Scalable Distributed DL Training: Batching Communication and Computation

Shaoqi Wang, Aidi Pi, Xiaobo Zhou
Department of Computer Science
University of Colorado, Colorado Springs, CO, USA
{swang, epi, xzhou}@uccs.edu

Abstract

Scalability of distributed deep learning (DL) training with parameter server architecture is often communication constrained in large clusters. There are recent efforts that use a layer by layer strategy to overlap gradient communication with backward computation so as to reduce the impact of communication constraint on the scalability. However, the approaches cannot be effectively applied to the overlap between parameter communication and forward computation. In this paper, we propose and design iBatch, a novel communication approach that batches parameter communication and forward computation to overlap them with each other. We formulate the batching decision as an optimization problem and solve it based on greedy algorithm to derive communication and computation batches. We implement iBatch in the open-source DL framework BigDL and perform evaluations with various DL workloads. Experimental results show that iBatch improves the scalability of a cluster of 72 nodes by up to 73% over the default PS and 41% over the layer by layer strategy.

Introduction

Deep learning (DL) (LeCun, Bengio, and Hinton 2015), a class in machine learning (ML) field, has achieved remarkable success across a wide range of applications, including image recognition, object detection, and natural language processing. In DL, deep neural network (DNN) models achieve high accuracy through the use of deeply layered structures with many parameters (He et al. 2016; Szegedy et al. 2017).

In order to reduce the training time of DNN models in a single compute node, DL frameworks (e.g., TensorFlow (Abadi et al. 2016), BigDL (Wang et al. 2018c)) implement distributed DNN models, in which the training data is partitioned over distributed clusters with multiple nodes by taking advantage of data parallelism. Among the distribution implementations, parameter server (PS) architecture (Li et al. 2014; Dai et al. 2015; Xing et al. 2015) is a common communication architecture that synchronizes parameter updates (i.e., gradients) among multiple nodes. PS separates the cluster nodes into workers and servers. The servers serve as a distributed storage of model parameters. In the default setting, the one iteration execution contains four sequential operations. First, the workers pull the parameters from the servers. Then, they conduct forward computation based on the pulled parameters, and compute the gradients through backward computation. Finally, they push the gradients during each synchronization into the servers.

However, as the number of nodes in a cluster increases, the distributed implementation with PS often scales poorly. For instance, a recent study (Zhang et al. 2017) shows that training VGG19-22K network that contains 229M (million) parameters with open-source TensorFlow on 32 nodes can be slower than training on a single node. The reason is that when the number of nodes increases, more nodes share the network bandwidth of the cluster so that the available bandwidth for each node decreases (Ahmad et al. 2014; Wang et al. 2018b). As the result, the parameter communication time and gradient communication time increase, prolonging the execution time of one iteration. Thus, the cluster network bandwidth becomes a severe bottleneck for large-scale distributed DL training.

Previous efforts reduce the impact of communication constraint on the scalability of distributed DNN model training from two aspects. From the aspect of algorithm, Gradient Quantization (Zhou et al. 2016; Wen et al. 2017; Micikevicius et al. 2018) and Sparsification (Aji and Heafield 2017; Lin et al. 2018; Chen et al. 2018) reduce the gradient communication time by cutting down the size of gradients in communication. These techniques need to balance the trade-off between the model accuracy and the size of gradients. From the aspect of PS design, Poseidon (Zhang et al. 2017) proposes a novel PS architecture, in which gradient communication is overlapped with backward computation layer by layer to reduce the execution time of these two operations. With the layer by layer strategy, the gradients of each layer are pushed to the servers immediately after the backward computation in the layer. However, Poseidon does not overlap parameter communication with forward computation.

The trend of Gradient Quantization and Sparsification continues reducing the size of gradients in communication. As the result, parameter communication accounts for an increasingly large part of the total communication between the workers and the servers. To this end, we propose to overlap parameter communication with forward computation so as to reduce the execution time including parameter communication time and forward computation time. However, the layered model structures of DNN pose severe challenges on the overlap so that the straightforward layer by layer overlap strategy cannot be effectively applied here. First, we find that pulling parameters from the servers layer by layer brings significant overhead in parameter communication. Second, the parameter communication time can be longer or shorter than the forward computation time across layers so that the communication can only be partially overlapped with the computation. For instance, the communication time is usually much longer than the computation time in fully-connected layers, while the computation time is usually much longer than the communication time in convolutional layers (Krizhevsky, Sutskever, and Hinton 2012).

We propose and design iBatch, a novel communication approach that batches parameter communication and forward computation to overlap them with each other. Instead of using a constant batch size, iBatch uses communication and computation batches with various sizes in order to maximize the overlap. Specifically, we first profile the parameter communication time and the forward computation time in DNN model training. Then, we formulate the batching decision as an optimization problem of execution time minimization based on the profile. Finally, we use greedy algorithm that maximizes the overlap to solve the optimization problem and derive the batches.

