NORTHWESTERN UNIVERSITY

Scalable Parallelization Strategy for Large-Scale Deep Learning

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Engineering

By

Sunwoo Lee

EVANSTON, ILLINOIS

December 2020

ABSTRACT

Recently, a myriad of applications take advantage of deep learning methods to solve regression/classification problems. Although deep neural networks have shown powerful learning capability, many deep learning applications suffer from the extremely time-consuming training of the neural networks. In order to reduce the training time, researchers usually consider parallel training on distributed-memory systems. Synchronous Stochastic Gradient Descent (SGD) with data parallelism is the most popular parallelization strategy for neural network training, which guarantees the same convergence rate as the sequential training at the cost of having expensive inter-processing communications. Despite the poor scalability, many real-world deep learning applications usually adopt the synchronous parallel approach due to such a good convergence property.

In this thesis, we discuss how to improve the scalability of synchronous parallel training in several different aspects. First, we propose a parallel training algorithm that leverages the overlaps of communications and computations across model layers. Our overlapping strategy makes a large portion of the communication time hidden behind the backpropagation computation time, and thus the scaling efficiency is improved. Second, we re-design the gradient computation method in data parallel training. The proposed gradient computation algorithm not only reduces the communication cost but also enables to overlap the

communications with the forward computation at the next iteration. Finally, we propose an adaptive hyper-parameter adjustment method that improves the degree of parallelism while maintaining a good model accuracy. The proposed method gradually increases the batch size at run-time in order to make a good trade-off between the degree of parallelism and the generalization performance.

All these three research works address different performance issues in synchronous parallel training. Our performance evaluation results demonstrate that, by harmonizing these separate contributions, the synchronous parallel training can effectively scale up on High-Performance Computing (HPC) platforms achieving the same classification/regression performance as the sequential training.

Acknowledgements

I would like to thank my advisors, Prof. Alok Choudhary, Prof. Wei-keng Liao, and Prof. Ankit Agrawal for their continuous support and advise. I also thank to my lab-mates for making graduate study more successful and enjoyable: Qiao Kang, Kai-yuan Hou, Reda Al-Bahrani, Dipendra Jha, Arindam Paul, and Dr. Dianwei Han. I would like to thank my family as well for their endless support and encouragement. Especially my wife, Sunkyung Yoo, took care of our young son helping me fully focus on my study. I could finish my graduate study thanks to such a great dedication.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Recently, deep learning has become one of the most popular machine learning techniques. A myriad of applications have adopted deep learning methods to solve domain problems such as Computer Vision [**2, 3**], Natural Language Processing [**4, 5**], Social Media Mining [**6**], Robotics [**7**], and various scientific applications [**8, 9, 10**]. Thanks to its superior learning capability, deep neural networks have provided the state-of-the-art classification/regression performance in many domain applications.

Despite its powerful learning capability, deep learning applications usually suffer from the time-consuming training of the neural networks. Training a modern Convolutional Neural Network (CNN) usually takes hours or even days making it less practical. Researchers parallelize the training on distributed-memory systems to reduce the training time. Many popular deep learning software frameworks, such as TensorFlow [**11**], pyTorch [**12**], Caffe [**13**], and MXNet [**14**], also support parallel training. Considering the ever-increasing available data size in this 'Big Data' era, efficient and scalable parallel training is essential to build up powerful deep learning solutions exploiting such abundant data. Especially for large-scale domain applications, scalable deep learning methods will provide researchers with unprecedented opportunities to solve their problems.

However, achieving a good speedup of parallel training has been a major challenge in large-scale deep learning due to the following obstacles. First, the training algorithm suffers from the limited degree of parallelism. Neural networks are typically trained using mini-batch Stochastic Gradient Descent (SGD) algorithm. The algorithm repeatedly adjusts the model parameters using gradients computed from a random subset of training samples (called mini-batch) until the gradients become sufficiently small. Such an iterative algorithm causes a strong data dependency of the model parameters across the consecutive iterations. Each mini-batch can be processed only after the previous one is processed. Thus, the degree of parallelism is limited to the number of samples in a single mini-batch. Second, the frequent and expensive inter-process communications significantly degrade the scaling performance. Synchronous SGD with data parallelism is the most popular parallelization strategy for neural network training. This parallel training algorithm evenly distributes each mini-batch to all workers and averages the locally-computed gradients across all the workers by performing global inter-process communications. Since the number of gradients is the same as that of the overall model parameters, such expensive communications easily become the performance bottleneck.

Many researchers have put much effort into addressing these two performance issues. Asynchronous SGD proposed in [**15**] allows multiple workers to asynchronously update the shared model parameters making a trade-off between the convergence rate and the scaling efficiency. Many different optimization methods also have been proposed to improve the scalability by reducing the communication frequency [**16, 17, 18, 19**]. Some other researchers have studied how to directly reduce the communication cost by sparsification, quantization, and compression of the gradients [**20, 21, 22**]. Although all these works effectively reduce the communication cost, they commonly improve the scalability at the cost of having a loss in accuracy or requiring a larger number of training epochs.

Improving scalability of synchronous SGD has a huge impact on many deep learning applications. Especially, many large-scale scientific applications adopt synchronous SGD with data parallelism to scale up their deep learning solutions [**23, 24, 25, 26**]. When the batch size and learning rate are sufficiently small, synchronous parallel training guarantees the convergence of the training regardless of the number of workers. Many applications use synchronous parallel approach due to this stable convergence property [**27, 28**]. In addition, the synchronous parallel approach can be applied to many different optimization algorithms, such as AdaGrad [**29**], Adadelta [**30**], rmsprop [**31**], and Adam [**32**], because it does not have any dependency on the parameter update rule. Therefore, countless applications can directly benefit from a scalable synchronous parallel training strategy.

In this thesis, we propose three contributions towards improving scalability of synchronous SGD-based neural network training: 1) overlapping communications with computations in data parallel training, 2) communication-efficient gradient computation algorithm, and 3) adaptive hyper-parameter adjustment method that improves the degree of parallelism without losing model accuracy. All three research works tackle different performance issues in synchronous SGD training. By harmonizing these separate contributions, synchronous parallel training can effectively scale up on large-scale High-Performance Computing (HPC) platforms achieving a high model accuracy.

1.1. Overlapping Communications with Computations

The synchronous parallel training guarantees exactly the same parameter updates as the sequential training, and thus a good convergence rate. However, the parallel training suffers from the expensive communication cost for averaging the gradients among all workers. In data parallel training, as the number of processes increases, each process is assigned with a proportionally reduced number of training samples. In contrast to the linearly reduced computation workload, the communication cost increases and it results in making the speedup saturated.

Neural networks do not have data dependency on the gradients across the layers. The gradient communication at one layer and the gradient computations at other layers can be performed simultaneously. Motivated by this fact, we propose to overlap the communications with the computations during training to improve the scalability. By dedicating a small number of compute cores for communications, the communication time can be effectively hidden behind the computation time while almost not affecting the computation performance.

We present a data parallel training strategy that aggregates the gradients across all model layers into two subsets and averages them across all workers using asynchronous communications. Our approach minimizes the number of communications per iteration by having only two subsets of gradients. In addition, the one gradient communication is overlapped with the backward computation while the second gradient communication is overlapped with the model parameter updates. The effectiveness of the overlap depends on how many gradients are aggregated into each subset. We analyze the impact of the gradient aggregation on the exposed communication time and suggest a good practice for maximizing the degree of overlap.

In order to explicitly control the overlap, our approach employs a communication-dedicated POSIX thread per MPI process, which performs synchronous MPI communications. The main thread and the communication thread synchronize using POSIX conditional variable and mutex. By pinning the I/O thread on a single physical core, the parallel training can avoid the context switching overhead and the cold cache effects.

We evaluate the proposed overlapping strategy using ImageNet, a popular open benchmark dataset for image classification and VGG-16, a deep CNN model with 16 tunable layers. Our experimental results and analysis demonstrate the effectiveness of the proposed overlapping strategy and give an important insight about how to better utilize the compute cores not only for parallel computations but also for communications.

1.2. Communication-Efficient Parallel Gradient Computation

In data parallel neural network training, the inter-process communications for averaging gradients are the performance bottleneck. Given a mini-batch, each worker locally computes the gradients of a cost function with respect to the model parameters from the assigned subset of the mini-batch. Then, the local gradients are averaged across all the workers using inter-process communications. Typically, *allreduce* communications are used to aggregate and sum up the gradients. To the best of our knowledge, all the existing parallel deep learning frameworks such as Horovod and pyTorch use *allreduce* communications when averaging the gradients.

The *allreduce*-based data parallelism is under a strong assumption that the gradients are locally computed first and then averaged across all the workers. In this approach, every worker ends up having the same number of gradients regardless of the number of workers or the mini-batch size. Thus, as the number of processes increases, the overall communication cost increases and it results in becoming the performance bottleneck. We break such a prevalent assumption and re-design a parallel gradient computation algorithm which has a cheaper communication cost than the *allreduce*-based approach. We will analyze and compare the communication cost complexity between our proposed algorithm and the *allreduce*-based approach.

Our parallel gradient computation algorithm consists of several communication steps. There are two benefits from having multiple communication steps rather than a single *allreduce* operation. First, we can design a fine-grained communication overlapping strategy that overlaps the gradient communications with not only the backward computations but also the forward computations at the next iteration. Second, it allows to re-design the parameter update method which has a cheaper computation complexity. Instead of updating the model parameters after averaging all the gradients, our approach enables for each worker to update a part of model parameters locally and then aggregate the updated parameters across all the workers.

We evaluate the effectiveness of the proposed parallel gradient computation algorithm and the overlapping strategy using ImageNet dataset and ResNet50 [**33**], a deep residual network with more than 50 layers. Our study demonstrates that the scalability of synchronous SGD training can be significantly improved by breaking the traditional practice of using *allreduce* communications in data parallelism.

1.3. Adaptive Hyper-Parameter Adjustment Method

In data parallel training of neural networks, each mini-batch is evenly distributed to all workers and independently processed. Thus, the maximum number of workers is the number of training samples in each mini-batch. Intuitively, the larger the batch size, the better the degree of parallelism.

Unfortunately, the mini-batch size is typically tuned to a small value that provides a sufficiently high degree of noise in the gradients. For instance, when AlexNet [**2**] won the ImageNet competition by a large margin (15.3% vs 26.2% (second place) validation errors), their mini-batch size was 128 while the overall number of training images was about 1.2 millions. That batch size is lower than 0.0001% of the overall data samples. Assuming the samples for a mini-batch is uniformly drawn from the given dataset, the gradients can be considered as a random variable with mean of the optimal gradient computed from the entire dataset. It has been already theoretically explained that the larger the batch size, the lower the variance of the gradients. In other words, the gradients become less noisy as the batch size increases. When the stochastic gradients are closer to the optimal gradient, the model tends to rapidly converge into a local minimum on the parameter space achieving poor generalization performance.

We propose how to use a large batch size without significantly affecting the model accuracy. Our training strategy is to adaptively adjust the mini-batch size at run-time to make a good trade-off between the degree of parallelism and the model accuracy. We first analyze the impact of the batch size on the regression performance and the scaling performance. Then, based on the analysis, we design a practical metric for estimating the quality of the current model parameters in terms of the generalization performance. Finally, we discuss how to adjust the mini-batch size and learning rate using the proposed metric at run-time. Although the proposed training strategy uses different batch sizes during the training, all the workers always view the same model parameters at every iteration. Therefore, the training can be considered to be a special case of synchronous SGD, and thus the convergence is still guaranteed.

The proposed adaptive batch size training method is evaluated using two image regression applications, image super-resolution and image restoration. These two representative regression works commonly use a small batch size such as $16 \sim 64$ [**34, 35**]. We demonstrate that our adaptive batch size training method enables to increase the batch size to 256 effectively improving the degree of parallelism while maintaining the high regression accuracy. In order to show that the proposed training strategy is generally applicable to any optimizers, we also evaluate it using CIFAR10 image classification benchmark.

CHAPTER 2

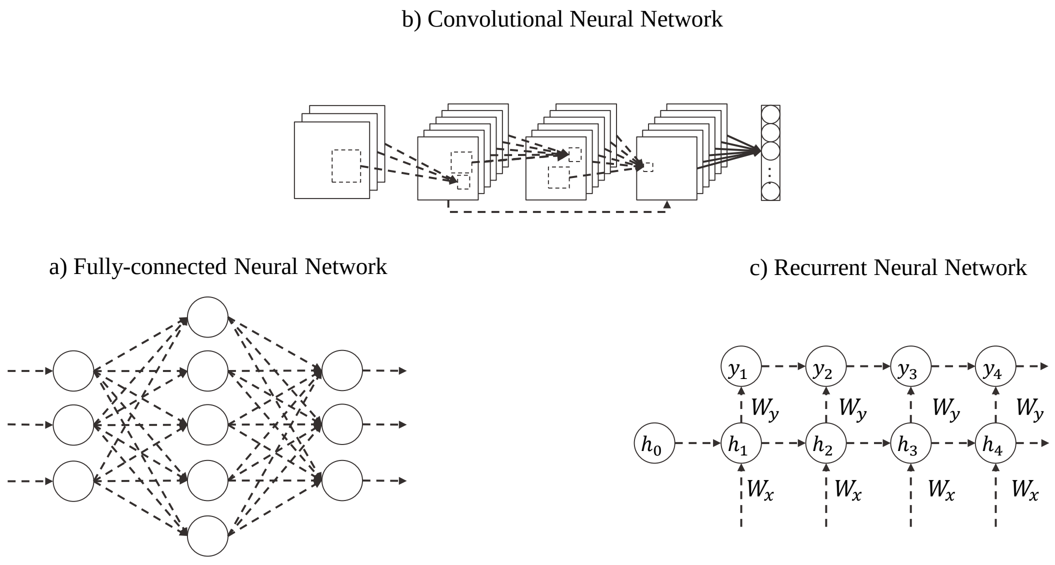
Background

2.1. Artificial Neural Network and Deep Learning

Artificial Neural Network (ANN) is a biologically-inspired algorithm. A network consists of multiple layers of ‘neuron’s. Each neuron is a perceptron with a various type of activation functions [**36**].

A network receives a set of input values at ‘input layer’. The data go through multiple layers which are called ‘hidden layer’s. Finally, the prediction values are generated at ‘output layer’. In supervised learning, the prediction values are compared with the expected correct values (usually called labels). The model is considered to be accurate when the prediction values are sufficiently close to the expected values.

Depending on the connection pattern of the neurons, there are several types of ANNs such as fully-connected network, Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN). Figure 2.1 presents the three representative types of neural network. Each type of network exploits different data characteristics. For example, CNNs have a special connection pattern which is called ‘local connection’. Each neuron receives data from a subset of neurons at the previous layer. This special connection pattern is designed based on a principle that there is a strong correlation among the neighbor neurons. Thanks to such the local connection pattern, CNNs show a superior learning capability for image handling problems. RNNs have also its own special connection pattern such



b) Convolutional Neural Network
a) Fully-connected Neural Network
c) Recurrent Neural Network
y_1
y_2
y_3
y_4
W_y
h_0
h_1
h_2
h_3
h_4
W_x

Figure 2.1. Three representative types of neural network. a) Fully-Connected Neural Network has a dense connection pattern such that all neurons at one layer is connected to every neuron at the next layer. b) Convolutional Neural Network (CNN) has a local connection pattern such that a subset of neurons at one layer is connected to each neuron at the next layer. c) Recurrent Neural Network (CNN) recursively train the same model parameters at multiple layers.

that the same model parameters are recursively used to process multiple input data. This connection pattern enables the network to learn the information across the time steps. Usually, researchers choose a type of neural network considering the problem-specific data characteristics.

Deep learning indicates a set of machine learning methods that are based on ANNs. Recently, the meaning of deep learning has been enlarged to a research field in which a variety of machine learning problems are studied using ANNs. A large portion of the deep learning applications deal with supervised learning problems in which the data consists of the training samples and its labels. There are deep learning methods for unsupervised learning or semi-supervised learning such as Generative Artificial Neural Network (GAN).

2.2. Training Algorithms

Training neural networks is a representative non-convex optimization problem. There are largely two different optimization methods for neural network training: the first-order optimization methods such as Gradient Descent (GD) [**37**] and the second-order optimization methods such as Newton's method [**38**] and Broyden-Fletcher-Goldfarb-Shanno (BFGS) [**39**]. In this thesis, we only consider the first-order optimization methods, the most popular and practical training algorithm for neural networks.

The most popular variant of GD for neural network training algorithm is Stochastic Gradient Descent (SGD) [**40, 37**]. Especially, the algorithm that approximates the gradients from a random subset of training samples is called 'mini-batch SGD'. Algorithm 1 presents the mini-batch SGD algorithm. In the algorithm, n, m, and l represent the number of training samples, the mini-batch size, and the number of layers in the network, respectively. The model parameters w is initialized with random values w_0 at line 1. There are many different initialization strategies such as Xavier method and MSRA method [**41**]. First, a random subset of training samples $\mathcal{B}$ is extracted from the dataset at line 4. Second, the gradients of the given cost function f is computed with respect to the model parameters w from $\mathcal{B}$ at line 5. Note that $\nabla f_{\mathcal{B}}(w_i) = \frac{1}{m}\sum_{j=0}^{m}\nabla f(w_i, x_j)$, where w_i is the model parameters at iteration i and x_j is the j^{th} training sample in the given mini-batch. Finally, the model parameters w is updated using the gradients at line 7. The algorithm repeats these three steps until the gradients become too small to

Algorithm 1 Mini-batch SGD (n: the number of training samples, m: the mini-batch size, f: the cost function)

```
1: $w \leftarrow w_0$
2: **while** stop condition is not met **do**
3:     **for** $i \leftarrow 1 \cdots \frac{n}{m}$ **do**
4:         $\mathcal{B} \leftarrow i^{th}$ mini-batch of size $m$.
5:         $\nabla f_{\mathcal{B}}(w_i) \leftarrow$ Compute_Gradient($f$, $\mathcal{B}$, $w_i$).
6:         $w_{i+1} = w_i - \mu \nabla f_{\mathcal{B}}(w_i)$.
```

effectively adjust the model parameters. In practice, researchers usually stop the training when an acceptable accuracy is achieved before the training loss actually converges.

Algorithm 1 has two user-tunable hyper-parameters, the mini-batch size m and the learning rate μ. The convergence rate of the training loss is strongly affected by these two hyper-parameters. Researchers typically tune these in a trial-and-error manner.

There are several variants of mini-batch SGD that are designed to achieve a faster convergence of training loss, such as AdaGrad [**29**], rmsprop [**31**], and Adam [**32**]. These algorithms commonly calculate the first-order gradients of the cost function first, and then update the parameters using a different update rule. Due to the different update rules, these algorithms have their own additional hyper-parameters such as momentum parameters or decaying parameters.

2.3. Parallelization Strategies

Researchers have put much effort into improving scalability of parallel neural network training. Depending on the degree of asynchrony in the local model parameters across the workers, the training algorithm can be categorized into ‘synchronous’ or ‘asynchronous’ algorithm. The training algorithms can also be categorized into ‘model parallelism’ or ‘data parallelism’ depending on how to distribute the computation workload on the workers.

2.3.1. Synchronous Training vs. Asynchronous Training

Synchronous parallel training is the most popular parallelization strategy for deep learning. It is called ‘synchronous’ training if all workers always view the same state of the model parameters. The most significant benefit from the synchronous parallel training is that the quality of the parameter updates is not affected by the number of workers. Many existing large-scale deep learning applications scale up the training in a synchronous way [**23, 24, 25, 26**].

However, the synchronous parallel training usually suffers from its poor scalability. In Algorithm 1, each iteration at line 3 is to process a single mini-batch. In order to make all the workers always use the state-of-the-art parameters when processing a mini-batch, the model parameters should be globally synchronized at every iteration. Depending on the implementation, either the gradients or the locally updated model parameters should be averaged among all the workers using inter-process communications. Such expensive and frequent communications result in becoming the performance bottleneck.

In order to address the issue of expensive communications, Dean et al. proposed asynchronous parallel algorithm [**15**]. Instead of synchronizing the model/gradient every iteration, the algorithm allows workers to use out-of-date model parameters to calculate the gradients. Asynchronous approach typically assumes the parameter-server communication model such that a centralized parameter server keeps the model parameters and receives the locally computed gradients from all individual workers asynchronously.

Asynchronous parallel training enjoys a good scaling performance since multiple mini-batches can be processed at the same time without any synchronizations. However, in order to guarantee the convergence, the number of workers should be sufficiently small so that the degree of asynchrony in the model parameters is bounded. The number of available workers strongly depends on the dataset size. The larger the number of training samples, the more the workers can asynchronously process different mini-batches at the same time without significantly affecting the convergence rate. Kurth et al. proposed a hybrid approach that makes a good trade-off between synchronous and asynchronous training strategies [**42**].

2.3.2. Communication Model

The parallel training can be implemented using two different communication models: fully-distributed communication model and client-server model. If the model parameters are replicated by all the processes and the gradients are globally averaged using inter-process communications, it is called 'fully-distributed' communication model. The most popular implementation of data parallelism based on the fully-distributed communication model is *allreduce*-based approach. If the global model parameters are managed by one or a small number of parameter servers, it is called 'client-server' model (also known as parameter server model). TensorFlow [**11**], one of the most popular deep learning software framework, supports data parallelism based on the parameter server approach. Asynchronous SGD assumes the parameter server communication model such that multiple workers asynchronously contribute to the shared model by sending the locally computed gradients to the parameter server.

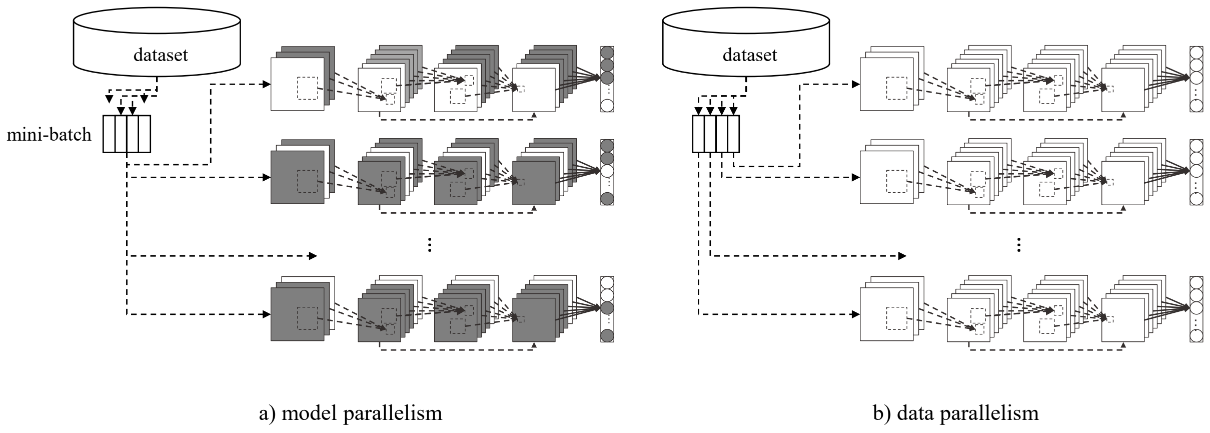


Figure 2.2. Two representative parallelization strategies for neural network training: a) model parallelism and b) data parallelism. In model parallelism, each worker trains on a distinct subset of the model parameters using the entire data. In data parallelism, each worker trains the entire model parameters using a subset of each mini-batch.

2.3.3. Model Parallelism vs. Data Parallelism

Depending on how to distribute the workloads to workers, the parallel training can be categorized into either 'model parallelism' or 'data parallelism'. If a model is split among multiple workers and trained on the same data, it is called model parallelism. In contrast, if the data is distributed to multiple workers and they train a single globally-shared model, it is called data parallelism. Figure 2.2 illustrates the model parallelism and data parallelism.

