MG-WFBP: Merging Gradients Wisely for Efficient Communication in Distributed Deep Learning

Shaohuai Shi, Student Member, IEEE, Xiaowen Chu, Senior Member, IEEE, and Bo Li, Fellow, IEEE

Abstract—Distributed synchronous stochastic gradient descent has been widely used to train deep neural networks (DNNs) on computer clusters. With the increase of computational power, network communications generally limit the system scalability. Wait-free backpropagation (WFBP) is a popular solution to overlap communications with computations during the training process. In this paper, we observe that many DNNs have a large number of layers with only a small amount of data to be communicated at each layer in distributed training, which could make WFBP inefficient. Based on the fact that merging some short communication tasks into a single one can reduce the overall communication time, we formulate an optimization problem to minimize the training time in pipelining communications and computations. We derive an optimal solution that can be solved efficiently without affecting the training performance. We then apply the solution to propose a distributed training algorithm named merged-gradient WFBP (MG-WFBP) and implement it in two platforms Caffe and PyTorch. Extensive experiments in three GPU clusters are conducted to verify the effectiveness of MG-WFBP. We further exploit the trace-based simulation of 64 GPUs to explore the potential scaling efficiency of MG-WFBP. Experimental results show that MG-WFBP achieves much better scaling performance than existing methods.

Index Terms—Deep Learning; GPU; Distributed Stochastic Gradient Descent; Gradient Communication; Merged-gradient

1 INTRODUCTION

The data-parallel synchronous stochastic gradient descent (S-SGD) method is commonly used as the optimizer to train large-scale deep neural networks (DNNs) [1][2]. In S-SGD, the computing tasks for each mini-batch of training data are distributed to a cluster of computing nodes, and the individual results (e.g., gradients) are aggregated to update the global network model before the next iteration begins. However, with more computing nodes and the fast-growing computing power of hardware accelerators, the data communication between computing nodes gradually becomes the performance bottleneck [3][4][5]. For example, the computing power of Nvidia GPUs has increased by 30x in the last 10 years, whilst it took about 15 years for the network speed to improve from 10Gbps to 100Gbps. Hence it becomes a critical issue to address the imbalance between computing and communication.

Some recent works try to reduce the impact of data communication at either algorithmic or system level. On one hand, gradients could be quantized or sparsified [6][7][8][9][10] in order to reduce the amount of data to be exchanged so that the communication time could be reduced. But these methods usually sacrifice the training convergence speed. On the other hand, the high-performance computing (HPC) community has proposed several methods to improve the communication performance of the cluster by optimizing the hardware or communication software library [11][12]. In terms of hardware, InfiniBand (IB) and Omni-Path networks can provide much higher communication bandwidth and lower latency, and are deployed to shorten the performance gap between communication and computation [13]. Regarding the software, the implementation of message passing interface (MPI) has been optimized to support more efficient communication in DNN training [14][15]. Nvidia’s NCCL [16] is another highly optimized communication library for deep learning frameworks on multi-GPU settings. The scaling efficiency of distributed deep learning systems can be modeled as a function of the communication-to-computation ratio [17]. For example, training ResNet-50 [18] requires about 7.8 billion floating point operations in computation, while it needs to all-reduce 102 MB of data in one iteration. Higher communication-to-computation ratio results in lower scaling efficiency.

The layered structure of DNNs makes it possible to overlap the communication and computation during the backward propagation [15][17][18], which is known as wait-free backpropagation (WFBP). WFBP begins to exchange the gradients of a layer immediately after they have been calculated; so if the data communication time of a layer is shorter than the computation time of the gradients of its previous layer, then this communication cost can be fully hidden. However, if very fast hardware accelerators are used while the network speed is relatively slow (i.e., a high communication-to-computation ratio), there can exist many layers whose communication time is longer than the corresponding computation time. In such case, it becomes important to optimize the communications.
that the layer-wise gradient communication in WFBP is suboptimal due to the fact that all-reducing a small amount of data cannot fully utilize the network bandwidth in current network topology due to the startup time of message transmitting (or transmission latency). For example, on our 10GbE platform, all-reducing a set of 200 KB vectors across 8 nodes using MPI requires about 1.5 ms, while all-reducing a set of 400 KB vectors only requires 1.8 ms, which means that if we merge the two sets of 200 KB vectors to a single set of 400 KB vectors, then the total communication time can be reduced from 3 ms to 1.8 ms. The same phenomena can also be found in RDMA-based networks[19][20]. You et al. [21] have also noticed this problem, and proposed a single-layer communication (SyncEASGD) method which merges the gradients of different layers into a single tensor and then transfers only once per iteration. As compared to the layer-wise communication in WFBP, it can eliminate most of the startup time of data communications. But in their proposed method, gradient communication can only start after the backward propagation, thus it misses the opportunity of overlapping the communication with computation.

We argue that the best way to reduce the training time needs to consider not only how to overlap communication with computation, but also how to improve the communication efficiency by avoiding transmitting small messages.

In this paper, we first formulate the communication scheduling problem in S-SGD as an optimization problem that aims to minimize the total training time of an iteration. We then propose a merged-gradient wait-free backward propagation (MG-WFBP) method and prove its optimality. The time complexity of MG-WFBP is $O(L^2)$ where $L$ is the number of layers (or tensors) in the DNN, and it only needs to be executed once before the whole training process. We implement MG-WFBP atop the popular deep learning frameworks Caffe[22] and PyTorch[23], and make it publicly available.

To validate the effectiveness of our proposed MG-WFBP, we evaluate its performance using various DNNs on multi-GPU settings with both 10Gbps Ethernet (10GbE) and 56Gbps InfiniBand (56GbIB) interconnects. On the relatively slow Nvidia Tesla K80 GPU clusters with 10GbE, MG-WFBP achieves about 1.2x to 1.36x improvement than the state-of-the-art communication algorithms WFBP and SyncEASGD, respectively. On the latest Nvidia Tesla V100 GPU clusters with 10GbE or 56GbIB, MG-WFBP achieves an average of 18.8% faster than WFBP and SyncEASGD in terms of end-to-end training time. To investigate its performance on large clusters, we resolve to Nvidia Tesla V100 GPU clusters with 10GbE Ethernet (10GbE) and 56Gbps InfiniBand (56GbIB) interconnects. The rest of the paper is organized as follows. We present the preliminaries in Section 2 followed by the formulation of the existing problem in Section 3. We derive an optimal solution to the problem and then present our MG-WFBP algorithm in Section 4. The system implementation atop PyTorch is present in Section 5. Section 6 demonstrates the experimental studies on the proposed method compared to existing methods. Section 7 introduces the related work, and finally we conclude this paper in Section 8.

2 Preliminaries

For ease of presentation, we summarize the frequently used mathematical notations in Table 1.

| Name | Description |
|------|-------------|
| $N$  | The number of computing nodes in the cluster. |
| $\alpha$ | Latency (startup time) of the network between two nodes. |
| $\beta$ | Transmission time per byte between two nodes. |
| $\gamma$ | Summation time of two floating point numbers in one node. |
| $a$  | Latency (startup time) of all-reduce. |
| $b$  | Transmission and computation time per byte of all-reduce. |
| $M$  | The size of a message in bytes. |
| $W$  | Weights of the DNN. |
| $D_i^0$ | The input data size for the $i^{th}$ node of the $i^{th}$ mini-batch. |
| $L$  | The number of learnable layers (or tensors) of a DNN. |
| $p_l^{(i)}$ | The number of parameters in the learnable layer $l$. |
| $t_{iter}$ | Time of one training iteration with one batch of data. |
| $t_f$ | Time of the forward pass in each iteration. |
| $t_b$ | Time of the backward propagation in each iteration. |
| $t_m$ | Time of the model update in each iteration. |
| $t_k^{(i)}$ | The timestamp when layer $l$ begins to communicate gradients. |
| $t_g^{(i)}$ | The timestamp when layer $l$ begins to compute gradients. |
| $t_c^{(i)}$ | Time of gradient aggregation in each iteration. |
| $t_{ag}^{(i)}$ | The non-overlapped communication cost in each iteration. |

2.1 Mini-batch SGD

Consider an $L$-layer DNN with a loss function $\mathcal{L}(W,D)$ which defines the difference between the prediction values and the ground truth over the training data set $D$, where $W$ is the set of model weights. To minimize the loss function, the mini-batch SGD updates the parameters iteratively. Typically, the $i^{th}$ iteration of the training includes four steps: 1) A mini-batch of data $D_i$ ($D_i \subset D$) is read as inputs of the DNN. 2) $D_i$ is fed forward across the neural network from layer 1 to layer $L$ to compute the prediction values, and finally the loss function $\mathcal{L}(W,D)$ is computed. 3) The first order gradients w.r.t. parameters and inputs are calculated and backpropagated from layer $L$ to layer 1. 4) Finally, the parameters are updated with the layer-wise gradients.

