$$

B. Pipeline Modeling

As shown in Fig. 2, there are two pipelines that are used for overlapping data loading and the rest, as well as overlapping backward propagation and parameter aggregation. We have the following strict definitions for two pipelines.

Definition 5. (IO Pipeline) *IO is the only process related to data loading and irrelevant to other operations. Therefore, IO and other processes form a two-stage pipeline, overlapping with other processes.*

Definition 6. (Layer Pipeline) *Since the layerwise structure of DNN, the calculation of the current layer is only related to the output result from the previous layer. The calculated gradient error of each layer can be obtained immediately after the calculation of the current layer is completed. And the parameter server aggregates the gradient of the same layer from all workers and broadcasts those back. A pipeline can be formed that hides the aggregation gradient process.*

To linearize the pipeline, we distinguish between two processes in the pipeline. For the process that is only blocked by the previous of the same type, we call it a type I process. For the type I process, we can calculate the completion time of the process in each iteration, which is the cumulative sum of the previous time spent as shown in Eq. 3.

$$\tau_I^{m,n} = \sum_{i=1}^{m} t_I^{i,n} \tag{3}$$

Furthermore, the process that is blocked by the process of its previous iteration and the previous process of the current iteration at the same time, we call it a type II process. Type II process run in the first iteration after the end of the previous process. After that, it will run after the last iteration of the current process and the next iteration of the previous process.

$$\tau_{II}^{m,n} = \begin{cases} \tau_{last}^{m,n} + t_{II}^{m.n} & m=1 \\ \max\left\{\tau_{last}^{m.n}, \tau_{II}^{m-1,n}\right\} + t_{II}^{m,n} & else \end{cases} \tag{4}$$

Through the above analysis, we can know that the IO process in the IO pipeline is a type I process, and all the remaining processes can be unified into a process as a type II process. The purpose of the IO pipeline is to hide the time of the IO process, so we hide the IO time and convert the pipeline into a timeline. We can express IO times that are not hidden as explicit times by Eq. 5.

$$e\left(t_{IO}^{m}\right) = \begin{cases} t_{IO}^{m} & m=1 \\ \max\left\{0, \tau_{IO}^{m} - \tau_{DT}^{m-1}\right\} & else \end{cases} \tag{5}$$

Combined with the above definitions, we know that the backward propagation process is a type I process, and the other processes are considered a type II process. But we want to hide the non-computation process in the type II process. Because there is always a process in front of the type II process, the explicit time exposed by the type II process is after the previous process is all completed. We get the explicit time of FC, AG, BC by Eq. 6.

$$e\left(t_{fc/ag/bc}\right) = \tau_{fc/ag/bc} - \tau_{bp/fc/ag} \tag{6}$$

C. Scalability Modeling

In this section, we highlight the influence of the number of workers for modeling scalability.

We treat the explicit time of different workers are independent random variables, which satisfies the normal distribution. It is easy to obtain the probability density function and probability cumulative function of GST.

$$F\left(m,t\right) = \int_{-\infty}^{t} \frac{1}{\sigma\left(m\right)\sqrt{2\pi}} e^{-\frac{\left(t-\mu(m)\right)^2}{2\sigma(m)^2}} dt \tag{7}$$

where μ and σ are the sum of the corresponding values of the process. The variance of different processes is unified into a single value to facilitate parameter determination. The latter is included as a tunable parameter in the model

Since these N workers are independent, we can know the probability distribution satisfied by the GST that spends the most time by Eq. 8. The probability density function $f_{\max}\left(m,t\right)$ is obtained by derivatizing the probability distribution function.

$$F_{\max}\left(m,t\right) = \prod_{n=1}^{N} P\left(t^{m,n} \leq t\right) = F^{N}\left(m,t\right) \tag{8}$$

Combining Eq. 2, the epoch time considering scalability is calculated by Eq. 9.

$$T = \sum_{m=1}^{M} E\left[f_{\max}\left(m,t\right)\right] \tag{9}$$

This formula summarizes the modeling of overall performance with overlapping techniques and consideration of scalability. Below, we will focus on modeling the network.

IV. Network Modeling

In this section, we model the AG process at the packet level according to the traffic characteristics in PS architecture.