In a nutshell, we make the following technical contributions: 1) We find the naive implementation of the layer by layer strategy is not effective to overlap parameter communication with forward computation. 2) We design iBatch based on problem formulation and algorithm design to conduct the overlap through batching parameter communication and forward computation. 3) We implement iBatch in the open-source DL framework BigDL and perform comprehensive evaluations with various DL workloads. Experimental results show that iBatch achieves up to 45x speedup for VGG19-22K network in a cluster of 72 nodes, 73% improvement over the default PS (26x speedup) and 41% improvement over the layer by layer strategy (32x speedup).

Related Work

Distributed DL frameworks with PS. Based on PS architecture (Li et al. 2014; Dai et al. 2015), a number of distributed DL frameworks have been developed. DistBelief (Dean et al. 2012) is a distributed framework that trains deep networks using asynchronous stochastic gradient descent. TensorFlow (Abadi et al. 2016) is Google's distributed DL framework that uses a dataflow graph to represent DL models and synchronizes model parameters via PS. MXNet (Chen et al. 2015) is another DL framework that uses PS for distributed execution and supports graph representations for DL models. The above frameworks, however, lack the benefits of tight integration with general-purpose computational frameworks such as Apache Spark (Zaharia et al. 2012). To this end, BigDL (Wang et al. 2018c) is proposed as a distributed DL library for Spark. With BigDL, users can write their DL applications as standard Spark programs that can directly run on top of existing Spark clusters.

Distributed DL implementation based on PS has limited scalability due to the high volume of communication in parameter synchronization. Thus, there are techniques to reduce network communication. Poseidon (Zhang et al. 2017) uses wait-free backpropagation that overlaps the backward propagation computation with the gradient communication. It also uses a hybrid communication scheme that optimizes the number of bytes required to synchronize each layer. However, the layer by layer overlap strategy is not efficient for overlapping the forward propagation computation with the parameter communication.

Gradient Quantization and Sparsification that reduce the size of data in communication are also extensively studied. In gradient quantization, 1-bit SGD (Seide et al. 2014) is proposed to reduce gradients transfer data size and achieved 10x speedup in traditional speech applications. TernGrad (Wen et al. 2017) uses 3-level gradients and DoReFa-Net (Zhou et al. 2016) uses 1-bit weights with 2-bit gradients. In gradient sparsification, threshold quantization (Strom 2015) and gradient dropping (Aji and Heafield 2017) are proposed to sparsify the gradients by a single threshold based on the absolute value. However, the threshold is hard to choose in practice. AdaComp (Chen et al. 2018) proposes to automatically tune the compression rate depending on local gradient activity. DGC (Lin et al. 2018) reduces the communication bandwidth through momentum correction, local gradient clipping, momentum factor masking, and warm-up training. These techniques need to balance the trade-off between model accuracy and the size of data in communication.

Distributed DL frameworks without PS. There are DL frameworks that make use of decentralized training to remove the burden of PS deployment in distributed environments while maintaining data parallelism. In MALT (Li et al. 2015), workers exchange gradients with a subset of workers selected by a Halton sequence. Due to the synchronization delay, it suffers from slow convergence, especially for complex neural network models. SFB (Xie et al. 2016) and Ako (Watcharapichat et al. 2016) parallelize DL applications using peer-to-peer communication.

Motivation

In this section, we first introduce the PS architecture for parallelizing DNN training on clusters. We then describe the communication and computation overlap that takes advantage of decomposing the procedure of DNN training into a sequence of communication and computation operations. Finally, we provide a case study to show the potential gain of overlapping parameter communication and forward computation in a batch of layers.

Parameter Server Architecture

Most distributed ML/DL frameworks (e.g., Spark, MXNet, TensorFlow, BigDL) employ the PS architecture to train DNN models iteratively as shown in Figure 1. In this architecture, there are two types of cluster nodes: servers and workers. Specifically, the parameters in a DNN model are partitioned among the servers and the training data are split among the workers. In one iteration, each worker first pulls the parameters from the servers and computes parameter updates (i.e., gradients) locally through forward and backward

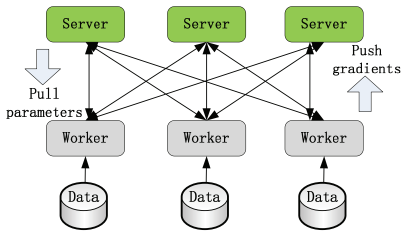
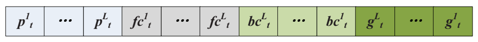
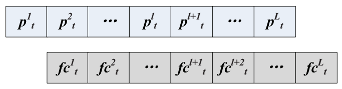


Figure 1: The parameter server architecture.


p^1_t … p^L_t fc^1_t … fc^L_t bc^L_t … bc^1_t g^L_t … g^1_t
(a) The procedure of DNN training in iteration *t*.


p^1_t p^2_t … p^l_t p^{l+1}_t … p^L_t
fc^1_t fc^2_t … fc^{l+1}_t fc^{l+2}_t … fc^L_t
(b) The overlap between parameter communication and forward computation in iBatch.

