Journal of Parallel and Distributed Computing 159 (2022) 10–23

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Improving scalability of parallel CNN training by adaptively adjusting parameter update frequency

Sunwoo Lee *, Qiao Kang, Reda Al-Bahrani, Ankit Agrawal, Alok Choudhary, Wei-keng Liao

Northwestern University, 2145 Sheridan Rd, Evanston, IL 60208, USA

ARTICLE INFO

Article history:
Received 6 April 2020
Received in revised form 19 August 2021
Accepted 20 September 2021
Available online 29 September 2021

Keywords:
Deep learning
Data parallelism
Communication cost
Parameter update frequency

ABSTRACT

Synchronous SGD with data parallelism, the most popular parallelization strategy for CNN training, suffers from the expensive communication cost of averaging gradients among all workers. The iterative parameter updates of SGD cause frequent communications and it becomes the performance bottleneck. In this paper, we propose a lazy parameter update algorithm that adaptively adjusts the parameter update frequency to address the expensive communication cost issue. Our algorithm accumulates the gradients if the difference of the accumulated gradients and the latest gradients is sufficiently small. The less frequent parameter updates reduce the per-iteration communication cost while maintaining the model accuracy. Our experimental results demonstrate that the lazy update method remarkably improves the scalability while maintaining the model accuracy. For ResNet50 training on ImageNet, the proposed algorithm achieves a significantly higher speedup (739.6 on 2048 Cori KNL nodes) as compared to the vanilla synchronous SGD (276.6) while the model accuracy is almost not affected (<0.2% difference).

1. Introduction

Recently, Convolutional Neural Network (CNN) has become one of the most popular machine learning techniques. CNNs have achieved great successes in a variety of applications such as image classification [14,32], image regression [25,35], object detection [34,45], natural language processing [24,43], and scientific applications [10,19,28]. However, training a deep CNN is a computationally intensive task that can take hours or even days. For instance, training ResNet50 [14] on ImageNet [9] using a single GPU (NVIDIA M40) takes about two weeks [42]. For large scale deep learning applications, efficient parallelization is crucial to finish the training in a reasonable amount of time.

Synchronous Stochastic Gradient Descent (SGD) [17] with data parallelism is the most popular parallel neural network training strategy. The algorithm evenly distributes each batch of training samples (called mini-batch) to all workers and processes them independently. Then, the locally computed gradients are averaged among all the workers using inter-process communications. In data parallel training, the local gradient size is fixed to the model size regardless of the number of processes. As scaling up, thus, the computational workload per process proportionally decreases while the communication cost increases. Many previous works showed such an increasing communication time in their experiments [21,22,38]. Especially, when accelerators are employed, such as GPU, Intel Xeon Phi, or TPU, the communication to computation ratio becomes higher at each iteration. Given a fixed network bandwidth, the faster the computation for each mini-batch, the higher portion of the iteration time the communication time takes up.

Many researchers have studied how to overlap the communications with the computations to improve the scalability [13,22,37,38]. Since the gradients do not have data dependency across layers, the communication for averaging the gradients at a layer and the gradient computations at other layers can be performed simultaneously. However, the next mini-batch can be processed only after the parameters are updated using the gradients computed from the current mini-batch. Thus, all the communications posted at each iteration should be finished before starting the next iteration. If the communication time is longer than the computation time within each iteration, a part of the communication time will be exposed making all the processes blocked until the communications are finished. Therefore, in order to achieve a good speedup of the parallel training, the per-iteration communication time should be reduced so that the exposed communication time is minimized keeping all the processes busy.

* Corresponding author.
E-mail address: slz839@ece.northwestern.edu (S. Lee).

https://doi.org/10.1016/j.jpdc.2021.09.005
0743-7315/

In this paper, we discuss how to address the expensive per-iteration communication cost in parallel training by delaying the parameter update at a part of model parameters. Since the gradients are computed from the output side layers in the back-propagation phase, only the communications at a few input side layers are likely exposed causing a blocking time. We propose to adjust the parameter update frequency such that the parameters at those input side layers are less frequently updated than the other layers. Instead of updating the parameters at every iteration, our lazy update method accumulates the gradients if the difference of the direction between the accumulated gradients and the latest gradients is bounded by a sufficiently small angle on the inner product space. The parameter update interval is adaptively adjusted using a constant back-off algorithm at run-time. This adaptive update frequency control enables to find the maximum update interval which allows the model to keep moving towards minimizing the cost function. The gradient compression method proposed in [26] similarly accumulates the gradients, however, the compression method only considers the magnitude of the gradients, based on a heuristics: "larger gradients are more important than the others". Our work explores the performance impact of the gradient accumulation in a layer-wise manner considering not only the magnitude but also, more importantly, the direction on the parameter space.

Our training strategy has several advantages: First, the inter-process communications are less frequently performed at the layers to which our lazy update method is applied, and thus a better scaling performance can be achieved. Second, the proposed lazy update method automatically adjusts the parameter update interval during training without requiring users to tune any extra hyper-parameters. Finally, the proposed algorithm determines the lazy update interval based on the observed degree of noise in the gradients at run-time so that it can be applied to parallel training independently of optimizers or hyper-parameter settings such as mini-batch size.

To evaluate the performance of the proposed training algorithm, we conduct image classification and regression experiments and analyze the performance results on KNL nodes of Cori supercomputer at NERSC. We verify the effectiveness of the proposed training method across several network architectures (Wide-ResNet20, ResNet50, and EDSR), datasets (CIFAR10, ImageNet, and DIV2K), optimizers (mini-batch SGD and Adam). Our experimental results demonstrate that, by having a different parameter update frequency across layers, the communication time can be significantly reduced while maintaining the model accuracy. For ImageNet classification with ResNet50, our method improves the speedup of the parallel training from 50.96 to 115.11 using up to 256 compute nodes with a negligible effect on the validation accuracy ($< 0.1\%$ difference).

We also study the impact of the proposed training algorithm on the large batch training performance. The large batch size improves the degree of parallelism by having more training samples per mini-batch. However, the batch size does not affect the per-iteration communication cost. Regardless of the batch size, as a nature of strong scaling, the ratio of computation to communication is supposed to be reduced as more processes are employed. We empirically verify that our proposed method can be applied to large-batch training by presenting and analyzing the large-batch training performance of ResNet50. For ResNet50 training with a batch size of 8192, our proposed method improves the speedup from 276.59 to 739.56 using up to 2048 nodes, while achieving almost the same accuracy to the reported accuracy ($< 0.2\%$ difference) in [11].

2. Background and related works

2.1. Mini-batch SGD-based CNN training

In this paper, we consider minimization problems of the form

$$\min_{w \in \mathbb{R}^d} F(w) = \frac{1}{n} \sum_{i=1}^{n} f(w, x_i), \tag{1}$$

where $w \in \mathbb{R}^d$ is the parameter vector of the given network, x is the training dataset of size n, and $f(w, x_i)$ is a cost function. In this paper, we will refer to mini-batch SGD by SGD for short. SGD computes a stochastic gradient of the cost function f with respect to the model parameters w from a random subset of training samples $\mathcal{B}$ using Equation (2).

$$\nabla f_{\mathcal{B}}(w) = \frac{1}{m} \sum_{i \in \mathcal{B}} \nabla f(w, x_i), \tag{2}$$

where m is the mini-batch size. Then, the parameters are updated by Equation (3).

$$w_i = w_{i-1} - \mu \nabla f_{\mathcal{B}}(w_i), \tag{3}$$

where μ is the learning rate. The algorithm repeats these two steps until the cost function f is minimized.

2.2. Parallelization strategies

Synchronous SGD with data parallelism is the most popular parallel CNN training algorithm used in many applications. The algorithm evenly distributes each mini-batch to all workers and each worker locally processes the assigned training samples. Then, the local gradients are averaged among all the workers using inter-process communications. Many existing works use allreduce operation for aggregating and summing up the local gradients.

In parallel training, the communications at each layer can overlap with the computations at other layers. Since the gradients do not have data dependency across layers, the communication at one layer and the gradient computation at other layers can be performed simultaneously. Once the gradients are computed at each layer in the backpropagation phase, a non-blocking communication is posted so that the communication overlaps with the later backward computations. Many existing works exploit the overlap of the communications with the computations in parallel training [13,19,22,28,37,38].

Asynchronous SGD (ASGD) has been proposed in [8] which tackles the scalability issue of mini-batch SGD. ASGD allows multiple mini-batches to be concurrently processed at the cost of having a certain degree of gradient asynchrony. However, the number of asynchronous workers should be sufficiently small to guarantee the convergence, which is impractical in large scale applications. Sparse Aggregation SGD is proposed in [7], which sparsely averages the parameters among workers after multiple local updates. The algorithm effectively lowers the inter-process communication frequency and a better scalability can be expected, however the convergence accuracy is still largely affected by the number of workers.

Instead of designing a new optimization algorithm, some researchers proposed to sparsify or quantize the gradients to reduce the communication cost [1,2,29,33,39,40]. Similarly, some researchers proposed gradient compression techniques [6,12,26]. However, the gradient sparsification methods proposed in [1,33] have a user-tunable threshold which is used to drop out small gradients. In practice, tuning an additional hyper-parameter is not a negligible extra work for large-scale deep learning applications.

Algorithm 1 Mini-batch SGD with Adaptive Lazy Parameter Update (b: the number of layers to which the lazy update method is applied. k: the number of iterations for accumulating the gradients.)

```
1:  w ← w_0, g ← 0, k ← 1
2:  while stop condition is not met do
3:      for i ← 1 ··· n/m do
4:          B ← i^th mini-batch of size m.
5:          ∇f_B(w_i) ← Compute_Gradient(f, B, w_i).
6:          for j ← 1 ··· l do
7:              if j > b then
8:                  Update w_{i,j} with ∇f_B(w_{i,j}).          ▷ Eq. (3)
9:              else
10:                 Accumulate ∇f_B(w_{i,j}) to g_j.            ▷ Eq. (4)
11:                 if (i mod k) is 0 then
12:                     Update w_{i,j} with g_j.                ▷ Eq. (5)
13:         if (i mod k) is 0 then
14:             k = Calculate_Interval(g, ∇f_B(w_i), b).
15:             g ← 0.
```

In addition, these works either do not consider the extra computational cost or present a limited performance improvement due to the expensive extra computations. For instance, the gradient quantization method proposed in [2] shows limited speedups ($1.1 \sim 2.1$) while the data precision is reduced from 32-bit to 4-bit. The weight sharing technique in [12] performs K-Means data clustering algorithm off-line. The gradient compression technique in [26] consists of a set of processes such as quantization, encoding, and clipping. As pointed out in [38], these gradient compression techniques also need to balance the trade-off between the accuracy and the communication cost.

