Communication-Efficient Distributed Deep Learning with Merged Gradient Sparsification on GPUs

Shaohuai Shi[†], Qiang Wang[†], Xiaowen Chu[†*], Bo Li[‡], Yang Qin[§], Ruihao Liu[¶], Xinxiao Zhao[¶]
[†]High-Performance Machine Learning Lab, Department of Computer Science, Hong Kong Baptist University
[‡]Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
[§]Department of Computer Science and Technology, Harbin Institute of Technology (Shenzhen)
[¶]MassGrid.com, Shenzhen District Block Technology Co., Ltd.
[†]{csshshi, qiangwang, chxw}@comp.hkbu.edu.hk, [‡]bli@cse.ust.hk, [§]csyqin@hitsz.edu.cn, [¶]{wany,zhxx}@massgrid.com

***Abstract*—Distributed synchronous stochastic gradient descent (SGD) algorithms are widely used in large-scale deep learning applications, while it is known that the communication bottleneck limits the scalability of the distributed system. Gradient sparsification is a promising technique to significantly reduce the communication traffic, while pipelining can further overlap the communications with computations. However, gradient sparsification introduces extra computation time, and pipelining requires many layer-wise communications which introduce significant communication startup overheads. Merging gradients from neighbor layers could reduce the startup overheads, but on the other hand it would increase the computation time of sparsification and the waiting time for the gradient computation. In this paper, we formulate the trade-off between communications and computations (including backward computation and gradient sparsification) as an optimization problem, and derive an optimal solution to the problem. We further develop the optimal merged gradient sparsification algorithm with SGD (OMGS-SGD) for distributed training of deep learning. We conduct extensive experiments to verify the convergence properties and scaling performance of OMGS-SGD. Experimental results show that OMGS-SGD achieves up to 31% end-to-end time efficiency improvement over the state-of-the-art sparsified SGD while preserving nearly consistent convergence performance with original SGD without sparsification on a 16-GPU cluster connected with 1Gbps Ethernet.**

***Index Terms*—Distributed Deep Learning; Gradient Communication; Merged Gradient**

I. Introduction

Deep learning has been successfully applied in many real-world AI applications [1], while it requires massive computations to train a satisfactory model. Distributed computing environments (e.g., GPU clusters or Google TPU pods) are deployed to accelerate the training process of large deep neural network (DNN) models using distributed stochastic gradient descent (SGD) methods [2]. The data-parallel synchronous SGD algorithm is one of the most widely used optimizers for distributed training of DNNs because it has the same convergence performance as the classical SGD. However, synchronous SGD requires additional data communication and model synchronization in each iteration, which limit the scalability of the distributed system and reduce the utilization of computing resources. E.g., when training ResNet-50 (with model size 97MB) on a 16-node GPU cluster (Nvidia P102-100) connected by 1Gbps Ethernet, the computing time is 0.18s (with an overall mini-batch size of 256) while the communication time is 1.92s. To address the communication challenge, there exist (1) algorithmic-level approaches that try to reduce the communication traffic; and (2) system-level approaches that try to hide some communication overhead by pipelining communications with computations.

On the algorithmic level, gradient compression techniques (especially Top-k sparsificaiton) [3][4][5][6] have been recently proposed to significantly reduce the communication traffic with little impact on the model accuracy, which makes it possible to train large DNNs on low bandwidth networks (e.g., 1Gbps Ethernet). Top-k sparsification [3][4] in distributed SGD (TopK-SGD hereafter) only exchanges a small portion of gradients (e.g., top 0.1% local gradients) in each iteration and accumulates other gradients until they become large enough later. TopK-SGD has been theoretically and empirically verified that it has convergence guarantee with the same order of convergence rate as vanilla SGD [4][7][8]. Although the TopK-SGD algorithm reduces the communication traffic, it introduces extra computing overheads on the selection of top-k gradients at each iteration, which brings a new challenge for many-core processors [9][10]. When the computing overhead of gradient sparsification is comparable to the gradient computation time and the communication time, it will again limit the scaling efficiency.

Regarding the system-level optimization, one can increase the workload of the accelerator by large-batch training [11][12][2] to reduce the communication-to-computation ratio. However, large batch size may sacrifice the convergence performance and its theoretical convergence property remains as an active open research problem. On the other hand, by exploiting the layer-wise structure of DNN models, the wait-free backpropagation (WFBP) algorithm [13][14] can pipeline the layer-wise communications and computations so that some communication overheads can be hidden. We call this kind of pipelined distributed SGD algorithms as P-SGD. In P-SGD, each node needs to invoke one communication for each layer (or tensor) and hence many rounds of communications will be generated, resulting in many times of startup communication latency [15][16]. To alleviate the impact of the multiple

*Corresponding author.

startup time problem, merged-gradient (or called tensor fusion) techniques [15][16][17][18] have been proposed recently by both academia and industry.

To embrace both advantages of gradient compression and pipelining, we may naively integrate TopK-SGD and pipelining together [19], which is called LAGS-SGD. Unfortunately, LAGS-SGD could result in even worse performance as there still exists the startup problem of many-layer communications. Merged-gradient methods could alleviate this phenomenon by merging tensors from multiple layers into a single one, but the merged layers may bring higher computational overheads on gradient sparsification and longer waiting time for the backward computation. In this paper, we call the set of LAGS-SGD schemes with merged gradient sparsification MGS-SGD. We aim to explore whether it is possible to derive an optimal merging scheme that can maximize the overlap between communications and computations with sparsified gradients.

To this end, we first formulate the communication challenge in LAGS-SGD as an optimization problem which targets at minimizing the iteration time. Then we propose an optimal solution named Optimal Merged Gradient Sparsification (OMGS)[1], whose time complexity is $O(L^2)$ where L is the number of layers or tensors[2]. We implement the OMGS based distributed SGD (OMGS-SGD) algorithm, and conduct experiments on various DNN models with several data sets to verify the convergence performance of OMGS-SGD. We demonstrate the improved scaling efficiency compared to the existing methods. Experimental results show that our proposed OMGS-SGD achieves up to 31% improvement compared to TopK-SGD on the 16-GPU cluster connected with 1GbE. Our contributions are summarized as follows:

- We formulate an optimization problem for pipelined TopK-SGD which overlaps communications and computations. We derive an optimal solution for the optimization problem without affecting the training speed.
- We propose the OMGS-SGD training algorithm which explores the optimal merged gradient sparsification on SGD. We implement a prototype of OMGS-SGD on PyTorch and make it publicly available[3].
- We conduct extensive experiments on a 16-node GPU cluster to verify the convergence of OMGS-SGD, and show that it outperforms existing algorithms in terms of wall-clock running time.

For the ease of discussion, we list the different distributed SGD algorithms and compare them in Table I.

The rest of the paper is organized as follows. We present some preliminaries in Section II, followed by the formulation of the problem we need to address in Section III. We derive an optimal solution to the formulated problem and propose an OMGS-SGD training algorithm in Section IV. We then evaluate the performance of OMGS-SGD with extensive experiments in Section V. Section VI introduces related work, and we conclude the paper in Section VII.

[1]Although we mainly focus on the sparsified SGD algorithms, it is also applicable to other compression methods such as gradient quantization.

[2]In current deep learning frameworks (e.g., TensorFlow [20] and PyTorch [21]), a layer may have two tensors (weight and bias).

[3]https://github.com/HKBU-HPML/OMGS-SGD.

TABLE I
COMPARISON OF DIFFERENT DISTRIBUTED SGD ALGORITHMS.

