Communication-Efficient Parallelization Strategy for Deep Convolutional Neural Network Training

Sunwoo Lee*, Ankit Agrawal*, Prasanna Balaprakash†, Alok Choudhary*, and Wei-keng Liao*
*EECS Department, Northwestern University, Evanston, USA
†Argonne National Laboratory, Lemont, USA
Email: *{slz839, ankit, choudhar, wkliao}@eecs.northwestern.edu, †pbalapra@anl.gov

***Abstract*—Training Convolutional Neural Network (CNN) models is extremely time-consuming and the efficiency of its parallelization plays a key role in finishing the training in a reasonable amount of time. The well-known synchronous Stochastic Gradient Descent (SGD) algorithm suffers from high costs of inter-process communication and synchronization. To address such problems, asynchronous SGD algorithm employs a master-slave model for parameter update. However, it can result in a poor convergence rate due to the staleness of the gradient. In addition, the master-slave model is not scalable when running on a large number of compute nodes. In this paper, we present a communication-efficient gradient averaging algorithm for synchronous SGD, which adopts a few design strategies to maximize the degree of overlap between computation and communication. The time complexity analysis shows our algorithm outperforms the traditional allreduce-based algorithm. By training the two popular deep CNN models, VGG-16 and ResNet-50, on ImageNet dataset, our experiments performed on Cori Phase-I, a Cray XC40 supercomputer at NERSC show that our algorithm can achieve 2516.36× speedup for VGG-16 and 2734.25× speedup for ResNet-50 using up to 8192 cores.**

***Keywords*-Convolutional Neural Network; Deep Learning; Parallelization; Distributed-Memory Parallelization**

I. Introduction

Deep Convolutional Neural Network (CNN) has been used in many applications such as visual recognition [1], [2], speech recognition [3], [4], natural language processing [5], [6] and various scientific applications [7], [8], [9], showing promising learning capabilities. Over time, the networks are becoming deeper and larger in order to fulfill the ever increasing demands on acquiring better classification results [10]. Adding more hidden layers into the network architecture allows the model to be flexible in learning better abstract hierarchical features from the training data. Modern deep CNN models have tens of layers and some of them can have even more than a hundred layers. Furthermore, the amount of available training data also increases in a faster, if not the same, pace. Training a model on such large datasets can take days or even weeks, making it impractical.

Considering the increasing model and data size, parallelizing CNNs is imperative to finish the model training in a reasonable amount of time. However, designing a scalable parallel CNN is challenging due to the inherent nature of sequential data dependency of the training algorithm. One of the most popular algorithms for training CNNs is Stochastic Gradient Descent (SGD) [11]. The algorithm runs iteratively such that the model parameters are updated until it converges to an optimum. Due to the data dependency of the model parameters between any two consecutive iterations, parallel SGD may suffer from an expensive inter-process communication cost.

Recently, researchers have put much effort into improving the scalability of parallel SGD algorithm. The traditional synchronous SGD guarantees the optimal parameter update at the cost of low parallel efficiency due to frequent synchronizations. Asynchronous SGD was developed to address such performance bottleneck, however it does not guarantee convergence unless the number of workers is sufficiently small [12], [13]. An alternative is a hybrid approach that mixes both synchronous and asynchronous algorithms [7]. To deal with the low efficiency issue, large mini-batch training techniques have been proposed in [14], [15]. By increasing the mini-batch size, the computation to communication ratio can be raised as the time spent on parameter updates within each epoch is relatively reduced. The large mini-batch size can adversely affect convergence rate, however the proposed techniques recover it by actively adjusting the learning rate at the run time so that it achieves the similar level of classification result to that of the standard size (relatively small) mini-batch training.

