TensorExpress: In-Network Communication Scheduling for Distributed Deep Learning

Minkoo Kang, Gyeongsik Yang, Yeonho Yoo, Chuck Yoo
Department of Computer Science and Engineering, Korea University
{*mkkang, ksyang, yhyoo, chuckyoo*}*@os.korea.ac.kr*

***Abstract*—TensorExpress provides in-network communication scheduling for distributed deep learning (DDL). In cloud-based DDL, parameter communication over a network is a key bottleneck. Previous studies proposed tensor packet reordering approaches to reduce network blocking time. However, network contention still exists in DDL. TensorExpress mitigates network contention and reduces overall training time. It schedules tensor packets in-network using P4, a switch programming language. TensorExpress improves latency and network blocking time up to 2.5 and 2.44 times, respectively.**

***Keywords*-distributed deep learning; parameter server architecture; P4; communication scheduling; in-network delay;**

I. Introduction

State-of-the-art deep learning is now moving forward to distributed systems, known as distributed deep learning (DDL). This evolution is attributable to the dramatic increase of input data volume and deep-learning model complexity. DDL trains models in multiple GPU nodes (workers) via mini-batches and synchronizes the model parameters of neural networks at the end of each training iteration. As these DDL tasks require a large-scale distributed computing infrastructure with vast computing resources (e.g., GPU and network), DDL becomes a major workload on cloud [1].

Various DDL architectures have been proposed, and the most well-known is the parameter server (PS) architecture [2]. In this architecture, each worker node performs "forward propagation" (fp) from neural network layer 0 to N, resulting in the prediction value for the given input. Then "back propagation" (bp) sequentially calculates the local gradients for adjusting model parameters (e.g., weight and bias) from a loss function between the predicted value and real value. Worker nodes "push" these gradients to the PS. After the PS receives all the local gradients from workers, it updates the model parameters for the next training iteration. Then the worker "pulls" the updated parameters from the PS and continues its training. The unit for "push" and "pull" operations is called a tensor packet.

The duration of each training iteration depends on the time spent in computation (fp and bp) and network communication (push and pull). A recent study has shown that workers spent most of their training time in communication (network blocking time), even up to 92.8% [2]. Such delays are caused by two overheads: 1) network contention caused by limited bandwidth and bursty DDL traffic, and 2) inefficient communication without considering the computation order. Several studies have proposed communication scheduling [1], [3] to reduce the second overhead. At each worker, these studies change the transmission order of tensor packets to the order in line with the computing operation. For example, the tensor packets of bp at layer 0 are sent as soon as bp ends to avoid the delay in the DDL training time (§II-A).

However, communication scheduling does not reduce network contention. Although tensor packets are properly ordered from each host, in-network delay (e.g., queueing delay) is inevitable. To illustrate this, we run experiments with three workers that simultaneously send and receive tensor packets of three layers. Because tensor packets are accumulated from multiple workers, our results show that the in-network delay of tensor packets increases up to 3.44 times (§IV). Considering that recent deep-learning models consist of tens or hundreds of layers, this delay poses a critical challenge in DDL. Also, as the number of workers increases and their computing speed accelerates, this delay could represent an overwhelming portion of the DDL training time.

To address the problem, we propose TensorExpress, a novel in-network communication scheduling framework that reduces in-network delays of high-priority tensor packets. TensorExpress reduces the delay by making each switch sends tensor packets with its priority using multiple queues.

The challenge is the realization of in-network scheduling with traditional switches, which do not allow new packet processing logic to be implemented. Therefore, with a traditional switch, identifying tensor packets and their layers is impossible. Also, tensor packet scheduling according to the layer dependency cannot be realized. As a solution, we use P4 [4], a switch programming language, to implement such mechanisms in-network. P4 allows the implementation of custom packet processing logic in programmable switches,

*Corresponding author: Chuck Yoo.
†This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation grant funded by the Korea government (No. 2015-0-00280, (SW Starlab) Next generation cloud infra-software toward the guarantee of performance and security SLA). This research was also supported by National Research Foundation of Korea funded by the Ministry of Science, ICT (No. NRF-2019H1D8A2105513).