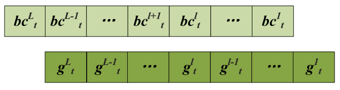

bc^L_t bc^{L-1}_t … bc^{l+1}_t bc^l_t … bc^1_t
g^L_t g^{L-1}_t … g^l_t g^{l-1}_t … g^1_t
(c) The overlap between gradient communication and backward computation in previous efforts.

Figure 2: The communication and computation overlap.

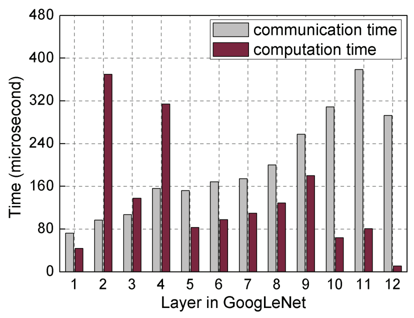


Figure 3: The parameter communication time and the forward computation time in each layer in GoogLeNet.

propagation computation using its data partition. The forward computation computes the value of an objective function (i.e., loss function) based on the parameters. The backward computation generates the gradients based on the value of objective function. The workers then push the gradients to the servers. After receiving the gradients, the servers update the model parameters based on Stochastic Gradient Descent algorithm. There are different types of synchronizations between the servers and the workers, such as BSP (Zaharia et al. 2012), A-BSP (Wang et al. 2018a), SSP (Xing et al. 2015), and ASP (Chilimbi et al. 2014).

Communication and Computation Overlap

In the default PS, each worker conducts four operations sequentially in one iteration: parameter communication (i.e., pulling parameters), forward computation, backward computation, and gradient communication (i.e., pushing gradients). The four operations are defined as P_t, FC_t, BC_t, and G_t, respectively, where t denotes the number of iterations. Thus, the procedure of DNN training in iteration t can be notated as $[P_t, FC_t, BC_t, G_t]$.

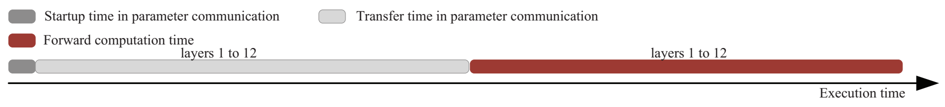
The forward and backward computation is performed through DNN layer by layer. If we define a forward and a backward computation through the *l*th layer of a network as fc_t^l and bc_t^l, respectively, the computation $[FC_t, BC_t]$ is notated as $[fc_t^1, fc_t^2, ..., fc_t^L, bc_t^L, bc_t^{L-1}, ..., bc_t^1]$, where L denotes the number of layers in the network. Meanwhile, as every layer of a network contains an independent set of parameters, P_t and G_t can be decomposed as $[p_t^1, p_t^2, ..., p_t^L]$ and $[g_t^L, g_t^{L-1}, ..., g_t^1]$, respectively, where p_t^l is defined as pulling the parameters of layer l and g_t^l is defined as pushing the gradients of layer l. Thus, the training procedure can be written as $[p_t^1, ..., p_t^L, fc_t^1, ..., fc_t^L, bc_t^L, ..., bc_t^1, g_t^L, ..., g_t^1]$, as illustrated in Figure 2(a). The communication and computation perform sequentially, waiting for each other to finish.

The communication can be overlapped with computation based on two independencies in the training procedure: (1) the parameter communication p_t^l is independent of the forward computation fc_t^i $(i < l)$; (2) the gradient communication g_t^l is independent of the backward computation bc_t^i $(i > l)$. As the result, two overlaps can be performed as shown in Figures 2(b) and 2(c): (1) The overlap between parameter communication and forward computation; (2) The overlap between gradient communication and backward computation. Previous efforts explored the latter overlap by overlapping gradient communication with backward computation layer by layer. Specifically, g_t^l is performed immediately after finishing bc_t^l so that g_t^l and bc_t^i $(i > l)$ can be executed concurrently without blocking each other. In contrast, iBatch explores the former overlap through batching parameter communication and forward computation.

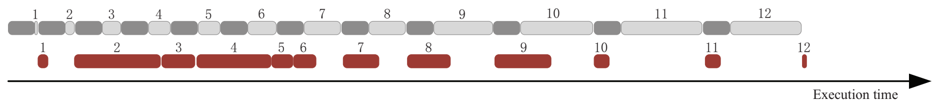
Case Study

The straightforward layer by layer strategy cannot be effectively applied to overlap parameter communication with forward computation due to two reasons. First, this strategy brings significant overhead in parameter communication. Specifically, the total parameter communication time depends on two factors: the number of communications and the time in each communication (Lee et al. 2017). The time in each communication consists of a startup time (e.g., searching the servers that store parameters and handling TCP connections) and a transfer time. The startup time is independent of the size of parameters in the communication and the transfer time is in direct proportion to the size. Thus, if a worker pulls the parameters in a neural network layer by layer, the number of communications equals the number of layers in the network, leading to significant overhead in the startup time.