In this paper, we present a parallelization strategy for deep CNN training, which is based on synchronous SGD and data-parallelism. Our parallelization strategy aims to reduce the communication time during the process of each mini-batch by re-designing the gradient averaging algorithm. We relocate the intermediate data across all the workers before computing the gradients so that each worker computes the gradient sums for a distinct subset of the model parameters. Then, the computed gradient sums are aggregated across all the workers using allgather operation. The time complexity of our new communication mechanism is considerably lower than that of the traditional allreduce-based gradient averaging method. The proposed design also enables each worker to update only a subset of model parameters locally so that the computation for parameter update can

978-1-7281-0180-4/18/$31.00 ©2018 IEEE
DOI 10.1109/MLHPC.2018.000-4

scale linearly. Additionally, we overlap the communication with the computation to further improve the performance scalability. The communication time is overlapped with the computation time at other layers in both feed-forward and backpropagation stages and it results in improving the speedup.

To evaluate the proposed algorithms, we conducted image classification on ImageNet[1] dataset, which is a popular and standard benchmark for deep learning, and compare the performance against the traditional allreduce-based approach. We measured the execution time and speedup for training two popular CNN models, VGG-16 [2] and ResNet-50 [16]. All the experiments were performed on Cori Phase-I, a Cray XC40 parallel computer with Intel Haswell nodes. Compared to the allreduce-based approach, our algorithms show significantly improved speedups, achieving up to 2516.36× speedup for VGG-16 and 2734.25× speedup for ResNet-50 when running on up to 8192 cores using the mini-batch size of 256.

The rest of the paper is organized as follows. Section 2 summarizes the definition of CNN and the traditional parallelization strategies. Section 3 describes our parallelization strategy in detail. Section 4 presents the experimental results and the comparison of our proposed parallelization strategy and the existing works. Finally, in Section 5, we conclude our study.

II. Background and Related Work

In this section, we briefly describe background on deep learning and the existing parallelization strategies.

A. Convolutional Neural Network

Convolutional Neural Network (CNN) is a type of artificial neural network which includes convolution layers [17]. The convolution layers enable the model to exploit the spatially-local correlation in the input images by using the local connectivity pattern. The local connectivity pattern is called a filter. Figure 1 illustrates the typical structure of a CNN model. Given the input data at the first layer, a few convolution layers extract the abstract features. Then, a set of fully-connected layers perform the classification based on the information provided by the convolution layers. Each convolution layer can be followed by a pooling layer to ignore less-important information. All the layers except the pooling layers may have different activation functions such as sigmoid, hyperbolic tangent (tanh), or Rectified Linear Unit (ReLU) [18].

B. Mini-Batch Stochastic Gradient Descent

The most popular optimization technique for training CNNs is Mini-Batch Stochastic Gradient Descent (SGD). Mini-Batch SGD is a variant of the traditional SGD [11]. A model is trained on a small subset of the given dataset (known as mini-batch) iteratively until it converges to an optima. The entire dataset is randomly shuffled before partitioned into mini-batches so as to avoid the overfitting.

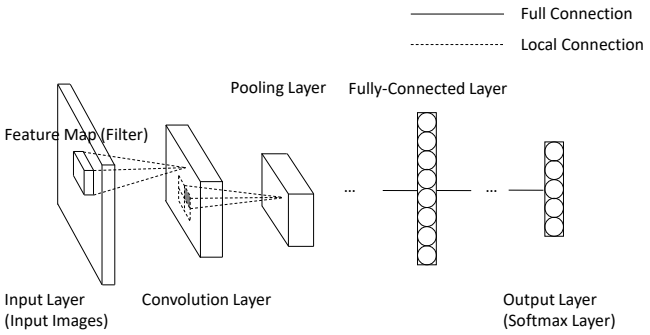

Figure 1: The structure of a CNN model. The model consists of convolution layers, pooling layers, and fully-connected layers.

