Geryon: Accelerating Distributed CNN Training by Network-Level Flow Scheduling

***Abstract*—Increasingly rich data sets and complicated models make distributed machine learning more and more important. However, the cost of extensive and frequent parameter synchronizations can easily diminish the benefits of distributed training across multiple machines. In this paper, we present Geryon, a network-level flow scheduling scheme to accelerate distributed Convolutional Neural Network (CNN) training. Geryon leverages multiple flows with different priorities to transfer parameters of different urgency levels, which can naturally coordinate multiple parameter servers and prioritize the urgent parameter transfers in the entire network fabric. Geryon requires no modification in CNN models and does not affect the training accuracy. Based on the experimental results of four representative CNN models on a testbed of 8 GPU (NVIDIA K40) servers, Geryon achieves up to 95.7% scaling efficiency even with 10GbE bandwidth. In contrast, for most models, the scaling efficiency of vanilla TensorFlow is no more than 37% and that of TensorFlow with parameter partition and slicing is around 80%. In terms of training throughput, Geryon enhanced with parameter partition and slicing achieves up to 4.37x speedup, where the flow scheduling algorithm itself achieves up to 1.2x speedup over parameter partition and slicing.**

I. Introduction

Nowadays, Artificial Intelligence (AI) and Machine Learning (ML) have achieved great success in a variety of applications, including image classification, machine translation and speech recognition [1]–[5]. Considering the increasing volume of training data and the growing complexity of training models, Distributed Machine Learning (DML), in particular DML with data parallelism, has become a common practice to train large amounts of data in an acceptable amount of time [6]–[12].

However, as the scale of DML cluster grows, communication can become a bottleneck that constrains the speed of training. Since it is quite difficult, if not impossible at all, to obtain the analytical solution directly, ML algorithms approach the optimal solution by iterative refinement. Stochastic Gradient Descent (SGD) is one typical method in use. During every iteration, each worker first pulls the updated parameters, and then calculates gradients based on their training data portion. Finally the gradients from different workers are aggregated together to update the model parameters. With rapid development of hardware accelerators on the computing side, such as GPUs and FPGAs, frequent parameter/gradient exchanges can easily make network side the bottleneck, and thus diminish the benefits of distributed training across machines [6], [7], [13].

To cope with expensive cost of parameter synchronization, high-speed and low-latency network technologies, such as Remote Direct Memory Access (RDMA), have been continually developed to reduce the transmission time, but still fall far behind the upgrading speed of computation hardware. For example, network bandwidth has increased from 10GbE to 100GbE since 2012 [14]. But GPU performance has increased at least 35 times during the same period [15]. And such a trend exacerbates the mismatch between computation and communication performance. Besides, the size of training model is growing rapidly in the past decades [16]–[18]. Dating back to the year of 2012, the typical neural network model only has less than 10 layers, whereas the recent model has reached even more than 1000 layers [17]. The growing size of training model indicates more parameters to synchronize during iterations, and it also imposes heavier burden to the network capability and further delays the training process.

In order to accelerate DML with relatively underdeveloped network capability, one effective way is to improve the overlap between computation and communication, and hide the communication overheads behind computation. Some recent work attempts to achieve this by explicitly scheduling the order of parameter transfers at the end host. For instance, TICTAC [19] obtains near-optimal scheduling by analyzing critical path of the model computation graph, then the parameters are handed to the communication module in that order. P3 [20] slices parameters and assigns a priority for them based on the order of consumption. The slice with the highest priority in the end host queue is sent out first. However, such solutions only control the sending order at each end host, while ignore the overall situation of the network, which is undesirable in practical setting with multiple senders/receivers. More specifically, the end-host-based solution can only exert the prioritization on the data packets of the same node, but cannot coordinate well the traffic from different nodes. When there are multiple nodes sending traffic to the network, which is especially common in large-scale DML, the low-priority packets from one sender will contend the bandwidth resource with the high-priority packets from another sender, hence causing the prioritization scheduling to lose its effect.

In this paper, we propose Geryon, a network-level flow scheduling scheme, to accelerate distributed Convolutional Neural Network (CNN) training. The key idea behind Geryon is to leverage multiple flows with different priorities to transfer parameters of different urgency levels. We say a parameter is more *urgent* if it is needed earlier for computation, and we will transfer it with a higher-priority flow to the end host. Since the high-priority tag can be identified across the entire network, including both the NICs and the switches, the transfer of the urgent parameters can be better guaranteed, compared to the simple end-host-level solutions.

We implement Geryon in TensorFlow and evaluate Geryon by extensive experiments. We run four representative CNN models, trained on a testbed of 8 GPU (NVIDIA K40) servers. The results show that Geryon can significantly improve the overlap between computation and communication, especially when cooperated with parameter partition and slicing. To be specific, Geryon can achieve up to 95.7% scaling efficiency even with 10GbE bandwidth. In contrast, for most models, except ResNet-50 which is computation hungry, vanilla TensorFlow can only achieve the scaling efficiency of no more than 37%. Moreover, even for TensorFlow that adopts parameter partition and fine-grained parameter slicing, the scaling efficiency is only around 80%. In terms of training throughput, Geryon can achieve up to 1.2x speedup over TensorFlow that adopts parameter partition and slicing. If compared to the vanilla TensorFlow, Geryon enhanced with parameter partition and slicing can achieve up to 4.37x speedup.

II. Background and Motivation

A. Distributed CNN

As shown in Figure 1, CNN is a typical category of Deep Neural Network (DNN), which has a layer-wise structure. Each layer contains a large number of neurons, and these neurons in adjacent layers are connected to each other. Generally speaking, CNN layers can be classified into the following main types according to their functions, namely, convolutional layers (CONV), pooling layers (POOL), activation layers (ReLU) and fully connected layers (FC), etc. Among them, CONV layers contain fewer parameters but require more computing power than FC layers. Besides input layer, both ReLU layers and POOL layers contain no parameters.

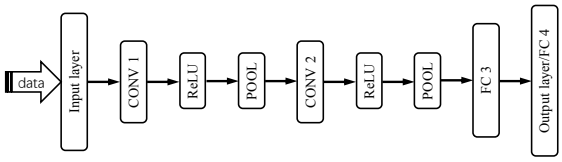


Fig. 1. CNN structure.