Recently, a few large batch training techniques have been proposed. You et al. proposed Layer-wise Adaptive Rate Scaling (LARS) [41]. Goyal et al. and Hoffer et al. introduced linear scaling rule [11] and root scaling rule [16] of mini-batch size and learning rate, respectively. All these large-batch training techniques have showed that the batch size can be increased to a certain problem-dependent threshold without a significant loss in accuracy. Larger batch size implies fewer parameter updates within each epoch, and thus fewer communications for averaging gradients among all workers. However, the batch size does not affect the communication cost for averaging gradients at each iteration. As the number of processes increases, the communication time increases while the per-process computation workload is proportionally reduced. So, regardless of the batch size, the communication time ends up being dominant over the computation time and the speedup is saturated. In this paper, we focus on how to improve the scaling efficiency by reducing the communication cost for processing each mini-batch, regardless of the mini-batch size.

3. CNN training with lazy parameter update

In this section, we propose a mini-batch SGD-based CNN training algorithm which adaptively adjusts the parameter update frequency. We begin with describing the lazy parameter update method.

$$g_i = g_{i-1} + \nabla f_{\mathcal{B}}(w_i) \tag{4}$$

$$w_{i+k} = w_i - \mu g_{i+k} \tag{5}$$

where $\nabla f_{\mathcal{B}}(w_i)$ is the stochastic gradient of a cost function f with respect to w_i, the parameters at iteration i, g_{i+k} is the accumulated gradients from iteration i to $i+k$, and k is the lazy update interval. At every iteration, the gradients computed from a mini-batch are accumulated using Equation (4). After k iterations, the parameters are updated with the accumulated gradients using Equation (5).

Algorithm 1 shows a mini-batch SGD-based CNN training algorithm with the described lazy update method. The algorithm repeatedly traverses over n training samples until the stop condition is satisfied at line 2. Given the batch size m, the algorithm processes $\frac{n}{m}$ mini-batches sequentially at line 3. For each mini-batch, the gradients of the cost function f are computed with respect to the model parameters at line 5. Note that we refer the input side of network as 'bottom' and the output side as 'top'. The parameters at the top $(l-b)$ layers are updated every iteration using Equation (3) at line 8, where l is the number of layers in the model. The gradients at the bottom b layers are accumulated using Equation (4) at line 10. The lazy update interval k is initially set to 1 and re-calculated after every lazy update at line 14.

When $1 \leq b < l$, Algorithm 1 partitions the model into two parts and the parameters of them are updated at different frequencies. The backpropagation algorithm can be considered as a dynamic programming of multi-layer perceptrons. The parameters of each layer are adjusted to minimize its own error. Since the errors are back-propagated through all the layers, the principle of optimality holds in the backward direction and the backpropagation algorithm effectively minimizes the cost function that is attached at the end of the model. In Algorithm 1, the consecutive top side layers, either $(l-b)$ or l, are updated at every iteration. Thus, it is guaranteed that the parameters of each layer have been updated more than any of its bottom layers. This condition still holds the principle of optimality in dynamic programming, and Algorithm 1 can effectively minimize the cost function when $1 \leq b < l$.

The lazy update interval k plays a key role in reducing the communication cost in parallel training as well as achieving a high accuracy. The longer the lazy update interval, the more sparser the communications for averaging gradients, which result in a better scalability. On the other hand, the accumulated gradients g in Equation (5) becomes noisier as more gradients are accumulated. Note that this is because we keep updating the top $l-b$ layers every iteration. Assuming m training samples for each mini-batch are randomly extracted from the dataset, the k sets of gradients that are accumulated by Equation (4) can be considered as independent random variables. Each of them is approximately normally distributed with mean

$$\mathbf{E}[\nabla f(w)] = \frac{1}{n}\sum_{i=1}^{n} \nabla f(w, x_i) = \nabla F(w), \tag{6}$$

where $\nabla F(w)$ is the optimal gradient with respect to the parameters w, x_i is a training sample, and n is the number of training samples. Thus, the expected accumulated gradient at iteration $(i+k)$ is $\mathbf{E}[g_{i+k}] = \sum_{j=1}^{k} \nabla F(w_{i+j})$. The expected optimal gradient at iteration $(i+k)$ is $\mathbf{E}[\nabla f_{\mathcal{B}}(w_{i+k})] = \nabla F(w_{i+k})$. We can consider the accumulated gradients g_{i+k} as noisy when the difference between $\mathbf{E}[\nabla f_{\mathcal{B}}(w_{i+k})]$ and $\mathbf{E}[g_{i+k}]$ is large. Intuitively, the higher the degree of noise, the slower the convergence rate.

3.1. Bounding gradient noise with respect to direction

To keep the difference between $\mathbf{E}[\nabla f_{\mathcal{B}}(w_{i+k})]$ and $\mathbf{E}[g_{i+k}]$ from becoming too large, Algorithm 1 controls the lazy update interval at line 14 such that the gradients are accumulated only until g is still in a descent direction with respect to the latest parameters.

Proposition 1. *A sufficient condition for the accumulated gradient g_{i+k} to be in a descent direction with respect to the parameters at iteration $(i+k)$ is as below.*

$$\|g_{i+k} - \nabla F(w_{i+k})\| < \|g_{i+k}\| \tag{7}$$

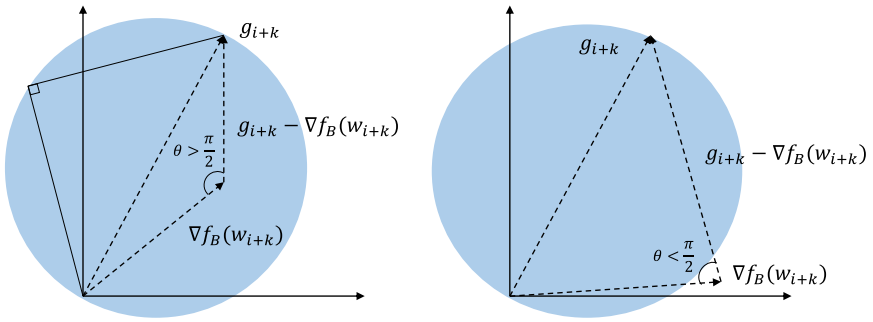

Fig. 1. Example of the gradients on 2-D vector space. With Inequality (8), g_{i+k} and $\nabla f_{\mathcal{B}}(w_{i+k})$ in both the left and right figures are considered as the same 'descent direction'. With Inequality (9), in contrast, the right side figure shows the angle between $g_{i+k} - \nabla f_{\mathcal{B}}(w_{i+k})$ and $\nabla f_{\mathcal{B}}(w_{i+k})$ which is smaller than $\frac{\pi}{2}$ and the two vectors are not considered as the same descent direction. Inequality (9) becomes true only when $\nabla f_{\mathcal{B}}(w_{i+k})$ is within the blue circle as shown in the left side figure. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Proof. g_{i+k} is the descent direction if and only if a condition below is true:

$$(g_{i+k})^T \nabla F(w_{i+k}) > 0$$

Considering the accumulated gradients g_{i+k} and the optimal gradients $\nabla F(w_{i+k})$ are two vectors, the above condition guarantees that the angle between the two vectors in the inner product space is smaller than $\frac{\pi}{2}$ so that they can be considered as the same 'descent' direction. Beginning from the proposed Inequality (7),

$$\|g_{i+k} - \nabla F(w_{i+k})\| < \|g_{i+k}\|$$
$$\|g_{i+k} - \nabla F(w_{i+k})\|^2 < \|g_{i+k}\|^2$$
$$\|g_{i+k}\|^2 - 2(g_{i+k})^T \nabla F(w_{i+k}) + \|\nabla F(w_{i+k})\|^2 < \|g_{i+k}\|^2$$
$$(g_{i+k})^T \nabla F(w_{i+k}) > \frac{1}{2}\|\nabla F(w_{i+k})\|^2 \geq 0$$

So, when Inequality (7) holds, g_{i+k} is a descent direction with respect to w_{i+k}. □

However, the left-hand side of Inequality (7) is prohibitively expensive to compute since the optimal gradient $\nabla F(w_{i+k})$ is computed from the entire training samples. To address this problem, we replace the optimal gradient on the left-hand side with the current stochastic gradient $\nabla f_{\mathcal{B}}(w_{i+k})$. Note that, as shown in Equation (6), each stochastic gradient is a random variable with mean of $\nabla F(w)$. The classical convergence analysis of SGD usually assumes that the stochastic gradients have a bounded variance σ^2 [5]. Thus, Inequality (7) can be approximated by a relaxed condition with a bounded difference as below.

$$\|g_{i+k} - \nabla f_{\mathcal{B}}(w_{i+k})\| < \|g_{i+k}\| \tag{8}$$

Since $\nabla f_{\mathcal{B}(w_{i+k})}$ is already computed at line 5 in Algorithm 1, Inequality (8) can be computed without extra computations.

