PA-ATP: Progress-Aware Transmission Protocol for In-Network Aggregation

Zhaoyi Li†, Jiawei Huang†, Tao Zhang‡, Shengwen Zhou†, Qile Wang†,
Yijun Li†, Jingling Liu†, Wanchun Jiang†, Jianxin Wang†
†Central South University, Changsha, Hunan, China.
Email: {lizhaoyi, jiaweihuang, zhousw, qilewang, yijunli, jinglingliu, jiangwc, jxwang}@csu.edu.cn
‡Changsha University, Changsha, Hunan, China. Email: tzhang@ccsu.edu.cn

2023 IEEE 31st International Conference on Network Protocols (ICNP) | 979-8-3503-0322-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICNP59255.2023.10355636

***Abstract*—Large-scale machine learning typically adopts distributed machine learning (DML) techniques to accelerate model training. Due to the large communication overhead, unfortunately, the phase of gradient aggregation has become the performance bottleneck for DML. To reduce traffic volume, several in-network aggregation (INA) transmission protocols are proposed to offload gradient aggregation function into the programmable switches. However, since existing INA transmission protocols use synchronous congestion control mechanism to drive each round of gradient aggregation, the straggling workers lead to long iteration time and significant performance degradation.**

To solve the above problem, we propose PA-ATP, a progress-aware INA transmission protocol, which adopts the progress-aware asynchronous congestion control. PA-ATP adjusts the sending rate in accordance with the transmission progress, allowing the straggling flow to grab more bandwidth than the leading flow and control the asynchronous degree of straggling job. We use a P4 programmable switch and a kernel-bypass protocol stack to implement PA-ATP. The results of testbed and large-scale NS3 simulations show that PA-ATP reduces training time by up to 62% compared to the state-of-the-art INA transmission protocols.

***Index Terms*—In-network aggregation, transmission protocol, straggler**

I. Introduction

In recent years, machine learning (ML) is widely utilized in various fields, such as speech recognition, natural language processing and autonomous vehicles. With the huge dataset size and very complex model of ML (e.g., Switch Transformer [1] has 1.6 trillion parameters), it becomes hard to complete the training in acceptable time by using single-host computation [2]. Therefore, the practitioners usually adopt distributed machine learning (DML) technology to deploy ML models in data centers and use multiple workers with GPUs for parallel training to reduce the training time.

In each training iteration of DML, all of the workers need to synchronize parameters to guarantee the convergence accuracy. During such communication phase, a large amount of traffic is transferred among multiple workers. Therefore, the communication becomes a performance bottleneck of DML (e.g., LSTM [3] spends 94% of its iteration time in communication at 10Gbps with 8 workers [4]). Recently, a series of in-network aggregation (INA) solutions [4]–[6] use programmable switches to aggregate gradients, therefore significantly reducing the traffic volume to PS. The main mechanism of INA is that each worker of the ML job streams its gradient to the programmable switch, which organizes the memory into an array of aggregators to store and aggregate the gradients from different workers, and finally broadcasts the aggregated gradients back to all workers.

Unfortunately, the existing INA protocols suffer from the straggler problem due to adopting a typical barrier-synchronized congestion control to aggregate gradients from all workers. Specifically, in each round of gradient aggregation, all workers upload their own gradients to the aggregating switch. After receiving all gradients with the same index, the aggregator broadcasts one aggregated gradient as ACK to all workers, synchronously increasing congestion windows of all workers. However, due to the ACK-clocking and synchronous window adjustment, the congestion windows of fast workers are suppressed by a few slow ones. This phenomenon is called as Straggler Problem, which commonly occurs due to dynamic bandwidth contention under inter-rack traffic. Since a stalled worker leads to the degradation of sending rate of all the other workers, it easily becomes the bottleneck of INA.

One intuitive choice is to leverage asynchronous congestion control to let the aggregator immediately reply an ACK to the corresponding worker once it receives a gradient. Thus, the workers could adjust the window size independently to mitigate the straggler problem. However, as we reveal in this paper, the simple adoption of asynchronous congestion control still suffers from low efficiency due to progress-agnostic and asynchronization-unlimited transmission in the presence of severe congestion and insufficient switch memory. (i.e. $\sim$10MB in Barefoot Wedge 100BF-32X switch).

In this work, to resolve the straggler problem of INA, we argue that the synchronous congestion control of current INA protocols should be changed into asynchronous. Furthermore, the workers should share the transmission-progress information and elaborately control the sending rate to coordinate the transmission progresses of all workers, especially under heavy contention scenario. Besides, the maximum asynchronous degree for each job should be carefully controlled to avoid severe aggregator overflow. We list our contributions as follows:

- We conduct an empirical study to exploit why the existing INA transmission protocols suffer from serious performance degradation in large-scale distributed machine learning. We reveal that the both synchronous and asynchronous congestion control lead to suboptimal performance of INA under the impact of frequent straggler.

- We propose a Progress-Aware in-network Aggregation Transmission Protocol (PA-ATP). First, we introduce the asynchronous congestion control into INA, so as to break the synchronous window adjustment and avoid the transmission stagnation due to stalled workers. Second, based on the shared transmission-progress information through the aggregator-driven coordination, the leading workers relinquish their bandwidth to the straggling ones, accelerating the whole process of gradient aggregation. Third, the maximum asynchronous degree of each job is adaptively adjusted in accordance with aggregator occupancy level, improving the aggregation efficiency.
- We implement PA-ATP in the P4-programmable switch and end host with kernel-bypass protocol stack. We evaluate PA-ATP in a real-world testbed and a large-scale simulation. The experimental results show that PA-ATP effectively reduces the iteration time by up to 62% compared with the state-of-the-art INA solutions.

The rest of this paper is organized as follows. In Section II and III, we describe the motivation and design of PA-ATP, respectively. PA-ATP's implementation is given in Section IV. In Section V and VI, we show the test results of real testbed and NS3 simulation experiments. We discuss the related works in Section VII, and conclude this paper in Section VIII.

II. Background and Motivation

A. Background

1) Distributed machine learning: With the rapidly increasing scale of ML, the model training time also increases significantly. To accelerate the training progress, the commonly used DML technology is data parallelism, which divides the dataset into multiple workers for parallel training. Specifically, each worker maintains the same ML model and then performs iterative training. After one round of iteration, the parameters among all of the workers are synchronized (called as gradient aggregation) to ensure high convergence accuracy.

Commonly, Parameters Server (PS) [7] or Ring-AllReduce [8] communication patterns are used for gradient aggregation. However, the above two communication patterns generate a large amount of traffic among multiple workers, resulting in network congestion. Consequently, the communication phase becomes the bottleneck of DML training.

2) In-network aggregation: To reduce the communication overhead, a series of INA protocols have been proposed in recent years [4]–[6]. By offloading the gradient aggregation function into the programmable switches, these solutions achieve lower communication cost than existing frameworks PS and Ring-AllReduce [4], [5]. Specifically, each worker first divides the gradient into the segments of fixed size, and each segment is assigned a unique sequence number. These fixed-size segments are sent to the switch for aggregation. At the switch, the memory is organized into an array of aggregators, each of which has a unique index number. Gradients from different workers with the same sequence number are aggregated in a same aggregator. When all the gradients with the same sequence number have been aggregated in the aggregator, the switch sends the aggregation results back to each worker (in SwitchML [4] and PANAMA [6]) or PS (in ATP [5], PS also processes the gradients which miss aggregators and broadcast the aggregation results to each worker). As a result, the amount of traffic is greatly reduced, alleviating the network congestion.