As the CNN model becomes more complex, the computation workload correspondingly increases. It rules out running CNN training on a single node within affordable time, and makes distributed training necessary in practice. Reviewing the distributed solutions to CNN training, Parameter Server (PS) architecture is most widely adopted [8], [21], and has been well supported by mainstream DML frameworks, such as TensorFlow [22], PyTorch [24], etc. Besides, distributed CNN is usually trained under PS architecture in a data-parallel way, with Bulk Synchronous Parallel (BSP) [25] employed as the synchronization mode. Therefore, we focus on this setting in the paper. During the training process, each worker performs parameter refinement based on different training data portions, and the model updates from different workers are aggregated by the PSes in each iteration. Specifically, each iteration includes the following processes, namely, pulling parameters, forward-pass computation, back-propagation computation, pushing gradients and aggregating gradients. Some of them can be overlapped to achieve a higher training speed.

B. Inefficient Overlap in Current Practice

Figure 2 shows network throughput and GPU utilization of the worker when training AlexNet with one dedicated PS and one worker. We can see that the overlap between pulling parameters (indicated by inbound traffic) and computation is poor. That is, computation cannot be executed most of the time when pulling parameters, wasting computing resources. To make full use of GPU's computing power, the overlap between computation and communication needs to be improved.

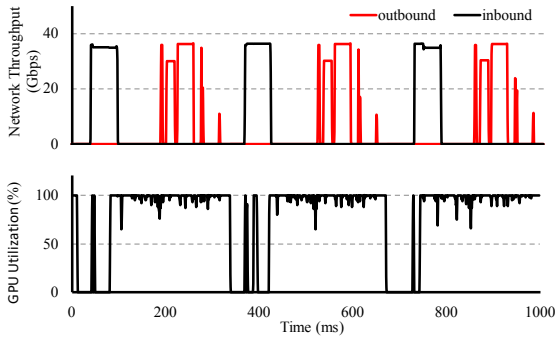
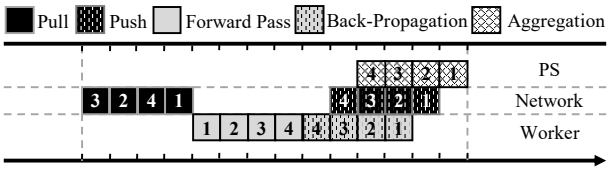


Fig. 2. Network throughput and GPU utilization of the worker in DML training.

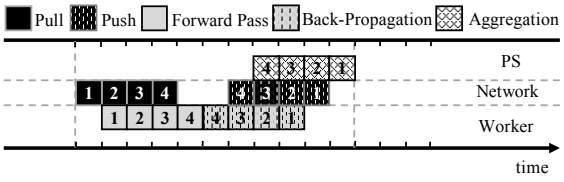
To explore how to better overlap computation and communication, we analyze the impact of pulling parameters on forward-pass computation. We observe that parameters are pulled in a random order (in the latest TensorFlow implementation) because of the concurrent execution of the split computation graph. Figure 3 shows one iteration in distributed CNN training with PS architecture and qualitatively compares the effect of parameter transfer order on the overlap between forward-pass computation and pulling parameters. We can see that in both cases, forward-pass computation and back-propagation computation are executed sequentially on the worker side. During the forward-pass phase, input data is processed layer by layer from the first layer to the last layer, so as to calculate the loss. But it is worth noting that the forward pass cannot continue until the parameters of the current layer is received. As shown in Figure 3(a), the forward-pass computation of layer 1 is blocked until the worker has received the parameters of layer 1. In real models, the parameters of FC layers are usually much larger than those of CONV layers. As a result, if large-size FC parameters are sent before small-size CONV parameters from the PS side, the latter will suffer from high queuing delay, and further postponing the start of new iterations.

C. Potential Performance Gains

In order to further quantify the potential performance gains by parameter scheduling, we model one iteration in DML



(a) Training using random parameter transfer order



(b) Training using specified parameter transfer order

Fig. 3. One iteration in DML training.

TABLE I
NOTATION USED IN THIS PAPER

Name	Description
L	The number of layers of the ML model
P^l	The number of parameters of layer l
B	Link bandwidth
τ_{iter}	Time to finish one iteration
τ_f^l	Time to execute forward pass computation of layer l
τ_b^l	Time to execute backward propagation computation of layer l
τ_g^l	Time to push gradient of layer l from the worker to the PS
τ_p^l	Time to pull parameter of layer l from the PS to the worker
t_{ff}^l	Timestamp to finish executing forward pass computation of layer l
t_p^l	Timestamp to finish pulling parameter of layer l
t_{bf}^l	Timestamp to finish pushing gradient of layer l
t_b^l	Timestamp to finish executing backward propagation computation of layer l

training. Variables used in the performance model are listed in Table I and we assume that the start timestamp is 0. We ignore the aggregation time of PS, because with parameter slicing, which we will demonstrate in Section IV, the PS can start to aggregate as soon as receiving one of the parameter slices. In this case, time of aggregation can be hidden by the time of pushing gradients.

For the forward-pass stage, pulling parameters is not affected by computation. We use $k_j = l$ to represent that layer l is the jth parameter that is transferred from the PS to the worker. Thus, the timestamp when finishing receiving the parameters of layer l is

$$t_p^l = \sum_{i=1}^{j} \tau_p^{k_i} \tag{1}$$

For the forward-pass computation, two prerequisites for calculating layer l are: 1) the parameter of layer l has been pulled from the PS; 2) layer $l-1$ has finished calculating. As a result, the time to finish forward-pass computation of layer l can be represented by

$$t_{ff}^l = \begin{cases} t_p^l + \tau_f^l, & l = 1 \\ \max\{t_{ff}^{l-1}, t_p^l\} + \tau_f^l, & 2 \le l \le L \end{cases} \tag{2}$$

For the backward propagation stage, computation starts from layer L to layer 1 sequentially and is not affected by communication, thus we have

$$t_b^l = t_{ff}^L + \sum_{i=l}^{L} \tau_b^i \tag{3}$$

Similar to forward-pass computation, two prerequisites for pushing the gradient of layer l to the PS are: 1) the gradient of layer l has been calculated; 2) the gradient of layer $l+1$ has been pushed to the PS. Therefore, the time to finish pushing gradients of layer l can be represented by

$$t_{bf}^l = \begin{cases} t_{ff}^L + \tau_g^l, & l = L \\ \max\{t_{bf}^{l+1}, t_b^l\} + \tau_g^l, & 1 \le l \le L-1 \end{cases} \tag{4}$$

