2017 IEEE 24th International Conference on High Performance Computing

Parallel Deep Convolutional Neural Network Training by Exploiting the Overlapping of Computation and Communication

Sunwoo Lee, Dipendra Jha, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, IL USA
{*slz839, dkj755, ankitag, choudhar, wkliao*}*@eecs.northwestern.edu*

***Abstract*—Training Convolutional Neural Network (CNN) is a computationally intensive task whose parallelization has become critical in order to complete the training in an acceptable time. However, there are two obstacles to developing a scalable parallel CNN in a distributed-memory computing environment. One is the high degree of data dependency exhibited in the model parameters across every two adjacent mini-batches and the other is the large amount of data to be transferred across the communication channel. In this paper, we present a parallelization strategy that maximizes the overlap of inter-process communication with the computation. The overlapping is achieved by using a thread per compute node to initiate communication after the gradients are available. The output data of backpropagation stage is generated at each model layer, and the communication for the data can run concurrently with the computation of other layers. To study the effectiveness of the overlapping and its impact on the scalability, we evaluated various model architectures and hyperparameter settings. When training VGG-A model using ImageNet data sets, we achieve speedups of 62.97× and 77.97× on 128 compute nodes using mini-batch sizes of 256 and 512, respectively.**

***Keywords*-Convolutional Neural Network; Deep Learning; Parallelization; Communication; Overlapping**

I. INTRODUCTION

Convolutional Neural Networks (CNNs) have shown the state-of-the-art classification accuracies in various applications such as visual recognition [1], [2], speech recognition [3], [4] and natural language processing [5]. Since AlexNet [1] won the ImageNet competition with a large margin over the other traditional approaches, CNN has become one of the most promising machine learning algorithms. It is widely known that deeper network can possess better learning capability [6]. Recently, deep CNNs have been popularly used to solve problems involving large-scale datasets [7], [8]. However, training deep CNNs on large datasets is extremely computing-intensive and takes an enormous execution time.

Parallelizing CNNs has been considered to be challenging due to the inherent sequential nature of Stochastic Gradient Descent (SGD) algorithm used for training. In SGD, the gradient of a cost function is calculated using a small group of input data (typically called mini-batch) and the cost function is minimized by updating the parameters using their gradients [9]. Given a set of mini-batches, the algorithm iteratively updates the parameters using the gradient at each mini-batch and there is a data dependency on the parameters across different mini-batches. Such sequential nature of SGD limits the scalability of the training CNNs.

CNN training can be parallelized across either model-dimension or data-dimension. If a model is split among multiple compute nodes and trained on the same data, it is called model-parallelism. In contrast, if the data is distributed on multiple nodes and the same model is used for training, it is called data-parallelism. Hybrid approaches leveraging both parallelisms also have been proposed [6], [10], [11]. In the hybrid approaches, a small number of nodes are grouped to train a model together using the model-parallelism, and a data set is partitioned to the groups to be processed concurrently exploiting the data-parallelism. Many of them employs master-slave model such that a parameter server controls the parameter updates in a centralized fashion [12], [6], [10], [13]. However, the parameter server eventually leads to performance bottleneck since all the groups have to communicate with the single parameter server. Another popular approach to solve the scaling issue is asynchronous SGD. The asynchronous scheme allows a better scaling at the cost of suboptimal parameter updates. Many asynchronous SGD-based training algorithms have been proposed for better scaling [6], [14], [15].

We address the scaling issue of CNN training algorithm by proposing a novel approach for overlapping communication with computation. In backpropagation algorithm, the gradient of the cost function is calculated with respect to the parameters. Since there is no data dependency between the gradients across different model layers, the communication for exchanging the gradients among all compute nodes can run concurrently with the computations of other layers. The computation and communication overlap is essential to fully utilize the given hardware resources and achieve a high speedup. The existing works mostly focus on the efficient workload distribution for a lower communication cost, while overlapping is largely overlooked. We demonstrate the impact of the overlapping on the scalability of CNN training.