These two parallelization strategies have different communication patterns. First, data parallelism aggregates and sums up the entire gradients across all workers while model parallelism exchange the intermediate data such as activations and errors among subsets of the workers. Second, data parallelism and model parallelism have a different data dependency that affects the communication pattern. Data parallelism does not have data dependency on the gradient across the layers. So, the gradient communications can be performed at anytime within the iteration. In contrast, model parallelism has a strong data dependency on the activations and errors across any two consecutive layers. So, at every layer, the corresponding communications should be finished before going to the next layer.

Recently, data parallelism is more widely used in deep learning applications. Most of the large-scale deep learning applications scaled up on HPC platforms are based on data parallelism. Some researchers have proposed a hybrid approach [**43, 44, 45**].

2.3.4. Large-Batch Training

In data parallel training, the degree of parallelism is limited by the number of training samples in each mini-batch. Thus, increasing the batch size is an intuitive solution to improve the degree of parallelism. However, it has been empirically shown that, the larger the batch size, the poorer the generalization performance [**46, 47, 48, 49**]. The mini-batch size is usually tuned to a small value between $32 \sim 256$ in many applications.

Researchers have proposed several large batch training methods that aim to increase the batch size without losing the model accuracy. You et al. proposed Layer-wise Adaptive Rate Scheduling (LARS) that adjusts the learning rate considering the magnitude ratio of the gradients and the parameters [**47**]. Goyal et al. proposed 'linear scaling rule' that increases the batch size and learning rate together by the same factor [**50**]. Hoffer et al. proposed 'root scaling rule' that increases the learning rate by the root scale of the increased batch size [**48**]. Although these research works have derived different conclusions, there is a common principle behind of them: as the batch size increases, the learning rate should be also increased to maintain the model accuracy.

Keskar et al. discussed the concept of ‘flat’ and ‘sharp’ minima on the parameter space and explained the correlation between the batch size and the model sharpness [**46**]. As the batch size increases, the model tends to be attracted by sharp minima that poorly generalize to the test data. By using a small batch size, the gradients become sufficiently noisy to avoid falling into a sharp minimum and it ends up converging to a flat minimum. It is a common practice of neural network training that the batch size is fixed to a certain small value and the learning rate decreases as the training progresses. This training technique is usually called ‘learning rate decay’. Instead, Smith et al. proposed to increase the batch size as training progresses instead of decaying the learning rate [**51**]. The authors showed that increasing the batch size and reducing the learning rate make a similar impact on the degree of noise in gradients. Some researchers also proposed adaptive batch size training methods that aim to make a good trade-off between the degree of parallelism and the generalization performance [**52, 53**].

Although all these works enable to use a large batch size for training, there is a problem-dependent threshold of the batch size. If the batch size is increased further than the threshold, the model starts to lose the generalization performance. In other words, the model overfits to the training dataset, and thus provides a poor validation accuracy. For example, LARS enables to use a batch size of 16K for ImageNet classification problem without a significant loss in accuracy. However, when the batch size becomes larger than 16K, the validation accuracy significantly drops. A lot of existing works have showed a similar result [**52, 54, 53, 55**] with a variety of applications. Considering the ever-increasing available data size, it is crucial to break such a limitation to fully utilize the abundant data on large-scale HPC platforms.

Recently, local SGD training with periodic model averaging has been highlighted due to its less frequent synchronizations. There have been many theoretical and empirical studies of the algorithm [**56, 57, 58, 59**]. The algorithm allows to locally train multiple models in parallel and periodically averages the model parameters among all workers. While this approach effectively tackles the expensive communication cost issue, the degree of parallelism is still limited by the number of workers and the local batch size. If the effective batch size, the local batch size multiplied by the number of workers, is beyond a certain problem-dependent threshold, the convergence rate of the training loss is significantly reduced. In addition, the training convergence rate is strongly dependent on the number of workers. In order to guarantee a reasonably fast convergence, the number of workers should be sufficiently small similarly to the asynchronous SGD training.

CHAPTER 3

Overlapping Communications with Computations in Parallel Training

In data parallel neural network training, as the number of workers increases, the cost of gradient computation is linearly reduced while the communication cost increases. Consequently, as the nature of strong scaling, the speedup of parallel training is supposed to be saturated at a certain point. Especially in synchronous parallel training, the entire gradients of the model parameters are aggregated and summed up across all workers at every iteration causing extremely expensive inter-process communications. In order to achieve a good speedup, it is essential to reduce the cost of such expensive and frequent communications.

The gradients do not have data dependency across layers. Once the input data from the previous layer and the errors back-propagated from the next layer are provided, the gradients at the layer can be computed independently of any computations at other layers. Likewise, the gradient communication at one layer and the gradient computations at other layers can be performed simultaneously. By asynchronously posting the gradient communications, therefore, a large portion of communication time can be hidden behind the computation time, and thus a better scaling efficiency can be expected.

It is common practice that one communication is performed at one layer to average the gradients. For instance, Horovod, one of the most popular software framework for data parallel training, performs one *allreduce* operation at every layer. However, modern neural networks usually have decades layers or even more than a hundred layers. If the gradients are averaged one layer after another separately, such many communications per iteration will increase the latency cost of the communications.

In this chapter, we discuss how to organize the communications and effectively overlap them with the backward computations in data parallel training. We propose to aggregate the gradients across the model layers into two subsets and then average them across all the workers using only two communications per iteration. Due to the data dependency, all the communications posted at the current iteration should be finished before starting the next iteration. That is, the communications can only be overlapped with the backward computations at the current iteration. Give such a restriction, we design a communication strategy that maximizes the degree of overlap while minimizing the number of communications per iteration. In addition, we also study how to reduce the communication cost at the fully-connected layers by replicating the gradient computations. Compared to other types of layers, the fully-connected layers have an enormous number of gradients due to the dense connectivity. We propose a scalable gradient computation method that significantly reduces the communication cost while slightly increasing the computational cost.

Algorithm 2 presents the data parallel training algorithm that leverages the computation and communication overlap. The algorithm updates the model parameters using the globally averaged gradients. Therefore, it is guaranteed to produce the same level of accuracy as Algorithm 1. In the following sections, we will analyze the computation and communication patterns, as well as the data dependency of the parameters in Algorithm

2. Then, we validate our proposed method by comparing the communication cost between the typical *allreduce*-based approach and our proposed method.

We define a set of notations to describe data structure: D_o/D_i are the depth of output/input feature maps, R_o/R_i are the number of rows of the output/input neurons, C_o/C_i are the number of columns of the output/input neurons, R_f/C_f are the number of filter rows/filter columns, K_b/K_c are the number of neurons in the bottom/current layers, and N is the number of images.

3.1. Computation Workload and Data Layout

In this research work, we focus on CNNs. Many CNN models contain both convolution layers and fully-connected layers. We discuss and analyze the computation workload for these two representative connection patterns.

At convolution layers, we use *im2col* technique [**60**] to rearrange the input data so that the computation pattern is changed from convolution to matrix multiplication. At fully-connected layers, the computation pattern is also a matrix multiplication. Therefore, the overall computation workload is a set of data transformations and matrix multiplications. This unified computation pattern allows an efficient implementation in practice.

im2col and *col2im* are well-known data transformation techniques used in open source frameworks such as Caffe [**13**] and Torch [**61**]. Given an input data and a filter size, *im2col* rearranges filter-sized blocks of the input data into columns and concatenate them into a 2-dimensional matrix. The *col2im* transforms columns to blocks of the original data layout. We customize these functions to transform the input activations from multiple images

Algorithm 2 Data parallel training
(M: number of mini-batches, N: size of mini-batch, L: number of layers, k: number of layers for the first gradient chunk, f: number of fully-connected layers that replicate the gradient calculation)

```
1: Define $s \leftarrow$ the layer ID of the first fully-connected layer.
2: for each worker $p \in \{0, ...P-1\}$ parallel do
3:     for each mini batch $m = 0, ...M-1$ do
4:         Get the sub-mini batch $D_p^m \leftarrow \frac{N}{P}$ images from $D^m$, the $m^{th}$ mini batch.
5:         Initialize the local gradient sum, $G_p^l = 0$
6:         for each layer $l = 0, ...L-1$ do                                    ▷ Feed-Forward
7:             if $l \in \{s, ..., s+f\}$ and $m \neq 0$ then
8:                 Wait for the communications on $A_p^{l-1}$ and $E_p^l$, posted in iteration $m-1$,
   to be finished (line 12 and 25).
9:                 Calculate weight gradients $\Delta W_p^{[l:l+f]}$ for $D^m$
10:                Update the corresponding weights $W_p^{[l:l+f]}$.
11:            Calculate activations $A_p^l$ using the given sub mini-batch $D_p^m$.
12:            if $l \in \{s-1, ..., s+f-1\}$ then
13:                Post asynchronous communication: Allgather $A_p^l$.
14:        for each layer $l = L-1, ...0$ do                                    ▷ Backpropagation
15:            Calculate errors $E_p^l$.
16:            if $l \notin \{s, ..., s+f\}$ then
17:                Calculate weight gradients $\Delta W_p^l$ for $D_p^m$
18:                Add the weight gradiets to the local gradient sum: $G_p^l += \Delta W_p^l$.
19:                if $l$ is equal to $k$ then
20:                    Post asynchronous communication: Allreduce $G_p^{[L-k:L-1]}$.
21:        Post asynchronous communication: AllReduce $G_p^{[0:L-k-1]}$.
22:        Wait for the communication on $G_p^{[L-k:L-1]}$ to be finished (line 19).
23:        for each layer $l = L-1, ...0$ do                                    ▷ Parameter Update
24:            if $l$ is equal to $L-k$ then
25:                Wait for the communication on $G_p^{[0:L-k-1]}$ to be finished (line 20).
26:                for each layer $l = s, ...s+f$ do
27:                    Post asynchronous communication: Allgather $E_p^l$.
28:            if $l \notin \{s, ..., s+f\}$ then
29:                Update parameter, $W_p^l$.
```

into a single large matrix. Figure 3.1 illustrates the workload of the first convolution layer in feed-forward stage.

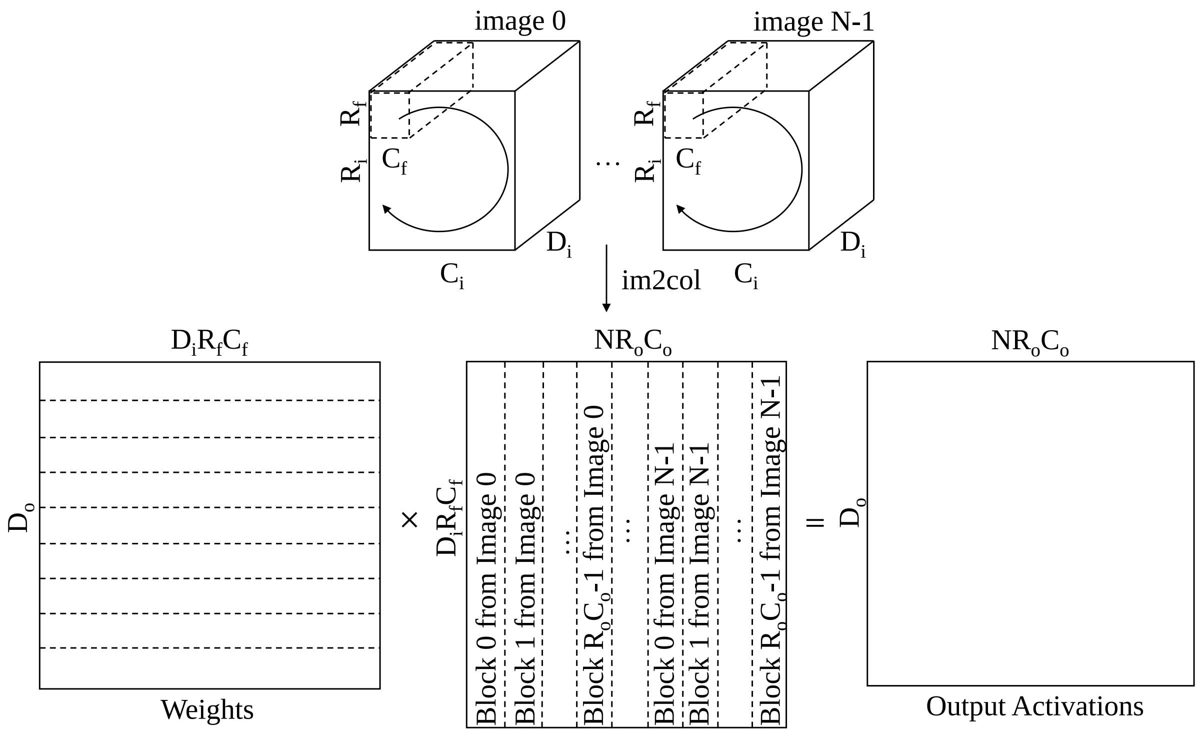


Figure 3.1. Computing activations using *im2col* at the first fully-connected layer. N training samples are transformed to a single matrix and the weight matrix is multiplied by the transformed matrix. The result is the output activation matrix.

In feed-forward, the first convolution layer receives the input images that are stored in row-major order. Each mini-batch of N images is organized into a matrix of size $N \times D_iR_iC_i$, which is then transformed by *im2col* into a $D_iR_fC_f \times NR_oC_o$ matrix. Then, the matrix is multiplied by the weight matrix of $D_o \times D_iR_fC_f$ and added by the bias vector of D_o. The output activation matrix of $D_o \times NR_oC_o$ is computed by Equation 3.1 and 3.2.

$$C^{l-1} = im2col(A^{l-1}) \tag{3.1}$$

$$A^l = \sigma(W^l C^{l-1} + B^l), \tag{3.2}$$

where σ is an activation function, A^l is the activation matrix calculated at layer l, B is the bias vector and C is the matrix generated by *im2col*. From the second convolution layer, the input from the previous layer is an activation matrix of size $D_i \times NR_iC_i$. We make *im2col* support both data layouts, $N \times D_iR_iC_i$ and $D_i \times NR_iC_i$, so that it can be used in any convolution layers. In Algorithm 2, these computations are performed at line 10.

In backpropagation, a convolutional layer receives a $D_i \times NR_iC_i$ error matrix from the previous layer. Instead of transforming the error matrix using *im2col*, we multiply that by the weight matrix first and then transform the result into the output error matrix using *col2im*. This approach avoids an extra layout transformation of weight matrix [**60**]. Note that the error matrix E and activation matrix A have the same data layout. E is computed by Equation 3.3 and 3.4. In Algorithm 2, these computations are performed at line 14.

$$C^l = W^{l+1}E^{l+1} \tag{3.3}$$

$$E^l = \mathit{col2im}(C^l) \tag{3.4}$$

In addition to the errors, the gradients of parameters are calculated in backpropagation. When calculating the gradients at layer l, the activation matrix at layer $l-1$ is transformed to a $D_oR_fC_f \times NR_iC_i$ matrix using *im2col*. Then, the $D_i \times NR_iC_i$ error matrix of layer l is multiplied by the new activation matrix. Due to the order of dimensions, the error matrix should be transposed. The result is $D_i \times D_o R_f C_f$ gradient matrix of layer l. The computations are shown in Equation 3.1 and 3.5, and they are performed at line 16 of Algorithm 2. The *im2col* performs the same transformations in feed-forward and backpropagation. Therefore, if memory space is sufficiently large, the *im2col* in backpropagation can be avoided by saving the matrix C^{l-1} calculated in the feed-forward stage.

$$\nabla W^l = C^{l-1} E^l \tag{3.5}$$

The proposed computation using *im2col* and *col2im* is independent from both model architecture and mini-batch size. Any convolution layer can be trained with only three layout transformations and three matrix multiplications using Equation 3.1, 3.2, 3.3, 3.4, and 3.5.

For the fully-connected layers, the computations can be described by Equation 3.2, 3.3, and 3.5 if the matrix C is replaced with its original matrices A^{l-1}, E^l, and A^{l-1}, respectively. Each fully-connected layer has a $K_b \times K_c$ weight matrix. When multiplying this weight matrix by the input activation matrix, to make the memory accesses contiguous, all the activations from each image should be stored in a contiguous memory space. So, in the first fully-connected layer, we transform the input activation matrix into a $N \times D_i R_i C_i$ matrix. Note that K_b is equal to $D_i R_i C_i$ at fully-connected layers. In the backpropagation stage, to make the memory accesses contiguous when multiplying the weight matrix and the input error matrix, we transform the input errors into a $D_o \times N R_o C_o$ matrix in the first fully-connected layer.

3.2. Inter-process Communications

As our parallel CNN training algorithm adopts the data parallelism, each mini-batch is partitioned among multiple computing nodes and an individual model is trained on the assigned subset of the mini-batch in each node. The gradients should be aggregated across all the nodes and averaged to update the weights and biases. The gradients from different images are summed up within each node first. Then, the gradient sums are aggregated across all the nodes. The number of weight gradients in each node, S_w, is calculated by Equation 3.6.

$$S_w = \sum_{i=0}^{L_c-1} D_o^i D_i^i R_f^i C_f^i + \sum_{i=0}^{L_f-1} K_b^i K_c^i, \tag{3.6}$$

where L_c and L_f are the number of convolution layers and fully-connected layers. Likewise, the number of bias gradients in each node, S_b, is calculated by Equation 3.7.

$$S_b = \sum_{i=0}^{L_c-1} D_o^i + \sum_{i=0}^{L_f-1} K_c^i \tag{3.7}$$

The communication pattern in data parallel training is defined such that each node has $S_w + S_b$ gradient sums, and they are aggregated across all nodes by a reduction sum.

Data Dependency Analysis – In the feed-forward stage, the activations are propagated from the first layer to the last layer and data dependency exists between any two consecutive layers. Likewise, in backpropagation stage, the errors are propagated from the last layer to the first layer and data dependency exists between any two consecutive layers. The gradients are also calculated in backpropagation using the activations of the next layer and the errors of the current layer. Thus, the gradients are dependent on the

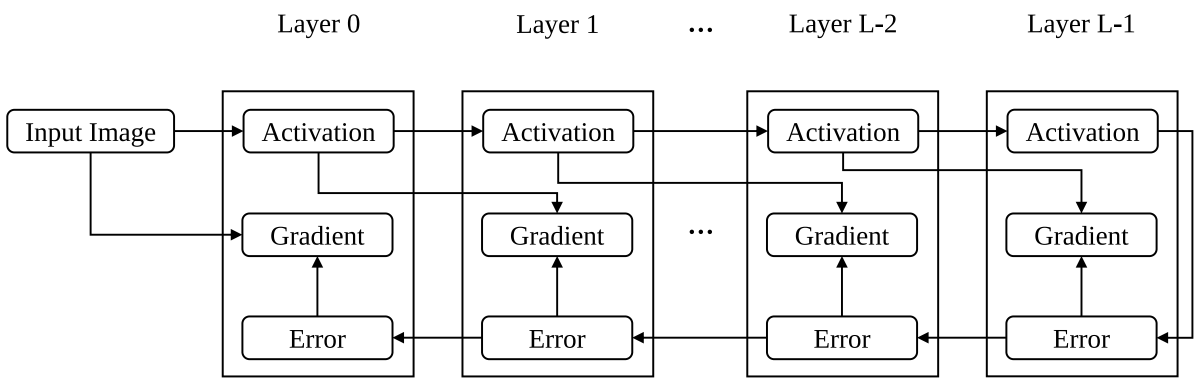


Figure 3.2. Data dependency in a neural network. The activations are propagated from left to right in feed-forward stage, and the errors are propagated from right to left in backpropagation stage. The gradients are computed using the activations and errors. Each arrow indicates the data dependency.

activations and errors. In contrast, the gradients in different layers are independent of each other. Figure 3.2 illustrates the data dependencies in a neural network. Each arrow in the figure indicates the data dependency.

The weights and biases of each layer should be updated before it reaches the layer in the feed-forward stage for the next mini-batch. Since the gradients in different layers are independent of each other, the parameter updates in different layers are also independent of each other. In other words, the parameters can be updated out-of-order across the layers. In the following section, we present our overlapping strategy based on these data dependencies.

3.3. Overlapping Strategy and Implementation

Algorithm 2 has a few communications that aggregate the gradients across all nodes. They are overlapped with computation such that the communication time is hidden behind the computation time as much as possible. We first propose two methods to maximize the overlap, and then explain the communications in Algorithm 2 in detail.

3.3.1. Overlap of Communication for Gradients and Computation

To overlap communication with computation as much as possible, two factors should be taken into account: number of communications and data size for each communication. First, the number of communication should be minimized to reduce the overall communication cost.

Each communication consists of two times, T_s and T_m, startup time and transfer time. Every communication has T_s regardless of the data size. So, the overall communication time can be reduced by having less communications. Second, in order to maximize the overlap, the communications should aggregate as many gradients as possible before the backpropagation is finished. Since the last communication cannot be overlapped with the backpropagation, the data size for the last communication should be minimized.

Considering these two factors, we gather the entire gradients across all nodes with two communications. The first communication is started after the backpropagation at the first few layers is finished. Then, the second communication is started after the entire backpropagation is ended. This approach enables to overlap much of the communication time with computation time while the number of communications is considerably reduced. In the later sections, we will call the gradients for each communication gradient chunk.

Unfortunately, the optimal size of gradient chunks cannot be known in advance. The optimal data size varies depending on many factors such as computing power, network speed, and model architecture. We define a new hyper-parameter, k, the number of layers for the first communication. The value of k should be tuned heuristically such that the first communication is overlapped with the backpropagation as much as possible. We will study the impact of various k values on the scalability in Section 3.4.

3.3.2. Communications in Data Parallel Training

Algorithm 2 has $2f + 2$ communications at each iteration. In feed-forward, f asynchronous communications are posted to gather the activations of the first f fully-connected layer across all nodes at line 12. In backpropagation, once the gradients of k layers are computed, they are summed across the local images first, and then an asynchronous communication is posted to aggregate the gradient sums across all the nodes at line 19. When the backpropagation is finished, another asynchronous communication is posted again to aggregate the last of the gradients of the model at line 20. Finally, f asynchronous communications are posted to gather the errors of the first f fully-connected layers across all nodes at line 26. Algorithm 2 has three blocking points: line 8, 21, and 24. Before updating parameters, it should wait until the corresponding gradients are gathered across all the nodes. Note that the parameter update for f fully-connected layers is delayed to the next mini-batch training.

