Efficient Scheduling in Training Deep Convolutional Networks at Large Scale

Can Que and Xinning Zhang, Senior Member, IEEE

Abstract—Deep convolutional network is one of the most successful machine learning models in recent years. However, training large deep networks is a time consuming process. Due to large number of parameters in these networks, the efficiency of data parallel methods is usually limited by the communication speed of networks. In this paper, we introduce two new algorithms to speed up training large deep networks with multiple machines: (1) propose a new scheduling algorithm to reduce communication delay in gradients transmission (2) present a new collective algorithm based on reverse-reduce tree to reduce link contentsions. We implement our algorithms on a well-known library Caffe and obtain near linearly scaling performance on commodity Ethernet networks.

Index Terms—All reduce, data parallel, deep learning, scheduling

I. INTRODUCTION

Deep learning is a class of neural networks that uses multiple neuron layers to represent different level of features. These networks, which get notable success in image classification [1] and voice recognition [2], usually contain millions of parameters trained on large-scale datasets. For example, the Resnet-50 model [1] trained 25 million parameters on Imagenet-1K dataset, which contains 1.2 million images with 1000 different categories. Given the scale of the problem, training deep networks is time consuming, usually costs several days or weeks using a single GPU or CPU. The long training time presents a key challenge in developing new deep architectures. For this reason, it is important to parallelize the training of deep networks to reduce training time.

The most common approach is data-parallel synchronous Stochastic Gradient Descent (SGD) [3], where each worker maintains a latest copy of the model and trains a subset of images. In each training iteration, the workers need to synchronize their models by exchanging gradients and parameters. Since deep neural models usually consist of millions of parameters, the synchronization process introduces significant communication overhead.

There are several factors that affects the performance of data-parallel synchronous SGD. To retain a certain accuracy, there is usually a limit on the size of mini-batch and we will drop accuracy if we increase the batch size beyond a threshold [4]. However, as we scale the system to a larger number of worker nodes, the training batch size increases linearly. Therefore, in a training system with a large number of worker nodes, the batch size of each worker node needs to be small. The small batch size in the worker nodes leads to shorter computation time and higher communication to computation ratio, which makes the optimization of communication algorithms become a critical factor in the performance of a large scale training system.

One strategy for parameter communication is using parameter server as in Ref. [5]. Fig. 1 shows the communication pattern of parameter server, where all the nodes are directly connected to the server. As pointed out by Ref. [3], one problem of parameter server is that, the communication overhead in parameter server increases linearly when we add more worker nodes to the system, which makes parameter server become the bottleneck of whole system. The problem can be alleviated by splitting parameters into smaller pieces and adding more nodes as parameter server. However, as the computation speed of worker nodes grows, the number of parameter servers may dramatically increase.

The AllReduce collective pattern in Message Passing Interface (MPI) [6] is used by Ref. [3] for gradient accumulation. Fig. 2 shows an implementation of AllReduce using reduction tree. While there are many different implementations of AllReduce [6] [7], the key property is the communication complexity grows as the log of the number of worker nodes.

Modern deep neural networks usually consist of hundreds of sequentially linked layers [1], directly applying AllReduce to deep neural networks as in Ref. [3] causes following two problems:

- Communication delay. Fig. 2 shows a typical reduction tree, although AllReduce can reduce communication complexity, it increases the delay at leaf nodes. For example in Fig. 2, the reduced gradients need to be transmitted from root node L0 to leaf nodes, so the communication delay at leaf node L6 is $O(\log_2^7)$, where $\log_2$ is the height of the reduction tree. The situation becomes worse when link contentsions and message queues in the internal nodes induce additionally delay. In this paper, we argue that the communication delay can be significantly reduced if we properly schedule the priority of packets.

- Link contentsions. For example, in Fig. 2, the internal node L1 needs to receive gradients from both L3 and L4, which means L3 and L4 are contending the receiving bandwidth of L1. The situation becomes worse when broadcasting and reducing communications are overlapped in AllReduce, which means L1 needs to receive broadcasts from L0, gradients from L3 and L4 at the same time. The link contentsions among L0, L3 and L4 significantly waste most of the receiving bandwidth in L1, thus limit the effective reducing speed of L1 to be less than $\frac{1}{7}$ of the full bandwidth.