2159-6190/20/$31.00 ©2020 IEEE
DOI 10.1109/CLOUD49709.2020.00014

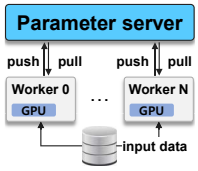

Figure 1: Distributed deep learning

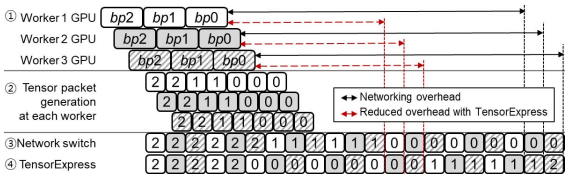

Figure 2: Communication scheduling

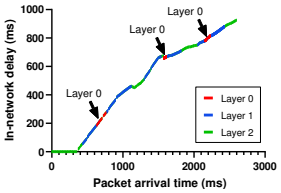

Figure 3: Motivation experiment

and so it is the proper architecture to realize in-network communication scheduling.

TensorExpress is composed of two parts: ticketer and TensorExpress switch. The ticketer runs in workers and writes the layer information of the model to the tensor packets. The TensorExpress switch, based on P4, then parses the information and schedules the tensor packets according to layer priority. The evaluation results show that the in-network delay of layer 0 (the layer with the highest priority) improves up to 2.5 times, and the network blocking time between training iterations is reduced by up to 2.4 times.

II. Background and Motivation

A. Distributed deep learning and network communication

Fig. 1 depicts the DDL system based on the PS architecture. Each worker node performs fp and bp operations, and the local gradient and parameters are communicated between workers and the PS. Fig. 2 shows a communication scheduling example. Let us assume that three workers train a deep-learning model consisting of three layers. Whenever the bp of a layer is finished (① in Fig. 2), the corresponding tensor packets are generated (②). These tensor packets are then sent to the PS via the network. In a traditional network switch, the tensor packets are processed first-in-first-out (③).

As shown in Fig. 2, the tensor packets of layer 0 (bp0) are generated last, but they should come back to the worker before tensor packets of other layers because the next training iteration needs updated parameters with bp0 tensor packets. However, in the traditional switch, the network blocking time (the black arrow in Fig. 2) increases. This delay is clearly due to network contention and the bursty generation of tensor packets.

B. In-network delays

To demonstrate the network contention problem, we run a well-known model called VGG16 for image classification and measure the in-network delay of tensor packets. Our measurement includes one PS and three workers, and they are connected through one bmv2 software switch. We simulate the network traffic of three layers (layer 0, layer 1, and layer 2) in the VGG16 model.

Fig. 3 shows the in-network delay of tensor packets in relation to arrival time. Because three workers send tensor packets, the group of layer 0 tensor packets (indicated by three arrows in Fig. 3) appears three times, and their in-network delay grows with the tensor arrival time. This is because the tensor packets of layers are generated from three workers in bursts, and so the packets from multiple workers are accumulated at the switch. On average, three workers incur delays for layer 0 tensor packets of 232 ms, 662 ms, and 797 ms. Additionally, three workers' network blocking time for the next training iteration (the time for the layer 0 tensor packets to arrive) are 316 ms, 698 ms, and 856 ms. These results show that although the tensor packets are sent according to the computation order, network contention can severely prolong the overall DDL training.

C. Enhancing in-network delays with P4

We use P4 to design the packet scheduling at a switch. P4 [5] is a switch programming language that allows the implementation of packet processing logic in programmable switches. Traditional switches have a rigid packet processing logic which cannot be modified. Specifically, we cannot recognize which packets are the tensor packets. Also, because the switches cannot distinguish the layers of the tensor packets, in-network communication scheduling is impossible. However, P4 can set and parse arbitrary packet header fields or implement a new packet queueing mechanism. Therefore, we can make the switch to parse packet header fields that contain a model's layer. Also, P4 allows us to define the packet matching over the model's layer and the subsequent queueing policy. To that end, we used P4 to design TensorExpress. With P4, we can realize ④ in Fig. 2 by sending layer 0 prior to the other layers.

III. Design

In this section, we explain the design of TensorExpress. Fig. 4 exhibits the TensorExpress architecture, which consists of two components: a ticketer and TensorExpress switch. First, the ticketer marks each tensor packet to indicate which layer it belongs to. We denote this metadata as a ticket. The ticketer uses 8 bits of the IP option field to store the layer ID. Second, the TensorExpress switch performs in-network communication scheduling. We design this switch based on a P4 architecture without additional hardwares, and as such, it can be deployed with any switch that is compatible with P4. The TensorExpress switch works in two steps: ticket parsing and priority window sliding.