Algorithm	Pipeline	Sparsification	Merged Gr.	Opt. M.
P-SGD [13]	✓	✗	✗	✗
MG-WFBP [16]	✓	✗	✓	✓
TopK-SGD [4]	✗	✓	✓	✗
LAGS-SGD [19]	✓	✓	✗	✗
MGS-SGD	✓	✓	✓	✗
OMGS-SGD	✓	✓	✓	✓

Note: "Merged Gr." indicates whether gradients are merged or not, and "Opt. M." indicates whether it uses an optimal merging strategy.

II. PRELIMINARIES

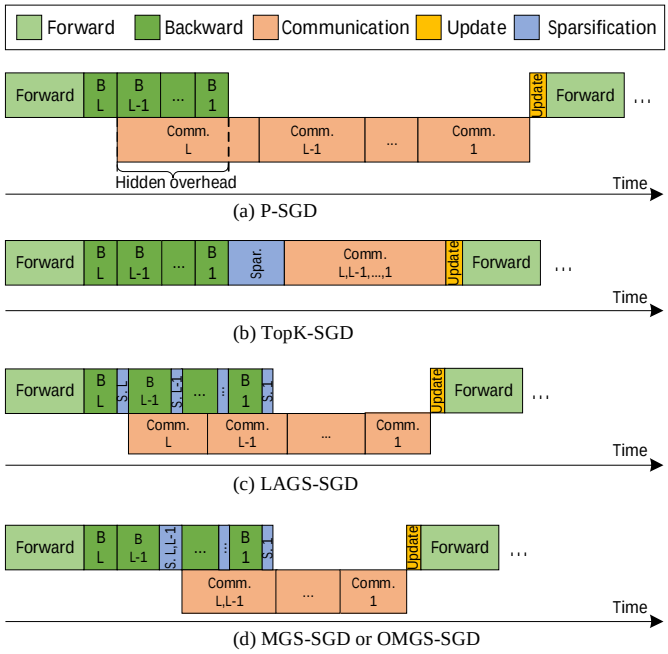


Fig. 1. Comparison of four distributed SGD algorithms.

In this section, we introduce some preliminaries which help us formulate the problem in Section III. For ease of presentation, we list some frequently used notations throughout this paper in Table II.

A. Mini-batch SGD

We briefly introduce the process of mini-batch SGD on a single worker (e.g., a GPU), which is an iterative algorithm that aims to train an L-layer model with parameters W whose dimension is d. Each iteration consists of several steps. Let us consider the i^{th} iteration. First, it loads a mini-batch of data D_i, and the data is further transferred to GPU memory. Second, it launches a sequence of GPU kernels to do the feed-forward computation from layer 1 to layer L to derive the loss value $\mathcal{L}(W_i, D_i)$. Third, it launches another sequence of GPU kernels to calculate the gradients $\nabla\mathcal{L}(W_i, D_i)$ from layer

TABLE II
FREQUENTLY USED NOTATIONS

Name	Description
P	The number of workers (GPUs) in the cluster.
α	Startup time of a single operation of gradients aggregation.
β	Transmission and aggregation time per gradient.
d	The number of parameters of a deep model.
$d^{(l)}$	The number of parameters in the learnable layer l.
L	The number of learnable layers (or tensors) of a deep model.
t_{iter}	Time of an iteration.
t_f	Time of the forward pass in each iteration.
t_b	Time of the backward pass in each iteration.
t_u	Time of the model update in each iteration.
$t_b^{(l)}$	Time of the gradient calculation of layer l in each iteration.
$\tau_b^{(l)}$	The timestamp when layer l begins to calculate gradients.
$t_s^{(l)}$	Time of the gradient sparsification of layer l in each iteration.
$\tau_s^{(l)}$	The timestamp when layer l begins to sparsify gradients.
t_c	Time of gradient aggregation in each iteration.
$t_c^{(l)}$	Time of gradient aggregation of layer l in each iteration.
$\tau_c^{(l)}$	The timestamp when layer l begins to exchange gradients.
t_c^{no}	The non-overlapped communication cost in each iteration.

L back to layer 1, which is referred to as backpropagation. Finally it updates the model parameters by

$$W_{i+1} = W_i - \eta \cdot \nabla \mathcal{L}(W_i, D_i). \tag{1}$$

In general, the most time-consuming parts are the second (the forward pass) and third (the backward pass) steps, whose time are represented as t_f and t_b respectively. We ignore the data loading and transferring time at the first step since they can be easily parallelized with the previous iteration. We also ignore the update time since it is very small compared to the forward and backward computations [22][23]. The backward pass can only be started after the forward pass, so the gradient calculation of the last layer L begins after the forward pass. The backward pass time can be represented by summing up the layer-wise computations, i.e., $t_b = \sum_{l=1}^{L} t_b^{(l)}$.

If we denote the beginning time of an iteration by 0, then the layer-wise beginning timestamp of backpropagation can be formulated as

$$\tau_b^l = \begin{cases} t_f & l = L \\ \tau_b^{(l+1)} + t_b^{(l+1)} & 1 \le l < L \end{cases}. \tag{2}$$

It is also obvious that the iteration time of SGD on a single worker can be estimated by

$$t_{iter}^{SGD} = t_f + \sum_{l=1}^{L} t_b^{(l)}. \tag{3}$$

B. Pipelining on distributed SGD

Distributed SGD with data-parallelism exploits multiple workers to perform forward and backward computations in parallel with their local data. At the end of each iteration, it aggregates the gradients from all workers and then updates the model by the average gradients. Thus, distributed SGD introduces the communication overhead on the gradient aggregation and model distribution. On the traditional distributed SGD without gradient sparsification, pipelining between communications and computations is a common practice to hide some communication overheads [13][14][16][2] by exploiting the layer-wise structure of DNN models, which we refer to as P-SGD. The pipeline process between communications and backward computations is illustrated in Fig. 1(a). This is possible because the gradient computation of layer $i-1$ has no dependency on layer i's communication, hence the communication of layer i and the gradient computation of layer $i-1$ can be parallelized. Note that some layers' communications can be hidden by the gradient computations of their previous layers. We denote the total communication time cost as t_c.

C. TopK-SGD

Top-k sparsification SGD (TopK-SGD) [3][4] is a promising algorithm for reducing the communication traffic in distributed training. In TopK-SGD, each worker sparsifies its local gradients (e.g., it only selects the top-k gradients and k can be two to three orders smaller than the model dimension), whose training process is shown in Fig. 1(b). Notice that TopK-SGD introduces an extra computing overhead on gradient sparsification since every worker needs to select the top-k local gradients, which requires some additional computing time (denoted by t_s). Moreover, it is not possible to overlap communications and computations in the original TopK-SGD because the top-k gradients are selected from the whole set of gradients, i.e., the gradient communication can only happen after all gradients have been calculated. In TopK-SGD, the iteration time can be estimated by

$$t_{iter}^{TopKSGD} = t_f + \sum_{l=1}^{L} t_b^{(l)} + t_s + t_c. \tag{4}$$

D. LAGS-SGD: Pipelining on TopK-SGD

In order to enable the pipelining of computations and communications on TopK-SGD, the layer-wise sparsification selects the top-k gradients from each layer rather than the global top-k gradients [19]. This mechanism is referred to as LAGS-SGD, as illustrated in Fig. 1(c). The convergence properties of LAGS-SGD are analyzed in [19]. To overlap the communications with computations, one should sparsify the gradients of a layer immediately after they have been calculated so that the sparsified gradients can be ready for communication. Formally, we have the following equations to estimate the iteration time for LAGS-SGD, in which t_c^{no} denotes the non-overlapped communication cost in each iteration.

