# Modeling and Optimizing the Scaling Performance in Distributed Deep Learning Training

Ting Liu[1,2,*], Tianhao Miao[1,2,*], Qinghua Wu[1,3], Zhenyu Li[1,3], Guangxin He[1,2]
Jiaoren Wu[4], Shengzhuo Zhang[4], Xingwu Yang[4], Gareth Tyson[5,6], Gaogang Xie[2,7]
[1]Institute of Computing Technology, Chinese Academy of Sciences, China
[2]University of Chinese Academy of Sciences, China, [3]Purple Mountain Laboratories, China
[4]Kuaishou, China, [5]Hong Kong University of Science and Technology, Hong Kong
[6]Queen Mary University of London, U.K., [7]Computer Network Information Center, Chinese Academy of Sciences, China
{liuting19g,miaotianhao18z,wuqinghua,zyli}@ict.ac.cn,heguangxin17@mails.ucas.ac.cn
{wujiaoren,zhangshengzhuo03,yangxingwu}@kuaishou.com,g.tyson@qmul.ac.uk,xie@cnic.cn

## ABSTRACT

Distributed Deep Learning (DDL) is widely used to accelerate deep neural network training for various Web applications. In each iteration of DDL training, each worker synchronizes neural network gradients with other workers. This introduces communication overhead and degrades the scaling performance. In this paper, we propose a recursive model, *OSF* (Scaling Factor considering Overlap), for estimating the scaling performance of DDL training of neural network models, given the settings of the DDL system. *OSF* captures two main characteristics of DDL training: the overlap between computation and communication, and the tensor fusion for batching updates. Measurements on a real-world DDL system show that *OSF* obtains a low estimation error (ranging from 0.5% to 8.4% for different models). Using OSF, we identify the factors that degrade the scaling performance, and propose solutions to effectively mitigate their impacts. Specifically, the proposed adaptive tensor fusion improves the scaling performance by 32.2%∼150% compared to the constant tensor fusion buffer size.

## CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Distributed computing methodologies**; • **Networks** → *Network measurement*.

## KEYWORDS

distributed deep learning, scaling performance, performance modeling, tensor fusion

**ACM Reference Format:**
Ting Liu[1,2,*], Tianhao Miao[1,2,*], Qinghua Wu[1,3], Zhenyu Li[1,3], Guangxin He[1,2] and Jiaoren Wu[4], Shengzhuo Zhang[4], Xingwu Yang[4], Gareth Tyson[5,6], Gaogang Xie[2,7]. 2022. Modeling and Optimizing the Scaling Performance in Distributed Deep Learning Training. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3485447.3511981

*Co-first authors.



*WWW '22, April 25–29, 2022, Virtual Event, Lyon, France*

ACM ISBN 978-1-4503-9096-5/22/04.
https://doi.org/10.1145/3485447.3511981

## 1 INTRODUCTION

The Web unremittingly produces quintillions of bytes of data every single day. These data are fed to deep learning models to greatly improve the performance of Web applications, e.g., precision of image recognition [19, 46], speech recognition [17] and language translation [12]. With the ever-increasing sizes of Web data and larger deep learning model sizes, it imposes great challenges to model training on a single machine. For example, Google applies BERT models [12] to Search services, while the BERT-base model has 110 million parameters and needs about two weeks of training on a single TPUv2 card. Extremely long model training time significantly impedes the progress of web application development. There is a trend that Distributed Deep Learning (DDL) is used to accelerate such model training.

In DDL training, each worker trains a copy of the deep neural network locally, and synchronizes the model gradients with other workers at the end of the iteration [25, 27, 33]. Naturally, as a DDL system scales up, the communication overhead between these workers increases, affecting the performance of the training process [34]. There are two key architectures employed in DDL: the Parameter Server [29] and the All-Reduce [15] architecture. In the Parameter Server architecture, all training workers connect to each shared parameter server and upload their model gradients at the end of each iteration. The servers then update the gradients and broadcasts them back to all training workers before the next iteration. If a large number of workers exist, the servers may become a bottleneck [36]. In the All-Reduce architecture, each training worker communicates with its adjacent workers in a logical topology (either ring or tree). This is done in an *All-Reduce* [35] manner, which eliminates the bottleneck of the centralized server. Compared with the Parameter Server approach, the All-Reduce architecture is more commonly used in DDL systems for its better utilization of network bandwidth [15, 22, 44]. In this paper, we therefore focus on the All-Reduce architecture.

There are many works on improving the *scaling performance* of All-Reduce DDL training. Here, we define scaling performance as the incremental speedup of introducing an extra worker. For example, gradient quantization[3, 40] and sparsification [2, 23, 31] reduce

the communication overhead by decreasing the amount of transferred gradients, at the cost of increased computation overhead and the accuracy loss. Large Batch Training [53, 54] and Asynchronous SGD [20, 52, 56] reduce the frequency of gradient synchronization to save communication overhead, which may slow down the convergence of the model. As there are multiple neural network layers of gradients, the communication of these gradients can also be scheduled via tensor fusion [41], tensor partition [18, 24], or according to the computation time of the gradients [38, 55], to alleviate the impact of communication overhead.

Despite these efforts, to date, we lack a comprehensive understanding of how key factors affect the scaling performance of DDL training, and how they can be alleviated. In this paper, we first build a recursive model, *OSF* (Scaling Factor considering Overlap), to depict the scaling factor of training, given the parameters of the neural network model and the settings of the DDL system. The *OSF* grasps two main characteristics in distributed training: the overlap between computation and communication, and batching multiple small All-Reduce operations into one. We build one DDL testbed and use it to evaluate the accuracy of *OSF* at predicting the scaling performance of various neural network models and to measure the impact of tensor fusion on the scaling performance. Last, to mitigate the impact of communication time on the scaling performance, we propose replacing the communication-intensive layers with alternative model blocks to improve the scaling performance without reducing the accuracy. We propose an adaptive tensor fusion strategy to determine whether to fuse tensors according to the FLOPs and parameter size of each layer of the model. To summarize, the contributions of this paper are as follows:

- We propose the *OSF* model (§2) to estimate the scaling performance of DDL training, given the settings of the DDL system and the parameters of neural network (NN) models. The estimation error of *OSF* is as low as 0.5%∼8.4% for different NN models, which is over 83.8% lower than that of the C2C (communication-to-computation)-based estimation method.
- We measure the scaling performance of typical NN models (§3) with breakdown analysis of the communication time and find: (1) the efficiency of communication scheduling (e.g., overlapping communication and computation, tensor fusion) depends on the internal attributes of NN layers (e.g., number of tensors, tensor size); (2) the inappropriate setting of buffer size in tensor fusion can lead to either long *ALLREDUCE* time, or increased *WAIT* time, and thus degenerate the scaling performance.
- We propose two optimizations (§4): (1) *layer replacement* that mitigates the impact of communication intensive layers, improves the scaling performance from 0.25 to 0.75 (when using 32 GPUs); (2) *adaptive tensor fusion* (`adaFusion`) that determines whether to fuse tensors according to the FLOPs and parameter size of individual layers of the model. With `adaFusion`, the scaling factor is 32.2%-150.0% higher than that using constant tensor fusion buffer size, and is also 9.0%-19.6% higher than MG-MFBP [43].