In this paper, we parallelize CNN models using data-

0-7695-6326-0/17/$31.00 ©2017 IEEE
DOI 10.1109/HiPC.2017.00030

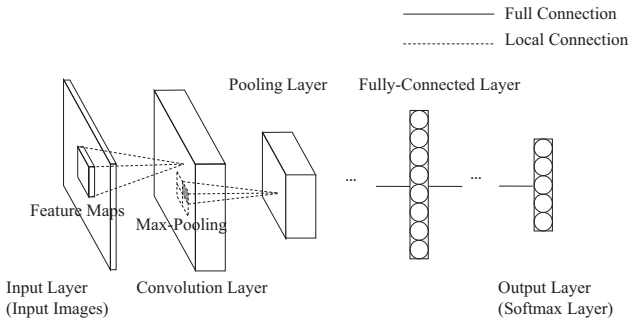

Figure 1: Structure of a CNN. A model consists of convolution layers, pooling layers and fully-connected layers.

parallelism. The models are trained using synchronous SGD such that the result from parallel training is exactly same as the sequential training result. Our parallelization strategy is based on two key techniques to maximize the computation and communication overlap. First, we aggregate all the gradients of parameters into two large chunks and reduce them across all the nodes using asynchronous communications. Based on a data dependency analysis, we maximize the overlap by choosing the optimal size of each gradient chunk. Second, we replicate gradient calculation in a few fully-connected layers. Depending on model architecture, the replicated gradient calculation can significantly reduce the communication cost. It also allows to overlap the communication time with feed-forward time for the next mini-batch. When training VGG-A [7] model, we achieve up to 62.97× and 77.97× speedups using 128 nodes with mini-batch sizes of 256 and 512, respectively. We also explore various model architectures and hyper-parameter settings to evaluate the proposed algorithm and the overlapping methods.

II. Convolutional Neural Network

Convolutional Neural Network (CNN) is a type of artificial neural networ which contains convolution layers [16]. 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 either a feature map or a filter.

A CNN model can contain three types of layers, convolution layer, pooling layer and fully-connected layer. Figure 1 illustrates the typical structure of a CNN model. Given the training images at the first layer, a few convolution layers are connected to the input layer. Each convolution layer can be followed by a pooling layer. Finally, a set of fully-connected layers is at the end of the model. Typically, the last layer uses softmax as the activation function, while all the other layers use Rectified Linear Unit (ReLU) [17]. Various activation functions such as $sigmoid$ or $tanh$ can be used depending on the applications. In Figure 1, the dotted lines indicate the local connections, while the solid lines present the full connections. The local connection is the connection pattern where a neuron takes inputs from a subset of neurons in the previous layer. In contrast, the full connection is the case where a neuron receives inputs from all the neurons in the previous layer. Convolution layers have the local connections, whereas fully-connected layers have the full connections only.

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

```
1:  for each mini batch m = 0, ...M − 1 do
2:      Initialize ΔW = 0
3:      Get the m^th mini batch, D^m.
4:      for each layer l = 0, ...L − 1 do
5:          Calculate activations A^l based on D^m.
6:      for each layer l = L − 1, ...0 do
7:          Calculate errors E^l.
8:          Calculate weight gradients ΔW^l.
9:      for each layer l = 0, ...L − 1 do
10:         Update parameters, W^l and B^l.
```

A. Mini-Batch SGD CNN Training Algorithm

A CNN model is trained using backpropagation algorithm[18]. The most popular optimization technique for the backpropagation is Stochastic Gradient Descent (SGD) [19]. In SGD, a model is trained on a small subset of the given dataset (known as mini-batch), which is typically on the order of $10 \sim 512$ for visual recognition tasks.

The training consists of two stages, feed-forward and backpropagation. In feed-forward, the activations are calculated using the input images and propagated in a forward direction. In backpropagation, based on the activations at the last layer, the errors are calculated and propagated in a backward direction. The activations and errors are calculated using Equation 1 and 2.

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

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

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 a feature map, and σ is the activation function. Once the errors are computed, the gradients of all parameters are calculated by Equation 3.

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

where $|X|$ is the number of neurons in the $l-1^{th}$ layer. In the first layer, a_{n+i}^{l-1} is a pixel of the input image.