$$t_{iter}^{LAGS} = t_f + \sum_{l=1}^{L} t_b^{(l)} + \sum_{l=1}^{L} t_s^{(l)} + t_c^{no}, \tag{5}$$

$$\tau_b^{(l)} = \begin{cases} t_f & l = L \\ \tau_s^{(l+1)} + t_s^{(l+1)} & 1 \le l < L \end{cases}, \tag{6}$$

$$\tau_s^{(l)} = \tau_b^{(l)} + t_b^{(l)}, \tag{7}$$

$$\tau_c^{(l)} = \begin{cases} \tau_s^{(l)} + t_s^{(l)} & l = L \\ \max\{\tau_s^{(l)} + t_s^{(l)}, \tau_c^{(l+1)} + t_c^{(l+1)}\} & 1 \le l < L \end{cases}, \tag{8}$$

and

$$t_c^{no} = \tau_c^{(1)} + t_c^{(1)} - \tau_s^{(1)} + t_s^{(1)}. \quad (9)$$

E. Top-k Selection on GPUs

The number of parameters of modern DNN models range from millions to billions. Top-k sparsification requires an efficient Top-k selection algorithm [4]. However, the Top-k selection algorithm [9] contains many irregular data access which are not friendly to the GPU architecture, hence the compression overhead is non-negligible. To model the time cost of the gradient sparsification with the Top-k selection algorithm, we build a relationship between the elapsed time $t_s(d, \rho d)$ and the dimension d for selecting $k = \rho \times d$ elements ($0 < \rho \leq 1$) from a d-dimension tensor as

$$t_s(d, k) = \gamma \rho d \log d, \quad (10)$$

where γ is an estimated parameter for a particular GPU.

F. Communication Model

For the sparsified gradient aggregation, the gradients should be communicated across multiple workers. The local Top-k selections would generate irregular indices of selected gradients so that the commonly used AllReduce collective [2][16] cannot be applied for TopK-SGD or LAGS-SGD. We adopt the recently proposed AllGather method [24] for the sparse gradient aggregation. The efficient AllGather collective utilizes the doubling recursive algorithm [24][25], whose running time can be modelled as

$$t_c(d) = \alpha + \beta \times d, \quad (11)$$

where α is the overhead that is not related to the message size, β is the transmission time per byte, and d is the message size in bytes. The time model of Eq. (11) indicates that gradient aggregation consists of the startup time and the transmission time, which is a commonly used communication model [26][27][16].

III. Problem Formulation

It is obvious that transmitting two messages (with sizes of d_1 and d_2) separately is more expensive than transmitting a single message with a size of $d_1 + d_2$ according to Eq. (11). It is easy to see that if one merges all layers' gradients into a single one (i.e., the single-layer communication [15]), the communication cost will be the lowest. However, the single-layer communication requires all gradients be calculated before transmitting, and hence misses the pipelining opportunity. Therefore, our target is to find the optimal overlapping between communications and computations during the training process by merging multiple tensors into one when possible so that the network bandwidth can be better utilized. Unlike the problem formulated in [16] whose computation time of backpropagation keeps unchanged throughout the whole training process, the overlapping problem with sparsified gradients introduces extra costs due to the computation of gradient sparsification. Furthermore, the compression overhead according to Eq. (10) would be enlarged on merged gradients, that is

$$t_s(d_1, \rho d_1) + t_s(d_2, \rho d_2) < t_s(d_1 + d_2, \rho d_1 + \rho d_2). \quad (12)$$

For a given deep model that needs to be trained with LAGS-SGD on a specific P-GPU cluster, we want to determine which layers should be merged together. The gradients of the merged layers are further sparsified and aggregated. Our purpose is to minimize the iteration time. Before formulating the problem, we introduce the concept of merged-layer.

Definition 1. Merged-layer: A layer l is called a merged-layer if at the timestamp of $\tau_s^{(l)}$, instead of compressing the gradients of the layer, merging its gradients into its previous layer $l-1$ to be compressed together. The operator $\oplus$ defines the gradients merging between two consecutive layers, say $(l) \oplus (l-1)$.

If layer l is a merged-layer, then layer l and $l-1$ have the following properties:

- l is larger than 1, which indicates that the first layer cannot be a merged-layer because there is no previous layer to be merged.
- Layer l has no compression and communication operations, i.e.,
$$t_s^{(l)} = t_c^{(l)} = 0. \quad (13)$$
- The gradient size of layer $l-1$ is enlarged by $d^{(l)}$, i.e.,
$$d^{(l-1)} = d^{(l-1)} + d^{(l)}. \quad (14)$$
- The gradient calculation of layer $l-1$ can immediately begin after the gradient calculation of layer l finishes, i.e.,
$$\tau_b^{(l-1)} = \tau_b^{(l)} + t_b^{(l)}. \quad (15)$$

Note that in our analysis, a layer $1 < l \leq L$ is either a merged-layer or a normal layer. We use l_m to denote that l is a merged-layer, and l_n for a normal layer. Therefore, there are 2^{L-1} combinations for $1 < l \leq L$ and $l \in \{l_m, l_n\}$. Let $\mathbb{M}$ denote the combination set, i.e.,

$$\mathbb{M} = \{[L, L-1, ..., 1] | 1 < l \leq L \text{ and } l \in \{l_m, l_n\}\}. \quad (16)$$

For the LAGS-SGD algorithm that allows the merged-layers to exist, which is called MGS-SGD shown in Fig. 1(d), the iteration time can be formulated by

$$t_{iter} = \tau_c^{(1)} + t_c^{(1)}, \quad (17)$$

where $\tau_c^{(1)}$ is the time moment that layer 1 begins its communication, and $t_c^{(1)}$ is the time duration of layer 1's communication which is the last operation at each iteration. Since the beginning time of layer 1's communication is determined by layer 2's communication and layer 1's computations, we can formulate the problem as follows.

Given a deep model with L layers trained with MGS-SGD across a P-node cluster, we want to find a combination $\boldsymbol{m} \in \mathbb{M}$ such that the iteration time t_{iter} is minimal. That is

$$\text{minimize: } \max\{\tau_s^{(1)} + t_s^{(1)}, \tau_c^{(2)} + t_c^{(2)}\} + t_c^{(1)}, \quad (18)$$

$$\text{s.t. } \boldsymbol{m} \in \mathbb{M}. \quad (19)$$

IV. Solution

A. Theoretical Analysis

Due to the sequential layer-wise structure of backpropagation, we can determine the gain from a merged-layer, which is not affected by other layers. To find the optimal solution, the key idea is to determine whether merging a layer could bring performance gain or not. If yes, we make the layer a merged-layer; otherwise, we keep the layer as a normal one.