Ticket parsing: The ticket parser parses the packet header and extracts the ticket. Then, the switch knows the layer ID to which the packet belongs. If the ticket is valid (i.e., a tensor packet), the TensorExpress switch schedules packets

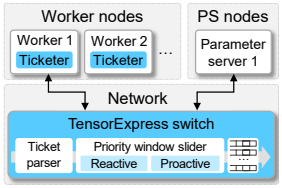

Figure 4: TensorExpress architecture

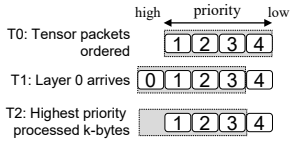

Figure 5: Priority sliding

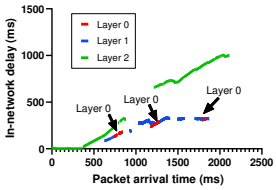

Figure 6: In-network delay

Table I: Network blocking time (ms)

Worker	1	2	3
No TensorExpress	315	697	856
TensorExpress	222	338	352
Improvement	29%	51%	59%

through priority window sliding, as explained next.

Priority window sliding: The TensorExpress switch uses priority queuing. This means that packets in a higher priority queue are always processed before those in a lower priority queue. The TensorExpress switch sets a higher priority to tensor packets from layer 0 because it should be transmitted before any other layers. Packets from layers 1 and 2 are assigned their priorities in order.

Because, in general, the number of layers is larger than the number of queues, we need a way to associate priorities with queues. TensorExpress introduces a priority window sliding that is designed to map priorities and layers. The design of priority window sliding comes from the observation that the bp goes through layers in descending order only, and so the packets entering the switch are also in that descending order. Therefore, when packets from a new layer arrive at the switch, the priority window updates the mapped layers of the priorities to include the new layer. We call this scheme reactive sliding. For example, in T0 of Fig. 5, suppose that the priority window is set for layers 1 to 4. If a tensor packet of layer 0 arrives, the priority window is shifted to layers 0 to 3 (T1). If the packets of the rear layers exist (e.g., layer 4), they are placed in the lowest priority queue.

Additionally, the TensorExpress switch provides proactive sliding. Proactive sliding shifts the priority window to map the highest priority queue when layer 0 packets arrive (before enqueue). The reason for proactive sliding is that the highest priority queue might have layer 1 packets that were inserted before the priority window shifted, and layer 1 packets are waiting to be sent. Then, if the layer 0 packet is enqueued, it must wait until the layer 1 packets are sent, which causes an additional delay. To reduce this delay, we proactively shift the priority window if the k-bytes of packets in the highest priority queue are sent (T2 in Fig. 5). So, TensorExpress can process the highest priority packets (e.g., layer 0) without an additional queueing delay.

After receiving layer 0 packets, meaning that all the bp operations have finished, the priority window shifts in the opposite direction, supporting the packets for the remaining tensor packets until the network communication finishes.

IV. Evaluation

We evaluate TensorExpress using bmv2 [5]. The evaluation environment and traffic are the same as the experiment described in §II-B. Fig. 6 shows the in-network delay when TensorExpress is applied. Compared to Fig. 3 (where TensorExpress is not applied), layer 0 of workers 1, 2, and 3 experiences 50% less in-network delay. This is because the tensor packets of layer 0, which is the frontmost layer of the deep-learning model, are processed sooner than other layers in the switch. Similarly, layer 1 achieves the improvement of 49%. In contrast, the in-network delay of layer 2 increases 11.5% on average. However, in Fig. 6, the delay of layer 2 overlaps with the fp process of layers 0 and 1, so it does not cause an additional delay in the entire DDL training process.

Table I shows the network blocking time, which is the time taken to send and receive all tensor packets of layer 0. Compared to the network blocking time in Fig. 3, TensorExpress reduces the network blocking time by 46% on average. The maximum reduction is 59%, as indicated by worker 3, which sends layer 0 tensor packets when more tensor packets are present on the network. Considering that a recent study [2] shows that network blocking time consumes up to 92.8% of the entire training process, this improvement could effectively speed up the overall training time.

V. Conclusions and Future Study

This paper presents TensorExpress, a P4-based in-network communication scheduling framework. In our evaluation, the in-network delay of layer 0 improves 2.5-fold, and the network blocking time between training iterations is reduced by up to 59%. As a future study, we plan to integrate TensorExpress with deep-learning libraries like TensorFlow.

References

[1] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[2] C. Chen, W. Wang, and B. Li, "Round-robin synchronization: Mitigating communication bottlenecks in parameter servers," in *IEEE Conference on Computer Communications 2019*. IEEE, 2019, pp. 532–540.

[3] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "TicTac: Accelerating distributed deep learning with communication scheduling," *2019 Conference on Systems and Machine Learning (SysML'19)*, 2019.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[5] p4lang, "p4lang/behavioral-model," May 2020. [Online]. Available: https://github.com/p4lang/behavioral-model