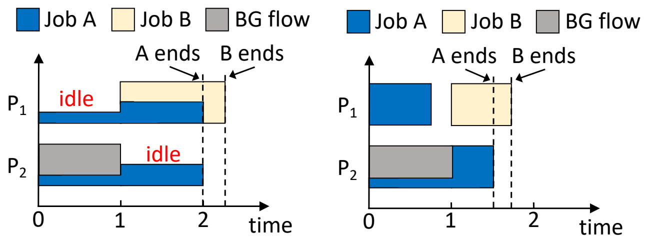
Moreover, to provide the reliability of INA, the existing INA protocols generally adopt the ACK-clocking mechanism similar to TCP. Each worker maintains a sliding congestion window with a size less than bandwidth-delay product (BDP) and uses the aggregated gradients as ACKs. The sequence numbers of aggregated gradients are tracked for loss recovery. Besides, each worker triggers the transmission of next round once receiving an aggregated gradient.

B. Motivation

1) Straggler problem in INA: In INA, the aggregator needs to wait for all worker's gradients before completing the aggregation and broadcasting the aggregated gradient to all of the workers. Therefore, if the gradient of one worker arrives later than the other workers, the job suffers from straggler problem, which is common due to dynamic bandwidth contention in large-scale data centers. Specifically, a large-scale cluster hosts a variety of jobs, which generate a large number of flows competing inter-rack links, resulting in non-uniform host-to-host available bandwidth [9], [10]. For example, [10] observes that different pairs of hosts in real datacenter (e.g., EC2) deliver various throughput and volatile latency, and the faster pairs have more than 2x throughput of slower ones. The gradients in the same training job will experience varying levels of congestion and queueing delays as traveling through different paths to the aggregator, leading to straggler problem.

It is possible to actively prevent straggler problem through flow scheduling and routing. For example, if we have prior knowledge of ML job workloads and traffic patterns, the flows can be allocated to uncongested paths to avoid bandwidth contention and eliminate stragglers. However, in shared clusters, these ML flows also compete for bandwidth with non-ML flows [5], [6], which are highly dynamic and hard to predict. Therefore, the straggler still frequently occurs in this scenario.

2) Synchronous congestion control vs. asynchronous congestion control: The existing INA protocols adopt synchronous congestion control mechanism. Specifically, the aggregation results are treated as ACKs, where its sequence number are tracked for loss recovery. Once receiving an 'ACK', each worker will send out a new gradient for aggregation. All workers synchronously slide and adjust their respective congestion windows. If any worker becomes stalled, all the other workers will synchronously slow down their sending rates, even their links connected to PS are idle. We use an example under dynamic bandwidth contention to illustrate this problem. We assume two jobs A and B, each of which aggregates gradient sized of $\frac{3}{4}$ unit. Job A has two workers deployed on two servers, which start transmitting gradients on path P_1 and P_2 at time $t = 0$. Job B has only one worker starting transmission at time $t = 1$. A background flow preempting $\frac{3}{4}$ bandwidth of P_2 starts at $t = 0$ and stops at $t = 1$.



(a) Synchronization (AJCT=1.625) (b) Asynchronization (AJCT=1.125)

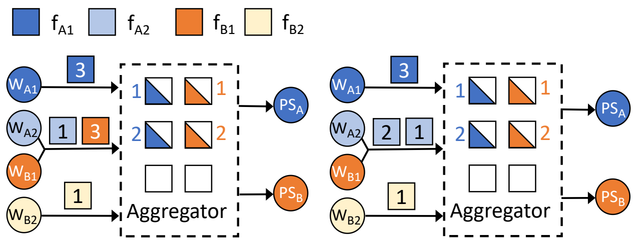
Fig. 1: Synchronous vs. asynchronous congestion control.

As shown in Fig. 1(a), due to rate limitation of the stalled worker during the time $t = 0{\sim}1$, job A on P_1 also only uses $\frac{1}{4}$ of the full bandwidth. During the time $t = 1{\sim}2$, job B competes the bandwidth with job A in P_1, and the job completion time (JCT) of both jobs increase. The average JCT of A and B is 1.625 time units.

An intuitive solution is to utilize asynchronous congestion control for INA. The aggregator generates an ACK to the corresponding worker immediately after receiving a gradient, triggering the next gradient. Therefore, even if one worker suffers from straggler, other workers can still send gradients with best effort to the aggregator. As shown in Fig. 1(b), though the flows in P_2 are blocked by the background traffic, job A fully uses the bandwidth of P_1. Since the worker of job A completes transmission on P_1 before job B starts, job B utilizes the full bandwidth of P_1, reducing the average JCT of A and B to 1.125 time units.

3) Low efficiency under strawman-asynchronous congestion control: Though asynchronous congestion control mechanism achieves low average JCT in the above scenario, the strawman proposal using progress-agnostic and asynchronization-unlimited manners still lead to low aggregation throughput.

Progress-agnostic manner. When multiple jobs share the bottleneck link, the leading flow competes with the straggling flow for bandwidth, resulting in the low efficiency (a flow is defined as a sequence of packets from a worker to PS). We use an example to show the problem of progress-agnostic manner under strawman-asynchronous congestion control.

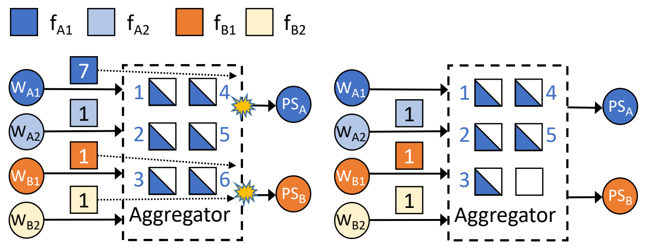


(a) Progress Agnostic (b) Progress Aware

Fig. 2: Progress-agnostic manner vs. progress-aware manner.

As shown in Fig. 2(a), there are two jobs A and B with straggler problem, each of which has two workers. The stalled flow f_{A2} of job A competes the link bandwidth with the leading flow f_{B1} of job B, and the link capacity is 2 packets. Each of the leading flows f_{A1} and f_{B1} has sent 2 gradients to the switch to wait for aggregation. Under the progress-agnostic manner, f_{B1} fairly occupies half of the bandwidth, and the 3rd gradient will be cached into the aggregator until the stalled one arrives. Assuming that the stalled flow f_{B2} only can send 1 gradient to aggregator in this round. Thus, each of jobs A and B only aggregates only one gradient. Fig. 2(b) shows the progress-aware manner. Since the aggregation throughput of a job is determined by the rate of the most stalled flow, the leading flow should release the bandwidth for straggling one. Thus, the leading flow f_{B1} relinquishes bandwidth to the stalled flow f_{A2}. Then, job A aggregates 2 packets in this round, while the aggregation throughput of job B is still 1. Compare with progress-agnostic manner, the overall aggregation throughput increases from 2 to 3.

Asynchronization-unlimited manner. When multiple jobs compete for the aggregators, the jobs with straggler occupy the aggregators without limit and do not release the aggregator resources until the most stalled gradient arrives, resulting in severe aggregator overflow. Thus, the newly arrived gradients have no available aggregators, generating large volume of non-aggregated traffic and leading to low aggregation efficiency. We use another example to illustrate this problem.