Inequality (8) enables to check the direction of the accumulated gradients with a feasible computational cost. However, $\nabla f_{\mathcal{B}}(w_{i+k})$ is a biased estimator due to its variance. If the variance is large, the direction of the accumulated gradients can be much different from the optimal gradients while Inequality (8) still holds. We propose another practical sufficient condition which alleviates such an effect as follows.

$$\|g_{i+k} - \nabla f_{\mathcal{B}}(w_{i+k})\|^2 + \|\nabla f_{\mathcal{B}}(w_{i+k})\|^2 < \|g_{i+k}\|^2 \tag{9}$$

By expanding the left hand side, we can get an inequality,

$$(g_{i+k})^T \nabla f_{\mathcal{B}}(w_{i+k}) > \|\nabla f_{\mathcal{B}}(w_{i+k})\|^2 > 0,$$

which provides the same condition of g_{i+k} as Inequality (8). We define θ as the angle between two vectors, $g_{i+k} - \nabla f_{\mathcal{B}}(w_{i+k})$ and $\nabla f_{\mathcal{B}}(w_{i+k})$. The l2-norm of the stochastic gradients is most likely smaller than that of the accumulated gradients. Under this condition, $\|g_{i+k} - \nabla f_{\mathcal{B}}(w_{i+k})\|^2 + \|\nabla f_{\mathcal{B}}(w_{i+k})\|^2$ is smaller than $\|g_{i+k}\|^2$ only when θ is larger than $\frac{\pi}{2}$. Fig. 1 illustrates the proposed condition on the simplified 2-D vector space. The blue circle shows the region in which $\theta \geq \frac{\pi}{2}$. Inequality (9) becomes true only when $\nabla f_{\mathcal{B}}(w_{i+k})$ is within the blue region so that the angle θ is larger than $\frac{\pi}{2}$. So, this condition can be considered as a stricter condition than Inequality (8) for having g_{i+k} with a descent direction.

One potential issue of Inequality (9) is that, as k grows up, the blue circle shown in Fig. 1 is enlarged and it becomes easier for $\nabla f_{\mathcal{B}}(w_{i+k})$ to be inside of the circle. In order to check the direction of the accumulated gradients without being affected by k, we use the averaged accumulated gradients instead.

$$\|\frac{1}{k}g_{i+k} - \nabla f_{\mathcal{B}}(w_{i+k})\|^2 + \|\nabla f_{\mathcal{B}}(w_{i+k})\|^2 < \|\frac{1}{k}g_{i+k}\|^2 \tag{10}$$

By using the averaged accumulated gradients $\frac{1}{k}g_{i+k}$, the blue circle in Fig. 1 does not grow as k increases, and thus the angle can be checked more robustly. If $\|\nabla f_{\mathcal{B}}(w_{i+k})\|^2$ is larger than $\|\frac{1}{k}g_{i+k}\|^2$, we use the following Inequality instead.

$$\|\frac{1}{k}g_{i+k} - \nabla f_{\mathcal{B}}(w_{i+k})\|^2 + \|\frac{1}{k}g_{i+k}\|^2 < \|\nabla f_{\mathcal{B}}(w_{i+k})\|^2 \tag{11}$$

Inequality (10) and (11) equally check if the two gradients, g_{i+k} and $\nabla f_{\mathcal{B}}(w_{i+k})$, are close to each other.

3.2. Bounding gradient noise with respect to magnitude

To minimize the effect of the lazy updates on the convergence, the magnitude of the accumulated gradients also should be taken into account when adjusting the parameter update frequency. We assume F has Lipschitz-continuous gradients, that is, there is a constant $L > 0$ such that

$$\|\nabla F(u) - \nabla F(v)\| \leq L\|u - v\| \quad \forall u, v \in \mathbb{R}^d. \tag{12}$$

As shown in [3,4], the change in F is bounded by

$$F(u) - F(v) \leq \nabla F(v)^T (u - v) + \frac{L}{2}\|u - v\|^2. \tag{13}$$

From the above Inequality, we can get the bounded change of F for the lazy update as follows.

$$F(w_i) - F(w_{i+k}) \leq \nabla F(w_{i+k})^T (w_i - w_{i+k}) + \frac{L}{2}\|w_i - w_{i+k}\|^2$$

By inserting Equation (5) into the above Inequality,

$$F(w_i) - F(w_{i+k}) \leq \mu \nabla F(w_{i+k})^T (g_{i+k}) + \frac{L\mu^2}{2} \|g_{i+k}\|^2.$$

Considering $\mathbf{E}[g_{i+k}] = \sum_{j=1}^{k} \nabla F(w_{i+j})$,

$$F(w_i) - F(w_{i+k}) < \mu \|g_{i+k}\|^2 + \frac{L\mu^2}{2} \|g_{i+k}\|^2 = (\mu + \frac{L\mu^2}{2}) \|g_{i+k}\|^2. \tag{14}$$

Likewise, in SGD without the lazy update method, the change in F for k iterations is bounded as follows.

$$\begin{aligned} F(w_i) - F(w_{i+k}) &\leq \sum_{j=1}^{k} (\mu \nabla F(w_{i+j})^T \nabla f_{\mathcal{B}}(w_{i+j}) \\ &\quad + \frac{L\mu^2}{2} \|\nabla f_{\mathcal{B}}(w_{i+j})\|^2) \\ &\approx k(\mu + \frac{L\mu^2}{2}) \|\nabla f_{\mathcal{B}}(w_{i+k})\|^2 \end{aligned} \tag{15}$$

The above approximation is under an assumption that the magnitude of gradients is not significantly different among k iterations.

If the right-hand side of Inequality (14) is smaller than that of Inequality (15), SGD with the lazy update method can converge more slowly than without the lazy update method. To avoid such a potential slow convergence, we propose to check the following condition when adjusting the parameter update frequency.

$$k \|\nabla f_{\mathcal{B}}(w_{i+k})\|^2 < \|g_{i+k}\|^2 \tag{16}$$

Note that Inequality (16) is under a strong assumption that the magnitude of stochastic gradients is not significantly different across k iterations. In practice, if the batch size is small, the stochastic gradients have a high variance and the magnitude of the gradients can be unstable among k iterations. So, satisfying Inequality (16) cannot solely guarantee a sufficiently low degree of noise in the accumulated gradient. In our training algorithm, therefore, both Inequality (9) and (16) are taken into account when adjusting the parameter update frequency.

3.3. Adaptive lazy update interval

To strictly force the lazy update to be a descent direction, the state of the accumulated gradients should be checked at every iteration so as to stop the accumulation before the gradients become too noisy. However, such frequent checks will incur an expensive extra computational cost. Furthermore, we empirically found that the accumulated gradients violate Inequality (10) and (16) after a sufficiently large number of accumulations.

We design an adaptive lazy update interval calculation method which sparsely re-calculates the lazy update interval based on Inequality (10) and (16). As shown in Algorithm 1 at line 1, the lazy update interval k is initialized to 1 at the beginning of the training. During the training, k increases by 1 if both Inequality (10) and (16) are true. On the other hand, the interval is reduced by 1 if both Inequality (10) and (16) are not satisfied.

This constant back-off algorithm enables to sparsely check the state of the accumulated gradients and automatically makes the lazy update interval stay around the maximum value that allows descent direction gradients. Note that k is adjusted by 1, which is the minimum granularity of the interval adjustment. With a larger step size, k will approach to the maximum allowed interval faster, and thus a better scaling efficiency can be expected. On the other hand, the larger step size can give a larger difference between the left-hand side and the right-hand side in both Inequality (10) and (16), which implies a slower convergence. In this work, we focus on minimizing the accuracy drop while improving the scalability. So, we propose to use the minimum granularity of the interval adjustment.

Algorithm 2 Calculate_Interval($g, \nabla f_{\mathcal{B}}(w), b, k$).

```
1: angle ← false
2: magnitude ← false
3: b' ← layer with the largest |w| among bottom b layers.
4: diff ← ‖(1/k) g − ∇f_B(w)‖² at layer b'.
5: norm_g ← ‖(1/k) g‖² at layer b'.
6: norm_f ← ‖∇f_B(w)‖² at layer b'.
7: if (diff + norm_f) < norm_g then
8:     angle ← true
9: if (k · norm_f) < norm_g then
10:    magnitude ← true
11: if angle is true and magnitude is true then
12:    Increase k by 1.
13: else if angle is false and magnitude is false then
14:    Decrease k by 1.
```

We also propose to calculate Inequality (10) and (16) from a single layer that has the largest number of parameters among the bottom b layers. If they are computed from the whole parameters of the bottom b layers, one layer that satisfies the conditions by a huge margin can cancel out other layers' violations. Note that, due to the non-linearity between layers caused by non-linear activation functions or pooling layers, the gradients at different layers do not have a strong correlation. By calculating the two conditions from a single largest layer, we can achieve a more reliable interval adjustment as well as a cheaper computational cost.

3.4. Parallel implementation

Distributed gradient computation – In data parallel training, the communications for averaging the gradients at one layer and the gradient computations at other layers can be performed simultaneously since the gradients do not have data dependency across layers. However, if the overall communication time is longer than the backward computation time, a part of communications may not be hidden behind the computations causing a blocking time between every two consecutive iterations. In this work, we set b to the number of bottom layers whose communications cannot overlap with any computations. In practice, b can be found by checking how many communications have not been started when the back-propagation computation is completed at the bottom layer.

Equation (17) shows how the accumulated gradient g_{i+k} is computed in parallel training.

$$g_{i+k} = \sum_{p=1}^{P} g_{i+k}^p = \sum_{p=1}^{P} \sum_{j=1}^{k} \nabla f_{\mathcal{B}}^p(w_{i+j}), \tag{17}$$

where P is the number of workers, $\nabla f_{\mathcal{B}}^p(w)$ is the local stochastic gradients computed from $\frac{m}{P}$ training samples by worker p, and g^p is the locally accumulated gradients at worker p. On the right-hand side of Equation (17), the first summation is performed by inter-process communications (typically allreduce operations) and the second summation is the local accumulation based on Equation (4). Since $|g|$ and $|\nabla f_{\mathcal{B}}(w)|$ are the same, the overall communication cost within k iterations is reduced to $\frac{1}{k}$. Therefore, the bigger the k, the lower the communication cost.