## 2 MODELING THE SCALING FACTOR

In this section we model the scaling performance of neural networks in DDL training. Throughout this paper, all neural network models

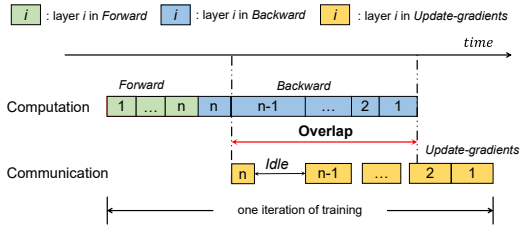



**Figure 1: The computation, communication and their overlap in DDL training**

are trained in the All-Reduce architecture. We use the *scaling factor* to represent the scaling performance of neural network models in DDL training. *Scaling factor (SF)* is defined by Equation 1:

$$SF = \frac{T_1}{T_N \cdot N}\ , \tag{1}$$

where $T_1$ represents the execution time by *one* training worker for a training job with preset datasets, models, and training targets. $T_N$ denotes the execution time of the system with $N$ workers when dealing with the same training job. For example, for a given training job, imagine 1 worker consumes 9 hours to complete the job, and 8 workers consume 1.25 hours to complete it. The scaling factor of the system with 8 workers would be 9/(1.25*8) = 0.9. The higher the scaling factor, the higher the performance of the training.

### 2.1 Scaling Factor Considering Overlap (OSF)

The DDL training time consists of roughly two parts: computation on individual workers and communication between workers. Existing work [44] has proposed the *communication to computation (C2C) Ratio* to depict the scalability of a neural network model. The higher the C2C ratio, the less scalable the model is. The *C2C-based Scaling Factor (CSF)* in the All-Reduce architecture is then deduced as in Equation 2, where $T_{comp}$ is the computation time of a model in one iteration of training, $T_{comm}$ is the communication time in one iteration of training, and $C2C$ is the ratio of communication time over computation time.

$$CSF = \frac{T_{comp}}{T_{comp} + T_{comm}} = \frac{1}{1 + C2C} \tag{2}$$

As the deep neural network model is composed of multiple layers, in the *backward* pass, the communication of one layer can start as soon as its computation completes, and the computation of the upper layer can also start simultaneously. Thus there exists overlap between computation and communication, which C2C and *CSF* omit and thus fail to predict the scaling performance.

**Defining overlap:** The overlap between computation and communication is exemplified in Figure 1. From the figure, one iteration of training in the DDL system consists of three components: *forward*, *backward* and *update-gradients* operations. Each GPU computes the gradients locally during the *forward* and *backward* stages, where *forward* pass refers to the calculation of intermediate variables and outputs for a model in the order from input layer to output layer, and *backward* pass refers the calculation of the gradient of model parameters which traverses the network in reverse order. It then executes gradient synchronization and SGD optimization during the *update-gradients* operation, which constitutes a communications overhead. When the computation thread initiates the *backward*

**Table 1: Variables in modeling Scaling Factor**

| Variable | Meaning |
|---|---|
| $N$ | the number of workers (i.e., GPUs) |
| $\beta$ | the network bandwidth between adjacent GPUs in the topology |
| $\delta$ | the communication overhead, determined by the latency between GPUs and the scale of the topology |
| $n$ | number of layers in the model |
| $D^{(i)}$ | the size of parameters of layer $i$ in the model |
| $M$ | the size of a mini-batch with SGD |
| $C^{(i)}$ | the number of float-point operations a mini-batch needs to compute in an iteration of *forward* pass for layer $i$ |
| $\pi$ | the peak computation capacity of the computing platform (in FLOPS) |
| $c$ | the coefficient that gradient operations of a model can utilize GPU |
| $s$ | the number of bytes of memory occupied by one float-point parameter |

operation, the communication thread checks whether there are gradients ready for transfer. Once they are ready, the communication thread synchronizes them with other GPUs in an all-reduce way.

Ideally, the *backward* and *update-gradients* operations should run in parallel to improve the scaling performance. This is because the less non-overlapped communication time there is, the less impact that it will have on the scaling performance. From Figure 1, when layer $n$ of a neural network model completes the *backward*, layer $n-1$ starts its *backward* operation and layer $n$ starts the *update-gradients* simultaneously. For layer $i$ ($n-1 \geq i \geq 1$), the beginning of its *update-gradients* is constrained by both the completion of the *backward* of layer $i$ and the completion of the *update-gradients* of layer $i+1$.

With the above insights in mind, we next propose the Scaling Factor considering Overlap (*OSF*) metric. Shown in Equation 3, *OSF* is defined as the ratio of computation time $T_{comp}$ *vs* the sum of the computation time and the communication time $T_{update}$, minus the overlap time, $T_{overlap}$. It is worth noting that the $T_{update}$ in *OSF* is not equal to the $T_{comm}$ in *CSF*, as $T_{update}$ contains the *idle* time between the communication of adjacent layers, as illustrated in Figure 1. We next define $T_{comp}$, $T_{update}$ and $T_{overlap}$ respectively. All the variables involved are listed in Table 1.