Figure 3.3 presents an example time-flow chart of Algorithm 2. *GatherA* and *GatherE* are f communications for gathering activations and errors respectively. *ReduceW0* and *ReduceW1* are the reductions for gradient chunks. Ideally, if each communication time is shorter than the corresponding computation time, the entire communication can be hidden behind the computation. It is worth noting that the gradient computation and parameter update at the first f fully-connected layers are delayed to the next mini-batch

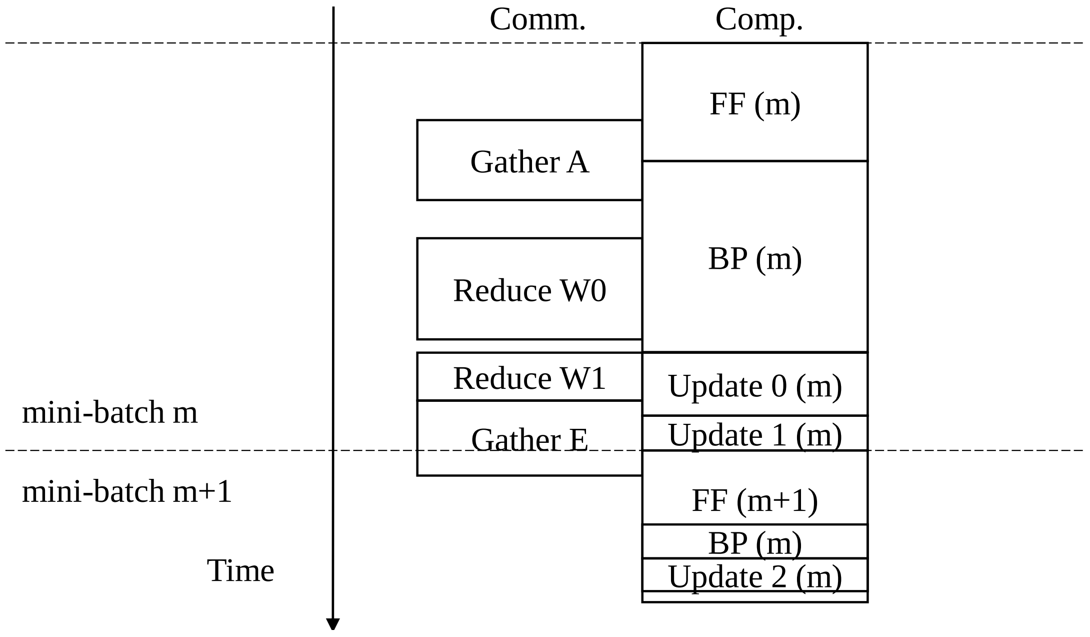


Figure 3.3. Time-flow chart with the maximized overlap. This figure illustrates the ideal case on which all communications are hidden behind the computation. In this case, a linear speedup can be expected.

training. It allows to overlap *GatherE* with the feed-forward computation for the next mini-batch.

3.3.3. Replicated Gradient Calculation in Fully-Connected Layers

The convolution layers are computationally more expensive than the fully-connected layers. In majority of CNN models, a convolution layer has much more neurons than a fully-connected layer. In contrast, the fully-connected layers cause heavier communications than the convolution layers. The fully-connected layers have full connections while the convolution layers have local connections as explained in Section 2.1. Since each connection has the corresponding weight parameter, a fully-connected layer typically has more weights than a convolution layer. For example, VGG-A [**62**] model has three fully-connected layers and each of them has about 103 millions, 17 millions, and 4 millions of weights respectively whereas the convolution layers have 5 millions of weights in total. In data-parallelism, since the gradient is calculated with respect to the parameters, the communication for such a huge number of gradients can be a performance bottleneck.

We replicate the gradient calculation in fully-connected layers to reduce the communication cost. First, the local activations and errors calculated using the assigned subset of mini-batch are gathered across all the nodes. The data size of the communication is $(K_b^l + K_c^l)N$. Then, the gradients for all images in the mini-batch are computed in every node. Note that if the gradients are calculated in parallel, the data size of the communication is $K_b^l K_c^l$. In modern CNNs, most likely, $K_b^l K_c^l$ is much larger than $(K_b^l + K_c^l)N$ at fully-connected layers. Thus, the communication cost can be dramatically reduced by replicating the gradient calculation. Since every node calculates the gradients for all the image, it takes a constant time regardless how many nodes it runs on. However, the computation time rather effectively overlaps the communication time for other gradients and does not adversely affect the scaling performance.

In VGG-A model, for example, K_b^l is 25,088 and K_c^l is 4,096 in the first fully-connected layer. Saying N is 256 which is the most popular mini-batch size in large-scale visual recognition tasks, the values of $K_b^l K_c^l$ and $(K_b^l + K_c^l)N$ are 102,760,448 and 7,471,104. If the gradient calculation is replicated in the first fully-connected layer, assuming the data is 4-byte single-precision floating point numbers, the reduction of 392MB is replaced with the gathering of 28.5MB.

We define another hyper-parameter, f, the number of fully-connected layers that replicate the gradient calculation. Selecting the optimal value of f is also crucial to maximize the overlap. If the gradient calculation is replicated in all the fully-connected layers, the early backpropagation time does not overlap any communication time. Furthermore, the replicated computation can take so much time that the speedup is lowered. Thus, f should be heuristically tuned to allow both a large overlap and the reasonable amount of constant computation time. In Section 3.4, we demonstrate the impact of various values of f on the scalability.

3.3.4. Communication-Dedicated POSIX Thread

In order to explicitly control the overlap of communication and computation, we employ a POISX thread which performs blocking MPI communications in background. Each MPI process creates a POSIX thread at the beginning of the training. When the gradients have been computed, the main thread sends a communication request to the communication thread using a shared message queue. The queue is protected using a POSIX mutex. After the main thread queues its request, it broadcasts a software signal, and the communication thread wakes up catching the signal using a POSIX conditional variable. When there is no requests in the shared queue, the communication thread goes to sleep. This software mechanism enables the communication thread to rapidly serve the requested communications in background.

We pin the communication thread on a single physical core so that the communications can be performed without being affected by the context switching or the cold cache effect. On a modern CPU-based system, the best achievable scaling efficiency is much lower than the ideal 100% performance [**63**]. Even when one core is dedicated for the communication thread, we have not observed any computation performance degradation. In addition, it is a common practice that only a single process runs on each node for parallel neural network training on distributed-memory systems due to the large memory footprint of each process. So, it is reasonable to assume that only a single communication-dedicated thread will run on each node occupying a physical core.

3.4. Performance Evaluation

We implemented a deep learning software framework in C language. The software framework has been designed for parallel training of neural networks on distributed-memory platforms. The data parallelism is implemented using MPI for all the inter-process communications and non-kernel loops are parallelized using OpenMP. The kernel functions such as matrix operations are implemented using Intel MKL library. All experiments are performed on Cori Phase I, a Cray XC40 supercomputer at the National Energy Research Scientific Computing Center. Each compute node has two sockets and each socket contains a 16-core Intel Haswell processor at 2.3GHz. The system has Cray Aries high speed interconnections with 'dragonfly' topology.

The most representative dataset for visual recognition tasks is ImageNet [**64**]. It contains 1.2 million 3-channel (RGB) images of various sizes. The classification on this dataset is considered to be extremely challenging not only because the images are high-resolution real-world pictures, but also because the training on such a large dataset takes an enormous execution time. We use the preprocessed ImageNet dataset that has $3 \times 224 \times 224$ pixels in each image.

We use VGG-A model [**62**], a deep CNN with 16 layers and 133 millions of parameters. Additionally, we built three variants of VGG-A, VGG-128, VGG-256, and VGG-512. These models have 128, 256, and 512 feature maps in convolution layers and contain 48 millions, 76 millions, and 140 millions of parameters respectively. Since the same workload is repeated across the mini-batches, we measure the execution time to process a single mini-batch 10 times and average the timings.

3.4.1. Single-Node Performance Study

We compared the performance of our implementation with Caffe, a popular open-source framework for neural network training. We compiled the main branch source code [**13**] with Intel MKL library. Figure 3.4 presents the single-threaded and multi-threaded performances with different mini-batch sizes. We observe that our implementation provides a shorter execution time than Caffe for all the mini-batch sizes. First, the performance gain of the single threaded training comes from the computation pattern described in Section 3.1. We perform only three matrix multiplications and three layout transformations for each layer, while Caffe performs $3N$ matrix multiplications and $3N$ layout transformations. Given the same amount of workload, fewer matrix operations likely provide better performance due to the reduced function call overhead. Second, in the multi-core training, we have approximately 10% of additional performance gain by parallelizing *im2col* and *col2im*. This experimental result demonstrates that the multi-node performance study following this section is based on the reasonable level of single-node performance.

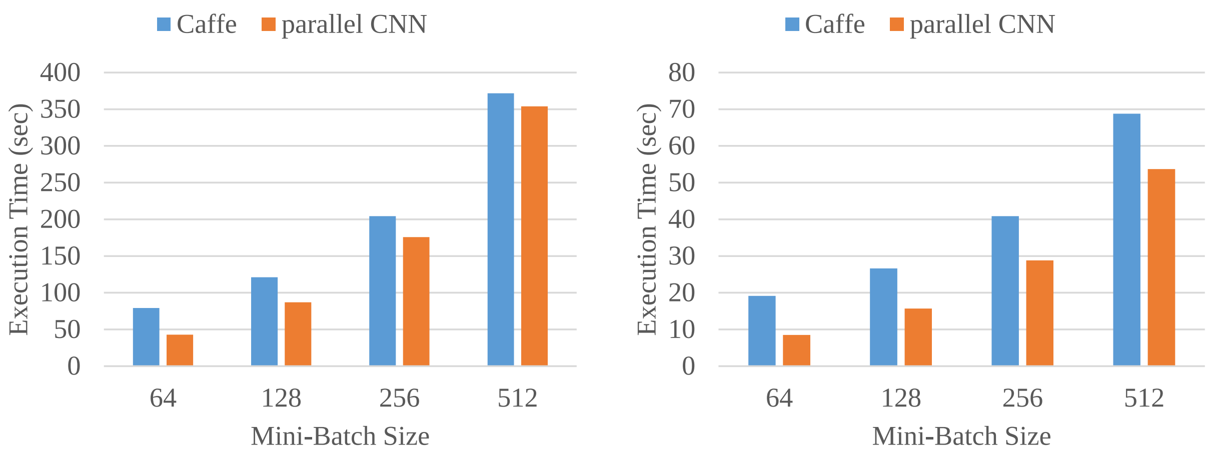


Figure 3.4. VGG-A training time on a single node: sequential performance (left) and multi-threaded (32 cores) performance (right). Caffe is an open-source framework and parallel CNN is our implementation. The experiments are performed with varying mini-batch size.

3.4.2. Multi-Node Performance Study

In this section, we present the scaling performance of our implementation with various software settings and analyze the experimental results. All the speedup charts are the strong-scaling results.

End-to-end training time and speedup – We compare our parallelization strategy to the pure data-parallel training of CNN (DP). Three hyper-parameters are set in advance: N (mini-batch size) is set to 256, k (number of layers for the first gradient chunk) is set to 9, and f (number of layers that replicate gradient calculation) is set to 2. Figure 3.5 presents the execution time of one iteration and its speedup. The speedups are calculated with respect to the number of nodes, based on the single-node performance

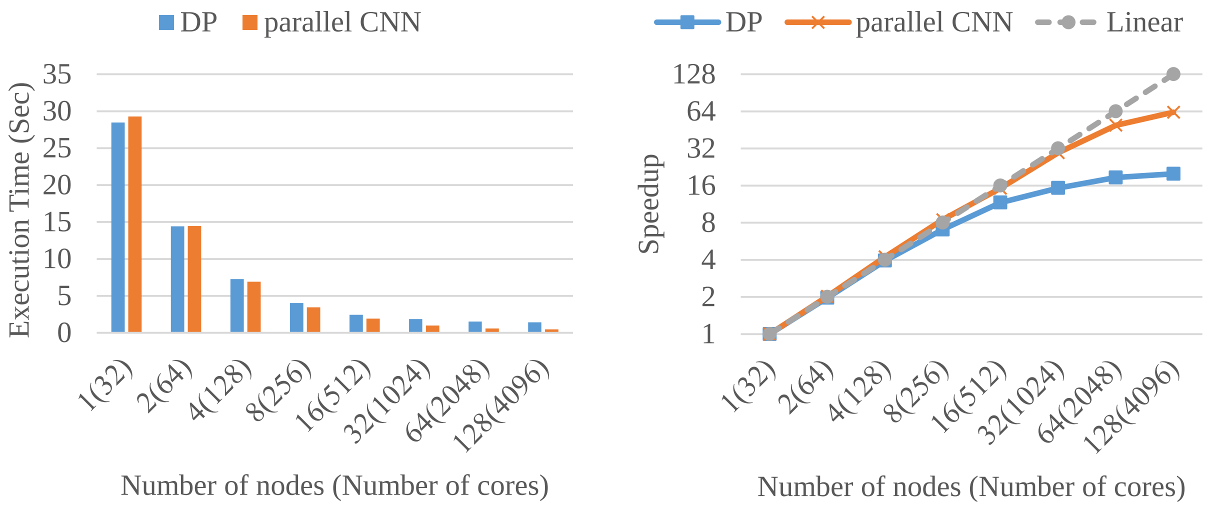


Figure 3.5. VGG-A training time (left) and speedup (right) for a single mini-batch size of 256). k is set to 9 and f is set to 2. We compare our approach (parallel CNN) with the pure data-parallel training algorithm (DP). The speedup is calculated with respect to the number of nodes.

presented in the previous section. We see that our parallelization strategy shows a significantly improves the scaling efficiency. 'parallel CNN' achieves a speedup of 62.97 on 128 nodes, while DP achieves a speedup of 19.92.

Timing breakdown – If a gradient communication is not finished before updating the corresponding parameters, the update will be blocked until the communication is finished. We define this blocking time as 'measurable communication time'. To evaluate the proposed overlapping strategy, we compared the measurable communication time as well as the actual communication time. Figure 3.6 presents the timing breakdown (y-axis is in log scale) for training VGG-A on a single mini-batch. We see the clear gap between the communication time and the measurable communication time. This gap implies that our proposed methods effectively overlaps the communications with the computations. The measurable communication time appears from 8 nodes since the computation time

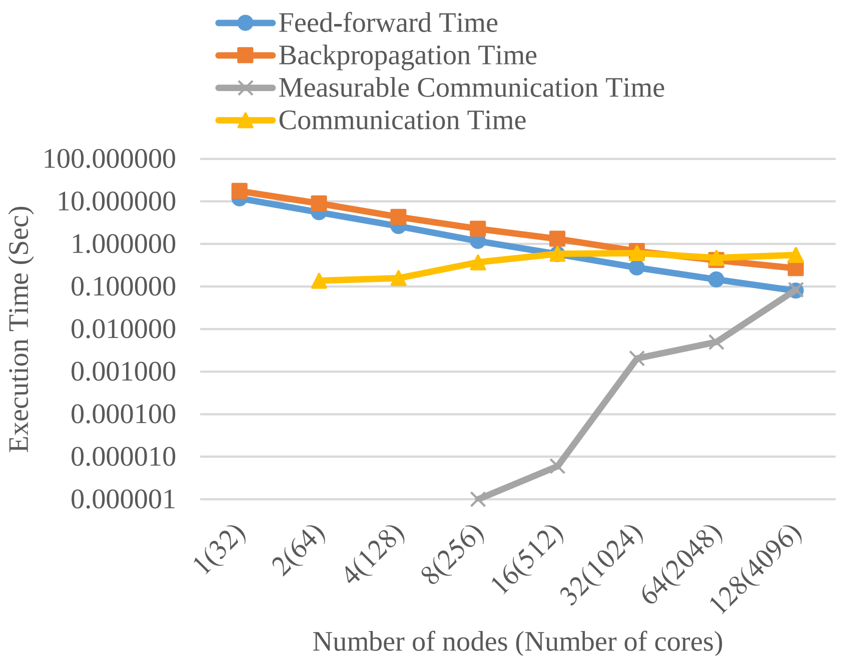


Figure 3.6. Timing breakdown for VGG-A training (mini-batch size of 256). The communication time is the accumulated time for all the inter-process communications. Measurable communication time is a part of the communication time which is not overlapped with any computation time.

becomes so short that it does not hide the entire communication time. As the training scales up beyond 8 nodes, the measurable communication time increases and it ends up becoming almost the same as the backpropagation time.

3.4.3. Computation and communication overlapping

– We investigate the scalability of the proposed algorithm with various model architectures and hyper-parameter settings. There are four hyper-parameters that affect the performance: mini-batch size, number of parameters, number of layers for each gradient chunk, and number of fully-connected layers that replicate the gradient calculation. With various settings of these hyper-parameters, we evaluate the proposed overlapping strategy and discuss the impact on the scalability. We define overlapping ratio, a metric for analyzing how much communication is overlapped with the computation. The ratio R is calculated by Equation 3.8.

$$R = \frac{100 \times \sum_{i=0}^{2f+1}(T_c^i - T_b^i)}{\sum_{i=0}^{2f+1} T_c^i}, \tag{3.8}$$

where T_b^i is the measurable communication time and T_c^i is the actual communication time for the i_{th} communication.

Scalability with respect to mini-batch size – To replicate the gradient calculation in f fully-connected layers, the activations and errors are gathered across all the nodes. The communication cost depends on mini-batch size since the number of the activations and errors in each layer is $(K_b^l + K_c^l)\frac{N}{P}$. Figure 3.7 shows the overlapping ratio and the speedup for VGG-A training with varying size of mini-batch-128, 256 and 512. As shown, the larger mini-batch size allows higher overlapping ratio and it results in achieving higher speedup. The maximum speedups are 17.82, 62.97, and 77.97, respectively.

We observe that the speedup curve drops suddenly on a certain number of nodes. The reason is that the overlapping ratio sharply drops if the computation time becomes less than the communication time, and the increased measurable communication time lowers the speedup. In data parallelism, the larger mini-batch size gives more computation workload while it does not affect the communication cost. Thus, we can expect better scalability with a larger mini-batch size.

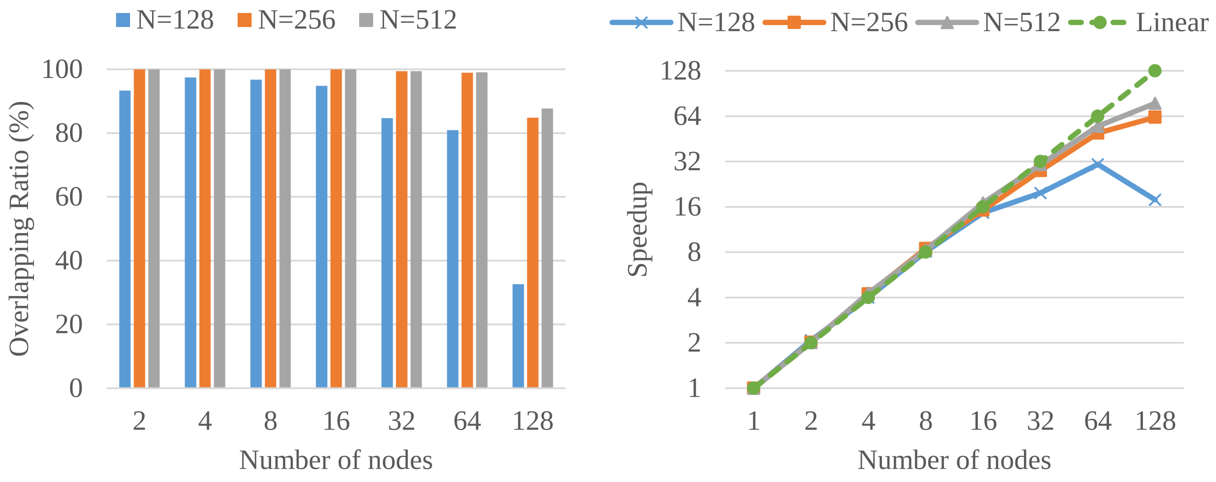


Figure 3.7. Overlapping ratio (left) and speedup (right) with varying mini-batch size. k is set to 9 and f is set to 2. The larger mini-batch size increases the computation workload and allows the higher overlapping ratio.

Scalability with respect to number of parameters – The number of parameters affects both the computation time and the communication time. To compare the scaling performance across the sizes of model, we measured the performance of training VGG-128, VGG-256 and VGG-512. We set the hyper-parameters: N to 256, k to 9, and f to 2. Figure 3.8 presents the overlapping efficiencies and the speedups. We see that the model with more parameters achieves higher speedup. The computation complexity of the training algorithm is $(\frac{C}{P} + F)NK^2)$ whereas the communication cost is directly proportional to the number of parameters. If the number of parameters is increased, due to the N term that is independent of the number of parameters, the computation cost is more increased than the communication cost and it allows higher overlapping ratio.

Replicating the gradient calculation in fully-connected layers – To evaluate the impact of replicating the gradient calculation on the scalability, we measured the overlapping raito and speedup with varying value of f. Figure 3.9 shows the performance

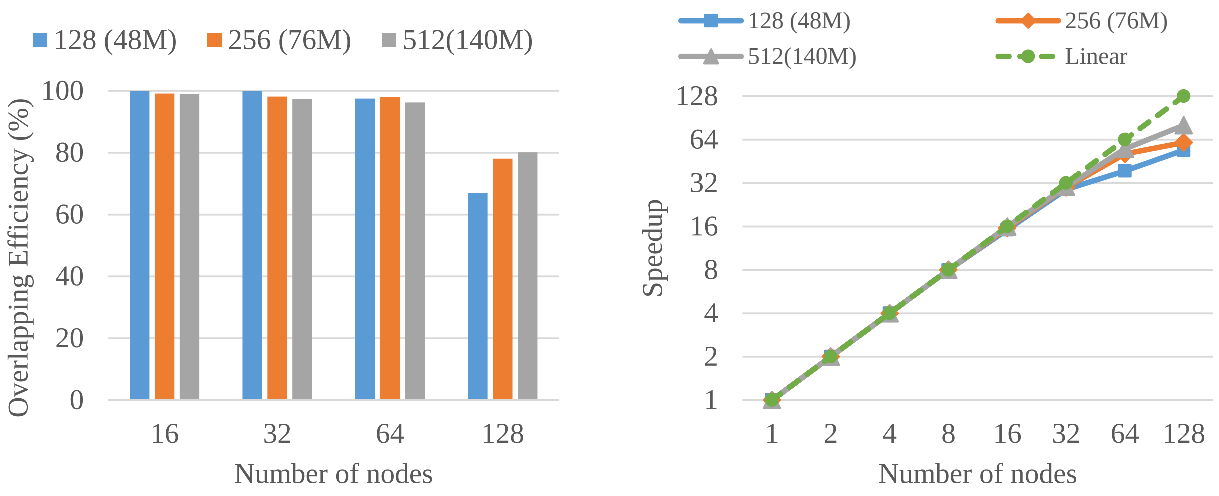


Figure 3.8. Overlapping ratio (left) and speedup (right) with varying number of parameters. VGG-128, VGG-256, and VGG-512 models are trained on a single mini-batch size of 256. The maximum speedups are 54, 61, and 80, respectively.

results. If all the fully-connected layers replicate the gradient calculation (f is 3), due to the large number of activations and errors to be gathered across all the nodes, the measurable communication time can be rather increased while the early backpropagation time does not overlap any communication time. In contrast, if none of the fully-connected layers replicate the gradient calculation (f is 0), the large gradient chunks engender the expensive communications. In our experimental environment, we achieved the best speedup when f is set to 2.

Number of layers covered by each communication – We measured the speedup with varying value of k-3, 6, 9, and 12. The values are selected for dividing the entire gradients into two chunks based on pooling layers. We skipped the case where k was set to 1 since it allowed almost no overlap and showed a similar speedup with DP. Figure 3.10 presents the overlapping ratio and speedup. When k is set to 3 or 6, due to the small

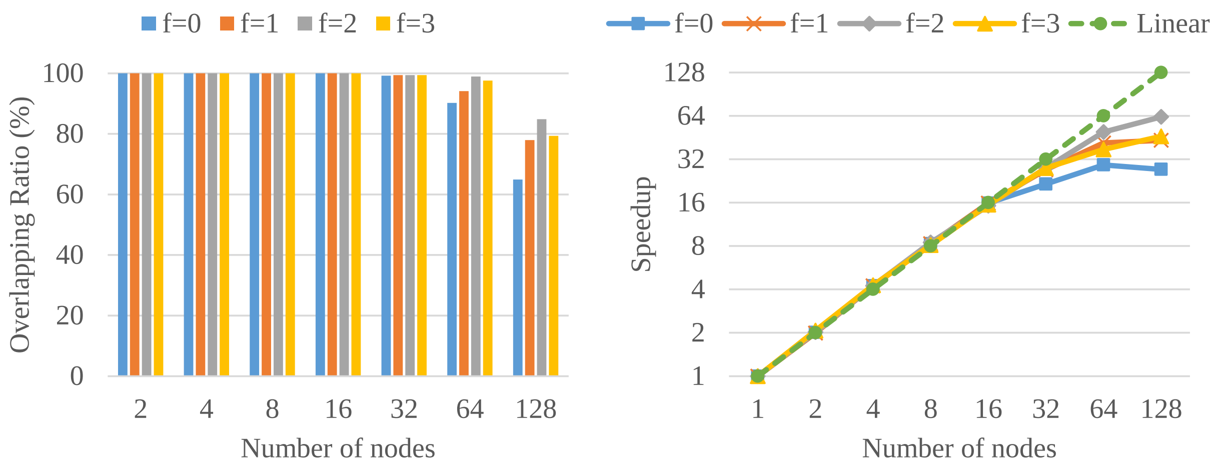


Figure 3.9. Overlapping ratio (left) and speedup (right) with varying number of fully-connected layers that replicate the gradient calculation. Replicating the gradient calculation at all the fully-connected layers can drop the speedup.

size of the first gradient chunk, most of the backpropagation time does not overlap any communication time and it gives a higher chance to have a longer measurable communication time for the second gradient chunk reduction. In contrast, if k is set to 12, the first gradient chunk is so large that the communication is not fully overlapped with the backpropagation. We achieve the best overlapping ratio when k is set to 9.