Based on the above results, the iteration time can be represented by

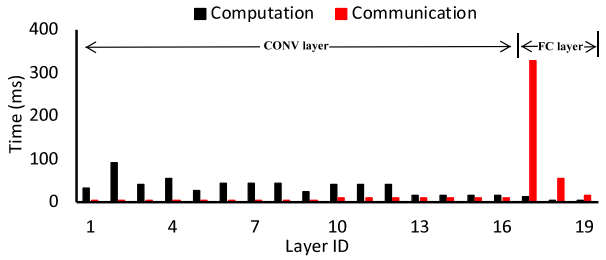
$$\begin{aligned} \tau_{iter} =& t_{bf}^1 = \max\{t_{bf}^2, t_b^1\} + \tau_g^1 \\ =& \max\{\max\{t_{bf}^3, t_b^2\} + \tau_g^2, t_{ff}^L + \sum_{i=1}^{L} \tau_b^i\} + \tau_g^1 \\ =& \cdots \end{aligned} \tag{5}$$

Equation 5 shows that the iteration time is determined by τ_f^l, τ_b^l,τ_g^l, τ_p^l and the parameter transfer order. We collect the trace of training VGG-19 on one single K40 GPU, and extract the execution time of each operation from the trace, including τ_f^l and τ_b^l. As for τ_g^l and τ_p^l, they are calculated as follows

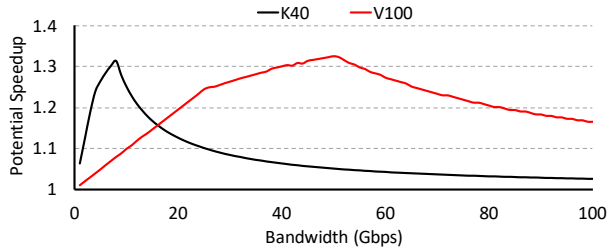
$$\tau_p^l = \tau_g^l = P^l / B \tag{6}$$

Taking the training of VGG-19 as an example, the computation time and communication time of different layers in forward-pass is shown in Figure 4(a). Back-propagation has the similar trend, thus we omit the discussion of it. Based on these measurement results, the potential speedup achieved by parameter scheduling when training VGG-19 under different link bandwidths is shown in Figure 4(b). Potential speedup is defined as the ratio of the training throughput when the parameters are transferred in order from layer 1 to layer L to that when the parameters are transferred in reverse order. We can see that the potential improvement room is huge under some certain bandwidth range, where communication time is commensurate with computation time. For example, when training with K40 GPU under 8Gbps bandwidth, the potential speedup is 31%. To achieve the same performance,

link bandwidth of beyond 100Gbps is required if no parameter scheduling is applied, which means that hardware cost can be significantly saved if the parameter transfer is optimized. In addition, with higher-performance GPU, like V100, parameter scheduling will take effect over a larger bandwidth range.



(a) Computation time and communication time of different layers with K40 GPU under 10Gbps bandwidth.



(b) Potential speedup with K40/V100 GPU under different bandwidths.

Fig. 4. VGG-19 model training.

D. Necessity of Network-level Parameter Scheduling

Having noticed the necessity of parameter scheduling in distributed CNN training, recent works, such as TICTAC [19] and P3 [20], are proposed to optimize the training performance and enforce the overlap between computation and communication. P3 slices the parameters and assigns a priority for them. The slices with different priorities are queued and the slice with the highest priority is first sent to the fabric. TICTAC approximates the optimal schedule by analyzing the DAG dependencies to prioritize important transfers at the sender.

These parameter scheduling solutions explicitly control the order in which the parameters are transferred only at the end host. However, the end-host-level solutions fail to guarantee the prioritization throughout the whole transmission process. More specifically, when the parameters go into network fabric, the end-host-level scheduling cannot help to coordinate the parameters from different PSes, as what we can see in Figure 5 later. Therefore, they can be inefficient in large-scale CNN training, especially when the network is oversubscribed.

Based on the above findings, we argue that parameter scheduling should take into consideration the entire DML systems, and scheduling needs to work both at the end host and in the network. At the end host, parameters of different priorities (urgency) are scheduled to different send queues, which guarantees that high-priority parameters should be sent ahead of low-priority ones, when they coexist at the end host. More than that, when high-priority parameters and low-priority ones both enter the network fabric, the priority scheduling via switch forwarding can still come into effect, and guarantee that high-priority parameters are transferred in advance. With these two design principles in mind, we design Geryon, which provides parameter scheduling that covers the whole training process, and can achieve better overlap between computation and communication.

III. Design

Geryon executes the prioritized scheduling policy across the whole network, which can bring much more benefit compared to the end-host-level solutions. In this section, we first illustrate our design of Geryon and explain how it remedies the drawbacks of end-host-level solutions. Then, we describe our policy for tagging parameters with different priorities.

A. Network-Level Scheduling

Geryon divides parameters into different groups based on their urgency levels during computation, and uses equal number of flows with different priorities to transfer different groups. The flow carrying the most urgent parameters is assigned to the highest priority, and as the urgency level decreases, the priority of the flow carrying the parameter decreases gradually. In this way, the queuing delay of urgent parameters at the end host can be significantly reduced. For example, in typical CNN models, CONV layers are usually placed ahead of FC layers, and the training workflow of one iteration starts from CONV layers. Therefore, we consider the parameters of CONV layers as more urgent than those of FC layers. Moreover, the priority tag in each packet header can be identified by commodity switches. When packets of different priorities coexist, switches will first forward the high-priority ones to help them arrive at the worker node earlier. And Geryon can naturally coordinate the order of parameter transfers between different PSes, which is difficult for scheduling solutions that only work at the end host.