Software framework – We implemented a software framework for neural network training in C language. The framework is specifically designed for parallel training on CPU-based distributed memory platforms. The implementation details can be found in our previous work [21]. We use Intel Math Kernel Library (Intel

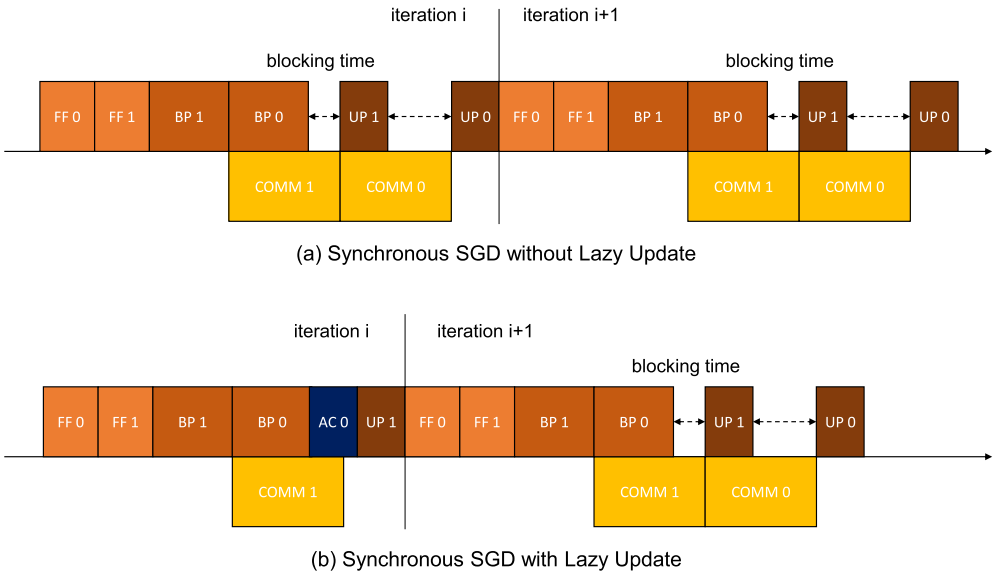

Fig. 2. A schematic time-flow chart for synchronous SGD with and without lazy update method. This example assumes a network with two layers (0 and 1). (**a**) shows the vanilla synchronous SGD without lazy update and (**b**) shows the synchronous SGD with lazy update. *FF* and *BP* indicate feedforward and backpropagation, respectively. *UP* indicates the parameter update. *AC* is the gradient accumulation that corresponds to Equation (4).

MKL) for the kernel functions such as matrix operations. We employ MPI-OpenMP programming model such that each MPI process runs on one node and the process performs the kernel operations, such as matrix multiplications, using OpenMP. This design option is due to the following two reasons. First, training a neural network requires a large amount of memory space to keep the intermediate data such as activations, errors, and gradients. Therefore, it is a common practice that only one process runs on each CPU node so that all the memory space is fully utilized by the process. On GPU systems, the common practice is to run as many processes as the number of GPUs per node for the same reason. Second, our design option minimizes the local data size per node in all the gradient communications. Because each process computes a fixed number of local gradients regardless of the number of processes, the data exchanged across the nodes proportionally increases as more processes run on each node. Therefore, depending on the allowed network bandwidth at each node, the communication time can increase if many processes run on each node.

Overlap of computation and communication – Fig. 2 shows a schematic illustration of the vanilla synchronous SGD without lazy update (**a**) and the synchronous SGD with lazy update (**b**). These illustrations show how the communications can be overlapped with the computations during parallel training. *FF*, *BP*, *UP*, and *AC* denote feedforward, backpropagation, parameter update, and gradient accumulation, respectively. *COMM* indicates the gradient communication. The postfix digit of each operation means the layer ID. In this example, we consider a network with two layers, and thus the layer ID is either 0 or 1. Once the gradients at layer 1 is computed (*BP 1*), the corresponding communication (*COMM 1*) is posted so that the communication time is overlapped with the backpropagation time at layer 0. At each iteration, the communications are posted twice, and a part of the communication time is exposed (blocking time) if the total communication time is longer than the backpropagation time and the parameter update time. In (**a**), the same pattern is repeated across all the iterations, and thus the blocking time takes up a large portion of the whole training time. In (**b**), the lazy update algorithm skips updating layer 0 accumulating the gradients. Because (**b**) does not have *COMM 0* at iteration i, the training can proceed to the iteration $i+1$ without having a blocking time. When updating the layer 0 at iteration $i+1$, a communication is posted for layer 0 to average the accumulated gradients among all the workers. So, this iteration has exactly the same computations and communications as the vanilla synchronous SGD.

There are several possible design options for implementing the overlap of the computations and communications. First, the gradient averaging algorithm we employed [21] can be implemented using MPI asynchronous API such as `MPI_Iallgather` and `MPI_Alltoall`. Many modern MPICH implementations support asynchronous progress of these collective MPI communications. Second, a helper thread can be employed to use MPI synchronous API explicitly overlapping the communications with the computations. Our software framework is implemented using a POSIX thread that is dedicated to the communications. The implementation details can be found in our previous works [21,23]. Third, any third-party communication libraries can be employed for asynchronous communications. For instance, one can consider using Casper, an adaptive asynchronous progress method designed for parallel applications [31]. As long as the requested communications are immediately started in background, a similar performance gain can be expected regardless of the design choice. Note that, the proposed lazy update method is readily applicable to any implementations because Algorithm 1 and 2 are independent of the underlying communication patterns.

4. Performance evaluation

We evaluate the performance of the proposed parallel CNN training algorithm using open benchmark datasets, CIFAR-10, ImageNet, and DIV2K. All experiments were carried out on Cori, a Cray XC40 supercomputer at National Energy Research Scientific Computing Center (NERSC). Each compute node has an Intel Xeon Phi Processor 7250, Knights Landing (KNL), that has 68 cores. AVX-512 vector pipelines with a hardware vector length of 512 bits are available at each node. In all our experiments, we used the 'cache' mode of MCDRAM. The system has Cray Aries high-speed interconnections with 'dragonfly' topology. When building our software framework on Cori, we used the system default Intel C++ compiler (19.0.3) and cray-MPICH (7.7.6) on Cori. Before evaluating the performance of our proposed lazy update method, we first present

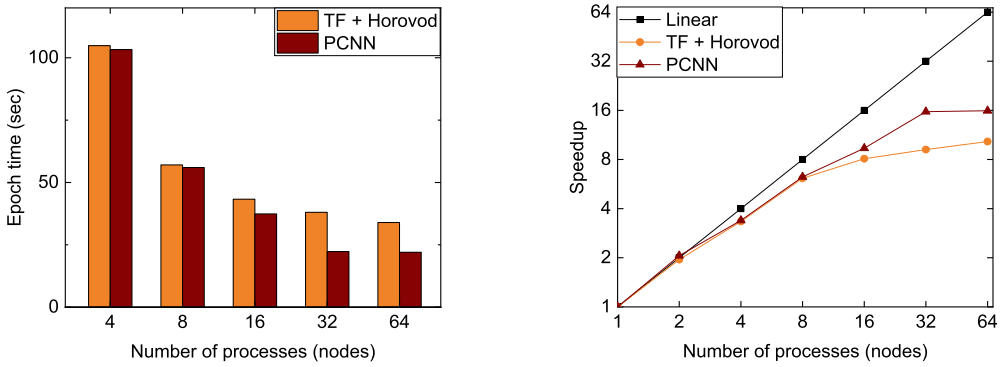

Fig. 3. The scaling performance comparison between TensorFlow + Horovod and *PCNN*, our own software framework [23]. We trained ResNet20 on CIFAR-10 using synchronous SGD and compared the average epoch time (sec) and the speedup on Cori KNL nodes. We see that *PCNN* slightly outperforms TensorFlow + Horovod mainly due to the overlap of computation and communication.

the scaling performance comparison between our software framework and TensorFlow (2.2.0) + Horovod (0.19.0), one of the most popular parallel training software packages. Both our framework and Horovod are built based on the system default cray-MPICH library. We trained ResNet20 using synchronous SGD and compared the average epoch time. Fig. 3 presents the scaling performance comparison. This comparison demonstrates that our scaling performance study is based on a reasonable baseline performance. The same experimental result can be found in our previous work [23].

We compare scaling performance among four different parallel training settings. First, 'No overlap' represents data parallel training without overlapping. In this setting, the gradient communications are posted after computing all the gradients, and thus the entire communication time is exposed. Horovod is one of the most popular software framework that falls into this setting [30]. Second, 'Without LazyUp' indicates data parallel training with overlapping. We employed an overlapping strategy that allows to overlap the communications with not only the backward computations but also the forward computations at the next iteration, proposed in [21]. A similar overlapping strategy was proposed in [38]. Third, 'With LazyUp' is data parallel training that employs the lazy update method. The communications are overlapped with computations using the same algorithm as 'Without LazyUp'. Finally, 'Comp only' is the computation only performance. We consider this computation timings as the ideal performance we can achieve in parallel training.

Note that we do not compare the performance between the proposed method and gradient quantization/sparsification methods. These methods zero out small gradients or replace the similar gradient values with a single representative value based on the gradient properties such as magnitude of each gradient or statistical distribution of the absolute gradient values. In our proposed method, the gradients are not modified but accumulated at a part of model layers. We consider the gradient quantization and sparsification methods and our proposed method are orthogonal techniques. In other words, we can apply both of them to parallel training without any conflicts, expecting a better scaling performance. Therefore, we do not directly compare the performance between them.

4.1. CIFAR-10 classification

CIFAR-10 has 50,000 training images and each image size is 32×32. We present the performance of Wide-ResNet20 training for CIFAR-10 classification. Wide-ResNet20 is a variant of ResNet20, which has an increased width of residual blocks compared to the original model architecture. In this experiment, we double the number of filters at all the convolution layers. All the hyper-parameter settings are the same as shown in [44]. We use mini-batch SGD with the batch size of 128, the initial learning rate of 0.1, and the momentum factor of 0.9. The learning rate decreases by a factor of 5 after 60, 80, and 120 epochs. The training is performed for 200 epochs in total. We use up to 64 compute nodes in which the baseline execution time starts to increase. We found that the communications at the bottom 6 layers do not overlap with any computations on 64 nodes, so we set $b = 8$.