$$OSF = \frac{T_{comp}}{T_{comp} + T_{update} - T_{overlap}} \tag{3}$$

**Defining $T_{comp}$:** $T_{comp}$ can be calculated as the computational operations required divided by the computation capacity of one worker. Here we assume that gradients are the same size as parameters, according to [44]. The time complexity of a *backward* operation is twice that of a *forward* operation [10, 42]. Thus the computation time is described as:

$$T_{comp} = T_{forward} + T_{backward} \quad = 3 \cdot T_{forward} \tag{4}$$

The training platform uses different algorithms (e.g., GEMM, Winograd [26]) for different models to speed up the convolution operations, according to their attributes (e.g., filter size, batch size), and thus have different coefficients of GPU utilization ($c$), which can be determined through measurement in real DDL system. In all, $T_{comp}$ can be expressed as Equation 5, where $M$ represents the size of a mini-batch with SGD, $C^{(i)}$ represents the number of float-point operations a mini-batch of data requires to compute in one iteration of *forward* pass for layer $i$, and $M \cdot C^{(i)}$ is the time of computation during *forward* pass in a mini-batch for layer $i$ [16, 29].

$$T_{comp} = 3 \cdot T_{forward} = \sum_{1}^{n} \frac{3M \cdot C^{(i)}}{c\pi} \tag{5}$$

**Defining $T_{update}$:** Let $t^{(i)}_{backward}$ denote the *backward* time of layer $i$, defined in Equation 6.

$$t^{(i)}_{backward} = 2 \cdot t^{(i)}_{forward} = \frac{2M \cdot C^{(i)}}{c\pi} \tag{6}$$

Moreover, let $t_{update}(D)$ denote the *update-gradients* time of layer or tensors with size $D$, defined in Equation 7.

$$t_{update}(D) = \frac{2(N-1)}{\beta \cdot N} \cdot D + \frac{(N-1)}{N \cdot \pi \cdot s} \cdot D + \delta \ , \tag{7}$$

where $D$ represents the size of parameters in of the layer or tensors, $\frac{2(N-1)}{\beta \cdot N} \cdot D$ represents the transmission time of gradients with $(N-1)$ *Scatter-Reduce* iterations and $(N-1)$ *AllGather* iterations, $\frac{(N-1)}{N \cdot \pi \cdot s} \cdot D$ signifies the computation time of gradients aggregation in $(N-1)$ *Scatter-Reduce* iterations, $s$ represents the number of bytes each float point parameter occupies, and $\delta$ is the communication overhead in each All-Reduce operation, which can be measured directly given the DDL system.

We use $\tau^{(i)}$ to denote the start time of *update-gradients* of layer $i$, which is defined in a recursive way. For layer $n$, the start time of *update-gradients* $\tau^{(n)}$ is 0. For layer $i$ ($n > i \geq 1$), the start time of *update-gradients* $\tau^{(i)}$ is no earlier than the end of *update-gradients* of layer $i+1$ (denoted as $\tau^{(i+1)} + t_{update}(D^{(i+1)})$) as well as the accumulated *backward* time from layer $n-1$ to layer $i$ (denoted as $\sum_{i}^{n-1} t^{(j)}_{backward}$). The recursive definition of $\tau^{(i)}$ is shown in Equation 8.

$$\tau^{(i)} = \begin{cases} 0 \ , & \text{if } i = n \\ \max(\tau^{(i+1)} + t_{update}(D^{(i+1)}), \sum_{i}^{n-1} t^{(j)}_{backward}) \ , & \text{otherwise} \end{cases} \tag{8}$$

Finally, according to the definition, we have:

$$T_{update} = \tau^{(1)} + t_{update}(D^{(1)}) \ . \tag{9}$$

**Defining $T_{overlap}$:** For the *update-gradients* of one iteration, the beginning cannot be earlier than the completion of *backward* of layer $n$. Further, the completion cannot be earlier than the completion of *backward* of layer 1. Considering that the *backward* operation of one iteration is executed continuously, the overlap time $T_{overlap}$ is estimated to be the sum of the *backward* time of layers from $n-1$ to 1, as shown in Equation 10. This gives us an estimate of the scaling performance, with the overlap taken into consideration.

$$T_{overlap} = \sum_{1}^{n-1} t^{(i)}_{backward} \tag{10}$$

By replacing the $T_{comp}$, $T_{update}$, and $T_{overlap}$ in Equation 3 with Equation 5, 9, 10 respectively, the scaling factor of any neural network model given the settings of the DDL system can be estimated with our OSF model.

## 2.2 OSF under Tensor Fusion

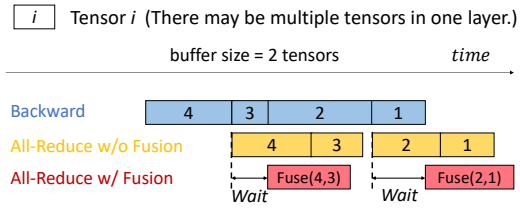



**Figure 2: Example of tensor fusion in DDL training**

The *OSF* defined above only considers the scenario in which All-Reduce operations take place immediately after the gradients of a layer have been calculated. During DDL training, the communication scheduler could also batch multiple small All-Reduce operations into one to reduce communication overhead. This is called tensor fusion, as illustrated in Figure 2. The buffer size determines how many tensors should be fused together for one iteration of All-Reduce communication. In the example, the buffer size is set to 2 tensors for ease of illustration. For Tensor 4 and Tensor 3, if they are fused together, the total communication time is reduced, because the saved time in one iteration of All-Reduce is larger than the *WAIT* time. However, for Tensor 2 and Tensor 1, if they are fused together, the total communication time is increased because the *WAIT* time is longer than the saved *ALLREDUCE* time by fusion. Thus, we next model the impact of tensor fusion on the scaling performance, as shown in Equation 11:

$$OSF_{fusion} = \frac{T_{comp}}{T_{comp} + T_{update}^{fusion} - T_{overlap}} , \tag{11}$$

where $T_{comp}$ and $T_{overlap}$ have the same definition as those in Equation 3. Next, we formulate the communication time $T_{update}^{fusion}$ under tensor fusion to model its impact on the scaling performance.

There are two main parameters in tensor fusion which impact the performance of distributed training: fusion buffer size $Q$, which is the maximum amount of tensors transmitted in one all-reduce communication, and timeout $timeout$, which is the longest time to wait for the tensors of the upper layer. For each layer, after its backward pass completes, the tensors are put into the buffer for fused transmission. When the buffer is full or a timeout occurs, the tensors in the buffer are transferred in an all-reduce manner and the buffer is cleared. The computation time and overlap time in DDL training are not affected by tensor fusion.