(a) Asyn. Unlimited (b) Asyn. Limited

Fig. 3: Asyn. unlimited manner vs. asyn. limited manner.

As shown in Fig. 3(a), there are two jobs, each of which has 2 workers. Job A is straggler while job B is non-straggler. The number of total aggregators are 6 and the link capacity is 1 packet. Under asynchronization-unlimited manner, the job A consumes all aggregators to cache 6 leading gradients. However, the 7th gradient of f_{A1} and the 1st gradient of f_{B1} and f_{B2} miss the switch aggregators, and these gradients have to be uploaded to PS directly for aggregation, resulting in link congestion and low aggregation efficiency. Fig. 3(b) shows the asynchronization-limited manner. The asynchronous degree for jobs with straggler is limited to 5, avoiding the link congestion caused by aggregator overflow and leaving the one necessary aggregator for job B. The aggregation throughput of switch is increased from 1 to 2.

4) Summary: Based on the above studies, we make the following conclusions: (i) Due to dynamic bandwidth contention under inter-rack traffic and multi-jobs sharing of the same end-host, the straggler problem is very common in large-scale distributed machine learning system. (ii) Since the existing INA transmission protocols use the synchronous congestion control mechanism, the straggler leads to the serious link under-utilization. (iii) Though the strawman-asynchronous congestion control mechanism can break the synchronized ACK-clocking, due to the progress-agnostic and asynchronization-unlimited manners, the strawman proposal still suffers from low aggregation efficiency.

III. PA-ATP Design

A. Design Overview

To tackle the straggler problem of existing INA transmission protocols, we propose a progress-aware asynchronous congestion control mechanism PA-ATP, achieving low job completion time. However, the design and implementation involve several key challenges. (i) The workers should share the progress information and have the global information whether they are the stalled or leading. (ii) The sending rate of each worker should be elaborately coordinated to release the proper bandwidth from the leading flows for the stalled ones. (iii) The asynchronous degree of each job should be controlled to avoid severe aggregator overflow.

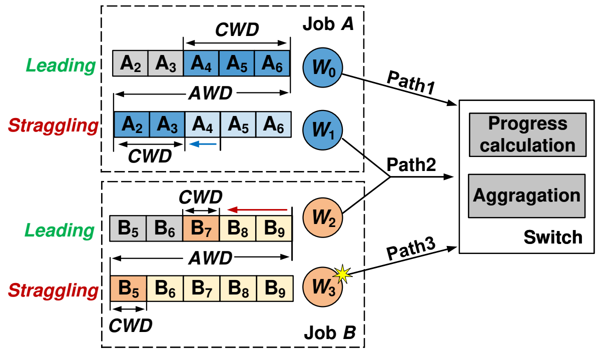


Fig. 4: Examples of congestion control behavior in PA-ATP.

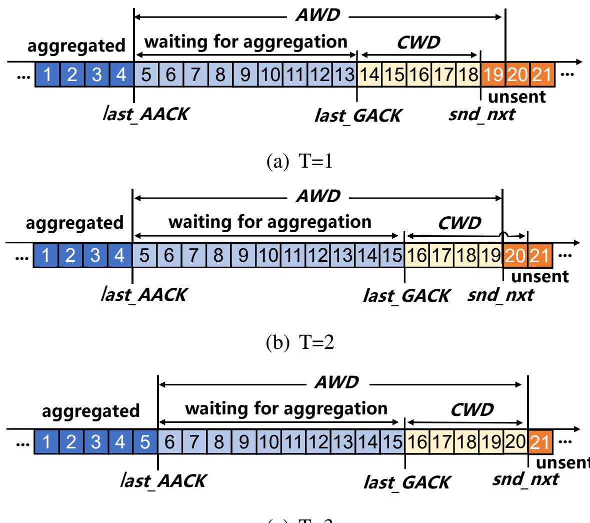
We use an example to illustrate how PA-ATP works. As shown in Fig. 4, job A and job B have two workers, respectively, and there are three paths between the workers and the aggregating switch. The path is determined by routing schemes, such as ECMP. For example, workers W_1 and W_2 share $Path2$. We assume that $Path1$ and $Path2$ have link capacity of 3, and the available bandwidth of $Path3$ is 1 due to background traffic. The switch calculates the average progress of each job and embeds it into the ACK header. After receiving the ACK packets, each worker knows whether it is stalled or leading according to its own progress and the average progress of the job. In job A, W_0 and W_1 have sent $A_1{\sim}A_3$ and A_1, respectively. Thus, as the straggler in job A, W_1 gently reduces its congestion window CWD from 3 to 2 after detecting congestion. W_2 and W_3 have finished transmission of $B_1{\sim}B_6$ and $B_1{\sim}B_4$, respectively. As the leading worker in job B, W_2 aggressively reduces its congestion window from 3 to 1. Therefore, more bandwidth is allocated for W_1 than W_2. Through accelerating the straggler, jobs A and B achieve a higher aggregation efficiency. Besides, each job maintains an asynchronous degree window AWD to limit the maximum difference between the progress of leading worker and straggling worker. In this case, we assume that the AWD sizes of Job A and Job B are 6 to avoid severe aggregator overflows. Thus, A_7 and B_{10} cannot be sent before completing the aggregation of gradients A_2 and B_5, respectively. AWD is updated adaptively in accordance with the aggregator occupancy level.

Note that, PA-ATP only uses asynchronous congestion control and still follows the global synchronous aggregation, thus avoiding the staleness problem caused by asynchronous updates methods such as ASP [11], SSP [12], etc.

B. Design Detail

Our design consists of following three modules: dual sliding windows, progress-aware congestion control, and asynchronous degree control.

1) Dual sliding windows: PA-ATP uses congestion window CWD and asynchronous degree window AWD to allocate the link bandwidth for each worker and control the maximum asynchronous degree for each job, respectively. Moreover, PA-ATP generates two types of ACK packet, gradient ACK (GACK) and aggregation ACK (AACK). When the aggregating switch receives a gradient into aggregator (or PS receives the non-aggregated gradient due to aggregator overflow), it immediately replies one GACK packet to the corresponding worker, triggering the sliding of CWD. After receiving the gradients with the same sequence number from all workers, the aggregating switch forwards the aggregation result to PS. Then, PS generates an AACK that carried the aggregation result and broadcasts it back to all workers, triggering the sliding of AWD. If there are no idle aggregators, the overflowed gradients will be directly forwarded to PS for aggregation. Besides, to guarantee the transmission reliability of INA, once a sequence number gap in AACK is detected, the workers quickly retransmit the lost gradients without waiting for the timeout of failing aggregation. Similar to ATP, the retransmission gradient triggered by the gap of AACK will release the partitioned aggregation result in aggregator and be directly sent to PS, avoiding the switch memory leak [5].



(a) T=1

(b) T=2

(c) T=3

Fig. 5: PA-ATP window structure at the worker.

Fig. 5 shows the dual sliding windows maintained by each worker. We assume that the sizes of AWD and CWD are fixed as 15 and 5, respectively. The worker tracks whether the corresponding gradient has been aggregated according to $last\_AACK$, which is the latest sequence number of aggregated gradient confirmed continuously. $last\_GACK$ is the latest sequence number of gradient confirmed continuously. snd_nxt is the next gradient sequence number to be transmitted, which is limited by smaller value between the right boundary of the two windows CWD and AWD. Specifically, at time T=0, since the right boundary of CWD is smaller than that of AWD, snd_nxt is limited to 18, avoiding the link congestion. At time T=1, the worker receives 2 $last_GACK$s, and thus CWD slides 2 packet. However, the worker only sends 1 packet due to the smaller right boundary of AWD as 19, preventing the excessively aggregator overflow. At time T=3, after the worker receives a $last\_AACK$, AWD slides 1 packet and the worker immediately sends the 20th packet. Note that, for non-straggler job, the difference between $last\_AACK$ and $last\_GACK$ is going to be zero.

