ATP: In-network Aggregation for Multi-tenant Learning

ChonLam Lao [††*], Yanfang Le[†*], Kshiteej Mahajan[†], Yixi Chen[††],
Wenfei Wu[††], Aditya Akella[†], Michael Swift[†]

University of Wisconsin-Madison[†], Tsinghua University[††]

Abstract

Distributed deep neural network training (DT) systems are widely deployed in clusters where the network is shared across multiple tenants, i.e., multiple DT jobs. Each DT job computes and aggregates gradients. Recent advances in hardware accelerators have shifted the the performance bottleneck of training from computation to communication. To speed up DT jobs' communication, we propose ATP, a service for in-network aggregation aimed at modern multi-rack, multi-job DT settings. ATP uses emerging programmable switch hardware to support in-network aggregation at multiple rack switches in a cluster to speedup DT jobs. ATP performs *decentralized, dynamic, best-effort* aggregation, enables efficient and equitable sharing of limited switch resources across simultaneously running DT jobs, and gracefully accommodates heavy contention for switch resources. ATP outperforms existing systems accelerating training throughput by up to 38% - 66% in a cluster shared by multiple DT jobs.

1 Introduction

Traditional network design relied on the end-to-end principle to guide functionality placement, leaving only common needs implemented within the network, primarily routing and forwarding. However, datacenter networks and workloads have evolved, and there is a strong case to support common application functionality within the network [20, 37, 66].

Deep Neural Networks (DNN) are emerging as a critical component of more and more enterprise applications such as computer vision [30], natural language processing [24, 62], databases [60], compilers [61], systems [63], and networking [50]. These applications all require distributed DNN training (DT) to iteratively train better DNNs for improved prediction performance. Enterprises typically run DT on multi-rack clusters [10] shared by other applications. Each DT job has several workers and parameter servers (PS) spread across several machines. Workers compute gradients and send these gradients to the PS(s) over the network for aggregation. *Gradient aggregation*, which combines partial results from multiple workers and returns a single aggregated result, is commonly used in DT, and contributes substantially to overall training time [44]. Recent advances in special hardware [4, 10] have shifted the performance bottleneck of distributed training from computation to communication [44, 52]: VGG16 training can be 4X faster without network communication [52].

Further, datacenter networks are becoming feature-rich with the introduction of new classes of programmable network devices such as programmable switches (e.g., Intel's FlexPipe [6], Cavium's XPliant [11], Barefoot Tofino [2]) and network accelerators (e.g., Cavium's OCTEON and LiquidIO products [7], Netronome's NFP-6000 [8], and FlexNIC [39]). Together, they offer in-transit packet processing and in-network state that can be used for *application-level stateful computation* as data flows through the network.

Current DT stacks implement gradient aggregation purely in the application. However, the emergence of DT as a common application and its reliance on gradient aggregation, as well as the emergence of application-level stateful computation as a network feature, suggests an opportunity to reduce training time by moving gradient aggregation inside the network. This reduces network bandwidth consumption from workers to the PS(s). For both single DT and multiple DT jobs (i.e., multi-tenant settings) this bandwidth allows pushing more gradients through the network, and increases the total throughput of gradient flows thereby reducing training times.

Recent proposals show the initial promise of such in-network aggregation: e.g., SwitchML [52] increases training throughput for VGG16 by 2X via in-network aggregation on a programmable top-of-rack switch. However, the general problem of making aggregation a true in-network service to be leveraged by multiple DT tenants in a multi-rack/multi-switch cluster has not received systematic attention. Realizing such a service calls for mechanisms to share limited multi-switch aggregation resources across multiple tenants, and mechanisms to deal with heavy contention for switch resources.

The key goal of our work is to speed up multiple DT jobs running simultaneously in a cluster by maximizing the benefits from in-network multi-switch aggregation, and distributing these benefits across multiple DT jobs in an equitable manner. To do so, we propose a new network service for multi-rack clusters called Aggregation Transmission Protocol, i.e., ATP. ATP supports *dynamic aggregation* at rack switches. DT jobs go through 'on' and 'off' gradient aggregation phases, and ATP uses decentralized mechanisms to ensure that switch resources used by DT job entering its off phase can be dynamically reused by DT job in its on phase. ATP supports *best-effort* aggregation. This enables DT jobs to gracefully fall back to end-host aggregation under heavy contention from many tenants without extra overhead.

ATP chunks gradients for each DT job into fixed size fragments that we refer to as *gradient fragment packets* and par-

*Co-primary authors

titions programmable switch resources into the same fixed size fragments called *aggregators*. As these gradient fragment packets flow through the network, ATP opportunistically aggregates them by accumulating results at the earliest available programmable switch, or in the worst-case at the PS end-host.

ATP proposes a decentralized aggregator allocation mechanism that supports aggregation at line rate for multiple jobs by dynamically allocating free aggregators when gradient fragment packets arrive at a switch. A key issue with an in-network aggregation service is that traditional end-to-end protocols do not work when gradient fragment packets are consumed in the network due to aggregation, as that may be misinterpreted as packet loss. Thus, ATP co-designs the switch logic and end host networking stack, specifically to support reliability and effective congestion control.

Our implementation works atop clusters using P4-programmable switches. Such switches expose a limited set of in-network packet processing primitives, place ungenerous memory limits on network state, and have a constrained memory model restricting reads/writes. We overcome these constraints, and show how ATP can support highly effective dynamic, best-effort aggregation that can achieve 60Gbps.

Our implementation also has mechanisms that improve state-of-the-art floating point value quantization to support limited switch computation. ATP's implementation adopts a kernel bypass design at the end-host so that existing protocol stacks are not replaced by ATP's network stack and non-ATP applications can continue to use existing protocol stacks.

We run extensive experiments on popular DNN models to evaluate ATP in a single rack testbed with multiple jobs. Our evaluation shows that in multi-tenant scenarios, dynamic, best-effort in-network aggregation with ATP enables efficient switch resource usage. For example, the performance only decreases by $5-10\%$ when only half of the desired aggregators are available, and outperforms current state-of-the-art by 38% when there is heavy contention for on-switch resources. We simulate multi-rack cluster experiments with a typical topology and show a 66% reduction in network traffic with ATP. We benchmark loss-recovery and congestion control algorithms proposed in ATP. The loss recovery mechanism of ATP outperforms the state-of-the-art (SwitchML) by 34% and an ATP job with congestion control speeds up $3X$ compared to one without congestion control.

2 Background and Motivation

2.1 Preliminaries

PS Architecture. This design, shown in Figure 1, enables data-parallel training, where training data is partitioned and distributed to workers. There are two phases: gradient computation, where workers locally computes gradients; and gradient aggregation, where workers' gradients are transmitted over the network to be aggregated (which involves addition of gradients) at one or more end-hosts called parameter servers (PSs). The aggregated parameters are then sent back to the workers. Gradients are tensors, i.e., arrays of values. With multiple PSs, each PS has a distinct partition of parameters.

Programmable Switch. The recent emergence of programmable switches provides opportunities to offload application-level stateful computation [37, 43, 66]. A popular example is the Tofino switch [2], which we use.