The training consists of two stages, feed-forward and backpropgation. In the feed-forward stage, the input data is evaluated to compute activations, going through the model layers in a forward direction. Then, in the backpropagation stage, the errors are calculated based on the output activations at the output layer and the errors are propagated in a backward direction to adjust the parameters. Once the activations and errors are computed, the gradient of the cost function with respect to the parameters is computed at each layer. The parameters are subtracted by the averaged gradients multiplied by a learning rate. The activations, errors and gradients are computed using Equation 1, 2, and 3 respectively. Algorithm 1 presents the simplified version of the training algorithm.

$$a_n^l = \sigma\Big(\sum_{i=0}^{|W|-1} w_i^l a_{n+i}^{l-1} + b_n^l\Big) \tag{1}$$

$$e_n^l = \sum_{i=0}^{|W|-1} w_i^{l+1} e_{n-i}^{l+1} \tag{2}$$

$$\Delta w_n^l = \sum_{i=0}^{|X|-|W|} e_i^l a_{n+i}^{l-1}, \tag{3}$$

where a_n^l, b_n^l, and e_n^l are the n^{th} activation, bias, and error in layer l, respectively, w_i^l is an weight on the i^{th} connection between layer l and layer $l-1$, $|W|$ is the number of weights in the filter between the $l-1^{th}$ layer and l^{th} layer, $|X|$ is the number of neurons in the $l-1^{th}$ layer, and σ is the activation function. At the first layer, a_{n+i}^{l-1} is the input data.

We define a few notations to analyze the complexity of algorithm: N is the largest number of neurons among all the layers, L is the total number of layers in the model, M is the number of mini-batches, and K is the size of each mini-batch. Since the maximum feature map size in the convolution layers is N, the complexity of processing

Algorithm 1 Mini-Batch SGD CNN Training Algorithm (M: the number of mini-batches, L: the number of layers

```
for each mini batch m = 0, ...M − 1 do
    Initialize ΔW = 0
    Get the m^th mini batch, D^m.
    for each layer l = 0, ...L − 1 do
        Calculate activations A^l based on D^m.
    for each layer l = L − 1, ...0 do
        Calculate errors E^l.
        Calculate weight gradients ΔW^l.
    for each layer l = 0, ...L − 1 do
        Update parameters, W^l and B^l.
```

each image at a layer is $O(N^2)$. Algorithm trains L layers on MK images. Therefore, the complexity of an epoch of training is $O(LMKN^2)$.

C. Existing Parallelization Strategies

1) Synchronous SGD: Synchronous SGD can be implemented using either model-parallelism or data-parallelism [19]. If a model is distributed on multiple workers and each worker contributes to the assigned partial model only, it is called model-parallelism. On the other hand, if each mini-batch is distributed on multiple workers and the model is trained by the averaged information across all the workers, it is called data-parallelism. In both approaches, the gradients are computed in a synchronous way so that the parameter updates are always optimal. Recently, data-parallelism is prefered over model-parallelism due to the less frequent synchronizations. In model-parallelism, the activations and errors at each layer should be aggregated across all the workers before starting the next layer computation due to the data dependency across the layers. In contrast, in data-parallelism, the gradients are aggregated and summed up at each iteration. The data dependency exists only across two consecutive iterations and it results in having less frequent synchronizations. Many existing works adopt data-parallelism[7], [20], [21], [22].

2) Asynchronous SGD: Asynchronous SGD (ASGD) has been proposed to address the poor scalability issue of synchronous SGD. The most popular asynchronous training algorithm is Downpour SGD [10]. The model parameters are located in a parameter server and all the workers contribute to the shared model parameters in an asynchronous way. Each worker processes a mini-batch and sends the computed gradients to the parameter server. All the workers are assigned with different mini-batches and they work on the assigned mini-batches independently of each other. In this way, the scalability is significantly improved since the workers only perform point-to-point communications with the parameter server. However, due to the asynchrony across the workers, every worker has a certain degree of gradient staleness and it results in lowering the convergence rate.

3) Training with Large Mini-Batch size: Using a large batch size can improve the scalability of parallel SGD algorithm. Since the ratio of the computation to the communication increases, it results in improving the scalability. However, the larger batch size also adversely affects the classification accuracy. To tackle the large batch size, Goyal et al. [15] and You et al. [14] proposed a few techniques which improve the classification results by dynamically tuning the hyper-parameters such as learning rate or momentum factor in a more fine-grained way. The techniques in [15], [14] enable the large batch trainings to achieve the reasonable level of classification results while enjoying the improved scalability.