The training is terminated when some stopping criteria are satisfied. The update of $W$ can be formulated as follows:

$$W_{i+1} = W_i - \eta \cdot \nabla \mathcal{L}(W_i, D_i),$$ (1)
where $\eta$ is the learning rate of SGD, $W_i$ denotes the weights at the $i^{th}$ iteration, and $\nabla L(W_i, D_i)$ denotes the gradients. The time consumed in the training process is mainly in steps 2 and 3, because step 1 of the $i^{th}$ iteration can be scheduled to overlap with the $(i-1)^{th}$ iteration, and the time of step 4 is negligible. Therefore, we can simplify the timeline of SGD as a forward pass followed by a backward passes. The time of one iteration is represented by $t_{iter} = t_f + t_b$, where $t_f$ is the time of the forward pass, and $t_b$ is the time of the backward pass.

The averaging operation of gradients across the cluster involves extra computation and communication overheads. As a side-effect, it is not easy to achieve linear scaling in the distributed SGD training. The timeline of the naive S-SGD (i.e., computation and communication are not overlapped) with communication overheads is illustrated in Fig. 1a. The naive S-SGD algorithm suffers from the waiting period of data communication of model synchronization at every iteration. In practice, the gradients of a layer is stored as a tensor; hence the averaging process can be implemented by many all-reduce operations, once per layer. The layer-wise nature introduces many startup times for layer-wise gradients when they are communicated. The iteration time of the naive S-SGD can be estimated as

$$t_{iter} = t_f + t_b + t_c,$$

where $t_b = \sum_{l=1}^{L} t_b(l)$ is the layer-wise backward propagation time and $t_c = \sum_{l=1}^{L} t_c(l)$ is the layer-wise gradient aggregation time which heavily relies on the communication performance.

Considering S-SGD running on $N$ workers, we define the speedup of S-SGD compared to the vanilla single-worker SGD:

$$S(N) = \frac{N |D_f^P|/(t_f + t_b + t_c)}{|D_f^P|/(t_f + t_b)} = \frac{N}{1 + \frac{t_c}{t_f + t_b}},$$

where $|D_f^P|$ is the number of training samples per worker at the $i^{th}$ iteration. Let $r = \frac{t_c}{t_f + t_b}$, which reflects the communication-to-computation ratio, we have

$$S(N) = \frac{N}{1 + r}.$$

### 2.3 WFBP-SGD

In WFBP S-SGD (WFBP-SGD), the gradient communication of layer $l$ ($l > 1$) can be overlapped with the backward propagation of layer $l-1$. The timeline of WFBP-SGD is illustrated in Fig. 1b. For simplicity, we assume that the start timestamp of the forward pass is 0, and the start timestamp of the backward pass is $\tau_b^{(L)} = t_f$. Then the timestamp when layer $l$ begins to calculate the gradients, denoted by $\tau_b^{(l)}$, can be calculated by:

$$\tau_b^{(l)} = \begin{cases} t_f + t_b, & l = L \\ \tau_b^{(l+1)} + t_b^{(l+1)}, & 1 \leq l < L \end{cases}$$

Notice that the communication of gradients of layer $l$ ($l < L$) can only begin if the following two conditions are satisfied: (1) the gradients of layer $l$ have been calculated; (2) the communication of gradients of layer $(l+1)$ has finished. So, the timestamp when layer $l$ begins the communication of gradients, denoted by $\tau_c^{(l)}$, can be calculated by:

$$\tau_c^{(l)} = \begin{cases} \tau_b^{(l)} + t_b^{(l)} + t_c^{(l+1)}, & l = L \\ \max\{\tau_c^{(l+1)} + t_c^{(l+1)}, \tau_b^{(l)} + t_b^{(l)}\}, & 1 \leq l < L \end{cases}$$

The iteration time of WFBP-SGD can be calculated as

$$t_{iter} = t_f + t_b^{(L)} + (\tau_c^{(1)} - \tau_c^{(L)}) + t_c^{(1)}$$

$$= t_c^{(1)} + \max\{\tau_c^{(2)}, \tau_c^{(2)}, \tau_b^{(1)} + t_b^{(1)}\}.$$
Since some communications are overlapped with the computation, the non-overlapped communication cost, \( t_{\text{non}} \), becomes the bottleneck of the system. In WFBP-SGD, we redefine \( r = \frac{t_{\text{non}}}{t_f + t_b} \), so the main problem of WFBP-SGD is that when the communication cannot be fully overlapped by computation, i.e., \( r_{c(l+1)} + t_c(l+1) > r_b(l) + t_b(l) \), \( t_{\text{non}} \) will limit the system scalability.

### 2.4 Single-Layer S-SGD

As layer-wise communications introduce many startup times especially for large-scale clusters, the startup times dominate the communication time so that overlapping communications and computations may lead to even worse scaling efficiency. Therefore, You et al. \[21\] propose a single-layer communication mechanism (SyncEASGD) which merges all gradients to be communicated by a single all-reduce operation at the end of each iteration, as shown in Fig. 1(c). The iteration time of SyncEASGD can be estimated as

\[
\tau_{\text{iter}} = t_f + t_b + t_c,
\]

where \( t_c \) is composed by the startup time and the transmission time.

#### 2.5 Communication Model

In Eq. (2), we use \( \Delta W_i = \sum_{g=1}^{N} \nabla L(W_i, D_i^g) \) to represent the aggregation of gradients from \( N \) workers, which is an all-reduce operation. There are many optimized algorithms for the all-reduce operation with different number of processes and message sizes \[25,26,27\]. To simplify the problem, we assume that the number of workers is power-of-two, and the peer-to-peer communication cost is modeled as \( \alpha + \beta M \), where \( \alpha \) is the latency component (or called start-up time), \( \beta \) is the communication speed, and \( M \) is the message size. Without loss of generality, we do not limit the communication model to one specific algorithm. Given \( N \) workers, the time cost of all-reduce can be generalized as

\[
T_{\text{ar}}(M) = c + b \times M,
\]

where \( a \) and \( b \) are two constants that are not dependent on \( M \). Some well optimized all-reduce algorithms are summarized in Table 2.

| All-reduce Algorithm     | \( a \)                        | \( b \)                        |
|--------------------------|-------------------------------|-------------------------------|
| Binary tree              | \( 2\alpha \log N \)          | \( (2\beta + \gamma) \log N \) |
| Recursive doubling       | \( \alpha \log N \)           | \( (\beta + \gamma) \log N \)  |
| Recursive halving/doubling| \( 2\alpha \log N \)          | \( 2\beta - \frac{\gamma}{N} (2\beta + \gamma) + \gamma \) |
| Ring                     | \( 2(N-1)\alpha \)           | \( \frac{4\gamma N - \beta + \frac{\gamma}{N-1} N^2}{N} \) |

With a given hardware configuration (i.e., \( N, \alpha, \beta, \) and \( \gamma \) are fixed), the time cost of the all-reduce operation is a linear function of the message size \( M \) with a y-intercept \( a \) and a slope \( b \). We empirically validate this linear model in Section 6.2.