Programmable switches, such as the Tofino, have constrained compute resources and memory ($\sim$ 10MB [49]) and expose on-chip memory as stateful and stateless objects. Stateless objects, *metadata*, hold the state for each packet, and the switch releases this object when that packet is dropped or forwarded. Stateful objects, *registers*, hold state as long as the switch program is running. A *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. A register is an array of 32-bit integers. In the context of in-network aggregation, each packet has a subset of gradient values and needs a set of registers to aggregate them. We call this set of registers an *aggregator*.

Register memory can only be allocated when the switch program launches. To change memory allocation, users have to stop the switch, modify the switch program and restart the switch program. The computation flexibility is limited by the number of stages, parser limitations, and the time budget at each stage: only independent functions can be placed in the same stages and the number of such functions in the same stage is also limited. These limits lead to small packet sizes for in-network computation and storage applications: the payload size of SwitchML and NetCache is 128B [36, 37, 42, 52]. [1]

In-Network Aggregation. Gradients can be seen as a sequence of fragments (each fragment has a subset of gradient values), and aggregation (addition of gradients) of all the gradients is aggregation of each of these fragments. In-network aggregation for each fragment is done in a specific aggregator. Figure 2 exemplifies this for a DT job with two workers using one programmable switch. Workers 1 and 2 create packets having a fragment with 3 tensor values and send them to the switch. Suppose the switch first receives the packet $p1$ from worker 1. It stores the tensor values contained in $p1$ in the aggregator's registers $R1$, $R2$, $R3$. The switch then drops packet $p1$. When the switch then receives packet $p2$ from worker 2, it aggregates the tensor values contained in $p2$ with contents of $R1$, $R2$, $R3$. If there were additional workers, the switch would update the registers with the aggregation of both packets. In this example, because $p2$ is from the last worker, the switch overwrites the values in packet $p2$ with the aggregated result and multicasts $p2$ to both the workers.

A recent work, SwitchML [52], prototypes this idea for a single DT job in a rack-scale network. We use SwitchML as an example to illustrate the design space and underscore

[1] The exact parameters of programmable switches and ATP are specific to "Tofino" programmable switches; if other programmable switches has the similar limitations, ATP can be used similarly.

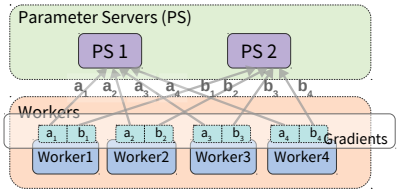


Figure 1: Parameter servers (PS)

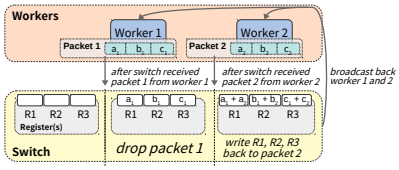


Figure 2: In-network aggregation example

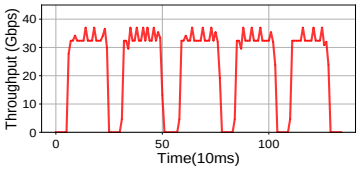


Figure 3: A DT job training VGG16 shows on-off communication pattern for a simple one worker-one PS setting.

the key attributes of an ideal in-network aggregation service. SwitchML offloads gradient aggregation *entirely* to the top-of-rack switch. It allocates a static pool of aggregators in the rack switch to a DT job, and streams gradient fragment(s) from workers to the switch only after previously sent gradient fragment(s) are aggregated and have vacated aggregator(s) on the switch. We argue next that static aggregator allocation approaches like SwitchML fall short under real-world multi-tenant training, necessitating a systematic in-network service.

2.2 In-Network Aggregation Service

When applied to multiple DT jobs, SwitchML requires static partitioning of switch resources, where each job is statically assigned to a partition. In a multi-tenant scenario, this results in underutilization of switch resources. DT jobs go through on and off gradient aggregation phases as shown in Figure 3, and switch resources belonging to a DT job in the off phase can be shared with a DT job in the on phase in a *dynamic manner*, but static partitioning precludes this.

SwitchML offloads gradient aggregation for each DT job entirely to the rack switch. With heavy switch resource contention, DT jobs have to wait for switch resources leading to underutilization of the network link bandwidth from the workers to the PS(s). In a better design, a DT job could instead aggregate a fraction of gradients at the switch in a *best-effort manner* while aggregating the rest at the end-host.

Thus, an ideal in-network aggregation service should support *dynamic, best-effort* multi-tenant gradient aggregation for optimal efficiency and speedup. As we show in Section 3, realizing dynamic best-effort aggregation requires key innovations at end-hosts and in how switch resources are apportioned and dynamically (re)used. In addition, an in-network aggregation service brings to fore two other aspects of the network stack that need redesign, namely, reliability and congestion.

Rethinking Reliability. In-network aggregation breaks end-to-end semantics as some packets are consumed inside the network during aggregation. Traditional end-host based reliability mechanisms can misinterpret in-network packet consumption as a packet loss, leading to unnecessary retransmissions and lead to incorrect gradient aggregation due to inability of existing reliability mechanisms in dealing with these new class of packet events. Thus, we need a new *reliability algorithm* to deal with this new class of packet events.

Rethinking Congestion-Control. In the multi-tenant case, the network resources (switch aggregators and network bandwidth) available to a DT job fluctuates because (1) DT jobs exhibit on-off communication phases (Figure 3), (2) the total number of DT jobs varies and (3) background traffic varies. Utilizing fluctuating network resources efficiently and sharing them fairly depends on congestion control. However, as end-to-end semantics are broken we cannot use traditional congestion control algorithms that rely on RTT or drops as congestion signal. We need a new *congestion control algorithm* that identifies the right congestion signal so as to modulate the throughput of gradient fragments from workers' for each DT job to meet the requirements of efficient use and fair division of network resources across DT jobs.

3 Design

ATP is a network service that performs *dynamic, best-effort aggregation* across DT jobs. ATP design aligns with guidelines for building robust and deployable in-network computation [49]: (1) Offload reusable primitives: ATP is a network service for in-network aggregation and a common function to different DT frameworks; (2) Preserve fate sharing: ATP is able to progress in the event of network device failure via fall-back to aggregation at the end-host; (3) Keep state out of the network: ATP's reliability algorithms are able to recover lost data and deal with partial aggregation; (4) Minimal interference: ATP chooses aggregation only at Top-of-Rack (ToR) switches to sidestep issues owing to probabilistic routing in the network.

3.1 ATP Overview

ATP is a transport that specifically targets in-network aggregation of gradient tensors in DT applications; it is not a general-purpose transport. Compared to general-purpose TCP: (a) ATP redesigns specific transport features, such as reliability, congestion control, and flow control to apply optimally to its target context. (b) ATP does not implement TCP's in-order byte-stream and full-duplex abstractions as they do not apply to the target context.

ATP performs aggregation at the granularity of fragments of a gradient that fit in a single packet, i.e., *gradient fragment packets*. ATP chunks the gradient tensor at each worker into a sequence of fixed-size fragments such that each fragment fits in a packet and assigns each a sequence number. Gradient aggregation for a DT job merges values at the same sequence number from each worker's tensor.