III. Parallelization Strategy

In this section, we describe our parallelization strategy for deep CNN training. We begin with a high-level description of the communication-efficient gradient averaging algorithm. Then, we analyze the theoretical communication cost and compare it to the optimal cost of the traditional algorithms. Based on the proposed algorithm, we also propose a scalable model parameter update method and a strategy for overlapping computation and communication.

A. Communication-Efficient Gradient Averaging

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.

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 nodes using allreduce operation. Finally, the gradient sums are averaged by multiplying the reciprocal of K to all the elements.

In our gradient averaging algorithm, instead of computing a $N \times N'$ gradient matrix for $\frac{K}{P}$ training samples at each node, we relocate the activations and errors across the nodes and calculate a partial gradient matrix of size $N \times \frac{N'}{P}$ for K training samples at each node. First, the activations are scattered across all the nodes using all-to-all operation. The size of the scattered activation matrix is $K \times \frac{N'}{P}$. Second, the errors are gathered across all the nodes using 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 which is the partial gradient sums for K training samples.

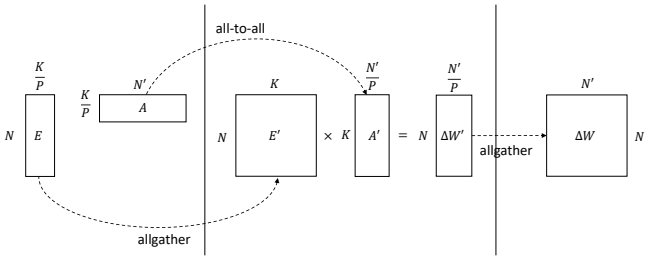

Figure 2: Communication-efficient gradient calculation. Given an error matrix E of size $N \times \frac{K}{P}$ and an activation matrix A of size $N' \times \frac{K}{P}$, E is gathered and A is scattered across all the nodes. Then, the gathered error E' is multiplied by the scattered activation A' to compute the partial gradient $\Delta W'$ 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 ΔW.

Finally, the gradient sums are gathered across all the nodes using allgather operation and each node ends up having $N \times N'$ gradient sums. Figure 2 illustrates the proposed gradient averaging algorithm.

2) Convolution Layers: The standard way of averaging the gradients at a convolution layers is as follows: As shown in Equation 3, 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 nodes using allreduce operation. Finally, the gradients are averaged by multiplying the reciprocal of K to all the elements.

Instead of performing a single allreduce, we sum up the gradients using two communication steps and one computation step. First, we scatter the gradient matrix using all-to-all operation and each node becomes to have P sub-matrices (each of size $\frac{D}{P} \times F$). Second, the P sub-matrices are summed up within each node to have a single sub-matrix of size $\frac{D}{P} \times F$. Finally, the sub gradient sums are gathered across all the nodes using allgather operation. Figure 3 illustrates the proposed approach.

B. Theoretical Cost

In our theoretical cost analysis, we use the cost model used in many previous works [23], [24], [25].

$$T = s\alpha + w\beta, \tag{4}$$

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 [24].

In this discussion, we refer to the theoretical communication cost for the collective communications shown in Table 1. n in the table is the overall data size. Note that the costs are calculated for the long message algorithms in [24], [26]

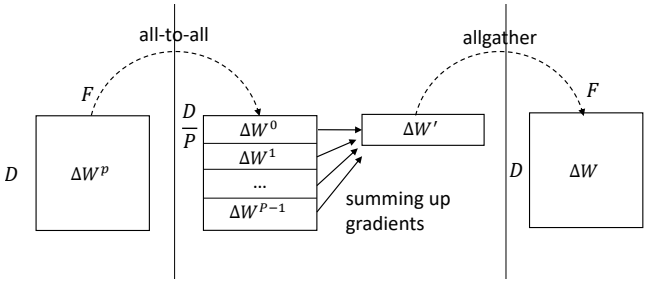

Figure 3: Three-step reduction for gradient averaging at convolution layers. First the local gradients ΔW^p are scattered across all the nodes. The received partial gradient matrices are summed up and then gathered across all the nodes to obtain the entire gradient sums ΔW.