A. Gradient Communication

Due to the link load being different in the PS architecture, we define it as follows to highlight these important links.

Definition 7. (Barrier Link). *The link most likely to significantly affect the synchronization barrier time when performing gradient aggregation is called the barrier link.*

According to Definition 7, it can be known that the existence of barrier links is related to the number between workers and parameter servers. In Fig. 1, when the number of parameter servers is less than the number of workers, the barrier links are the links from the top of rack switch to the parameter servers. On the contrary, the barrier links are all links from the top of rack switch to the workers.

To highlight key links while simplifying the model, we use two modeling methods. For the non-barrier link, we use a linear function to represent the time spent on it.

$$t_{non-barrier}\left(l\right) = t_{base-delay} + \alpha \times PN\left(l\right)\frac{pks}{B} \tag{10}$$

where $t_{base-delay}$ includes the startup delay, propagation delay, and processing delay on the non-barrier link, α, and pks are associated with the network protocol used.

For the barrier link, we use queuing theory for modeling. We regard the forwarding at the barrier link as the queuing process of M/M/1 where the input and output satisfy the Poisson process. So we can get the time spent on forwarding.

$$t_{forward} = \frac{1}{\mu - \lambda} = \frac{1}{S_f - \frac{1}{B}} = \frac{B}{S_f \times B - 1} \tag{11}$$

Assuming that the worker to the aggregator has experienced h hops in the barrier link, the forward communication time on the barrier link can be calculated.

$$t_{barrier} = t_{forward} \times h \tag{12}$$

The combination of barrier time and non-barrier time can get the time spent in forward and backward communication.

B. Gradient Aggregation

The parameter servers must receive the complete gradient from all workers, so we make the following definition.

Definition 8. (Aggregation Granularity) *Because the primitives at the application layer are different in specific communication architectures. So we definite Aggregation Granularity is the minimum gradient data size to be aggregated, the symbol denote as G.*

The aggregation granularity is divided and sent to K parameter servers for aggregation in the PS architecture.

$$G = \frac{PN\left(l\right)}{K} \tag{13}$$

The aggregate gradient is an additive operation of gradient, so we model it as a linear function of aggregation granularity.

$$t_{ag} = a + b \times G \tag{14}$$

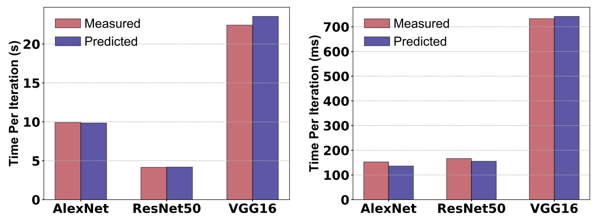
Because the parameter server needs to receive complete aggregation granularity to start aggregation from all workers, so there is a gradient delay before aggregating.

There are two extreme cases of maximum and minimum aggregation delay. When one of the aggregation granularities is sent far ahead of other workers, the smallest data reception is G, the smallest aggregation delay.

$$t_{ag-delay}^{\min} = \frac{G}{B} \tag{15}$$

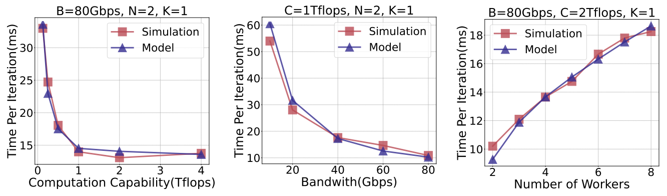
Another extreme case is that packets sent by all workers to PS keep almost the same pace from the same starting point. So when one aggregation granularity is completely received, other aggregation granularities will be almost completely received, which is the maximum aggregation delay.

$$t_{ag-delay}^{\max} = \frac{G + \left(N-1\right)\left(G-1\right)}{B} = \frac{NG - N + 1}{B} \tag{16}$$



(a) 1G TCP network (b) 100G RDMA network

Fig. 3. Performance model validation on the testbed.



(a) Computation (b) Communication (c) # of workers

Fig. 4. Performance model validation on the simulator.

The actual aggregation time would be somewhere in between, so we use it as a range tunable parameter in our model.