Figure 5 illustrates the comparison between network-level scheduling and end-host-level scheduling. We can see that end-host-level scheduling fails to work under complex communication pattern, whereas network-level scheduling can remedy the drawbacks. As shown in Figure 5, there are multiple PSes and multiple workers in the DML cluster, but we only show two PSes and one worker for simplification. A parameter that is not urgently needed is placed on PS0, and another parameter that is urgently needed is placed on PS1. With network-level scheduling, as shown by the dashed box in the lower left corner, the non-urgent parameter is assigned to a low-priority flow, and the urgent one is assigned to a high-priority flow. When these two parameters reach the switch at the same time (in other words, the high-priority data packets and low-priority ones coexist in the network), the high-priority parameter will be scheduled to be sent before the low-priority one. This is because the switch can distinguish them by tags and further forward them according to the priority tag. The network-level scheduling thus guarantee that urgent parameters can be

delivered to the worker more timely. However, when it comes to the end-host-level scheduling, as shown by the dashed box in the lower right corner, the prioritization scheduling fail to work when the two parameters have both come to the network. Since each PS (end host) only controls the sending order of parameters on its own node, and it is oblivious to the parameter priority on other end hosts, they just send their own parameters with best effort. Then, the low-priority parameter on PS0 (not so urgent) and the high-priority parameters on PS1 (very urgent) both come to the network without scheduling coordination, and these two parameters equally share the limited link bandwidth. As a result, the high-priority parameter needs to take twice the time of network-level scheduling to reach the worker. What's worse, this time will increase further as the number of PSes increases.

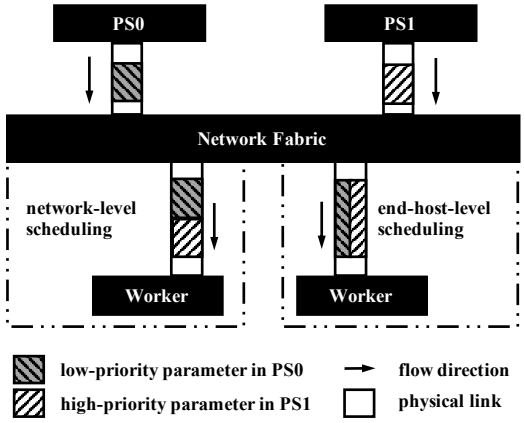


Fig. 5. Network-level scheduling *v.s.* end-host-level scheduling.

Figure 3(b) shows one iteration in DML training with Geryon. Compared to the DML training without Geryon, the training time for one iteration is reduced by 3 time units, due to the better overlap brought by Geryon. The parameters of the real model are different in size, and Geryon can bring better performance improvement.

In Geryon, only the process of assigning parameters to different flows is performed by the DML framework. The scheduling logic of parameters transfers is controlled by the underlying network fabric. In contrast, the end-host-level solutions, such as TICTAC, explicitly control the transfer order in DML framework. Compared to them, Geryon enjoys a finer granularity and can schedule more timely.

B. Coarse Categorization of Parameters

We design Geryon on top of RDMA over Commodity Ethernet (RoCE) [26], which uses Priority-based Flow Control (PFC) [27] to guarantee lossless network. Due to the limited buffer, the commodity switches only use a limited number of priority queues, typically 2 priorities [26], so it is impractical to assign one unique priority to each layer of parameters. Besides, more priorities also add complexity to the scheduling at the end host and incur more scheduling overheads.

To address this challenge, we analyze the parameter size distribution of several representative CNN models. As shown in Figure 6, the skew distribution of parameter size, i.e., the last few layers occupy the majority of the total parameters, is common in CNN models. For example, in AlexNet, VGG-19 and YOLO, the parameters in the FC layer are 93.99%, 86.06%, 77.86% of the total parameters, respectively.

Based on the above observation, we argue that two priorities are enough to help Geryon achieve high performance. Because the CONV layers have only a small part of the total parameters but the computation time of CONV layers accounts for the vast majority of the total computation time, a coarse categorization of parameters is to transfer the CONV layer parameters with the higher priority and the FC layer ones with the lower priority. This allows the computation of CONV layers and the transfer of FC layers to be performed simultaneously.

C. Flexible Categorization of Parameters

Though coarse categorization is enough for the most CNN models, there are some models where the parameters of FC layers occupy only a small fraction of the total parameters. For example, the parameters of FC layers are only 8.02% of the total parameters in ResNet-50. Therefore, we design a more flexible and self-adaptive parameter distribution strategy. *Priority threshold* is introduced to decide whether a parameter should be sent by the high-priority flow or the low-priority one. If the urgency level [1] of one parameter is more than the priority threshold, then the parameter will be sent by the high-priority flow, and vice versa. At the first few iteration, some different thresholds are used in turn, and their corresponding iteration times are collected. Then the relationship between iteration time and priority threshold can be fitted, hence we can choose the threshold corresponding to the minimum value of iteration time for the following training. It is worth noting that the priority threshold search process is only executed at the first few iterations and we do not need to start training from the beginning again after the priority threshold is determined.

Although only two priorities are used, Geryon can bring remarkable performance gains to distributed CNN training, as shown in Section V. Currently, the number of priorities that Geryon uses is mainly constrained by the hardware. However, considering the rapid development of RDMA technologies and the increase of hardware memory, more priorities can be supported in the future [28]–[30] to make finer categorization and gain better performance.

IV. Implementation

We have implemented Geryon in TensorFlow r1.10 with about 500 lines of code (LoC) for modifications/additions. The primary modifications are for the component `rendezvous`, an abstract of communication in TensorFlow.

In current implementation of RDMA-based rendezvous component, each server establishes an RDMA flow with the others at the beginning of the training and each of these flows is assigned an unique local ID, i.e., the name of the remote server. The mapping between the flows and the IDs

[1]For the parameter of layer l, its urgency level is defined as $1 - l/L$.