One important property of WFBP-SGD is that the messages are communicated layer by layer, which means that it needs to invoke many all-reduce operations. In each all-reduce operation, however, there is an extra cost of \( a \) which is not related to \( M \). Importantly, the linear function with a positive y-intercept value has a property of

\[
T_{\text{ar}}(M_1) + T_{\text{ar}}(M_2) > T_{\text{ar}}(M_1 + M_2).
\]

In other words, communicating a single message of size \( M_1 + M_2 \) is more efficient than communicating a message of size \( M_1 \) and a message of size \( M_2 \) separately.

### 3 Problem Formulation

Eq. (11) indicates that merging the gradients can improve the communication efficiency. If one merges all layers into one layer so that the communication is only invoked once (i.e., the single-layer communication \[21\]), then the overall communication time is minimal. However, the single-layer communication requires all gradients to be calculated first, which prohibits the overlap between communications and computations. Therefore, we would like to merge the layers appropriately so that it not only reduces the communication by merging, but also exploits the pipelining between communications and computations.

Before formulating the problem, we formally define the concept of merged-gradient layer as follows.

**Definition 1. (Merged-gradient layer).** A layer \( l \) is called a merged-gradient layer if at the timestamp of \( \tau_c(l) \), instead of communicating the gradients of that layer, we merge its gradients to layer \( l - 1 \) and postpone the communication. The operator \( \oplus \) defines the gradients merging between two consecutive layers, say \( l \oplus (l - 1) \). Merging more than two layers is possible by setting consecutive layers into merged-gradient layer.

**Definition 2. (Normal layer).** If a layer \( l \) is not a merged-gradient layer, then it is called a normal layer and its gradients will not be merged into layer \( l - 1 \). Its gradients (including those merged from other layers if any) should be communicated as earlier as possible, i.e., when its own gradients have been calculated and the previously scheduled communication has finished.

There are several properties if layer \( l \) is a merged-gradient layer.

- \( l > 1 \), since the first layer of the DNN cannot be a merged-gradient layer according to the definition.
- There is no communication dedicated for layer \( l \), i.e.,

\[
\tau_c(l) = 0.
\]
- The number of updated parameters of layer \( l - 1 \) becomes the summation of that of layer \( l \) and layer \( l - 1 \).

\[
p^{(l-1)} = p^{(l-1)} + p^{(l)}.
\]
- The timestamp when layer \( l - 1 \) can begin the gradient communication is updated to

\[
\tau_c^{(l-1)} = \max\{\tau_c^{(l)}, \tau_b^{(l-1)} + t_b^{(l-1)}\}.
\]

Intuitively, if merging the gradients of two consecutive layers can save time, then we should merge the two layers. In the following, we discuss a complete set of four cases of computation and communication patterns that may happen.
Backward computation
Communication
Startup time
Transmission time

(a) Case 1.

(b) Case 2.

(c) Case 3.

(d) Case 4.

Fig. 2: Four cases of gradient communication at one iteration on layer $l$ in WFBP-SGD. Note that the forward computation is not plotted as it is not related to the pipelining timeline during the training process with WFBP for layer $l$. The four cases with potential merging are illustrated in Fig. 2.

**Case 1.** In the ideal case, the communication of layer $l$ is fully hidden by its previous layer’s computation, that is

$$\tau_c(t) + \tau_t(t) \leq \tau_b(t-1) + \tau_b(t-1).$$

The overhead of gradient communication is totally hidden by computation so that it is not necessary to merge the gradients.

**Case 2.** The communication of layer $l$ is partially overlapped with the computation of layer $l - 1$, and the communication of layer $l$ begins before the end of the computation of layer $l - 1$, that is

$$\tau_c(t) + \tau_t(t) > \tau_b(t-1) + \tau_b(t-1) > \tau_c(t).$$

Without merging, the communication of layer $l$ can immediately begin after the gradients of layer $l$ have been calculated, i.e., $\tau_c^{(l-1)} = \tau_c^{(l)} + \tau_c^{(l)}$. On the other hand, if we want to merge layer $l$ with layer $l - 1$, the communication can only happen after the gradients of layer $l - 1$ have been calculated. So we should consider whether merging layer $l$ and $l - 1$ could bring any benefits or not. As shown in Fig. 2(b), the merged communication cost takes shorter time to finish, which indicates that the reduced time by merging is greater than the additional waiting time for the gradient computation of layer $l - 1$. Formally,

$$\tau_b^{(l-1)} + \tau_b^{(l-1)} - \tau_c(t) < T_{ar} p(t) + p(t-1) - (T_{ar} p(t) + T_{ar} p(t-1)) = a. \tag{17}$$

In this case, we prefer to merge the gradients of layer $l$ to layer $l - 1$, i.e., making layer $l$ be a merged-gradient layer.

**Case 3.** In this case, the communication of layer $l$ is also partially overlapped with the computation of $l - 1$ as Case 2. However, different from Case 2, the merging operation results in a longer time because the reduced communication time is not as significant as the additional waiting time. To be specific,

$$\tau_c^{(l)} + \tau_t(t) > \tau_b^{(l-1)} + \tau_b^{(l-1)} > \tau_c^{(l)}, \quad \tag{18}$$

and

$$\tau_b^{(l-1)} + \tau_b^{(l-1)} - \tau_c(t) \geq T_{ar} p(t) + p(t-1) - (T_{ar} p(t) + T_{ar} p(t-1)) = a. \tag{19}$$

Therefore, we would not make layer $l$ be a merged-gradient layer because merging the gradients of layer $l$ to layer $l - 1$ will decrease the time efficiency.

**Case 4.** Very different from the previous cases, there is no overlap between the communication of layer $l$ and the computation of layer $l - 1$ as shown in Fig. 2(d). This happens when the previous communication time is longer than the previous computation time. That is,

$$\tau_c^{(l)} \geq \tau_b^{(l-1)} + \tau_b^{(l-1)}. \tag{20}$$

In this case, the communications of layer $l$ and layer $l - 1$ do not need to wait for the end of the computation of layer $l - 1$; hence merging gradients of layer $l$ to layer $l - 1$ does not introduce any waiting time for the computation, which would obviously reduce the communication time, i.e.,

$$T_{ar} p(t) + p(t-1) - (T_{ar} p(t) + T_{ar} p(t-1)) = a > 0. \tag{21}$$

Thus, we would like to make layer $l$ be a merged-gradient layer in this case.

From the above discussions, we can see that not all gradient merging can bring benefits of reduced iteration time (e.g., Case 3). Therefore, our problem is to find all merged-gradient layers such that the overall iteration time is minimal. Since a layer is either a normal-layer or a merged-gradient layer, we use $l_n$ and $l_m$ to denote the type of normal-layer and the merged-gradient layer respectively. Let the variable $e(l)$ denote the type of layer $l (l = 1, 2, \ldots, L)$, $e(l) \in \{l_n, l_m\}$. For an $L$-layer DNN model, it can be represented by

$$\mathcal{M} = \{e(1), \ldots, e(L)\} | e(l) \in \{l_n, l_m\} \text{ and } 1 \leq l \leq L \}. \tag{22}$$

Obviously, the number of combinations of normal layers and merge-gradient layers is $|\mathcal{M}| = 2^L$. Therefore, our goal is to find an $m \in \mathcal{M}$ such that the iteration time is minimal.
Assuming the linear communication model of Eq. (10), the communication time of each layer is represented by

\[ t_c^{(l)} = T_{ar}(p^{(l)}) . \]  

(23)

For a given DNN training with a specific mini-batch size on a hardware environment, the computation time of one iteration can be easily measured at the beginning of training. Since the architecture of the DNN would not change during the training, the feed-forward and backward propagation computation time is very stable [29]. That is, \( t_c^{(l)} \) is known for \( l = 1, 2, ..., L \). However, the beginning timestamp (\( \tau_c^{(l)} \)) and the communication time (\( t_c^{(l)} \)) of layer \( l \) will be different when \( e^{(l)} = l_n \) or \( e^{(l)} = l_m \) as we discussed before. Therefore, we generalize the problem as follows.