Upon booting, each ATP programmable switch allocates a portion of switch register memory to be shared by ATP jobs. This memory is organized as an array of fixed-size

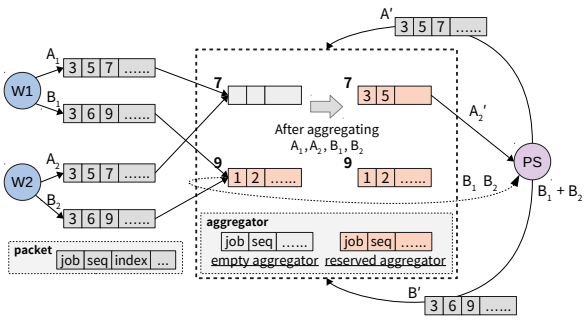


Figure 4: ATP dynamic, best-effort aggregation example. The *directions* fields are *job ID, sequence and aggregator index* in packet. *soft-state* is the values in the aggregators.

segments, which we refer to as *gradient fragment aggregators*, or just *aggregators*. Each aggregator is accessed by its index in the array, and aggregates gradient packets with a specific sequence number belonging to the same DT job.

ATP workers stream gradient fragment packets to the PS(s). ATP aggregates gradient fragment packets inside the network when in-network resources are available. If in-network resources are unavailable, gradient fragment packets are sent to the end-host PS for aggregation. ATP restricts in-network aggregation to ToR programmable switches. This means that gradients from each worker can at most be aggregated at two levels – (1) the rack switch at the worker and (2) the rack switch at the PS. This requires coordination of decisions to ensure that each gradient fragment packet is aggregated *exactly once*. We use a decentralized, dynamic, best-effort approach to determine where aggregation occurs.

Gradient fragment packets contains *direction* fields. These directions interact with the ATP switch logic at the programmable switches, to program *soft-state* in the aggregator to elicit a coordinated decision. The aggregator soft-state can be discarded at any time, leading to aggregation at the PS instead of the switch. The directions in a gradient fragment packet comprise fields that help decide at which programmable switch(es) to aggregate, in which gradient aggregator to aggregate, and to identify completion or failure of aggregation at an aggregator. Switch logic uses these directions to program soft-state in the switch that identifies whether a gradient aggregator already exists for an incoming gradient fragment, and keep track of intermediate aggregation results and completion of aggregation.

Soft-state in switches and directions in packets ensure that ATP does not require job-specific switch program changes (and avoids switch restarts) upon job arrival/departure.

Figure 4 exemplifies how ATP achieves dynamic, best-effort aggregation. A job with ID 3 has two workers, $w1$ and $w2$. The workers compute gradients which are subsequently broken by ATP at end hosts into two packets each - (A_1, B_1), and (A_2, B_2). ATP aggregates gradient packets A_1 with A_2, and B_1 with B_2, either at the switch or at the PS, as explained next. Packets A_1 and A_2 are routed and hashed to aggregator 7; since the aggregator is empty, it is "reserved" by packet A_1 by changing the aggregator's *soft-state* to its job ID and packet sequence. When A_2 arrives at the switch, it hashes to the same aggregator and triggers aggregation; then, the resulting packet containing the aggregation result, A'_2, is sent to the PS. In contrast, packet B_1 can not reserve aggregator 9, because it is reserved by a packet with job ID 1 and sequence 2. Thus, packet B_1 is forwarded directly to the PS; the same occurs with B_2. Packets B_1 and B_2 are aggregated at the PS. For either pair of packets, the PS sends the parameter packets (A' and B') via multicast back to workers $w1$ and $w2$. When the switch receives A', aggregator 7 is deallocated and set as empty (i.e., A' is hashed to aggregator 7, and the aggregator's job ID and sequence match with those in A') to enable aggregator 7 to be used for by future fragments from another job.

To detect and deal with packet losses ATP uses timeout based retransmission or out-of-sequence parameter ACK packets from the PS. When a packet is retransmitted, it sets the resend flag. This serves as a direction for the switch to deallocate and transmit any partially allocated result to the PS. Also, to deal with congestion if there is queue buildup, say when queues were above a certain threshold when packet A_2 was received, an ECN flag in A_2 is set and carried over to A'_2. This is copied to the parameter packet from PS A' and received by the workers who adjust their windows. The window adjustment is synchronized in both the workers as it is triggered by the same ECN bit in A'.

3.2 ATP Infrastructure Setup

ATP requires a one-time static setup involving programming and restarting switches to bring the service up. Any dynamic per-job setup is managed by inserting the appropriate job-specific directions in gradient fragment packets.

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—gradient and parameter packets—and allocates a portion of switch resources—aggregators—to aggregate ATP traffic. The end host installs an ATP networking stack, which intercepts all the push or pull gradient calls from DT jobs. End-hosts have knowledge of the network topology—switch, end-host port connectivity, and count of aggregators at a switch—so they can orchestrate aggregation across multiple switches.

Dynamic Per-Job Setup. Each new DT job is assigned a unique job ID. The job assigns each worker an ID from 1 to W, where W is the total number of workers. The job tracks the location of workers in the network topology to build an aggregation hierarchy. In case workers and PS are spread across racks, the job can use multiple switches for in-network aggregation. The ATP networking library computes the job's worker fan-in at each level of the aggregation hierarchy, which is used to determine when aggregation is complete (§3.5).

ATP uses IGMP to build a multicast distribution tree for

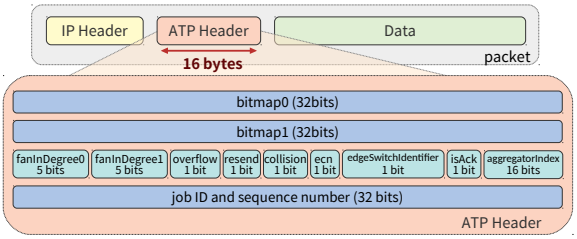


Figure 5: ATP packet format

the PS to return parameters to workers.

3.3 Data Structures

Packet Format. Figure 5 shows the gradient fragment packet format. The ATP header fields comprise directions and contain metadata about the fragment. The fields use one-hot encoding to identify the worker's position in the aggregation hierarchy (`bitmap0`), and the first-level switch's position at the second edge-switch (`bitmap1`). The fan-in degree indicates the number of workers attached to the first edge switch (`fanInDegree0`) and workers or switches attached to the second edge switch (`fanInDegree1`). These four fields are used to determining when aggregation has completed (§3.5).

The `resend` flag is set if it is a retransmitted gradient packet (§3.7). The `edgeSwitchIdentifier` flag is set to 0 if the packet is en-route to the first edge-switch in the aggregation hierarchy and 1 if the packet is en-route to the second edge switch. The `ECN` flag is marked by a switch when the switch's output queue length exceeds some threshold when the packet was sent. The `collision` flag is marked by a switch when it forwards a gradient packet onward due to the aggregator not being available because it is in use by a different job. The job ID and sequence number are used to match gradient packets from different workers on the same job. Finally, the data field contains tensor values (or aggregated tensor values).

Parameter packets use the same packet format, but indicate the different contents by setting the `isAck` flag. They are multicast from the PS to workers when an aggregation is complete, and serve as acknowledgments (ACKs) for the gradient packets sent by the workers. Workers detect dropped packets when they receive out of order parameter packets, which triggers them to resend gradient packets for aggregation (§3.7).

Switch Memory. Switch memory is organized as an array of fixed-size aggregators, each accessed by its index in the array. The `value` field contains aggregated data from different workers. The size of the value field is the same as the size of a gradient fragment value. The `bitmap` field records which workers have already been aggregated to the aggregator's value field. The `counter` field records the number of distinct workers included in the aggregated value. The `ECN` field records congestion status and is set if any aggregated packet had the `ECN` flag set. The `timestamp` field is updated when an aggregation is performed, and is used to detect when an aggregator has been abandoned (e.g., when a worker fails) and can be deallocated (§3.7). The identifier fields <`Job ID, Sequence Number`> uniquely identify the job and the fragment that this aggregator serves. Figure 15 in §A.1 shows this layout.