Table I: 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	$P-1$	$n(P-1)/P$

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).

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 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.

$$T_s = (P-1)\alpha + \frac{N'K}{P^2}(P-1)\beta \tag{5}$$

$$T_{g1} = (P-1)\alpha + \frac{NK}{P}(P-1)\beta \tag{6}$$

$$T_{g2} = (P-1)\alpha + \frac{NN'}{P}(P-1)\beta \tag{7}$$

$$T_f = T_s + T_{g1} + T_{g2} \tag{8}$$

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) [27], [24]. The communication costs are calculated as followings.

$$T_{binomial} = log(P)\alpha + log(P)n\beta \tag{9}$$

$$T_{Rabenseifner} = 2log(P)\alpha + 2\frac{P-1}{P}n\beta, \qquad (10)$$

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. We can derive the following condition by comparing the bandwidth terms of Equation 8 and 10. Note that n in Equation 10 is NN'.

$$\frac{N'K}{P} + NK < NN' \qquad (11)$$

If the above condition is satisfied, our gradient averaging method 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.

2) Convolution Layers: As explained, our gradient averaging algorithm for convolution layers consists of three steps, all-to-all operation, accumulating matrices, and allgather operation. The overall communication cost at a convolution layer, T_c, is calculated by the following equation.

$$T_c = 2(P-1)\alpha + 2\frac{DF}{P}(P-1)\beta \qquad (12)$$

Rabenseifner's algorithm is implemented in MPICH using a reduce-scatter operation followed by an allgather operation. Since n in Equation 10 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 sub-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 subsection.

C. Scalable Model Parameter Update

The cost of model parameter update is easily overlooked, however it can be a significant performance bottleneck in parallel CNN training. In our proposed gradient averaging 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 allgather for the updated model parameters. In this way, the computation complexity of parameter update is $O(\frac{NN'}{P})$ without any extra communications.

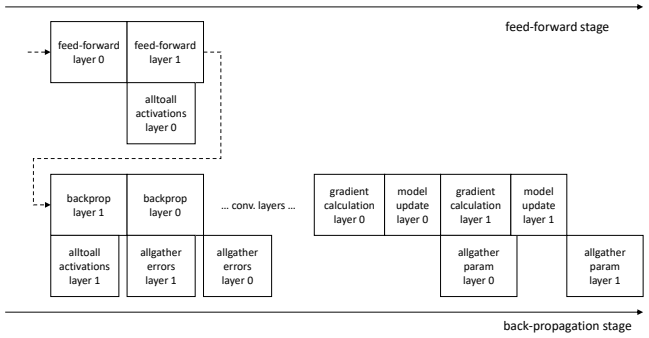

Figure 4: An example of the ideal overlapping of 2 layers such that the computation time at each layer is sufficiently large to overlap the communication time. After the errors are back-propagated through all the fully-connected layers, process the convolution layers first. Then, calculate the average gradients, update the partial model parameters, and post allgathers for the updated model parameters.

In allreduce-based gradient averaging algorithm, 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 [28] 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.