Fig. 4 presents the training loss (left) and the validation accuracy (right). We performed the parallel training on 64 KNL nodes and collected the learning curves. We can observe that the lazy update method provides almost the same convergence accuracy as the original mini-batch SGD (Without Lazy Update: $94.23 \pm 0.1\%$, With Lazy Update: $94.36 \pm 0.2\%$). Note that the reported accuracy of similar Wide-ResNet series in [44] is $93.58\% \sim 94.27\%$. Fig. 5 shows the strong scaling performance. First, 'No overlap' shows a significantly longer execution time than the others. The difference between 'No overlap' and 'Comp only' can be considered as the overall communication time. Our method reduces the average execution time per epoch from 64.05 seconds to 42.47 seconds on 64 nodes. We can see that 'With LazyUp' and 'Comp only' timings are almost the same, which means the communication cost is effectively reduced by the proposed method.

One noticeable observation is that the computation time does not linearly scale up. As the number of workers increases, the number of local training samples is proportionally reduced in data parallelism. It is already known that a sufficiently large amount of workload is needed to fully utilize the computation resources in each KNL node [18,27]. In our experiments, we found that Intel MKL shows a poor scaling performance when the number of local training samples is lower than 16. Another factor that affects the scaling efficiency is the model size. The input data size of Wide-ResNet20 for CIFAR-10 classification is 32×32 and the number of filters at the first few convolution layer is 16 only. Such a small model size causes small matrix operations which cannot fully-utilize the computation power.

4.2. ImageNet classification

ImageNet is a large-scale image dataset for classification, that consists of 1.2 millions of high-quality images with varying sizes. We train ResNet50 on ImageNet and compare the performance between with and without the lazy update method. We use mini-batch SGD with the batch size of 256, the initial learning rate of 0.1, and the momentum factor of 0.9. The learning rate is reduced by a factor of 10 after 30, 60, and 80 epochs as shown in [11]. The

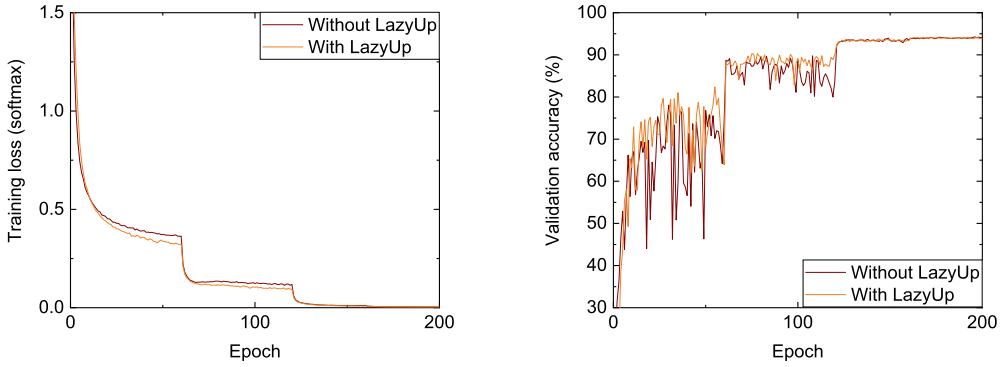

Fig. 4. Training loss curves (left) and validation accuracy curves (right) of Wide-ResNet20 training on CIFAR-10. Both the training loss and the validation accuracy are almost not affected by the proposed lazy update method (Without LazyUp: $94.23 \pm 0.1\%$, With LazyUp: $94.19 \pm 0.1\%$).

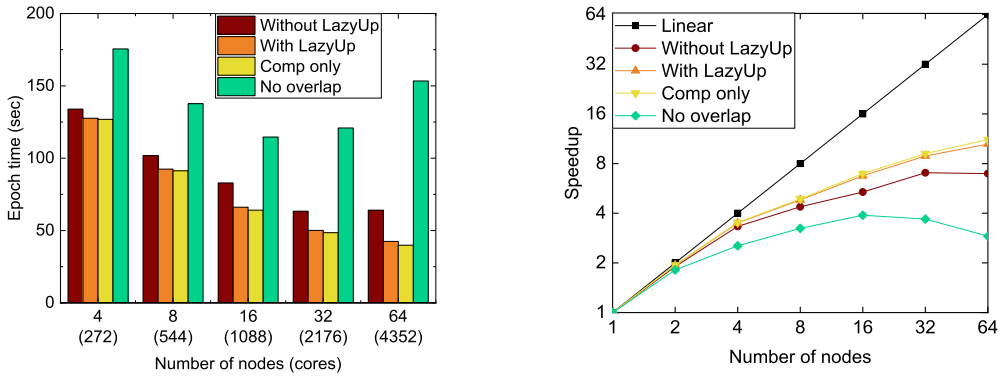

Fig. 5. Execution time and speedup of parallel Wide-ResNet20 training on CIFAR-10. The proposed method effectively reduce the communication time so that it achieves almost the same execution time to the computation-only time.

training is scaled up to 256 nodes where the training time without the proposed method starts to increase. We found that the communications at the bottom $24 \sim 25$ layers do not overlap with the computations on 256 nodes, so we set $b = 25$.

Fig. 6 shows the training loss (left) and the validation accuracy (right). We performed the parallel training on 128 KNL nodes and collected the learning curves. Our proposed training method quickly increases the validation accuracy in the early epochs and then it ends up being saturated to a similar accuracy to the baseline (Without LazyUp: $75.89 \pm 0.2\%$, With LazyUp: $75.81 \pm 0.2\%$). Note that our baseline accuracy is slightly higher than that in [15] (75.3%) and lower than that in [11] (76.26%). Considering the long training time, we employed only the basic data augmentation as shown in [41], such that each image is re-scaled to 256×256 and a random 224×224 patch is extracted from it. So, our validation accuracy is slightly lower than that reported in [11] achieved by rich data augmentation. Fig. 7 shows the average execution time per epoch (left) and the speedup (right). The proposed method reduces the execution time from 0.62 hours to 0.27 hours. We see that the execution time of the proposed method is almost the same as the computation-only time. As a result of the reduced parameter update frequency at the bottom 25 layers, the per-iteration communication time is effectively reduced, and thus 'With LazyUp' achieves a similar speedup to 'Comp only'.

4.3. DIV2K image super-resolution

We perform image super-resolution using Enhanced Deep Super-Resolution (EDSR) [25] and DIV2K dataset. EDSR is a deep CNN which has 32 residual blocks and 256 filters per layer. DIV2K is a dataset from NTIRE2017 Super-Resolution Challenge [36], which contains 800 high-quality 2K resolution pictures. We use Adam with a batch size of 64 and a learning rate of 0.0004. All the other hyper-parameters are set to the same values as shown in [25]. For the regression accuracy, we use Peak Signal-to-Noise Ratio (PSNR) which measures how much different the given two images are. The training is scaled up to 64 nodes where the number of local training samples becomes 1. The communications at the bottom 14 layers do not overlap with the computations on 64 nodes, so we set $b = 14$.

Fig. 8 presents the training loss curves (left) and the validation PSNR curves (right). We performed the parallel training on 64 KNL nodes and collected the learning curves. Our training method achieves a comparable validation PSNR (33.54 ± 0.1 dB) to the baseline (33.55 ± 0.1 dB). Fig. 9 shows the execution time per epoch and speedup. The training with our method takes 31.23 seconds per epoch while the baseline takes 56.06 seconds. Like the other experiments, we see that the proposed method reduces the communication cost and the scaling efficiency is improved.

4.4. Performance analysis and discussion

Impact of the number of lazy update layers on accuracy – We set b to the number of layers whose communications do not overlap with any computations due to the high ratio of communication to computation. Thus, the value of b is affected by the factors that determine the ratio of communication to computation, such as hardware configurations and hyper-parameter settings. To investigate the impact of the value of b on the training results, we compare the learning curves of ImageNet training across different

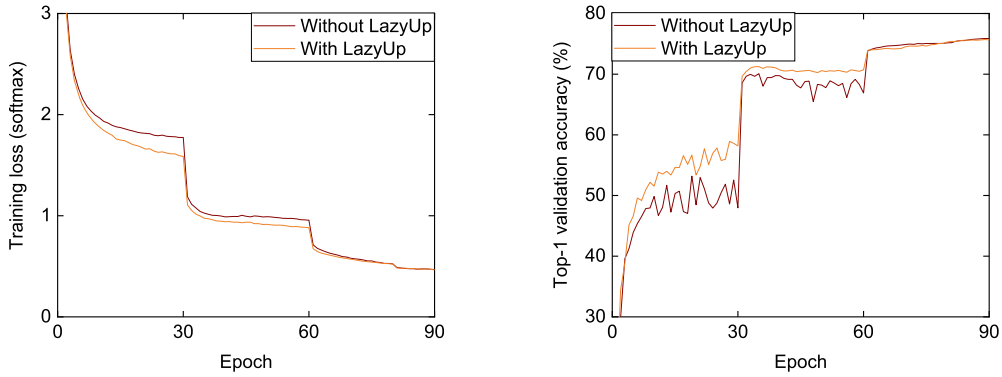

Fig. 6. Training loss (left) and validation accuracy (right) of ResNet50 training on ImageNet. The proposed method increases the validation accuracy faster, however both curves end up being saturated to a similar accuracy (Without LazyUp: $75.89 \pm 0.2\%$, With LazyUp: $75.81 \pm 0.2\%$).

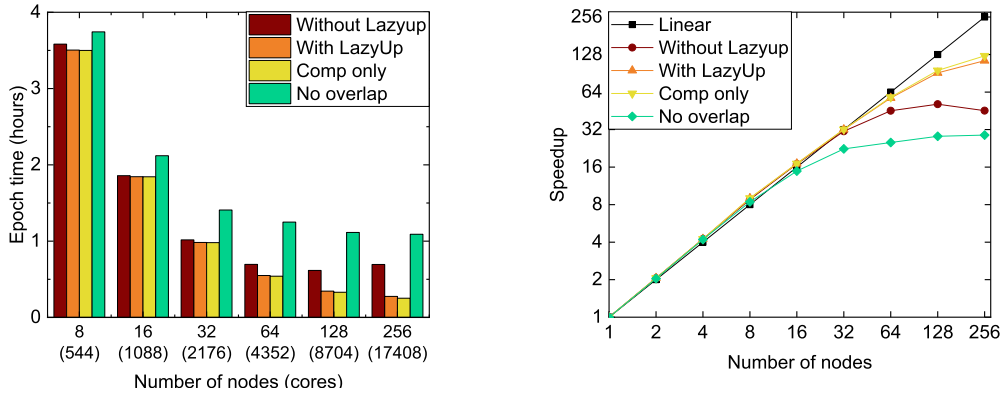

Fig. 7. Execution time and speedup of parallel ResNet50 training on ImageNet. Our proposed algorithm achieves a speedup of 115.11 on 256 nodes while the baseline peaks on 128 nodes achieving a speedup of 50.96.

b values. We use three b settings, 12, 25, and 37. ResNet50 has 50 layers with tunable parameters in total. So, these three b settings roughly represent 25%, 50%, and 75% of the parameters, respectively. Fig. 10 presents the training loss (left) and the validation accuracy (right). We first see that all the three b settings provide a similar convergence training loss. In contrast, the larger the b, the lower the convergence validation accuracy. The three b settings give a validation accuracy of $75.86 \pm 0.1\%$, $75.81 \pm 0.2\%$, and $75.33 \pm 0.2\%$, respectively.