Algorithm 2 Data-Parallel CNN 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** until communications for A_p^{l-1} and E_p^l, posted in iteration $m-1$, are finished (line 12 and 25).
9: Calculate weight gradients $\Delta W_p^{[l:l+f]}$ for D^m and update the corresponding weights $W_p^{[l:l+f]}$.
10: Calculate activations A_p^l using the given sub mini-batch D_p^m.
11: **if** $l \in \{s-1, ..., s+f-1\}$ **then**
12: **Post** asynchronous communication: Allgather A_p^l.
13: **for** each layer $l = L-1, ...0$ **do** ▷ Backpropagation
14: Calculate errors E_p^l.
15: **if** $l \notin \{s, ..., s+f\}$ **then**
16: Calculate weight gradients ΔW_p^l for D_p^m
17: Add the weight gradiets to the local gradient sum: $G_p^l += \Delta W_p^l$.
18: **if** l is equal to k **then**
19: **Post** asynchronous communication: Allreduce $G_p^{[L-k:L-1]}$.
20: **Post** asynchronous communication: AllReduce $G_p^{[0:L-k-1]}$.
21: **Wait** until the communication for $G_p^{[L-k:L-1]}$ is finished (line 19).
22: **for** each layer $l = L-1, ...0$ **do** ▷ Parameter Update
23: **if** l is equal to $L-k$ **then**
24: **Wait** until the communication for $G_p^{[0:L-k-1]}$ is finished (line 20).
25: **for** each layer $l = s, ...s+f$ **do**
26: **Post** asynchronous communication: Allgather E_p^l.
27: **if** $l \notin \{s, ..., s+f\}$ **then**
28: Update parameter, W_p^l.

Algorithm 1 presents the simplified version of the training algorithm. The input dataset is randomly partitioned to M mini-batches and the model is trained on mini-batches one after another. Each iteration of the outermost loop has a strong data dependency caused by the parameter update at line 10. The scalability of SGD-based training algorithm is limited by this data dependency.

We define a few notations to analyze the complexity of the algorithm: K is the largest number of neurons among all layers, L is the total number of layers, M is the number of mini-batches, and N is the size of each mini batch. The size of input feature maps and output feature maps cannot be larger than K. Thus, the complexity of processing each image at a layer is $O(K^2)$. Algorithm 1 trains L layers on MN images. Therefore, the complexity of the overall training is $O(LMNK^2)$.

III. Parallelization Strategy for CNN

Algorithm 2 presents the data-parallel CNN training algorithm that leverages the computation and communication overlap. The algorithm updates the parameters synchronously and the result of training is exactly same as that of 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.

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

A. Computation Workload and Data Layout

CNN has two types of computation-intensive layers, convolution layer and fully-connected layer. In convolution layers, we use $im2col$ [20] to rearrange the input data so that the computation pattern is changed from convolution to matrix multiplication. In fully-connected layers, the computation pattern is also a matrix multiplication [21].

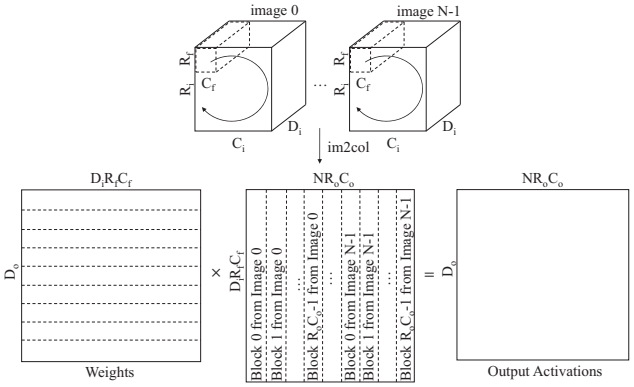

Figure 2: Computing activations using $im2col$ at the first fully-connected layer. N images are transformed to a single matrix and the weight matrix is multiplied by the transformed matrix. The result is the output activations.

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 utilities used in open source frameworks such as Caffe[21] or Torch[22]. 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 into a single large matrix. Figure 2 illustrates the workload of the first convolution layer in feed-forward stage.