D. 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 averaging algorithm.

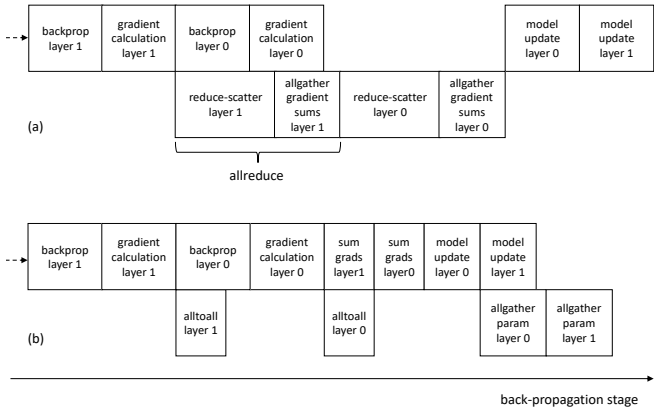

Figure 5: An example of overlapping computation and communication at convolution layers. (a) shows the allreduce approach and (b) shows the proposed two-step communications. (b) enables computation and communication overlaps across layers.

1) Fully-Connected Layers: The proposed gradient averaging algorithm has three communication steps at a fully-connected layer. First, once the activations are ready, an all-to-all 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 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 2 are ready and the gradient sums 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 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 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.

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 5 shows example time-flow charts. Figure 5.(a) is the allreduce-based approach and Figure 5.(b) is the proposed two-step communications for averaging gradients.

IV. Evaluation

In order to evaluate our proposed 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 [29], Intel Caffe [28], PyTorch [30], and Horovod [31]. Most of them adopt the traditional allreduce-based gradient averaging algorithm. Intel Caffe supports a gradient averaging algorithm which uses reduce-scatter and allgather operations. PyTorch and Horovod use ring-allreduce algorithm which utilizes the network bandwidth more efficiently than the other allreduce algorithms [32]. 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 de facto benchmark for deep learning study.

A. 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 [33] 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 [2]. 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 [16]. 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 [2], [16].

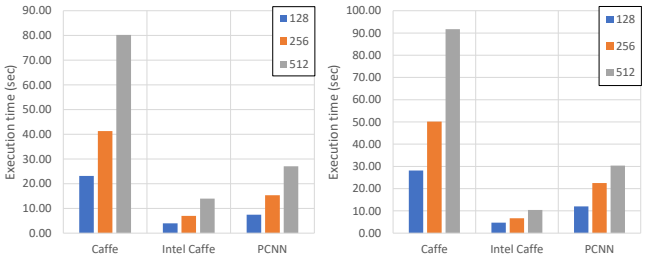

Figure 6: Single-node execution time for processing a single mini-batch. VGG-16 model (left) and ResNet-50 (right) with varying mini-batch sizes. The performance is measured on a Haswell node of Cori.

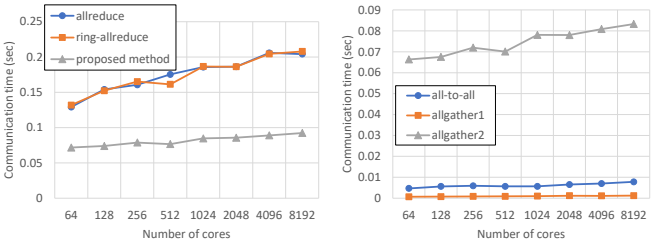

Figure 7: Communication time comparison (left) and communication timing breakdown (right). Our approach is compared with allreduce in MPICH as well as ring-allreduce. The overall data size is 392MB (the gradient size at the first fully-connected layer of VGG-16 model). Our proposed method has a shorter communication time than the other two methods.

We implemented a software framework for deep CNNs in C language. Intel parallel MKL library is used for kernel functions such as matrix operations and all the non-kernel loops are parallelized with OpenMP. We used MPI for the distributed-memory parallelization. To maximize the overlapping, we use POSIX thread library and has a communuication-dedicated thread which calls the blocking MPI functions.

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. 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).

B. Results

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 [28] 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 6 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.

1) Communication-Efficient Gradient Averaging: We first compare our proposed gradient averaging algorithm with the traditional allreduce-based algorithm. 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 as well as ring-allreduce. The reduce-scatter algorithm in MPICH allreduce [26], the circular algorithm in ring-allreduce [32], 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 averaging approaches. Figure 7 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 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 as explained in Section 3.