For each layer $i$, after its *backward* pass completes, the communication scheduler tries to put the tensors of this layer in the fusion buffer as long as the buffer capacity allows. Thus, the size of buffered tensors after putting tensors of layer $i$ in the buffer, denoted as $B^{(i)}$, is defined in Equation 12.

$$B^{(i)} = \begin{cases} D^{(i)} , & \text{if } i = n \text{ or } B^{(i+1)} + D^{(i)} > Q \text{ or } t_{backward}^{(i)} \geq timeout \\ B^{(i+1)} + D^{(i)} , & \text{otherwise} \end{cases} \tag{12}$$

When the tensors of layer $i$ are buffered for fusion, the communication scheduler decides to transfer the buffered tensors, if one of the following conditions satisfies:

- This is layer 1, meaning there are no more layers in this iteration;
- A timeout occurs before the *backward* pass of layer $i-1$ completes;
- The buffer does not have enough space to store the tensors of layer $i-1$.

Otherwise, the tensors of the current layer are stored in the buffer. Let $F^{(i)}$ denote the size of transferred tensors after the completion of *backward* pass of layer $i$, which is defined in Equation 13.

$$F^{(i)} = \begin{cases} B^{(i)} , & \text{if } i = 1 \text{ or } t_{backward}^{(i)} > timeout \text{ or } B^{(i)} + D^{(i-1)} > Q \\ 0 , & \text{otherwise} \end{cases} \tag{13}$$

For layer $i$, when the communication scheduler decides to transfer the tensors (up to layer $i$) in the buffer, the tensors of layer $i$ may have been buffered for a short duration, denoted as $t_{wait}^{(i)}$, which is defined in Equation 14.

$$t_{wait}^{(i)} = \begin{cases} timeout , & \text{if } i > 1 \text{ and } t_{backward}^{(i-1)} > timeout \\ t_{backward}^{(i-1)} , & \text{else if } i > 1 \text{ and } B^{(i)} + D^{(i-1)} > Q \\ 0 , & \text{otherwise} \end{cases} \tag{14}$$

For layer $i$, given the amount of transferred tensors $F^{(i)}$, the wait time before transmission $t_{wait}^{(i)}$ and the duration of *backward* pass $t_{backward}^{(i)}$, the start time of tensor transfer is no earlier than the end time of tensor transfer of layer $i+1$ as well as the completion time of *backward* pass of layer $i$. Thus, the start time of the tensor transfer of layer $i$, denoted as $\tau^{(i)}$, is defined in Equation 15, where $t_{update}(F^{(i+1)})$ is the time required for all-reduce operation of buffered tensors $F^{(i+1)}$.

$$\tau^{(i)} = \begin{cases} 0 , & \text{if } i = n \\ \max(\tau^{(i+1)} + t_{wait}^{(i+1)} + t_{update}(F^{(i+1)}), \sum_{i}^{n-1} t_{backward}^{(j)}), & \text{otherwise} \end{cases} \tag{15}$$

Summing up the above, the overall communication time under tensor fusion is defined in Equation 16:

$$T_{update}^{fusion} = \tau^{(1)} + t_{update}(F^{(1)}) \tag{16}$$

By replacing the $T_{update}^{fusion}$ in Equation 11 with Equation 16, we can calculate the scaling factor of any neural network model, given the settings of the DDL system, as well as the parameters of tensor fusion.

# 3 MEASURING SCALING PERFORMANCE

In this section, we examine the accuracy of OSF in estimating the scaling performance and evaluate the impact of various factors on the scaling performance of DDL training. To this end, we first build a DDL system which is representative of the specification and configuration of DDL systems used by web applications [4, 5, 7].

## 3.1 Measurement Setup

**Hardware:** We conduct our experiments on a cluster of 4 machines connected with 25Gbps. Each machine is equipped with 8 Nvidia GeForce RTX 2080 Ti GPU and 1 Mellanox ConnectX-4 Lx NIC.

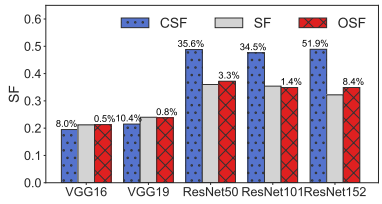



**Figure 3: The *CSF*, *SF* and *OSF* of different models**

Each GPU has 11 GB of memory and connected via PCIe 3.0 x16 in a machine.

**Software:** TensorFlow is used for training the model locally, and Horovod, a widely used platform for distributed deep learning training, is used for the synchronization of model gradients among workers. The communication across training workers is organized in a ring All-Reduce architecture, which is supported by communication library NVIDIA NCCL. The software version used in experiments are Horovod 0.20.0, TensorFlow 2.3.0, NCCL 2.8.4. Docker with images horovod/horovod:0.20.0-tf2.3.0-py3.7-cuda10.1 is used for the quick deployment of the software system.

We select five representative neural network models: ResNet50, ResNet101, ResNet152 [19], VGG16 [46] and VGG19. The parameters and related information of the models are listed in Table 2. We adopt synthetic dataset, which is more suitable than real-world dataset when measuring the scaling performance, as random generated samples and labels eliminate the elements which are irrelevant to the measurements. In the Synthetic dataset, images with size 224x224x3 (the same as that in ImageNet [11] and classification labels are generated randomly, according to the guideline of Horovod [21].

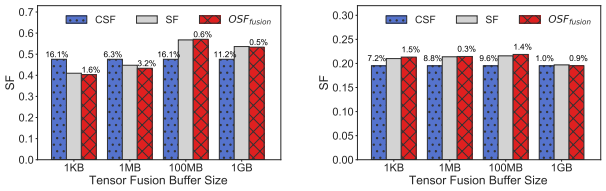
**Table 2: Benchmark Models**

| Name | GFLOPs | Parameter Size (MB) | # Layers | # Tensors |
|---|---|---|---|---|
| ResNet50 | 4 | 98 | 50 | 176 |
| ResNet101 | 8 | 170 | 101 | 346 |
| ResNet152 | 12 | 230 | 152 | 516 |
| VGG16 | 16 | 528 | 16 | 22 |
| VGG19 | 20 | 549 | 19 | 24 |

## 3.2 Measuring the Accuracy of OSF

We first evaluate the accuracy of the proposed *OSF* model. Figure 3 shows the *SF*, *CSF* and *OSF* for different models trained on our testbed. Here, we fix the number of GPUs at 32 and the tensor fusion is turned off. In the figure, the number on each bar of *SF* represents the measured scaling factor; the number on each bar of *CSF* and *OSF* represents their estimation error relative to SF, respectively. We see that the estimation error of *OSF* ranges from 0.5% to 8.4%, much low than that by *CSF* (ranging from 8% to 51.9%). Specifically, in comparison with CSF, OSF reduces the estimation error by over 83.8%. The results demonstrate that the efficiency of *OSF* modeling in estimating the scaling performance given neural network models and DDL system settings.