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 $N \times D_i R_i C_i$, which is then transformed by $im2col$ into a $D_i R_f C_f \times N R_o C_o$ matrix. Then, the matrix is multiplied by the weight matrix of $D_o \times D_i R_f R_c$ and added by the bias vector of D_o. The output activation matrix of $D_o \times N R_o C_o$ is computed by Equation 4 and 5.

$$C^{l-1} = im2col(A^{l-1}) \tag{4}$$

$$A^{l} = \sigma(W^{l} C^{l-1} + B^{l}), \tag{5}$$

where σ is an activation function, A^l is the activation matrix calculated in layer l, B is 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 $D_i \times N R_i C_i$. We make $im2col$ support both data layouts, $N \times D_i R_i C_i$ and $D_i \times N R_i C_i$, 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 $D_i \times N R_i C_i$ 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 [21]. Note that the error matrix E and activation matrix A have the same data layout. E is computed by Equation 6 and 7. In Algorithm 2, these computations are performed at line 14.

$$C^{l} = W^{l+1} E^{l+1} \tag{6}$$

$$E^{l} = col2im(C^{l}), \tag{7}$$

In addition to the errors, the gradients of parameters are calculated in backpropagation. When calculating the gradients in layer l, the activation matrix of layer $l-1$ is transformed to a $D_o R_f C_f \times N R_i C_i$ matrix using $im2col$. Then, the $D_i \times N R_i C_i$ 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 4 and Equation 8, 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.

$$\Delta W^{l} = C^{l-1} E^{l}, \tag{8}$$

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 3 layout transformation and 3 matrix multiplications using Equation 4, 5, 6, 7, and 8.

For the fully-connected layers, the computations can be described by Equation 5, 7, and 8 if the matrix C is replaced with its original matrices A^{l-1}, E^l and A^{l-1} repectively. 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 $D_i R_i C_i$ in 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.

B. 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 9.

$$S_w = \sum_{i=0}^{L_c-1} D_o^i D_i^i R_f^i C_f^i + \sum_{i=0}^{L_f-1} K_b^i K_c^i, \quad (9)$$

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 10.

$$S_b = \sum_{i=0}^{L_c-1} D_o^i + \sum_{i=0}^{L_f-1} K_c^i \quad (10)$$

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.

1) Data Dependency: 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, 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 activations and errors. In contrast, the gradients in different layers are independent of each other. Figure 3 illustrates the data dependencies in CNN. 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.

C. Overlapping Computation and Communication

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.

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.

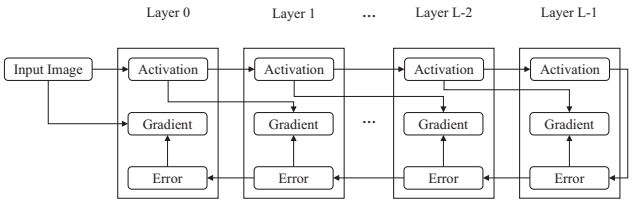

Figure 3: Data dependency in CNN. The activations are propagated from left to right in feed-forward stage, and the errors are propagated from right to left in backpropagation stage. The gradients are computed using the activations and errors. Each arrow indicates the data dependency.

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 5.

2) 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. Since each connection has the corresponding weight parameter, a fully-connected layer typically has more weights than a convolution layer. For example, VGG-A [7] 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

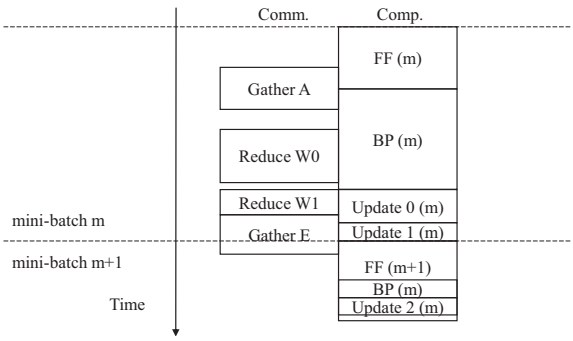

Figure 4: Time-flow chart with the maximized overlap. This figure illustrates the ideal case on which all communications are hidden behind the computation. In this case, a linear speedup can be expected.

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 $(K_b^l + K_c^l)N$ in 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 5, we demonstrate the impact of various values of f on the scalability.

3) Communications in data-parallel CNN: 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 4 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 training. It allows to overlap $GatherE$ with the feed-forward computation for the next mini-batch.