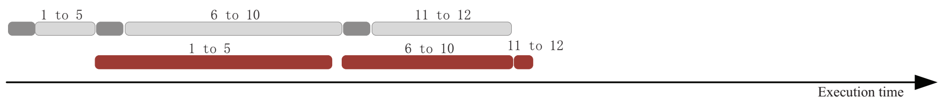
Second, the parameter communication time can be longer or shorter than the forward computation time across layers. As the result, the communication can only be partially overlapped with the computation. To validate this, we created a BigDL cluster containing 6 workers and 6 servers and ran



(a) Sequential execution of communication and computation in the default PS without overlap.



(b) Overlap the communication with computation layer by layer.



(c) Overlap the communication with computation in a batch of layers with iBatch.

Figure 4: The procedure of parameter communication and forward computation with three communication approaches. The time in each communication is divided into a startup time and a transfer time. The number denotes the layer in the network.

the representative GoogLeNet (Szegedy et al. 2015) using the data from Imagenet (Krizhevsky, Sutskever, and Hinton 2012). Figure 3 illustrates the communication time and the computation time in each layer (adjacent convolutional layer and pooling layer are regarded as one layer since there are no parameters in pooling layers). The result shows that the communication and computation time varies across layers. The communication time is much longer than the computation time in fully-connected layers (e.g., layer 11). The computation time is much longer than the communication time in convolutional layers (e.g., layer 2).

According to the time in Figure 3, we present the procedure of parameter communication and forward computation with the default PS, the layer by layer strategy, and iBatch in Figure 4. In the default PS, a worker pulls all parameters from the servers once and then conducts the forward computation layer by layer. Figure 4(a) illustrates the timeline of this procedure. The figure shows that there is only one startup time since the number of communications is one. However, this procedure is highly sequential, in which the computation waits for the communication to finish.

With the layer by layer strategy, a worker pulls the parameters from the servers layer by layer and conducts the forward computation in each layer once the corresponding parameters are pulled from the servers. We present the timeline of this procedure in Figure 4(b). The figure shows that the communication is partially overlapped with the computation. However, this strategy brings significant overhead (i.e., multiple startup time) since the number of communications equals the number of layers. In some layers (e.g., layers 10 and 11), the forward computation has to wait for a long time before the communication finishes, prolonging the execution time.

iBatch proposes to overlap the communication with the computation in a batch of layers as shown in Figure 4(c). Concretely, a worker pulls all parameters through three batches. The first batch pulls the parameters from layers 1 to 5. The second batch pulls the parameters from layers 6 to 10. The third batch pulls the parameters from layers 11 to 12. Compared with the layer by layer strategy, the startup time is saved in iBatch since the number of communications is reduced to three. Also, with iBatch, more communication is overlapped with the computation in this case, leading to a significant reduction in the execution time.

iBatch Design and Implementation

The goal of iBatch is to minimize the execution time including the total parameter communication time and the forward computation time. We first formulate the batching decision as an optimization problem of execution time minimization based on the profile of the parameter communication time and the forward computation time. Then, we use greedy algorithm that maximizes the overlap to solve the problem and derive communication and computation batches.

Given a cluster and a neural network, we measure the forward computation time in each layer in the network and define the time in ith layer as C^i. For the parameter communication, we measure the startup time as well as the transfer time in each layer. The startup time is notated as T_s and the transfer time in ith layer is notated as T_t^i.

Problem Formulation

According to the case study, the execution time consists of three parts. The first part is the parameter communication time in the first batch (e.g., layers 1 to 5 in Figure 4(c)). The communication in this part cannot be overlapped with computation since there are no parameters for any computation. The second part is the parameter communication time from the second batch to the last batch (e.g., layers 6 to 10 and layers 11 to 12 in Figure 4(c)). In this part, the communication in one batch is independent of the computation in its previous batch so that the communication can be overlapped with the computation. The third part is the forward computation time in the last batch (e.g., layers 11 to 12 in Figure 4(c)). The computation in this part cannot be overlapped with any communication since there is no communication in the part.

To formulate the execution time, the number of batches in a L-layer network is notated as N and the batches are notated as $[[l_0 + 1 = 1, l_1], [l_1 + 1, l_2], ..., [l_{N-1} + 1, l_N = L]]$, where l_i denotes the last layer in batch *i*. In other words, batch *i* includes layers from $l_{i-1}+1$ to l_i. For instance, batch 1 (i.e., the first batch) includes layers from 1 to l_1. Batch *N* (i.e., the last batch) includes layers from $l_{N-1}+1$ to L. The batches in Figure 4(c) is [[1, 5], [6, 10], [11, 12]].

Based on the notations, the first part in the execution time is formulated as:

$$EXE_1 = T_s + \sum_{1 \leq i \leq l_1} T_t^i \tag{1}$$

In the second part, the number of communications is *N*-1. So, the total startup time is $(N-1) * T_s$ and this part is formulated as:

$$\begin{aligned} EXE_2 = T_s + \sum_{l_1+1 \leq i \leq l_2} T_t^i + ... + T_s + \sum_{l_{N-1}+1 \leq i \leq L} T_t^i \\ = (N-1) * T_s + \sum_{l_1+1 \leq i \leq L} T_t^i \end{aligned} \tag{2}$$

The third part is formulated as:

$$EXE_3 = \sum_{l_{N-1}+1 \leq i \leq L} C^i \tag{3}$$

We define the execution time as $ExeTime$. Thus, we minimize $ExeTime$ by solving the following optimization problem:

$$min\ EXETime = EXE_1 + EXE_2 + EXE_3 \tag{4}$$

subject to

$$T_s + \sum_{l_k+1 \leq i \leq l_{k+1}} T_t^i \geqslant \sum_{l_{k-1}+1 \leq i \leq l_k} C^i \quad 0 < k < N \tag{5}$$

Objective Eq. 4 minimizes the sum of the three parts in the execution time. Constraint Eq. 5 ensures the overlap between the communication in batch *k+1* and the computation in batch k.