3.4.4. Comparison with Previous Works

In this section, we compare our approach with the existing works. Table 3.1 summarizes the previous works.

Comparison with parallel algorithms on GPUs – GPUs have been popularly used to speedup the computing-intensive workload of neural network training. Many of the large-scale deep learning applications on GPUs are based on master-slave model. The

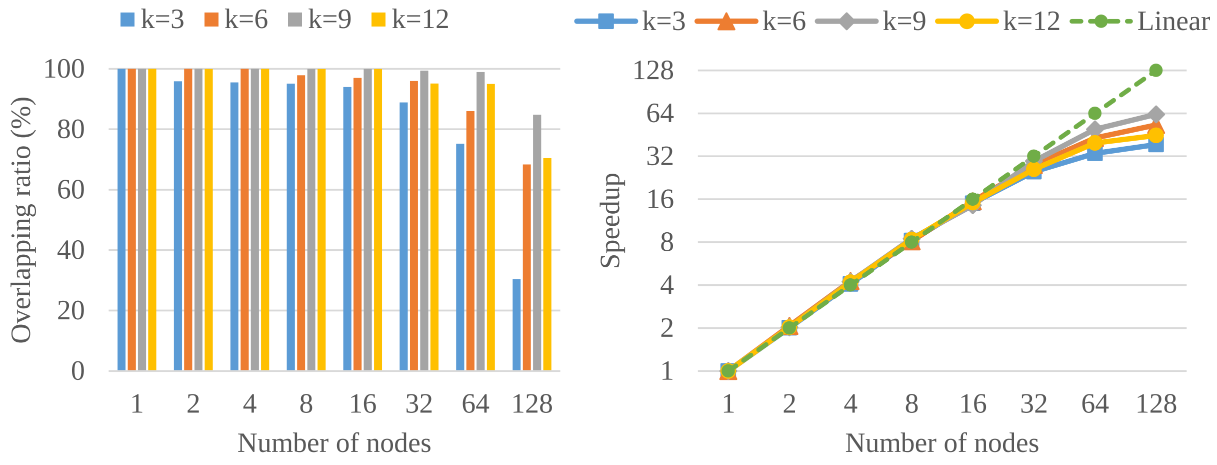


Figure 3.10. Overlapping ratio (left) and speedup (right) with varying number of layers that have the gradients for the first gradient chunk. The gradients should be grouped into two chunks such that the overlapping ratio of two communications are maximized.

parameter server plays a role as a master to update the parameters in a centralized fashion. Our approach is a fully-distributed parallel algorithm which only performs collective-communications such as all-to-all reduction or all-to-all gather, while the master-slave model has point-to-point communications. We do not compare the execution time directly between our approach and the GPU-based training algorithms. First, due to the different underlying hardware architecture, the exeuction time comparison is unfair. Second, all the existing works have different software settings such as model architecture, mini-batch size, and optimization method. Considering these differences, we only compare the scalability instead of the execution time. FireCaffe [**65**] reports 47 speedup of training with synchronous SGD. For asynchronous SGD, Strom et al. achieved a speedup of 54.

Comparison with parallel algorithms on CPU-based clusters – Many researchers have put much effort into scaling neural network training on CPU clusters

Table 3.1. Summary of the previous works. The columns are HW/SW settings. The Max speedup column shows the maximum speedup (left) and how many machines are used (right).

Publication	Communication model	GPU/CPU	sync/async SGD	Max speedup
Theano-mpi	fully-distributed	GPU	sync	7.3/8
GeePS	master-slave	GPU	sync	13/16
FireCaffe	master-slave	GPU	sync	47/128
Strom et al.	fully-distributed	GPU	async	54/80
Dean et al.	master-slave	CPU	sync	12/128
Adam	master-slave	CPU	async	20/90
Das et al.	fully-distributed	CPU	sync	90/128

[**15, 1, 43, 66**]. Recently, Dipankar Das et al. [**1**] reported the state-of-the-art speedup by developing PCL-DNN framework using their multi-threaded communication library which enables to overlap communication with computation. For a mini-batch size of 512, they trained VGG-A on 128 nodes and achieved a speedup of 90. PCL-DNN performs a communication for each layer and overlaps the communication with the backpropagation. Some communications are delayed to the next iteration such that the communication is overlapped with the feed-forward too. We reproduced the work in [**1**] based on the common ground such as distributed-memory parallelism, fully-distributed communication model, and synchronous SGD. To compare the overlapping strategy only, we used the same versions of Intel MKL library and MPI. Figure 3.11 presents the comparison. DP is the baseline which has no overlap and DP+Overlap is the reproduced work. We see that our approach scales better than the others. DP+Overlap hardly scales beyond 32 nodes due to the expensive communications at the fully-connected layers and the poor overlapping ratio.

Recently, in order to tackle larger and more complicated datasets, the deep learning models are getting larger and deeper. For example, modern CNNs for image classification

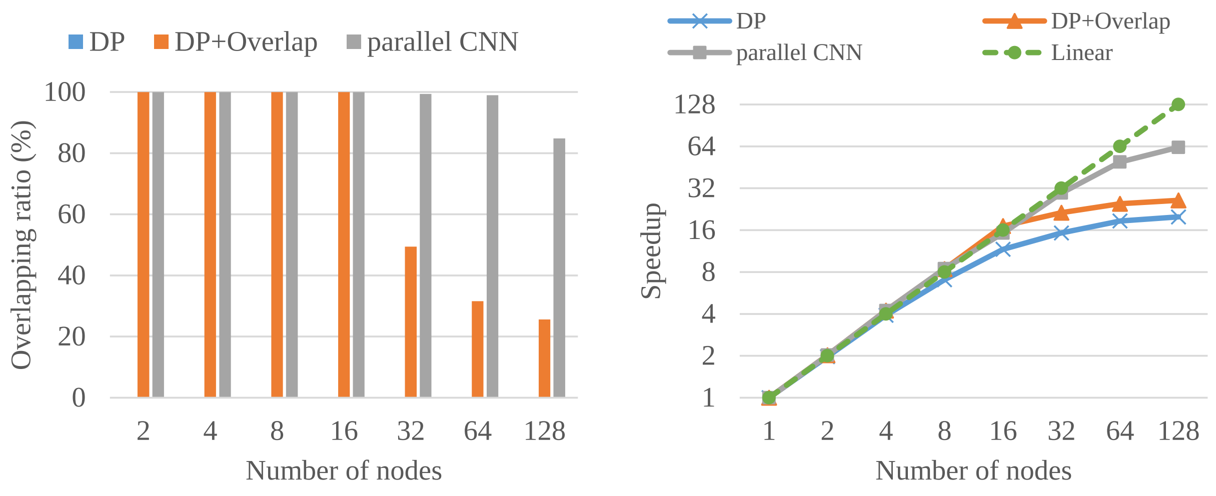


Figure 3.11. Overlapping ratio (left) and speedup (right) of VGG-A training (mini-batch size of 256). DP is the baseline without overlapping, DP+Overlap is the reproduced work based on [**1**], and parallel CNN is our proposed approach.

problems usually have about one hundred layers. For image regression problems such as image restoration and super-resolution, researchers also use very deep networks that have more than 50 $\sim$ 60 layers [**34, 33, 35**]. As shown in many existing works, it is assumed that one communication is performed at each layer. Considering such deep networks, this traditional approach may increase the communication cost due to the large number of communications per iteration. Our experimental results presented in the previous section clearly demonstrate that the proposed gradient aggregation method effectively reduces the per-iteration communication cost. The performance comparison between DP+Overlap, the most popular overlapping strategy, and our proposed approach also shows how the number of communications per iteration affects the scaling performance.

3.5. Discussion

In this Chapter, we have proposed a communication overlapping strategy that improves the scalability of neural network training. We analyzed how much communication time can be hidden behind the computations time every iteration. Then, we have designed a parallel training algorithm that leverages the overlap based on the analysis. Our experimental results demonstrate the effectiveness of the proposed training strategy by showing the significantly improved speedups.

Impact of overlapping on parallel training – Although synchronous parallel training suffers from the expensive communications, many researchers still scale up the deep learning-based scientific applications in a fully synchronous way since it guarantees the same convergence rate to the sequential training [**42, 23, 24, 67**]. Our overlapping strategy is designed considering the data dependency that commonly exists in all types of neural networks. Therefore, many large-scale deep learning applications can directly take advantage of our overlapping strategy to efficiently scale up the training.

In our experiments, we also found that the scaling efficiency of the computations on each node is not close to the linear even when using the highly-optimized math library such as Intel MKL. It has been already known that the computational workload should be sufficiently large to achieve a near linear speedup [**63**]. In neural network training, the matrix size depends on the input data size and the filter size, and the matrices are most likely smaller than 1024 on each dimension. So, even if we dedicate one physical CPU core for communications, the computation performance would not be much affected. We actually observed that the computation time is almost not affected by running the communication-dedicated thread pinned on a physical core while the communication time is effectively overlapped with the computation time.

Insight regarding the communication pattern – This research work is based on an assumption that the locally computed gradients are averaged across all the workers using *allreduce* operations. In MPICH, the *allreduce* is implemented using the optimal algorithm such that it is guaranteed that the communication cost is minimal regardless of the number of processes and the data size. To the best of our knowledge, all the existing popular deep learning software frameworks, including TensorFlow, Horovod, pyTorch, and Caffe, use *allreduce* communications for data parallel training.

Depending on the mini-batch size and the model architecture, the gradient matrix can be much larger than the activation matrix and error matrix. Especially, the fully-connected layers have all-to-all connection patterns and they most likely cause the gradient matrix that is much larger than the activation and error matrices. This observation gives us an insight that the data parallelism does not necessarily have to be implemented using *allreduce* operations. Instead of having each worker compute a full set of gradients from a part of mini-batch, we can consider the opposite way such that each worker computes a part of global gradients from the full mini-batch. This approach allows to exchange the intermediate data such as activations and errors instead of the gradients. In this case, the communication pattern will also be different from the traditional *allreduce*-based approach.

In the following Chapter, we will analyze the communication cost of the traditional *allreduce*-based data parallelism and discuss how to reduce the per-iteration communication cost by re-designing the gradient computation algorithm. Likely to the overlapping strategy proposed in this chapter, we also pursue a general communication strategy that can be directly applied to all the types of neural networks.

CHAPTER 4

Communication-Efficient Parallel Gradient Computation Algorithm

For synchronous data parallel training, the expensive gradient communications are the primary performance bottleneck that hinders efficient scaling. In the previous Chapter, we have shown that the scalability of parallel training can be significantly improved by overlapping the communications with the computations during training. As an extension of the research work, we present a novel gradient computation algorithm that not only further improves the degree of overlap but also reduces the communication cost complexity at fully-connected layers.

The most popular implementation of data parallelism is *allreduce*-based approach. Once every worker locally computes the gradients from the assigned training samples, the entire gradients can be aggregated and summed up using a single *allreduce* operation. This traditional data parallelism implementation is based on an assumption that the gradients are aggregated only after all the workers have the full gradients locally computed from the assigned subset of mini-batch. However, especially at fully-connected layers, the gradient matrix is always larger than the activation matrix or the error matrix. Our design principle is to perform the inter-process communications when the data size is minimized during the gradient computation steps. We re-design the parallel gradient computation algorithm such that the activations and errors relocated across all the workers in advance so that each worker can compute the gradient sums of a distinct subset of the model parameters. Then, the gradient sums are aggregated among all the workers using *allgather* communications. We will analyze and compare the overall communication cost complexity between our approach and the *allreduce*-based approach.

For the later discussion, we first define a few notations: P is the number of workers, K is the mini-batch size, N is the number of neurons at a layer, N' is the number of neurons at the previous layer, D is the number of filters at a convolution layer and F is the size of each filter. Note that our discussion considers only a single layer since the same analysis can be applied to all the layers in the same way.

4.1. Parallel Gradient Computation Algorithm

4.1.1. Fully-Connected Layers

In data parallelism, the standard way of averaging the gradients at a fully-connected layer can be defined as follows: Given the current layer's error matrix of size $N \times \frac{K}{P}$ and the previous layer's activation matrix of size $N' \times \frac{K}{P}$, compute the gradient matrix of size $N \times N'$ by multiplying the two matrices (one may be transposed depending on the data layout). Then, sum up the gradient matrix across all the workers using an *allreduce* operation. Finally, the gradient sums are averaged by multiplying the reciprocal of K to all the elements.

In our gradient computation algorithm, instead of computing a $N \times N'$ gradient matrix for $\frac{K}{P}$ training samples with each worker, the activations and errors are relocated across all the workers and a partial gradient matrix of size $N \times \frac{N'}{P}$ for K training samples are computed by each worker. First, the activations are scattered across all the workers using

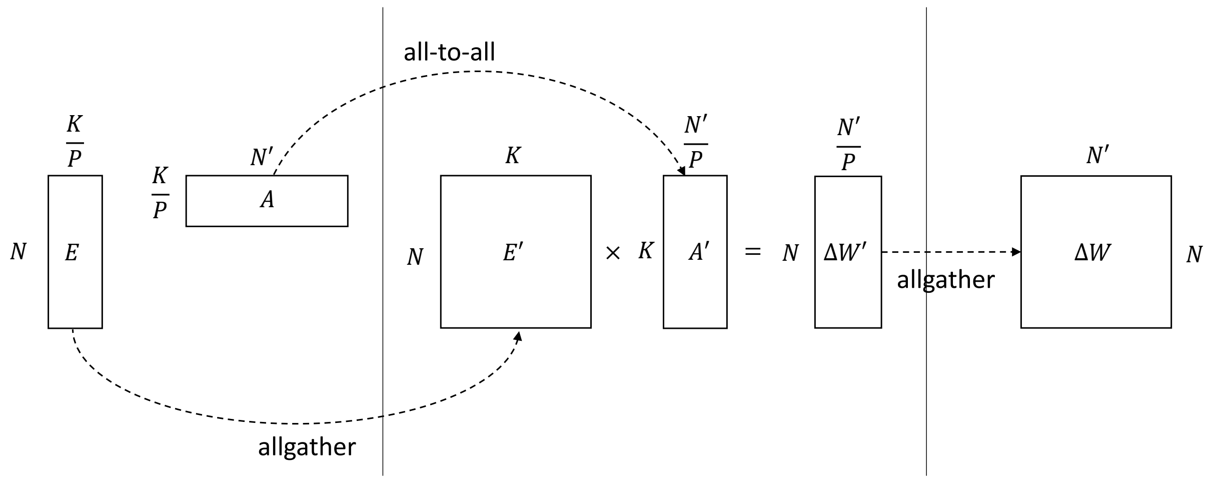


Figure 4.1. Communication-efficient gradient calculation. Given an error matrix E of size $N \times \frac{K}{P}$ and an activation matrix A of size $N' \times \frac{K}{P}$, E is gathered and A is scattered across all the nodes. Then, the gathered error E' is multiplied by the scattered activation A' to compute the partial gradient $\nabla W'$ of size $N \times \frac{N'}{P}$. Finally, the partial gradient matrix is gathered across all the nodes and each node ends up having $N \times N'$ gradient matrix ∇W.

an *all-to-all* personalized communication. The size of the scattered activation matrix is $K \times \frac{N'}{P}$. Second, the errors are gathered across all the workers using an *allgather* operation. The size of the gathered error matrix is $K \times N$. Then, the two matrices are multiplied to be a $N \times \frac{N'}{P}$ matrix that is the partial gradient sums for K training samples. Finally, the gradient sums are gathered across all the workers using an *allgather* operation and each worker ends up having $N \times N'$ gradient sums. Figure 4.1 illustrates the described gradient computation algorithm for fully-connected layers. Overall, the gradient computations algorithm consists of three inter-process communications and one computation at each layer.

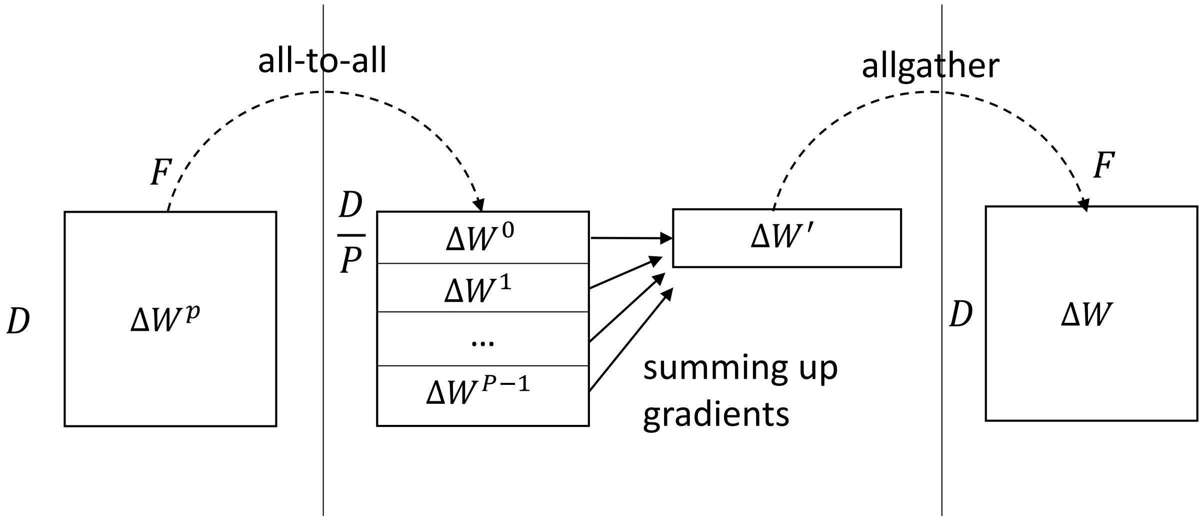


Figure 4.2. Three-step reduction for gradient averaging at convolution layers. First the local gradients ∇W^p are scattered across all the nodes. The received partial gradient matrices are summed up and then gathered across all the nodes to obtain the entire gradient sums ∇W.

4.1.2. Convolution Layers

The standard way of averaging the gradients at a convolution layer is as follows: the gradient matrix of size $D \times F$ is computed by multiplying activations and errors within the local reception field. Then, the gradient matrix is summed up across all the workers using an *allreduce* operation. Finally, the gradients are averaged by multiplying the reciprocal of K to all the elements.

Instead of performing a single *allreduce* operation, our approach is to divide it to multiple steps of communications. First, we scatter the gradient matrix using an *all-to-all* operation and each worker becomes to have P sub-matrices (each of size $\frac{D}{P} \times F$). Second, the P sub-matrices are locally summed up by each worker to have a single sub-matrix of size $\frac{D}{P} \times F$. Finally, the sub-gradient sums are gathered across all the workers

Table 4.1. Theoretical cost of communication patterns for large messages.

Communication pattern	Latency (s)	Bandwidth (w)
reduce-scatter	$P-1$	$n(P-1)/P$
all-to-all	$P-1$	$n(P-1)/P$
allgather	$P-1$	$n(P-1)/P$

using an *allgather* operation. Figure 4.2 illustrates the proposed approach. Overall, our approach consists of two inter-process communications and one computation.

4.2. Communication Cost Analysis

In our theoretical cost analysis, we use the cost model used in many previous works [**68, 69, 70**].

$$T = s\alpha + w\beta, \tag{4.1}$$

where s is the number of messages, w is the overall message size, α is the latency per message, and β is the reciprocal bandwidth. We follow all the assumptions described in [**69**].

In this discussion, we refer to the theoretical communication cost for the collective communications shown in Table 4.1. n in the table is the overall data size. Note that the costs are calculated for the long message algorithms in [**69, 71**] and it is higher than the theoretical lower bounds (pair-wise exchange algorithm is used for reduce-scatter and *all-to-all* while ring algorithm is used for *allgather*).

4.2.1. Fully-Connected Layers

The proposed gradient averaging algorithm for fully-connected layers has three communication steps: *all-to-all* for activations, *allgather* for errors, and another *allgather* for the partial gradient sums. Based on Table 4.1, the communication costs, T_s, T_{g1}, and T_{g2} are computed as following equations. The overall communication cost at a fully-connected layer, T_f is the sum of the three costs.

$$T_s = (P-1)\alpha + \frac{N'K}{P^2}(P-1)\beta \tag{4.2}$$

$$T_{g1} = (P-1)\alpha + \frac{NK}{P}(P-1)\beta \tag{4.3}$$

$$T_{g2} = (P-1)\alpha + \frac{NN'}{P}(P-1)\beta \tag{4.4}$$

$$T_f = T_s + T_{g1} + T_{g2} \tag{4.5}$$

In MPICH, allreduce is implemented with two different algorithms, the binomial tree algorithm for short messages ($\leq$ 2KB) and Rabenseifner's algorithm for long messages ($>$ 2KB) [**72, 69**]. The communication costs are calculated as followings.

$$T_{binomial} = log(P)\alpha + log(P)n\beta \tag{4.6}$$

$$T_{Rabenseifner} = 2log(P)\alpha + 2\frac{P-1}{P}n\beta, \tag{4.7}$$

where n is the overall message size. Since the gradient size at a layer of modern CNNs is most likely larger than 2KB, we only consider Rabenseifner's algorithm in this discussion. In practice, due to the large data size, the bandwidth term $w\beta$ is dominent over the latency cost term $s\alpha$. So, we focus on the second term in Equation 4.1. We can derive the following condition by comparing the bandwidth terms of Equation 4.5 and 4.7. Note that n in Equation 4.7 is NN'.

$$\frac{N'K}{P} + NK < NN' \tag{4.8}$$

If the above condition is satisfied, our gradient computation algorithm guarantees a cheaper communication cost than *allreduce*-based approach. Note that, in modern CNNs, K is most likely smaller than N or N' and the condition is satisfied.

4.2.2. Convolution Layers

As explained, our gradient computation algorithm for convolution layers consists of three steps, an *all-to-all* operation, the computation for accumulating the received matrices, and an *allgather* operation. The overall communication cost at a convolution layer, T_c, is calculated by the following equation.

$$T_c = 2(P-1)\alpha + 2\frac{DF}{P}(P-1)\beta \tag{4.9}$$

Rabenseifner's algorithm is implemented in MPICH using a *reduce-scatter* operation followed by an *allgather* operation. Since n in Equation 4.7 is DF at a convolution layer, the bandwidth term w is same as that of T_c. However, our approach has three practical benefits: First, our approach enables to efficiently sum up the gradients using multiple threads. To the best of our knowledge, most of the MPI implementations do not support multi-threaded internal computation. In data parallelism, since the entire gradients are averaged at each iteration, the multi-threaded reduction can make a significant performance improvement. Second, the communication time can be overlapped with the computation time across different reductions. Since our approach separates the computation step and the communication step, the computation time can overlap the communication time of other reductions. The overlapping strategy will be discussed in the following section in detail. Finally, each workers can locally update only a part of model parameters. The scalable model update is also discussed in the following section.