According to Eqs. (5)-(9), for any layer l, $1 < l \leq L$, the completion time of its previous layer $l-1$ is formulated as

$$\begin{aligned}
\mu_c^{(l-1)} =& \tau_c^{(l-1)} + t_c^{(l-1)} \\
=& \max\{\tau_s^{(l-1)} + t_s^{(l-1)}, \tau_c^{(l)} + t_c^{(l)}\} + t_c^{(l-1)} \\
=& \max\{\tau_b^{(l-1)} + t_b^{(l-1)} + \gamma\rho d^{(l-1)} \log d^{(l-1)} \\
& , \tau_c^{(l)} + \alpha + \beta d^{(l)}\} + \alpha + \beta d^{(l-1)} \\
=& \max\{\tau_s^{(l)} + t_s^{(l)} + t_b^{(l-1)} + \gamma\rho d^{(l-1)} \log d^{(l-1)} \\
& , \tau_c^{(l)} + \alpha + \beta d^{(l)}\} + \alpha + \beta d^{(l-1)} \\
=& \max\{\tau_b^{(l)} + t_b^{(l)} + t_b^{(l-1)} + \\
& \gamma\rho(d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)}), \tau_c^{(l)} + \alpha + \beta d^{(l)}\} \\
& + \alpha + \beta d^{(l-1)}.
\end{aligned}$$

We would like to check whether $\mu_c^{(l-1)}$ can be shorten by merging layer l to layer $l-1$. In other words, if making l be a merged-layer would achieve smaller $\mu_c^{(l-1)}$, then layer l becomes a merged-layer; otherwise l is a normal layer. Assume that l is a merged-layer, then we use properties of Eqs. (13)-(15) to derive the completion time of layer $l-1$, which is

$$\begin{aligned}
\widetilde{\mu}_c^{(l-1)} =& \max\{\tau_b^{(l-1)} + t_b^{(l-1)} + \gamma\rho d^{(l-1)} \log d^{(l-1)}, \\
& \tau_c^{(l)} + \alpha + \beta d^{(l)}\} + \alpha + \beta d^{(l-1)} \\
=& \max\{\tau_b^{(l)} + t_b^{(l)} + t_b^{(l-1)} \\
& + \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}), \tau_c^{(l)}\} \\
& + \alpha + \beta(d^{(l-1)} + d^{(l)}).
\end{aligned}$$

Let $t_{gain} = \mu_c^{(l-1)} - \widetilde{\mu}_c^{(l-1)}$ denote the saving time by making layer l be a merged-layer. We have

$$\begin{aligned}
t_{gain} =& \max\{\tau_b^{(l)} + t_b^{(l)} + t_b^{(l-1)} + \\
& \gamma\rho(d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)}), \tau_c^{(l)} + \alpha + \beta d^{(l)}\} \\
& + \alpha + \beta d^{(l-1)} - \\
& - \max\{\tau_b^{(l)} + t_b^{(l)} + t_b^{(l-1)} \\
& + \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}), \tau_c^{(l)}\} \\
& - \alpha - \beta(d^{(l-1)} + d^{(l)}) \\
=& \max\{\tau_b^{(l)} + t_b^{(l)} + t_b^{(l-1)} + \\
& \gamma\rho(d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)}), \tau_c^{(l)} + \alpha + \beta d^{(l)}\} \\
& - \max\{\tau_b^{(l)} + t_b^{(l)} + t_b^{(l-1)} \\
& + \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}), \tau_c^{(l)}\} \\
& - \beta d^{(l)}.
\end{aligned}$$

From the above equation, we need to eliminate the two maximum operators for further derivation. For ease of presentation, we use $u^{(l)} = \tau_b^{(l)} + t_b^{(l)} + t_b^{(l-1)}$. Then we have

$$\begin{aligned}
t_{gain} =& \max\{u^{(l)} + \gamma\rho(d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)}) \\
& , \tau_c^{(l)} + \alpha + \beta d^{(l)}\} - \\
& \max\{u^{(l)} + \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}), \tau_c^{(l)}\} \\
& - \beta d^{(l)}.
\end{aligned}$$

To eliminate the two maximum operators in the above t_{gain}, we need to discuss all 4 cases (a maximum operator has 2 cases). Due to the fact that

$$\begin{aligned}
&\gamma\rho(d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)}) \\
&\qquad < \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}), \quad (20)
\end{aligned}$$

we only need to discuss the other 3 cases:

C1.

Q1 $u^{(l)} + \gamma\rho(d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)}) > \tau_c^{(l)} + \alpha + \beta d^{(l)}$ and

Q2 $u^{(l)} + \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}) > \tau_c^{(l)}$.

It yields

$$\begin{aligned}
t_{gain} =& \gamma\rho d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)} \\
& - (d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}) - \beta d^{(l)}. \quad (21)
\end{aligned}$$

It is obvious that $t_{gain} < 0$; so in such case, making layer l a merged-layer cannot bring time saving.

C2.

Q3 $u^{(l)} + \gamma\rho(d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)}) < \tau_c^{(l)} + \alpha + \beta d^{(l)}$ and

Q4 $u^{(l)} + \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}) < \tau_c^{(l)}$.

It yields

$$t_{gain} = \alpha, \quad (22)$$

which is obviously larger than zero. Therefore, we can conclude that making the layer a merged-layer can save execution time under the conditions of **C2**.

C3.

Q3 $u^{(l)} + \gamma\rho(d^{(l)} \log d^{(l)} + d^{(l-1)} \log d^{(l-1)}) < \tau_c^{(l)} + \alpha + \beta d^{(l)}$ and

¬Q4 $u^{(l)} + \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}) > \tau_c^{(l)}$.

It yields

$$t_{gain} = \tau_c^{(l)} + \alpha - u^{(l)} - \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)}).$$

Therefore, only if

Q5 $\tau_c^{(l)} + \alpha > u^{(l)} + \gamma\rho(d^{(l-1)} + d^{(l)}) \log(d^{(l-1)} + d^{(l)})$,

then making layer l a merged-layer can save execution time.

From the above analysis, we can see that only under particular conditions, making layer l a merged-layer can shorten the iteration time. More formally,

$$(\textbf{Q3} \wedge \textbf{Q4}) \vee (\textbf{Q3} \wedge \neg\textbf{Q4} \wedge \textbf{Q5}) \implies l = l_m, \quad (23)$$

which is equivalent to

$$\textbf{Q3} \wedge \textbf{Q5} \implies l = l_m. \quad (24)$$

The statement of (24) indicates that if a layer satisfies **Q3** and **Q5**, then the layer should be a merged-layer to bring the performance gain.

Theorem 1. Given a DNN model with L layers to be trained with MGS-SGD on a P-node cluster. If the computing performance of sparsification satisfies (10) and the communication performance satisfies (11), then we can make any layer l ($1 \leq l \leq L$) to be a merged-layer or a normal layer according to the following equation to achieve the minimal iteration time.

$$l = \begin{cases} l_m, & 1 < l \leq L \text{ and } \mathbf{Q3} \wedge \mathbf{Q5} \text{ is true} \\ l_n, & \text{otherwise} \end{cases}. \quad (25)$$

Proof. Given a specific DNN model and the communication speed of a cluster, we use t^*_{iter} to denote the iteration time when assigning layers $1 \leq l \leq L$ according to the above equation. It is obvious that by changing any layer l from the merged-layer to a normal layer ($t_{gain} = \widetilde{\mu}^{(l)} - \mu^{(l)} < 0$), or changing any layer l from the normal layer to a merged-layer ($t_{gain} = \mu^{(l)} - \widetilde{\mu}^{(l)} < 0$), it would not bring shorter time than t^*_{iter}, which concludes the optimality of t^*_{iter}. □

B. Algorithm

According to Theorem 1, we derive the algorithm to find all merged-layers for a given DNN model. The pseudo-code of the algorithm is shown in Algorithm 1.