Another interesting observation is that the *CSF* value of ResNet models is higher than *OSF* and *SF* values, indicating that the overlap between communication and computation (captured by *OSF*) degenerates the scaling performance of ResNet models. Comparing *CSF* and *OSF*, all tensors in *CSF* are transferred in one All-Reduce operation to eliminate the communication overhead ($\delta$), all tensors in one layer is transferred immediately after the *backward* pass completes, which eliminates the *Wait* time for the *backward* pass of layer 1. From Table 2, ResNet models have many tiny layers and tensors. For such models, the "communication after all computation" (on which *CSF* is built) has higher communication efficiency than the "communication immediately after one computation" (on which *OSF* is built). It also demonstrates the usefulness of tensor fusion, which will be investigated later in this section.




**(a) ResNet101** **(b) VGG16**

**Figure 4: The *CSF*, *SF* and $OSF_{fusion}$ of models under different tensor fusion buffer sizes**

Next, we evaluate the accuracy of our OSF model when tensor fusion is turned on. Figure 4 shows the *CSF*, *SF* and $OSF_{fusion}$ values of ResNet101 and VGG16 under different tensor fusion buffer sizes. For ResNet101 in Figure 4a, we see $OSF_{fusion}$ obtains a maximum estimation error (3.2%), much lower than that by *CSF* (16.1%), which demonstrates the accuracy of $OSF_{fusion}$ on modeling the scaling performance under tensor fusion. Note that in this set of experiments, *CSF* fuses all tensors into one All-Reduce operation. When the buffer size is 1KB or 1MB, the *CSF* value is higher than that of *SF* and $OSF_{fusion}$; but when the buffer size is 100MB or 1GB, the *CSF* value is lower than that of *SF* and $OSF_{fusion}$. This observation indicates that the fusion buffer size needs to carefully tuned for better scaling performance, which will be investigated in the following section.

For VGG16 in Figure 4b, the estimation error of *CSF* is comparable with that of $OSF_{fusion}$, meaning that the overlap between computation and communication has little impact on the scaling performance. Moreover, when the tensor fusion buffer size is less than 1GB, the *SF* and $OSF_{fusion}$ values keep nearly unchanged, meaning that tensor fusion has a negligible impact on its scaling performance. These two phenomena can be explained by that one single fully-connected layer of VGG16 occupy about 74.2% of the total communication time, which is not affected by the overlap between computation and communication or tensor fusion. When the buffer size is set to 1GB, the *SF* and $OSF_{fusion}$ values decrease slightly, due to the increased *WAIT* time by tensor fusion, which will be investigated next. Nevertheless, $OSF_{fusion}$ can always estimate the *SF* value under tensor fusion with pretty low error.

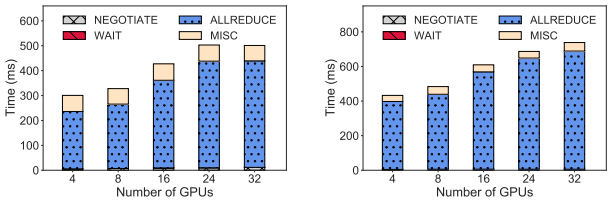
## 3.3 Measuring the Impact of Tensor Fusion

The rationale of tensor fusion is to batch multiple small All-Reduce operations into one to decrease the *ALLREDUCE* time, at the cost of increasing *WAIT* time. To investigate the impact of tensor fusion on

the scaling performance, we first break down the communication time into different constituents.

**Decomposing Communication Time:** The communication time in DDL training is composed of four parts, namely *NEGOTIATE*, *ALLREDUCE*, *WAIT* and *MISC*. During training, the computation graph executed by TensorFlow results in different orders of tensor transfer among multiple GPUs. Thus, there is one GPU as the master (which is responsible for tensor ordering). Once gradients of one layer are ready in *backward*, the GPU sends the tensor information to the master. The master determines the order of tensors for transfer and reports the order to workers. The stage used for determining the order of tensor transfers among GPUs is called *NEGOTIATE*.

Following this, all involved GPUs synchronize the tensors (i.e., gradients) via All-Reduce communication. This phase is named *ALLREDUCE*. If tensor fusion is enabled, after receiving the response from master, each GPU can also wait (buffer) for the fusion with other tensors before transmission. This phase is named *WAIT*. There is also the *MISC* phase (including memory copy, waiting for computation), which occupies a relatively small and constant communication time.

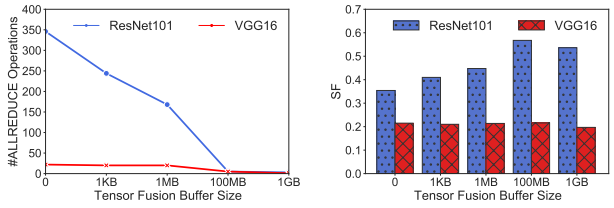



(a) ResNet101 (b) VGG16

**Figure 5: Breakdown of communication time of ResNet101 and VGG16 under different number of GPUs**

Next, we measure the proportion of each phase, to investigate how the scaling performance is affected by the four constituents of the communication time. Figure 5 shows the breakdown of communication time of ResNet101 and VGG16. For ResNet101 in Figure 5a, when increasing the number of GPUs, *ALLREDUCE* occupies 88% of communication time under 32 GPUs. Compared with ResNet101, VGG16 always has a higher proportion of *ALLREDUCE* and a lower proportion of *MISC*. This is due to the larger parameter size of the (communication intensive) VGG16 model. We also see that, for these models, the *NEGOTIATE* time among multiple GPUs is negligible, contradicting prior observations in [39], where long spin waits [6] in *NEGOTIATE* causes significant slowdown. In our DDL system, the equal computing capacity of GPUs and the use of synthetic datasets eliminates the imbalance of computing and workload respectively, leading to relatively short *NEGOTIATE* time. From this, we conclude that GPUs with equal computing capacity improve distributed training, as the *NEGOTIATE* time (as well as the scaling performance) is determined by the slowest worker.