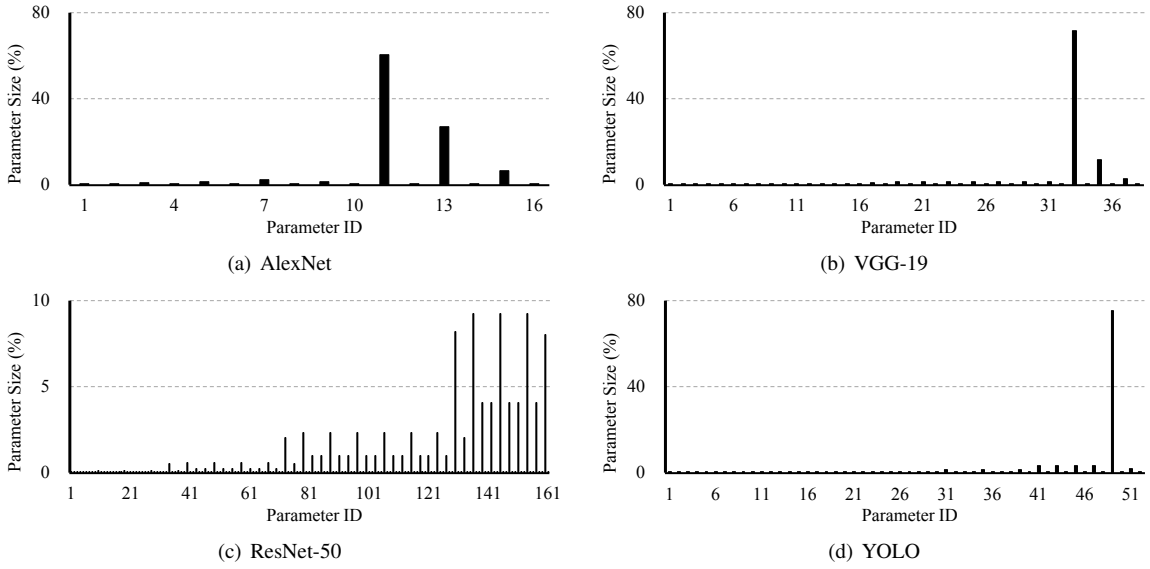


Fig. 6. Parameter size distribution.

is maintained in a table, named `channel_table`. When a parameter needs to be sent, rendezvous component looks up the `channel_table` to find the corresponding flow according to the name of the parameter, which contains the name of remote server.

We inherit the design principle of TensorFlow in our implementation. During the start-up of the training, two flows with different priorities are established between different nodes. The remote server's name and priority number are concatenated as a local ID. Besides the `channel_table`, we use a `map_table` to maintain the mapping of parameter names to the priorities. When a parameter needs to be sent, Geryon first looks up the `map_table` with the parameter name. If no entry can be found for this parameter, the corresponding priority will be assigned according to its urgency level and a new entry will be created with this priority, otherwise, a priority number will be obtained. `Rwlock` is used to ensure reliable `read/write`. Because the `map_table` is written only when the priority threshold is changed, there are many more `read` operations than `write` operations, making Geryon high-efficient. With the remote server's name and the priority number, i.e., the local ID, a flow is picked up after looking up the `channel_table`. Then the parameter is sent into the network fabric. In the entire network fabric, the priority tag can always be carried in the data packet header. For RoCEv1, the priority tag is carried in the `PCP` field of the VLAN tag.

Besides `rendezvous` component, Geryon also adds functions to record parameter variables information when the computation graph is created, which is transparent to DML framework users. This information is used to identify the parameter's urgency level and assign the parameters to different priority flows in subsequent training process.

It is worth noting that as the number of workers increases, the load imbalance among PSes may become a communication bottleneck. In order to improve load balance among multiple PSes, we partition each parameter into multiple shards, with one shard on each PS. Hence, the amounts of parameters on different PSes are equal. And fortunately, TensorFlow provides an API, named `fixed_size_partitioner`, to partition parameters into specify number of shards. Further, finer granularity of slicing can increase the overlap between pushing gradients and aggregation, as we will show in Section V. Thus, we perform fine-grained parameter slicing based on partition result. To avoid modifying the source code of CNN model, we reuse the `fixed_size_partitioner` API and encapsulate it into the `get_variable` API, which is used to create variables, to slice the parameters. Parameters that are larger than the predefined slicing threshold will be sliced to keep the slice size smaller than the slicing threshold.

V. EVALUATION

A. Evaluation Setup

Testbed Setting: We conduct our experiments on a testbed of 8 servers, each with 1 NVIDIA Tesla K40c 12 GB GPU, dual 8-core Intel Xeon E5-2630 v3 CPU at 2.40GHz, 64 GB of DDR4 RAM and 1 Mellanox Connect-X3 NIC, which supports 1/10/40 GbE. These servers are connected to a Mellanox SN2700 switch, which supports 10/25/40/50/56/100 GbE. Due to the poor performance of our GPUs, we use 10GbE as the default bandwidth. RoCEv1 is used as the underlying transport protocol. The switch and the servers are configured with PFC and set strict priority. In order to make full use of hardware resources, a PS process is co-hosted with a worker process, with the worker using GPU and the PS using CPU.

CNN Models and Datasets: We mainly evaluate Geryon's performance with three representative CNN models, AlexNet, VGG-19 and ResNet-50, using the TensorFlow benchmarks [31] open-sourced by Google. Besides, we also use

YOLO [32], a state-of-the-art, real-time object detection system, to evaluate Geryon. We apply a moderate batch size per GPU to them: 512 for AlexNet and 64 for the other three. Since Geryon neither reduces the precision of the parameters nor changes the order in which the gradients are applied, it has no impact on the convergence of model accuracy. Therefore, we use synthetic data built into the code as the model's input.

Baselines: We compare the following three solutions: (1) *Vanilla*: vanilla TensorFlow with default settings; (2) *Partition+Slicing*: TensorFlow with parameter partition and fine-grained parameter slicing; (3) *Partition+Slicing+Geryon*: Geryon enhanced with parameter partition and slicing.

Performance Metrics: We mainly focus on *training throughput*. Training throughput is defined as the total number of images processed by all the workers per second, which can be evaluated in two dimensions. One is *speedup*, which is defined as the ratio of the training throughput of Geryon to that of vanilla TensorFlow; The other is *scaling efficiency*, which is defined as the ratio of the number of images processed by one worker per second in DML training to that in single-GPU training. The former indicates the improvement of Geryon over vanilla TensorFlow, and the latter indicates the scalability of Geryon. When the scaling efficiency is 100%, the communication overhead is completely hidden. The experimental results are collected when the training is stable after a few iterations.

B. Parameter Partition and Parameter Slicing

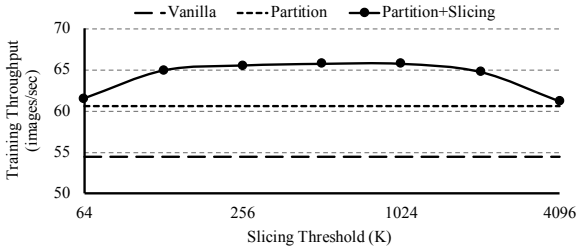


Fig. 7. Benefits of parameter partition and slicing.