Algorithm Design

To minimize the execution time, l_i $(0 < i < N)$ in the batches is derived one by one through greedy algorithm. Specifically, we use two greedy algorithms that makes greedy choices at each step to ensure that the objective function in Eq. 4 is optimized. Each algorithm derives one candidate of communication and computation batches. From two candidates of the batches, we choose the one with the minimal execution time to batch parameter communication and forward computation.

Algorithm 1 generates l_i from l_1 to l_N. The first step (lines 2 to 5) that chooses l_1 and l_2 decides the communication time $EXE1$ and the overlapping time between the communication in the second batch and the computation in the first batch. Thus, the greedy choice maximizes the overlapping time while minimizing the communication time. Specifically, lines 2 to 3 first construct a set of pairs S2 that meet Constraint Eq. 5. That is, for all pairs in S2, the communication in the second batch can be overlapped with the computation in the first batch. From S2, line 4 then selects the pairs with the maximum overlapping time. Finally, from the pairs selected by line 4, line 5 selects one with the minimum communication time in the first batch. In the rest steps (lines 7 to 13), the algorithm chooses one layer in one step. For instance, the choice of layer l_k decides the overlapping time between the communication in batch k and the computation in batch $k-1$. Thus, the greedy choice maximizes the overlapping time by minimizing the difference between the communication time in batch k and the computation time in batch $k-1$.

Algorithm 1
Greedy algorithm that generates l_i from l_1 to l_{N-1}

```
1: /* The first step */
2: Derive set S1 that contains all pairs of [$l_1$, $l_2$];
3: From S1, select pairs that meet Eq. 5 and form them as set S2;
4: From S2, select pairs with the maximum $\sum_{1 \leq i \leq l_1} C^i$;
5: From the selected pairs, choose one with the minimum
   $T_s + \sum_{1 \leq i \leq l_1} T_t^i$;
6: /* The rest steps */
7: Previous two layers in batches: $l_{k-1}$ = $l_2$, $l_{k-2}$ = $l_1$,;
8: repeat
9:     Define current layer as $l_k$;
10:    From [$l_{k-1}$ + 1, $L$], select all $l_x$ that meet
       $T_s + \sum_{l_{k-1}+1 \leq i \leq l_x} T_t^i \geqslant \sum_{l_{k-2}+1 \leq i \leq l_{k-1}} C^i$;
11:    From the selected $l_x$, choose one as $l_k$ that has the minimum
       $T_s + \sum_{l_{k-1}+1 \leq i \leq l_k} T_t^i - \sum_{l_{k-2}+1 \leq i \leq l_{k-1}} C^i$;
12:    $l_{k-2}$ = $l_{k-1}$;
13:    $l_{k-1}$ = $l_k$;
14: until $l_{k-1} = L$;
```

Algorithm 2 generates l_i from l_{N-1} to l_1. The first step (lines 2 to 5) that chooses l_{N-1} and l_{N-2} decides the computation time $EXE3$ and the overlapping time between the communication in the last batch and the computation in batch $N-1$. Thus, the greedy choice maximizes the overlapping time while minimizing the computation time. Specifically, lines 2 to 3 first construct a set of pairs S2 that meet Constraint Eq. 5. That is, for all pairs in S2, the communication in the last batch can be overlapped with the computation in batch $N-1$. From S2, line 4 then selects the pairs with the maximum overlapping time. Finally, from the pairs selected by line 4, line 5 selects one with the minimum computation time in the last batch. In the rest steps (lines 7 to 13), the algorithm chooses one layer in one step. For example, the choice of layer l_k decides the overlap between the communication in batch $k+1$ with the computation in batch k. Thus, the greedy choice maximizes the overlapping time by minimizing the difference between the communication time in batch $k+1$ and the computation time in batch k.

Implementation

We have implemented iBatch in BigDL (version 0.5.0) by modifying source files in package `com.intel.analytics.bigdl`.

