ATP: In-network Aggregation Protocol for Multi-tenant Learning

Paper # 273, 12 pages

Abstract

Distributed deep learning systems are widely deployed in clusters wherein the network is shared across multiple tenants. We propose a communication protocol that supports in-network aggregation for distributed deep learning training, called ATP. ATP uses the emerging programmable switch hardware to support in-network aggregation and co-designs the switch processing logic and end host networking stack. ATP handles the multi-tenant case and supports the inter-rack scale in-network aggregation by addressing congestion control, failure recovery, and switch state orchestration. ATP is backward compatible with non-programmable switches and allows for incremental deployment of programmable switches. ATP outperforms existing systems with speedup in application throughput of up to 869%, with higher gains with network-intensive models.

1 Introduction

Deep Neural networks (DNN) are being successfully used in a wide range of artificial intelligence applications such as image recognition [29, 30], natural language processing [8], etc. As these DNN models become deeper [8] and the datasets become larger, the training times have increased rapidly, from several hours to weeks to even months. To cope with the increasing training time, distributed DNN training (DT) is becoming the norm. These DT systems are being widely deployed in industry and academia [12, 22, 23, 32] to accelerate DNN training via data parallelism. Parallelization techniques such as mini-batch stochastic gradient descent (SGD) training have significantly sped the computation of gradients. However, recent studies [33] show that these workloads are becoming increasingly network-bound. Network communication comprising of gradient and parameter exchange over the network, contributes to a significant portion of the training time.

As hardware accelerators (such as TPUs, GPUs [2, 15, 32]) are expensive, DT systems are increasingly deployed in a datacenter-scale cluster shared by multiple tenants for cost-savings via multiplexed usage of resources. In such settings, multiple applications involved in the end-to-end machine learning pipeline co-exist across hundreds of cluster nodes spanning multiple racks connected with a multi-tiered network fabric. These include storage systems for storing training data and models; systems for data cleaning, transformation and featurization; distributed DNN training systems; and systems for inference with the ensemble of models.

With contention from multiple competing tenants, DT job workers are increasingly susceptible to inter-rack placement. Furthermore, these jobs compete for network resources — network bandwidth, switch buffers, and end-host CPU cycles for packet operations — with other tenants. This increases the communication latencies of DT jobs. With extraneous traffic from competing tenants, the exchange of gradients and parameters incurs higher latencies as network bandwidth is a scarce resource. The training time is highly impacted by the resulting high tail latencies [26, 31] as the exchange of gradients and parameters across the network is bound by the slowest network transfer (typically the cross-rack transfers) for the case of synchronous SGD [6, 17]. This exacerbates the network-bound nature of these jobs in a multi-rack, multi-tenant setting.

Recent works [4, 10, 32] outline techniques for reducing network latencies by using high-performance networking stacks such as RDMA; by using alternative network communication patterns such as ring all-reduce to avoid incast-related network bottlenecks; by reordering the transmission of different DNN layers to hide network latency via overlap of communication with computation. However, in all these works, the amount of data transmitted over the network remains more-or-less the same. The key to reducing network latencies in multi-tenant, multi-rack scenario, where network bandwidth is a scarce resource, is to reduce data transferred over the network.

With the emergence of programmable switches, offloading aggregation of gradients to the network offers promise. Aggregation of gradients at the earliest opportunity in the network reduces the amount of data transmitted over the network. This increases the throughput of gradient transfers and reduces latencies. Typically, schedulers [3] try to pack DT jobs densely in as few racks as possible. This presents a great opportunity to aggregate gradients from several workers at its first point of contact with the network i.e. the edge switch and greatly reduces the amount of data transferred over the network. Additionally, in-network aggregation not only reduces the amount of data transmitted over the network, but also reduces the CPU usage due to packet I/O and aggregation computation at the end-hosts. In addition, given that the

packetization, depacketization and aggregation are done in ASIC [19, 20, 25], this is faster than doing the same operations on general-purpose CPU cores. This reduces network contention and frees CPU cycles at end-hosts which is especially appealing in a multi-tenant scenario.

However, there are issues to realize network offload of aggregation at a multi-rack, multi-tenant scenario.

First, the usage of programmable switch resources especially switch-memory for aggregation needs to be isolated and limited so as not to disrupt other essential network services. Also, the switch-memory isolated for aggregation purposes needs to be carefully allocated across multiple DT jobs. A simplistic long-term static allocation mechanism is inefficient as DT jobs exhibit an on-off communication pattern and switch-memory remains unused during the off phase. Also, complete offload of aggregation to the network might end up halting DT jobs for long durations until a successful allocation.

Second, reliability and congestion control algorithms at the end-host networking stack cannot be used as-is. In-network aggregation breaks the end-to-end semantics, and aggregation-induced packet "drops" misinterpreted as packet losses can cause unnecessary retransmissions.

In this work, we present an in-network aggregation protocol, ATP, for the multi-rack, multi-tenant scale. Our key contribution is the *best-effort in-network aggregation* primitive. ATP protocol is an enabler for a judicious division of gradient aggregation across the network and the end-host. ATP proposes reliability and congestion control algorithms via careful co-design of the end-host networking stack and programmable dataplane logic with the core design philosophy of minimizing protocol complexity in the network and the goal of maximizing utilization of switch resources. This means that, gradients are opportunistically aggregated in the network whenever switch-memory is available for no additional cost.

We run extensive experiments on popular models, e.g., VGG16 [28], ResNet50 [11] to evaluate ATP in a multi-rack scale testbed with multiple jobs. The evaluation results show that ATP outperforms existing systems by 8.7X, and it is even slightly better than the currently-best Horovod with RDMA in our testbed. More importantly, ATP consumes less link capacity, supports dynamic job deployment, and enables efficient switch resource usage, all of which are good properties for deployment in multi-tenant networks.

2 Background

2.1 Distributed Training

In the distributed training, data is partitioned and distributed to the workers for data parallelisms. Each worker locally computes a gradient, the gradients from all workers are gathered in one host (a parameter server in PS approach) over the network, and the host performs the aggregation on the gradients and finally multicasts the updated parameters to the workers. The above process is repeated until the model converges.

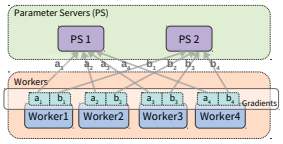


Figure 1: Parameter Servers (PS)

Parameter server (PS). In this approach, there are processes at end-hosts with two roles - parameter server (PS) and workers. Each worker generates the gradients, on distinct partitions of the dataset and sends parameters to PS. With multiple PS, each PS has a distinct partition of parameters. As shown in Figure 1, the gradients are partitioned into 2 subsets as there are two parameter servers. Then, each worker sends the corresponding subsets to the parameter server and the parameter server sends back the aggregation results. With this approach, each worker sends and receives $2|M|$ bytes, where $|M|$ is the total number of bytes to be aggregated from each worker. In aggregate, the amount of data transmitted is $2|M||W|$ where $|W|$ is total number of workers. This volume reduces with in-network aggregation and in the best case with all workers on one rack is $2|M|$.