As b increases, more parameters are updated using the accumulated gradients that are noisy. As explained in Section 3.1, the accumulated gradients are noisier than the stochastic gradients computed from a single mini-batch. Although Inequality (10) and (16) bound the degree of noise, we can expect a lower quality of updates as more parameters are updated using the accumulated gradients. However, even when the lazy update method is applied to about 75% of parameters, the validation accuracy is still comparable to that of the baseline ($< 0.5\%$ difference). This experimental result demonstrates that the proposed lazy update method can be generally applied to different systems regardless of the ratio of computations to communications.

Adaptive lazy update interval – Algorithm 1 repeatedly adjusts the lazy update interval based on Inequality (9) and Inequality (16). Fig. 11 shows the average lazy update interval k within each epoch of (a) CIFAR-10, (b) ImageNet, and (c) DIV2K training. The average lazy update interval among the whole training epochs is about 10, 18, and 5 for the three datasets, respectively. Considering the number of iterations per epoch, 390 for CIFAR-10, 5,004 for ImageNet, and 50 for DIV2K, such large update intervals mean that the parameter update frequency at the bottom side layers is remarkably reduced, and thus the per-iteration communication cost is expected to be reduced.

Timing breakdowns – We analyze how the proposed method affects the computation time and communication time. The gradient accumulation, Equation (4), has a cheaper computational cost than the regular parameter update shown in Equation (3). So, the only extra computational cost of our method comes from the lazy update interval computation in Algorithm 2. The computational complexity of Algorithm 2 is $O(|w_{b'}|)$, where b' is the layer with the largest number of parameters among the bottom b layers. Table 1 shows the timing breakdown of the average computation time per epoch. Compared to the feed-forward, backpropagation, and parameter update times, the interval calculation time takes up almost a negligible portion of the total computation time.

Table 2 shows the timing breakdown of the average execution time per epoch. The 'regular' columns show the timings at k^{th} iteration while the 'lazy' columns show the timings at all the other iterations $(0, \cdots, k-1)$. We see that the communication time is significantly reduced when the gradients are accumulated for $k-1$ iterations. In our implementation, the communications at the top $(l-b)$ layers overlap with the backpropagation computations. The 'Overall' row shows the actual end-to-end execution time which is the sum of the computation time and the exposed communication time. If all the communications are overlapped with the computations ideally, the 'Overall' timing becomes as short as the computation time. That is, the shorter the 'Overall' time, the better the scaling efficiency. Note that the communication time at k^{th} iteration is almost the same as that of the baseline because $|g_{i+k}|$ and $|f_{\mathcal{B}}(w_{i+k})|$ are the same at the bottom b layers.

Scalability of computation – In all our experiments, we observe that the computation-only timings do not provide linear speedup.

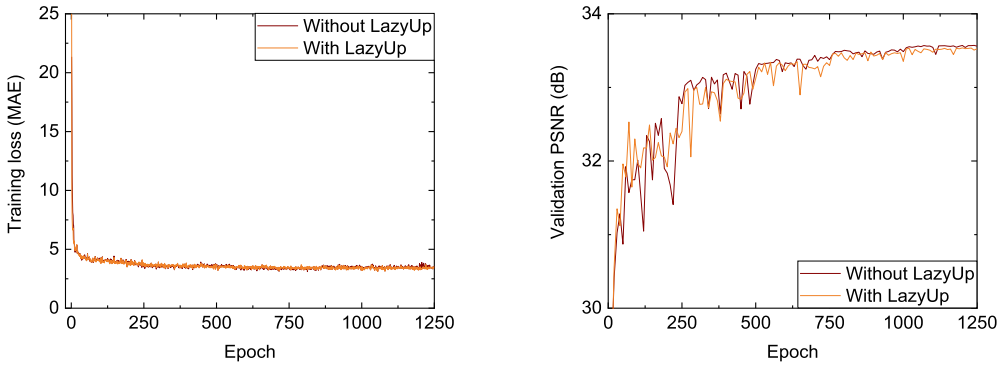

Fig. 8. Training loss (left) and validation accuracy (right) of EDSR training on DIV2K dataset. Both the training loss and the validation accuracy are almost the same for the whole training.

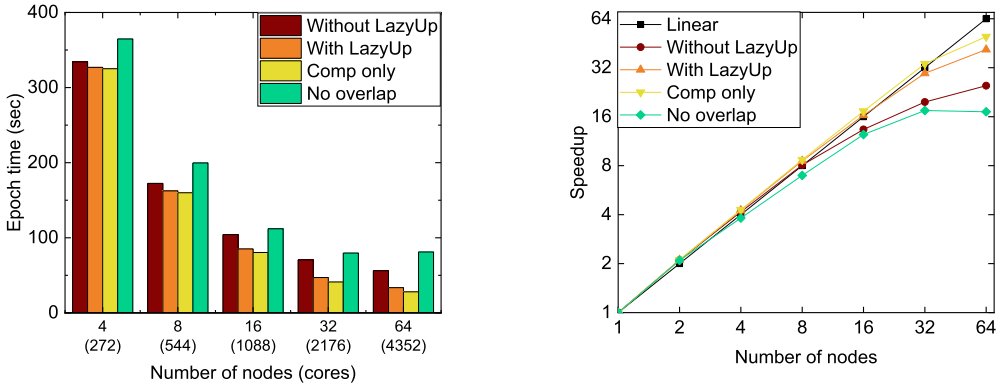

Fig. 9. Execution time and speedup of parallel EDSR training on DIV2K. Our proposed lazy update method improves the speedup from 25.01 to 44.67 on 64 KNL nodes.

As the number of workers increases, the number of local training samples is proportionally reduced in data parallelism. It is already known that a sufficiently large amount of workload is needed to fully utilize the computation resources in each KNL node [18,27]. In our experiments, we found that Intel MKL BLAS library shows a poor scaling performance when the number of local training samples is lower than 16. Another factor that affects the scaling efficiency is the model size. For example, in Wide-ResNet20 training, the input data size is 32×32 and the number of filters at the first few convolution layer is 16 only. Such a small model size causes small matrix operations which cannot fully-utilize the computation power.

Potential drawbacks of lazy update method – In order to present a holistic view of our study, we discuss the potential drawbacks of the lazy update method. First, Algorithm 1 spends an extra memory space to keep the accumulated gradients g at the bottom b layers ($\sum_{i=1}^{b} |g_i|$ floating-point numbers). However, considering the rich memory space of modern HPC systems, this memory consumption can be considered as negligible. For instance, ResNet50 has ~ 25.6 millions of parameters that take 98 MB of memory space while each Cori KNL node has 96 GB DDR4 memory space. Moreover, the extra memory space is smaller than the whole model size because $b < l$. Second, the lazy update method can cause a minor validation accuracy drop. The impact of the delayed updates on the model accuracy heavily depends on the input data. Although the noise of the accumulated gradient is bounded in both its direction and magnitude, our theoretical analysis is focused only on the training loss. Our future work includes analyzing the impact of the lazy update method on the generalization performance.

Table 1
Timing Breakdown of Computation Time (sec).

-	CIFAR-10	ImageNet	DIV2K
# of nodes	64	256	64
Feed-forward	9.23	190.98	8.89
Backprop	27.38	544.43	18.30
Update	6.51	163.14	3.74
Interval Calc.	0.27	7.77	0.20
Total	43.49	906.32	31.13

Table 2
Timing Breakdown of Overall Execution Time (sec).

-	CIFAR-10		ImageNet		DIV2K	
# of nodes	64		256		64	
Update type	regular	lazy	regular	lazy	regular	lazy
Comp	42.01	43.81	900.47	908.49	30.08	32.01
Comm	72.34	42.21	2101.48	900.37	51.39	27.39
Overall	77.66	44.49	2432.24	958.4	56.07	32.27

4.5. Large batch training

Increasing the batch size is an intuitive way to improve the degree of parallelism. A large batch size means that each mini-batch can be distributed to more processes and concurrently processed. Recently, several large-scale scientific applications used large batch sizes to scale up the training [10,19,20,28]. However, the batch size does not affect the communication cost for averaging the gradients at each iteration. As the number of processes increases, the ratio of computation to communication within each iteration drops by the nature of the strong scaling, and the communication time ends

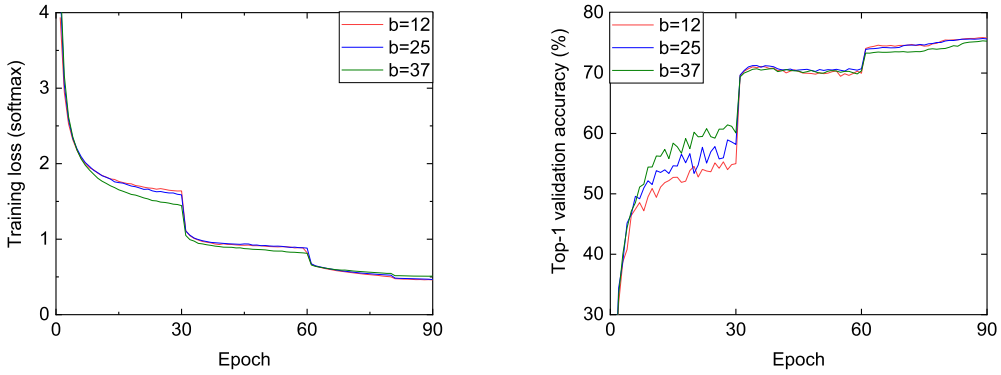

Fig. 10. Learning curves of ResNet50 training on ImageNet with varying b values. $b = 12$, $b = 25$, and $b = 37$ give a validation accuracy of $75.86 \pm 0.1\%$, $75.81 \pm 0.2\%$, and $75.33 \pm 0.2\%$, respectively.