2) Progress-aware congestion control: In this module, PA-ATP uses two signals including progress degree and ECN feedback to guide the adjustment of congestion windows.

Progress degree. We use progress degree p to denote the relative transmission progress of the current worker in all N workers of the current job. Each worker calculates p according to its local progress and average progress of all N workers. We define the local progress p_l as $last\_GACK - last\_AACK$, indicating the number of gradients waiting for aggregation in the switch. For each job, the switch maintains the last gradient's sequence number s_n of the nth worker. Once receiving a gradient, the switch updates s_n and calculates the sum of sequence number s_{sum} as $s_{sum} = \sum_{n=1}^{N} s_n$. Then s_{sum} is carried in GACK's header and sent back to all workers. Each worker gets the average sequence number s_{avg} as $s_{avg} = s_{sum}/N$. The average progress p_{avg} is calculated as $p_{avg} = s_{avg} - last\_AACK$. Thus, the progress degree p is calculated as $p = \frac{p_l}{p_{avg}}$. The range of progress degree is $p \in [0, p_{max}]$, where p_{max} is calculated as $p_{max} = AWD/p_{avg}$. Therefore, if p is close to 0, it means that the flow is straggler and relatively slower than other flows in the job. On the contrary, p close to p_{max} indicates that the flow is relatively leading compared to others.

ECN feedback. To speed up the large-scale distributed machine learning, the gradients from workers are aggregated in a multi-rack/multi-switch cluster. In the large-scale deployment, the gradients need to go through multiple switches to the aggregating switch. PA-ATP adopts ECN to sense the end-to-end network congestion level, similar to DCTCP [13]. If the queue length at any switch exceeds the ECN marking threshold, the arriving gradient will be marked with the CE code-point. The CE marked is carried in the GACK's header and delivered to the corresponding worker. Specifically, the congestion level α_{i+1} of $(i+1)$th RTT is updated as $\alpha_{i+1} = (1-f) \times \alpha_i + f \times e_i$, where e_i is the ratio of packets marked in the recent RTT round and f is a weight factor.

Update of congestion window. We integrate the progress degree into the congestion control algorithm and each worker independently updates its congestion window. Once receiving a GACK packet, the worker will increase its window by one. When network congestion occurs, the leading flow will cut the window more aggressively than the straggling one.

Specifically, the window decreasing factor d_i is determined by congestion level α_i and progress degree p as $d_i = \alpha_i^{1-p/p_{max}}$. PA-ATP reduces the congestion window of ith RTT according to decreasing factor d_i, as follows:

$$CWD_{i+1} = CWD_i \times (1 - \frac{d_i}{2}), \tag{1}$$

When α_i is close to 1, the congestion window is halved regardless of whether the flow is leading or straggling. For progress degree p, if p approaches p_{max}, the congestion window gets halved no matter how the congestion level is. If the straggling flow has a small p close to 0, the congestion window will be reduced in accordance with α_i. Thus, the sending rates of leading flows will be reduced more aggressively, enabling the leading flows release bandwidth for the straggling ones. The most straggling flow in a job will get a decreasing factor of the congestion window similar to that of DCTCP. Thus, the background traffic (which commonly adopts DCTCP in datacenter) can still obtain a fair shared bandwidth.

3) Asynchronous degree control: To limit the maximum asynchronous degree for each job, PA-ATP adopts asynchronous degree control module, which adjusts the asynchronous degree window AWD according to the aggregator overflow signal AECN.

AECN feedback. PA-ATP uses the proportion of AECN marking to indicate the aggregator occupancy level. At the switch, when the number of used aggregators and overflowed gradients exceeds $\eta \times (M + BDP)$ (M is the total number of aggregators and BDP is the Bandwidth-Delay Product in packets), the next arrived AACK is marked as AECN (1bit in packet header). Specifically, the aggregator occupancy level β_{i+1} of $(i+1)$th RTT round is updated as $\beta_{i+1} = (1-w) \times \beta_i + w \times g_i$, where g_i is the ratio of packets marked with AECN in the last round and w is a weight factor.

Update of asynchronous degree window. To address the severe aggregator overflow problem of strawman asynchronous proposal, each job maintains the asynchronous degree window AWD. When no AACK is marked as AECN, AWD is additively increased by one per RTT round. When aggregator congestion occurs, PA-ATP multiplicatively reduces AWD as:

$$AWD_{i+1} = AWD_i \times (1 - \frac{\beta_i}{2}). \tag{2}$$

C. Model Analysis

We analyze the behavior of the asynchronous degree control of PA-ATP in the steady state. We consider a total number of aggregators M and the bandwidth-delay product BDP shared by N jobs with straggler. Assuming that the total aggregation throughput is C_A, which is equal to the sum of most stalled workers' rate of all jobs. Besides, we assume that the most leading worker of each straggling job will transmit $AWD(t)$ packets per round, and each job changes AWD synchronously and follows the same sawtooth. This sawtooth is quantified as the number of total $AWD(t)$ size R_{max}, the amplitude of the number of packets A, and the period of oscillation T. Therefore, $R(t)$ at time t is given by:

$$R(t) = N \times AWD(t). \tag{3}$$

In each sawtooth period, $R(t)$ in the last round will exceed the $AECN$ marking threshold $\eta \times (M + BDP)$ and then be reduced according to the $AECN$ marking ratio β. Let $AWD^* = \eta \times (M + BDP)/N$ be the boundary value that the system starts to be overloaded, and AWD^* continues to increase by one to $AWD^* + 1$. Thus, the window amplitude for a single job S is:

$$S = (AWD^* + 1) - (AWD^* + 1)(1 - \frac{\beta}{2}). \tag{4}$$

According to [13], β is approximated as $\beta \approx \sqrt{2/AWD^*}$. As there are N jobs, we get:

$$\begin{aligned} A = NS = N(AWD^* + 1)\frac{\beta}{2} &\approx \frac{N}{2}\sqrt{2AWD^*} \\ &= \frac{1}{2}\sqrt{2N\eta(M + BDP)}. \end{aligned} \tag{5}$$

And the oscillation period is given by:

$$T = S = \frac{1}{2}\sqrt{2\eta(M + BDP)/N}. \tag{6}$$

According to Eq.(3), the maximum number of sent packets R_{max} is defined as:

$$R_{max} = N(AWD^* + 1) = \eta(M + BDP) + N. \tag{7}$$

To avoid overflowed packets block aggregation traffic, the following constraints should be satisfied:

$$R_{max} < M + BDP - C_A. \tag{8}$$

Plugging Eq.(7) and Eq.(8), we get the upper bound of threshold η as:

$$\eta < 1 - \frac{C_A + N}{M + BDP}. \tag{9}$$

We next analyze the throughput of ATP and PA-ATP. Due to synchronous congestion control, the throughput of ATP is always equal to aggregation throughput C_A as $Thr_{atp} = C_A$. However, under PA-ATP, when $R(t)$ is larger than M, it utilizes the idle bandwidth to upload the leading gradients to PS. To maximize the throughput, η is set to its upper bound and thus R_{max} is equal to $M + BDP - C_A$. Then, we calculate the average throughput Thr_{pa} of PA-ATP in each sawtooth period in the steady state. According to Eq.(5), the $R(t)$'s lower bound R_{min} is defined as:

$$\begin{aligned} R_{min} &= R_{max} - A \\ &= R_{max} - \frac{1}{2}\sqrt{2N(R_{max} - N)}. \end{aligned} \tag{10}$$