2.2 Programmable Switch

The recent emergence of the programmable switch provides opportunities to offload applications [16, 21, 34] as it exposes the stateful and stateless objects in dataplane. Stateless object, *metadata* , holds the state for each packet, and the memory is released when a packet gets dropped or forwarded. Stateful object, *register*, holds the state forever as long as the switch program is running. The *register* value can be read and written in the dataplane, but can only be accessed once, either for read or write or both, for each packet. The register memory can only be allocated when the switch program launches. To add memory allocation, users have to stop the switch service and modify the switch program and restart the switch program. A register variable is presented as an array of 32-bit integers. In the context of in-network aggregation, we call an item of a register as *aggregator*.

3 Motivation

Existing networks define as a principle a clear boundary between the network and the end-hosts when it comes to placement of application functions. Any application function is strictly placed at the end-hosts. However, the recent introduction of programmable switches exposes programmability of dataplane logic and dataplane state

to applications. This provides opportunities for renegotiating the boundaries for application function placement across the end-hosts and the network.

In this work, the application we focus on is distributed deep neural network training (DT). In DT jobs, the computation phase has seen drastic improvements in performance over the last few years owing to advances in GPUs and hardware accelerators. Network performance now is a substantial contributor to overall training times.

A key contributor to network performance is the *aggregation* of gradients from all workers in the synchronization phase. Recent work, SwitchML [26], proposes offloading aggregation in DT jobs entirely to the network with the insight that in-network aggregation is a means to provide network acceleration to DT jobs. Aggregation reduces the amount of data transmitted during synchronization phases, which increases throughput and diminishes latency, and speeds-up training time. SwitchML demonstrates this idea at a single rack-scale and for a single DT job scenario with upto 300% speedup in training.

While an important first step, SwitchML cannot be adopted in most clusters where DT jobs run today. Typically, clusters span multiple racks and is used by multiple tenants (comprising of multiple DT and non-DT jobs). The goal of our work is to provide network acceleration to DT jobs for the multi-rack, multi-job scenario.

Consider a straw man that scales SwitchML at inter-rack scale for a multi-job scenario wherein – (a) all worker gradients for each DT job are aggregated at the edge switch connected to the PS; and (b) in keeping with SwitchML's design choices, switch memory is statically allocated for the lifetime of the DT job and aggregation is offloaded entirely to the network. This has several issues which we highlight next and in doing so motivate the set of requirements for ATP.

Need for dynamic, short-lived allocation: Static allocation of switch memory for the lifetime of a DT job can lead to inefficient switch memory utilization. DT jobs typically exhibit an on and off communication pattern corresponding to synchronization and computation phases respectively. With lifelong static allocation, switch memory remains unused during the off phase when workers compute gradients which, in a multi-job scenario, can potentially be reallocated to other DT jobs that are in their synchronization phase. This is inefficient. Thus, an in-network aggregation scheme that does *dynamic, short-lived allocations* to multiplex limited switch memory is required to avoid this inefficiency.

Need for end-host based aggregation fallback: Offloading aggregation entirely to the network can lead to job slowdown. A DT job remains blocked until it receives explicit in-network switch memory allocation(s). However, when there is contention from competing DT jobs, a job may experience long waiting time for switch memory allocation. This leads to unbounded job slowdown as contention increases. Thus, an in-network aggregation scheme that can *fallback to end-host based aggregation* is required to avoid unbounded job slowdown.

Need for redesign of reliability and congestion control: Any aggregation offload to the network has implications on reliability and congestion control. Aggregation in the network breaks end-to-end semantics as gradient packets are consumed and "dropped" by design in the network. A packet loss can no longer be considered as an indicator of the congestion in the network. This means networking protocols at the end-host networking stack such as TCP that offers reliability and congestion control that depend on such indicators cannot be used as-is.

Reliability needs to distinguish packet losses due to queue drops or corruption from aggregation-induced packet "consumption". This requires new packet loss identification and recovery algorithms at the end-host networking stack. Furthermore, dataplane logic for in-network aggregation needs careful co-design with the reliability mechanism. Packet retransmissions should not lead to aggregation incorrectness i.e., repeat aggregation of gradient packets. Also, switch memory should not be indefinitely occupied in the face of permanent loss of gradient packets such as in the case of end-host failures.

Congestion control with in-network aggregation not only needs to maximize the utilization of network bandwidth but also maximize the utilization of switch memory for aggregation. This requires new algorithms to detect and react to congestion at the end-host networking stack.

Thus, *new algorithms for reliability and congestion control are needed* and their implementation needs *careful co-design of dataplane logic with the end-host networking stack*.

4 Design Challenges and Overview

ATP is a network protocol that provides in-network acceleration for DNN training (DT) jobs at inter-rack scale for multi-tenant scenarios.

ATP proposes the *best-effort in-network aggregation* primitive. ATP splits gradient updates (obtained from computation phase at each worker) across a sequence of gradient packets. Each gradient packet has a unique sequence number, and the task of aggregating gradients from all workers is broken into smaller tasks of aggregating gradient packets with the same sequence number from all workers. These packets are streamed from the workers using the ATP protocol and aggregated opportunistically in the network as and when switch memory is available en-route the parameter servers(s) (PS). The switch memory can be imagined as an array of aggregators – each aggregator can aggregate gradient packets for

a particular sequence number for a DT job. In the case of aggregator unavailability, aggregation eventually takes place in the parameter server(s) (PS) at the end-hosts. The PS receives these aggregated gradient packets — either completely aggregated gradients or several individual or partially aggregated gradients for each sequence number. If needed, the PS does a final aggregation of gradients and updates the subset of parameters corresponding to this sequence number in the aggregated gradients. The PS then sends the updated parameters as a packet with the same sequence number back to all the workers.

There are several design challenges in achieving this. We outline these challenges and give an overview of our design choices next. The key design principle we adhere to in making these choices is to minimize protocol complexity in the network whenever possible.

4.1 Towards aggregation at the edge

Challenge: Workers and PS for a job can span multiple racks. Aggregating gradient packet streams from workers at the earliest switch where these streams meet in the network is the optimal strategy to reduce the amount of data transmitted in the network. The flow of these (partially) aggregated streams can be imagined as an aggregation hierarchy. In this aggregation hierarchy, the workers are the leaf nodes, the programmable switches are the intermediate nodes, and the PS is the root node. In this aggregation hierarchy, only on completion of aggregation of gradient updates from all the child nodes is the aggregated gradient packet stream pushed upstream to the parent node. The flow of streams in the network is non-deterministic owing to the probabilistic nature of ECMP-based routing which is common to muti-tiered network topologies. This means the intermediate nodes can keep changing and this makes the aggregation hierarchy dynamic. A dynamic aggregation hierarchy is undesirable as it requires careful encoding of control state at each node. This control state encodes child node connectivity of this node so as to keep track of which child node packet streams are yet to be aggregated so as to push the aggregated packet stream to the parent node only on completion. This makes the in-network protocol complex as this control state needs to be dynamically updated whenever there is a change in the aggregation hierarchy.