2) Strong Scaling Results: To evaluate the impact of the proposed algorithms on scalability, we measure end-to-end execution time for processing a single mini-batch and

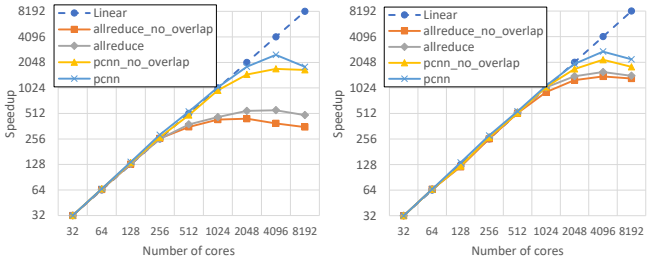

Figure 8: Strong scaling results for VGG-16 (left) and ResNet-50 (right) models. The mini-batch size is 256. 'allreduce' is the traditional allreduce-based data-parallelism and 'pcnn' is the proposed parallelization strategy. The speedups are measured using up to 8192 cores.

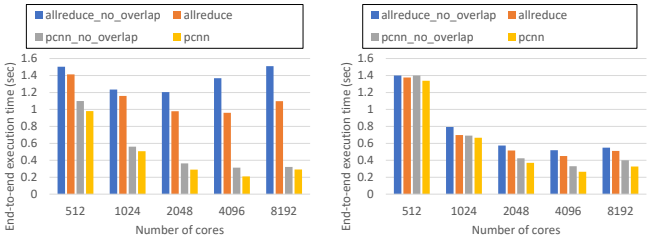

Figure 9: End-to-end execution time for processing a single mini-batch. VGG-16 (left) and ResNet-50 (right). The mini-batch size is 256. The results show that 'pcnn' outperforms the other approaches.

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 not only the backpropagation time but also the feed-forward time at the next iteration. So, we chose the best overlapping strategy among them and reproduced it using allreduce-based averaging algorithm. 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 is posted right after the local gradients are computed at each layer. 'pcnn' is our software framework which uses all the proposed algorithms. Figure 8 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

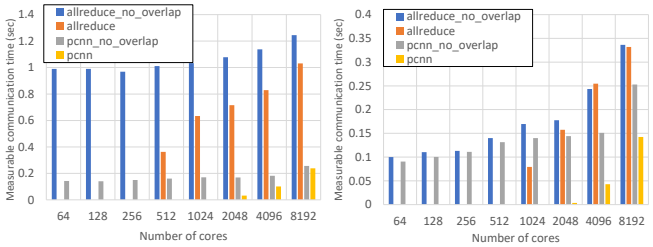

Figure 10: Measurable communication time comparison. VGG-16 (left) and ResNet-50 (right). The mini-batch size is 256. If the communication time is entirely overlapped with the computation time, the measurable communication time would be zero. 'pcnn' always shows lower measurable communication time than that of 'allreduce'.

9 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.

3) Overlapping Computation and Communication: As explained in Section 3, overlapping computation and communication plays a key role in our parallelization strategy. If the computation time is not long enough to hide the entire communication 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, 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 10 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 8, on the right chart, 'pcnn' achieves a linear speedup on 2048 cores. In contrast, 'allreduce' has non-zero measurable communication time on 1024 cores. The results demonstrate that our proposed methods effectively improve the degree of overlap.

C. Discussion

1) 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 averaging algorithm reduces the theoretical communication cost at fully-connected layers. So, if a model has many fully-connected layers, more performance improvement can be expected. In this paper, we used a regular CNN (VGG-16) as well as a residual network (ResNet-50). Compared to the regular CNNs, the residual networks likely have fewer fully-connected layers. In Figure 8, VGG-16 shows a clearer speedup difference between 'pcnn' and ' allreduce' than ResNet-50. Second, the proposed overlapping strategy hides each communication time behind the computation time at other layers. 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 10, ResNet-50 shows a larger difference of measurable communication time between 'pcnn' and 'pcnn_no_overlap' than VGG-16.