Computation and communication profile. The `forward` function in `AbstractModule.scala` takes a layer as input and performs the forward computation of the layer. Thus, the forward computation time C^i is profiled by

Algorithm 2
Greedy algorithm that generates l_i from l_{N-1} to l_1

```
1:  /* The first step */
2:  Derive set S1 that contains all pairs of [$l_{N-1}$, $l_{N-2}$];
3:  From S1, select pairs that meet Eq. 5 and form them as set S2;
4:  From S2, select pairs with the maximum
    $\sum_{l_{N-2}+1 \leq i \leq l_{N-1}} C^i$;
5:  From the selected pairs, choose one with the minimum
    $\sum_{L_{N-1} \leq i \leq L} C^i$;
6:  /* The rest steps */
7:  Next two layers in batches: $l_{k+1}$ = $l_{N-2}$, $l_{k+2}$ = $l_{N-1}$;
8:  repeat
9:      Define current layer as $l_k$;
10:     From [0, $l_{k+1}$ - 1], select all $l_x$ that meet
        $T_s + \sum_{l_{k+1}+1 \leq i \leq l_{k+2}} T_t^i \geqslant \sum_{l_x+1 \leq i \leq l_{k+1}} C^i$;
11:     From the selected $l_x$, choose one as $l_k$ that has the minimum
        $T_s + \sum_{l_{k+1}+1 \leq i \leq l_{k+2}} T_t^i - \sum_{l_k+1 \leq i \leq l_{k+1}} C^i$;
12:     $l_{k+2}$ = $l_{k+1}$;
13:     $l_{k+1}$ = $l_k$;
14: until $l_{k+1} = 0$;
```

Table 1: Neural networks for evaluation.

Model	# Params	Dataset
GoogLeNet	5M	ILSVRC12
Inception-V3	27M	ILSVRC12
VGG19	143M	ILSVRC12
VGG19-22K	229M	ImageNet22K

measuring the runtime of this function. The `getWeights` function in `AllReduceParameter.scala` first locates the parameters (i.e., on which servers) using function `getWeightBlockId` and then calls function `fetchBlockSync` in `BlockTransferService.scala` to transmit the parameters from the first layer to the last layer. Thus, the transfer time T_t^i is profiled by measuring the runtime of `fetchBlockSync`. The runtime difference between `getWeights` and `fetchBlockSync` is regarded as T_s.

Computation and communication overlap. To batch the forward computation, we implemented a new function `iBatchforward` that takes batch *i* as input and calls `forward` to perform the computation from layers $l_{i-1}+1$ to l_i. To batch the parameter communication, we modified the function `getWeights` and rename it as `iBatchgetWeights`. The modified function takes batch *i* as input, locates the parameters in the batch, and transmits the parameters from layers $l_{i-1}+1$ to l_i. Default PS implementation in BigDL uses one thread to run `getWeights` and `forward` sequentially without the overlap. To enable the overlap, we implemented two threads that concurrently run `iBatchgetWeights` input with batch *i* and `iBatchforward` input with the previous batch $i-1$.

Evaluation Setup

Testbed

We conduct our experiments on a CPU cluster in a private cloud. The cloud runs on 8 HP BL460c G6 blade servers interconnected with 10Gbps global Ethernet. The number of nodes in the cluster ranges from 1 to 72 to evaluate the scalability of distributed DL. All nodes run Ubuntu Server 14.04 with Linux kernel 4.4.0-64. To achieve high performance in the forward and backward computation, BigDL uses Intel Math Kernel Library and multithreaded programming in each computation task.

Dataset and DL Models

Our experiments focus on the image classification applications where DL is most successfully applied. We use two well-known image classification datasets. (1) ImageNet22K, the largest public dataset for image classification, including 14.2 million labeled images from 21841 categories. (2) ILSVRC12, a subset of ImageNet22K that has 1.28 million of training images;

The scalability of distributed DL is evaluated using different neural networks: (1) GoogLeNet: a 22-layer convolutional neural network with 5M parameters. (2) Inception-V3: an improved version of GoogLeNet; (3) VGG19: a 16 convolutional layers and 3 fully-connected layers network, in total 143M parameters; (4) VGG19-22K: an improved version of VGG19 network (Zhang et al. 2017). The improved network has 229M parameters. Table 1 lists their statistics and configurations in.

Metrics

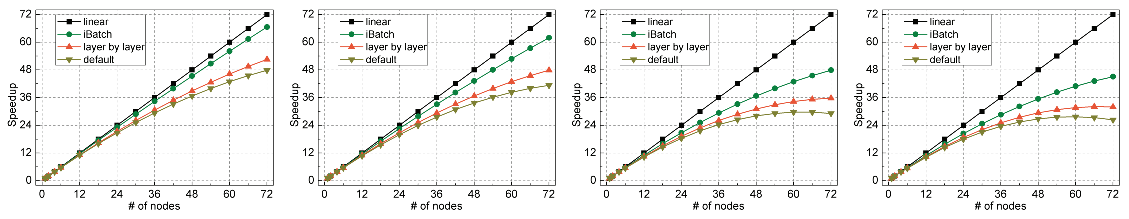
The performance metrics include scalability and normalized execution time $ExeTime$. The scalability denotes the speedup on throughput (number of iterations finished per hour) compared with single node DL. We evaluate the performance of three communication approaches: iBatch, sequential execution in the default PS (default), and the layer by layer overlap strategy (layer by layer). The performance is compared with the ideal linear speedup. Note that the performance metrics do not include DNN model accuracy since iBatch does not impact the accuracy. iBatch does not change the synchronization model in PS and it also does not change any hyperparameters in DNN.