Solution: To overcome the above challenge, the key insight we have is that restricting the aggregation hierarchy to only contain edge switches can make it static. The edge switches connected to the workers and the PS are invariant. The child-node connectivity and in-degree in this pruned aggregation hierarchy with only the edge-switches is invariant to the probabilistic in-network routes that packet streams take. This requires only a one-time setup at the beginning of the DT job. While this is not optimal in terms of minimizing network traffic, it is efficient in practice — DT jobs are typically packed in as few racks as possible [3] which means that worker gradient packet streams are aggregated at its first point of contact with the network i.e.,the edge switch immediately connected to it.

4.2 Towards dynamic, short-lived aggregator allocations with end-host fallback

Challenge: Programmable switches have limited memory which means a fixed size array of aggregators is available for in-network aggregation. This limited aggregator array needs to be carefully managed across contending DT jobs. A naive straw man is to do the management of an aggregator pool in a centralized manner. A centralized mechanism would explicitly allocate a subset of aggregators to each DT job. DT jobs exhibit on-off network usage pattern and for efficient aggregator array usage, an ideal centralized mechanism would have to do frequent aggregator re-allocations. This is susceptible to being a scalability bottleneck with an increasing number of DT jobs. Also, this can lead to state inconsistency issues during the transition of ownership of the aggregators from one DT job to the other. The packet streams from the previous DT job that were inflight during the transition may falsely claim ownership of and aggregate gradient to the aggregators reserved for the new DT job. This leads to aggregation incorrectness. Avoiding this requires careful co-ordination which can be expensive as it may require pause and resume of DT jobs during aggregator re-allocation and may also require more complex validity checks on packet headers before aggregation at the switch.

Solution: ATP adopts a distributed mechanism for aggregator allocations, where all aggregator array management is end-host driven. Any aggregator array operations such as allocation, value modification, and deallocation are triggered by the appropriate packets sent from the end-hosts. This avoids in-network protocol complexity. In simple terms, on arrival of a gradient packet a hash is calculated on the packet header and the aggregator at that hashed index in the aggregator array is allocated to that job if available. In case of unavailability, the gradient aggregation is done at the end-host. In the case when the gradients packets from some workers are aggregated at switch, and while the gradient packets from the other workers are aggregated at the end-host, ATP treats this scenario as packet loss.

4.3 Towards reliability

Challenge: In-network aggregation breaks end-to-end semantics. Packet loss due to queue drops or corruption needs to be distinguished from aggregation-induced in-network packet “drops” or “consumption”.

A straw-man proposal for reliability is to assign ACK semantics to the parameter packet stream from the PS to the workers. The worker then does timeout-based gradient packet retransmission (similar to SwitchML). If the ACK i.e., the parameter packet for a certain sequence number, is not received at the worker for a certain time duration, the worker resends the gradient packet for that sequence number. This can lead to aggregation incorrectness for the case when the original gradient packet was already aggregated in an aggregator and the parameter packet to a worker from the PS was lost. In this case, the duplicate gradient packet from the worker will be re-aggregated, leading to application incorrectness. This is avoided by maintaining a bitmap tracking which gradients have already been aggregated. The aggregator is deallocated when all the gradient packets from all the workers are received.

However, this straw-man can cause switch memory leaks. A switch memory leak is a wasted aggregator which cannot be used by any DT job. In case of an end-host failure, an aggregator might wait indefinitely for a gradient packet from that end-host which lays that aggregator to waste.

Solution: ATP retains the reliability mechanism proposed by the straw-man except that ATP takes the design decision of aggregator deallocation on seeing any retransmitted packet. The partially aggregated gradient is forwarded upstream and any yet-to-be aggregated gradients are eventually aggregated at the end-host. This reliability mechanism avoids switch memory leaks as an end-host failure will cause timeout at one of the live workers and seeing retransmissions will deallocate aggregators reserved for aggregation of these gradient packets.

4.4 Towards congestion control

Challenge: With in-network aggregation, jobs are not only contending for network bandwidth, receiver CPU and switch buffers but also switch memory.

TCP-based congestion control uses RTT at the sender, duplicate ACKs, or ECN marking as signal of network congestion and adjusts send window in response to these signals. However, TCP-like congestion control mechanism cannot be used as-is as end-to-end "round-trip" semantics are broken due to aggregation induced packet "drops" and many-to-one semantics of aggregation.

Solution: ATP maintains a window at each worker. Each worker sends no more than a window's worth of in-flight gradient packets. The window is updated on receiving the ACK. Recall, the parameter packets from the PS serve as ACKs. For rate control, ATP applies an Additive-Increase-Multiplicative-Decrease (AIMD) scheme to scale the window at each worker with the objective of maximizing network resource usage and achieving approximate fair-share of resources to all DT jobs. The scaling of a window is a means to increase or decrease the memory pressure on switches as the number of aggregator requests is in direct proportion to the window i.e., the maximum number of inflight gradient packets. In case of high memory pressure, many gradient packets will not be aggregated in-network causing an increase in traffic volume, which can trigger queue length buildup in switches or packet loss in switch buffers or receiver NIC overload. Indirectly, traditional congestion signals can also be symptomatic of high memory pressure. Thus, ATP also relies on ECN markings and packet losses as congestion signals.

5 Design

To realize the ATP protocol, we change the infrastructure – the programmable switches and the end-host networking stack – to realize routing, aggregation, reliability, and congestion control in sending and receiving gradient and parameter packets for DNN training (DT).

The overarching theme to our design thinking is to reduce in-network protocol complexity. To do so, we set the constraint of one-time static setup of the in-network infrastructure i.e. the programmable switches. This setup involves isolating switch-compute and switch-memory for ATP and programming the switches with ATP logic so as to process ATP packets and maintain the appropriate in-network state. The key to avoiding in-network complexity is two-fold. First, we keep the switch logic purely *event-triggered*, where the event is the arrival of an ATP packet at the switch. Second, we keep switch logic *stateless*. This means that any dynamic control state, such as DT job specific state for job specific packet operations that the switch logic needs to operate on is carried in ATP packets. This avoids the dynamic configuration of in-network infrastructure. Overall, any in-network state operation - allocation, modification or deallocation - is triggered by the arrival of an ATP packet.

This requires careful co-design of the end-host networking stack and the switch logic.