To illustrate the benefit of parameter partition and parameter slicing, we conduct an experiment on 2 servers, with a PS process and a worker process running on each server. Figure 7 shows the training throughput when training with three different settings. Compared to vanilla TensorFlow, parameter partition improves training throughput by 11.3%, because parameter partition makes both the traffic load and the aggregation load between different PSes more balanced. Parameter slicing further improves training throughput by up to 9.6% depending on the parameter slicing threshold. Parameter slicing allows PSes to aggregate parameters without waiting for the whole layer of parameters to be transferred, hiding the time overhead of aggregating parameters on PSes. It is worth noting that too small parameter slicing threshold leads huge overhead for merging slices, so we use $1,024K$ as the default parameter slicing threshold for the following experiments, unless otherwise stated.

TABLE II
SCALING EFFICIENCY WITH 8 WORKERS USING 10GBE BANDWIDTH

Model	Vanilla	Partition + Slicing	Partition + Slicing + Geryon
AlexNet	36.4%	80.2%	95.4%
VGG-19	26.9%	79.3%	94.6%
ResNet-50	87.6%	90.0%	95.7%
YOLO	20.9%	75.8%	91.5%

TABLE III
SCALING EFFICIENCY WITH 8 WORKERS USING 40GBE BANDWIDTH

Model	Vanilla	Partition + Slicing	Partition + Slicing + Geryon
AlexNet	79.4%	90.0%	95.1%
VGG-19	61.7%	85.3%	94.8%
ResNet-50	94.1%	94.1%	96.1%
YOLO	53.2%	84.2%	93.6%

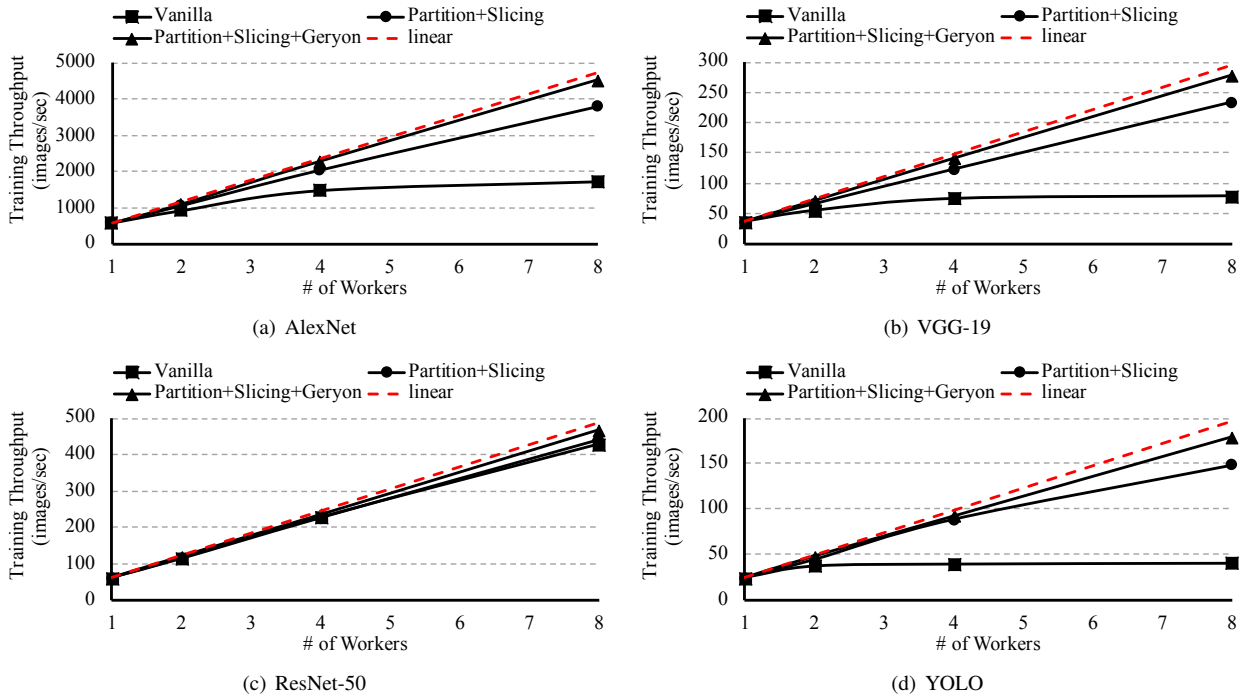
C. Performance Comparison

In this experiment, we evaluate the performance improvements achieved by Geryon when training the above models with varying number of workers and different bandwidths.

Figure 8 shows the training throughput on a 10GbE network with different numbers of workers. In general, Geryon can significantly speed up DML training. For example, when training YOLO with 8 workers, parameter partition and slicing can achieve 3.62x speedup compared to vanilla TensorFlow, and with the help of Geryon, the speedup can reach 4.37x.

As shown in Table II, Geryon can achieve near-linear scalability for these models. In particular, the scaling efficiencies of AlexNet, VGG-19 and YOLO, can reach 95.4%, 94.6% and 91.5%, respectively, even when training on 8 workers with 10GbE bandwidth. With the same cluster size and bandwidth, the scaling efficiencies of AlexNet, VGG-19 and YOLO are only 80.2%, 79.3% and 75.8%, even for TensorFlow optimized by parameter partition and slicing. What needs to be pointed out is that ResNet-50 can also benefit from Geryon, although it achieves high scaling efficiency based on vanilla TensorFlow.

We also evaluate Geryon's performance on a 40GbE network (shown in Figure 9 and Table III). Though the communication time is significantly reduced when training on a 40GbE network, Geryon can still improve training throughput by up to 9.5% compared to the training with parameter partition and slicing. Considering that a distributed training job usually takes days or even weeks, it is still very helpful to increase the training speed by 9.5%. Moreover, based on the analysis in Section II-C, we can see that as GPU speeds up, like with V100, Geryon's improvements will become more apparent again.



(a) AlexNet (b) VGG-19

(c) ResNet-50 (d) YOLO

Fig. 8. Training throughput with 1/2/4/8 workers using 10GbE bandwidth. 1 worker's training throughput refers to the performance of single-GPU training.