For a given \( L \)-layer DNN trained with WFBP-SGD on a specific cluster with \( P \) workers, we would like to determine \( e^{(l)} \) to be \( l_n \) or \( l_m \) such that the iteration time of training is minimal. Formally, we would like to minimize the iteration time of WFBP-SGD in Eq. (5), i.e.,

minimize: \( t_{iter} = t_c^{(1)} + \max \{ \tau_c^{(2)} + t_c^{(2)}, \tau_b^{(1)} + t_b^{(1)} \} . \)  

(24)

4 SOLUTION: MG-WFBP

In this section, we first perform some theoretical analysis on the optimization problem, and then propose an optimal and efficient solution named merged-gradient WFBP (MG-WFBP) to the problem.

4.1 Theoretical Analysis

It is obvious that the objective function of Eq. (24) can be rewritten by

\[ t = t_c^{(1)} + \max \{ \tau_c^{(2)} + t_c^{(2)}, \tau_b^{(1)} + t_b^{(1)} \} \]

\[ = T_{ar}(p^{(1)}) + \max \{ \tau_c^{(2)} + T_{ar}(p^{(2)}), \tau_b^{(1)} + t_b^{(1)} \} \]

\[ = T_{ar}(p^{(1)}) + \max \{ \max \{ \tau_c^{(3)} + T_{ar}(p^{(3)}), \tau_b^{(2)} + t_b^{(2)} \} + T_{ar}(p^{(2)}), \tau_b^{(1)} + t_b^{(1)} \} . \]  

(25)

It can be seen that the objective function consists of embedding max functions from the first layer to the last layer. We first analyze the difference of layer 2 be a normal layer or a merged-gradient layer, and then we extend it to a general layer \( \hat{t} \) to prove its optimality.

Assume that layers \( L, L - 1, ..., 3 \) are normal layers, and layer 2 is a merged-gradient layer, we have \( e^{(2)} = 0 \) and \( t_c^{(2)} = T_{ar}(p^{(2)} + p^{(1)}) \). We plug in these two new values to Eq. (25) to obtain

\[ \hat{t} = T_{ar}(p^{(2)} + p^{(1)}) + \max \{ \tau_c^{(2)} + \tau_b^{(1)} + t_b^{(1)} \} . \]  

(26)

Compare Eq. (25) to Eq. (26), we want to find out under what conditions \( \hat{t} < t \), i.e., layer 2 can be a gradient-merged layer. Specifically, we would like to derive the conditions such that

\[ \hat{t} = T_{ar}(p^{(2)} + p^{(1)}) + \max \{ \tau_c^{(2)} + \tau_b^{(1)} + t_b^{(1)} \} \]

\[ < t = T_{ar}(p^{(1)}) + \max \{ \tau_c^{(2)} + T_{ar}(p^{(2)}), \tau_b^{(1)} + t_b^{(1)} \} , \]  

(27)

which is equivalent to

\[ b \times p^{(2)} + \tau_b^{(1)} + t_b^{(1)} < \tau_b^{(1)} + t_b^{(1)} . \]  

(28)

Since there are two max functions in the above inequality, we need to decompose the max functions. Decomposing the two max functions explicitly corresponds to the four cases we discuss in the previous section. Note that it is impossible that \( \tau_c^{(2)} + T_{ar}(p^{(2)}) \leq \tau_b^{(1)} + t_b^{(1)} \) and \( \tau_c^{(2)} > \tau_b^{(1)} + t_b^{(1)} \) hold simultaneously. Therefore we decompose the two max functions with the following three conditions.

**Condition 1.** \( \tau_c^{(2)} + T_{ar}(p^{(2)}) \leq \tau_b^{(1)} + t_b^{(1)} \). Then \( \tau_c^{(2)} \leq \tau_b^{(1)} + t_b^{(1)} \) also holds. The inequality (28) becomes

\[ b \times p^{(2)} + \tau_b^{(1)} + t_b^{(1)} < \tau_b^{(1)} + t_b^{(1)} , \]  

which obviously does not hold as \( b \times p^{(2)} > 0 \). Therefore, layer 2 should be a normal layer in this case, since making layer 2 a merged-gradient layer cannot reduce the iteration time.

**Condition 2.** The condition is

\[ \tau_c^{(2)} + T_{ar}(p^{(2)}) > \tau_b^{(1)} + t_b^{(1)} . \]  

(29)

We can decompose inequality (28) to

\[ b \times p^{(2)} + \tau_b^{(1)} + t_b^{(1)} < \tau_b^{(1)} + t_b^{(1)} . \]  

(30)

which is equivalent to

\[ \tau_b^{(1)} + t_b^{(1)} < \tau_c^{(2)} + a . \]  

(31)

So if inequality (31) is true, then we can make layer 2 a merged-gradient layer to save the iteration time; otherwise we make it a normal layer.

**Condition 3.** The condition is

\[ \tau_c^{(2)} + T_{ar}(p^{(2)}) > \tau_b^{(1)} + t_b^{(1)} . \]  

(32)

We decompose inequality (28) to

\[ b \times p^{(2)} + \tau_c^{(2)} < \tau_c^{(2)} + T_{ar}(p^{(2)}) . \]  

(33)

It is equivalent to

\[ b \times p^{(2)} + \tau_c^{(2)} < \tau_c^{(2)} + a + b \times p^{(2)} , \]  

(34)

which is obviously true as \( a > 0 \). Therefore, under this condition, we prefer to make layer 2 a merged-gradient layer.

To summarize, under Condition 2 with inequality (31) and Condition 3, making layer 2 a merged-gradient layer can reduce the iteration time. Now we extend the above analysis to a general layer \( l \) and \( l > 1 \). When we just consider the end time of layer \( l - 1 \), making layer \( l \) be a merged-gradient layer if Condition 2 with inequality (31)
Lemma 1. Given an $L$-layer DNN which is trained with WFBP-SGD in a cluster of $N$ workers, if the gradient communication is done through all-reduce, layer $l > 1$ should be a merged-gradient layer to reduce the iteration time if and only if
\[
\tau_b(l-1) + t_b(l-1) < \tau_c(l) + a.
\]  
(35)

Proof. As we discussed in the above three conditions, if Condition 2 together with inequality \([31]\) or Condition 3 holds, layer $l$ should be a merged-gradient layer to reduce the iteration time, otherwise it should be a normal layer. The combination of Condition 2 together with inequality \([31]\) and Condition 3 is
\[
\tau_b(l-1) + t_b(l-1) < \tau_c(l) + a,
\]  
(36)
which concludes the proof.

From Lemma 1, it is seen that whether layer $l$ should be a merged-gradient layer or not depends on the end of computation time of layer $l-1$ (i.e., $\tau_b(l-1) + t_b(l-1)$) and its own beginning time of communication (i.e., $\tau_c(l)$). Thus, the communications of higher layers are not affected by the lower layers, while the lower layers are affected by the higher ones as the lower layer can only begin after the higher layers have finished. If layer $l$ is a normal layer, we can continue to determine layer $l-1$ by checking the above three conditions. If layer $l$ is a merged-gradient layer, layer $l-1$ has earlier end time according to the benefit of the merged-gradient layer. Again we also continue to determine the type of layer $l-1$ as the same way of layer $l$, which results in a recursive way from layer $L$ to layer 2. Consequently, we determine the last layer $L$ whether it can be a merged-gradient layer or a normal layer, and then determine layer $L-1$, and finally to layer 2 to find the final solution $m \in \mathbb{M}$ such that Eq. (35) is minimal.