In addition to determining the aggregation time on a single server, we still need to consider the latest aggregation time among all parameter servers. We know that aggregate completion time is affected by all workers, so we can make the following transformation to decouple it from GST. And then the goal of solving is transformed into solving the latter.

$$\max_n \{GST(m,n)\} \Rightarrow \max_n \{GST(m,n) - t_{ag}^{m,n}\} + \max_n \{t_{ag}^{m,n}\} \tag{17}$$

Because the gradients of all workers sent to different servers need to be forwarded by the same location, the arrival of the last packets sent means the latest complete time to aggregation.

$$\begin{aligned}\max_n \{t_{ag}^{m,n}\} &= \max_{n,k}\left[\{t_{ag-delay} + \left(c^{m,n,k}+1\right)(a+bG)\}\right] \\ &= t_{ag-delay} + N\lceil \frac{PN(l)}{G\times K}\rceil (a+bG)\end{aligned} \tag{18}$$

where $c^{m,n,k}$ represents the order in which worker n completes the aggregation gradient in parameter server k at iteration m.

V. Verification and Use Case

In this section, we validate the performance model using the popular training framework and simulation under real workloads. Then we provide two use cases of the model.

A. Environment Setup

Testbed environment. Our testbed includes 4 servers (3 workers and 1 parameter server) with Intel Silver 4210 CPU, 128 GB RAM, NVIDIA RTX 3090 GPU, and a Mellanox ConnectX-6 100G network card, which are interconnected by two alternative networks: 1 Gbps TCP and 100 Gbps RoCEv2. We chose PyTorch and BytePS as the training framework.

Simulation environment. We develop an emulator to bypass hardware limitations and validate the model from more perspectives. We use gRPC in mininet to achieve gradient aggregation. For the rest of the processes, we use multiple processes to simulate runtime based on the actual DNN.

Benchmark. For testbed experiment, we chose three models, including AlexNet [15], ResNet50 [2] and VGG16 [16]. For the simulation, we choose AlexNet to evaluate performance. The average iteration time is used as the metric. We use public DNN information including parameter quantity and communication as inputs for the performance model, and preset network parameters that match the environment.

B. Verification

Testbed validation. Fig. 3 show the measurement on testbed and prediction of the performance model under 1G TCP and 100G RDMA networks. These two kinds of networks represent the extreme cases of two kinds of network resources. The results showed that our model closely matches the average iteration time on real hardware with an average error of 5%.

We found that the improvement in communication speed is model-specific. For example, at 1G bandwidth, the average iteration time of Alexnet is twice that of ResNet, and at 100G bandwidth, the average iteration time of the two is almost the same. The improvement in communication performance has brought AlexNet a 2x speed increase compared to ResNet. This shows that the marginal effect of various resources in distributed training is closely related to the DNN structure, but in practice, there are not enough optional hardware and enough time cost to explore the marginal effect of each model. Our performance model can be obtained with a small amount of stand-alone data and certain parameters.

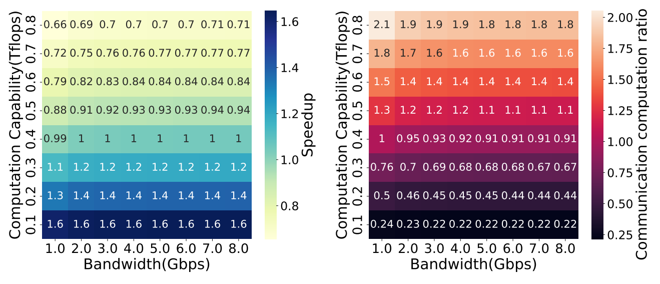
Simulation Validation. We chose three key parameters as inducements to affect the system: compute capability of GPU, network bandwidth, and the number of workers. Fig. 4 shows the results of simulation and model prediction. We can see that our performance model estimates training time fairly and accurately. The maximum error is within 10%, and the average error is still less than 5%.

C. Use Case

The goal of the performance model is to allow analysis in the case where hardware is not available. We illustrate the benefits with two use cases. These two use cases answer the following two questions: 1). How to evaluate the currently distributed training bottleneck? 2). How to choose the compression ratio when using gradient compression?