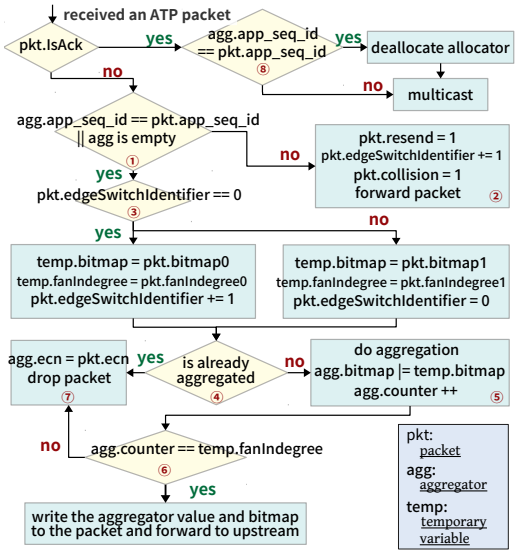


Figure 6: Pseudocode of the switch logic in the ideal case.

3.4 Inter-rack Aggregation

Scaling aggregation beyond a single rack provides more flexibility w.r.t. where DT executes in a cluster. Aggregating just at a worker's local ToR switch is simple, but leads to unnecessary network traffic to the PS when workers reside in different racks. Alternatively, aggregation can be done at higher layers of the network topology. However, this approach greatly increases protocol complexity because the system has to handle route changes in the interior of the network. For example, ECMP-based routing can change the number of gradient streams incident at a particular switch in the interior of the network. This necessitates careful coordination between network routing and the aggregator allocation mechanism. Thus, ATP only deploys in-network aggregation in ToR switches, either at the worker's rack (level-0) or at the PS's rack (level-1). This complements a recent study which shows that programmable switches today are usually deployed at ToR switches for near-server computation offloading [25].

To coordinate where aggregation occurs, ATP uses two groups of `bitmap` and `fanInDegree` fields in gradient packets, i.e., `bitmap0/1` and `fanInDegree0/1` as shown in Figure 5. The `edgeSwitchIdentifier` field indicates which bitmap and degree a switch should use when processing the gradient packet, the first or second level of aggregation. When a level-0 switch forwards a packet, it sets the `edgeSwitchIdentifier` bit in the packet header. The bitmap size limits the total number of workers ATP can support. As our testbed switch supports only 32-bit values, our implementation can support up to 1024 (=32×32) hosts; in general if the programmable switch can support n-bit values, ATP can support up to n^2 workers.

3.5 Switch Logic

Switch logic implements the algorithm that supports the *dynamic, best-effort* in-network aggregation service. It provides aggregator allocation, deallocation, and gradient aggregation. The allocation policy in ATP is First-Come-First-Serve without preemption: if a gradient fragment packet can reserve an aggregator, it keeps the aggregator until it is deallocated. In the ideal case, each edge switch completes aggregation of all incident worker gradient packets and sends the aggregated result upstream. We describe how failure cases (e.g., packet loss) are handled in §3.7. A detailed flowchart outlining switch logic can be found in Figure 6.

Aggregator Allocation. The arrival of a gradient fragment packet triggers aggregator allocation. When a gradient fragment packet arrives, the switch checks the availability of the aggregator at the packet's `aggregatorIndex` field. End-hosts compute `aggregatorIndex` as HASH(<Job ID, Sequence Number>)%`numAggregators`, which is consistent across all workers in a job. However, hash collisions could cause `aggregatorIndex` from different fragments and different jobs to map to the same index: e.g., in Figure 4, gradient fragment packets of Jobs 1 and 3 collide at index 9.

If the identifier field <Job ID, Sequence Number> in the aggregator is empty, we store the packet's identifier in the aggregator, and copy the gradient packet's `data` field into the aggregators `value` field. The switch copies the bitmap filed in the aggregator from the appropriate bitmap field in the packet (`bitmap0` if `edgeSwitchIdentifier` is 0, else `bitmap1`), and initializes the counter to 1.

If the aggregator's identifier field is non-empty, the switch compares it to packet's identifiers (box ① in Figure 6). If they are different, there is a hash collision and ATP pushes the gradient fragment packet downstream to be processed at the PS. To avoid aggregation on this packet at downstream switch and propagate the collision information to the PS, ATP sets `resend` and `collision` flags. To indicate this packet came from a switch and not a worker, it flips the edge switch identifier in the packet (② in Figure 6) as well. If the aggregator and packet identifiers are equal, gradient aggregation occurs.

Gradient Aggregation. If an aggregator is available for a gradient fragment packet, ATP uses the `edgeSwitchIdentifier` to fetch the fan-in degree and the bitmap for this switch from the packet (③ in Figure 6). Then, ATP checks whether this packet has been aggregated by testing the packet's bitmap against the aggregator's bitmap (④ in Figure 6). If not, ATP aggregates (adds) the packet's gradient data to the value field in the aggregators and also or's the bitmap field in the gradient packet to the bitmap field in the aggregator (⑤ in Figure 6). ATP increments the aggregator's `counter` field. If the packet already been aggregated (e.g., it was resent), ATP `OR`'s the `ECN` field from the packet into the aggregator's `ECN` field and drops the packet (⑦ in Figure 6).

If the `counter` in the aggregator is less than the corresponding fan-in degree (⑥ in Figure 6), ATP drops the gradient fragment packet — this is the step that saves bandwidth to the PS. If, however, they are equal then aggregation at this switch is complete and ready to be pushed upstream. The switch replaces the packet's data field with the aggregator's value field, and the corresponding bitmap field with the aggregator's bitmap and sends the packet upstream towards the PS.

Aggregator Deallocation using Parameter Packets. The PS multicasts parameter packets back to the workers when aggregation of fragments completes. A parameter packet works as the acknowledgement (ACK) of the gradient fragment packets, and must traverse the edge switches used for aggregation. When an ATP switch processes a parameter packet, the switch checks if the aggregator at the packet's index has a matching identifier, and if so deallocates the aggregator by changing all fields to null (⑧ in Figure 6).

3.6 End Host Logic

Workers in ATP push gradient fragment packets toward the PS and receive updated parameters back. The PS accepts gradient fragment packets, as well as partially or fully aggregated packets, to compute the full aggregation and sends the updated parameters back to workers. The PS also addresses collisions over aggregator indexes by *rehashing*.

Worker Pushing Gradients. The ATP end-host network stack obtains gradients by intercepting push or pull calls from DT jobs. It chunks these gradients into a sequence of 306B packets (58B header + 248B gradient values). ATP converts floating point numbers in gradients to 32-bit integers [52] to work with switches that do not support floating-point operations. These gradient fragment packets are small and ATP introduces optimizations for high packet I/O throughput (§4).