Theorem 1. Given an $L$-layer DNN which is trained with WFBP-SGD in a cluster of $N$ workers, if the gradient communication is done through all-reduce, one can find $m \in \mathbb{M}$ such that the iteration time is minimal, and
\[
m = [e(L), e(L-1), ..., e(1)],
\]  
(37)
where
\[
e(l) = \begin{cases}  
    l_m & \text{if } \tau_b(l-1) + t_b(l-1) < \tau_c(l) + a \text{ and } l > 1 \\
    l_n & \text{otherwise}
\end{cases} 
\]  
(38)
for $1 \leq l \leq L$.

Proof. A layer $l$ is either a merged-gradient layer or a normal layer. According to Lemma 1 for $l > 1$ and $\tau_b(l-1) + t_b(l-1) < \tau_c(l) + a$, $e(l) = l_n$ has shorter time than $e(l) = l_m$. For $l = 1$ or $\tau_b(l-1) + t_b(l-1) \geq \tau_c(l) + a$, $e(l) = l_n$ has shorter time than $e(l) = l_m$. Consequently, if $m = [e(L), e(L-1), ..., e(1)]$ and $e(l)$ is assigned by Eq. (38), then changing the merged-gradient layers to normal layers or changing the normal layers to merged-gradient would bring longer iteration time, which conclude the proof.

4.2 Algorithms

Assume that the $N$-node cluster is connected by an interconnection with a bandwidth $B$, we can measure the all-reduce cost with respective to message size to derive the parameter $a$ and $b$ in Eq. (10). Therefore, we can estimate the communication time of all-reduce for any message size.

For the backward computation time, we can also benchmark for a particular GPU at the beginning of training. Thus, $t_f$, $t_f(1)$ and $t_f(0)$, where $1 \leq l \leq L$, are known. According to Theorem 1, we derive the algorithm to find $m$ as shown in Algorithm 1.

Algorithm 1 Find optimal $m \in \mathbb{M}$

Input: $a, b, L, t_b[1...L], p = [p(1), p(2), ..., p(L)]$.
Output: $m$

1: Initialize $t_c[1...L]$; // Communication time cost
2: Initialize $\tau_c[1...L]$; // Backward computation start time
3: Initialize $m[1...L] = \{l_n\}$; // Initialize all layers be normal layers
4: for $l = 1 \rightarrow L$ do
5:   $t_c[l] = a + b \times p[l];$
6:   $\tau_c[L] = 0;$
7:   for $l = L - 1 \rightarrow 1$ do
8:      $\tau_c[l] = \tau_c[l + 1] + t_b[l + 1];$
9:      $\tau_c[l] = \text{CALCULATECOMMSTART}(t_c, t_b, \tau_c, L);$  
10:     for $l = L \rightarrow 2$ do
11:        if $\tau_c[l - 1] + t_b[l - 1] - \tau_c[l] < a$ then // Eq. (38)
12:           \text{MERGE($\tau_c, t_c, p, l$)};
13:         $\tau_c[l] = \text{CALCULATECOMMSTART}(t_c, t_b, \tau_c, L);$  
14:        $m[l] = l_m;$ // Make $l$ be the merged-gradient layer
15:     Return $m;$
16:     procedure \text{MERGE($\tau_c, t_c, p, l$)}
17:      $t_c[l] = 0;$
18:      $p[l] = p[l - 1] + p[l];$
19:      $t_c[l] = a + b \times p[l];$
20: procedure \text{CALCULATECOMMSTART($t_c, t_b, \tau_c, L$)}
21: Initialize $\tau_c[1...L]$; // Communication start time
22: $\tau_c[L] = \tau_c[L] + t_b[L];$
23: for $l = L - 1 \rightarrow 1$ do
24:   $\tau_c[l] = \max\{\tau_c[l + 1] + t_c[l + 1], \tau_c[l] + t_b[l]\};$
25: Return $\tau_c;$

The algorithm first (line 1-8) initializes the layer-wise gradient communication cost $\tau_c(l)$, the computation start time $\tau_b(l)$ according to Eq. (10) and Eq. (6) respectively with system settings and benchmarks in the first several iterations. Then (line 9, line 20-25) the layer-wise start time of communication is calculated based on Eq. (7). After that (line 10-14), the merged-gradient layers are found according to Eq. (35), in which if there is a layer found as a merged-gradient layer, the communication time of its previous layer should be updated (line 16-19) according to Eq. (12), Eq. (13) and Eq. (14).

The proposed algorithm has a time complexity of $O(L^2)$. For a merged-gradient layer, the algorithm needs to re-calculate the start time of communication of each layer, which is an $O(L)$ search, and it has maximal $L - 1$ merged-gradient layers, so the time complexity of the algorithm is $O(L^2)$. Since the algorithm is a one-time calculation at the beginning of the training and it needs not to be re-calculated during the training process, the overhead of finding $m \in \mathbb{M}$ has no side-effect to the training performance.
Algorithm 2 MG-WFBP S-SGD at worker $g$

Input: $D = \{\{X_1, y_1\}, \ldots, \{X_n, y_n\}\}$, $I$, $net$, $N$, $bs$
Output: $W = [W^{(1)}, W^{(2)}, \ldots, W^{(L)}]$

1: Initialize a shared and synchronized queue $Q$;
2: Obtain the parameter size $p[1\ldots L]$ from $net$;
3: Allocate memories $W$;
4: Initialize $W$ in all accelerators;
5: if rank $== 0$ then
6: \hspace{1em} Benchmark several iterations to achieve $t_b[1\ldots L]$;
7: \hspace{1em} Get $m$ from Algorithm 1;
8: \hspace{1em} Bcast$(m, \text{root}=0)$; // Broadcast the optimal solution to all workers
9: \hspace{1em} ASYNCHANDLECOMMUNICATION$(Q, m)$;
10: for $i = 1 \rightarrow I$ do
11: \hspace{2em} Sample a mini-batch of data from $D$ to $d$;
12: \hspace{2em} ASYNCHANDLECOMMUNICATION$(Q, d, L)$;
13: \hspace{2em} WaitForLastCommunicationFinished();
14: \hspace{2em} $W = W - \eta \cdot \nabla W$,
15: \hspace{2em} NotifyFinished(); // Set isRunning to false
16: procedure ASYNCHANDLECOMMUNICATION$(Q, d, L)$
17: \hspace{2em} $o = d$;
18: \hspace{2em} for $l = 1 \rightarrow L$ do
19: \hspace{3em} $o = \text{FeedForward}(l, o)$;
20: \hspace{2em} for $l = L \rightarrow 1$ do
21: \hspace{3em} $\text{BackwardPropagation}(l)$;
22: \hspace{3em} $Q$.push($l$);
23: procedure ASYNCHANDLECOMMUNICATION$(Q, m)$
24: \hspace{2em} Initialize $lb$; // layerBuffer
25: \hspace{2em} while isRunning do
26: \hspace{3em} $l = Q$.pop();
27: \hspace{3em} $lb$.push($l$);
28: \hspace{3em} if $m[l] = l_0$ then
29: \hspace{4em} $\text{SynchronizedAllReduce}(lb)$;
30: \hspace{4em} $lb$.clear();
31: \hspace{4em} if $l = 1$ then
32: \hspace{5em} NotifyLastCommunicationFinished();

We denote the WFBP algorithm integrated with the optimal solution $m$ derived from Algorithm 1 as MG-WFBP. In MG-WFBP, the merged-gradient layers should be communicated with their previous layers. As a result, MG-WFBP achieves the minimal iteration time of S-SGD under known DNNs and system configurations. The algorithm of MG-WFBP S-SGD is shown in Algorithm 2. For each worker, the algorithm first (line 1-7) initializes related variables and calculates $m \in \mathbb{M}$ by using Algorithm 1. Then the root worker (rank 0) broadcasts (line 8) the solution $m$ to all other workers. Line 9 starts a communication thread, and the thread reads the layer number from the shared queue $Q$ and decides whether its gradients should be communicated (line 24-32). After that (line 10-14), it starts the loop of iteration, and iteratively (line 16-22) reads data to do feed forward operations and backward propagation followed by pushing the layer number into the shared queue. Finally, the algorithm notifies a message of isRunning=false to finish the training.