In Algorithm 1, there are several procedures including "TopKPerfModel" (TKPM, Lines 15-16), "AllGatherPerfModel" (AGPM, Lines 17-18), "CalculateCommStart" (CCMS, Lines 19-26), "CalculateCompStart" (CCPS, Lines 27-36) and "Merge" (Lines 37-41). The first two are the performance models of the Top-k selection operation on GPUs and the communication model of AllGather for gradients aggregation according to Eq. (10) and (11) respectively. "CalculateCommStart" and "CalculateCompStart" are two functions that calculate the layer-wise beginning timestamps of the communications and computations respectively. 'Merge" is a function that makes a layer be a merged-layer and applies Eqs. (13)-(15). The algorithm first initializes all layers as normal layers (Line 1), and then (Lines 2-3) calculates the start timestamps and estimates the compression time and communication time using "CalculateCommStart" and "CalculateCompStart" procedures. After the prerequisites are computed, the algorithm searches the layers from the last one to the second one to check whether it satisfies **Q3** and **Q5** (Lines 4-7). If a layer l satisfies **Q3** and **Q5**, then a merge operation is executed to update the information and the pre-defined normal layer is changed to a merged-layer (Lines 8-13).

Algorithm 1 has a time complexity of $O(L^2)$, and is only executed once before the training process begins. In Line 4, the algorithm needs to scan from layer L to 2 and check if a layer can be assigned as a merged-layer, which requires $O(L)$ steps. For each check, if the layer can be a merged-layer, then it requires to calculate the start time of computation and communication, which has a complexity of $O(L)$. Therefore, the total time complexity of the algorithm is $O(L^2)$.

Algorithm 1 FindMergedLayers

Input: γ, α, β, ρ, L, $\boldsymbol{tb} = [1...L]$, $\boldsymbol{d} = [d^{(1)}, d^{(2)}, ..., d^{(L)}]$.
Output: $\boldsymbol{m}$

1: Initialize $\boldsymbol{m}[1...L] = [1_n, 2_n, ..., L_n]$;
2: $\boldsymbol{\tau b}$, $\boldsymbol{\tau s}$, $\boldsymbol{ts}$ = CCPS($\boldsymbol{tb}$, $\boldsymbol{d}$, L, 0);
3: $\boldsymbol{\tau c}$, $\boldsymbol{tc}$ = CCMS($\boldsymbol{ts}$, $\boldsymbol{\tau s}$, $\boldsymbol{d}$, L);
4: **for** $l = L \rightarrow 2$ **do**
5: $\mu = \boldsymbol{\tau b}[l] + \boldsymbol{tb}[l] + \boldsymbol{tb}[l-1]$;
6: $Q3 = \mu + \text{TKPM}(\boldsymbol{d}[l]) + \text{TKPM}(\boldsymbol{d}[l-1]) < \boldsymbol{\tau c}[l] + \text{AGPM}(\boldsymbol{d}[l])$;
7: $Q5 = \boldsymbol{\tau c}[l] + \alpha > \mu + \text{TKPM}(\boldsymbol{d}[l] + \boldsymbol{d}[l-1])$;
8: **if** $Q3$ and $Q5$ **then**
9: Merge($\boldsymbol{tb}$, $\boldsymbol{ts}$, $\boldsymbol{tc}$, $\boldsymbol{d}$, l);
10: $\boldsymbol{\tau b2}$, $\boldsymbol{\tau s2}$, $\boldsymbol{ts2}$ = CCPS($\boldsymbol{tb}[1...l]$, $\boldsymbol{d}[1...l]$, l, $\boldsymbol{\tau b}[l]+\boldsymbol{tb}[l]$);
11: $\boldsymbol{\tau b}[1...l] = \boldsymbol{\tau b2}$; $\boldsymbol{\tau s}[1...l] = \boldsymbol{\tau s2}$;
12: $\boldsymbol{\tau c}$, $\boldsymbol{tc}$ = CCMS($\boldsymbol{ts}$, $\boldsymbol{\tau s}$, $\boldsymbol{d}$, L);
13: $\boldsymbol{m}[l] = l_m$;
14: Return $\boldsymbol{m}$;
15: **procedure** TOPKPERFMODEL(d) //TKPM
16: Return $\gamma * \rho * d * \log d$;
17: **procedure** ALLGATHERPERFMODEL(d) //AGPM
18: Return $\alpha + \beta * d$;
19: **procedure** CALCULATECOMMSTART($\boldsymbol{ts}$, $\boldsymbol{\tau s}$, $\boldsymbol{d}$, L) //CCMS
20: Initialize $\boldsymbol{\tau c}[1...L]$, $\boldsymbol{tc}[1...L]$;
21: **for** $l = L \rightarrow 1$ **do**
22: $\boldsymbol{tc}[l] = \text{AGPM}(d[l])$;
23: $\boldsymbol{\tau c}[L] = \boldsymbol{\tau s}[L] + \boldsymbol{ts}[L]$;
24: **for** $l = L-1 \rightarrow 1$ **do**
25: $\boldsymbol{\tau c}[l] = \max\{\boldsymbol{\tau c}[l+1] + \boldsymbol{tc}[l+1], \boldsymbol{\tau s}[l] + \boldsymbol{ts}[l]\}$;
26: Return $\boldsymbol{\tau c}$, $\boldsymbol{tc}$;
27: **procedure** CALCULATECOMPSTART($\boldsymbol{tb}$, $\boldsymbol{d}$, L, τ) //CCPS
28: Initialize $\boldsymbol{\tau b}[1...L]$, $\boldsymbol{\tau s}[1...L]$, $\boldsymbol{ts}[1...L]$;
29: **for** $l = L \rightarrow 1$ **do**
30: $\boldsymbol{ts}[l] = \text{TKPM}(d[l])$;
31: $\boldsymbol{\tau b}[L] = \tau$;
32: $\boldsymbol{\tau s}[L] = \boldsymbol{\tau b}[L] + \boldsymbol{tb}[l]$;
33: **for** $l = L-1 \rightarrow 1$ **do**
34: $\boldsymbol{\tau b}[l] = \boldsymbol{\tau s}[l+1] + \boldsymbol{ts}[l+1]$;
35: $\boldsymbol{\tau s}[l] = \boldsymbol{\tau b}[l] + \boldsymbol{tb}[l]$;
36: Return $\boldsymbol{\tau b}$, $\boldsymbol{\tau s}$, $\boldsymbol{ts}$;
37: **procedure** MERGE($\boldsymbol{tb}$, $\boldsymbol{ts}$, $\boldsymbol{tc}$, $\boldsymbol{d}$, l)
38: $\boldsymbol{tb}[l-1] = \boldsymbol{tb}[l-1] + \boldsymbol{tb}[l]$; $\boldsymbol{tb}[l] = 0$;
39: $\boldsymbol{d}[l-1] = \boldsymbol{d}[l-1] + \boldsymbol{d}[l]$; $\boldsymbol{d}[l] = 0$;
40: $\boldsymbol{tc}[l-1] = \text{AGPM}(\boldsymbol{d}[l-1])$; $\boldsymbol{tc}[l] = 0$;
41: $\boldsymbol{ts}[l-1] = \text{TKPM}(\boldsymbol{d}[l-1])$; $\boldsymbol{ts}[l] = 0$;

Based on the optimal merging strategy, we propose the optimal merged gradient sparsification SGD (OMGS-SGD) training algorithm in Algorithm 2. Compared to MGS-SGD, OMGS-SGD checks if a layer l is defined as a merged-layer (Line 14 and Line 24). If yes, there will be no compression or communication for layer l and its gradients will be buffered (Line 18 and Line 28); otherwise, it invokes the compression and communication using buffered gradients (Lines 15-16 and Lines 25-26).