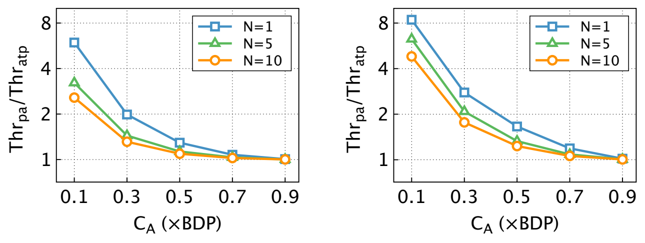
Thus, Thr_{pa} is the average throughput during the increase of $R(t)$ from R_{min} to R_{max}. When $R_{min} \geq M$, we have:

$$\begin{aligned} Thr_{pa} &= \frac{R_{max} + R_{min}}{2} - M + C_A \\ &= R_{max} - \frac{1}{4}\sqrt{2N(R_{max} - N)} - M + C_A. \end{aligned} \tag{11}$$

When $R_{min} < M$, the average throughput is:

$$\begin{aligned} Thr_{pa} &= \frac{(R_{max} - M)/N}{T} \times (\frac{R_{max} + M}{2} - M) + C_A \\ &= \frac{(R_{max} - M)^2}{\sqrt{2N(R_{max} - N)}} + C_A. \end{aligned} \tag{12}$$

Fig. 6(a) and Fig. 6(b) show the theoretical gain of throughput under different bandwidth (40Gbps and 100Gbps), respectively. The round-trip time is set to 10μs, so BDP sizes for 40Gbps and 100Gbps are 166 and 416 packets ($\sim$300Bytes per packet in INA), respectively. The aggregation throughput C_A varies from 0.1BDP to 0.9BDP. The number of aggregators M is 40K. The results show that PA-ATP achieves greater gains under higher bandwidths. Besides, the gain of throughput decreases when C_A increases due to less idle bandwidth in PA-ATP. Moreover, the larger number of jobs leads to an increase of the window amplitude and thus the gain decreases.



(a) Bandwidth=40Gbps (b) Bandwidth=100Gbps

Fig. 6: Throughput gain under varying aggregation throughput.

IV. Implementation

End host. Similar to ATP, we implement PA-ATP as a plugin of BytePS, which sets up the context for workers to communicate with the PS. We modify the $Push()$ operation in BytePS and utilize PA-ATP for gradient aggregation. Since BytePS is integrated in DL framework such as Pytorch, we use $register()$ that provided by Pytorch to register $Push()$ operation, which will be called after the calculation of each layer's parameters is completed. Moreover, we implement the packets handling, progress detection, congestion control, and loss recovery operations in the user space, which introduces CPU overhead. However, PA-ATP uses UDP-like RDMA Raw Ethernet verbs to send/recv packets, bypassing the kernel space. It is still beneficial from high throughput and low latency of RDMA. We employ the ATP header [5] to deliver information for gradient aggregation. Meanwhile, we add extra 32bits into the packet header, where 1bit ($isGACK$ field) is used to identify whether the packet is a GACK, and the remaining 31bits ($PROG$ field) is used to carry the progress information of transmission.

Switch. The progress calculation, dual acknowledgments, and in-network aggregation logic of PA-ATP are implemented on the Wedge 100BF-32X programmable switch with P4 language. Fig. 7 shows the ingress pipeline for packets processing, which consists of multiple match-action tables. Specifically, the packet first matches the $Read\_Write\_Register\_S_n$

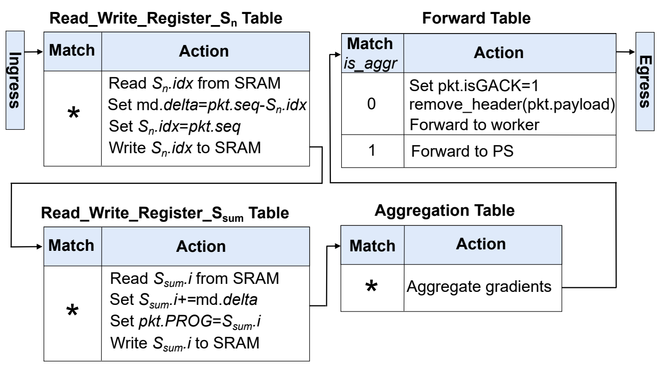


Fig. 7: P4 implementation.

table. The register $S_n.idx$ stores the last sequence number of idxth worker. The difference $md.delta$ between $S_n.idx$ and the sequence number of the current packet $pkt.seq$ is calculated, and then $S_n.idx$ is updated with $pkt.seq$. In $Read\_Write\_Register\_S_{sum}$ table, the register $S_{sum}.i$ stores the sum of the sequence number of all workers in ith job. $S_{sum}.i$ is added by $md.delta$ and then $S_{sum}.i$ is embed in $pkt.PROG$ field of the packet header. In $Aggregation$ table, the aggregating operations are similar to ATP, where the gradients with the same sequence number are accumulated in the same register. In $Forward$ table, is_aggr denotes whether the aggregation completes. If the aggregation doesn't complete, is_aggr is zero, and PA-ATP will remove the packet payload $pkt.payload$ through the $remove\_header$ primitive and return the packet header to the corresponding sender as a GACK, unlike SwitchML, ATP and other synchronization schemes that drop the packet. If the aggregation completes, the aggregated gradient is forwarded to PS.

V. Testbed Evaluation

A. Testbed Setup

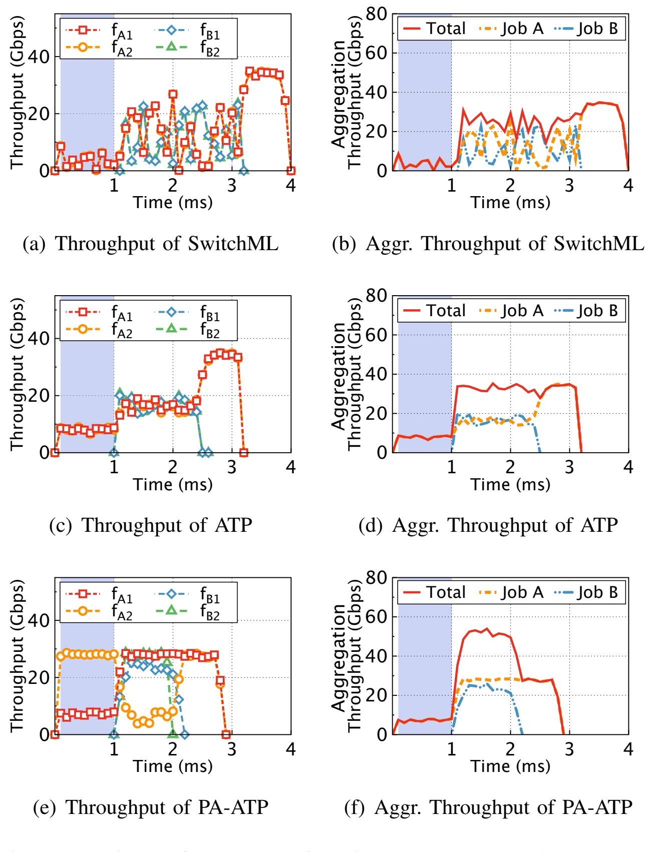
Topology. We use a dumbbell topology, which consists of 11 GPU servers and 2 Barefoot Wedge 100BF-32X programmable switches. One switch S_1 connects to 8 servers and the other one S_2 connects to 3 servers. There are 4 parallel paths (p_1~p_4) between two switches. The INA functions are deployed on S_2. Each link has 40Gbps bandwidth. Besides, each server is equipped with a Mellanox ConnectX5 NIC, an Intel Xeon W-2255 CPU (3.7GHz, and 19.75MB Cache), 64GB DDR4 DRAM and NVIDIA RTX3090 GPU.