5 System Implementation

As shown in Algorithm 2 to implement MG-WFBP, our system is required to be equipped with three main features. First, the system needs to measure the backward propagation computation time of each layer (i.e., $t_b(l)$) for any configured deep neural networks. Second, the backward computation and gradient aggregation should be executed in parallel to pipeline communications and computations. Third, the merging operation of the merged-gradient layer should be efficient. It is non-trivial to implement the above three functions in current state-of-the-art deep learning frameworks (e.g., TensorFlow [30] and PyTorch [23]) which exploit a directed acyclic graph (DAG) to represent computing operations during training. Considering that PyTorch becomes more and more popular due to its easy-to-use Pythonic programming style and high performance operators [23], in this section we describe the implementation of MG-WFBP algorithm atop PyTorch.

5.1 Time Measurement of Backward Propagation

When deploying the DAG to GPUs in PyTorch, different operators could be executed concurrently due to the execution nature of CUDA streams [31]. Therefore, during the backward propagation, the gradients of different variables could be calculated concurrently on the same GPU such that the time measurement of each variable is not straightforward. To correctly collect the backward propagation time, we design a lightweight profiling tool for backward propagation in PyTorch with sequential execution of different variables. For each tensor that has gradients, we synchronize the tensor after it finishes invoking the gradient computation with CUDA synchronization (torch.cuda.synchronize). Consequently, we can collect the time interval of gradient computation between two nearby tensors, and the two nearby tensors with gradients should be from a single layer or from two nearby layers.

5.2 Parallelism between Gradient Computation and Aggregation

It is known that current deep learning frameworks provide Python APIs for end-users. In general, one can use multi-threading or multi-processing to make gradient computation and aggregation executed on different threads or processes. On one hand, however, there exists the GIL problem [32] in multi-threading of Python, which would result in very poor performance when paralleling two computation tasks. On the other hand, the multi-processing mechanism requires the memory copy between two processes as the gradients are updated every iteration. Since multiple processes cannot share the GPU memory address, when a process needs to copy its GPU data to another process, it needs to copy the data to host memory and then to another process, which causes performance degradation. To avoid the GIL problem in Python and memory copy between processes, we implement the gradient aggregation in a C++ daemon thread, and the original training codes are kept unchanged and the original training process (forward and backward computation) is running in the main thread. The C++ daemon thread well addresses the GIL problem in Python, and it can share the data of gradient with the main thread so that no memory copy is required. The architecture is shown in Fig. 3.
5.3 Efficient Gradient Merging

For every iteration, we need to copy two layers’ gradient data to a single segment of continuous memory if the current layer is a merge-gradient layer. We pre-allocate all memory for merged-gradient layers. For example, we assume layer 2 is a merge-gradient layer, which has \( p^{(2)} \) parameters, and layer 1 has \( p^{(1)} \) parameters. Note that layer 1 and layer 2 have different tensors so that the memory for these two tensors may not be continuous. Then we allocate a buffer whose size is \( (p^{(2)} + p^{(1)}) \times \text{BytesPerElement} \), where \( \text{BytesPerElement} \) is 4 for single precision floats and 2 for half precision floats (e.g., Mixed precision training). Therefore, for every merged-gradient layers and their preceded normal layers, there exist pre-allocated buffers. When any buffer is full, the gradient aggregation thread invokes the all-reduce operation. In PyTorch, the data copy between GPU tensors is fast as it just needs to copy data in GPU memory without copying back to host memory. For example, Nvidia Tesla V100 GPU delivers a peak memory bandwidth of 900GB/s.

6 Experimental Studies

6.1 Experimental Settings

We conduct extensive experimental studies to show the effectiveness of MG-WFBP. Our test-beds contain three GPU clusters with 10Gbps Ethernet (10GbE) and 56Gbps InfiniBand (56GbIB) interconnections. One is an 8-node Nvidia Tesla K80 cluster which has a total of 16 K210 GPUs (one Tesla K80 card contains two K210 GPUs), and the 8 nodes are connected by 10GbE; the other two are 4-node Nvidia Tesla V100 clusters, in which each node contains 4 GPUs, resulting in a total of 16 GPUs, and the 4 nodes are connected with 10GbE and 56GbIB. The cluster settings are listed in Table 3.

6.2 Measurement of All-reduce Communication

To verify the communication model in Eq. (10) empirically, we first present some foregone results of the time of the all-reduce operation in the three configured clusters. The measured time of all-reduce under cluster 1, cluster 2 and cluster 3 are shown in Fig. (a), (b) and (c) respectively. Take the size of parameters \( 4p \) in single precision floating points as the variable, we can see that the startup overheads (e.g., \( 2(N-1) \times 10 \) in the ring-based all-reduce algorithm) are \( 972\mu s, 908\mu s \) and \( 236\mu s \) on cluster 1, cluster 2 and cluster 3 respectively.

We also show the statistical distributions of the layer-wise tensor size in different DNNs in Fig. (5) which shows that a large proportion of tensors are with small number of gradients. For example, ResNet-152 has 150 tensors whose size is 1024 bytes (in 32-bit precision), and DenseNet-161 has 160 tensors whose size is 768 bytes (in 32-bit precision).

6.3 Real-world Experiments

We implement WFBP [15][17], single-layer communication Sync EASGD (SyncEASGD) [21] and our proposed MG-WFBP with PyTorch and OpenMPI, and test the performance across two 16-GPU (K80 and V100) clusters with 10GbE and 56GbIB. We also compare the scaling efficiencies with TensorFlow. The compared TensorFlow version is at v1.3, and it uses parameter servers to do S-SGD using the official benchmark script⁶. We also run 13 epochs to verify the convergence of the model training, in which 50,000 images are used to test the top-1 accuracy.

6.3.1 Results on Cluster 1

The experimental results of GoogleNet and ResNet-50 on the G210 GPU cluster are shown in Fig. (6) and Fig. (7) respectively. The non-overlapped communication cost compared to the computation time is shown in Fig. (8). The baseline is the iteration throughput of two GPUs in a single

Note: # MACs indicates the number of matrix multiplication and accumulation in the forward calculation with a batch size of 1.
machine, in which no communication is required. And the speedup of throughput on multiple workers are compared to the baseline. From Fig. 5, we can observe that for both GoogleNet and ResNet-50, MG-WFBP performs better than WFBP, SyncEASGD and TensorFlow. SyncEASGD does not overlap the communication with computation; and hence the communication cost increases when the number of workers increases. As a consequence, the scaling efficiency of SyncEASGD is poor. WFBP achieves near linear scaling on 2 and 4 nodes, in which the non-overlapped communication overhead are small. When scaling to 8 nodes, however, WFBP has an obvious drop in efficiency due to the increased startup time of layer-wise communication which cannot be totally hidden by computation. Regarding the performance of TensorFlow, it uses parameter servers to do the model

Fig. 4: The communication time of all-reduce along with the size of parameters on three different clusters. (a) Cluster 1 with \( a = 9.72 \times 10^{-4}, b = 1.97 \times 10^{-3}; \) (b) Cluster 2 with \( a = 9.08 \times 10^{-4}, b = 7.4 \times 10^{-10}; \) (c) Cluster 3 with \( a = 2.36 \times 10^{-4}, b = 4.06 \times 10^{-10}. \)

Fig. 5: Tensor size distribution. The two curves are the all-reduce communication time with respect to tensor size, and the scatter markers indicate the number of tensors that have a specific size in a DNN.