4.3. Scalable Model Parameter Update

The cost of model parameter update is easily overlooked, however it can be a significant performance bottleneck in parallel neural network training. In our proposed gradient computation algorithm, at both convolution layers and fully-connected layers, the final communication step is *allgather*. We take advantage of the communication pattern to reduce the computation complexity of model parameter update. Instead of exchanging the gradient sums, we locally update the partial model parameters at each worker and perform an *allgather* for the updated model parameters. In this way, the computation complexity of parameter update is $O(\frac{NN'}{P})$ without any extra communications.

In *allreduce*-based data parallelism, all the workers end up having the gradient sums for the entire model parameters. Then, the parameters can be updated in two different ways. On the first hand, the entire model parameters are updated at each worker. In this case, the computation complexity of parameter update is $O(NN')$ which is not scalable.

On the other hand, each worker can update a distinct subset of the parameters and the updated parameters are aggregated across all the workers. In this case, the computation complexity is $O(\frac{NN'}{P})$ but the extra *allgather* should be performed after the update. Most of the existing works perform the former method while Intel distribution of Caffe [**13**] supports the latter one. To further reduce the communication cost, Intel Caffe also supports a gradient averaging algorithm which uses *reduce-scatter* and *allgather*. The algorithm enables the $O(\frac{NN'}{P})$ computation complexity of the parameter update, however the overall communication cost of *reduce-scatter* algorithm is the same as that of the *allreduce*-based algorithm.

4.4. Overlapping Computation and Communication

Overlapping computation and communication is an essential technique for improving the scalability. We present an overlapping strategy based on the proposed gradient computation algorithm.

4.4.1. Fully-Connected Layers

The proposed gradient computation algorithm has three communication steps at a fully-connected layer. First, once the activations are ready, an *all-to-all* communication is posted in the feed-forward stage. The communication time is overlapped with the computation time until it comes back to the layer in the backpropagation stage. Second, an *allgather* communication is posted when the errors are computed and the communication time is overlapped with the later backpropagation time. When these two communication steps are finished, the matrix A' and E' in Figure 4.1 are ready and the gradient sums

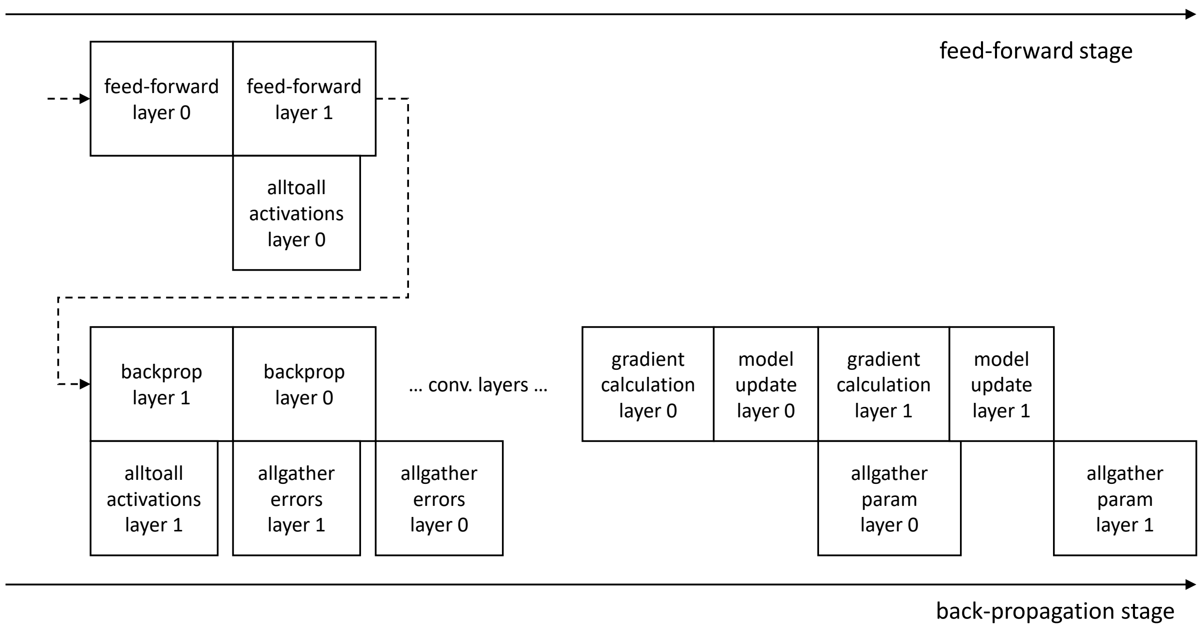


Figure 4.3. An example of the ideal overlapping of 2 layers such that the computation time at each layer is sufficiently large to overlap the communication time. After the errors are back-propagated through all the fully-connected layers, process the convolution layers first. Then, calculate the average gradients, update the partial model parameters, and post an *allgathers* communication for the updated model parameters.

are computed. Based on the scalable model update technique we proposed, a part of model parameters are updated by each worker using the local gradient sums. Finally, an *allgather* is posted to exchange the new parameters across all the workers. The communication time is overlapped with the gradient computation and parameter update times at other layers. Figure 4.3 illustrates the overlapping strategy.

It is worth noting that the final allgathers are posted in the forward order while the errors and the gradients are computed in the backward order. This inversed order enables to overlap the final allgather time with the feed-forward computation time at the

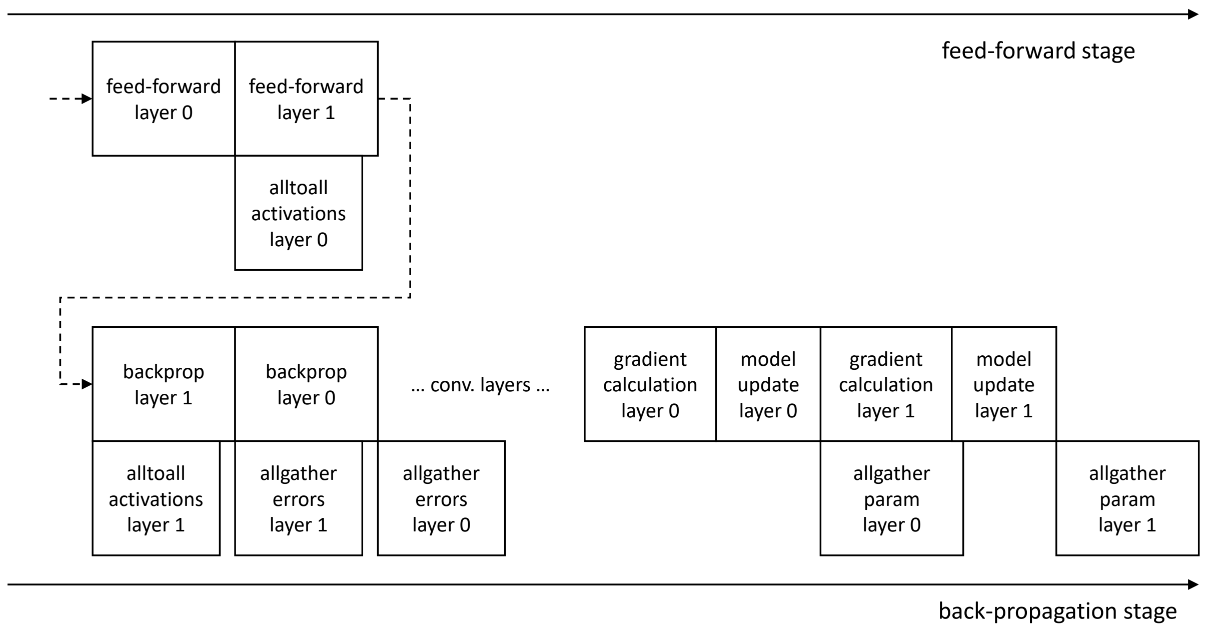


Figure 4.4. An example of overlapping computation and communication at convolution layers: (a) shows the *allreduce*-based approach and (b) shows the proposed two-step communications. (b) enables computation and communication overlaps across layers.

next iteration. While the activations are computed at a layer, the communications for later layers can be performed simultaneously since the model parameters have no data dependency across layers.

4.4.2. Convolution Layers

At convolution layers, we overlap the computation time for summing up the gradients and the *allgather* communication time. As explained, the gradients are averaged with two communication steps at each convolution layer. Between the two communications, the local gradients should be summed up and each worker ends up having the global gradient sums of a subset of model parameters. Since the gradients do not have data dependency across different layers, the computation time can be overlapped with the final step communication at other layers. Figure 4.2 shows example time-flow charts. Figure 4.2.(a) is the *allreduce*-based approach and Figure 4.2.(b) is the proposed two-step communications for averaging gradients.

4.5. Performance Evaluation

In order to evaluate our proposed gradient computation algorithms, we compare it to other parallelization strategies. Recently, there are many open-source software frameworks that support distributed-memory parallel training, such as TensorFlow [**11**], Intel Caffe [**13**], PyTorch [**12**], and Horovod [**73**]. Most of them adopt the traditional *allreduce*-based data parallelism. Intel Caffe supports a gradient averaging algorithm which uses *reduce-scatter* and *allgather* operations. PyTorch and Horovod use *ring-allreduce* algorithm [**74**] which utilizes the network bandwidth more efficiently than the other *allreduce* algorithms. Since all the open-source frameworks use different data structures, computation algorithms, and communication libraries, instead of comparing them directly, we implement the representative parallelization strategies and compare our parallelization strategy with them. Note that we do not compare the classification accuracy since we only consider synchronous-parallel SGD which guarantees the optimal parameter update. For our evaluation, we perform ImageNet classification which is the most popular benchmark for deep learning study.

4.5.1. Experimental Settings

We perform the experiments on Cori, a Cray XC40 supercomputer at the National Energy Research Scientific Computing Center. Each Haswell node has two sockets and each socket contains a 16-core Intel Haswell processor at 2.3GHz. The system has Cray Aries high speed interconnections with 'dragonfly' topology.

We use ImageNet-1K dataset [**64**] for our experiments. ImageNet has 1.2 million 3-channel(RGB) images of various sizes for training and 50,000 images for validation. We isotropically rescaled all the images such that the shorter side has 256 pixels. Then, we randomly cropped them to 224×224. Finally, all the pixels are subtracted by the mean value.

We use two representative CNN models: VGG-16 and ResNet-50. VGG-16 is a regular CNN model proposed by VGG group in Oxford [**62**]. The model consists of 8 convolution layers followed by 3 fully-connected layers. The number of parameters is 138 million in total. ResNet-50 is one of the most popular residual network which is a variant of the regular CNN [**33**]. The model consists of 49 convolution layers and 1 fully-connected layer. The overall number of parameters is 25.5 million. We use the mini-batch size of 256 which has been used by the original model designers in [**62, 33**].

We use our own parallel deep learning software framework introduced in Section 3.4. In our software framework, a single MPI process runs on each node and each process employs shared-memory programming model to utilize all the cores within a node. This programming model allows to have only a single model in the memory space on each node. When spawning threads using OpenMP, we use all 32 physical cores in each node.

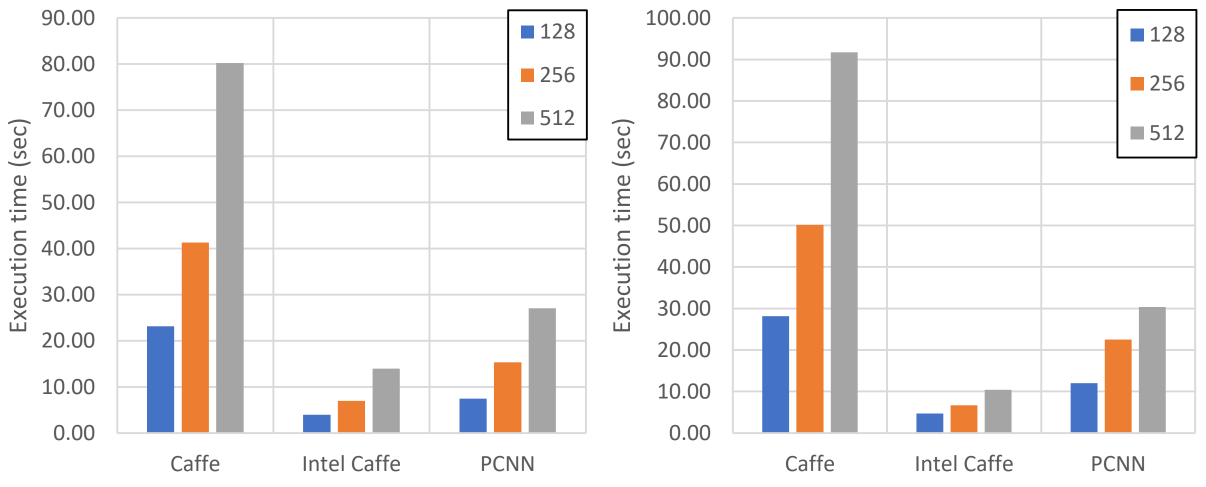


Figure 4.5. Single-node execution time for processing a single mini-batch. VGG-16 model (left) and ResNet-50 (right) with varying mini-batch sizes. The performance is measured on a Haswell node of Cori.

We calculate the speedup based on the number of cores. Since the mini-batch size in our experiments is 256, using data-parallelism, we use up to 256 nodes (8192 cores in total).

4.5.2. Single-Node Performance

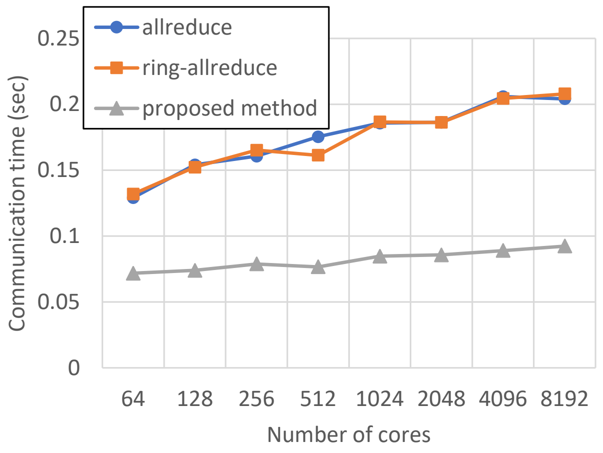
We begin with reporting the single node performance of our software framework. In the later experiments, we calculate the speedup with respect to the number of compute cores using these single node execution times. We compare our software framework, Parallel CNN (PCNN), with the original Caffe [**13**] as well as the intel distribution of Caffe. Caffe is one of the most popularly used open-source frameworks for deep learning. Intel Caffe is a highly optimized version of Caffe for utilizing the Intel CPU hardware features. We believe this comparison can demonstrate that our parallel performance study is based on the reasonable level of the single node performance.

Figure 4.5 presents the single node performance with varying mini-batch sizes. The left-side and right-side charts show VGG-16 and ResNet-50 execution times respectively. The performance was measured on a Haswell node of Cori. Note that we only consider the execution time for processing a single mini-batch since the same workload is repeated for all the mini-batches. The execution time is the average of 5 times of measurements. We see that PCNN shows a comparable single node performance to Intel Caffe.

4.5.3. Communication-Efficient Gradient Computation

We first compare our proposed gradient computation algorithm with the traditional *allreduce*-based approach. In order to compare the communication time only, we emulate the communication patterns using MPI primitive functions and compare the overall communication times. The traditional *allreduce*-based data parallelism is implemented using MPICH *allreduce* and *ring-allreduce*. The *reduce-scatter* algorithm in MPICH *allreduce* [**71**], the circular algorithm in *ring-allreduce* [**74**], and our proposed algorithm are implemented using `MPI_Send` and `MPI_Recv functions`.

In VGG-16, the first fully-connected layer has 102,760,448 weight parameters (392 MB) which take up about 77% of the overall parameters. We measure the communication times for averaging the gradients at the layer and compare the timings among different approaches. Figure 4.6 shows the experimental results. The left-side chart is the overall communication time comparison and the right-side chart is the timing breakdown of our algorithm. We see that our proposed algorithm significantly reduces the communication time. For the first fully-connected layer in VGG-16, the number of activations at the previous layer N' is 25,088, the number of errors at the current layer N is 4,096, the



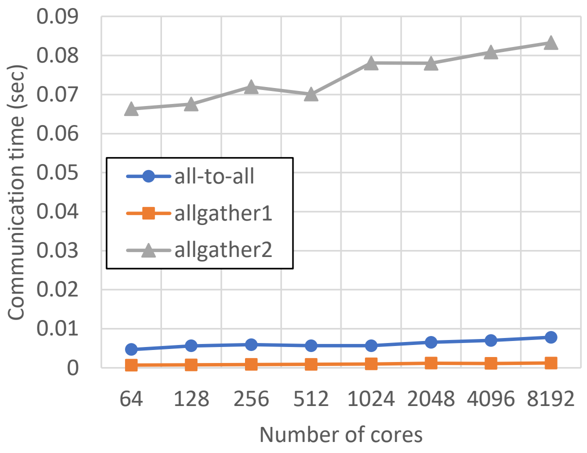


Figure 4.6. Communication time comparison (left) and communication timing breakdown (right). Our approach is compared with *allreduce* in MPICH as well as *ring-allreduce*. The overall data size is 392MB (the gradient size at the first fully-connected layer of VGG-16 model). Our proposed method has a shorter communication time than the other two methods.

mini-batch size K is 256. So, the layer satisfies the condition, $\frac{N'K}{P} + NK < NN'$, and our algorithm takes a shorter communication time compared to the other algorithms. The timing breakdown on the right-side shows how much time is spent for each of the three communication steps in our algorithm. This result demonstrates that the proposed algorithm has a communication complexity of $O(1)$ for all the three communication steps.

4.5.4. Strong Scaling Performance

To evaluate the impact of the proposed algorithms on scalability, we measure the end-to-end execution time for processing a single mini-batch and the speedup with respect to the number of cores. We use VGG-16 and ResNet-50 models for this experiment and the mini-batch size of 256.

All the open-source frameworks have different overlapping strategies. For example, Tensorflow overlaps the *allreduce* time with backpropagation time only whereas Intel Caffe overlaps the communication time using both the backpropagation time and the feed-forward time at the next iteration. So, we chose the best overlapping strategy among them and reproduced it using *allreduce*-based data parallelism. We categorized all the parallelization strategies into 4 cases: 'allreduce_no_overlap', 'allreduce', 'pcnn_no_overlap', and 'pcnn'. 'allreduce' is the implementation of the overlapping strategy used in Intel Caffe, which utilizes the backpropagation time, feed-forward time and model update time for overlapping. The *allreduce* communication is posted right after the local gradients are computed at each layer. 'pcnn' is our software framework which uses all the proposed algorithms. Figure 4.7 presents the speedup comparison among the four parallelization strategies. We see that, for both models, 'pcnn' shows a clear improvement over the others. For VGG-16, 'pcnn' achieves up to 2516.36× speedup while 'allreduce' achieves up to 559.12× speedup. For ResNet-50, 'PCNN' achieves up to 2734.25× speedup while 'allreduce' achieves up to 1572.01× speedup. Figure 4.8 shows the execution times for both models. The charts present the results from 512 cores (16 nodes) to clearly show the difference among the four cases. 'pcnn' always out performs 'allreduce' and the proposed overlapping strategy further reduces the execution time and it results in achieving the higher speedup.

4.5.5. Overlapping Communications with Computations

The efficient overlap of computation and communication plays a key role in our parallelization strategy. If the computation time is not long enough to hide the entire communication

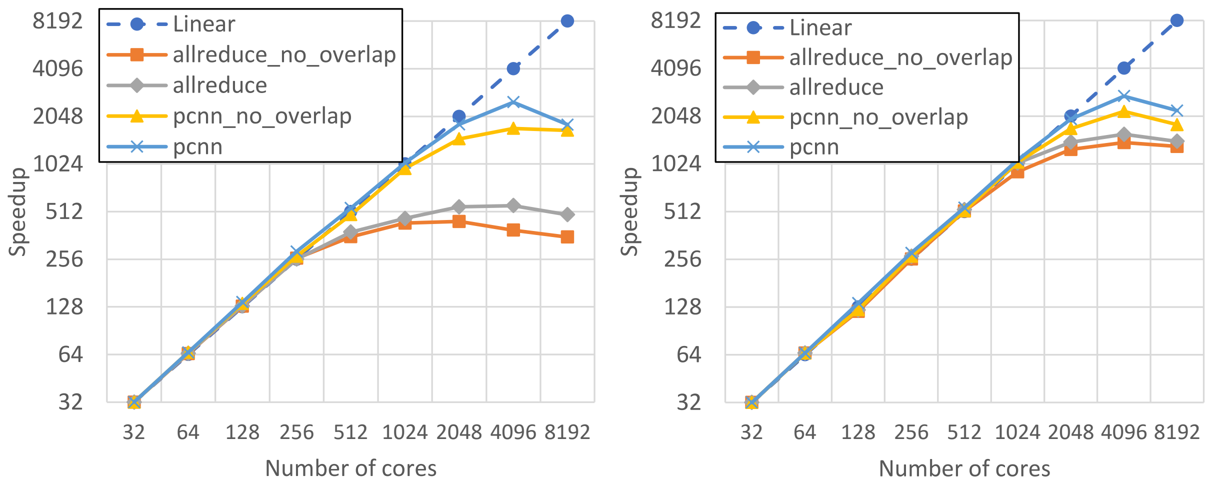


Figure 4.7. Strong scaling results for VGG-16 (left) and ResNet-50 (right) models. The mini-batch size is 256. ‘allreduce’ is the traditional allreduce-based data-parallelism and ‘pcnn’ is the proposed parallelization strategy. The speedups are measured using up to 8192 cores.