D. Comparison with End-Host-Level Scheduling

Existing works, such as TICTAC, leverages prioritization scheduling at the end host. We have ported TICTAC to RDMA transport according to the design principles of its TCP version for fair comparison. We train VGG-19 on 8 servers with Geryon and TICTAC, respectively. Both partition and slicing are used. The slicing thresholds are set as $1024K$, $512K$ and $256K$, and the numbers of the corresponding slices are 320, 440 and 680. The training throughput with three slicing thresholds is shown in Table IV. Though the performance gap between Geryon and TICTAC is small when the number of slices is not too much, the gap becomes larger as the slicing threshold decreases. And it is worth noting that, the performance gap between Geryon and end-host solutions can be different if we use different experiment settings (GPU types, network speed and network size), but our experimental results demonstrate the consistent advantage of Geryon over end-host solutions. We believe the potential gain will be larger in bigger networks with more hosts and switches.

Besides the performance loss due to the inability to coordinate multiple PSes, software scheduling overhead of TICTAC is not negligible. To guarantee the parameters are sent in the predefined order, more threads needs to be created when the existing threads are blocked by the mutex locks, because the low-priority parameters managed by the existing threads cannot be sent by now. Therefore, as the parameter slices increase, the number of scheduling threads also grow correspondingly, thus causing more overheads for thread coordination. Figure 10 shows the CDF of the time to schedule parameter slices in one iteration when three different slicing thresholds are used. With slicing threshold of $256K$, the scheduling time of TICTAC can be up to $800ms$, and more than 50% of the scheduling time is more than $700ms$. Due that the time for pulling parameters is around $400ms$, if the max scheduling time is less than $400ms$, the impact on training throughput is slight. But as the scheduling time increases, the impact becomes significant and cannot be hidden anymore. We can see that with the slicing threshold of $256K$, the iteration time is increased by about $450ms$.

TABLE IV
TRAINING THROUGHPUT WITH DIFFERENT SLICING THRESHOLD

	Training throughput (images/sec)	
Slicing threshold	**Geryon**	**TICTAC**
$1024K$	279.2	272.0
$512K$	276.0	268.0
$256K$	272.8	216.8

By contrast, the scheduling time of Geryon at the end host is within $1ms$. This is because Geryon offloads the scheduling logic to the NIC hardware and network fabric, thus relieving the workload at the end host. During the scheduling process, Geryon only looks up the `map_table` (without lock) to choose one of the flows in software, and afterwards the parameters are scheduled by the network fabric efficiently.

VI. RELATED WORK

End-host-level scheduling: P3 [20] slices the parameters and assigns a priority for them. The slices with different

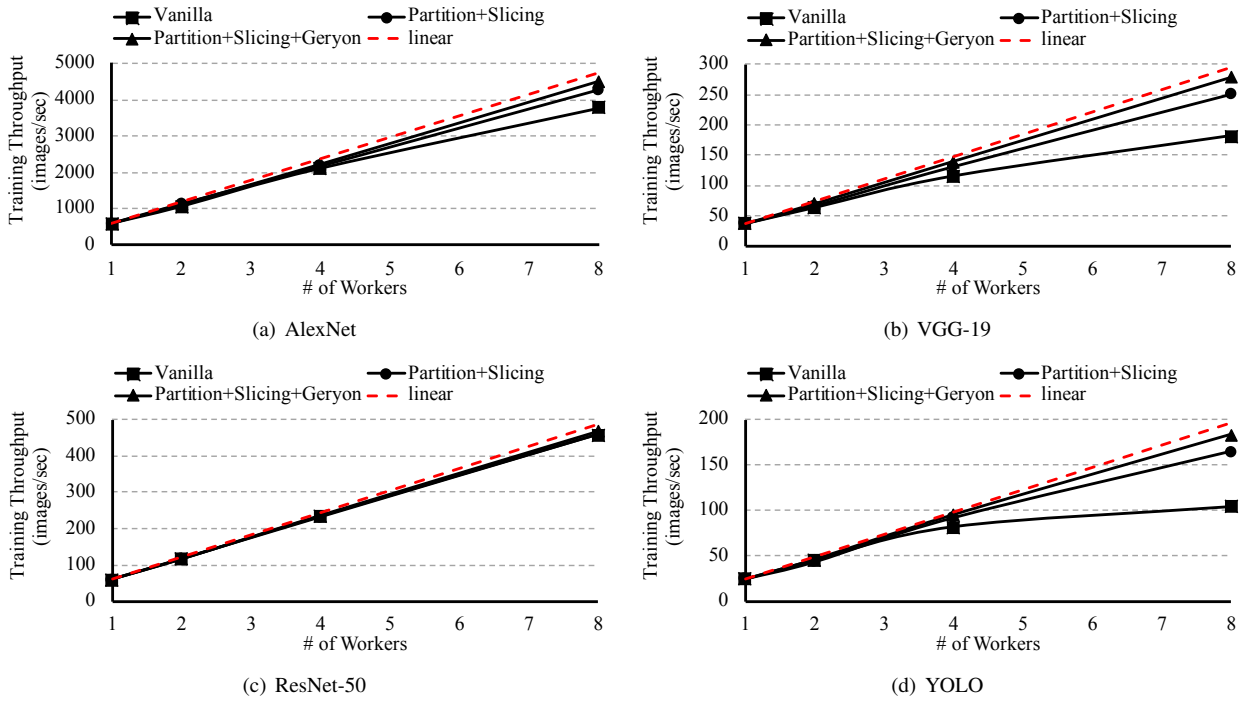


Fig. 9. Training throughput with 1/2/4/8 workers using 40GbE bandwidth. 1 worker's training throughput refers to the performance of single-GPU training.

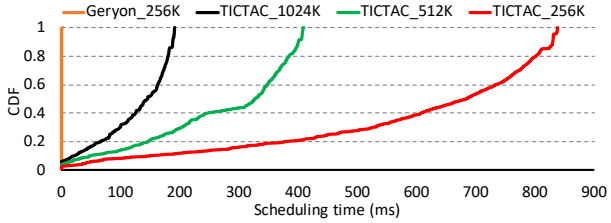


Fig. 10. Scheduling overhead of TICTAC and Geryon.

priorities are queued and scheduled to be sent into the fabric according to their priorities. But P3 relies on the blocking call of TCP to send parameters with the highest priority in the priority queue. Besides the similar drawbacks to TICTAC, it may cause additional performance degradation when implemented with RDMA. Since the send/receive operations of RDMA are executed asynchronously, P3 needs the confirmation for each send operation via polling the Completion Queue, thus leading to possible delays between send operations and more consumption of CPU resources.