PS Updating Parameters. ATP allocates an area of memory for each job at the PS for collecting aggregated gradients as an array of <`bitmap`, `value`> indexed by sequence number. The bitmap tracks which workers' gradient fragments have been aggregated in the `value` field. PS maintains a bitmap for each value to track which worker values have been aggregated. When a gradient fragment packet arrives, the PS compares its bitmap with that of the packet for an overlap. If they do not overlap, the PS aggregates the packet's data into its stored value, and updates the stored bitmap from the packet's bitmap. For example, if the incoming gradient packet is an individual gradient packet, ATP checks if a packet from that worker was already aggregated (i.e., PS bitmap for the worker is set to 1) due to a resent packet, drops duplicates and otherwise updates the value and bitmap. On completion of aggregation of a parameter fragment, the PS multicasts the updated parameter fragment to send it back to all workers in the job.

For a single gradient fragment, it is possible that the aggregator is busy when the first few packets arrive (hash collision), but available for later packets (released). In this case, the first packets are forwarded directly to the PS, while the remaining ones are aggregated at the switch. However, without intervention the switch will never send along the aggregated

values because it is waiting for *packets that have already been sent*. Workers detect this stalled aggregation when they receive parameter packets for higher-sequenced fragments, and all workers will treat the stalled fragment as *a loss*. Each worker retransmits the stalled fragment with the `resend` bit set. This ensures completion of aggregation (by piggybacking on packet loss recovery; §3.7).

To reduce the frequency of aggregator collisions, we propose a *dynamic hashing scheme*. PS checks the 'collision' bit of gradient packets. If the 'collision' bit is set, the PS rehashes to get a new aggregator index (as HASH(<`aggregatorIndex`>)%`numAggregators`). It sends this new `aggregatorIndex` to workers in the unused `bitmap` field of the parameter packet. Workers associate the remap the gradients using the collision-prone index to the new index, and send any subsequent gradient that would have been sent with the old index with the new index instead. This simple approach helps *evolve* the hash function in a dynamic manner over time at each worker to make it collision resistant.

Worker Pulling Parameters. The network stack maintains a sliding window over the sequence of gradient fragment packets. After sending an initial window of packets, the worker records the first un-ACKed sequence number as the *expected sequence number* and waits for parameter packets from the PS. The worker uses the parameter packets from PS to slide the window and send new gradients packets. When a worker receives a parameter packet, ATP checks if it has the expected packet sequence number. If the parameter packet was already received (e.g., because it is lost by some other worker and retransmitted at that worker's request), it is ignored. If it has the expected number, ATP increments the expected sequence number and invokes the congestion control algorithm (§3.7) to update the current window. If the number of in-flight packets is less than the congestion window, ATP sends the remaining window (congestion window size - in-flight packets) of gradients fragment packets. If the parameter has a sequence number higher than expected, ATP may consider the expected gradient fragment as lost, triggering loss recovery (§3.7).

3.7 Reliability and Congestion Control

Dealing with Packet Retransmissions. Due to loss of gradient fragment or parameter packets, the PS may not send parameter packets in sequence. As noted previously, when this occurs, a worker updates the received parameters but does *not* update expected sequence number. When a worker receives three consecutive parameter packets other than the expected sequence number, it detects loss of the gradient fragment with the expected sequence number. In this case, ATP worker retransmits the missing fragment packet with the `resend` bit set; this indicates to switches that there may be partial aggregation state in switch.

ATP takes a simple approach and does not try to do in-network aggregation of resent gradients. It takes different steps at the first and second levels of aggregation. At the first level, when a resent packet arrives the switch checks for a matching aggregator. If it exists, and the aggregator bitmap does not indicate that the resent packet's fragment has already been aggregated, then the switch aggregates the value from the packet into the aggregator, merges the bitmap from the packet into the aggregator's bitmap, forwards the results (which may be partial) upstream, and *deallocates* the aggregator. When subsequent resent packets arrive, the corresponding aggregator has already been deallocated, so the switch simply forwards the resent packet upstream.

At the second level, the switch is unable to merge partial results because it does not store a bitmap for each of the first-level aggregations it has seen. Instead, the second-level switch discards its aggregation state and forwards any resent packet (including partial aggregations from the first level) to the PS, where the aggregation ultimately completes. This ensures that upon packet loss, all gradient fragments are (re)sent to the PS and the aggregator for the fragment is deallocated.

Memory leaks can occur when a job stops abnormally before all workers send gradients to the PS. Without any mechanism, the aggregator would remain occupied because the PS never ACKs with a corresponding parameter packet. To handle this, on every parameter packet, the switch checks the `timeout` value for the register specified by the parameter packet's index. Even if its job ID and sequence number do not match the parameter packet, the switch will deallocate the aggregator if the timestamp is older than a configured value.

Congestion Control. In a 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.

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 will trigger queue length buildup in switches and packet loss due to switch buffer overflow. The observable symptoms in this case are similar to network congestion. Based on this, ATP uses *congestion control* to manage *all* contended resources.

In TCP, senders detect congestion using RTT, duplicated ACKs, or ECN marks, and respond by adjusting sending windows. For ATP, we pick ECN marks as the primary congestion signal. RTT measured from a worker sending a gradient packet to it receiving a parameter ACK packet will not work because it includes synchronization delay between workers. As all the workers receive the same parameter packet, using ECN ensures that all workers see similar congestion signals. We enable the ECN marking in switches, and use both ECN and (rare) packet loss as the congestion signal. To ensure that ECN marks are not lost during aggregation, ATP merges the `ECN` bit in fragment packets into the `ECN` bit in aggregator (⑦ in Figure 6), which is later forwarded to the PS when aggregation completes. This `ECN` bit is then copied to the parameter packet and eventually reaches all the workers.

Each ATP worker applies Additive Increase Multiplicative Decrease (AIMD) to adjust its window in response to congestion signals. The window size ATP starts at 200 packets, which at 300 bytes each packet is within bandwidth-delay product ($\sim$ 60KB) of a 100Gbps network. ATP increases window size by one MTU (1500 bytes or 5 packets) for each received parameter packet until it reaches a threshold, which is similar to slow start in TCP. Above the slow-start threshold, ATP increases window size by one MTU per window. When a worker detects congestion via ECN marking on a parameter ACK or three out-of-order ACKs, it halves the window, and updates the slow start threshold to the updated window.

3.8 Dealing with Floating Points

Gradient values are real numbers and DNN training frameworks offer several numerical types to represent them, each type offering varying trade-offs between range, precision and computational overhead. Gradient values are typically represented using 32-bit floating point type.

The current generation of programmable switches does not support 32-bit floating point arithmetic. Like prior work [52], ATP converts gradient values at each worker from 32-bit floating point representation to 32-bit integer representation by multiplying the floating point number by a scaling factor (10^8) and rounding to the nearest integer. The switch aggregates these 32-bit integers and PS converts the aggregated value back to 32-bit floating points by dividing by the scaling factor. With ATP, we choose a high scaling factor so as to minimize loss of precision. We note that 32-bit floating point representation provides precision of 7 decimal digits [5]. Thus, to provide an equivalent precision ATP chooses the scaling factor of 10^8, which ensures that the number of epochs required to train a model to a target accuracy remains unchanged due to scaling/quantization. A detailed justification of ATP's choice of scaling factor can be found in §B.1.