D. Complexity Analysis

In Algorithm 2, each mini-batch is partitioned to P nodes at line 3. Thus, the complexity is reduced to $O(\frac{LMNK^2}{P})$. However, due to the replicated gradient calculations in f fully-connected layers at line 9, the amount of serial workload should be taken into account. We use two notations, C and F, the number of convolution layers and fully-connected layers, respectively. The complexity of Algorithm 2 is $O((\frac{C}{P} + F)MNK^2)$.

IV. Evaluation

The parallel CNN training algorithm described in this paper is implemented in C using Intel MKL library for computational kernels such as matrix multiplication. The data-parallelism is implemented using MPI for all the inter-process communications and non-kernel loops are parallelized using OpenMP. 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 inteconnections with 'dragonfly' topology.

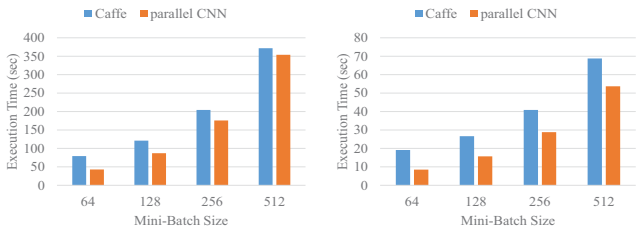

Figure 5: VGG-A training time on a single node: sequential performance (left) and multi-threaded (32 cores) performance (right). Caffe is an open-source framework and parallel CNN is our implementation. The experiments are performed with varying mini-batch size.

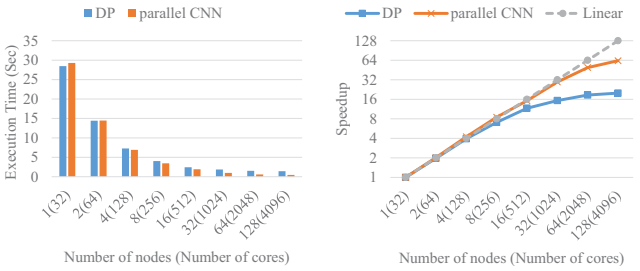

Figure 6: VGG-A training time (left) and speedup (right) for a single mini-batch size of 256). k is set to 9 and f is set to 2. We compare our approach (parallel CNN) with the pure data-parallel training algorithm (DP). The speedup is calculated with respect to the number of nodes.

The most representative dataset for visual recognition tasks is ImageNet. 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 [7], 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.

A. Single-Node Performance Study

1) Single-Node Performance Comparison with Caffe: 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 [23] with MKL library. Figure 5 presents the single-threaded and multi-threaded performances with varying mini-batch size. We observe that our implementation delivers shorter execution time than Caffe for all the mini-batch sizes. First,

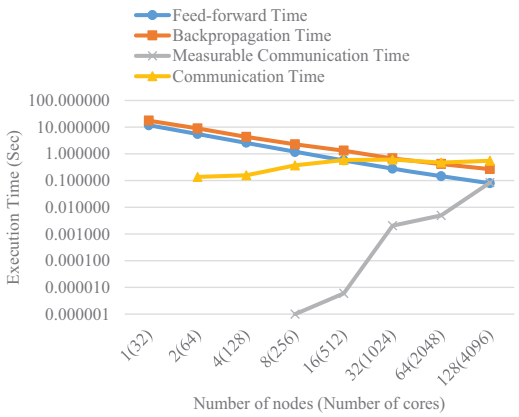

Figure 7: Timing breakdown for VGG-A training (mini-batch size of 256). Communication time is the accumulated time for all inter-process communications. Measurable communication time is a part of the communication time which is not overlapped with any computation time.

the performance gain of the single threaded training is achieved by the computation pattern described in section 3. We perform only 3 matrix multiplications and 3 layout transformations for each layer, while Caffe performs $3N$ matrix multiplications and $3N$ layout transformations. Given the same workload, less function calls improves the 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 $col2im$. Our results demonstrate that the multi-node performance study following this section is based on the reasonable level of single-node performance.