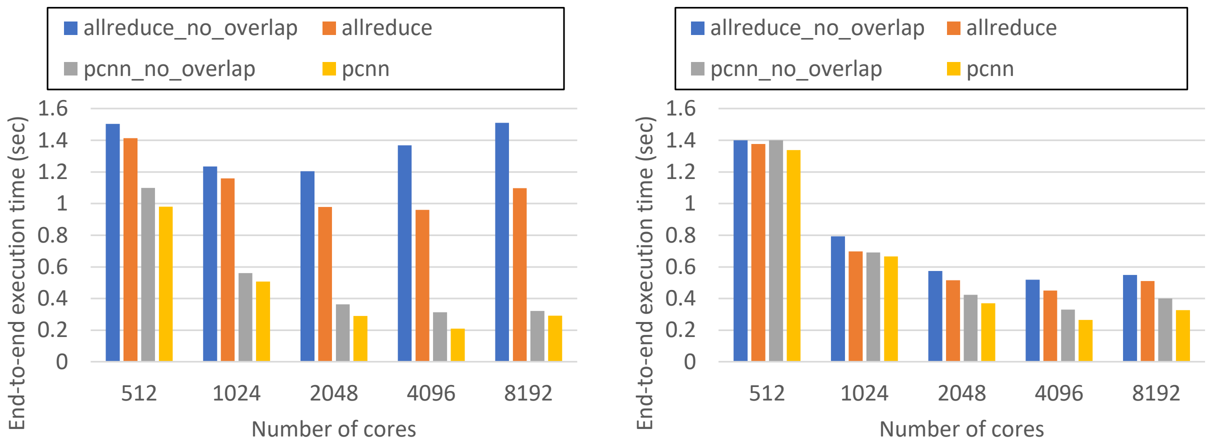


Figure 4.8. End-to-end execution time for processing a single mini-batch. VGG-16 (left) and ResNet-50 (right). The mini-batch size is 256. The results show that ‘pcnn’ outperforms the other approaches.

time, the next computation time should wait for the communication to be finished. We define this delay as ‘measurable communication time’. To evaluate the degree of overlap,

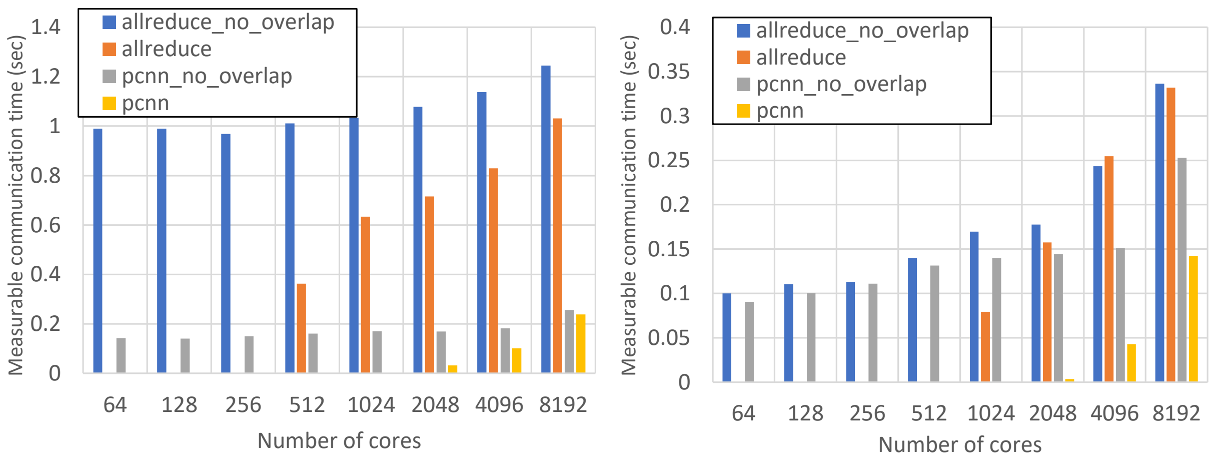


Figure 4.9. Measurable communication time comparison. VGG-16 (left) and ResNet-50 (right). The mini-batch size is 256. If the communication time is entirely overlapped with the computation time, the measurable communication time would be zero. 'pcnn' always shows lower measurable communication time than that of 'allreduce'.

we compare the measurable communication time at each layer among different parallelization strategies. We have the same four parallelization strategies: 'allreduce_no_overlap', 'allreduce', 'pcnn_no_overlap', and 'pcnn'.

Figure 4.9 shows the measurable communication times in VGG-16 training (left) and ResNet-50 training (right). On both charts, we clearly see that the measurable communication time is reduced by the overlapping. In VGG-16 training, the overall communication time is considerably reduced by our gradient averaging method at the fully-connected layers and the degree of overlap is also affected by the reduced communication time. 'pcnn' starts to have non-zero measurable communication time on 2048 cores while 'allreduce' does on 512 cores. In ResNet-50 training, the measurable communication time of 'pcnn' is almost zero on 2048 cores, which means that a linear speedup can be expected. In

Figure 4.7, on the right chart, 'pcnn' achieves a linear speedup on 2048 cores. In contrast, 'allreduce' has non-zero measurable communication time on 1024 cores. This result demonstrates that our proposed methods effectively improve the degree of overlap.

4.6. Discussion

Impact on Different Types of CNNs – Depending on the type of neural network and the model architecture, our proposed algorithms can affect the scalability differently. First, the proposed gradient computation algorithm provides a cheaper communication cost at fully-connected layers compared to the *allreduce*-based approach. Thus, if a model has many fully-connected layers, more performance improvement can be expected. In this research work, we used a regular CNN (VGG-16) as well as a residual network (ResNet-50). In practice, compared to the regular CNNs, the residual networks likely have fewer fully-connected layers. In Figure 4.7, VGG-16 shows a clearer speedup difference between 'pcnn' and ' allreduce' than ResNet-50. Second, the overlapping strategy we proposed hides the communications behind not only the backward computations but also the forward computations at the next iteration. If a model has a higher ratio of computation to communication, more communication time can be overlapped, and a higher speedup would be achieved. In Figure 4.9, ResNet-50 shows a larger difference of measurable communication time between 'pcnn' and 'pcnn_no_overlap' than VGG-16. Due to the dense residual connections in ResNet-50, the ratio of computation to communication is much higher than that of VGG-16, and it results in achieving a higher degree of overlap.

Large-Batch Training – As briefly introduced in Section 2.1, several techniques for large-batch training [**50, 47**] have been proposed recently. The techniques enable to increase the mini-batch size achieving a higher speedup of parallel training. Given a fixed number of training samples, the large mini-batch size reduces the number of iterations to process the entire training dataset (epoch). Therefore, the overall number of communications at each epoch is reduced, which results in achieving better scalability.

Unfortunately, the increased batch size does not affect the communication cost per iteration. Synchronous parallel training algorithm computes the gradients with respect to the entire model parameters and averages them across all the workers at every iteration. Therefore, the overall communication cost is always the same regardless of the batch size.

It is worth noting that our communication-efficient gradient computation algorithm and the existing large-batch training techniques tackle different problems. Our proposed algorithms address the expensive per-iteration communication cost issue while the large-batch training techniques aim to reduce the overall number of communications per epoch. If Inequality 4.8 is satisfied, the proposed gradient computation algorithm guarantees the lower communication cost at fully-connected layers than that of the *allreduce*-based approach. Therefore, we can consider these two approaches as orthogonal solutions for improving the scalability of parallel neural network training. In addition, our overlapping strategy will further improve the scalability regardless of the mini-batch size. Since the large mini-batch size raises up the ratio of computation to communication, a higher degree of overlap can be expected.

Another critical drawback of large-batch training is the extremely large memory footprint. As the batch size increases, more intermediate data such as activations, errors, and gradients should be located in the memory space. Therefore, the entire memory footprint rapidly grows as the batch size increases making it less practical. Recently, it is common to use GPU systems for large-scale deep learning applications. Considering the limited memory space of GPUs, increasing the batch size may cause the out of memory issue for large networks. Our proposed communication-efficient gradient computation algorithm improves the scalability without affecting the memory footprint. Therefore, we propose to use a moderate batch size that makes a good trade-off between the memory footprint and the degree of parallelism, and then employ the proposed gradient computation algorithm to achieve the best speedup of synchronous parallel neural network training.

CHAPTER 5

Adaptive Batch Size Adjustment Method for Scalable Deep Learning

Deep learning has provided the state-of-the-art solutions to a variety of real-world problems. While users enjoy its success, training deep neural networks is, in fact, an extremely compute intensive task that can take hours or even days to complete. Considering the ever-increasing size of available training data, efficient parallelization is crucial to finish the training in a reasonable amount of time. In data parallel synchronous SGD, each mini-batch is evenly distributed to all workers and concurrently processed. This parallelization strategy exhibits a strong data dependency between any two consecutive iterations, i.e. iteration $(i+1)$ cannot proceed before the completion of iteration i. Without the possible cross-iteration concurrency, the degree of parallelism is limited by the number of data samples in a mini-batch. Thus, increasing mini-batch size becomes an intuitive approach to employ more workers in the hope of reducing execution time.

Several recent parallelization works presented performance results scalable up to thousands of nodes using extremely large mini-batch sizes such as 8K or 16K [**25, 23, 42, 24**]. However, most of them also acknowledged that using large batch sizes can result in achieving a lower validation accuracy. The impact of batch sizes on the accuracy has been statistically analyzed in [**75, 46, 76**]. Figure 5.1 illustrates such impact using an example of an EDSR [**35**] training on the DIV2K super-resolution image dataset [**77**]. Each

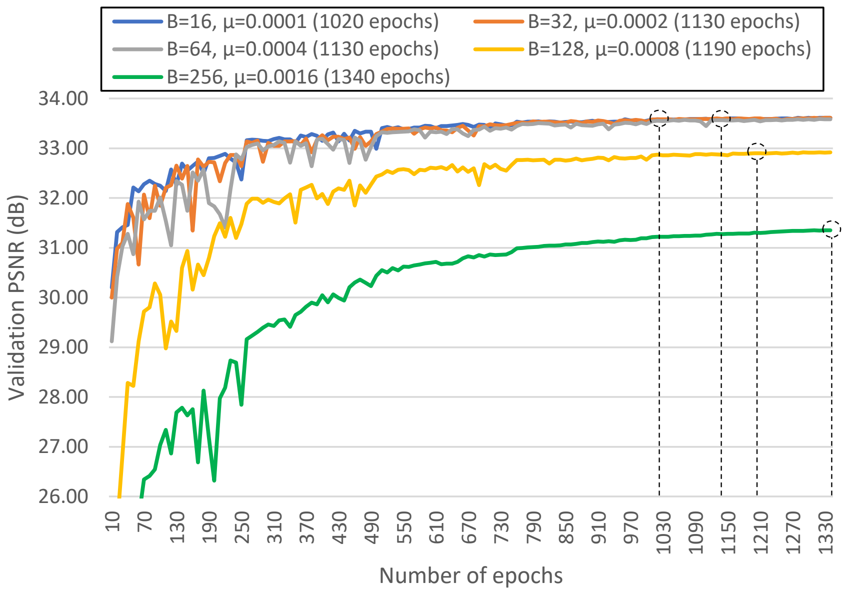


Figure 5.1. Learning curves for EDSR training on DIV2K dataset. B is the mini-batch size, μ is the learning rate, and the numbers shown in brackets are the number of epochs till model converged. The training terminates when the validation accuracy has not increased for 50 consecutive epochs. Batch sizes larger than 64 result in significantly lower accuracy.

learning curve corresponds to mini-batch sizes, ranging from 16 to 256 images. When the batch size increases, the training converges more slowly (in the number of epochs) and achieves a lower validation accuracy. Such similar trends of learning curves are also shown in [**50, 47, 23, 46, 78, 76, 48**]. Owing to this observation, we argue two evaluation principles below in order to ensure a fair performance comparison among different neural network training methods.

- Timing comparison is only fair among methods that produce the same model accuracy or within a small, tolerable margin.
- The training time should be measured from the beginning until the accuracy converges to a stable value.

The former argues a fair comparison under the condition of the same input and output. As shown in Figure 5.1, $B = 128$ and $B = 256$ give much lower accuracy than smaller B values. Models produced with a lower accuracy are usually regarded of no use to domain scientists. The latter describes the unique characteristics of neural networks whose training process is not considered completed until the convergence condition is met. This argument stems from our study of recent parallelization works that measured the time up to a fixed number of epochs to represent the performance of a training method when calculating the speedups and comparing against other methods. In this study, we present our experimental results and analysis by following the above two principles.

We propose a parallel CNN training strategy that adjusts the mini-batch size during the training. The common practice of neural network training is to tune the mini-batch size to a small value that produces the best accuracy. Especially for image restoration or super-resolution problems, the mini-batch size is typically tuned to a small value between $16 \sim 64$ [**79, 80, 81, 82, 34, 83, 35**], which is too small to effectively scale up the parallel training. Our goal is to improve the degree of parallelism without a significant loss in validation accuracy. In our design, the training begins with a small batch size and it gradually increases. To increase the batch size without affecting the gradient noise scale, we also adjust the learning rate as the batch size increases. The interval of batch size increase is adaptively determined based on the ratio of cost reduction to the distance between the initial parameters and the latest ones. We also propose to dramatically lower the learning rate when the training cost is flattened, to keep the generalization performance from being degraded.

Besides adjusting the mini-batch size and learning rate, our parallelization strategy also focuses on the overlapping of communication and computation. In data parallel training, the locally computed gradients are averaged among all workers before updating the parameters. We implement the averaging with MPI *all-to-all* personalized communication followed by a local summation and an MPI *allgather* communication. By having two separate communications, not only the backpropagation computations but also the feed-forward computations at the next iteration can overlap the communications.

5.1. Adaptive Batch Size Adjustment Method

In this section, we discuss the problems in large batch training and our solutions to them. We begin with describing the impact of the large batch size on the training result as well as the potential problems. Then, we present our training strategy which addresses the described problems by adaptively adjusting the batch size and learning rate at run-time.

5.1.1. Impact of Batch Size on Model Accuracy

In this research work, we consider minimization problems of the form

$$F(w) = \frac{1}{N}\sum_{i=1}^{N} f(w, x_i), \tag{5.1}$$

where N is the number of training samples, w is the model parameters, x_i is the i^{th} training sample, and f is the cost function of w and x.

Mini-batch SGD computes the gradients from a random subset of training samples (which is called mini-batch). The stochastic gradients can be considered as a random variable with mean of $\nabla F(w)$. Based on Central Limit Theorem, the variance of the random variable is inversely proportional to the mini-batch size [**53**]. If the variance is large, the gradients can be considered as noisy. Smith et al. analyzed the impact of the batch size on the gradient noise scale [**75**]. The noise scale describes the correlation between the batch size and the random fluctuation of SGD dynamics. For SGD, the noise scale is approximated by the following equation under the assumption of $N \gg B$.

$$g \approx \mu \frac{N}{B} \tag{5.2}$$

This analysis shows that, when using a large batch size, the learning rate should be proportionally increased to make the noise scale stay the same. The experimental results reported in [**75**] demonstrate that this analysis can be applied to variants of SGD such as momentum SGD and Adam [**32**]. Goyal et al. proposed 'linear scaling rule' in [**50**] that can be explained by this analysis. The authors empirically showed that large batch sizes can be used for ImageNet classification without losing the accuracy when the learning rate is proportionally increased. Hoffer et al. proposed 'root scaling rule' in [**48**]. The authors proved that the variance of the stochastic gradients is proportional to $\frac{\mu^2}{B}$. Their statistical analysis is that the variance can stay the same when the learning rate is proportional to the square root of the batch size increase as follows.

$$\mu \propto \sqrt{\frac{B}{B_0}}, \tag{5.3}$$

where B_0 is the best-tuned small batch size and B is the increased batch size. Recall that the stochastic gradient is considered as a random variable with mean of $\nabla F(w)$. By making the variance stay the same, one can expect a similar convergence rate.

Although these two works have derived different update rules, they provide a common insight that the reduced noise scale with an increased batch size can be compensated by increasing the learning rate. In practice, the gradients should be sufficiently noisy to achieve a good accuracy [**53, 75, 46**]. Especially, in non-convex problems such as neural network training, the noisy gradients help the model avoid falling into a sharp minima which poorly generalizes to the test dataset.

5.1.2. Impact of Batch Size on Parallel Performance

When using the data parallelism strategy, the degree of parallelism depends on the problem size that can be partitioned among all the available processes. In the case of CNN training, it is the mini-batch size, as the minimum indivisible unit of workload that can be assigned to a process is a single training sample. Data parallelism partitions the samples in each mini-batch evenly among all the processes. The highest degree of parallelism is thus B. Therefore, larger mini-batch size enables a parallel algorithm to run on more processes.

Using a large batch size can also reduce the communication cost per epoch. Recall the communication for each batch is to average the gradients across all the processes. Given a network, the number of model parameters is independent from the batch size. In other words, the communication amount for averaging the gradients is not affected by the batch size. However, the number of communications is equal to the number of mini-batches in each epoch, N/B. Increasing the value B effectively reduces the value of N/B.

In terms of computation, the amount of parameter updates per epoch iteration is also reduced when using a large batch size. Given N training samples, the number of parameter updates per epoch is $\frac{N}{B}$, where B is the batch size. Since each parameter update takes the same amount of computation, increasing B proportionally reduces the number of updates.

5.1.3. Problems in Training with Large Batch Size

In this research work, we focus on two problems that can be observed in large batch training.

- The training cost $F(w)$ is not effectively reduced yielding a poor convergence accuracy.
- The model easily loses generalization performance.

First, the large batch size causes a low variance of the stochastic gradients and SGD quickly converges providing a low convergence accuracy. This problem can be alleviated with warm-up techniques such that the training starts with a small learning rate and then increases it after a pre-defined number of epochs. However, if the batch size is larger than a certain problem-dependent threshold, the cost is still not effectively reduced [**50, 47**]. Second, it is already known that large batch sizes can make the model lose the generalization performance [**46, 78**]. In other words, the large batch training tends to over-fit the model so that the cost function is well minimized while providing a low validation accuracy.

We address these two problems by adjusting hyper-parameters during training. In the following subsections, we discuss how to address the described problems with adaptive batch size and learning rate control methods.

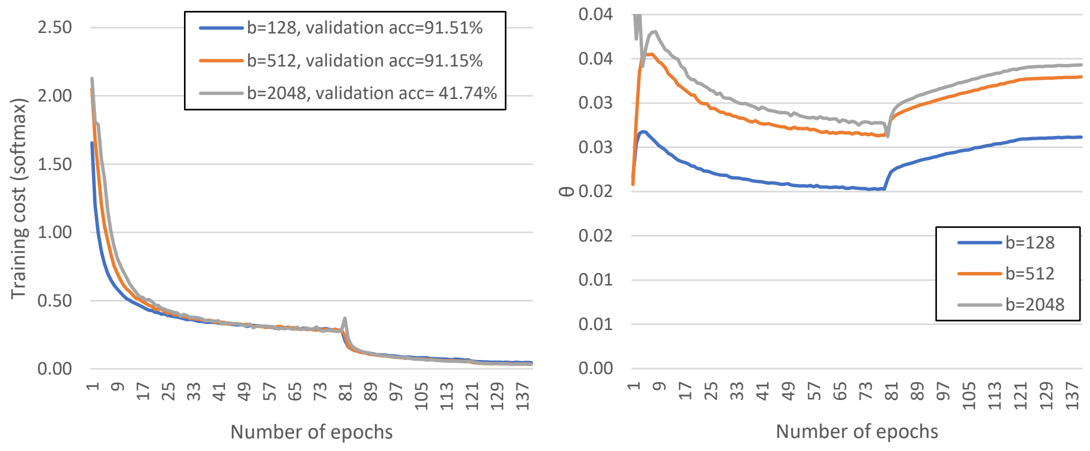


Figure 5.2. The training cost curves (left) and θ curves (right) for ResNet20 training on CIFAR10 datasets. All three batch sizes achieve almost the same training cost after 140 epochs. However, higher the θ curve, lower the validation accuracy. This result demonstrates that θ roughly shows how sharp the minimizer is. Note that the high θ at the beginning of 'b=2048' curve is due to the learning rate warmup.

5.1.4. Adaptive Batch Size Control

The main idea of our training strategy is to begin the training with a small batch size B_s and gradually increase the batch size during the training. As the batch size increases, we also increase the learning rate at run-time to minimize the impact of the increased batch size on the gradient noise scale. The batch size and learning rate are adjusted after every K epochs until the batch size reaches the maximum size B_m.

The small batch size at the early training epochs helps rapidly lower the training cost. So, K should be sufficiently large to effectively minimize the cost function. On the other hand, a large K indicates that the degree of parallelism is limited for more epochs due to the slow batch size increase. Our training algorithm aims to find the smallest K that effectively reduces the training cost.

We define a practical metric θ for estimating the sharpness of the minimizer.

$$\theta_i = \frac{F(w_0) - F(w_i)}{\|w_0 - w_i\|}, \tag{5.4}$$

where w_0 is the initial parameters. This metric shows the ratio of the cost reduction to the distance between the initial parameters and the latest ones. We can roughly estimate how sharp the current minimizer is by checking θ. Given a training cost $F(w_i)$, lower θ means the parameters have more moved to achieve the same cost reduction. Figure 5.2 shows the training cost curves (left) and the θ curves (right) of ResNet-20 training on CIFAR10 dataset with varying batch sizes. Even though all the batch sizes achieve almost the same training cost after 140 epochs, the validation accuracy varies significantly. We clearly see that higher the θ curve, lower the validation accuracy. This result demonstrates that θ can be considered as an indirect metric for measuring the sharpness of the minimizer.

We increase the batch size after the θ curve peaks so that the cost is sufficiently reduced before the batch size starts to increase. For example, in Figure 5.2, the batch size of 128 shows the peak θ at 5^{th} epoch. So, we set $K = 5$ so that the batch size increases after every 5 epochs. We also see that the θ curve for a large batch size peaks later than that of the smaller batch sizes. So, by increasing the batch size gradually after every K epochs, we can expect the batch size increases after the θ curve of each batch size has already peaked. Such careful adjustments also help avoid the cost fluctuation caused by the increased learning rate.

Algorithm 3 is a CNN training algorithm with the proposed adaptive batch size method. The algorithm iteratively traverses over all the training samples until the stop condition is satisfied. Typically, the training stops when either the parameters are not

Algorithm 3 SGD with Increasing Batch Size. (E: the number of epochs, N: the number of training samples, w_0: initial model parameters, μ_0: initial learning rate, B_s: the starting batch size, B_m: the maximum batch size, f: the cost function)

```
1: $w \leftarrow w_0, B \leftarrow B_s, \mu \leftarrow \mu_0, n \leftarrow 1, K \leftarrow \infty$
2: while stop condition is not met do
3:     for $i \leftarrow 1 \cdots \frac{N}{B}$ do
4:         $\mathcal{B} \leftarrow i^{th}$ mini-batch of size $B$.
5:         $\nabla w \leftarrow$ Compute_Gradient($f$, $\mathcal{B}$, $w$).
6:         Update $w$ using $\nabla w$.
7:     if $K$ is $\infty$ then
8:         Compute $\theta$ using Eq. 5.4.
9:         if $\theta$ is not changed more than 10%, then
10:            $K \leftarrow n$.
11:    if ($n$ mod $K$) = 0 and $B < B_m$ then
12:        Increase both batch size $B$ and learning rate $\mu$
13:    Increment $n$ by 1
```

further adjusted due to the small gradients or a target accuracy is achieved. For each mini-batch $\mathcal{B}$, the gradients of a cost function f are computed with respect to the parameters at line 5. The parameters are updated using the averaged gradients at line 6. The θ is monitored and K is determined when θ starts to be saturated at line 7 $\sim$ 10. In this study, we consider θ is saturated if it is changed less than 10%.

When increasing the batch size and learning rate at line 12, based on the statistical analysis in [**75**], we adjust the batch size and learning rate together with a same factor to make the gradient noise scale stay the same. To force the convergence of SGD, we lower the gradient noise scale by decaying the learning rate once the training cost is saturated.

The batch size can be increased to a certain problem-dependent threshold without affecting the accuracy [**46, 50**]. For example, it has been shown that, the batch size for ImageNet classification can be increased to 4096 $\sim$ 8192 without affecting the accuracy [**50**]. We call this batch size 'maximum stable batch size'. By setting B_s to the maximum stable batch size, we can significantly improve the degree of parallelism. In practice, the maximum stable batch size can be easily found by comparing l2-norm of the gradients among batch sizes. A sufficient condition for gradients to be a descent direction with respect to the parameters is as follows [**53**].

$$\|\nabla w - \nabla F(w)\| < \|\nabla w\|$$

Note that the expected value of the left-hand side of the above inequality is the variance of the gradients [**46**]. Assuming the above condition is satisfied at most of the iterations, if two batch sizes give a similar l2-norm of the averaged gradients, it implies they have a similar maximum allowed variance of the gradients and they likely have a similar convergence property. In the later discussion, we assume B_s is set to the maximum stable batch size found by the described method.

5.1.5. Adaptive Learning Rate Control

In Algorithm 3, the batch size increases to B_m most likely before the training loss is saturated. If the cost is minimized using such a large batch size, the model is easily attracted to a sharp minimizer. The generalization problem of large batch training has been already observed in many previous works [**48, 46, 78**].