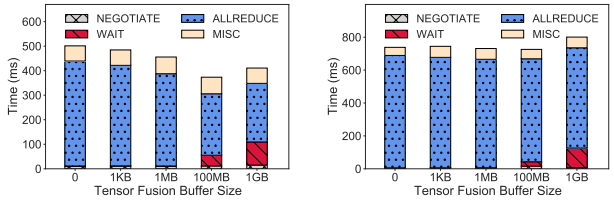
**Measuring impact of tensor fusion buffer size:** We start by measuring the number of All-Reduce operations in one iteration of training and the scaling factor of ResNet101 and VGG16 with various buffer sizes running on 32 GPUs, as shown in Figure 6. For ResNet101, there are a number of small tensors in the convolution layers. When the buffer size increases from 0 to 100MB, the number




(a) #(All-Reduce operations) (b) scaling performance

**Figure 6: The impact of tensor fusion buffer size**

of All-Reduce operations drops dramatically in Figure 6a. This leads to the growth of scaling factor in Figure 6b. When the buffer size increases to 1GB, the scaling factor of ResNet101 starts to decrease (although still 40% more than that without tensor fusion). In contrast, for VGG16, the scaling factor remains almost unchanged when the buffer size increases from 0 to 100MB. Since VGG16 has fewer layers, the number of All-Reduce operations is also relatively small, shown in Figure 6a. When the buffer size is set to 1GB, the scaling factor of VGG16 decreases by 9%, compared with that without tensor fusion.




(a) ResNet101 (b) VGG16

**Figure 7: Breakdown of communication time of ResNet101 and VGG16 with tensor fusion**

The difference of scaling factor when varying the buffer size comes from the fact that the tensor fusion has an impact on the tradeoff between *ALLREDUCE* time and *WAIT* time. Next, we profile the communication time of ResNet101 and VGG16 with various buffer sizes and investigate its impact, as shown in Figure 7. For ResNet101 in Figure 7a, the communication time decreases when the buffer size increases from 0 to 100MB, since the fusion of tiny tensors effectively reduces the *ALLREDUCE* time. As the buffer size continues to increase, the *WAIT* time increases correspondingly, which degrades the scaling factor. In contrast, for VGG16 in Figure 7b, when the buffer size increases to 100MB, the *ALLREDUCE* time is nearly unchanged, and the *WAIT* time is almost 0. When the buffer size is increased to 1GB, the largest tensor could be fused with other tensors, which saves *ALLREDUCE* time, at the cost of greatly increased *WAIT* time, which makes the communication time 8% more than that without tensor fusion.

## 3.4 Summary of Results

The proposed *OSF* model has much lower estimation error than the C2C-deduced *CSF* model on modeling the scaling performance of DDL training, as our *OSF* model precisely capture the overlap between computation and communication, and batching update. The *OSF* model can be leveraged by system and neural network model designers to estimate and improve the scaling performance of DDL systems.

**Table 3: The related layers of VGG16 and VGG16 with layer replacement**

| Model | Layer | Structure | Parameter Size (MB) |
|---|---|---|---|
| VGG16 | Fully Connected Layer 1 | 4096 | 392 |
| | Fully Connected Layer 2 | 4096 | 64 |
| | Fully Connected Layer 3 | 1000 | 15.6 |
| VGG16 w/ layer replacement | Convolution Layer 1 | (1,1,512) | 1 |
| | Convolution Layer 2 | (1,1,1000) | 1.95 |
| | Global Average Pooling Layer | Pooling | 0 |

We use this to better understand scaling performance. We find that for neural network models with many tiny tensors (e.g., ResNet101), overlapping computation and communication may result in worse scaling performance than the "Communication after all computation" strategy. Besides, for neural network models with one single large tensor (e.g., VGG16), tensor fusion has a negligible impact on its scaling performance. Moreover, the inappropriate setting (e.g., using default values) of buffer size in tensor fusion can further lead to either a long ALLREDUCE time, or increased WAIT time.

## 4 IMPROVING SCALING PERFORMANCE

Motivated by our analysis in the previous section, we improve the scaling performance by using two mechanisms. The first mechanism (called layer replacement) aims at mitigating the impact of communication intensive layers (with large parameter sizes) on the scaling performance. The second mechanism (called adaptive tensor fusion, `adaFusion`) enables appropriate setting of buffer size in tensor fusion.

### 4.1 Layer Replacement

Motivated by the network-in-network method in [30], we propose to replace the fully connected layers with convolution layers and a global average pooling layer. Table 3 illustrates the replacement for VGG16.

We measure the efficacy of this approach, and present the scaling performance in Figure 8. We train VGG16 and VGG16 with layer replacement with ImageNet dataset [11], and find that the scaling factor after layer replacement is much improved. When training with 32 GPUs, the scaling factor is increased from 0.25 to 0.72. We therefore confirm that layer replacement offers an effective strategy for improving scaling performance for communication heavy layers. Moreover, the training loss of VGG16 and VGG16 with layer replacement are respectively 1.82 and 1.39 after 120 epochs, as shown in Figure 9, indicating that layer replacement does not hurt model accuracy during training.

### 4.2 Adaptive Tensor Fusion

Based on the measurement in Section 3.3, the constant tensor fusion strategy fails to obtain optimal results for different models and tensors. Smaller buffer sizes introduce extra communication overhead, while larger ones might result in long wait times. Thus, we present an adaptive tensor fusion (`adaFusion`) strategy, which dynamically determines whether to fuse tensors during distributed training. The rationale of `adaFusion` is that we only fuse the tensor

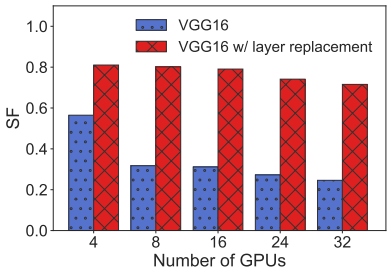



**Figure 8: Scaling factor of VGG16 and VGG16 with layer replacement under different number of GPUs**

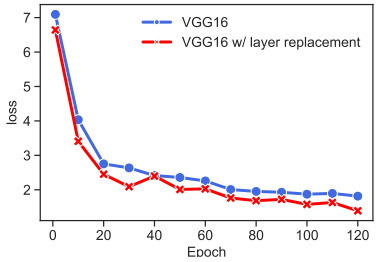



**Figure 9: Training loss of VGG16 and VGG16 with layer replacement**

when the sum of the waiting time and communication time of fused tensors are smaller than the communication time of transferring the tensors separately. That is, tensors of layer $i$ is fused only when

$$t_{update}(B^{(i+1)}+D^{(i)})+t_{wait}^{(i)} < t_{update}(B^{(i+1)})+t_{update}(D^{(i)}) \quad . \quad (17)$$

**Estimating Communication Time:** `adaFusion` relies on the accurate estimation of *ALLREDUCE* time. Traditional methods [43] estimate the communication time by assuming that it has linear relationship with data size given constant bandwidth. This assumption is only reasonable when the dataset is large enough to saturate the bandwidth, which always underestimates the communication time of the pervasive small tensors in DDL training.