V. EVALUATION

In this section, we first demonstrate the performance models of Top-k selection and the AllGather communication on the real-world environment, and then present the convergence

Algorithm 2 OMGS-SGD at worker g

Input: $\boldsymbol{D} = [\{X_1, y_1\}, ..., \{X_n, y_n\}]$, net

Initialize shared and synchronized queue Q;
Get $\boldsymbol{m}$ from Algorithm 1;
ASYNCCOMMUNICATION($Q, \boldsymbol{m}$);
while not stop **do**
  Sample a mini-batch of $data$ from $\boldsymbol{D}$;
  ASYNCCOMPUTATION($data, net, Q, \boldsymbol{m}$);
  WaitForLastCommunicationFinished();
  $net.W = net.W - \eta \cdot \nabla net.W$,
procedure ASYNCCOMPUTATION($data, net, Q, \boldsymbol{m}$)
  Initialize gb; //gradient buffer
  ForwardComputation($data$, $net.W$);
  for $l = L \rightarrow 1$ **do**
    BackwardComputation(l);
    if $\boldsymbol{m}[l]$ is l_n **then**
      CompressComputation(gb);
      Clear gb;
    else
      $gb.push(l)$
    Q.push(l);
procedure ASYNCCOMMUNICATION($Q, \boldsymbol{m}$)
  Initialize gb; //gradient buffer
  while *isRunning* **do**
    $l = Q$.pop();
    if $\boldsymbol{m}[l]$ is l_n **then**
      AllGather(gb);
      Clear gb;
    else
      gb.push(l);
    if $l == 1$ **then**
      NotifyLastCommunicationFinished();

performance of our proposed OMGS-SGD compared to the vanilla SGD without sparsification. Finally, we demonstrate and discuss the time performance of OMGS-SGD compared with other existing methods.

A. Experimental Settings

Cluster Configuration: We conduct experiments on a 16-node cluster connected with 1GbE, and each node installs an Nvidia P102-100 GPU. The hardware configuration of each node is shown in Table III. All GPU machines are installed with the Nvidia GPU driver at version 390.48 and CUDA-9.1. The communication libraries are OpenMPI-3.1.1[4] and NCCL-2.1.5[5]. We use the highly optimized distributed training library Horovod[6] [17] at version 1.4.1. The deep learning framework is PyTorch[7] at version 0.4.1 with cuDNN-7.1.

DNN Models: We choose various DNN models including convolutional neural networks (CNNs) and recurrent neural networks (RNNs) on different data sets. Specifically, the chosen DNN models are VGG-16 [28] on Cifar-10 [29] that contains $50,000$ training samples and $10,000$ validation samples, ResNet-50 [30] and Inception-v4 [31] on ImageNet [32] that contains about 1.2 million training examples and $50,000$ validation samples, and a 2-layer LSTM language model (LSTM-PTB) on the PTB [33] data set which contains $923,000$ training samples and $73,000$ validation samples. The hyper-parameters for training are shown in Table IV.

[4]https://www.open-mpi.org/
[5]https://developer.nvidia.com/nccl
[6]https://github.com/uber/horovod
[7]https://pytorch.org/

TABLE III
THE HARDWARE/SOFTWARE CONFIGURATION.

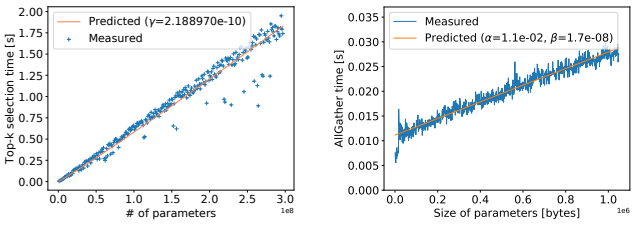
Name	Model
CPU	Intel(R) Celeron(R) CPU N3350 @ 1.10GHz
GPU	Nvidia P102-100 (@1.8GHz and 5GB Memory)
Memory	4GB DDR3 with a 16GB swap file
Disk	256GB SSD
Network	1 Gbps Ethernet (1GbE)
OS	Ubuntu-16.04

TABLE IV
HYPER-PARAMETERS FOR TRAINING DEEP MODELS.

Model	Data Set	# Epochs	Batch Size	ρ	η
VGG-16	Cifar-10	140	128	0.001	0.1
ResNet-50	ImageNet	90	16	0.001	0.01
InceptionV4	ImageNet	90	32	0.001	0.01
LSTM-PTB	PTB	40	100	0.005	1.0

Note: All models are trained with FP32 precision. Batch size is for each worker. ρ is the density and η is the learning rate.

B. Performance Models



(a) Top-k selection on a GPU (b) AllGather on a 16-node cluster

Fig. 2. Performance models.

From Eq. (10), it can be seen that the time cost of Top-k selection depends on γ, ρ and d. ρ and d are the input parameters for a given tensor, while γ is an estimated parameter for a particular GPU. To estimate γ, we measure the time cost on various sizes of tensors. We set $\rho = 0.001$, which is a commonly used density [4], and the number of parameters is in the range of $[1K, 1M]$. The modeling result of P102-100 GPU is shown in Fig. 2(a), and the estimated $\gamma = 2.2^{-10}$.

From Eq. (11), it can be seen that the time cost of AllGather aggregation is related to α, β and the tensor size d. α and β are two environment-aware parameters which could be affected by the number of workers, the bandwidth and latency of the network. Under our configured experimental environments (16 nodes connected with 1GbE), we measure the time cost of AllGather with a range of sizes $[1K, 1M]$ using the benchmark tool nccl-tests[8] which includes both the PCIe and Ethernet transmissions. The modeling results are shown in Fig. 2(b), which also indicates the estimated α and β.

[8]https://github.com/NVIDIA/nccl-tests

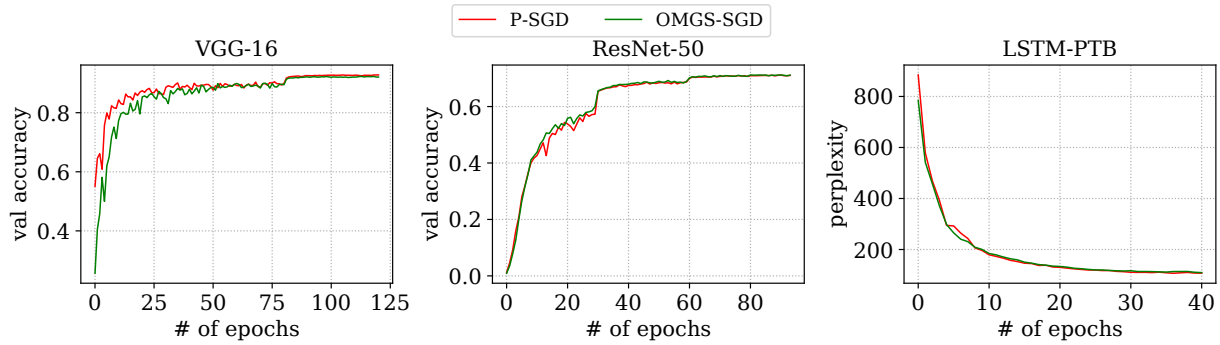


Fig. 3. Comparison of validation performance on three deep models.

C. Convergence Performance

Before demonstrating the wall-clock training time performance, we first present the convergence performance of OMGS-SGD compared to P-SGD. We use $\rho = 0.001$ and $\rho = 0.005$ for CNNs and LSTM respectively. We measure the top-1 validation accuracy (higher is better) for image classification problems (on Cifar-10 and ImageNet) and the perplexity (lower is better) for the language model (on the PTB data set) with respect to the number of epochs running on a 16-GPU cluster. The convergence comparison is shown in Fig. 3 on three deep models trained on three data sets. It can be seen that our proposed OMGS-SGD algorithm achieves nearly consistent validation accuracy with the dense version.