Evaluation. An important metric for distributed training evaluation is scalability. The measurement that can represent the scalability is the speedup ratio and the communication computing ratio, which help us to locate where the bottleneck of the current training performance improvement is and whether the state of the bottleneck is serious.

With two evaluation metrics, we can evaluate distributed training performance and the bottleneck state. In Fig. 5, we



(a) Speedup. (b) Communication/ Computation.

Fig. 5. Scalability evaluation in different scenarios.

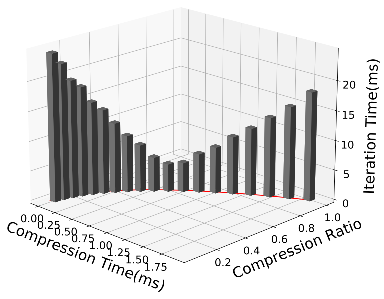


Fig. 6. Parameter optimization of gradient compression.

measure both under different bandwidths and computation capability using the experimental setup in § V-C (when the number of workers is 2). The results help us quantitatively evaluate the performance status of the current distributed training and clarify the direction of optimization.

Optimization. Gradient compression is a common and effective way to improve training speed. Gradient compression occurs between backward propagation and forward communication, and it takes a certain amount of time to compress the sent gradients. Assuming that the time spent on gradient compression has a certain relationship with the compression rate, which can be easily pre-measured in practical applications. Then the parameters can be easily optimized through our performance model.

The t_c and r_c respectively represent compression time and compression rate. We assume that both satisfy the inverse square relationship according. In order to introduce gradient compression, we only need to make corresponding small changes in the performance model. We include the time spent in gradient compression into the time of forward communication. For the total number of packets, we multiply it by $1 - r_c$.

We can easily get the performance trend by changing the value of different compression ratios. The example under the assumed parameters is shown in Fig. 6. We can obviously find that when $r_{compress} = 0.6$ has the fastest training speed.

VI. Conclusion

In this paper, we propose a performance model for distributed training. Overcoming the limitations of previous performance models, our performance model takes into account the overlapping techniques used by current popular distributed frameworks. In addition, we also analyze and model the scalability and network that affect performance with in-depth theory. The validation results on the testbed and emulator show that our performance model accurately predicts training performance. Finally, we show use cases of performance models to demonstrate how it can be applied in practice.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive comments. This work is partly supported by the National Key R&D Program of China under Grant No. 2022YFB2901700, the National Natural Science Foundation of China (NSFC) under Grant No. 62132022, and Fok Ying Tung Education Foundation under Grant No. 171059.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[3] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM computing surveys (CSUR)*, 52(1):1–38, 2019.

[4] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. In *USENIX NSDI 21)*, 2021.

[5] Hao Zhang, Zeyu Zheng, and others Xu. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *USENIX ATC 17*, pages 181–193, 2017.

[6] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *21th ACM SIGKDD*, pages 1355–1364, 2015.

[7] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous gpu/cpu clusters. In *USENIX OSDI 20)*, pages 463–479, 2020.

[8] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems*, 4:652–672, 2022.

[9] Shaohuai Shi, Xiaowen Chu, and Bo Li. Mg-wfbp: Merging gradients wisely for efficient communication in distributed deep learning. *IEEE TPDS*, 32(8):1903–1917, 2021.

[10] Ziqian Pei, Chensheng Li, Xiaowei Qin, Xiaohui Chen, and Guo Wei. Iteration time prediction for cnn in multi-gpu platform: modeling and analysis. *IEEE Access*, 7:64788–64797, 2019.

[11] Shaohuai Shi, Qiang Wang, and Xiaowen Chu. Performance modeling and evaluation of distributed deep learning frameworks on gpus. In *2018 IEEE(DASC/PiCom/DataCom/CyberSciTech)*, pages 949–957, 2018.

[12] Shaohuai Shi, Qiang Wang, Xiaowen Chu, and Bo Li. A dag model of synchronous stochastic gradient descent in distributed deep learning. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 425–432. IEEE, 2018.

[13] 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 14*, pages 583–598, 2014.

[14] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv:1802.05799*, 2018.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[16] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.