Considering the impact of slow (re-)start in RDMA [32], we adopt a piecewise function to estimate the communication time of different parameter sizes. For a large parameter size, where the communication time is relatively longer and mostly associated with stable bandwidth, a linear function is used to fit the data. For a small parameter size, a logarithmic function is adopted to mimic the behavior of the increasing window during slow start.

Consequently, the communication time given data size is expressed as:

$$t'_{update}(D) = \begin{cases} a_1 \times \log_2(D) + b_1 \ , \text{if } D < D_{Thresh} \\ a_2 \times D + b_2 \ , \qquad \text{otherwise} \end{cases} \quad (18)$$

The value of $D_{Thresh}$ is set to the Bandwidth-Delay Product (BDP) of the link in DDL system, denoting the maximum amount of data that senders could transmit in unit time (one RTT).

To evaluate the precision of our communication time prediction model, we measure the communication time of tensors in different models among 32 GPUs with constant tensor fusion on Horovod. Our measurements show that 53.1% of tensors are smaller than the

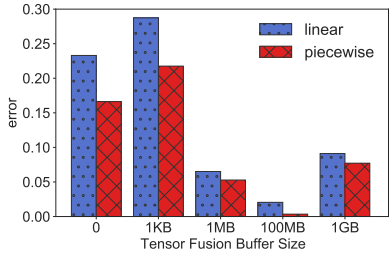



**Figure 10: Prediction error of *piecewise* and *linear***

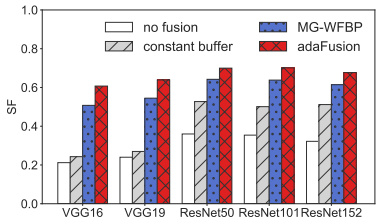



**Figure 11: Scaling performance of different fusion strategies**

BDP (45KB in our experiment), and it confirms that accurate estimation of small tensors communication is necessary for predicting total communication time.

We select different buffer size configurations in constant tensor fusion to evaluate prediction performance in a wide range of data size. The prediction error of the estimated communication time is defined as: $\frac{T_{estimated}-T_{measure}}{T_{measure}}$.

The prediction error of *piecewise* and *linear* functions for the communication time of tensors in ResNet101 is shown in Figure 10. ResNet101 has multiple layers with small tensors, whose communication time is heavily affected by tensor fusion strategy. Therefore, estimation of its communication time is useful for evaluating precision of communication models. From the figure, as expected, the piecewise model has a much lower prediction error than the linear method, especially for the tensors are small.

Given the DDL system and the training model, the computation time of an individual layer can be calculated directly by Equation 5, the communication time of each layer and the communication time if the tensors are fused can be estimated by Equation 18 with higher accuracy. `adaFusion` decides whether to fuse the tensors of each layer, according to the result of Equation 17.

Figure 11 shows the scaling performance of different fusion strategies. In the figure, the buffer size of constant tensor fusion strategy is set to 100MB, under which all the five models have the highest scaling factor. *MG-WFBP* [43] simply decides whether to fuse the tensors of layer $i$ by comparing the time required for *backward* pass of layer $i-1$ with the communication time of tensors of layer $i$ plus a small value which represents the communication overhead of one All-Reduction operation. We can observe from the figure that for all the considered models, the scaling factor with our proposed `adaFusion` is 32.2%-150% higher than that with constant tensor fusion size, and is 9.0%-19.6% higher than that with MG-MFBP. The advantage of `adaFusion` comes from that `adaFusion` can estimate the communication time of fused tensors with higher accuracy while MG-WFBP always underestimates the communication time of small tensors.

## 5 RELATED WORK

**Modeling and Measurement of DDL training:** Previous works [9, 45, 49, 50, 57] theoretically analyze the communication overhead introduced by the All-Reduce topology. The communication-to-computation (C2C) ratio [51] heavily affects the scalability of DDL systems, and a higher C2C ratio results in lower scaling efficiency [43]. Shi et al. [44] defined model intensity, and found a higher intensity and lower C2C ratio make the model easier to be parallelized. Nathan et al. [47] characterized model workloads by activations and parameters, then designed controllable computation and communication intensities in ResNet models. Some works have mentioned the significance of overlap in the analysis of the scaling factor [18, 38, 43, 45, 57], especially in the Parameter Server architecture [28].Recently pipeline parallelism [34] is proposed to partition model layers into multiple stages. In contrast to the works above that treat neural network models as a whole, our proposed *OSF* captures the impact of the internal attributes of the model as well as the settings of the DDL system on the scaling performance.

**Optimizing the communication performance in DDL training:** There are a number of communication libraries that are designed to develop and optimize DDL, including Message Passing Interface (MPI) [48], NVIDIA's NCCL [35], and Facebook Gloo [13]. Horovod [41] is a DDL framework that relies on the above communication libraries, and is easier to inter-operate with many DL frameworks, e.g., TensorFlow [1], PyTorch [37] and MXNet [8]. There are some limitations which degrade the scaling performance in practice though. Pumma et al. [39] found that TensorFlow and Horovod have resource contention, resulting in load imbalance and longer training times. For inter-node communication, Wang et al. [50] focus on the impact of network topology on DML performance and Geng et al. [14] propose a server-centric network topology to better utilize bandwidth among servers. Different from these works, we perform breakdown analysis of communication time and identify several opportunities that can be leveraged to improve the scaling performance of DDL training.

## 6 CONCLUSION

We have proposed the *OSF* model to capture the impact of various elements on the scaling performance of DDL training, as well as proposed solutions that can mitigate the issues observed. These solutions can be leveraged for better model design and improved DDL training performance in web applications. We emphasize that this work explores only a subset of the challenges. Hence, our future work will involve measuring other factors, such as the impact of dataset characteristics. We further intend to implement our proposed solutions to evaluate the potential for improving scaling performance in-the-wild.