Gradient Compression: Some work uses gradient compression to reduce the amount of traffic of parameter synchronization. For example, 1-bit SGD [13], DoReFa-Net [33] and QSGD [34] use much fewer bits to replace 32-bit floating point value. As a result, traffic is reduced by an order of magnitude, which accelerates DML training by 10x. These solutions work in the application layer and can be further integrated into Geryon for performance enhancement.

In-Network Aggregation: Some work [36], [37] attempts to apply In-Network Aggregation (INA) to DML training. Programmable switches are leveraged to aggregate the gradients from each sever and then only the aggregation result is sent out to the PSes. INA significantly reduces the amount of data transferred without losing parameter accuracy. As a network-level scheduling solution, Geryon can help INA to improve the overlap between communications and computation by prioritizing the sending of higher priority flows at the switch.

Other Approaches: Parameter Hub [15] improves the network capability of PS by using multiple NICs. Gaia [38] reduce communication traffic by sending only the important gradients. MG-WFBP [40] improves communication efficiency by merging multiple small-size parameters into a single large one. HiPS [12], [21] and BML [11] leverage the hierarchical network topology to synchronize parameters in parallel.

VII. Conclusion and Future Work

In this paper, we propose Geryon, a network-level flow scheduling scheme, to accelerate distributed CNN training. Geryon uses multiple flows with different priorities between any PS/worker pairs to transfer parameters of different urgency levels. We have implemented Geryon in TensorFlow and evaluated it on a small-scale testbed with 8 single-GPU servers. Based on our testbed experimental results, even with 10GbE bandwidth, Geryon can achieve up to 95.7% scaling efficiency. Compared to vanilla TensorFlow, Geryon enhanced with parameter partition and slicing can achieve up to 4.37x speedup in terms of training throughput. Currently we only use two priorities and the performance gains are remarkable. With the rapid development of hardware, more priorities can be used to further improve distributed CNN training performance.

REFERENCES

[1] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of IEEE CVPR'17*, 2017.

[2] S. Qiao, Z. Zhang, W. Shen, B. Wang, and A. Yuille, "Gradually updated neural networks for large-scale image recognition," in *Proceedings of ICML'18*, 2018.

[3] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, and et al., "The microsoft 2016 conversational speech recognition system," in *Proceedings of IEEE ICASSP'17*, 2017.

[4] D. He, H. Lu, Y. Xia, T. Qin, L. Wang, and T. Liu, "Decoding with value networks for neural machine translation," in *NeurIPS*, 2017.

[5] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[6] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *EuroSys*, 2016.

[7] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, and et al., "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *ATC*, 2017.

[8] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, and et al., "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.

[9] "Baidu-allreduce," https://github.com/baidu-research/baidu-allreduce, 2017.

[10] J. Geng, D. Li, and S. Wang, "Rima: An rdma-accelerated model-parallelized solution to large-scale matrix factorization," in *ICDE*, 2019.

[11] S. Wang, D. Li, Y. Cheng, J. Geng, Y. Wang, S. Wang, S.-T. Xia, and J. Wu, "Bml: A high-performance, low-cost gradient synchronization algorithm for dml training," in *NeurIPS*, 2018.

[12] S. Wang, D. Li, J. Geng, Y. Gu, and Y. Cheng, "Impact of network topology on the performance of dml: Theoretical analysis and practical factors," in *INFOCOM*, 2019.

[13] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns," in *ISCA*, 2014.

[14] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached," in *ATC*, 2012.

[15] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter hub: a rack-scale parameter server for distributed deep neural network training," in *SoCC*, 2018.

[16] J. Geng, D. Li, and S. Wang, "Elasticpipe: An efficient and dynamic model-parallel solution to dnn training," in *Proceedings of 10th workshop on Scientific Cloud Computing*, ser. ScienceCloud '19. New York, NY, USA: ACM, 2019.

[17] ImageNet, "Large scale visual recognition challenge 2016 (ilsvrc2016)," 2016. [Online]. Available: http://image-net.org/challenges/LSVRC/2016/results

[18] J. Geng, D. Li, and S. Wang, "Horizontal or vertical? a hybrid approach to large-scale distributed machine learning," in *Proceedings of 1st Workshop on Converged Computing Infrastructure*, ser. CCIW '19. New York, NY, USA: ACM, 2019.

[19] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," in *SysML*, 2019.

[20] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," in *SysML*, 2019.

[21] J. Geng, D. Li, Y. Cheng, S. Wang, and J. Li, "HiPS: Hierarchical parameter synchronization in large-scale distributed machine learning," in *NetAI*, 2018.

[22] M. Abadi, P. Barham, J. Chen, and et al., "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[23] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, 2015.

[24] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NeurIPS*, 2017.

[25] L. G. Valiant, "A bridging model for parallel computation." ACM, 1990.

[26] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *SIGCOMM*, 2016.

[27] "IEEE DCB. 802.1Qbb." https://1.ieee802.org/dcb/802-1qbb/, 2011.

[28] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, and et al., "Revisiting network support for rdma," in *SGICOMM*, 2018.

[29] "Resilient roce," https://community.mellanox.com/s/article/introduction-to-resilient-roce—faq, 2018.

[30] D. Firestone, A. Putnam, S. Mundkur, and et al., "Azure accelerated networking: Smartnics in the public cloud," in *NSDI*, 2018.

[31] "tensorflow/benchmarks," https://github.com/tensorflow/benchmarks, 2016.

[32] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[33] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016.

[34] D. Alistarh, D. Grubic, and et al., "Qsgd: Communication-efficient sgd via gradient quantization and encoding," in *NeurIPS*, 2017.

[35] M. Yu, Z. Lin, K. Narra, and et al., "Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed cnn training," in *NeurIPS*, 2018.

[36] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *HotNets*, 2017.

[37] L. Luo, M. Liu, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Motivating in-network aggregation for distributed deep neural network training," in *WAX*, 2017.

[38] K. Hsieh, A. Harlap, N. Vijaykumar, and et al., "Gaia: Geo-distributed machine learning approaching LAN speeds," in *NSDI*, 2017.

[39] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing," in *ISCA*, 2015.

[40] S. Shi, X. Chu, and B. Li, "MG-WFBP: Efficient data communication for distributed synchronous sgd algorithms," in *INFOCOM*, 2019.