To alleviate such effect, we intentionally lower the cost reduction speed after the first training cost saturation by dramatically lowering the learning rate. Once the training cost is saturated, one can decay the learning rate to further lower the cost. The decayed learning rate enables to fine-tune the parameters rather than exploring the parameter space, so that it further lowers the training cost and ends up converging into a minima. We consider step-wise learning rate decay with a decay factor β. Once the training cost

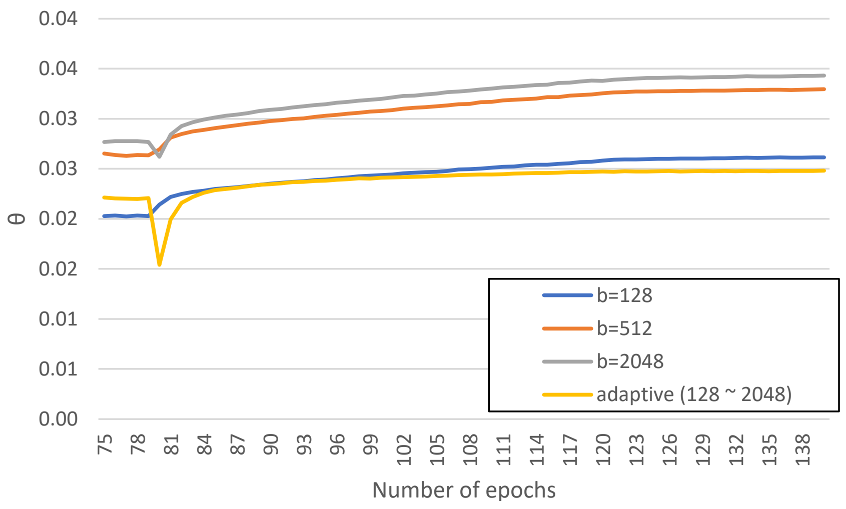


Figure 5.3. The ratio of cost reduction to the distance between the initial parameters and the current ones, θ, for ResNet20 training on Cifar10 datasets. The proposed adaptive learning rate method keeps θ curve from being increased after the first learning rate decay step at 80^{th} epoch.

is saturated, we scale down the learning rate to the initial learning rate μ_0 first and then multiply it by β. So, the effective learning rate decay factor becomes $\frac{B_s}{B_m}\beta$ for the first decay step. For the later decay steps, we use the fixed decay factor β only.

When the learning rate decays, as shown in Equation 5.2, we can expect a faster convergence rate due to the lower noise scale of the gradients. In our training algorithm, since the batch size has been increased to B_m, the variance of the gradients is lower than that of the small batch training. So, the model will rapidly converge to a minima which has a poor training cost. However, we found that such fast convergence enables to maintain the generalization performance even using the large batch size. Figure 5.3 shows the θ curves for ResNet-20 training on Cifar10 dataset after the first saturation of training cost. We see that θ of our proposed method does not significantly increase after the first learning rate decay step, while the other curves commonly increase. Note that our proposed method achieves 90.78% validation accuracy using up to 2048 batch size while the traditional SGD with the same batch size achieves 46.29% after 140 epochs.

5.2. Parallel Training with Adaptive Batch Size

Our design is based on the data parallelism that distributes data among processes while keeping the model parameters duplicated. Since the minimum, indivisible data unit that can be assigned to each process is a training sample, for instance an image, we distribute the samples in each mini-batch evenly among all the processes. In this case, the maximum number of processes that can participate in the training is bounded by the number of samples in each mini-batch.

5.2.1. Data Parallelism with Increasing Batch Size

Our proposed training algorithm increases the batch size during the training. There are two possible design choices of parallelization. The first is to employ the number of processes equal to B_s, the starting batch size, so that the number of local training samples per iteration increases as the batch size increases. In this way, the number of inter-process communications for averaging gradients per epoch decreases and it ends up having a higher scaling efficiency. The second option is to start the training with B_s active processes and increase the number of processes as the batch size increases. This design choice exploits the improved degree of parallelism. In this work, we chose the second option to employ as many workers as possible and focus on improving the scaling efficiency by overlapping the communications with the computations.

5.2.2. Overlap of Communication and Computation

In data parallel training, at the end of each iteration, the local gradients are averaged across all processes so that the model parameters are consistent before entering the next iteration. Intuitively, the communication for such task can be simply implemented by an MPI *allreduce* with the sum reduction operator. Many existing parallelizations adopt this approach [**42, 25, 23, 24, 50**]. However, by breaking MPI *allreduce* into MPI *all-to-all* and *allgather*, we can achieve a better overlapping effect for averaging the gradients.

We divide the gradient averaging operations among processes, so they can be performed in parallel. In other words, each process is responsible to calculate the averages for $1/P$ of gradients. The local gradients are first redistributed among processes using an MPI *all-to-all* communication, so each process ends up receiving P subsets of local gradients of size $\frac{G}{P}$ each, where G is the number of gradients in the layer and P is the number of processes. Once the remote gradients are received, the P gradient subsets are element-wisely averaged into a global gradient subset of size $\frac{G}{P}$. The updated gradients are then distributed among all the processes using an MPI *allgather* communication. At the end, all processes obtain the same globally averaged gradients.

Breaking the *allreduce* into an *all-to-all* and an *allgather* has the following advantages. For a given layer k, its *all-to-all* can be overlapped with the computation on calculating gradients for layers $(k-1)\cdots 1$ in the back-propagation phase. Once the gradient sum for layer k is computed, an *allgather* is initiated, which can overlap with the computation of activations for layers $1\cdots(k-1)$ in the feed-forward phase of the next iteration. The *allgather* also can overlap with the gradient summation for layers $(k+1)\cdots L$, where L is the number of layers in the network. Because the number of gradients is usually large for deep CNNs, the cost of element-wise summation is significant enough to provide more room for communication overlap. In addition, our implementation uses the MPI-OpenMP programming model such that each MPI process parallelizes computations using OpenMP within a node. So, our approach enables the gradient summation to employ more compute cores available in each node. If the MPI *allreduce* approach were used, the gradients would be summed by the MPI process on a single core, losing the advantage of OpenMP multi-threading.

5.2.3. Multi-threading Implementation

We allocate one MPI process per compute node and use OpenMP on each process to utilize all the compute cores available in a node. For the matrix operations, we use Intel MKL library which efficiently utilizes KNL cores. We employ a POSIX thread per compute node to handle all the MPI communication calls. By making MPI calls in the communication-dedicated thread, we explicitly forces the overlap of the computation and the communication. The communication thread makes blocking MPI communication function calls. MPI standard defines the progress rule for asynchronous communications, but MPI implementation is free to choose whether to delay the operations till the complete functions, such as `MPI_Wait` and `MPI_Test` [**84, 85, 86**]. In some MPI implementations, we found that their asynchronous communications start only when `MPI_Wait` or `MPI_Test` is called. Due to this finding, we chose to use a communication-dedicated thread over asynchronous MPI communications.

For multi-threading management, we use POSIX thread utilities for communication between the main and communication threads. Once the gradients are computed at each layer, the main thread registers a communication request to a shared queue and sends a signal to the communication thread. Then, the communication thread receives the signal and picks a request in a first-come-first-served manner to perform the communication. Once the communication completes, the communication thread sends back a signal to the main thread to notify that the requested communication is finished. This mechanism is implemented using a pthread conditional variable and a pthread mutex. Note that contention to the mutex lock may only occur between the main and communication thread, without any OpenMP threads involved. To avoid the context switching and possible cold cache for the communication thread, we pin the communication thread on a physical core to prevent it from migrating to a different core.

5.3. Related Works

Recently, a few studies have shown that a large batch size can be used for classification tasks without much loss in accuracy [**50, 47, 23, 24**]. Layer-wise Adaptive Rate Scaling (LARS) proposed in [**47**] adjusts the learning rate in a layer-wise way based on the magnitude of the gradients. Our proposed training strategy can be applied to the training with LARS independently. In [**47**], the authors proposed a parameter update rule based on momentum SGD. Our proposed method does not affect the parameter update rule. By applying LARS at line 6 in Algorithm 3, our training strategy and LARS can be employed together without affecting each other. Therefore, when the training with LARS yields a low accuracy because of the batch size larger than the problem-dependent threshold, our proposed method can be employed and a higher accuracy can be expected.

Adaptive batch size approaches have also been proposed in [**53, 75, 78**]. However, these adaptive batch methods commonly control the gradient noise scale by adjusting batch size. In other words, the small batch size should be used for a sufficient number of epochs to produce a good accuracy and it significantly lowers the degree of parallelism. In [**54**], the authors adjust the batch size and learning rate together to increase the batch size without accuracy loss. However, their approach adjusts them based on a pre-defined schedule which should be tuned by users, making it less practical.

5.4. Performance Evaluation

We evaluate the proposed parallel neural network training strategy using two image regression applications and a popular classification benchmark. We use EDSR [**35**] for image super-resolution with DIV2K [**77**] dataset and image restoration with Phantom [**87**] dataset. DIV2K is dataset from NTIRE2017 Super-Resolution Challenge [**77**], which contains 800 high-quality 2K resolution pictures. Phantom is a randomized version of the classical Shepp-Logan phantom [**87**], where orientation, shape and size of each of the ten ellipsoids are randomized. Phantom has 1600 training images and the size of each image is 256×256. For classification experiments, we use ResNet20 [**33**] and Cifar10 dataset. CIFAR10 has 50,000 3-channel training images and each image size is 32×32.

Our experiments were carried out on Cori, a Cray XC40 supercomputer at National Energy Research Scientific Computing Center (NERSC). Each compute node contains an Intel Xeon Phi Processor 7250 that has 68 cores with support for 4 hardware threads each (maximum 272 threads per node). AVX-512 vector pipelines with a hardware vector length of 512 bits are available at each node. The system has the Cray Aries high-speed interconnections with 'dragonfly' topology.

We compare the performance of our proposed training algorithm with three different training methods. First, we compare our training algorithm with the best-tuned fixed batch size training. Second, as a representative fixed large batch size method, we compare our training algorithm with the linear scaling rule in [**50**]. Finally, we also compare with the adaptive batch size approach proposed in [**75, 54**]. The authors in [**75**] proposed to swap the learning rate decay schedule and the batch size schedule. Similarly, the authors in [**54**] used a pre-defined schedule for increasing the batch size.

5.4.1. Image Regression Experiments

Super-resolution is one of the classic computer vision problems, which aims to recover high-resolution images from low-resolution images [**79, 80, 81, 82**]. Recently, many CNNs have been designed for super-resolution, such as VDSR [**34**], DRRN [**83**], and EDSR [**35**]. Image restoration is another representative image regression problem which aims to recover original images from noisy images. Many existing works use CNNs for image denoising or compression artifact removal [**79, 80, 81, 82**]. As we mentioned earlier, these applications typically use a small batch size between $16 \sim 64$ that gives the best accuracy. So, considering the ever-increasing available data size, it is crucial to improve the degree of parallelism by enabling the large batch size without a significant loss in accuracy.

We compare the proposed training algorithm with the same three other training methods. Table 5.1 and Table 5.2 show the training configurations for super-resolution and

Table 5.1. Training configurations for DIV2K training

configurations	batch size (b)	learning rate (μ)	warmup
best batch size	16	0.0001	-
linear scaling rule	256	0.0016	gradual (5 epochs)
fixed μ, adaptive b	$64 \sim 256$	$0.0004 \sim 0.0016$	-
Proposed method	$64 \sim 256$	$0.0004 \sim 0.0016$	-

Table 5.2. Training configurations for Phantom training

configurations	batch size (b)	learning rate (μ)	warmup
best batch size	16	0.0001	-
linear scaling rule	128	0.0008	gradual (5 epochs)
fixed μ, adaptive b	$32 \sim 128$	$0.0002 \sim 0.0008$	-
Proposed method	$32 \sim 128$	$0.0002 \sim 0.0008$	-

image restoration experiments, respectively. We use Adam for both applications. All the hyper-parameters were set to the same values as used in [**35**]. We use Peak Signal-to-Noise Ratio (PSNR) as the accuracy metric. PSNR measures the degree of similarity of the estimated image to the original image.

When adjusting the batch size and the learning rate in Algorithm 3 at line 12, we double them together after every K epochs because such slow increment prevents the training from diverging. Note that different increasing factors can be applied to the batch size and the learning rate such as root scaling rule depending on the problem.

For the super-resolution experiments, we randomly extract a 48×48 patch from each training image to generate mini-batches. The stop condition of the training is when the validation PSNR is not increased more than 0.1 dB for 50 consecutive epochs. The left chart in Figure 5.6 shows the DIV2K θ curve in the first 15 epochs. The θ peaks between $9^{th} \sim 10^{th}$ epoch. We use $K = 10$ for our proposed adaptive batch size method. We also set $B_s = 64$ since the batch size larger than 64 shows a significantly lower l2-norm. The number of DIV2K training images is 800 and 256 is the maximum power of 2 which allows more than one parameter update per epoch. So, we set $B_m = 256$.

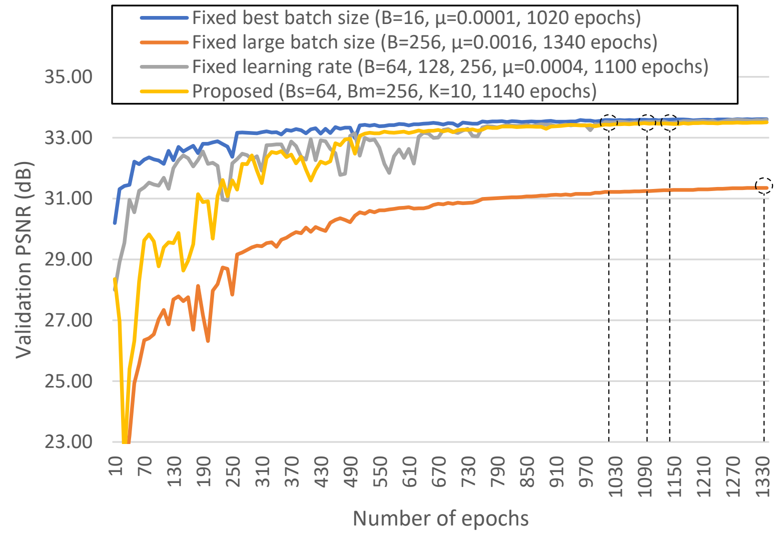


Figure 5.4. Comparison of learning curves of EDSR training on DIV2K dataset among various training strategies. The proposed training method achieves an accuracy almost the same as that of the best-tuned fixed-size method.

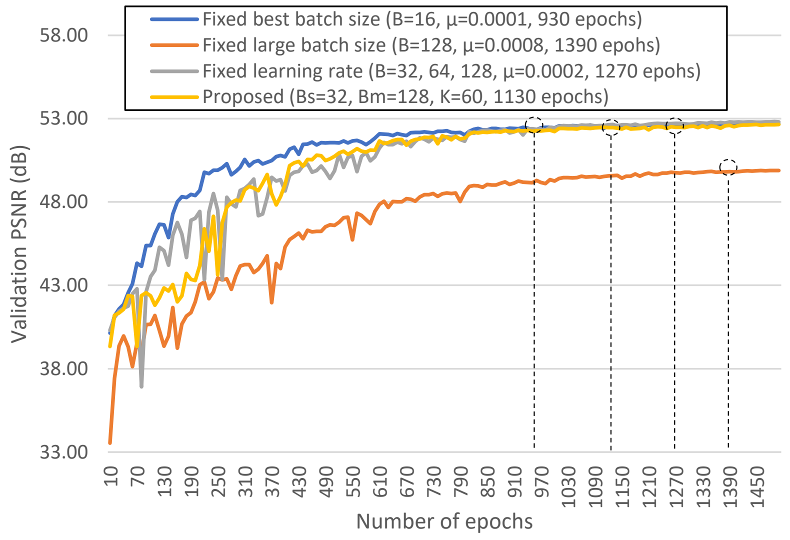
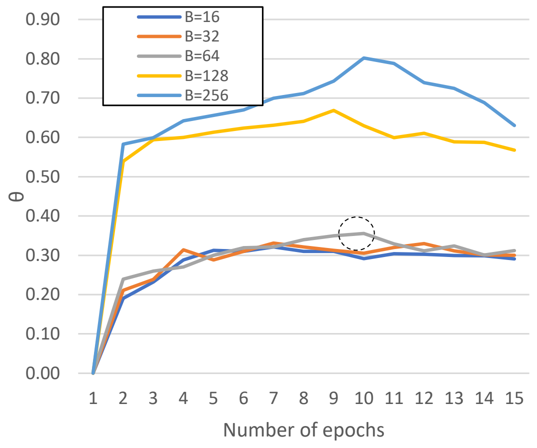


Figure 5.5. Comparison of learning curves of EDSR training on Phantom dataset among various training strategies. The proposed training method achieves an accuracy comparable to the best-tuned fixed-size method, i.e. using $B = 16$.

Figure 5.4 compares the DIV2K learning curves among the four training methods whose configurations are given in Table 5.1. The training with our method converges in 1100 epochs achieving a PSNR of 33.49 dB. The fixed best batch size training converges in 1020 epochs and achieves a PSNR of 33.59 dB while the large batch size training with linear scaling rule converges in 1340 epochs achieving a PSNR of 31.35 dB. The adaptive batch method with a fixed learning rate converges in 1140 epochs achieving a PSNR of 33.51 dB. Note that we calculate 1-crop validation PSNR during the training due to the significant evaluation time. So, the results can be slightly different from reported in [**35**].

For image restoration experiments, we use a modified EDSR which has 16 residual blocks and 32×32 input data size. Figure 5.5 compares Phantom dataset validation accuracy among all the training methods whose configurations are given in Table 5.2. We found that the maximum stable batch size, B_s for Phantom is 32. The right chart in Figure 5.6 shows that θ peaks between $55^{th} \sim 65^{th}$ epoch ($K = 60$). Note that the training with the linear scaling rule failed to converge in a reasonable number of epochs when the batch size is larger than 128. So, to compare with other methods, we also set $B_m = 128$.

Our training strategy achieves a PSNR of 52.47 dB in 1130 epochs. The best-tuned batch size training converges in 930 epochs achieving a PSNR of 52.47 dB and the large batch size training with linear scaling rule converges in 1390 epochs achieving a PSNR of 49.84 dB. The adaptive batch size with a fixed learning rate training achieves a PSNR of 52.51 dB in 1270 epochs. In both super-resolution and image restoration experiments, our proposed training strategy provides a comparable convergence accuracy to the best-tuned batch size training. The performance results demonstrate that the proposed adaptive



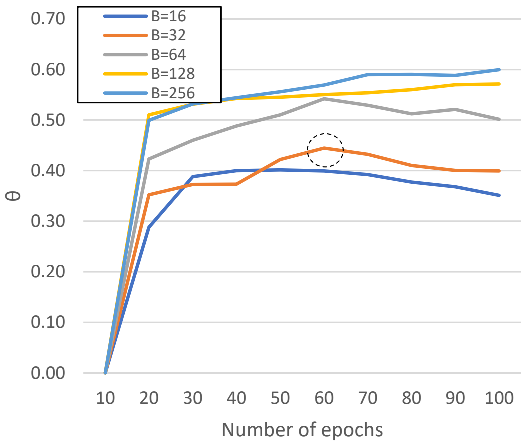


Figure 5.6. The θ curves with varying batch sizes for EDSR training on DIV2K (left) and a variant of EDSR training on Phantom (right). For DIV2K, we chose $B_s = 64$ and its θ peaks at $9^{th} \sim 10^{th}$ epoch. For Phantom, we chose $B_s = 32$ and its θ peaks at $55^{th} \sim 65^{th}$ epoch.

batch size and learning rate method allows to use a large batch size for as many epochs as possible without a significant loss in generalization performance.

We also present the strong scaling performance to verify the effectiveness of the proposed methods. Note that we do not compare the performance against linear scaling rule methods, as they yield a significantly lower accuracy. We consider a direct comparison as unfair between two methods that produce a significantly different accuracy. Figure 5.7 shows the end-to-end training time (left) and the speedup (right) of EDSR training on DIV2K dataset. The best batch size training achieves a speedup of 9.54 using 16 nodes. The adaptive batch size with a fixed learning rate achieves a speedup of 71.40 using up to 256 nodes. Our proposed method achieves a maximum of speedup 81.71 and can run on up to 256 nodes. Figure 5.8 shows the performance of the modified version of EDSR training on Phantom dataset. The best batch size training achieves a speedup of 5.64 using 16 nodes. The adaptive batch size with a fixed learning rate achieves a speedup

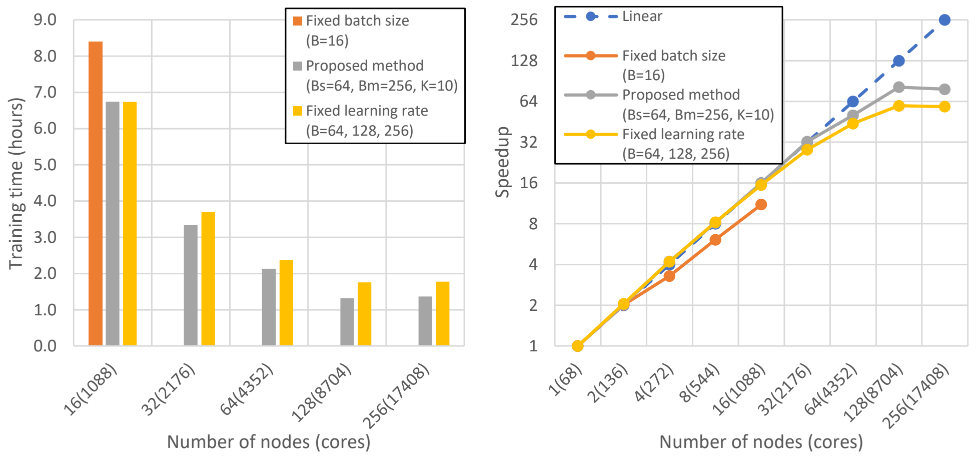


Figure 5.7. Strong scaling of EDSR training on DIV2K dataset: end-to-end training time (left) and speedup (right). We used $B_s = 64$, $B_m = 256$, and $K = 10$. Our method can use more compute nodes beyond 16 and up to 256, while 'fixed best batch size' method can only run up to 16 nodes, limited by the batch size of 16. 'fixed learning rate' has a longer execution time than our method due to the long period of training with small batch sizes. All the three approaches achieve a similar accuracy (33.59 dB / 33.49 dB / 33.51 dB).

of 30.79 using 128 nodes. Our proposed method achieves a maximum of speedup 42.96 using 128 nodes. The experimental results clearly show the advantage of being able to increase the batch size. Compared to the adaptive batch size with a fixed learning rate training, our method enables to use small batch sizes for fewer epochs, which results in achieving a shorter training time as well as a higher speedup.

5.4.2. Performance Analysis

Communication Cost Analysis — The main reason for the speedup saturation is due to the increasing 'blocked' time, as shown in Figure 5.9. In a typical strong scaling result, the communication cost becomes higher and the per-process computation reduces, as the number of processes increases. When the communication is not entirely overlapped with

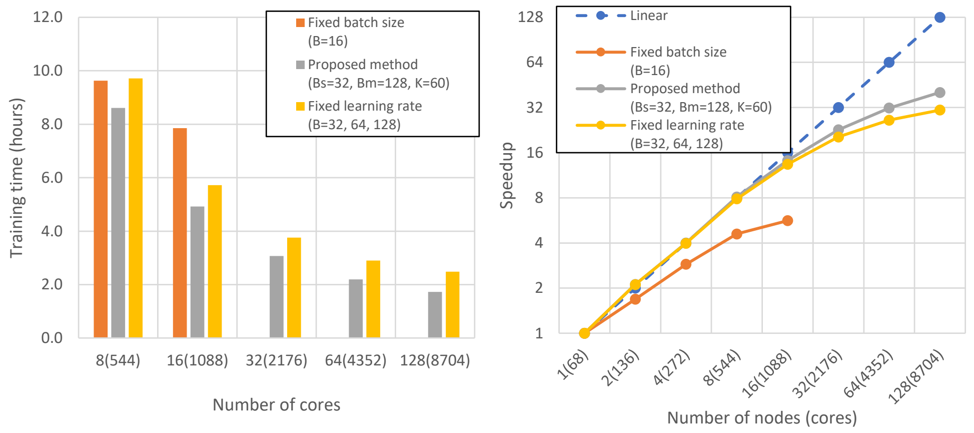


Figure 5.8. Strong scaling of EDSR training on Phantom dataset: end-to-end training time (left) and speedup (right). For our proposed method, we used $B_s = 32$, $B_m = 128$, and $K = 60$. Our method can use more compute nodes beyond 16 and up to 128, while 'fixed best batch size' method can only run on up to 16 nodes, limited by the batch size of 16. All the three approaches achieve a similar accuracy (52.47 dB / 52.47 dB / 52.51 dB).

the computation, the main thread is blocked until the communication thread finishes the transfer of data required by the main thread. For instance, when the number of processes is 128, the communication time measured at the communication thread, 'comm', grows to be similar to the main thread's computation time and 'blocked' starts to become significant. When the number of processes increases to 256, such effect becomes even more significant.