Evaluation

Scalability

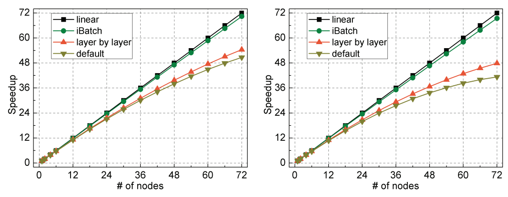
Figure 5 plots the scalability with three communication approaches, in which the layer by layer strategy is not used to speed up gradient communication and backward computation. Three approaches achieve almost linear speedup on throughput when the cluster size is small (the number of nodes $\leq$ 6). The reason is that the number of nodes that share the cluster network bandwidth is small so that the available bandwidth for each node is high. Thus, the scalability is not constrained by the communication time which is much less than the computation time.

When the cluster size is medium (6 < the number of nodes $\leq$ 30), the available bandwidth for each node is less than that in the small clusters and the communication time becomes non-negligible compared with the computation time. Thus, the scalability is constrained by the communication time. For example, the default PS only achieves 26x, 25x,



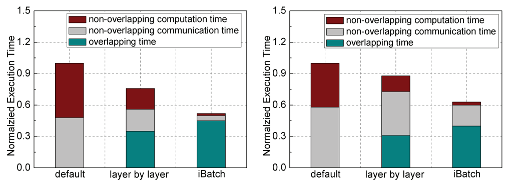
(a) GoogLeNet. (b) Inception-V3. (c) VGG19. (d) VGG19-22K.

Figure 5: Speedup vs. number of nodes when training GoogleNet, Inception-V3, VGG19, and VGG19-22K.



(a) Inception-V3. (b) VGG19.

Figure 6: Speedup vs. number of nodes with Gradient Sparsification when training Inception-V3 and VGG19.



(a) Inception-V3. (b) VGG19.

Figure 7: Execution time decomposed into the overlapping time, the non-overlapping communication time, and the non-overlapping computation time in two networks on 72 nodes.

23x, and 22x speedup on 30 nodes for the four networks. The scalability with the layer by layer strategy is better than that with the default PS, achieving 25x, 24x, 22x, and 21x speedup. Among the three approaches, iBatch achieves has the best scalability with 29x, 28x, 26x, and 25x speedup. Also, compared with GoogLeNet and Inception-V3, the communication time in VGG19 and VGG19-22K accounts for a larger part of the total execution time since the two VGG networks have a larger size of parameters. Thus, the scalability of the two VGG networks is worse than that of GoogLeNet and Inception-V3.

When the cluster size further increases (the number of nodes > 30), the communication time accounts for an increasingly large part of the total execution time, leading to the worse scalability. However, the scalability with iBatch is much better than that with the other two approaches. For instance, iBatch achieves 45x speedup on 72 nodes in VGG19-22K, 73% improvement over the default PS (26x speedup) and 41% improvement over the layer by layer strategy (32x speedup). iBatch has the best performance improvement in VGG19-22K since the communication time in the network is larger than that in the other three networks.

Although the scalability with iBatch is much better than that with the other two approaches, it is not close to the ideal scalability with linear speedup since the scalability is still constrained by the gradient communication time. To remove the constraint, we apply a simple Gradient Sparsification technique (Aji and Heafield 2017) on the gradient communication. Figure 6 further illustrates the scalability with the technique using two different neural networks. The result shows that iBatch achieves almost linear speedup on 72 nodes in two networks. Note that there is no technical challenge for the combination between iBatch and the Gradient Sparsification technique. iBatch batches parameter communication and forward computation, which does not impact gradient communication that can be optimized by the Gradient Sparsification technique.

Execution Time

Figure 7 plots the normalized execution time $ExeTime$ in Inception-V3 and VGG19 when the cluster size is 72. Specifically, the time is decomposed into the overlapping time, the non-overlapping communication time, and the non-overlapping computation time. The result shows that the overlapping time in the default PS is zero since the communication and the computation are performed sequentially. The layer by layer strategy can overlap the communication with the computation to some extent. The overlapping time in iBatch is longer than that in the layer by layer strategy, showing that batching the communication and the computation is more effective in the overlap. Also, since the layer by layer strategy brings significant communication overhead, its the non-overlapping communication time is longer than that in iBatch.

Conclusion

In this paper, we propose and design iBatch, a novel communication approach that batches parameter communication and forward computation to overlap them with each other. Given a network and a cluster, we first profile the parameter communication time and the forward computation time in the network training. Then, we formulate the batching decision as an optimization problem of execution time minimization and use greedy algorithm that maximizes the overlap to solve the problem as well as derive communication and computation batches. We have implemented iBatch in the open-source DL framework BigDL and performed evaluations with various DL workloads. Experimental results show that iBatch improves the scalability of a cluster of 72 nodes by up to 73% over the default PS and 41% over the layer by layer strategy. In future work, we plan to extend iBatch to other DL frameworks (e.g., TensorFlow).

Acknowledgments

We thank the anonymous reviewers for their feedback. This research was supported in part by U.S. NSF award CNS-1422119.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: a system for large-scale machine learning. In *Proc. of OSDI*.