In this paper, we propose a novel algorithm which reduces communication delay and link contentsions by jointly scheduling the priority of parameters and overlapping communication with both backpropagation and forwarding computations. Fig. 3 shows a simple convolutional neural networks with four layers. In current AllReduce implementations, the gradient updates are sent in the order of AllReduce call, which is the backward order from Softmax - Conv3 - Conv2 to Conv1. The
problem is that, in a worker node, even gradient updates of the Softmax layer are received, the node cannot start to compute until it receives outputs from Conv3. On the contrary, Conv1’s forwarding computation can be started immediately when the updated gradients of Conv1 are received, without waiting for Conv2 or other layers. In addition, the communication of gradient updates in subsequent layers such as Conv2 and Conv3 can be overlapped with the forwarding computation of Conv1. In the forwarding pass, internal nodes such as L1 and L2 do not receive gradients from their children, so the link contention ratios are lower than backpropagation stage, leading to higher communication speed. As a result, sending gradient updates of Conv3 firstly is not an optimal choice and our scheduling algorithm is proposed to make a better transmission sequence.

The main contribution of this paper is a novel scheduling algorithm to reduce communication delay in training deep neural networks. To the best of our knowledge, we are the first to introduce scheduling to speed up distributed deep learning system.

II. SYSTEM DESIGN

Training neural networks usually consists of three steps: forwarding, backpropagation and weight update. In the forwarding step, the activations, which are the outputs of neurons, are calculated layer by layer. Then, in backpropagation, the error terms are calculated and propagated in an inverse order. Using the error terms, we can calculate the gradients of weights at each layer. At the end of a training iteration, the parameter weights are updated as following:

$$W_{i+1} = W_i + \alpha \times \Delta W$$  

where $W_i$ is the weight of parameters in iteration $i$, $\alpha$ is the learning rate and $\Delta W$ is the gradient of parameters.

Let $N$ be the number of workers in a distributed training system, and the mini-batch size of each worker node is $b$. The total batch size of the training system is $B = N \times b$. In a training iteration, each worker calculates gradients by performing forwarding and backpropagation on a set randomly selected $b$ samples. The gradients at each node are then accumulated and synchronized via AllReduce. Finally, the worker nodes calculate a new version of parameters based on the reduced gradients using Eq. (1).

A. Reverse Reduce Tree

As shown in Fig. 2, in a binary reduction tree, the internal nodes such as L1 and L2 receive more packets than leaf nodes. To balance communication, we construct two subtrees with different internal nodes following Ref. [7], which is the state of art AllReduce implementation. Fig. 4 shows the reverse reduce tree to Fig. 2. The two binary trees consist of different set of internal nodes and leaf nodes. The internal nodes [L0, L1, L2] in the tree of Fig. 2 become leaf nodes in Fig. 4, while leaf nodes [L4, L5, L6] in Fig. 2 become internal nodes in the reverse reduce tree.

B. Turn-Round Scheduling

As shown in Fig. 3, the order of AllReduce call follows backpropagation sequence from Softmax - Conv3 - Conv2 to Conv1. Since the order of backpropagation is reverse to forwarding, worker nodes always get what they need to start forwarding in the end of transmission, which causes delay and affects training performance when communication speed becomes bottleneck. To solve this problem, we break the AllReduce call into two separated routings, reducing and broadcasting. In the reducing procedure, worker nodes aggregate their gradients to root nodes. After receiving reduced gradients from workers, the root nodes cache the reduced gradients and schedule the order of broadcasting to minimize delay at worker nodes.
For example in Fig. 3, the integrated version of AllReduce broadcasts the reduced gradients from Softmax to Conv1. In contrast, our separated AllReduce can cache the reduced gradients and reschedule the broadcasting order from Conv1 to Softmax, thus reduces communication delay by helping the worker nodes to get what they need in the beginning of transmission.

Formally, we formulate the reduced gradients to be broadcast as a vector \([0, 1, \ldots, K-1]\), where the index \(k\) of the array stands for network forwarding order executed by solvers like Caffe [8]. The order of gradients arriving at root node is reverse to forwarding, which is \([K-1, K-2, \ldots, 0]\). The target of our scheduling algorithm is to find an optimal transmission order \(S = [s_0, s_1, \ldots, s_{K-1}]\), where \(s_k\) is the time to broadcast layer \(k\).