A large scaling factor can lead to the overflow of aggregated gradients. There are two alternatives to deal with overflows: proactive and reactive. The former (described in §B.2) is cautious and wastes opportunities for in-network aggregation by sending some gradient packets directly to the PS for aggregation. We explain the pitfalls of this mechanism in §B.2. Instead, ATP uses a reactive mechanism: all gradient packets are sent with the intention of aggregation at a network switch. If a gradient packet triggers overflow at an aggregator in the switch, we utilize a switch feature (*saturation*) to set the aggregator value to the maximum value or minimum value represented with 32-bit integers. If the aggregator value is saturated, any further gradient packets destined at this aggregator only update the directions (i.e., bitmap, fanIndegrees) and the value remains saturated. When the aggregation is done, i.e., fanInDegree value is equal to the number of workers, the saturated aggregator value is written to the gradient packet and sent to the PS. If the PS finds the aggregator value is saturated, it requests the original gradient values in floating-point format from all workers. This triggers a retransmission and all the workers send packets with floating-point gradient values directly to the PS, which finally aggregates.

In the worst case, such a reactive overflow correction incurs the cost of retransmission of all gradient packets in an iteration. Note that the overhead incurred during overflow correction is exactly the same as that during packet loss recovery. In our evaluations, we see no deterioration in training throughput if the packet loss rate is $< 0.1\%$ (§5.2.4). This translates to overflow correction as well: if the frequency of overflow correction is $< 0.1\%$ (over packets), we will see no deterioration in training throughput. We empirically show that there is hardly any overflow for all popular models' training in §5.2.2. The frequency of overflow correction can be further reduced using a dynamic scaling factor (§C).

Because overflow correction adds little overhead, ATP's in-network aggregation yields substantial speed-up in per-epoch times. Coupled with quantization not affecting the number of epochs, ATP overall yields significant gains in time-to-target-accuracy, as we also show in §5.2.2.

4 Implementation

ATP's implementation consists of (i) P4 switch logic, and (ii) modifications to host network stacks to use hardware offloads to optimize small-packet performance.

Programmable Switch. The switch implementation has processing logic for gradient aggregation and control logic to allocate, de-allocate, and manage aggregators. The main challenge is that the whole packet must be parsed in a limited time budget and processed in the limited switch pipeline stages.

Aggregation. Prior work [52] processed packets in a single pipeline, which limited packets to 184B. ATP increases this limit by taking two passes (called **Two-pass**) at the switch for each packet, which is a mixed usage of the *resubmit* [2] and *recirculate* features. The details are in §D. This raises the maximum packet size to just 306B – larger, but still small packets. This leaves 4 switch stages for control operations.

Control logic. This is responsible for checking whether an aggregator is available, processing protocol flags, and updating aggregation state. To work within the restriction of one-time access to a register, ATP applies various techniques to handle otherwise complex operations. Consider the bitmap check process, which involves a read of the bitmap in the aggregator and then an arithmetic operation on the gradient value and bitmap value; finally, a write to the bitmap in the aggregator. We instead note that a write to bitmap is equivalent to setting a bit always. This allows us to reorder the write operation to just before the read operation. This read-followed-by-write serves as one time register access which is permissible. Another method in ATP to address one-time access restriction for one packet is to use two packets; e.g., ATP

[2] We leverage ' force_shift ingress' feature to drop the data part that has been aggregated before the resubmit.

allocates the aggregator with gradient packets but deallocates the associated aggregators using parameter packets.

End-Host Networking Stack. We implement ATP as a BytePS [47] plugin, which integrates in PyTorch [13], TensorFlow [12] and MXNet [17]. BytePS allows ATP's use without application modifications. ATP intercepts the *Push* and *Pull* function calls at workers as they communicate with ATP PS.

Small Packet Optimizations. ATP's network stack targets Mellanox's RAW ETHERNET userspace network programming model [1]. ATP uses **TSO** [46] and Multi-Packet QP (**MP-QP**) [38] hardware acceleration features to improve small-packet processing. TSO speeds packet sending by offloading packetization to the NIC and improves PCIe bandwidth via large DMA transfers. To improve packet receiving rate, MP-QP uses buffer descriptors that specify multiple contiguous packet buffers and reduce the NIC memory footprint by at least 512X. These features together reduce CPU cost via fewer calls to send/receive packets and fewer DMA operations to fetch packet send and receive descriptors.

ATP uses multiple threads to speed up packet processing. When ATP receives tensors to transfer, it assigns the tensor to a thread to send, which may cause load imbalance across different threads. Each worker in ATP has a centralized scheduler to receive tensors from the application layer, and maintains the total load for each thread. Whenever the scheduler receives tensors to be sent, it extracts the size and assigns the tensor to the least loaded thread to balance load.

Baseline Implementation. We implement a prototype of SwitchML [52], which uses the switch as the PS and provides a timeout-based packet-loss recovery mechanism. Our implementation uses TSO and MP-QP at end hosts to improve small-packet operations, but cannot use the two-pass optimization at the switch: it is incompatible with SwitchML's packet loss recovery mechanism. As a result, the maximum packet size for SwitchML is 180 bytes.

5 Evaluation

We evaluate ATP via testbed experiments and software emulation to answer the following questions:

1. How does ATP perform compared to state-of-the-art approaches for a single job (§5.2.1)?
2. How does conversion to integers in ATP affect time to accuracy (§5.2.2)?
3. How does ATP's inter-rack aggregation perform compared to an alternate rack-scale service (§5.2.3)?
4. What are the overheads of ATP's loss recovery mechanisms (§5.2.4)?
5. How does dynamic aggregator allocation compare to a centralized static scheme under multi-tenancy (§5.3)?
6. How effective is ATP's congestion control (§5.4)?

5.1 Experimental Setup

Cluster Setup. We evaluate ATP on a testbed with 9 machines. 8 of them have one NVIDIA GeForce RTX 2080Ti GPU with NVIDIA driver version 430.34 and CUDA 10.0. All machines have 56 cores of Intel(R) Xeon(R) Gold 5120T CPU @ 2.20GHz, 192GB RAM with Ubuntu 18.04 and Linux kernel 4.15.0-20. Each host has a Mellanox ConnectX-5 dual-port 100G NIC with Mellanox driver OFED 4.7-1.0.0.1. All the hosts are connected via a 32x100Gbps programmable switch with a Barefoot Tofino chip. We evaluate inter-rack aggregation using Tofino's software switch model [3] to emulate ATP switches in software with the same code.

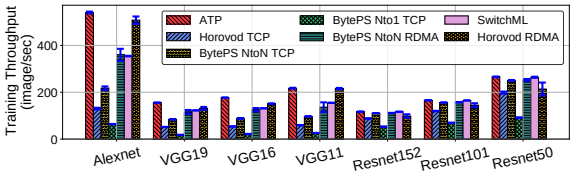


Figure 7: Single job throughput

Baselines. We compare ATP against BytePS [64]; both use a worker-PS architecture. BytePS supports TCP (BytePS + TCP) and RDMA over Converged Ethernet (RoCE) [33] (BytePS + RDMA) as network protocols. We turn on PFC [23] for RDMA to provide a lossless network. While ATP uses the default of N workers to 1 PS (labeled Nto1), we optimize BytePS with as many PSs as the workers (labeled NtoN) to alleviate the network bottleneck. Also, we co-locate one each of the N PSs and N workers in the same machine. We also compare ATP with our implementation of SwitchML, a state-of-the-art baseline with in-network aggregation support (validated by comparing to published results for training speedups). We also compare ATP against Horovod [54] with RoCE (Horovod+RDMA) and with TCP (Horovod+TCP) which uses a ring all-reduce architecture [57].