Ahmad, F.; Chakradhar, S. T.; Raghunathan, A.; and Vijaykumar, T. 2014. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *Proc. of USENIX ATC*.

Aji, A. F., and Heafield, K. 2017. Sparse communication for distributed gradient descent. In *Proc. of EMNLP*.

Chen, T.; Li, M.; Li, Y.; Lin, M.; Wang, N.; Wang, M.; Xiao, T.; Xu, B.; Zhang, C.; and Zhang, Z. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chen, C.-Y.; Choi, J.; Brand, D.; Agrawal, A.; Zhang, W.; and Gopalakrishnan, K. 2018. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Proc. of AAAI*.

Chilimbi, T. M.; Suzue, Y.; Apacible, J.; and Kalyanaraman, K. 2014. Project adam: Building an efficient and scalable deep learning training system. In *Proc. of OSDI*.

Dai, W.; Kumar, A.; Wei, J.; Ho, Q.; Gibson, G. A.; and Xing, E. P. 2015. High-performance distributed ml at scale through parameter server consistency models. In *Proc. of AAAI*.

Dean, J.; Corrado, G.; Monga, R.; Chen, K.; Devin, M.; Mao, M.; Senior, A.; Tucker, P.; Yang, K.; Le, Q. V.; et al. 2012. Large scale distributed deep networks. In *Proc. of NIPS*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proc. of IEEE CVPR*.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Proc. of NIPS*.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *nature* 521(7553):436.

Lee, S.; Jha, D.; Agrawal, A.; Choudhary, A.; and Liao, W.-k. 2017. Parallel deep convolutional neural network training by exploiting the overlapping of computation and communication. In *Proc. of HiPC*.

Li, M.; Andersen, D. G.; Park, J. W.; Smola, A. J.; Ahmed, A.; Josifovski, V.; Long, J.; Shekita, E. J.; and Su, B.-Y. 2014. Scaling distributed machine learning with the parameter server. In *Proc. of OSDI*.

Li, H.; Kadav, A.; Kruus, E.; and Ungureanu, C. 2015. Malt: distributed data-parallelism for existing ml applications. In *Proc. of EuroSys*.

Lin, Y.; Han, S.; Mao, H.; Wang, Y.; and Dally, W. J. 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *Proc. of ICLR*.

Micikevicius, P.; Narang, S.; Alben, J.; Diamos, G.; Elsen, E.; Garcia, D.; Ginsburg, B.; Houston, M.; Kuchaev, O.; Venkatesh, G.; et al. 2018. Mixed precision training. In *Proc. of ICLR*.

Seide, F.; Fu, H.; Droppo, J.; Li, G.; and Yu, D. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Proc. of Interspeech*.

Strom, N. 2015. Scalable distributed dnn training using commodity gpu cloud computing. In *Proc. of Interspeech*.

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2015. Going deeper with convolutions. In *Proc. of IEEE CVPR*.

Szegedy, C.; Ioffe, S.; Vanhoucke, V.; and Alemi, A. A. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proc. of AAAI*.

Wang, S.; Chen, W.; Pi, A.; and Zhou, X. 2018a. Aggressive synchronization with partial processing for iterative ml jobs on clusters. In *Proc. of ACM Middleware*.

Wang, S.; Chen, W.; Zhou, X.; Zhang, L.; and Wang, Y. 2018b. Dependency-aware network adaptive scheduling of data-intensive parallel jobs. *IEEE Transactions on Parallel and Distributed Systems*.

Wang, Y.; Qiu, X.; Ding, D.; Zhang, Y.; Wang, Y.; Jia, X.; Wan, Y.; Li, Z.; Wang, J.; Huang, S.; et al. 2018c. Bigdl: A distributed deep learning framework for big data. *arXiv preprint arXiv:1804.05839*.

Watcharapichat, P.; Morales, V. L.; Fernandez, R. C.; and Pietzuch, P. 2016. Ako: Decentralised deep learning with partial gradient exchange. In *Proc. of ACM SoCC*.

Wen, W.; Xu, C.; Yan, F.; Wu, C.; Wang, Y.; Chen, Y.; and Li, H. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Proc. of NIPS*.

Xie, P.; Kim, J. K.; Zhou, Y.; Ho, Q.; Kumar, A.; Yu, Y.; and Xing, E. P. 2016. Lighter-communication distributed machine learning via sufficient factor broadcasting. In *Proc. of UAI*.

Xing, E. P.; Ho, Q.; Dai, W.; Kim, J. K.; Wei, J.; Lee, S.; Zheng, X.; Xie, P.; Kumar, A.; and Yu, Y. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1(2):49–67.

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M. J.; Shenker, S.; and Stoica, I. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of USENIX NSDI*.

Zhang, H.; Zheng, Z.; Xu, S.; Dai, W.; Ho, Q.; Liang, X.; Hu, Z.; Wei, J.; Xie, P.; and Xing, E. P. 2017. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *Proc. of USENIX ATC*.

Zhou, S.; Wu, Y.; Ni, Z.; Zhou, X.; Wen, H.; and Zou, Y. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*.