B. Multi-Node Performance Study

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

1) End-to-End Training Time and Speedup: We compare our parallelization strategy with 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 6 presents the training time and speedup. The speedups are calculated with respect to the number of nodes, based on the single-node performance presented in the previous section. We see that our parallel CNN shows a significantly improved speedup. We achieve a speedup of 62.97× on 128 nodes, while DP delivers upto 19.92× speedup.

2) Timing Breakdowns: If a communication for the gradients is not finished before updating the corresponding parameters, the update is blocked until the communication is finished. We define this blocking time as measurable communication time. To evaluate the proposed overlapping strat-

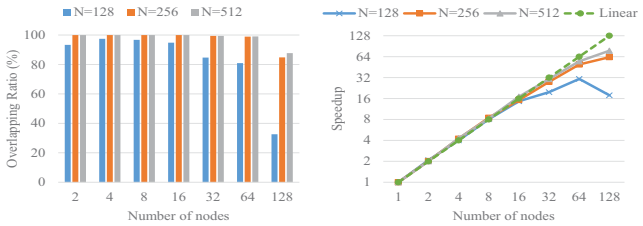

Figure 8: Overlapping ratio (left) and speedup (right) with varying mini-batch size. k is set to 9. and f is set to 2. The larger mini-batch size increases the computation workload and allows the higher overlapping ratio.

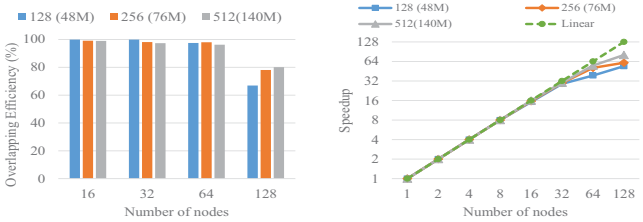

Figure 9: Overlapping ratio (left) and speedup (right) with varying number of parameters. VGG-128, VGG-256, and VGG-512 models are trained on a single mini-batch size of 256. Maximum speedups are 54×, 61×, and 80× respectively.

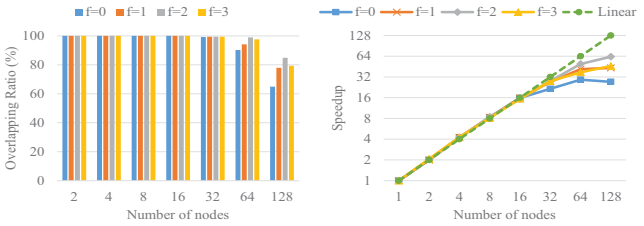

Figure 10: Overlapping ratio (left) and speedup (right) with varying number of fully-connected layers that replicate the gradient calculation. Replicating the gradient calculation at all the fully-connected layers can drop the speedup.

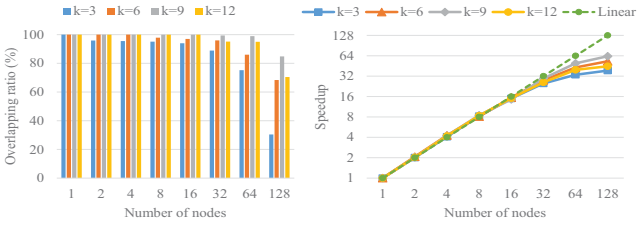

Figure 11: Overlapping ratio (left) and speedup (right) with varying number of layers that have the gradients for the first gradient chunk. The gradients should be grouped into two chunks such that the overlapping ratio of two communications are maximized.

egy, we compared the measurable communication time as well as the actual communication time. Figure 7 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 make the communication overlapped with the computation effectively. The measurable communication time appears from 8 nodes since the computation time becomes so short that it does not hide the entire communication time.

C. 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 11.

$$R = \frac{100 \times \sum_{i=0}^{2f+1}(T_c^i - T_b^i)}{\sum_{i=0}^{2f+1} T_c^i}, \tag{11}$$

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

1) 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 8 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 a better scalability with a larger mini-batch size.

2) 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 9 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.

3) 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 10 shows the 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), as explained in Section 3, the large gradient chunks engender the expensive communications. In our experimental environment, we achieved the best speedup when f is set to 2.

4) 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 11 presents the overlapping ratio and speedup. When k is set to 3 or 6, due to the small 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.