Theoretically, we can build a model to calculate the optimal value of \(S\) if we know all the hardware configurations, network structures, batch sizes, etc. However, due to complicated dynamics in reduction trees, such as contentions, congestions and message queues, it is hard to accurately predict the scheduling results. In practice, we found a suboptimal algorithm, which is fast, robust and self-adaption, often works better in distributed systems, where the performances are frequently affected by random factors like overhead, random errors, etc.

To schedule packet transmissions and reduce delay, our algorithm tries to find a turn-round index \(M\) in the array, where gradients before \(M\) are broadcast in backpropagation order from \(M-1\) to 0 and gradients behind \(M\) are broadcast in forwarding order from \(M\) to \(K-1\). The algorithm overlaps broadcasting from \(M-1\) to 0 with backpropagation computations in the \(i\)th iteration, while broadcasting from \(M\) to \(K-1\) is overlapped with forwarding computations in the \((i+1)\)th iteration.

Intuitively, if \(M\) equals \(K\), the turn-round algorithm falls back to the original AllReduce implementation and overlaps all the communications with only backpropagation computations. On the other hand, if \(M\) equals 0, the algorithm sequentially broadcasts all the reduced gradients in the forwarding pass. In this case, if layer \(k\) happens to be small and its forwarding computations finish in a short time, probably the updates of layer \(k+1\) haven’t arrived and the node must wait for gradient updates in layer \(k+1\). As a result, there is an optimal turn-round index between 0 and \(K\) that minimizes the communication delay.

When a new version of reduced gradient is received by root node, if its index is less than \(M\), it is broadcast immediately by the root node. Otherwise, the gradient is buffered in the root node, and the buffered gradients are sequentially sent out after the reduced gradients of layer 0 are received. In runtime, our algorithm only needs to execute one single comparison, so it is very efficient to implement.

Fig. 5 shows a scheduling example to a six-layer neural network, where \(M\) equals three. The reduced gradients of first two layers \([0, 1]\) are sent in the backpropagation time of \(i\)th iteration, while gradients of the last four layers \([2, 3, 4, 5]\) are sent sequentially in the forwarding time of \((i+1)\)th iteration. The scheduling proposal gets performance gain by overlapping broadcasting of reduced gradients \([3, 4, 5]\) with forwarding computations in \((i+1)\)th iteration. Moreover, since broadcasting of gradients \([3, 4, 5]\) is moved from \(i\)th iteration to \((i+1)\)th iteration, the reducing speed of \(i\)th iteration increases due to less contentions in this stage.

![Fig. 5. The order of backward propagation in a simple neural network](image)

C. Scheduling Metrics

In the beginning of training, we initialize \(M\) to be \(\lfloor \frac{K}{2} \rfloor\), and the value of \(M\) is dynamically tuned in the training process. To find the optimal index \(M\), we begin by designing a new metric to identify which node is the bottleneck of the system. Given a node \(n\) in the reduction tree, for a layer indexed \(k\) at iteration \(i\), let \(t_{nk}^i\) be the time stamp it receives gradients from its left child, \(t_n^i\) be the time stamp it receives gradients from its right child, and \(t_{nk}^i\) be the time stamp it generates gradients at layer \(k\). The node calculates the maximum time stamp among \(t_{nk}^i\), \(t_n^i\), and \(t_{nk}^i\), and sends the corresponding node id \(d_n\) to its parent node. Recursively, the root node gets to know which node is the slowest at layer \(k\). To get statics information on which layer delays the computation of slow nodes, in addition to node id, we attach a value \(p_{nk}^i\) on how long the node’s forward computation is blocked by the layer.

After collecting information about slow nodes in the system, the turn-round algorithm decides whether to increase or decrease the value of \(M\) to reduce delay. For example, if the slow node is blocked by layers with indices smaller than \(M\), we need to decrease \(M\) to reduce contentions in the backpropagation pass. Otherwise, we increase \(M\) to reduce communication loads in the forwarding pass.