Workloads. We choose ResNet50 [14] (98MB model size), AlexNet [15] (233MB model size), VGG11 (491MB model size) and VGG19 (548MB model size) [16] as the representative models on the dataset CIFAR10 [17]. We use SGD as the training optimizer and set the initial learning rate to 0.01.

Straggler injection. To simulate the bandwidth contention scenario in real-world datacenter and evaluate the impact of straggler on PA-ATP, we use 3 severs to generate background traffic transmitted through two switches. The flow size distribution of background flows obeys the realistic Hadoop workloads [18]. The arrival rates of background flows follow a Poisson process.

Baselines. We compare PA-ATP with BytePS [19], SwitchML [4], ATP [5] and StrawAsyn. BytePS transfers gradients from multiple workers to PS without INA, that is, all gradients are aggregated at PS. SwitchML, ATP and StrawAsyn are INA protocols that aggregate the gradients at switch. SwitchML and ATP use the synchronized congestion control. StrawAsyn adopts the asynchronous congestion control, which is unaware of transmission progress and unrestraint of asynchronous degree. Both weight factors f and w of PA-ATP are empirically set to 0.125. For a fair comparison, the gradient payload in all INA protocols is set to 248Bytes [5].

Metrics. (1) Aggregation throughput is the size of gradients aggregated by the switch per unit time. (2) Training throughput is the number of images processed by each worker per second. (3) Time to Accuracy (TTA) is the convergence time to 90% top-1 accuracy. (4) Average Job Completion Time (AJCT) is the average iteration time of model training in 100 iterations.



(a) Throughput of SwitchML (b) Aggr. Throughput of SwitchML

(c) Throughput of ATP (d) Aggr. Throughput of ATP

(e) Throughput of PA-ATP (f) Aggr. Throughput of PA-ATP

Fig. 8: Basic performance of SwitchML, ATP and PA-ATP.

B. Basic Performance

We compare the basic performance of PA-ATP with SwitchML and ATP by launching 2 benchmark jobs (always have data to transmit with no off phase [5]) A and B, which need to aggregate 7MB and 3MB gradients, respectively. Each job has two workers and one PS, and these 4 workers respectively generate flows f_{A1}, f_{A2}, f_{B1} and f_{B2}. The workers send gradients through three paths between S_1 and S_2 to PS. f_{A1} and one background UDP flow share path p_1.

f_{A2} and f_{B1} are transferred on path p_2. On path p_3, only f_{B2} is transferred. The background UDP flow is transmitted at 32Gbps and lasts from 0ms to 1ms. A and B start at 0ms and 1ms, respectively.

We measure the throughput of each flow and the total aggregation throughput at switch S_2. As shown in Fig. 8(a) and 8(c), due to the strict enforcement of synchronized congestion control, SwitchML and ATP are unable to fully use the spare link bandwidth, resulting in long job completion time. For example, when f_{A1} is suppressed by the background flow during 0$\sim$1ms, the other flow f_{A2} in Job A cannot make full use of p_2. Besides, due to the lack of congestion control mechanism, SwitchML suffers from serious packet loss. However, as shown in Fig. 8(e), under PA-ATP, f_{A2} makes the best use of bandwidth on p_2 during 0$\sim$1ms. Before f_{B1} starts transmission, f_{A2} has already buffered a portion of gradients at the aggregating switch S_2. Therefore, when job B starts at 1ms, f_{A2} releases some bandwidth for f_{B1} to reduce JCT of job B. Meanwhile, as shown in Fig. 8(b) and Fig. 8(d), the aggregation throughput is always limited by the bottleneck path p_2 under SwitchML and ATP. On the contrary, as shown in Fig. 8(f), though the leading flow f_{A2} is blocked by f_{B1} after 1ms, the gradients sent by f_{A1} are aggregated with the buffered gradients at S_2, leading to high aggregation throughput of job A.

C. Single Job

We compare PA-ATP against the state-of-the-art schemes on single-job training, which consists of 3 workers and one PS. The background traffic of Hadoop workload is injected into the paths between S_1 and S_2 following a Poisson arrivals.

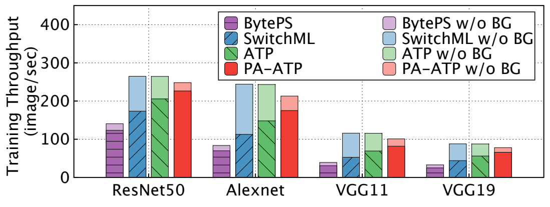


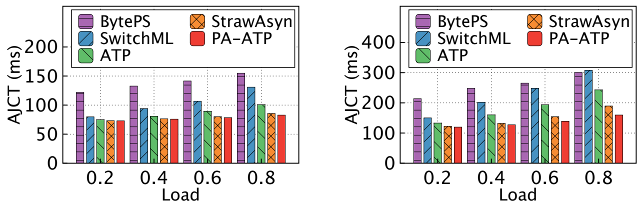
Fig. 9: Training throughput of single job.

We test the training throughput of the single job with and without background traffic under ResNet50, AlexNet, VGG11, and VGG19. As shown in Fig. 9, BytePS with in-network aggregation disabled suffers from serious congestion bottleneck at the last hop to PS, causing the lowest training throughput. SwitchML and ATP utilize in-network aggregation at switch to reduce the volume of exchanged data from multiple workers, thus eliminating the congestion bottleneck at the last hop. However, under bandwidth contention caused by the background traffic, SwitchML and ATP are prone to induce the bandwidth under-utilization and low training throughput due to using the synchronized congestion control. PA-ATP achieves the best performance by using the asynchronous congestion control mechanism.

However, the results in scenario without background traffic show that PA-ATP's performance is slightly lower than SwitchML and ATP. This is because the asynchronous GACKs generated by switches consume the link bandwidth. Due to hardware constraints, the packet size for INA solutions is only $\sim$300Bytes. Consequently, a 62Byte GACK results in $\sim$17% communication overhead. Fortunately, several solutions expand packet size to $\sim$1KB by recirculating packets across multiple ports [4] or employing more flexible programmable hardware such as FPGA [6]. The extra communication overhead will be reduced to $\sim$6%. We leave this work to future.