Fig. 6: The performance of GoogleNet on the K80 cluster with 10GbE. The baseline of the speedup of SGD is on a single machine with 2 GPUs.

Fig. 7: The performance of ResNet-50 on the K80 cluster with 10GbE. The baseline of the speedup of SGD is on a single machine with 2 GPUs.
aggregation. On one hand, the centralized parameter server based algorithm could easily suffer a bandwidth pressure in the parameter server on the lower speed network [17]. On the other hand, it takes two communication directions (workers to PS, and PS to workers) to finish the model synchronization, which introduces more overhead in the synchronization pass. Therefore, though TensorFlow exploits the WFBP technique, the PS-based method performs worse than the decentralized method. Our proposed algorithm has a very small non-overlapped communication cost even on the 8-node cluster, so the scaling efficiency is still close to linear. In summary, MG-WFBP achieves about 1.2x and 1.36x speedups compared to WFBP and SyncEASGD respectively on the 8-node (16 GPUs) K80 cluster on both GoogleNet and ResNet-50.

### 6.3.2 Results on Cluster 2 and Cluster 3

Note that MG-WFBP has no side-effect on the convergence performance (in terms of the number of iterations) as MG-WFBP can achieve consistent results of the aggregated gradients with the original S-SGD at each iteration. Therefore, in the following performance evaluation, we focus on the comparison on the average iteration wall-clock time to demonstrate how much performance improvement of our MG-WFBP over WFBP and SyncEASGD.

On cluster 2 and cluster 3, in addition to the general setting with single precision (FP32) training, we also apply our MG-WFBP algorithm to the mixed precision training technique [37], which is widely used on the GPUs with Tensor Cores (e.g., Tesla V100) to increase the computing efficiency and reduce the communication traffic. The results are shown in Fig. 9. In overall, it can be seen that for different DNN models, no one always outperforms the other one between WFBP and SyncEASGD algorithms as the both algorithms are sensitive to the cluster configurations, while our proposed MG-WFBP algorithm achieves the fastest training speed in all evaluated DNNs. The first row of Fig. 9 shows that MG-WFBP achieves up to 70% improvement over WFBP and SyncEASGD algorithms on Cluster 2 with 10GbE connection. The second row of Fig. 9 demonstrates that MG-WFBP outperforms WFBP and SyncEASGD up to 26% on Cluster 3 with 56GbIB connection.

On the ResNet-152 architecture, pipelining all FP32 tensors brings some benefits to hide some communication overheads so that WFBP trains faster than SyncEASGD. On both DenseNet and Inception architectures, however, pipelining for every tensors between communication and computation introduces many extra communication overheads so that WFBP performs slower training speed than SyncEASGD. On the ResNet-152 architecture with FP32 precision, the hidden communication time is longer than the extra time introduced by each layer’s startup overhead with pipelining so that WFBP is about 10% faster than SyncEASGD. Our MG-WFBP algorithm can further reduce the negative impact of the startup time by smartly merging some gradients, which results in extra 10% improvement. On the other hand, pipelining all tensors introduces larger overheads than hidden time. For example, SyncEASGD is 20% faster than WFBP in DenseNet-161. By merging the tensors smartly, MG-WFBP performs 7% faster than SyncEASGD.

In summary, MG-WFBP can always outperform WFBP and SyncEASGD. In the conducted extensive experiments, MG-WFBP generally achieves up to 15% improvement over the best of WFBP and SyncEASGD in both 10GbE and 56GbIB interconnections.

### 6.4 Simulation

Due to the hardware limitation, we do not have a very large GPU cluster to support more large-scale experiments. So we conduct simulations based on the real single-GPU performance and the network performance model. Based on the measured layer-wise backward propagation time on the real K80 GPU, we simulate WFBP, SyncEASGD and MG-WFBP by scaling from 4 workers to 64 workers.

#### Overall Performance

We simulate to train GoogleNet and ResNet-50 by scaling from 4 workers to 64 workers. The scaling performance is shown in Fig. 10. On the cluster with K80 GPUs, our proposed algorithm MG-WFBP achieves the best speedup. On the 64-node cluster, MG-WFBP outperforms WFBP and SyncEASGD by 1.78x and 1.35x, respectively on GoogleNet. On ResNet-50, MG-WFBP performs almost linear speedup, while WFBP and SyncEASGD only have around 55% scaling efficiency in the 64-node cluster. It is important to notice that the lines of WFBP and SyncEASGD have a crossing point in Fig. 10. This is because the two algorithms are sub-optimal in utilizing the network bandwidth; when the computation has the opportunity to overlap with communication, and the startup time of network communication is not that large (e.g., 4-16 workers in the K80 cluster), then WFBP would have the advantage to hide the communication compared to SyncEASGD. But when scaling to large number of workers (e.g., 64 workers), the startup time of communication becomes much larger so that it is hard to be hidden, then using a single-layer communication could become a better approach. As we can see, SyncEASGD achieves better scaling efficiency than WFBP in the 64-node cluster on both tested CNNs. MG-WFBP not only overlaps the communication with computation, but also finds the optimal communication message size. So it achieves better scaling efficiency than SyncEASGD and WFBP. Finally, on training ResNet-50, MG-WFBP achieves about 1.75x and 1.45x speedups compared to WFBP and SyncEASGD respectively on the simulated 64-node K80 cluster.

### 7 Related Work

The wait-free backward propagation (WFBP) algorithm has recently been proposed to reduce such impact by overlapping communication with computation [15][17]. In WFBP, the backward computation operations can be started without waiting for the completion of the previous round of data communication. If the communication cost of layer \( l + 1 \) is smaller than the cost of gradients computation of layer \( l \), then the communication cost can be completely hidden (except the first layer); and as a result, the scaling efficiency can be close to linear [15][17]. In practice, however, many DNN models are trained on high-throughput GPUs that result in very short computing time for each backward layer, while it needs to wait for gradient aggregation before starting the
(a) ResNet-152 with 10GbE
(b) DenseNet-161 with 10GbE
(c) DenseNet-201 with 10GbE
(d) Inception-v4 with 10GbE
(e) ResNet-152 with 56GbIB
(f) DenseNet-161 with 56GbIB
(g) DenseNet-201 with 56GbIB
(h) Inception-v4 with 56GbIB

Fig. 9: Time comparison of non-overlapped communication and computation on the two V100 GPU clusters (10GbE and 56GbIB). ‘WF.’, ‘S.E.’ and ‘M.W.’ indicate WFBP, SyncEASGD and MG-WFBP algorithms respectively. ‘Comp.’ refers to the computation cost (i.e., $t_c$), and ‘Comm.’ refers to the non-overlapped communication cost (i.e., $t_{no}$). The values in the brackets are the range of improvements of MG-WFBP over WFBP and SyncEASGD.

(a) GoogleNet
(b) ResNet-50

Fig. 10: The performance comparison on the simulated K80 cluster connected with 10GbE. Baseline of the speedup of SGD is on a single K80.

Zhang et al. [17] proposed the Poseidon system with hybrid communication of PS and SFB combined with the WFBP algorithm, and they have achieved 15.5x speedup on 16 single-GPU (TITANX Pascal) machines. Unfortunately, due to drawbacks of PS and SFB and the communication scheme, Poseidon could also be far away from linear scaling with the number of workers increased due to the communication bottleneck.

In the HPC community, the MPI data communication collectives have been redesigned for distributed training to improve the communication performance across multiple machines [15]. Many MPI-like implementations, such as OpenMPI, NCCL, Gloo7 and MVAPICH2-GDR8 support efficient CUDA-aware communication between GPUs via network, and many state-of-the-art deep learning frameworks (e.g., TensorFlow, PyTorch, Caffe2 and CNTK9) integrate NCCL or Gloo for their distributed training modules. Even though these libraries provide very efficient communication collectives, the data communication would still become bottleneck when the communication-to-computation ratio is high, and S-SGD does not scale very well.