Workloads. We evaluate many popular real world models: AlexNet, VGG11, VGG16, VGG19, ResNet50, ResNet101, and ResNet152 [30, 40, 56]. Each model trains on the ImageNet dataset. The DT job has 8 workers unless specified. For most experiments we use VGG16 (model size 528MB) and ResNet50 (model size 98MB), as representatives for network-intensive and compute-intensive workloads, respectively. We also run an aggregation microbenchmark where each worker repeatedly transfers 4MB tensors (maximum size BytePS supports), which are aggregated in the network (ATP) or at the PS(s) (BytePS), and are then sent back to the workers. In contrast to real jobs, this microbenchmark has equal-sized tensors and always has data to send with no "off" phase.

Metrics. We use three metrics to measure performance: (1) *training throughput* for DT jobs, which is the number of images processed per second (*image/sec*) normalized by the number of workers; (2) *time to accuracy* to show a DT job's quality, which is the training time to reach a target or maximum accuracy; and (3) *aggregation throughput* for the microbenchmark, which is the total bytes of parameters received at each worker per second (*Gbps*).

5.2 Single Job Performance

5.2.1 ATP Training Performance

We compare ATP against all baselines on single-job training throughput for all the models in our workload as shown in Figure 7. ATP achieves the best performance for all jobs with maximum speedup of 1.5X over BytePS NtoN RDMA, 1.24X over Horovod RDMA, 2.5X over BytePS NtoN TCP, 4.2X over Horovod TCP and 1.5X over SwitchML. Performance gains are larger on network-intensive workloads (VGG) than compute-intensive workloads (ResNet). PS-based ATP is comparable to, and in many cases outperforms, the state-of-the-art all-reduce approach (Horovod+RDMA). ATP outperforms SwitchML due to support for larger packet size made possible via our optimized two-pass implementation of switch logic.

5.2.2 ATP Time-to-Accuracy (TTA)

ATP changes the nature of computation of gradient aggregation via conversion of gradient values to integers to enable aggregation on the switch and has reactive mechanisms to deal with overflows. To confirm this does not affect training quality of our workload, we evaluate ATP's accuracy over time against the baseline (BytePS NtoN RDMA). We find that ATP spends the same number of epochs to achieve the same top-5 accuracy as BytePS NtoN RDMA for all the models. Figure 8 plots the top-5 accuracy with time for ResNet50, VGG16 and ResNet152. With VGG16, a network-intensive workload, ATP outperforms BytePS and reaches 75% top-5 accuracy 1.25X faster than BytePS. With ResNet50, ATP and BytePS reach 93% top-5 accuracy in comparable amount of time (ATP is 1.02X faster) as shown in Figure 8a. ATP does not speed up training for ResNet50 as it is a compute-intensive workload. ResNet152 (Figure 8c) exhibits the same trend as ResNet50. We also conduct TTA experiments on VGG19, ResNet101 and AlexNet models, and observe that ATP reaches target accuracy 1.2X, 1.01X and 2.39X faster, respectively. Figures for these results are in §E.3.

Overflow Correction. Retransmissions due to ATP's overflow correction mechanism can be a source of overheads. In our results, we see overflows only with ResNet152, with 445 aggregated gradient packets (i.e., 0.00002% of all aggregated gradient packets generated in an epoch) requiring overflow correction. These happen only in the first few iterations of the first epoch when the model is initialized with random weights and the magnitude of gradient updates is large.

Worst-case Precision Loss. ATP uses a scaling factor of 10^8. Gradient values with magnitude less than 10^{-8} are approximated as zero and experience complete loss of precision. We observe that an average of only 0.00002% of the gradients values per epoch in ResNet50 are less than 10^{-8}. As a result, ATP causes no loss of accuracy in the final trained model.

In summary, these results demonstrate that ATP does not compromise training quality and is able to achieve baseline training accuracy in less time than other methods owing to the acceleration provided by in-network aggregation.

5.2.3 Inter-rack Aggregations

ATP provides aggregation at two levels in inter-rack configurations. We pick a typical network topology as shown in Figure 9, where the PS is connected to switch $SW2$ and workers $w0$-$w5$ are connected to different switches. We compare against a rack-scale solution (*RSS*) that aggregates locally at each rack and forwards partial aggregates to the PS. In our test topology, *RSS* performs aggregation for $w0$-$w1$ at $SW0$, $w2$-$w3$ at $SW1$, $w4$-$w5$ at $SW2$, and then, aggregation from $SW0$, $SW1$ and $SW2$ at the PS. This approach simplifies the algorithm at the switch at the cost of more network traffic to the PS. We measure the amount of traffic the PS receives with ATP and *RSS* using the software simulator (so we cannot measure real throughput). PS with *RSS* receives 3X more traffic than PS with ATP. This is because *RSS* sends the partial aggregates from $SW0 - SW1$ to PS while ATP aggregates individual packets from $w5$ and $w4$ and partial aggregates from $SW0$ and $SW1$ at switch $SW2$ before they are sent to PS; this eliminates $^2/_3$ of the traffic to the PS.

5.2.4 Packet Loss Recovery Overhead

ATP handles packet loss at the end host, but guarantees aggregation correctness and prevents memory leaks in the switch. To evaluate the overhead of packet loss recovery, we configure one worker to adversarially drop packets with a packet loss rate between 0.001% and 1% (as in prior work [52]). We compare against SwitchML, which uses a timeout mechanism to detect packet loss, and turn off ATP's congestion control to avoid window back-off due to this adversarial packet loss. We use the timeout value (1ms) from SwitchML.

Figure 10 shows the training and microbenchmark throughput normalized to no loss for varying loss rates. Overall, ATP degrades gracefully when the loss rate increases, and to a lesser degree than SwitchML. This is because ATP adopts out-of-sequence ACKs as a packet loss signal, which enables ATP to detect and respond to packet losses faster than SwitchML.

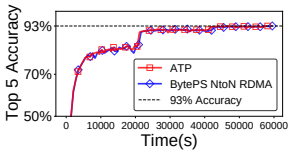
5.3 Multiple Jobs

ATP does dynamic best-effort sharing of switch resources across multiple jobs. Our multi-tenant extension of SwitchML statically partitions switch resources equally across jobs.

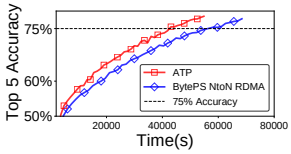
5.3.1 Dynamic vs. Static Sharing

We compare ATP's dynamic best-effort approach against SwitchML's static approach by launching 3 identical VGG16 jobs (with identical placement for workers and PS) connected to one programmable switch. We vary the number of aggregators for the 3 jobs on the switch. The static approach allocates a fixed $1/3$ fraction of the aggregators to each job, while ATP's dynamic approach shares these aggregators dynamically, so that when any job is in an off phase its aggregators are available to other jobs.