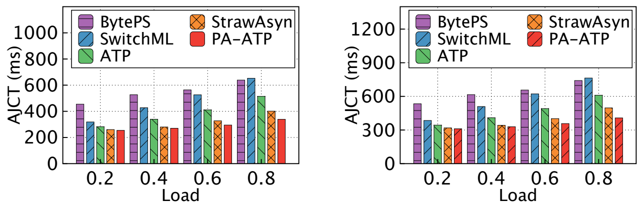
D. Multiple Jobs

We compare the AJCT and TTA of BytePS, SwitchML, ATP, StrawAsyn and PA-ATP under multiple jobs. We train the jobs on all models and vary the load of background traffic from 0.2 to 0.8. We launch two identical training jobs, each of which is deployed on 3 workers and 1 PS.



(a) ResNet50

(b) AlexNet



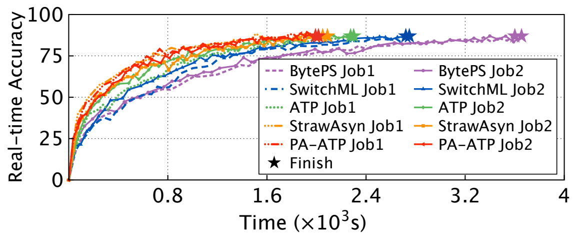
(c) VGG11

(d) VGG19

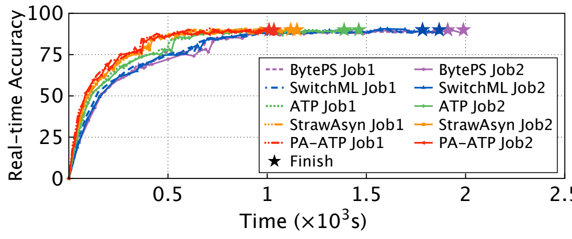
Fig. 10: Average JCT under varying load.

Fig. 10 shows that the AJCT increases with the higher strength of background traffic. Fortunately, with the progress-aware and asynchronization-limited congestion control, the leading flow relinquishes the bandwidth to the straggling ones and the aggregator overflows are controlled under PA-ATP, thus improving the aggregation efficiency and reducing the AJCT. Moreover, with the increasing strength of background traffic, PA-ATP obtains more improvement compared with other the state-of-the-art schemes. Specifically, under 0.8 load in VGG19 model training, compared with BytePS, SwitchML, ATP and StrawAsyn, PA-ATP reduces the AJCT by up to $\sim$45%, $\sim$47%, $\sim$33% and $\sim$18%, respectively.

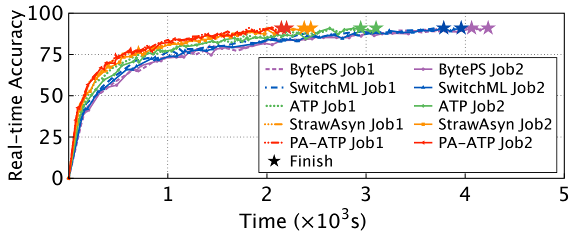
Fig. 11 shows the TTA of two training jobs when the load of background traffic is 0.6. PA-ATP achieves the fastest convergence rate in all model training. Moreover, the convergence rate of the two jobs is similar, indicating that the our design does not break the fairness between multiple jobs.



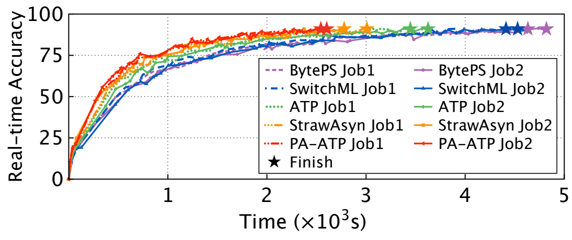
(a) ResNet50



(b) AlexNet



(c) VGG11



(d) VGG19

Fig. 11: Time to accuracy of multiple jobs.

VI. Simulation Evaluation

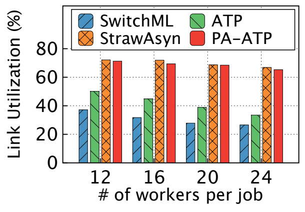
A. NS3 Simulation Setup

Network topology. We use a common 8x8 leaf-spine topology including 8 core switches, 8 top-of-rack (ToR) switches and 128 hosts, and each ToR switch connects to 16 hosts. The link bandwidth and propagation delay are 100Gbps and $2\mu s$, respectively. The switch buffer size is set to 256KB per port.

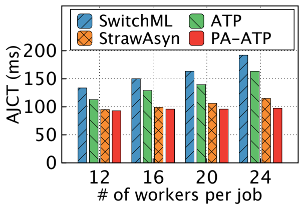
Workload. To emulate the traffic pattern in the real datacenter, the distribution of flow size follows the realistic datacenter workloads Hadoop and Cache Follower [18], where more than 80% bytes are contributed by less than 20% large flows. The flows randomly select the host pairs in the all-to-all transmissions, and their arrival rate obeys a Poisson process. Unless specified, each experiment generates 8 training jobs, which consist of ResNet50 jobs and VGG19 jobs, coexisting with background traffic of Hadoop. The ratio of the number of ResNet50 jobs to VGG19 jobs is 1:1. Each job is deployed on 12 randomly selected hosts.

B. Varying Number of Workers and Jobs

First, we evaluate the performance of PA-ATP with varying number of workers from 12 to 24 per job. As the span of job becomes wider, the jobs have a higher probability to experience straggler caused by bandwidth contention. The results are shown in Fig. 12(a), due to adopting synchronous congestion control, SwitchML and ATP suffer from ACK-clocking and synchronous window reduction, leading to low link utilization, and rapid increase of AJCT. StrawAsyn and PA-ATP cope with congestion better and keep the high link utilization under varying number of workers. PA-ATP achieves the shortest AJCT under this scenario. As shown in Fig. 12(b), PA-ATP reduces AJCT by up to ∼39%, ∼19% and ∼14% compared with SwitchML, ATP and StrawAsyn, respectively.



(a) Link Utilization



(b) Average JCT

Fig. 12: Varying number of workers.

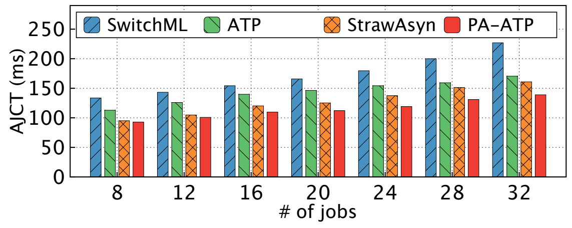


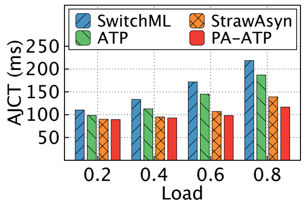
Fig. 13: Varying number of jobs.

We further evaluate the performance of the PA-ATP with varying number of jobs. Each job has 12 workers and is deployed on randomly selected hosts. Fig. 13 shows that SwitchML and ATP experience the large AJCT regardless of the number of jobs. Different from SwitchML and ATP, StrawAsyn has the low JCT when the number of jobs is small. However, when the number of jobs increases, the AJCT of StrawAsyn increases rapidly, and even becomes close to ATP under 32 jobs due to the intense bandwidth contention between jobs. PA-ATP accelerates the straggling flow with a progress-aware manner and control the asynchronous degree for all jobs, so it achieves the lowest AJCT.