5.1 ATP Infrastructure Setup

Static Infrastructure Setup. The infrastructure, comprising the switches and the end-host networking stack, is configured once to serve all ATP jobs. Each programmable switch installs a classifier to identify ATP traffic (i.e. gradient and parameter packets) and isolates a portion of switch resources (comprising memory and pipeline stages) to aggregate ATP traffic. The end host installs an ATP networking stack, which intercepts all the send or receive network calls from DT jobs. Also, the end-host is made aware of details of the complete network topology (including switch, end-host port connectivity) which is essential for per-job setup.

Dynamic Per-Job Setup. This uses the network topology information to configure the end-host networking stack for each worker and PS of an ATP job upon arrival. When an ATP job is launched, each worker is configured with a unique ID in the range $[1, 2, ..., W]$, where W is the total number of workers. Also, the location of the workers, PS and the network topology is used to identify the aggregation hierarchy. The aggregation hierarchy captures the edge switches (used for aggregation) and their connectivity to the workers, PS and other edge switches. Each gradient packet from the worker has at most two edge switches (which we also refer to as aggregation services) en-route to the PS. Each worker is also configured with the total number of worker packet routes that go through each of these intermediate edge-switches. Note, the fan-in at the edge-switch connected to the PS is fixed ($= W$). This fan-in is a threshold. When the number of gradient packets aggregated at a switch is equal to this threshold, the aggregated result is forwarded upstream. Carrying this control parameter in the packet avoids dynamic reconfiguration of the switches.

The PS is configured with a multicast IP. A multicast group and tree of all the workers is set up for each job when the DT job is launched using the IGMP protocol.

5.2 Datastructures

Packet Format. Both the gradient and parameters are tensors, and each tensor is divided into a sequence of fixed-size elements, such that each element fits in a packet. Each gradient packet has one element and a unique sequence number (equal to the sequence number of the element) and the task of aggregating gradients from all workers is broken into smaller tasks of aggregating gradient packets with the same sequence number from all workers. Figure 2 shows the gradient packet format. The packet has the ordinary Ethernet and IP headers, where the MAC and IP addresses are that of PS and used to route traffic towards the PS. Inside the payload, is ATP protocol specific fields. First is the configuration information of that worker i.e., the worker ID, the fan-in degree at the first edge-switch, and the fan-in degree at the second edge-switch (if there is a second edge-switch). The resend flag is set if it is a retransmitted gradient packet. The edge switch identifier is 0 if the packet is en-route to the first edge-switch in the aggregation hierarchy and is 1 if the packet is en-route to the second edge switch in the aggregation hierarchy. The sequence field has the sequence number of the element (or partially aggregated elements). The data field contains an element (or aggregated elements). The bitmap field tracks which worker gradient packets are aggregated in this packet.

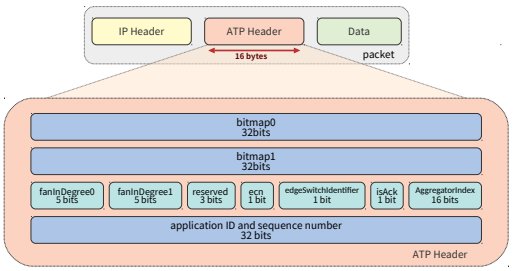


Figure 2: ATP Packet Format

Figure 2 shows the parameter packet format. The packet has ordinary Ethernet and IP headers, where the MAC and IP address correspond to the multicast group consisting of all the workers and is used to route traffic towards the workers. ATP protocol specific data is wrapped in the payload. It has a sequence number and an element data field. The sequence number identifies the sequence number of the element. The parameter packet is used as an acknowledgment (ACK) for the gradient packets sent by the workers with the same sequence number. The data field contains an element from the updated parameters for the next round.

Switch Memory Layout. Figure 3 shows the switch memory layout. Each programmable switch isolates a portion of switch memory for ATP jobs. The memory is organized as an array of fixed-size memory segments, which we refer to as aggregators. Each aggregator is accessed by its index in the array. Each aggregator has the following fields — a value field to accumulate the data i.e. element contained in gradient packets from different workers; an identifier field of <Job ID, Sequence Number> to uniquely identify the job and the element that this aggregator serves for; a bitmap field to record which workers have already been aggregated to the value field. The size of this value field is the same as the size of an element.

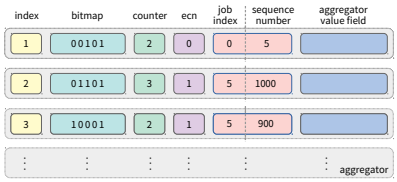


Figure 3: ATP Switch memory layout

5.3 Switch Logic

The gradient and parameter packets traverse the aggregation hierarchy from workers to the PS and the PS to the workers respectively. The arrival of an ATP packet triggers the switch logic. In the ideal case, each edge switch completes aggregation of all incident worker gradient packets and sends the aggregated result upstream. However, non-ideal cases (e.g. packet loss) can occur.

We explain the switch logic in ATP next and also detail handling of non-ideal cases.

Aggregator Allocation Mechanism. The programmable switch recognizes ATP packets using the AppType field. Only ATP packets can reserve aggregators.

When a gradient packet arrives, ATP assigns it to an aggregator, if available. ATP computes a globally consistent hash on <Job ID, Sequence Number> — HASH(<Job ID, Sequence Number>)%numAggregators. If the identifier field in the aggregator is empty (i.e. <Job ID, Sequence Number> is null), the packet claims the aggregator by assigning the tuple <Job ID, Sequence Number> in the gradient packet to the identifier field in the aggregator. The value field in the aggregator is set to the data field in the packet and the bitmap field in the aggregator is set to the bitmap field in the packet. If the identifier field in the aggregator is non-empty, we compare the gradient packet's <Job ID, Sequence Number> to the identifier field in the aggregator (① in Figure 4). If they are not equal, ATP pushes the gradient packet upstream without processing it. To avoid the aggregation on this packet at upstream switch, ATP sets resend flag and flips the edge switch identifier at the packet. However, if they are equal, the gradient aggregation takes place, which we describe next.

Gradient Aggregation. If a gradient packet claims its aggregator, ATP uses the edge switch identifier to fetch the fan-in degree and the bitmap for this edge switch (③ in Figure 4). The fan-in degree is used to track the completion of aggregation at an edge switch. Then, ATP checks whether this packet has been aggregated (④ in Figure 4). If it has already been aggregated and not a retransmitted packet, ATP drops the packet. If not, ATP aggregates the packets gradient value to the value field in the aggregators and also aggregates the bitmap field in the gradient packet to the bitmap field in the aggregator. If the packet is not a retransmitted packet (⑤ in Figure 4), ATP increases the number of set bits in bitmap field [1]. If the number of set bits in bitmap field in the aggregator is not equal to the fan-in degree (⑥ in Figure 4), ATP drops the gradient packet. However, if it is equal, the aggregation is complete and ready to be pushed upstream. The gradient packet element field is replaced by the value field of the aggregator and the gradient packet value field is replaced by the bitmap field of the aggregator. The edge-switch identifier in the gradient packet is flipped and ATP pushes the packet upstream.