## ACKNOWLEDGMENTS

This work was partially supported by National Key R&D Program of China (2019YFB1802800), the Informatization Plan of Chinese Academy of Sciences (CAS-WX2021SF-0506), National Natural Science Foundation of China (U20A20180, 62072437) and Beijing Natural

Science Foundation (JQ20024). Corresponding authors: Qinghua Wu & Zhenyu Li.

## REFERENCES


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).

[3] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. *Advances in Neural Information Processing Systems* 30 (2017), 1709–1720.

[4] Mario Almeida, Stefanos Laskaridis, Ilias Leontiadis, Stylianos I Venieris, and Nicholas D Lane. 2019. EmBench: Quantifying performance variations of deep neural networks across modern commodity devices. In *The 3rd international workshop on deep learning for mobile systems and applications*. 1–6.

[5] Cody J Blakeney, Xiaomin Li, Yan Yan, and Ziliang Zong. 2020. Parallel Blockwise Knowledge Distillation for Deep Neural Network Compression. *IEEE Transactions on Parallel and Distributed Systems* (2020).

[6] Johann Blieberger, Bernd Burgstaller, and Bernhard Scholz. 2003. Busy wait analysis. In *International Conference on Reliable Software Technologies*. Springer, 142–152.

[7] Tianyi Chen, Ziye Guo, Yuejiao Sun, and Wotao Yin. 2021. CADA: Communication-Adaptive Distributed Adam. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 613–621.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[9] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. 2019. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development* 63, 6 (2019), 1–1.

[10] Danny Hernandez Dario Amodei. [n. d.]. AI and Compute. [EB/OL]. https://openai.com/blog/ai-and-compute/ Accessed May 1, 2021.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

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

[13] FaceBook. [n. d.]. Gloo. https://github.com/facebookincubator/gloo

[14] Jinkun Geng, Dan Li, Yang Cheng, Shuai Wang, and Junfeng Li. 2018. HiPS: Hierarchical parameter synchronization in large-scale distributed machine learning. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*. 1–7.

[15] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[17] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 6645–6649.

[18] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. 2018. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288* (2018).

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

[20] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B Gibbons, Garth A Gibson, Gregory R Ganger, and Eric P Xing. 2013. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in neural information processing systems* 2013 (2013), 1223.

[21] Horovod. 2018. *Horovod Synthetic Benchmark*. https://github.com/horovod/horovod/tree/master/examples/tensorflow2

[22] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2592–2600.

[23] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Vladimir Braverman, Ion Stoica, and Raman Arora. 2019. Communication-efficient distributed SGD with sketching. *arXiv preprint arXiv:1903.04488* (2019).

[24] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. *arXiv preprint arXiv:1905.03960* (2019).

[25] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205* (2018).

[26] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4013–4021.

[27] Mu Li. 2017. *Scaling distributed machine learning with system and algorithm co-design*. Ph. D. Dissertation. PhD thesis, Intel.

[28] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[29] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 661–670.

[30] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400* (2013).

[31] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[32] Mellanox. 2021. *How to Enable Disable Lossy RoCE Accelerations*. https://community.mellanox.com/s/article/How-to-Enable-Disable-Lossy-RoCE-Accelerations

[33] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. 2018. Massively distributed SGD: ImageNet/ResNet-50 training in a flash. *arXiv preprint arXiv:1811.05233* (2018).

[34] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[35] NVIDIA. 2017. NVIDIA collective communications library (NCCL). https://developer.nvidia.com/nccl/

[36] Heng Pan, Zhenyu Li, JianBo Dong, Zheng Cao, Tao Lan, Di Zhang, Gareth Tyson, and Gaogang Xie. 2020. Dissecting the Communication Latency in Distributed Deep Sparse Learning. In *Proceedings of the ACM Internet Measurement Conference*. 528–534.

[37] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[38] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[39] Sarunya Pumma, Daniele Buono, Fabio Checconi, Xinyu Que, and Wu-chun Feng. 2020. Alleviating Load Imbalance in Data Processing for Large-Scale Deep Learning. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 262–271.

[40] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*.

[41] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[42] Serebryakov Sergey. 2019. Neural network runtime characteristics. https://github.com/sergey-serebryakov/nns.

[43] Shaohuai Shi, Xiaowen Chu, and Bo Li. 2021. MG-WFBP: Merging Gradients Wisely for Efficient Communication in Distributed Deep Learning. *IEEE Transactions on Parallel and Distributed Systems* 32, 8 (2021), 1903–1917.

[44] Shaohuai Shi, Zhenheng Tang, Xiaowen Chu, Chengjian Liu, Wei Wang, and Bo Li. 2020. Communication-Efficient Distributed Deep Learning: Survey, Evaluation, and Challenges. *arXiv preprint arXiv:2005.13247* (2020).

[45] Shaohuai Shi, Zhenheng Tang, Xiaowen Chu, Chengjian Liu, Wei Wang, and Bo Li. 2020. A Quantitative Survey of Communication Optimizations in Distributed Deep Learning. *IEEE Network* (2020).

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

[47] Nathan R Tallent, Nitin A Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie. 2017. Evaluating on-node gpu interconnects for deep learning workloads. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 3–21.

[48] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[49] Yuichiro Ueno and Rio Yokota. 2019. Exhaustive study of hierarchical allreduce patterns for large messages between GPUs. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 430–439.

[50] Shuai Wang, Dan Li, Jinkun Geng, Yue Gu, and Yang Cheng. 2019. Impact of network topology on the performance of DML: Theoretical analysis and practical factors. In *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 1729–1737.

[51] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *arXiv preprint arXiv:1705.07878* (2017).

[52] Yemao Xu, Dezun Dong, Yawei Zhao, Weixia Xu, and Xiangke Liao. 2020. OD-SGD: One-Step Delay Stochastic Gradient Descent for Distributed Training. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–26.

[53] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888* 6 (2017), 12.

[54] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[55] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 181–193.

[56] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2015. Staleness-aware async-sgd for distributed deep learning. *arXiv preprint arXiv:1511.05950* (2015).

[57] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. 2020. Is network the bottleneck of distributed training?. In *Proceedings of the Workshop on Network Meets AI & ML*. 8–13.