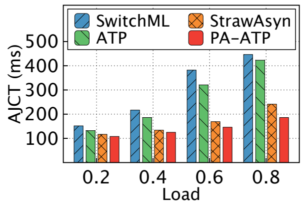
C. Varying Workloads

We use different workloads (Hadoop and Cache Follower) to evaluate PA-ATP and change the load in the range of 0.2 to 0.8. The evaluation results are shown in Fig. 14. As the network load becomes heavier, the job will experience more frequent straggling flows, and SwitchML and ATP rapidly increase AJCT due to ACK-clocking and synchronous window reduction. SwitchML has the longest AJCT due to its lack of congestion control. The network congestion leads to packet loss and timeout retransmission when network load increases.

StrawAsyn and PA-ATP still perform well under heavy load. For example, PA-ATP reduces AJCT by up to ∼62%, ∼60% and ∼22% compared with SwitchML, ATP and StrawAsyn under Cache Follower workload, respectively.



(a) Hadoop



(b) Cache Follower

Fig. 14: Varying workloads.

D. Ablation Experiment

In this part, we conduct the ablation experiment to evaluate the effectiveness of various modules in PA-ATP, including progress-aware congestion control and asynchronization-limited control. We compare the performance of PA-ATP against ATP, StrawAsyn, StrawAsyn with progress-aware congestion control, and StrawAsyn with asynchronization-limited control. We generate 8 Resnet50 jobs and each job has 12 randomly selected workers. Meanwhile, we generate cross traffic under 0.5 load, and the distribution of flow size are followed Hadoop workload. We measure the average JCT under varying switch memory from 2MB to 10MB.

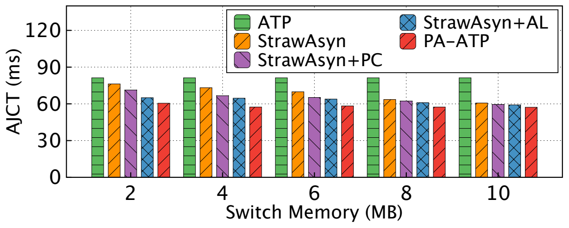


Fig. 15: Average JCT under varying switch memory. StrawAsyn+PC and StrawAsyn+AL represent StrawAsyn with progress-aware congestion control and StrawAsyn with asynchronization-limited control, respectively.

Fig. 15 shows the results. The four asynchronous manners perform better than ATP under varying switch memory size. However, due to unlimited asynchronous degree leading to severe aggregator overflows, the performance of StrawAsyn and StrawAsyn+PC degrade rapidly as the switch memory decreases. StrawAsyn+PC and PA-ATP perform better than StrawAsyn and StrawAsyn+AL, respectively. The reason is that progress-aware manner improves the aggregation throughput of switch and reduces the occupation time of each gradient in aggregator. Through combining progress-aware manner with asynchronization-limited manner, PA-ATP achieves the shortest average JCT under varying switch memories.

VII. Related Work

Recently, a series of related works have been proposed to speed up DML training. We divide these works into two categories: switch-based and host-based solutions.

Switch-based solutions. DAIET [20] first presents a concept design of INA, which utilizes the switches on whole path to reduce the traffic volume. Sharp [21] and Sharpv2 [22] design and implement the INA protocol in Infiniband network. SwitchML [4], PANAMA [6] and ATP [5] expand INA to Ethernet network. The recent INA solutions are ESA [2] and Trio [23], which alleviate straggler issues by modifying aggregation rules. For example, ESA forwards the partial aggregation results from the aggregators to the PS and completes the final aggregation at the PS when straggling gradients arrive. However, ESA still adopts synchronous congestion control, leading to the link under-utilization. Trio gives up the gradients from straggling workers, and the partial aggregation results are broadcast to all workers, introducing the noisy gradients.

Host-based solutions. P3 [24] overlaps the computation and communication phase to improve the utilization of GPU. ByteScheduler [25] partitions the tensors into the fine-granularity pieces, further optimizing the overlap of communication and computation. By coordinating All-Reduce and PS communication patterns, BytePS [19] achieves higher link utilization. BlueConnect [26] and PLink [10] propose a topology-aware manner to optimize the communication patterns. Recently, considering the sparsity of gradients, OminiReduce [27] reduces the volume of aggregation traffic by only transmitting non-zero gradient.

Compared with the above solutions, PA-ATP focuses on the congestion control mechanism and straggler problem in existing INA services. Through progress-aware congestion control and asynchronous degree control, PA-ATP improves the efficiency of INA transmission with multiple jobs in the large-scale datacenters.

VIII. Conclusion

In this paper, we propose a progress-aware INA transmission protocol PA-ATP, which combines asynchronous congestion control, progress-aware and asynchronization-limited manners. PA-ATP adjusts the sending rate according to the congestion level and transmission progress, and reduces the aggregation time by accelerating straggler flow. PA-ATP is implemented by using P4-programmable switch and kernel-bypass protocol stack at the end host. We evaluate PA-ATP in the testbed and NS3 simulations. The results show that PA-ATP speeds up the training time by up to 62% than the existing INA transmission protocols.

Acknowledgments

We would like to thank our shepherd Wenfei Wu and the anonymous reviewers for their insightful feedback. This work was supported by the National Natural Science Foundation of China (62132022, 61872403, 61972421, 62302524), Key Research and Development Program of Hunan (2022WK2005), and Natural Science Foundation of Hunan Province, China (2021JJ30867). This work was also carried out in part using computing resources at the High Performance Computing Center of Central South University. Jiawei Huang is the corresponding author.

REFERENCES

[1] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[2] Hao Wang, Yuxuan Qin, ChonLam Lao, Yanfang Le, Wenfei Wu, and Kai Chen. Efficient data-plane memory scheduling for in-network aggregation. *arXiv preprint arXiv:2201.06398*, 2022.

[3] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[4] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *Proc. USENIX NSDI*, pages 785–808, 2021.

[5] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. Atp: In-network aggregation for multi-tenant learning. In *Proc. USENIX NSDI*, pages 741–761, 2021.

[6] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network aggregation for shared machine learning clusters. In *Proc. MLSys*, pages 829–844, 2021.

[7] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems*, 27:19–27, 2014.

[8] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[9] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *Proc. USENIX ATC*, pages 947–960, 2019.

[10] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. Plink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training. In *Proc. MLSys*, pages 82–97, 2020.

[11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Proceedings of NeurIPS*, 24, 2011.

[12] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Proceedings of NeurIPS*, 26, 2013.

[13] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proc. ACM SIGCOMM*, pages 63–74, 2010.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, pages 770–778, 2016.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84–90, 2012.

[16] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[17] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[18] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM*, pages 123–137, 2015.

[19] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *Proc. USENIX OSDI*, pages 463–479, 2020.

[20] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proc. ACM HotNets*, pages 150–156, 2017.

[21] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (sharp): a hardware architecture for efficient data reduction. In *Proc. IEEE COMHPC*, pages 1–10, 2016.

[22] Richard L Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. Scalable hierarchical aggregation and reduction protocol (sharp) tm streaming-aggregation hardware design and evaluation. In *Proc. HiPC*, pages 41–59, 2020.

[23] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of ACM SIGCOMM*, pages 633–648, 2022.

[24] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. In *Proc. MLSys*, pages 132–145, 2019.

[25] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proc. ACM SOSP*, pages 16–29, 2019.

[26] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In *Proc. MLSys*, pages 241–251, 2019.

[27] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proc. ACM SIGCOMM*, pages 676–691, 2021.