Aggregator deallocation using Parameter Packets. PS multicasts parameter packets to the workers as and when aggregation of an element in parameter completes. A parameter packet works as the acknowledgement (ACK) of the gradient packets, thus, when ATP processes this packet at an edge switch, it deallocates the aggregator at that switch — setting all fields to 0 if the packet and the aggregator <Job ID, Sequence Number> are the same (② in Figure 4).

Dealing with Packet Retransmissions. Packet loss may occur due to drop at switch buffers or corruption. This may cause incomplete aggregation. There are two broad packet loss scenarios — (1) gradient packet en-route to the PS is lost, or (2) parameter packet en-route to worker(s) is lost.

In Figure 4, we elaborate the reliability mechanism and when ATP generates a retransmission packet. A retransmitted packet has the same format as the original gradient packet, except that the resend flag is set.

In case of an aggregator hit (i.e., gradient packets corresponding to this job and sequence number have an aggregator allocation) at a switch, ATP uses the retransmitted gradient packets unique worker ID to check if bitmap field in the aggregator is set. If set, ATP forwards the gradient packet upstream as-is (⑧ in Figure 4). If unset, the data field element is aggregated in the value field of the aggregator and the bitmap field in the aggregator at the unique worker ID index is set (④ in Figure 4). ATP copies the value field and the bitmap field from the aggregator to the data field and the bitmap field of the packet, and sends the packet upstream (⑨ in Figure 4). Note that at the second level of aggregation, ATP directly forwards a retransmitted gradient packet for simplicity (⑦ in Figure 4). In all the cases, ATP deallocates the aggregator and sets all fields to 0 (⑩ in Figure 4). ATP does this to switch memory leaks.

5.4 End Host Logic

Worker pushing gradients. In a worker, the entire gradient is split across a sequence of packets. A sliding window is maintained over this sequence. A worker takes several rounds to push the gradient packets to the PS and pull the updated parameters back and each round comprises one window of packets.

The worker intercepts send or receive calls from the DT system to get these gradients. ATP converts float representation of gradients to integer representation so as to allow for aggregation in-network which does not support float operations. We use the same strategy as in prior work [26]. These gradient packets are small and ATP introduces optimizations to achieve high throughput. ATP starts multiple threads to achieve high packet I/O rate. When the ATP library receives gradients to transfer, it assigns it to a thread to send. It can cause load imbalance across different threads. Each worker in ATP has a centralized scheduler to receive the gradients from the application layer and maintains the total load for each

[1] It is tricky to count the number of set bits in bitmap at switch, thus we keep a redundant variable for this.

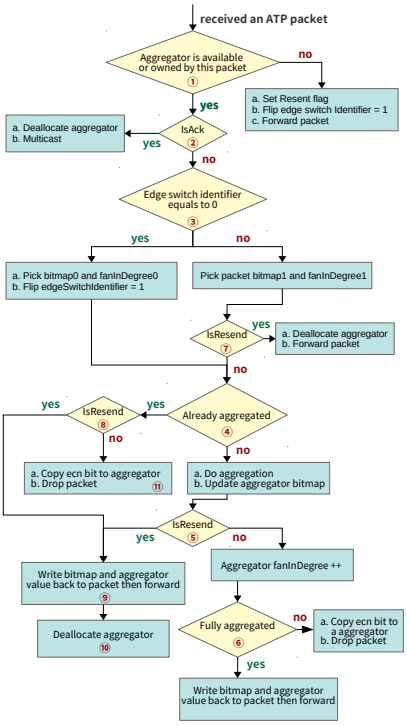


Figure 4: Workflow of the switch logic

thread. Whenever the centralized scheduler receives gradients to be sent, it extracts the size and assigns it to the least loaded thread. This balances the load between threads.

ATP keeps updating the window. It maintains an unacknowledged packet pointer (`UNA`) that points to the unacknowledged packet with the minimum sequence number, and an in-flight packet pointer (`INF`) that points to the packet with the maximum sequence number sent so far. ATP also maintains a window size (`WND`). The worker continuously sends packets until a window of packets are in-flight (i.e., `INF - UNA = WND`). ACKs from the PS move the window forward (i.e. `UNA < ACK.Seq ? UNA : ACK.Seq`).

PS updating parameters. ATP allocates an area of memory at the PS for collecting the aggregated gradients. It is an array of <bitmap, element>. ATP uses bitmap to track which worker gradient packets corresponding to that element have been aggregated. If the incoming gradient packet is completely aggregated, ATP replaces the value in the corresponding element (setting the bitmap to be all ones). If the incoming gradient packet is an individual gradient packet, ATP checks if gradient packet from that worker is already aggregated — duplicates (with bitmap at the PS for that worker already set to 1) are eliminated and unseen packets are aggregated (and entry in the bitmap for that worker is set to 1). Once a gradient element is completely aggregated, the aggregated value is added to the corresponding parameter element maintained separated at the PS.

Worker pulling parameters.

After sending an initial window of gradient packets, the worker waits for parameter packets from the PS. A parameter packet has an ACK number. In the ideal case, the parameter packet increases the `UNA` monotonically and sequentially. Once `UNA` increases, more gradient packets are sent (increasing `INF` until `INF - UNA = WND`). **Worker retransmitting lost packets.** Due to various reasons (*e.g.,*, gradient packet loss or parameter packet loss), the parameter packets sent by the PS may not be in a sequence (*e.g.,*, out of order or missing an intermediate ACK sequence number). In this case (`ACK > UNA`), the worker updates the received parameters, but does not update `UNA`. When three consecutive parameter packets have ACK number different from expected ACK sequence number, `UNA` (indicating the gradient or parameter packet with `UNA` is lost), ATP retransmits the gradient packet corresponding to the sequence number `UNA`. Meanwhile, when ATP receives a parameter packet with the sequence number `UNA`, it increases `UNA` to the next unacknowledged sequence number.

Rate Control. In the multi-tenant network, multiple ATP jobs and other applications share the network. They contend for various resources including network bandwidth, receiver CPU, and switch buffers. In ATP, multiple jobs also contend for aggregators at the switches. ATP integrates the resource sharing of switch aggregator with existing congestion control.

For the case of congestion control in TCP, the sender uses RTT, duplicated ACK, or ECN marking as the signal of network congestion. It actively adjusts its sending window in response to these congestion signals. Similarly in ATP, high contention for aggregators in the switch can lead to a situation where aggregators cannot aggregate all traffic. This causes the traffic volume to increase, which can trigger queue length buildup in switches, or packet loss in switch buffers, or receiver NIC overload. In any of these cases, the symptom is similar to network congestion.