The performance of distributed system is affected by many factors. For example, if a node in the system gets overheated and runs slowly, it will become the bottleneck of the system and generate false alarms. In our system, since the gradients are transmitted in multiple fixed reduction trees in a predefined order, we can calculate which nodes have maximal delay. For example, in the seven-node cluster shown in Fig. 2 and Fig. 4, the nodes get delayed by communications are \(L4\) in Fig. 2 and \(L3\) in Fig. 4. The turn-round algorithm only counts the iterations in which the slow nodes matches calculation.

Taking all above mentioned cases into account, our turn-round algorithm increases or decreases the value of \(M\) by 1 every 10 iterations. Let \(P_{nk}^i = [p_{nk}^{i0}, \ldots, p_{nk}^{iK-1}]\) be the delay vector at node \(n\) of iteration \(i\), where \(n\) is one of the expected delaying nodes. For every 10 iterations, we calculate decreasing weights: \(P_- = \frac{1}{10} \times \sum_{i=0}^{10} \sum_{k=0}^{M} p_{nk}^{ik}\), and increasing weights: \(P_+ = \frac{1}{10} \times \sum_{i=0}^{10} \sum_{k=M}^{K} p_{nk}^{ik}\). We increase \(M\) by 1 if \(P_- > \xi\) and \(P_+ > \xi\), and we decrease \(M\) by 1 if \(P_- > \xi\) and \(P_- > \xi\), otherwise \(M\) keeps unchanged. \(\xi\) is the random drift in training time and set to be 0.05 \(\times T\), where \(T\) is the average training time.
III. PERFORMANCE EVALUATION

We implemented our algorithms on Caffe [8] and run our distributed Caffe on a 40-nodes Xeon cluster interconnected by 10Gb/s Ethernet card. Each node is equipped with 44 processor cores and 128 GB memory. Comparing with experiments in Ref. [9], we use slower network links and our system can be scaled to even larger scale when equipped with faster interconnections. We run ResNet-50 model in Ref. [1] to evaluate the performance of our algorithm.

We compare the performance our algorithm with FireCaffe in Ref. [3], which is the state of art distributed training system for deep neural networks. We implemented FireCaffe based on OpenMPI and multi-color reduction trees in Ref. [7], which are the state of art AllReduce implementations. Fig. 6 shows the average training speed of 100 iterations. We use a batch size of $b = 64$ per CPU node and measure the number of images the system can process each second as we add more nodes to the system.

As we can see from Fig. 6, our implementation of multicolor reduction tree AllReduce [7] is 18% faster than OpenMPI. The training efficiency of AllReduce decreases as the number of worker nodes grows, for the communication delay grows in the order of $O(\log n)$. On the other hand, since our scheduling algorithm proposed in this paper significantly reduced the communication delay and link contentions, the training throughput grows near linearly, which is 20% faster than the state of art AllReduce implementation in Ref. [7].

We followed the linear scaling rule from Ref. [9] and use a learning rate of $0.1 \times \frac{N \times b}{256}$ where $b = 64$ is the batch size at each node and $N$ is the number of nodes in the system. We used warm-start learn rate schedule in Ref. [9], where the initial learning rate is set to 0.1 and linearly increased to $0.1 \times \frac{N \times b}{256}$. The learning rate is reduced by $\frac{1}{10}$ at $30^{th}$, $60^{th}$ and $90^{th}$ epoch.

The test accuracy results of ResNet-50 [1] are plotted as a function of training time in Fig. 7. The ResNet-50 model is trained in a 40-node cluster. As we can see from Fig. 7, our algorithm can correctly train ResNet-50 to top-1 accuracy of 0.752 at 100 epochs in about 34 hours, which is about 30 times of acceleration comparing with one single Xeon node.

Fig. 6. Training Speed of different collective algorithms

Fig. 7. Validation top-1 accuracy over time in hours on 40-node cluster

IV. CONCLUSION

In this paper, we described a scheduling algorithm for scaling deep neural networks. By overlapping communication with both backpropagation and forwarding computations, we significantly reduced the delay and improved scaling performance. We also introduced a new separated asynchronous AllReduce implementation based on reverse reduce tree, which can help to reduce link contentions. Our techniques are general and can be integrated into frameworks like Caffe, Torch, etc.

REFERENCES