D. Comparison with Previous Works

In this section, we compare our approach with the existing works. Table 1 summarizes the previous works.

1) Comparison with parallel algorithms on GPUs: GPUs have been used to speedup the computing-intensive workload of training CNNs [1], [12], [24], [7], [13]. Many of the large-scale CNN trainings on GPUs are based on master-slave model. The 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 [12] reports 47× speedup of training with synchronous SGD. For asynchronous SGD, Strom et al. achieved 54× speedup.

Table I: 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

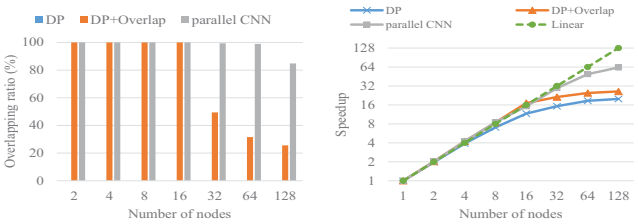

Figure 12: Overlapping ratio (left) and speedup (right) of VGG-A training (mini-batch size of 256). DP is the baseline without overlapping, DP+Overlap is the reproduced work based on [11], and parallel CNN is our proposed approach.

2) Comparison with parallel algorithms on CPU-based clusters: Many researchers have put much effort into scaling CNNs on CPU clusters [6], [11], [10], [25]. Recently, Dipankar Das et al. [11] 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 upto 90× speedup. PCL-DNN performs a communication for each layer and overlaps the communication with the back-propagation. Some communications are delayed to the next iteration such that the communication is overlapped with the feed-forward too. We reproduced the work in [11] 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 12 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.

V. Conclusion

Overlapping computation and communication is a fundamental technique to improve the scalability. Regardless of the type of neural network, the model architecture, and the optimization methods, the overlap should be maximized to fully-utilize the underlying hardware resources. In this paper, we propose a parallelization strategy for CNN training based on two major techniques to maximize the overlap.

We demonstrate that the communications can be overlapped with most of the computations and it results in achieving a considerably higher speedup. Since the proposed techniques are based on the data dependency in CNN, we believe that our approach can be applied to any other data-parallel deep neural networks.

VI. Acknowledgment

This work is supported in part by the following grants: NSF awards CCF-1409601; DOE awards DE-SC0007456, DE-SC0014330; AFOSR award FA9550-12-1-0458; NIST award 70NANB14H012; 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.

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] C. Nebauer, "Evaluation of convolutional neural networks for visual recognition," *IEEE Transactions on Neural Networks*, vol. 9, no. 4, pp. 685–696, 1998.

[3] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[4] P. Swietojanski, A. Ghoshal, and S. Renals, "Convolutional neural networks for distant speech recognition," *IEEE Signal Processing Letters*, vol. 21, no. 9, pp. 1120–1124, 2014.

[5] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[6] 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.

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

[8] Y. Qian, M. Bi, T. Tan, and K. Yu, "Very deep convolutional neural networks for noise robust speech recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 12, pp. 2263–2276, 2016.

[9] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[10] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system." in *OSDI*, vol. 14, 2014, pp. 571–582.

[11] 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.

[12] 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.

[13] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 4.

[14] S.-Y. Zhao and W.-J. Li, "Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee." in *AAAI*, 2016, pp. 2379–2385.

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

[16] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[17] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee, "Understanding deep neural networks with rectified linear units," *arXiv preprint arXiv:1611.01491*, 2016.

[18] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng, "On optimization methods for deep learning," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 265–272.

[19] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[20] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[21] 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.

[22] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Idiap, Tech. Rep., 2002.

[23] E. Shelhamer and . contributors, "Caffe," https://github.com/BVLC/caffe.git, 2017.

[24] N. Strom, "Scalable distributed dnn training using commodity gpu cloud computing." in *INTERSPEECH*, vol. 7, 2015, p. 10.

[25] I.-H. Chung, T. N. Sainath, B. Ramabhadran, M. Picheny, J. Gunnels, V. Austel, U. Chauhari, and B. Kingsbury, "Parallel deep neural network training for big data on blue gene/q," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1703–1714, 2017.