ATP picks the ECN mark as the congestion control signal. The other well-known congestion control signal RTT, measured from when a worker sends a model update packet until it gets the parameter ACK packet back, is not the right choice due to the two main reasons: (1) RTT cannot reflect the network congestion, this is because per packet RTT also includes the synchronization delay between workers. (2) In addition, different workers get different RTT values which worsens synchronization delays. A worker who sends the first gradient packet for a particular sequence number registers higher RTT than a later one. This causes different sending rates from

different workers. This can cause stragglers in the SGD computation and slow down the training speed. As all the workers receive the same aggregation packet, using ECN as the congestion signal avoids the above problem. We enable the ECN marking in switches, and use both ECN and packet loss as the congestion signal.

Each ATP worker applies Additive Increase Multiplicative Decrease (AIMD) scheme in response to these congestion signals to adjust its window (*i.e.,* adjusting `WND`). The window size in ATP is counted by bytes to be compatible when ATP packet size changes. ATP starts with 50 ATP packets initially per thread, which is within RTTbytes (60KB) in 100Gbps network. it then increases window size by one MTU (=1500Bytes) after receiving a parameter packet until it reaches a threshold, which is similar to the slow start phase in TCP. Whenever a worker receives a packet with ECN marked, it halves the window, updates the slow start threshold to the updated window. Otherwise, it increases window size by one MTU per window. The number of ATP packets in flight is a ceiling value of the window size divided by the ATP packet size.

6 Implementation

Programmable Switch. Switch logic implementation is challenging due to two main limitations. First, the switch only has a limited number of stages and a limited time budget for parsing each packet. Second, the same register can only be accessed once by each packet.

Packet-size limitation. Given that the whole packet has to be parsed and processed in the switch in limited stages and time means that there is a restriction on the packet size. This limits the packet size to 184 bytes in a prior work [26]. ATP increases this limit by leveraging multiple switch pipelines to do aggregations. Our optimizations increase the packet size limit in ATP to 306 bytes.

Switch-logic. There are two main pieces to switch-logic implementation. First is the implementation of aggregation operation. We carefully distribute aggregation across multiple stages and pipelines. Second is the implementation of control operations. After the aggregation operator implementation, only 4 stages remain for switch logic operations. Our control operation implementation fits this limited budget. Control operations involve hashing packets for aggregator addressing and membership verification, packet field updates, and other switch logic operations. Also, to operate with the restriction of one-time access to a register in the pipeline, ATP cleans the control operation information that is maintained in the registers during ACK processing received on the backward path from the PS.

End-Host Networking Stack.

Plugging ML Frameworks with ATP We implement ATP as a plugin of BytePS, which integrates PyTorch, TensorFlow, and MxNet. With BytePS, ATP intercepts the *Push* and *Pull* function calls at the workers and they communicate with an ATP parameter server.

Small Packet Optimizations The end host component is written with RAW ETHERNET [1], a Mellanox userspace network programming model. ATP leverages hardware acceleration features, i.e., TSO and Multi-Packet QP to accelerate small packets sending and receiving rate. TSO feature can speed up the packet sending rate by offloading packetization to the NIC and fully utilize the PCIe bandwidth via DMA of a large chunk of data. To enhance the packet receiving rate, Multi-Packet QP feature can reduce the NIC memory footprint by at least 512x via optimizations that shrink receiver queue entries. Both these features reduce CPU cost via fewer calls of packet send and receive, and fewer PCIe DMA reads for the NIC to fetch send-queue and receive-queue entries.

Baseline Implementation. We implement SwitchML prototype, which uses the switch as the PS and uses the packet loss recovery mechanism they propose. We leverage the small packets optimization from ATP at the end host to accelerate SwitchML performance. We do not implement multiple pipeline optimization from ATP to increase packet size limitations for SwitchML as it requires changes SwitchML's packet loss recovery. The default packet size of SwitchML is 180 bytes [26].

7 Evaluation

7.1 Experimental Setup

Cluster Setup. We evaluate ATP on a testbed with 9 machines, where 8 of them have one Nvidia GeForce RTX 2080Ti GPU with Nvidia driver version 430.34 and CUDA 10.0. All the 9 machines are equipped with Intel(R) Xeon(R) Gold 5120T CPU @ 2.20GHz, 192GB RAM and run Ubuntu 18.04 with Linux kernel 4.15.0-20. For the network component, a Mellanox ConnectX-5 dual-port 100G NIC is attached to each host with Mellanox driver OFED 4.7-1.0.0.1. All the servers are connected via a 32x100Gbps programmable switch with Barefoot Tofino chip.

Baselines. We compare ATP with BytePS (it employs worker-PS architecture). BytePS supports TCP (BytePS + TCP) and RDMA over Converged Ethernet [13] (BytePS + RDMA) as network protocols. To alleviate the network bottleneck for the traditional worker-PS architecture, we run as many parameter servers as the workers. Note that ATP only uses one parameter server in all the experiments. In addition, given that we have a limited number of machines, we co-locate the parameter servers and workers in the same machine, i.e., each machine runs one worker and one parameter server. We turn on PFC [24] to

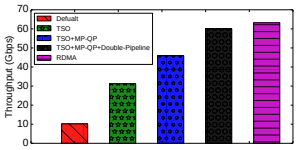


Figure 5: Throughput with different hardware acceleration.

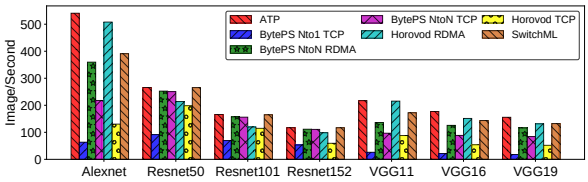


Figure 6: Performance

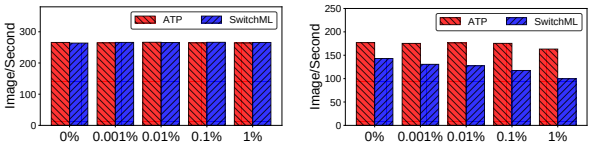
ensure network to be lossless at switch and hosts, which is required by RDMA over Converged Ethernet (RoCE) protocol. We also compare ATP with SwitchML, the state-of-the-art machine learning network scheme with in-network aggregation support. Apart from the parameter server approach, the other state-of-the-art aggregation approach is all-reduce. As BytePS does not support all-reduce aggregation mode, we compare ATP with ring all-reduce using Horovod with RoCE (Horovod+RDMA) and Horovod with TCP (Horovod+TCP).

Metrics. We use two performance metrics to benchmark and compare against other baselines - (1) application throughput, which is the number of images processed per second (*image/s*) normalized by the number of workers when training the popular models [11, 18, 28]; and (2) network throughput, which is the total bytes delivered to each worker per second (*network throughput*) when we run benchmark experiments.