2) Large-Batch Training: As briefly introduced in Section 1, a few techniques for large-batch training [15], [14] have been proposed recently. The techniques enable to increase the mini-batch size to up to 32K and it enables to achieve a higher speedup of parallel training. Given a fixed number of training images, the large-mini-batch size reduces the number of iterations to process the entire training dataset. Therefore, the overall number of communications at each epoch is reduced, which results in achieving the higher scalability. In this paper, we aim to improve the scalability in a different way. Given a fixed mini-batch size, our proposed algorithms reduce the communication time within each iteration. If Inequality 11 is satisfied, the proposed gradient averaging algorithm guarantees the lower communication cost at fully-connected layers than that of the allreduce-based approach. So, our approach can also improve the scalability of the large-batch trainings. In addition, our overlapping strategy will further improve the scalability regardless of the mini-batch size. Since the large mini-batch size raises the ratio of computation to communication, a higher degree of overlap can be expected.

V. Conclusion

In this paper, we reported the performance of our proposed algorithms for distributed-memory parallel CNN training. The experimental results demonstrate that the proposed algorithms make CNN training more scalable. The gradient averaging algorithm reduces the theoretical communication cost as well as the computation complexity of the parameter update. Our overlapping strategy further improves the scalability by maximizing the overlap of the computation and the communication in the proposed gradient averaging algorithm. We demonstrated that our proposed algorithms can be applied to not only the standard CNNs but also the more recently proposed residual networks. Since the proposed algorithms are general parallelization techniques for neural networks, we believe they can be applied to any kinds of large-scale neural network trainings in many different applications.

VI. Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. This work is also supported in part by NSF awards CCF-1409601, DOE awards DE-SC0007456, DE-SC0014330, and NIST award 70NANB14H012. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (Argonne). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

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

[3] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on audio, speech, and language processing*, vol. 20, no. 1, pp. 30–42, 2012.

[4] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[5] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[6] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, vol. 1, 2017, pp. 1107–1116.

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

[8] F. Gieseke, S. Bloemen, C. van den Bogaard, T. Heskes, J. Kindler, R. A. Scalzo, V. A. Ribeiro, J. van Roestel, P. J. Groot, F. Yuan *et al.*, "Convolutional neural networks for transient candidate vetting in large-scale surveys," *Monthly Notices of the Royal Astronomical Society*, vol. 472, no. 3, pp. 3101–3114, 2017.

[9] D. Ushizima, C. Yang, S. Venkatakrishnan, F. Araujo, R. Silva, H. Tang, J. V. Mascarenhas, A. Hexemer, D. Parkinson, and J. Sethian, "Convolutional neural networks at the interface of physical and digital data," in *Applied Imagery Pattern Recognition Workshop (AIPR), 2016 IEEE*. IEEE, 2016, pp. 1–12.

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

[11] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[12] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Advances in Neural Information Processing Systems*, 2015, pp. 2737–2745.

[13] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," *arXiv preprint arXiv:1511.05950*, 2015.

[14] Y. You, I. Gitman, and B. Ginsburg, "Large batch training of convolutional networks. arxiv preprint," *arXiv preprint arXiv:1708.03888*, 2017.

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

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

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[19] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[20] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. 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*, 2016, pp. 2592–2600.

[21] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," *arXiv preprint arXiv:1602.06709*, 2016.

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

[23] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, and J. Watts, "Interprocessor collective communication library (intercom)," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*. IEEE, 1994, pp. 357–364.

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

[25] R. W. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," *Parallel computing*, vol. 20, no. 3, pp. 389–398, 1994.

[26] R. Thakur and W. D. Gropp, "Improving the performance of mpi collective communication on switched networks," 11/2002 2002.

[27] R. Rabenseifner, "A new optimized mpi reduce algorithm," 1997.

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

[29] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

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

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

[32] A. Gibiansky. (2017, feb) Bringing hpc techniques to deep learning. http://research.baidu.com/bringing-hpc-techniques-deep-learning/.

[33] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.