8 Conclusion

In this work, we first showed that existing state-of-the-art communication strategies, say wait-free backward propagation (WFBP) and single-layer communication (SyncEASGD), are sub-optimal in the synchronized distributed deep learning training when the communication-to-computation ratio is high. Then we generalized the communication problem in pipelining communication and computation as an

7https://github.com/facebookincubator/gloo
8https://mvapich.cse.ohio-state.edu/
9https://docs.microsoft.com/en-us/cognitive-toolkit/
optimization problem and developed an optimal solution with an efficient algorithm. We then proposed the merged-gradient wait-free backward propagation (MG-WFBP) strategy by optimally merging gradients. We implemented MG-WFBP atop the popular deep learning framework PyTorch. Our implementation is also publicly available. Through extensive experiments on three 16-GPU clusters including Nvidia Tesla K80 GPUs with 10Gbps Ethernet connection and Nvidia Tesla V100 GPUs with both 10Gbps Ethernet and 56Gbps InfiniBand, we verified that MG-WFBP can achieve much better scalability than WFBP and SyncEASGD on various popular convolutional neural networks. Simulations were also studied to further explore the advantage of MG-WFBP on large-scale clusters.

ACKNOWLEDGEMENTS

The research was supported in part by Hong Kong RGC GRF grants under the contracts HKBU 12200418, HKUST 16206417 and 16207818, as well as an RGC CRF grant under the contract C7036-15G. We would also like to thank NVIDIA for providing the GPU clusters for experiments.

REFERENCES

[1] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le et al., “Large scale distributed deep networks,” in Advances in neural information processing systems, 2012, pp. 1223–1231.

[2] P. Goyal, P. Dollär, 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] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch, “Ako: Decentralised deep learning with partial gradient exchange,” in Proceedings of the Seventh ACM Symposium on Cloud Computing. ACM, 2016, pp. 84–97.

[4] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, “Geeps: Scalable deep learning on distributed GPUs with a GPU-specialised parameter server,” in Proceedings of the Eleventh Euro-Par Conference on Parallel and Distributed Systems (ICPADS), 2018 IEEE 24rd International Conference. IEEE, 2018, pp. 425–432.

[5] 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-IEEE Conference on Computer Communications. IEEE, 2019, pp. 1729–1737.

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

[7] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally, “Deep gradient compression: Reducing the communication bandwidth for distributed training,” in International Conference on Learning Representations, 2018.

[8] 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 Advances in Neural Information Processing Systems, 2017, pp. 1508–1518.

[9] 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 39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, 2019, pp. 2238–2247.

[10] S. Shi, K. Zhao, Q. Wang, Z. Tang, and X. Chu, “A convergence analysis of distributed SGD with communication-efficient gradient sparsification,” in the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, 2019, pp. 3411–3417.

[11] S. Potluri, K. Hamidouche, A. Venkatesh, D. Burreddy, and D. K. Panda, “Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with Nvidia GPUs,” in Parallel Processing (ICPP), 2013 42nd International Conference on. IEEE, 2013, pp. 80–89.

[12] C. Chen, W. Wang, and B. Li, “Round-robin synchronization: Mitigating communication bottlenecks in parameter servers,” in IN- FOCOM 2019-IEEE Conference on Computer Communications. IEEE, 2019, pp. 532–540.

[13] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. Panda, “Scalable reduction collectives with data partitioning-based multi-leader design,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2017, p. 64.

[14] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, “S-Caffe: Co-designing MPI routines and Caffe for scalable deep learning on modern GPU clusters,” in Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2017, pp. 199–209.

[15] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[16] 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 Proceedings of the 2017 USENIX Annual Technical Conference. USENIX Association, 2017, pp. 181–193.

[17] S. Shi, Q. Wang, and X. Chu, “Performance modeling and evaluation of distributed deep learning frameworks on GPUs,” in 2018 IEEE DataCom. IEEE, 2018, pp. 949–957.

[18] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, “Re-architecting datacenter networks and stacks for low latency and high performance,” in Proceedings of the Conference of the ACM Special Interest Group on Data Communication. ACM, 2017, pp. 29–42.

[19] S. Shi, X. Chu, and B. Li, “MG-WFBP: Efficient data communica- tion of distributed deep learning frameworks on GPUs,” in 2018 INFOCOM. IEEE, 2018, pp. 99–104.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An imperative style, high-performance deep learning library,” in Advances in Neural Information Processing Systems 32, 2019, pp. 8024–8035.

[21] Y. You, A. Buluç, and J. Demmel, “Scaling deep learning on GPU and Knights Landing clusters,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2017, p. 9.

[22] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” arXiv preprint arXiv:1408.5093, 2014.

[23] T. Hoefler, W. Gropp, R. Thakur, and J. L. Träff, “Toward per- formance modeling and evaluation of distributed deep learning software tools,” in Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2017, pp. 64.

[24] S. Ghemawat, G. Irving, M. Isard, R. Kennedy, T. Chandra, T. Franklin, J. Geels, M. Gruber, S. Hsieh, S. Keckler, D. Kohler, M. F. King, D. W. Leacock, D. A. Patterson, J. Peng, A. Steinhardt, and M. Zorn, “Google file system,” in USENIX Conference on File and Storage Technologies, 2004, pp. 5–16.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in Proceedings of the 13th Symposium on Operating Systems Design and Implementation, 2003, pp. 29–43.

[26] S. Ghemawat, H. Gobioff, S.-T. Leung, and M. Seltzer, “High performance networked storage,” in Proceedings of the 2006 USENIX Annual Technical Conference. USENIX Association, 2006, pp. 11–22.

[27] S. Shi, Q. Wang, and X. Chu, “Benchmarking state-of-the-art deep learning software tools,” in Proceedings of the 2016 7th International Conference on Cloud Computing and Big Data (CCBD). IEEE, 2016, pp. 99–104.

[28] S. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, “Re-architecting datacenter networks and stacks for low latency and high performance,” in Proceedings of the Conference of the ACM Special Interest Group on Data Communication. ACM, 2017, pp. 29–42.
Shaohuai Shi received the B.E. degree in software engineering from South China University of Technology, P.R. China, in 2010, and the Master degree in computer science from Harbin Institute of Technology, P.R. China in 2013. He is currently pursuing his Ph.D. degree with Hong Kong Baptist University. His research interests include GPU computing and machine learning systems. He is a student member of the IEEE.

Xiaowen Chu received the B.E. degree in computer science from Tsinghua University, P.R. China, in 1999, and the Ph.D. degree in computer science from the Hong Kong University of Science and Technology in 2003. Currently, he is currently a full professor in the Department of Computer Science, Hong Kong Baptist University. His research interests include parallel and distributed computing, cloud computing and wireless networks. He is serving as an Associate Editor of IEEE Access and IEEE Internet of Things Journal. He is a senior member of the IEEE.

Bo Li is a professor in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. He holds the Cheung Kong chair professor in Shanghai Jiao Tong University. Prior to that, he was with IBM Networking System Division, Research Triangle Park, North Carolina. He was an adjunct researcher with Microsoft Research Asia-MSRA and was a visiting scientist in Microsoft Advanced Technology Center (ATC). He has been a technical advisor for China Cache Corp. (NASDAQ CCIH) since 2007. He is an adjunct professor with the Huazhong University of Science and Technology, Wuhan, China. His research interests include: large-scale content distribution in the Internet, Peer-to-Peer media streaming, the Internet topology, cloud computing, green computing and communications. He is a fellow of the IEEE for contribution to content distributions via the Internet. He received the Young Investigator Award from the National Natural Science Foundation of China (NSFC) in 2004. He served as a Distinguished lecturer of the IEEE Communications Society (2006-2007). He was a corecipient for three Best Paper Awards from IEEE, and the Best System Track Paper in ACM Multimedia (2009).