D. Iteration Time

We evaluate the iteration time of the three training algorithms[9] (MGS-SGD with a hand-crafted threshold of $8,192$[10], TopK-SGD and our proposed OMGS-SGD) on a 16-node GPU cluster connected with a 1GbE switch.

The number of final merged layers is different on different neural networks due to the diverse number of layers and various GPU workloads. The number of layers after merging on different deep models and algorithms are shown in Table V. Compared to the threshold merged version of layer-wise algorithm (MGS-SGD), our OMGS-SGD finds the optimal merging solution to the problem under different environments of the clusters adaptively.

The experimental results of iteration time are shown in Table VI. It can been that OMGS-SGD always achieves the best performance among the three compared algorithms.

On the VGG-16 model, OMGS-SGD only achieves slight improvement over TopK-SGD. The main reason is that the computing time of VGG-16 is small due to the small input resolution ($3 \times 28 \times 28$), while the number of parameters is large. Therefore, OMGS-SGD prefers to merge layers because there is little pipelining opportunity. Compared to MGS-SGD, OMGS-SGD achieves 57% improvement. On very deep models with the ImageNet data set, OMGS-SGD is about $1.3\times$ and $2.4\times$ faster than TopK-SGD and MGS-SGD respectively. In particular, OMGS-SGD is up to 31% faster than TopK-SGD on the ResNet-50 model. Regarding the LSTM model, OMGS-SGD achieves 11% improvement over TopK-SGD.

TABLE V
THE NUMBER OF MERGED LAYERS.

Model	# of Layers for Communication			
	TopK-SGD	LAGS-SGD	MGS-SGD	OMGS-S.
VGG-16	1	54	13	2
ResNet-50	1	161	53	2
Inception-v4	1	449	150	7
LSTM-PTB	1	11	9	6

Note: MGS-SGD uses $8,192$ as a threshold to merge layers.

TABLE VI
COMPARISON OF THE AVERAGE ITERATION WALL-CLOCK TIME (IN SECONDS) OF 1,000 RUNNING ITERATIONS.

Model	TopK-SGD	MGS-S.	OMGS-S.	s1	s2
VGG-16	0.336	0.523	0.331	1.015	1.578
ResNet-50	0.666	1.217	0.507	1.313	2.4
Inception-v4	1.602	3.389	1.247	1.284	2.717
LSTM-PTB	1.019	0.956	0.916	1.113	1.042

Note: "s1" and "s2" represent the speedup of OMGS-SGD over TopK-SGD and MGS-SGD, respectively. MGS-SGD uses $8,192$ as a threshold to merge layers.

E. Time Breakdown

To understand the details of the improvement achieved by OMGS-SGD, we breakdown the time of one iteration into forward, backward, sparsification and non-overlapped communications. The time breakdown is shown in Fig. 4. As we can see that on VGG-16, the non-overlapped communication overhead of OMGS-SGD is very close to TopK-SGD so that it has only very small improvement. On the very deep models (ResNet-50 and Inception-v4), the non-overlapped communication time is shorten by about $2\times$ using OMGS-SGD. On the LSTM model, which is also relatively shallow, OMGS-SGD achieves about 18% shorter communication time than TopK-SGD. Overall, the computation time of sparsification has also slight improvements. However, the Top-k selection

[9]P-SGD is excluded as it runs very slow with dense gradients on 1GbE.

[10]Since current deep learning frameworks use tensors to represent parameters, it could have two tensors (bias and weight) for a layer, while the size of the bias tensor is much smaller than the weight tensor. Hence, the hand-crafted threshold is necessary to merge small tensors, otherwise the time performance will be extremely bad. So we do not show the time performance of LAGS-SGD; instead we use a threshold of $8,192$ for MGS-SGD.

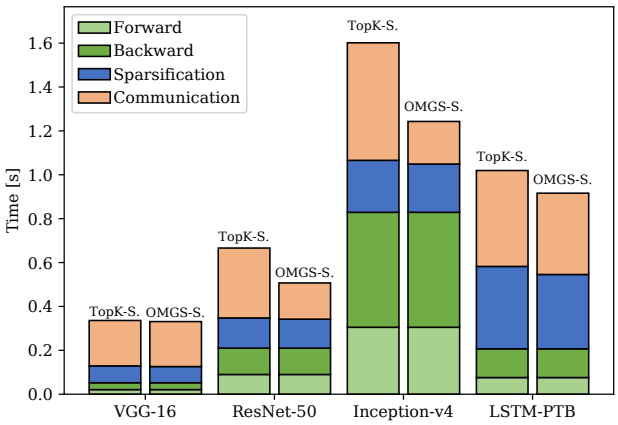


Fig. 4. Time breakdowns on four models with TopK-SGD and OMGS-SGD.

algorithm on the GPU is not efficient enough as it could have larger cost than forward and backward computations. Some efficient gradient sparsification methods are imperative to further increase the training efficiency. We will leave this as our future work.

VI. Related Work

There are several kinds of techniques to reduce the communication overhead in synchronous distributed SGD on deep learning applications. We classify them into three categories.

Efficient Collectives: The first one is from the HPC research community that provides the efficient communication algorithms that try to maximally utilize the network bandwidth, for example, ring-based AllReduce [34], optimized communication collectives [35][36][14][24][37] and productive communication libraries NCCL and Gloo[11]. The key ideas of the efficient communication algorithms are mainly exploiting the structure of messages (e.g., message size) and the network topology (e.g., Torus topology) to better utilize the network bandwidth and reduce the negative impact from the network latency.

Traffic Reduction: The second one is the algorithmic level approaches that try to reduce the communication traffic, e.g., gradient quantization [38][6][5][39], gradient sparsification [40][3][41][4][42][43] and delayed communication [44][45]. The gradient quantization techniques exploit the low precision representation (e.g., 16-bit floating points) to transmit the numbers, which could result in a maximum of $32\times$ traffic reduction when using 1-bit representation. Gradient sparsification is an orthogonal approach to quantization, and its key idea is transmitting the "significant" gradients (e.g., Top-k, where k could be 0.1% of all gradients) on each communication round.

Communication Scheduling: The third one is the communication scheduling under the same communication traffic. One straightforward method is to increase the workload of computation, which is also named large-batch training [2][15][46][18], such that the communication-to-computation ratio is as low as possible. Due to the layer-wise structure of deep models and the backpropagation algorithm, the communication and computation can be pipelined [14][13][47] to further reduce the impact of communications. With increasing number of workers, the startup time of communications could dominate the whole communication time, especially for small tensors. Hence merged gradients or tensor fusion techniques [17][16] have been recently applied to reduce the communication time. In [16], the authors proposed an optimal merging solution for the WFBP algorithm on the dense gradients, which is the most similar to our work. However, due to the extra overheads from gradient compression, the existing merging solution cannot be applied on LAGS-SGD. In LAGS-SGD, besides the backpropagation computations and gradient communications, it has extra computing overheads on gradient sparsification. As a result, one should consider the three variables together to determine whether the merging could bring performance gain. To this end, this paper explores the trade-off among three variables (backpropagation computations, sparsification computations and communications) as an optimization problem and propose an optimal solution.