System Configuration. The benchmark workload takes each tensor size as 4MB, which is the maximum tensor size supported by BytePS. The default job configuration is 8 workers for all evaluations, unless specified. We use two real-world models - VGG16 (model size 528MB) and ResNet50 (model size 98MB) - which represents the network-intensive and compute-intensive workload, respectively.

7.2 Microbenchmark

7.2.1 System optimization. ATP applies multiple hardware acceleration techniques to deal with small packet restrictions as mentioned in section 6 to boost throughput. To demonstrate the effectiveness of each optimization, we conduct the experiment by adding one optimization at a time, and measure the network throughput using benchmark workload. We launch two hosts, one as a worker, and the other as PS. The worker keeps sending $4MB$ data chunks to the PS. PS copies the received data from the receiver memory region to the sender memory region and then sends the data back to the worker.



(a) ResNet50 **(b) VGG16**

Figure 7: (a), (b) show the throughput in different packet loss rate between ATP and SwitchML in ResNet50, VGG16.

Figure 5 shows that the network throughput gains as we progressively add optimizations. ATP adopts **TSO+MP-QP+Double-Pipeline** optimization and this gives 6X improvement in throughput over the case with no hardware acceleration. In addition, we notice that the throughput does not double in going from **TSO+MP-QP** to **TSO+MP-QP+Double-Pipeline** when packet size doubles. We find that this is because throughput is bottlenecked by the memory copy, which also explains the reason **RDMA** does not get full line rate [2].

In summary, ATP is able to achieve high throughput with small packets by utilizing the recent advances in hardware acceleration. We believe these explorations can help boost the performance for applications that use the small packets.

7.2.2 ATP training performance. We evaluate the training performance on a suite of popular real world workloads by comparing ATP against all baselines: BytePS launching N workers and 1 PS over TCP (BytePS Nto1 TCP), BytePS launching N workers and N PS over TCP (BytePS NtoN TCP), BytePS launching N workers and N PS over RDMA (BytePS NtoN RDMA), SwitchML, Horovod over TCP (Horovod TCP) and Horovod over RDMA (Horovod RDMA). Figure 6 shows the training throughput for different models. Overall, it shows that ATP achieves the best performance with all kinds of models with speedup of up to 869%. ATP outperforms SwitchML because ATP uses larger packet size. In addition, we also can see that ATP gets larger performance gains on the network-intensive workloads (i.e., VGG) than the computation-intensive workloads (i.e., ResNet).

7.2.3 Packet Loss Recovery Overhead. ATP handles packet loss at the end host and guarantees aggregation correctness and prevents memory leaks in the switch. To evaluate the overhead of packet loss recovery, we configure one worker to drop packets with a packet loss rate between 0.001% and 1% and we turn off congestion control (to avoid window back-off with congestion control). We compare ATP against SwitchML. SwitchML uses

[2]The memory copy overhead can be eliminated by proper memory alignment. We leave this improvement for future work.

a timeout mechanism to detect packet loss. We set the timeout value as 1ms for SwitchML in our experiments.

Impact of loss rate. Figure 7a shows the training (network) throughput by varying loss rate. ATP degrades gracefully to a lesser degree than SwitchML when the loss rate increases. This is because ATP adopts out-of-sequence ACKs as a packet loss signal as well, which enables ATP to detect and respond to packet losses faster than SwitchML.

7.2.4 Emulate multi-job contention. There is no guarantee of an aggregator for every ATP packet. due to that the switch does not have enough aggregators available for the jobs. An ATP packet may experience: (1) the gradient packet from all the workers gets forwarded to the parameter server and gets aggregated in the parameter server. (2) the gradient packet from some workers gets aggregated in the switch, and the others get forwarded to the parameter server, which triggers the packets retransmission to take the partial results to the parameter server and deallocate the aggregator in the switch.

The main overhead from cases (1) is that the link to the PS can be the bottleneck as both computation and communication are added to the PS. We evaluate this overhead here by varying the forwarding rate. The forwarding rate is defined by the number of forwarding gradients packets divided by the total gradients packets per worker per iteration. In addition, case (2) also introduces the packet loss recovery overhead as section 4.2. The overall effect of case (1) and case (2) are also demonstrated in the later section.

Figure 8a shows the training throughput over VGG16 and ResNet50 for different forwarding rate. The performance degrades as the forwarding rate increases. Again, as expected, the performance degrades more for the communication-intensive workload (VGG16) than the computation-intensive workload (ResNet50). It inspires us that in the case of high contention, ATP can also launch multiple parameter servers to avoid the parameter server becomes the bottleneck.

7.3 Switch resource sharing between jobs

ATP allows multiple jobs to dynamically share the switch resources during the runtime. We show the effectiveness of such sharing. In the experiment, we start two jobs simultaneously such that both jobs contend for the switch resources. We tune the following experiment parameters: a job can be executed on ATP job, SwitchML, or the unmodified BytePS; the job model can be ResNet50 or VGG16; each job has one PS (for fair comparison) and a tunable number of workers. We measure the per worker training throughput as the metric.

7.3.1 Impact of the number of workers. We start two VGG16 jobs on ATP. In the first pass, two jobs have 2 and 5 workers respectively, and in the second, they both have 3. We observe that (1) when the same ATP workload (i.e., the same model and the same number of workers) compete for switch resources, they converge to an equal share, (2) A job with less number of workers is more competitive than that with more ones (149.5 image/s for 2 workers v.s. 129.3 image/s for 5). Because a job with a larger number of workers can cause higher traffic amount to the parameter server when it does not get an aggregator.

7.3.2 Dynamic sharing vs. Static allocation. One of the main features of ATP is the dynamic memory sharing mechanism. We evaluate the different combinations of dynamic/static job settings to demonstrate its effectiveness. A job can be configured as an ATP job (dynamically sharing the switch memory) denoted by "ATP"; or as a SwitchML job denoted by "SwitchML half/full" which is statically allocated half/full switch memory; or as an unmodified BytePS DNN job denoted by "Default". The static allocation strategy can either divide the switch memory by half for each job ("SwitchML half + SwitchML half") or it allocates the full memory to one job while the other job does not use the switch aggregation at all ("SwitchML full + Default").

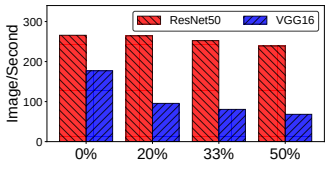
Same model. Figure 10a reports the training throughput for each job when the two jobs train the same VGG16 model. We observe that ATP outperforms all other settings as ATP uses the aggregation services (i.e. aggregators) as much as possible and is able to use memory more efficiently than that with static allocation of the same memory. In addition, two ATP jobs can share the switch memory equally with the same number of workers and the same workload. ATP dynamically achieves this goal, which indicates ATP is a good option for practical network aggregation without management overheads.