Computation Cost Analysis — Another reason for the speedup saturation is that the computation time is not linearly reduced as the number of processes increases. From the right chart of Figure 5.9, we observe that the computation time ('comp ($B = B_m$)') does not linearly decrease starting from 64 nodes. First, the gradient summation takes a constant amount of time regardless of the number of processes. When averaging the

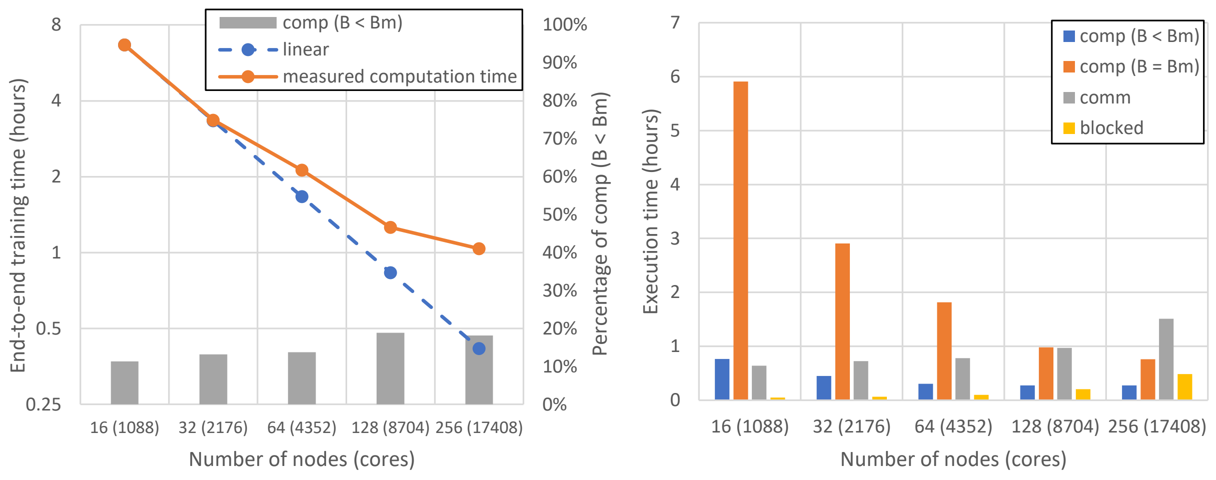


Figure 5.9. The left is the computation time of EDSR training and the percentages of process underutilized time. The right is the training timing breakdown. These results correspond to the training shown in Figure 5.7.

gradients, every process sums P gradient subsets and each is of size $\frac{G}{P}$. Thus, the computation cost for the summation is constant regardless of the number of processes. As the number of processes increases, this computation takes a larger portion of the total time. Second, when the volume of data assigned on each process is not large enough, the kernel operations, such as matrix multiplications, will not fully utilize the computation power. For example, we found that the activation computation at a convolution layer of EDSR for one sample takes ~0.0026 seconds on a single KNL node while the same operation for two samples takes only ~0.0038 seconds. It indicates that the hardware is underutilized when the workload is not sufficiently large, a well-known effect for KNL CPUs [**88, 89**]. Therefore, we suggest to assign each process at least two training samples per iteration. Third, when the batch size is smaller than the number of processes, our method replicates the training on $\frac{P}{B}$ process groups (or equivalently $(P - B)$ number of processes sitting idle). The left chart of Figure 5.9 also shows the percentage of computation time before the training reaches B_m from the end-to-end training time.

5.4.3. Image Classification Experiments

To verify that the proposed training method generally works for various applications, we also perform image classification experiments. We train ResNet20, one of the most popular deep CNN models, on CIFAR10 dataset. Table 5.3 summarizes the training configurations. We decay the learning rate after 80 and 120 epochs with a factor of 0.1. We found the maximum stable batch size for CIFAR10 is 256 ($B_s = 256$) and set $B_m = 2048$ which is the largest power of 2 smaller than 5% of the entire training samples. For the batch size of 256, θ peaks at 5^{th} epoch as shown in Figure 5.2, so we set $K = 5$. For the adaptive batch training with a fixed learning rate, we fixed the learning rate to 0.2 and doubled the batch size from 256 after every 30 epochs so that the batch size ends up reaching 2048. Figure 5.10 compare the training cost curves (left) and the validation accuracy curves (right) among various training methods. Our algorithm achieves a convergence accuracy which is $< 1\%$ lower than that of the best-tuned small batch training ($91.51\% \pm 0.3$ and $90.79\% \pm 0.2$). Both the fixed large batch training and the adaptive batch training with a fixed learning rate effectively minimize the cost function, however they significantly degrade the generalization performance.

The validation accuracy comparison in Figure 5.10 verifies that the proposed adaptive batch size and learning rate control methods effectively increase the batch size without a significant loss in accuracy for classification problems as well. Before the first learning rate decay step, the learning curve fluctuates due to the increasing learning rate. However,

Table 5.3. Training configurations for CIFAR10 training

configurations	batch size (b)	learning rate (μ)	warmup
best batch size	128	0.1	-
linear scaling rule	2048	1.6	gradual (5 epochs)
fixed μ, adaptive b	$256 \sim 2048$	$0.2 \sim 1.6$	-
Proposed method	$256 \sim 2048$	$0.2 \sim 1.6$	-

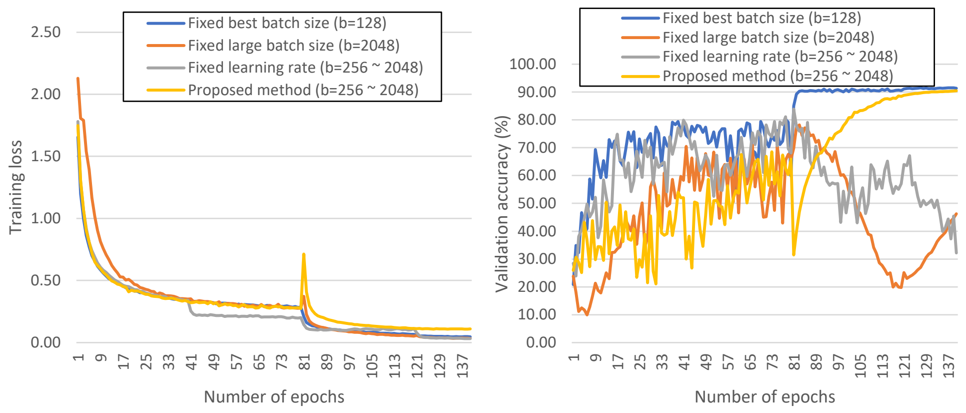


Figure 5.10. Training loss (left) and validation accuracy (right) comparison for ResNet20 training on CIFAR10. The fixed large batch training and the adaptive batch training with a fixed learning rate well minimize the training loss while they significantly degrade the validation accuracy. Our proposed method achieves a comparable accuracy to the best-tuned small batch training.

the validation accuracy increases dramatically after the learning rate is adjusted at 80^{th} epoch by the proposed method.

Figure 5.11 presents the strong scaling performance. We stopped scaling up when the execution time increases. The training with the best-tuned batch size (128) achieves a speedup of 3.49 only due to the high ratio of communication to computation. The adaptive batch training with a fixed learning rate achieves the maximum speedup of 20.01 on 128 nodes. Our proposed method achieves a speedup of 64.64 using 256 nodes thanks to the early increase of the batch size.

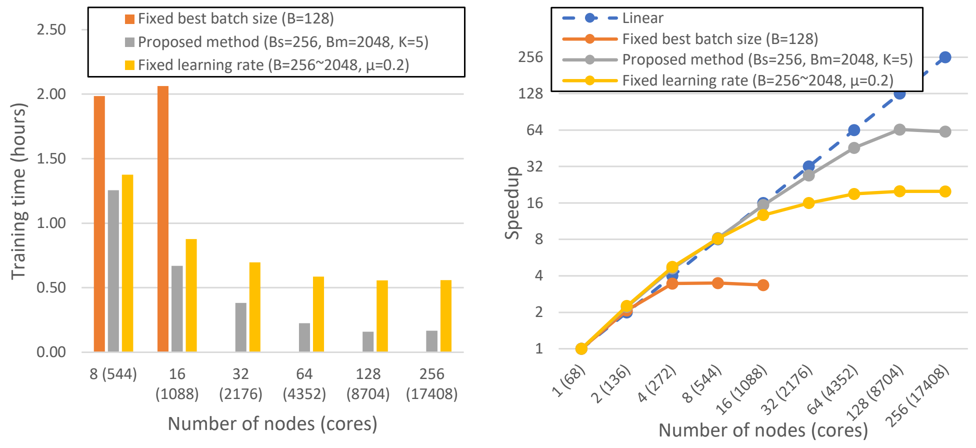


Figure 5.11. The end-to-end training time (left) of ResNet20 on CIFAR10 and speedup (right) comparison. We stopped scaling when the execution time increased. The proposed method out performs the others with a large margin.

5.5. Discussion

In this Chapter, we have discussed how to make a good trade-off between the degree of parallelism and the convergence rate by adaptively adjusting the mini-batch size at run-time. We have proposed a metric for estimating the sharpness of the model parameters, and then designed an adaptive batch size adjustment method based on the proposed estimator. Our experimental results demonstrate that the proposed adaptive batch size training strategy effectively improves the degree of parallelism for both regression and classification problems.

One interesting insight from this research work is that the parallel training method should not consider the convergence rate of the training loss as the top-priority criterion. Even if a training method provides a convergence rate faster than the other methods, if the model provides a poor generalization performance, such a training method would be considered of no use for researchers or practitioners. The major difference between our adaptive batch size adjustment method and the existing adaptive methods is that ours adjusts the batch size based on the estimated sharpness of the model while the others focus on the theoretical convergence rate of the training loss.

Unfortunately, the relationship between the hyper-parameters and the generalization performance of the model has not been well studied. Some researchers proposed a few practical method to 'predict' the generalization performance of a model [**90, 91, 92**]. However, these works discussed how to evaluate the generalization performance of a given model rather than how to improve the generalization performance during training. In contrast, our study aims to achieve a good validation accuracy by adjusting the hyper-parameters at run-time. The proposed distance metric enables to estimate the generalization performance of a model based on the external observations (loss value and the distance between the initial model and the current model) rather than understanding the internal behaviors of the network such as SGD dynamics or theoretical convergence rate. We believe that this novel approach to better understand the training is a valuable exploratory research direction.

CHAPTER 6

Conclusion and Future Work

In this thesis, we presented parallelization strategies that all aim to improve the scalability of synchronous data parallel training of neural networks. First, we presented how to effectively overlap the communications with the computations using a communication-dedicated thread in Chapter 3. Second, we proposed a communication-efficient gradient computation algorithm for data parallel training in Chapter 4. Finally, we explained how to adaptively increase the mini-batch size at run-time in order to improve the degree of parallelism in Chapter 5. All these three research works successfully addressed different factors that hinder the scalable parallel training of neural networks.

Scalable deep learning methods are essential to tackle important large-scale real-world problems. Even if the training scales up very well, however, if the model accuracy is significantly dropped, such parallelization strategies will be considered to be of no use. Therefore, parallel training algorithms should pursue both the scalability and the classificaiton/regression performance. In this thesis, we narrowed down the scope of the discussion to 'synchronous parallel' training based on data parallelism. Thus, we could fully focus on how to improve the scalability by reducing the communication cost without losing the accuracy. However, such a strict synchronous training may not be required in many real-world applications. By partially relaxing the restriction of the synchronous parameter updates, one can consider a practical trade-off between the model accuracy/convergence rate and the scalability. For instance, allowing a certain degree of asynchrony at a subset of model layers can be an interesting research topic.

Commonly, deep learning is still considered to be premature. Many researchers call deep learning ‘black box’ algorithm due to its poor interpretability. We believe that our research works have contributed to better understanding the internal behaviors of neural networks. Our proposed parallelization strategies extract useful information from the neural networks, such as data dependency existing in the networks and the impact of the batch size on the sharpness of the minimizer, and then fully utilize it to improve the scalability of parallel training. Therefore, the insights from our parallel training methods can be generally used to design any parallel deep learning methods expecting a better scalability.

We conclude this study with a few descriptions of potential future works.

local SGD with periodic model averaging – Despite the promising scaling performance of synchronous parallel training achieved in this study, fully utilizing extremely large-scale HPC platforms for deep learning applications still remains as an unanswered questions. We already have discussed that the large batch size can adversely affect the generalization performance of neural networks. Recently, local SGD with periodic model averaging has been highlighted thanks to its less frequent communications per epoch. In addition to the cheaper communication cost, it has also been observed that local SGD allows to use a larger effective batch size without losing the accuracy. Since multiple local models explore the parameter space independently of each other until they are averaged, the global model becomes to have a higher degree of noise in their parameter updates, and thus the better generalization performance. The existing local SGD studies tend to focus on adjusting the averaging interval only. However, the number of workers is also a critical factor that affects not only the convergence properties but also the generalization performance. Especially, in order to successfully apply local SGD to real-world applications, the impact of the number of workers on the generalization performance should be clearly explained.

I/O-efficient training – In real-world scientific applications that have an extremely large amount of data, the I/O time can take up a large portion of the iteration time making the compute resources idle. The size of each training sample decides the per-iteration I/O cost. For example, Cosmology dataset for dark matter distribution analysis consists of training samples of size 48 MiB [**23**]. So, reading such large samples can take up a large portion of iteration time degrading the scaling performance. One promising solution is to overlap the I/O time with the training time. By employing I/O-dedicated thread, the data for the next iterations can be pre-fetched. Such an asynchronous I/O time can be hidden behind the computation time, and thus a shorter iteration time can be expected. TensorFlow already supports the data pre-fetching, however, the feature is implemented under an assumption that one batch is read after another. Motivated by our own communication overlapping strategy discussed in this thesis, the I/O thread may read several batches at once to improve the I/O performance.

The efficiency of I/O overlapping strongly depends on the hardware configurations. In practice, it is very common to use accelerators such as GPUs for faster training. If the training runs on GPU-based systems, the ratio of I/O to computation will be much higher than that on CPU-only systems. Therefore, a larger portion of iteration time can be reduced if the I/O time is effectively overlapped with the computation time. We believe that, considering the ever-increasing available experimental data, an I/O strategy that generally works on different systems will make a huge impact on many real-world scientific applications.

Relaxing synchronization – In our research works, we also found that the communication cost could be dramatically reduced by slightly relaxing the synchronization requirement. It has already been well shown in many previous works that a low degree of asynchrony in the model parameters does not much affect the accuracy. To the best of our knowledge, most of the parallel training studies are under an assumption that the entire model parameters are considered as a single cost function. However, considering the non-linearity between layers caused by the non-linear activation functions such as ReLU, the asynchronous training at one layer may not much affect the other layers. Therefore, relaxing synchronization of a part of model parameters can lead to a practical trade-off between the scalability and the model accuracy. The followed important questions are how many parameters can relax the synchronization and how much the synchronization can be relaxed while maintaining the model accuracy. By answering these questions, deep learning applications can enjoy the communication-efficient partially-asynchronous parallel training while achieving a good model accuracy.

References

[1] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[3] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[4] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.

[5] Lei Zhang, Shuai Wang, and Bing Liu. Deep learning for sentiment analysis: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4):e1253, 2018.

[6] Norjihan Abdul Ghani, Suraya Hamid, Ibrahim Abaker Targio Hashem, and Ejaz Ahmed. Social media big data analytics: A survey. *Computers in Human Behavior*, 101:417–428, 2019.

[7] Javier Ruiz-del Solar, Patricio Loncomilla, and Naiomi Soto. A survey on deep learning methods for robot vision. *arXiv preprint arXiv:1803.10862*, 2018.

[8] Dimitri Bourilkov. Machine and deep learning applications in particle physics. *International Journal of Modern Physics A*, 34(35):1930019, 2019.

[9] Sarah Webb. Deep learning for biology. *Nature*, 554(7693), 2018.

[10] Jonathan Schmidt, Mário RG Marques, Silvana Botti, and Miguel AL Marques. Recent advances and applications of machine learning in solid-state materials science. *npj Computational Materials*, 5(1):1–36, 2019.

[11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

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

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

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

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[16] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Advances in neural information processing systems*, pages 685–693, 2015.

[17] Guojing Cong, Onkar Bhardwaj, and Minwei Feng. An efficient, distributed stochastic gradient descent algorithm for deep-learning applications. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 11–20. IEEE, 2017.

[18] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-sgd: A decentralized pipelined sgd framework for distributed deep net training. In *Advances in Neural Information Processing Systems*, pages 8045–8056, 2018.

[19] Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5693–5700, 2019.

[20] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[21] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, pages 1299–1309, 2018.

[22] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[23] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J Pennycook,

et al. Cosmoflow: Using deep learning to learn the universe at scale. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 819–829. IEEE, 2018.

[24] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 649–660. IEEE, 2018.

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

[26] Sam Ade Jacobs, Brian Van Essen, David Hysom, Jae-Seung Yeom, Tim Moon, Rushil Anirudh, Jayaraman J Thiagaranjan, Shusen Liu, Peer-Timo Bremer, Jim Gaffney, et al. Parallelizing training of deep generative models on massive scientific datasets. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2019.

[27] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[28] Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)*, 53(1):1–37, 2020.

[29] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[30] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[31] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[32] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[34] Jiwon Kim, Jung Kwon Lee, and Kyoung Mu Lee. Accurate image super-resolution using very deep convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1646–1654, 2016.

[35] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 136–144, 2017.

[36] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[37] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[38] Tjalling J Ypma. Historical development of the newton–raphson method. *SIAM review*, 37(4):531–551, 1995.

[39] Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[40] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[41] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[42] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail 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*, pages 1–11, 2017.

[43] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 571–582, 2014.

[44] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

[45] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[46] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

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

[48] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017.

[49] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

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

[51] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[52] Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Big batch sgd: Automated inference using adaptive batch sizes. *arXiv preprint arXiv:1610.05792*, 2016.

[53] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. *arXiv preprint arXiv:1612.05086*, 2016.

[54] Aditya Devarakonda, Maxim Naumov, and Michael Garland. Adabatch: adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

[55] Sunwoo Lee, Qiao Kang, Sandeep Madireddy, Prasanna Balaprakash, Ankit Agrawal, Alok Choudhary, Richard Archibald, and Wei-keng 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)*, pages 830–839. IEEE, 2019.

[56] Farzin Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Viveck Cadambe. Local sgd with periodic averaging: Tighter analysis and adaptive synchronization. In *Advances in Neural Information Processing Systems*, pages 11080–11092, 2019.

[57] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *arXiv preprint arXiv:1810.08313*, 2018.

[58] Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5693–5700, 2019.

[59] Tao Lin, Sebastian U Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't use large mini-batches, use local sgd. *arXiv preprint arXiv:1808.07217*, 2018.

[60] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 2006.

[61] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

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

[63] Intel. Dgemm, sgemm optimized by intel math kernel library on intel xeon processor, 2020. The official performance benchmark document from Intel published in 2020.

[64] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C.

Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

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

[66] I-Hsin Chung, Tara N Sainath, Bhuvana Ramabhadran, Michael Picheny, John Gunnels, Vernon Austel, Upendra Chauhari, and Brian Kingsbury. Parallel deep neural network training for big data on blue gene/q. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1703–1714, 2017.

[67] Wushi Dong, Murat Keceli, Rafael Vescovi, Hanyu Li, Corey Adams, Elise Jennings, Samuel Flender, Thomas Uram, Venkatram Vishwanath, Nicola Ferrier, et al. Scaling distributed training of flood-filling networks on hpc infrastructure for brain mapping. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 52–61. IEEE, 2019.

[68] Mike Barnett, Lance Shuler, Robert van De Geijn, Satya Gupta, David G Payne, and Jerrell Watts. Interprocessor collective communication library (intercom). In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 357–364. IEEE, 1994.

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

[70] Roger W Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel computing*, 20(3):389–398, 1994.

[71] Rajeev Thakur and W. D. Gropp. Improving the performance of mpi collective communication on switched networks. 11/2002 2002.

[72] Rolf Rabenseifner. A new optimized mpi reduce algorithm, 1997.

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

[74] Alex Gibiansky. Bringing hpc techniques to deep learning. http://research.baidu.com/bringing-hpc-techniques-deep-learning/, feb 2017.

[75] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[76] Lingjiao Chen, Hongyi Wang, Jinman Zhao, Dimitris Papailiopoulos, and Paraschos Koutris. The effect of network width on the performance of large-batch training. In *Advances in Neural Information Processing Systems*, pages 9322–9332, 2018.

[77] Radu Timofte, Eirikur Agustsson, Luc Van Gool, Ming-Hsuan Yang, Lei Zhang, Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, Kyoung Mu 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*, pages 1110–1121. IEEE, 2017.

[78] Stanisław Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Three factors influencing minima in sgd. *arXiv preprint arXiv:1711.04623*, 2017.

[79] Chao Dong, Yubin Deng, Chen Change Loy, and Xiaoou Tang. Compression artifacts reduction by a deep convolutional network. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 576–584, 2015.

[80] Pavel Svoboda, Michal Hradis, David Barina, and Pavel Zemcik. Compression artifacts removal using convolutional neural networks. *arXiv preprint arXiv:1605.00366*, 2016.

[81] Weisheng Dong, Peiyao Wang, Wotao Yin, and Guangming Shi. Denoising prior driven deep neural network for image restoration. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[82] Xiao-Jiao Mao, Chunhua Shen, and Yu-Bin Yang. Image restoration using convolutional auto-encoders with symmetric skip connections. *arXiv preprint arXiv:1606.08921*, 2016.

[83] Ying Tai, Jian Yang, and Xiaoming Liu. Image super-resolution via deep recursive residual network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, page 5, 2017.

[84] Sylvain Didelot, Patrick Carribault, Marc Pérache, and William Jalby. Improving mpi communication overlap with collaborative polling. *Computing*, 96(4):263–278, 2014.

[85] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 52. ACM, 2007.

[86] Ron Brightwell, Rolf Riesen, and Keith D Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *The International Journal of High Performance Computing Applications*, 19(2):103–117, 2005.

[87] Lawrence A Shepp and Benjamin F Logan. The fourier reconstruction of a head section. *IEEE Transactions on nuclear science*, 21(3):21–43, 1974.

[88] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack Dongarra. High-performance matrix-matrix multiplications of very small matrices. In *European Conference on Parallel Processing*, pages 659–671. Springer, 2016.

[89] Kyungjoo Kim, Timothy B Costa, Mehmet Deveci, Andrew M Bradley, Simon D Hammond, Murat E Guney, Sarah Knepper, Shane Story, and Sivasankaran Rajamanickam. Designing vector-friendly compact blas and lapack kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 55. ACM, 2017.

[90] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[91] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pages 6240–6249, 2017.

[92] Yiding Jiang, Dilip Krishnan, Hossein Mobahi, and Samy Bengio. Predicting the generalization gap in deep networks with margin distributions. *arXiv preprint arXiv:1810.00113*, 2018.