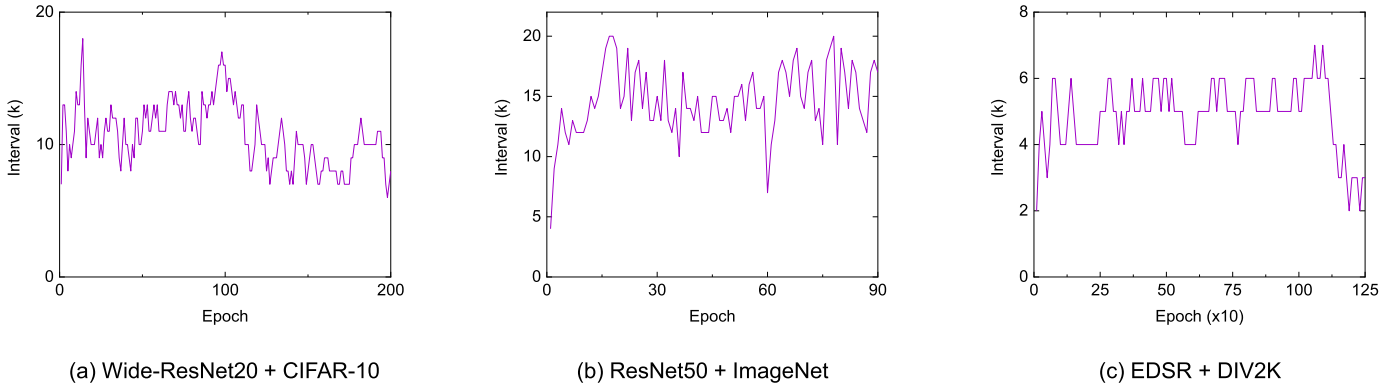

Fig. 11. The lazy update intervals measured in (a) CIFAR-10, (b) ImageNet, and (c) DIV2K training. The three interval curves correspond to the results shown in Fig. 4, Fig. 6, and Fig. 8, respectively. For all the three cases, the interval is much larger than 1, which means the parameters are less frequently updated at the bottom b layers.

up being dominant over the computation time. Thus, in order to achieve a good scaling efficiency, the expensive per-iteration communication problem should be addressed regardless of the batch size.

We report the large batch training learning curves and scaling performance and then analyze the impact of our proposed lazy update method on the large batch training. There are two possible learning rate settings for large batch training, linear scaling rule [11] and root scaling rule [16]. Goyal et al. empirically showed that the batch size and learning rate can be proportionally increased until a problem-dependent threshold without a significant loss in accuracy. Hoffer et al. theoretically explained that the variance of stochastic gradients stays the same if the learning rate is increased by a square root of the ratio of the increased batch size to the base batch size. However, it has been shown that the root scaling rule yields a significantly lower validation accuracy for AlexNet training with a batch size of $8,192$ [16] (about 4% accuracy difference). So, in this paper, we consider the large batch training with the linear scaling rule as a baseline and compare it to the large batch training with the proposed method.

Fig. 12 shows the learning curve comparison between with and without the lazy update method. The batch size is set to $8,192$ and the initial learning rate is 3.2. The learning rate decays by a factor of 10 after 30, 60, and 80 epochs. The gradual learning rate warm-up technique is used in the first 5 epochs as explained in [11]. Our parameter update rule is the classical update rule which multiplies the learning rate after adding the momentum term to the gradients. So, we do not apply the momentum correction method proposed in [11]. We can see that the proposed training method achieves a comparable validation accuracy ($75.22 \pm 0.1\%$) to the baseline ($75.41 \pm 0.1\%$). Note that, for the lazy update method, b is set to 24, which means the lazy update method is applied to the 24 bottom layers. We see that the training with the lazy update method achieves almost the same training and validation curves to the baseline. The average update interval at the bottom 24 layers is between 5 and 6.

Fig. 13 presents the scaling performance of large batch training for ImageNet classification. We scale the training up to $2,048$ nodes ($139,264$ cores) where the baseline stops scaling. First, without overlapping the communications with the computations ('No overlap'), the speedup is flattened from 256 processes. When the communications are overlapped ('Without LazyUp'), the speedup is improved, however the scaling still stops from 512 processes. By applying our proposed method ('With LazyUp'), the scaling efficiency is effectively improved and it achieves a speedup of 739.56 on $2,048$ nodes. The speedup of the computation time ('Comp only') is 1208.90 on $2,048$ nodes.

Discussion – The increased batch size improves the degree of parallelism. However, as the number of processes increases, it becomes more challenging to achieve a good scaling efficiency due to the increased communication cost. For example, let us consider a case in which the number of local training samples per process is 8. If the training with a mini-batch size of 256 is scaled up to 32 processes, each process works on 8 samples per iteration and the achieved speedup is 22.4 (70% scaling efficiency) even without overlapping the communications. If the batch size is increased to 8192 and the training is scaled up to 1024 processes, each process handles the same 8 samples per iteration. However, the achieved speedup is only 171.4 (16.7% scaling efficiency). Although each process has the same computational workload per iteration, the large batch training averages the gradients across a larger number of processes, and the communication cost is signif-

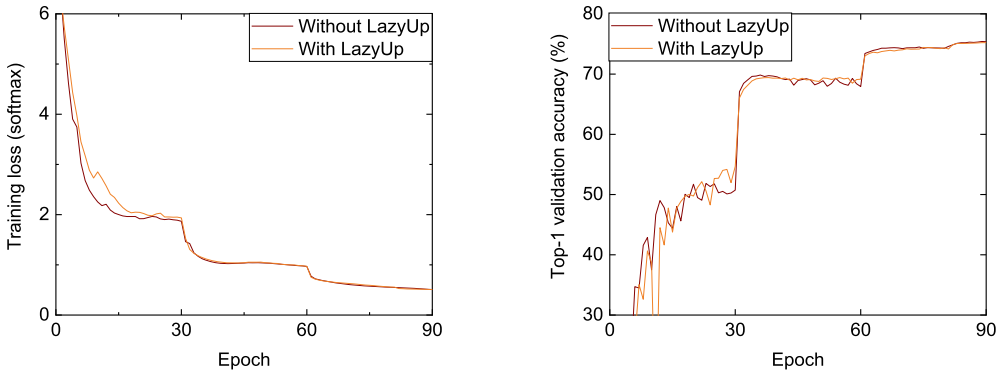

Fig. 12. The training loss (left) and top-1 validation accuracy of ResNet50 training with a batch size of 8, 192. Both the training and validation curves do not show a large difference ($75.41 \pm 0.1\%$ vs $75.22 \pm 0.1\%$).

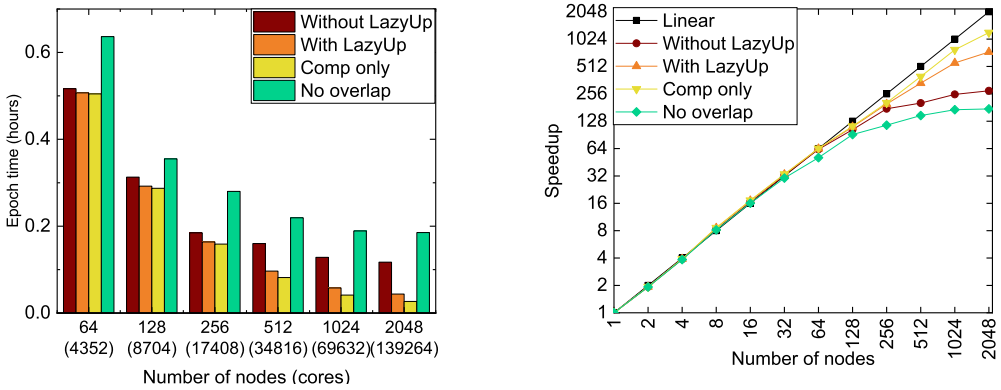

Fig. 13. The scaling performance of ResNet50 training with a batch size of 8, 192. The training scales up to 2, 048 processes on 2, 048 Cori KNL nodes. 'With LazyUp' shows a significantly improved scalability than 'Without LazyUp'.

icantly increased. This scaling efficiency issue can be alleviated by applying our proposed method. As shown in Fig. 13, the lazy update method lowers the update frequency at the bottom layers and the per-iteration communication cost is significantly reduced. This result demonstrates that the proposed method effectively improves the scaling efficiency regardless of the mini-batch size.

In our ImageNet training with a batch size of 8192, the update interval k moves between 5 and 6 during the whole training. This means that the parameters at the bottom 24 layers are updated using the accumulated gradients computed from 40K $\sim$ 48K training samples. This result implies that, by having a different update frequency across layers, we can break the problem-dependent threshold of batch size that degrades the generalization performance.

5. Conclusion

In this paper, we proposed a parallel CNN training algorithm that adjusts the parameter update frequency at a part of model layers at run-time. In this study, we get an insight that a fixed same parameter update frequency at all the layer may not be needed for minimizing the cost function. Our experimental results demonstrate that the lower parameter update frequency at the input side layers than the output side layers does not much affect the convergence accuracy while significantly reducing the per-iteration communication cost in parallel training. We also empirically proved that the proposed method effectively improves the scaling efficiency of the large batch training as well. Our approach is to control the parameter update frequency based on the degree of noise in the gradients during the training. So, although we studied the scaling performance of parallel CNN training, the lazy update method can be applied to other types of neural networks such as fully-connected network or recurrent neural network. If a small accuracy drop is acceptable by users, the proposed training strategy can be a practical option for large-scale deep learning-based applications. In addition, we could observe that our proposed training method increases the validation accuracy more rapidly than the classical mini-batch SGD training before decaying the learning rate. We believe that explaining this symptom and finding a way of keeping the good generalization performance until the convergence can be an interesting future work.