[11]https://github.com/facebookincubator/gloo

VII. Conclusion and Future Work

In this paper, we first showed that the current Top-k gradient sparsification techniques and pipelining between computations and communications are sub-optimal in terms of time performance in distributed synchronous SGD algorithms. Even if the gradients are significantly sparsified to reduce the communication traffic, some small messages are required to be merged together before transmitting across the network to reduce the overall communication time. The merging operation, however, would increase the computation time of gradient sparsification and the waiting time for the computation. We formulated the gradient merging problem to trade off communications and computations (including gradient sparsification) as an optimization problem, and derived an efficient optimal solution with theoretical guarantees. According to the solution, which can be computed with a small overhead before training, we proposed the OMGS-SGD algorithm for distributed training of deep learning. We conducted experiments on a 16-node cluster connected with 1GbE links, where each node is equipped with an Nvidia GPU. Experimental results showed that our proposed OMGS-SGD outperforms existing Top-k based distributed SGD algorithms with little impact on the model accuracy.

For the future work, it is worthy to explore efficient gradient sparsification algorithms to distributed deep learning such that one can further improve the training efficiency.

Acknowledgements

The research was supported in part by Hong Kong RGC GRF grants under the contracts HKBU 12200418, HKUST 16206417 and 16207818. We would also like to thank Nvidia AI Technology Centre (NVAITC) for providing the GPU clusters for some experiments.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch SGD: Training ImageNet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[3] A. F. Aji and K. Heafield, "Sparse communication for distributed gradient descent," in *Proc. of EMNLP*, 2017, pp. 440–445.

[4] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in *Proc. of International Conference on Learning Representations*, 2018.

[5] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Proc. of Advances in Neural Information Processing Systems*, 2017, pp. 1509–1519.

[6] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-efficient SGD via gradient quantization and encoding," in *Proc. of Advances in Neural Information Processing Systems*, 2017, pp. 1707–1718.

[7] P. Jiang and G. Agrawal, "A linear speedup analysis of distributed deep learning with sparse and quantized communication," in *Proc. of Advances in Neural Information Processing Systems*, 2018, pp. 2530–2541.

[8] S. Shi, K. Zhao, Q. Wang, Z. Tang, and X. Chu, "A convergence analysis of distributed SGD with communication-efficient gradient sparsification," in *Proc. of The 28th IJCAI*, 2019, pp. 3411–3417.

[9] A. Shanbhag, H. Pirk, and S. Madden, "Efficient Top-K query processing on massively parallel hardware," in *Proc. of The International Conference on Management of Data*, 2018, pp. 1557–1570.

[10] S. Shi, X. Chu, K. C. Cheung, and S. See, "Understanding top-k sparsification in distributed deep learning," *arXiv preprint arXiv:1911.08772*, 2019.

[11] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *Proc. of The 7th International Conference on Cloud Computing and Big Data*. IEEE, 2016, pp. 99–104.

[12] W. Wang and N. Srebro, "Stochastic nonconvex optimization with large minibatches," in *Proc. of The 30th International Conference on Algorithmic Learning Theory*, 2019.

[13] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *Proc. of USENIX ATC*, 2017, pp. 181–193.

[14] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters," in *Proc. of The 22nd ACM PPoPP*, 2017.

[15] Y. You, A. Buluç, and J. Demmel, "Scaling deep learning on GPU and Knights Landing clusters," in *Proc. of SC'17*, 2017.

[16] S. Shi, X. Chu, and B. Li, "MG-WFBP: Efficient data communication for distributed synchronous SGD," in *Proc. of IEEE INFOCOM*. IEEE, 2019, pp. 172–180.

[17] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[18] X. Jia, S. Song, S. Shi, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, and X. Chu, "Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes," in *Proc. of Workshop on Systems for ML and Open Source Software, collocated with NeurIPS 2018*, 2018.

[19] S. Shi, Z. Tang, Q. Wang, K. Zhao, and X. Chu, "Layer-wise adaptive gradient sparsification for distributed deep learning with convergence guarantees," in *Proc. of The 24th European Conference on Artificial Intelligence*, 2020.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. of USENIX OSDI*, 2016, pp. 265–283.

[21] 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 *Proc. of NIPS Autodiff Workshop*, 2017.

[22] S. Shi, W. Qiang, and X. Chu, "Performance modeling and evaluation of distributed deep learning frameworks on GPUs," in *Proc. of The 4th International Conference on Big Data Intelligence and Computing*. IEEE, 2018.

[23] Y. Wang, W. Wang, S. Shi, X. He, Z. Tang, K. Zhao, and X. Chu, "Benchmarking the performance and power of AI accelerators for AI training," *arXiv preprint arXiv:1909.06842*, 2019.

[24] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler, "SparCML: High-performance sparse communication for machine learning," in *Proc. of SC'19*, 2019, pp. 1–15.

[25] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[26] S. Sarvotham, R. Riedi, and R. Baraniuk, "Connection-level analysis and modeling of network traffic," in *The 1st ACM SIGCOMM Workshop on Internet Measurement*. ACM, 2001, pp. 99–103.

[27] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over commodity ethernet at scale," in *Proc. of ACM SIGCOMM*, 2016, pp. 202–215.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[29] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)," *URL http://www.cs.toronto.edu/kriz/cifar.html*, 2010.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of CVPR*, 2016, pp. 770–778.

[31] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. of The 31st AAAI*, 2017.

[32] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. of CVPR*, 2009, pp. 248–255.

[33] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of English: The Penn Treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[34] A. Gibiansky, "Bringing HPC techniques to deep learning.(2017)," *URL http://research. baidu. com/bringing-hpc-techniques-deep-learning*, 2017.

[35] V. Turchenko, L. Grandinetti, G. Bosilca, and J. J. Dongarra, "Improvement of parallelization efficiency of batch pattern BP training algorithm using Open MPI," *Procedia Computer Science*, vol. 1, no. 1, pp. 525–533, 2010.

[36] A. A. Awan, K. Hamidouche, A. Venkatesh, and D. K. Panda, "Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning," in *Proc. of The 23rd European MPI Users' Group Meeting*, 2016, pp. 15–22.

[37] 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 *Proc. of IEEE INFOCOM*, 2019, pp. 1729–1737.

[38] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and application to data-parallel distributed training of speech DNNs," in *Proc. of INTERSPEECH*, September 2014.

[39] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "SIGNSGD: Compressed optimisation for non-convex problems," in *Proc. of International Conference on Machine Learning*, 2018, pp. 559–568.

[40] N. Strom, "Scalable distributed DNN training using commodity GPU cloud computing," in *Proc. of INTERSPEECH*, 2015.

[41] C.-Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, and K. Gopalakrishnan, "Adacomp: Adaptive residual gradient compression for data-parallel distributed training," in *Proc. of The 32nd AAAI*, 2018.

[42] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," in *Proc. of Advances in Neural Information Processing Systems*, 2018, pp. 1299–1309.

[43] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu, "A distributed synchronous SGD algorithm with global Top-k sparsification for low bandwidth networks," in *Proc. of The 39th IEEE ICDCS*, 2019.

[44] T. Lin, S. U. Stich, and M. Jaggi, "Don't use large mini-batches, use local SGD," *arXiv preprint arXiv:1808.07217*, 2018.

[45] S. U. Stich, "Local SGD converges fast and communicates little," in *Proc. of International Conference on Learning Representations*, 2019.

[46] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "ImageNet training in minutes," in *Proc. of the 47th ICPP*, 2018, pp. 1–10.

[47] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, "Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training," in *Proc. of Advances in Neural Information Processing Systems*, 2018, pp. 8045–8056.