Different models. Figure 10b reports training throughput for each job when two jobs train two different models, ResNet50 and VGG16, respectively. We observe that ResNet50 job uses fewer network resources, and that the training throughput of the VGG16 job is higher than that of "ATP + ATP" as shown in Figure 10a. We observe that ATP outperforms all the other systems on the VGG16 job training, but has lower training throughput on ResNet50.

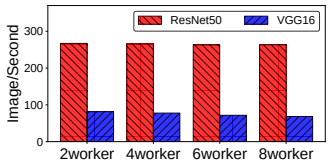
7.4 Co-locate with other traffic

ATP's congestion control mechanism aims to minimize packet loss. Network congestion happens (1) when ATP does not perform in-network aggregation, and there is incast at PS; (2) ATP traffic is co-located with bandwidth-hungry normal traffic, i.e., TCP, RDMA, etc. We evaluate congestion control mechanism in these cases.

With ATP traffic. For case (1), we repeat the experiment as in Figure 8a, where we have a forwarding rate of



(a) VGG16 and ResNet50

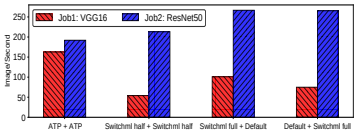


(b) Fixed 50% forward rate

Figure 8: Forwarding rate in different workload

Figure 9: Multiple jobs in different number of workers

(a) VGG16 and VGG16



(b) VGG16 and ResNet50

Figure 10: Running two jobs simultaneously in different systems

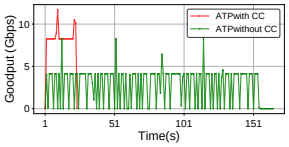
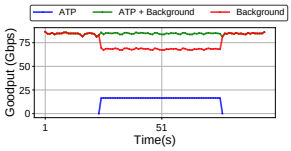
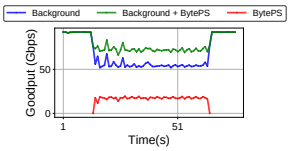


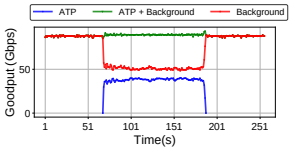
Figure 11: ATP job with and without congestion control



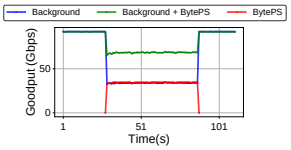
(a) ATP: VGG16



(b) BytePS+RDMA: VGG16



(c) ATP benchmark



(d) BytePS benchmark

Figure 12: System performance with background traffic

50%. We report the network throughput per second over time with and without congestion control in Figure 11. We observe that the throughput without congestion control is almost half of the throughput with congestion control. To understand the reason, we log packet loss events. Without congestion control, packets loss happens frequently and thus, introduces a lot of packet loss recovery overhead. The training throughput ATP with congestion control is 66.8 img/s while that without congestion control is 23.2 img/s, a decrease of 3X. This indicates that the congestion control for ATP is very effective.

With background traffic. With this experiment we validate the congestion control efficiency when co-locating with other traffic. We launch a VGG16 training job with 6 workers and 1 PS for ATP. As a comparison, we launch a BytePS over RDMA job with 6 workers and 6 parameter servers, where we put one worker and one parameter server at the same host. The RDMA traffic is configured to be lossless (using PFC). Then, we add background flows competing for bandwidth on a link to a worker. The background flows individually can achieve the line rate. We also conduct the same experiment on the benchmark workload for ATP and BytePS.

Figure 12a and Figure 12b report the goodput (the total parameter size finished per second) averaged per second over time from the worker that experiences network congestion. We can see that the aggregate throughput (ATP +Background) of ATP and background traffic is close to line rate while it is not for BytePS with RDMA. This is because each host runs both workers and servers for a BytePS+RDMA job. It indicates that ATP produces less network traffic compared with the systems that do not use the in-network aggregation service. In addition, we also see that the flows that use congestion control of ATP were able to respond quickly to changes in congestion, and converge to a new bandwidth share which is very stable. The results with benchmark (Figure 12c and Figure 12d) shows a similar trend.

8 Related Work

Network optimization for DNN training (DT) fall in two main categories. In the first category, the DT job is re-architected to increase the overlap between GPU/CPU compute and network use ([32], [23], [14], [9], [12], [5], [7], [10]). The second category offloads end-host computation to high-speed in-network devices [27], and ATP falls into this category. In addition, there are engineering efforts that accelerate network throughput, and these technical approaches can be integrated with most other architectural solutions. For example, BytePS supports using RDMA to transmit tensors [23]. ATP leverages TSO, Multi-packet QP, and DMA.

9 Conclusion

We build a traffic aggregation protocol ATP for DT jobs in multi-tenant networks. ATP does *best effort in-network aggregation* that requires careful co-design of switch logic (for aggregation) and end-host networking stack (for reliability and congestion control).

This work does not raise any ethical issues.

References

[1] RAW Ethernet Programming. https://docs.mellanox.com/display/MLNXOFEDv461000/Programming.

[2] Google tpu. urlhttps://cloud.google.com/tpu/.

[3] Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, February 2020. USENIX Association.

[4] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[5] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 571–582, 2014.

[6] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[9] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[10] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[13] InfiniBand Trade Association. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.1 Annex A17: RoCEv2. https://cw.infinibandta.org/document/dl/7781, 2014.

[14] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed dnn training. *arXiv preprint arXiv:1905.03960*, 2019.

[15] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Tech. Rep.*, 2018.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[17] Jack Kiefer, Jacob Wolfowitz, et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[19] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[20] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: energy-efficient microservices on smartnic-accelerated servers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 363–378, 2019.

[21] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 143–157, 2019.

[22] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[23] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[24] 802.1qbb. http://1.ieee802.org/dcb/802-1qbb/.

[25] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a programming system for nic-accelerated network applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 663–679, 2018.

[26] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. 2019.

[27] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

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

[29] Zhicheng Yan, Hao Zhang, Robinson Piramuthu, Vignesh Jagadeesh, Dennis DeCoste, Wei Di, and Yizhou Yu. Hd-cnn: hierarchical deep convolutional neural networks for large scale visual recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 2740–2748, 2015.

[30] Zhicheng Yan, Hao Zhang, Baoyuan Wang, Sylvain Paris, and Yizhou Yu. Automatic photo adjustment using deep neural networks. *ACM Transactions on Graphics (TOG)*, 35(2):1–15, 2016.

[31] Jiang Yimin. byteps. https://github.com/bytedance/byteps.

[32] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for

distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 181–193, 2017.

[33] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on {GPU} clusters. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 181–193, 2017.

[34] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan RK Ports, Ion Stoica, and Xin Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of the VLDB Endowment*, 13(3):376–389, 2019.