CRediT authorship contribution statement

Sunwoo Lee: Conceptualization, Investigation, Methodology, Software, Writing – original draft. **Qiao Kang:** Conceptualization, Writing – review & editing. **Reda Al-Bahrani:** Conceptualization, Writing – review & editing. **Ankit Agrawal:** Supervision, Writing – review & editing. **Alok Choudhary:** Supervision. **Wei-keng Liao:** Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This material is based upon work supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Sci-

entific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program award numbers DE-SC0021399 and DE-SC0019358, and the NIST award 70NANB19H005.

References

[1] A.F. Aji, K. Heafield, Sparse communication for distributed gradient descent, preprint, arXiv:1704.05021, 2017.

[2] D. Alistarh, D. Grubic, J. Li, R. Tomioka, M. Vojnovic, Qsgd: communication-efficient sgd via gradient quantization and encoding, in: Advances in Neural Information Processing Systems, 2017, pp. 1709–1720.

[3] L. Balles, J. Romero, P. Hennig, Coupling adaptive batch sizes with learning rates, preprint, arXiv:1612.05086, 2016.

[4] L. Bottou, F.E. Curtis, J. Nocedal, Optimization methods for large-scale machine learning, SIAM Rev. 60 (2018) 223–311.

[5] S. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, 2004.

[6] C.Y. Chen, J. Choi, D. Brand, A. Agrawal, W. Zhang, K. Gopalakrishnan, Adacomp: adaptive residual gradient compression for data-parallel distributed training, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[7] G. Cong, O. Bhardwaj, M. Feng, An efficient, distributed stochastic gradient descent algorithm for deep-learning applications, in: 2017 46th International Conference on Parallel Processing (ICPP), IEEE, 2017, pp. 11–20.

[8] 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.

[9] J. Deng, W. Dong, R. Socher, L.J. Li, K. Li, L. Fei-Fei, ImageNet: a large-scale hierarchical image database, in: CVPR09, 2009.

[10] W. Dong, M. Keceli, R. Vescovi, H. Li, C. Adams, T. Uram, V. Vishwanath, B. Kasthuri, N. Ferrier, P. Littlewood, Scaling distributed training of flood-filling networks on hpc infrastructure for brain mapping, preprint, arXiv:1905.06236, 2019.

[11] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch sgd: training imagenet in 1 hour, preprint, arXiv:1706.02677, 2017.

[12] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding, preprint, arXiv:1510.00149, 2015.

[13] S.H. Hashemi, S.A. Jyothi, R.H. Campbell, Tictac: accelerating distributed deep learning with communication scheduling, preprint, arXiv:1803.03288, 2018.

[14] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[15] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, M. Li, Bag of tricks for image classification with convolutional neural networks, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 558–567.

[16] E. Hoffer, I. Hubara, D. Soudry, Train longer, generalize better: closing the generalization gap in large batch training of neural networks, in: Advances in Neural Information Processing Systems, 2017, pp. 1731–1741.

[17] J. Kiefer, J. Wolfowitz, et al., Stochastic estimation of the maximum of a regression function, Ann. Math. Stat. 23 (1952) 462–466.

[18] K. Kim, T.B. Costa, M. Deveci, A.M. Bradley, S.D. Hammond, M.E. Guney, S. Knepper, S. Story, S. Rajamanickam, Designing vector-friendly compact blas and lapack kernels, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2017, p. 55.

[19] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, et al., Exascale deep learning for climate analytics, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, IEEE Press, 2018, p. 51.

[20] T. Kurth, J. Zhang, N. Satish, E. Racah, I. Mitliagkas, M.M.A. Patwary, T. Malas, N. Sundaram, W. Bhimji, M. Smorkalov, et al., Deep learning at 15pf: supervised and semi-supervised classification for scientific data, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2017, p. 7.

[21] S. Lee, A. Agrawal, P. Balaprakash, A. Choudhary, W.K. Liao, Communication-efficient parallelization strategy for deep convolutional neural network training, in: 2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC), IEEE, 2018, pp. 47–56.

[22] S. Lee, D. Jha, A. Agrawal, A. Choudhary, W.k. Liao, Parallel deep convolutional neural network training by exploiting the overlapping of computation and communication, in: 2017 IEEE 24th International Conference on High Performance Computing (HiPC), IEEE, 2017, pp. 183–192.

[23] S. Lee, Q. Kang, S. Madireddy, P. Balaprakash, A. Agrawal, A. Choudhary, R. Archibald, W.k. Liao, Improving scalability of parallel cnn training by adjusting mini-batch size at run-time, in: 2019 IEEE International Conference on Big Data (Big Data), IEEE, 2019, pp. 830–839.

[24] H. Li, Deep learning for natural language processing: advantages and challenges, Nat. Sci. Rev. (2017).

[25] B. Lim, S. Son, H. Kim, S. Nah, K. Mu Lee, Enhanced deep residual networks for single image super-resolution, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2017, pp. 136–144.

[26] Y. Lin, S. Han, H. Mao, Y. Wang, W.J. Dally, Deep gradient compression: reducing the communication bandwidth for distributed training, preprint, arXiv:1712.01887, 2017.

[27] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, J. Dongarra, High-performance matrix-matrix multiplications of very small matrices, in: European Conference on Parallel Processing, Springer, 2016, pp. 659–671.

[28] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S.J. Pennycook, et al., Cosmoflow: using deep learning to learn the universe at scale, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, IEEE Press, 2018, p. 65.

[29] F. Seide, H. Fu, J. Droppo, G. Li, D. Yu, 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns, in: Fifteenth Annual Conference of the International Speech Communication Association, 2014.

[30] A. Sergeev, M.D. Balso, Horovod: fast and easy distributed deep learning in TensorFlow, preprint, arXiv:1802.05799, 2018.

[31] M. Si, A.J. Pena, J. Hammond, P. Balaji, M. Takagi, Y. Ishikawa, Dynamic adaptable asynchronous progress model for mpi rma multiphase applications, IEEE Trans. Parallel Distrib. Syst. 29 (2018) 1975–1989.

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

[33] N. Strom, Scalable distributed dnn training using commodity gpu cloud computing, in: Sixteenth Annual Conference of the International Speech Communication Association, 2015.

[34] C. Szegedy, A. Toshev, D. Erhan, Deep neural networks for object detection, in: Advances in Neural Information Processing Systems, 2013, pp. 2553–2561.

[35] Y. Tai, J. Yang, X. Liu, Image super-resolution via deep recursive residual network, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 3147–3155.

[36] R. Timofte, E. Agustsson, L. Van Gool, M.H. Yang, L. Zhang, B. Lim, S. Son, H. Kim, S. Nah, K.M. Lee, et al., Ntire 2017 challenge on single image super-resolution: methods and results, in: Computer Vision and Pattern Recognition Workshops (CVPRW), 2017 IEEE Conference on, IEEE, 2017, pp. 1110–1121.

[37] H. Wang, S. Guo, R. Li, Osp: overlapping computation and communication in parameter server for fast machine learning, in: Proceedings of the 48th International Conference on Parallel Processing, 2019, pp. 1–10.

[38] S. Wang, A. Pi, X. Zhou, Scalable distributed dl training: batching communication and computation, in: Proc. of AAAI, 2019.

[39] J. Wangni, J. Wang, J. Liu, T. Zhang, Gradient sparsification for communication-efficient distributed optimization, in: Advances in Neural Information Processing Systems, 2018, pp. 1299–1309.

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

[41] Y. You, I. Gitman, B. Ginsburg, Large batch training of convolutional networks, preprint, arXiv:1708.03888, 2017.

[42] Y. You, Z. Zhang, C.J. Hsieh, J. Demmel, K. Keutzer, Imagenet training in minutes, in: Proceedings of the 47th International Conference on Parallel Processing, 2018, pp. 1–10.

[43] T. Young, D. Hazarika, S. Poria, E. Cambria, Recent trends in deep learning based natural language processing, IEEE Comput. Intell. Mag. 13 (2018) 55–75.

[44] S. Zagoruyko, N. Komodakis, Wide residual networks, preprint, arXiv:1605.07146, 2016.

[45] S. Zhang, L. Wen, X. Bian, Z. Lei, S.Z. Li, Single-shot refinement neural network for object detection, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 4203–4212.

Sunwoo Lee received his B.S. degree and M.S. degree in Computer Engineering from Hanyang University, South Korea. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at Northwestern University. Sunwoo Lee's research interests lie in scalable large-scale deep learning algorithms and its applications.

Qiao Kang received his first-class B.S. degree in Computer Science and Statistics from the University of St. Andrews. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at Northwestern University. Qiao Kang's research interests lie in High-Performance Computing and spatio-temporal anomaly detection.

Reda Al-Bahrani received his B.S. in Computer Science from King Fahd University of Petroleum and Minerals, Saudi Arabia and M.S. in eCommerce Technology from DePaul University. He recently obtained a Ph.D. from the Department of Electrical and Computer Engineering at Northwestern University. Reda Al-Bahrani's research interests lie in the area machine learning and its applications.

Ankit Agrawal (Ph.D. 2009, B.Tech. 2006) is a Research Associate Professor in the Department of Electrical and Computer Engineering at Northwestern University, USA. He specializes in interdisciplinary big data analytics via high performance data mining, based on a coherent integration of high-performance computing and data mining to develop customized solutions for big data problems.

Prof. Alok Choudhary is both a professor and an entrepreneur. Dr. Alok Choudhary is the founder of 4C insights, a data science and AI software company. Prof. Alok Choudhary's research over the last decades focused on big data science, supercomputing, scalable data mining, Machine Learning, AI and their applications in sciences (e.g. climate understanding, astrophysics, designing materials etc.), medicine and business applications.

Prof. Wei-keng Liao is a Research Professor in the Department of Electrical and Computer Engineering at Northwestern University. His research interests are in the area of high-performance computing, parallel I/O, parallel file systems, data mining, and data management for large-scale scientific applications.