We first tune the number of aggregators reserved for the 3 jobs to find the minimum number needed to get maximal training throughput for each job with the static approach. We

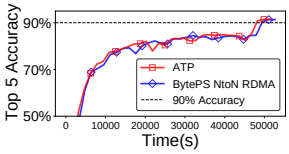


(a) ResNet50



(b) VGG16

Figure 8: Time to accuracy



(c) ResNet152

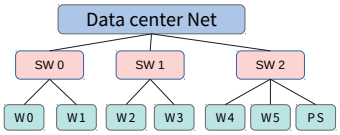
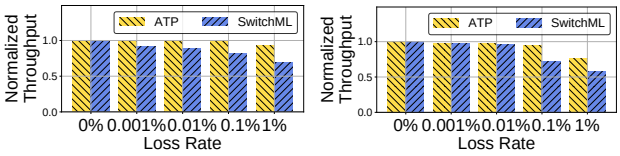


Figure 9: Inter-rack topology



(a) VGG16 (b) Microbenchmark

Figure 10: Throughput in different packet loss rate.

find that 1980 aggregators, referred to as *peak throughput aggregators* (PTA), equally divided across the three jobs maximizes throughput because it saturates the link from the switch to the PS. To measure the impact of sharing strategy under contention, we reduce the number of available aggregators from 100% of PTA to 33% of PTA, and measure training throughput with both static and dynamic approaches.

Figure 11a shows the average training throughput (measured after warm-up, from the second epoch of training onwards) as we vary the number of aggregators available to the 3 jobs from 100% of PTA (1980 aggregators) to 33% of PTA (660 aggregators). With 100%, the dynamic approach performs similarly to static approach. This is because in both cases all aggregation happens in-network, and the switch to PS link is fully utilized. As we reduce the number of aggregators available, throughput for the dynamic approach degrades less than with the static approach. This is because the dynamic approach allows sharing of unused aggregators across jobs and also uses available switch-to-PS link capacity for PS-based aggregation. We show this by delving into ATP's dynamic approach when using 100% and 33% of PTA.

Figure 11b shows the total number of aggregations at the PS every 10ms for each job with ATP's dynamic approach using 100% of PTA, starting at job warm-up. Initially, there are hash collisions because of jobs allocating the same aggregator index, which leads to aggregation at the PS. After 500ms, ATP's hash-collision avoidance kicks in and the dynamic approach converges to complete in-network aggregation.

Figure 11c shows how ATP's dynamic approach responds when jobs enter an off phase. The figure shows the number of aggregations performed at the switch every 10ms when only 33% of PTA are available. The trace shows that when one job enters an off phase (zero in-network aggregations), the number of in-network aggregations for the remaining jobs increases. For example, at sample 120, job 3 (red) enters an off phase, and the in-network aggregations for jobs 1 and 2 increase until job 3 resumes at sample 180.

5.3.2 Effectiveness of ATP's Hashing Scheme

In §5.3.1, we show ATP's hashing scheme works with 100% of PTA for 2 jobs. To evaluate at larger scale, we launch 8 microbenchmark jobs with 100% of PTA with ATP's hashing enabled (**Hash-based dynamic**) and without hashing enabled (**Linear-based dynamic** which allocates aggregator indexes in sequence, i.e., index = seq_num % N), and compare against the baseline scheme (static allocation with 100% of PTA).

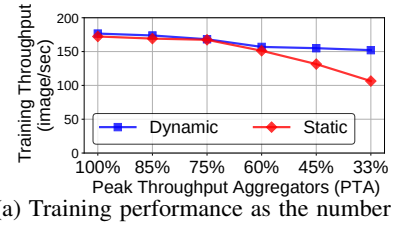
Figure 12 plots the aggregation throughput for the three schemes as number of jobs increases. ATP's hash-based scheme matches the baseline static scheme and greatly outperforms the linear-based scheme, as without hashing, a continuous sequence of gradients from multiple jobs collide, introducing a significant amount of retransmission. Overall, these experiments indicate that the dynamic hash function can effectively distribute aggregators to each job. It can achieve comparable performance to the static approach when there is no contention for switch aggregators, and outperform the static approach under contention.

5.4 Effectiveness of Congestion Control

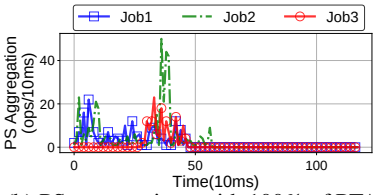
ATP's congestion control mechanism aims to minimize packet loss while maximizing the link utilization. Network congestion happens (1) ATP traffic is co-located with bandwidth-hungry normal traffic, such as TCP or RDMA transfers; (2) when ATP does not perform in-network aggregation due to a shortage of switch resources from other contending ATP jobs, causing an incast to the PS. We evaluate ATP's congestion control mechanism in both cases.

With non-ATP traffic. It is common to co-locate multiple systems and applications in a multi-tenant multi-rack cluster, such as storage and data pre-processing systems. We validate ATP's congestion control effectiveness when co-locating with such traffic. We launch a training job with 6 workers and 1 PS for ATP. We add background flows competing for bandwidth on a link to a worker, which individually can achieve line rate. The experiment starts with background traffic, and then starts a training job at $t = 25$s. We stop the training job after 50s. We perform this experiment both with VGG16 and with our microbenchmark to emulate large models on ATP.

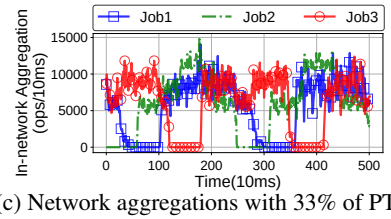
Figure 14a reports the aggregation goodput averaged per second from the worker that experiences network congestion for VGG16 (ATP), the goodput from the non-ATP traffic (non-ATP), and the aggregate goodput (ATP + non-ATP) over time. The dashed black line shows the peak goodput VGG16 job can achieve without background traffic; this link bandwidth demand is less than fair share. We see that the VGG16 job



(a) Training performance as the number of PTA changes.



(b) PS aggregations with 100% of PTA



(c) Network aggregations with 33% of PTA

Figure 11: Dynamic v.s. static allocation with different peak throughput aggregators (PTA) ratio.

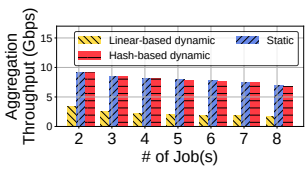


Figure 12: Large scales.

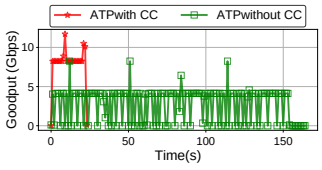
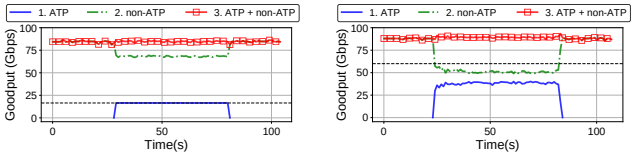


Figure 13: ATP job with and without congestion control



(a) ATP: VGG16 (b) ATP microbenchmark

Figure 14: Aggregation goodput with non-ATP traffic.

with ATP is able to achieve peak goodput (as demand is less than fair share) and that the sum of goodputs is close to line rate on the uplink from this worker. This indicates that ATP's CC is able to co-exist with non-ATP background traffic with max-min fair allocation in this setting.

Figure 14b plots the same scenario instead with the microbenchmark job. The dashed black line shows the peak goodput this job can achieve without background traffic, which is indicative of a link bandwidth demand greater than fair share of the network uplink rate at the worker. We see that ATP is able to achieve near the fair share of the bottleneck link (the uplink from this worker) and that the sum of goodput from both traffic types is close to link rate. This showcases near equitable sharing of link bandwidth. Figure 14a and Figure 14b show that the congestion control of ATP is able to respond quickly to changes in congestion, and converge to a new bandwidth which is very stable.

With other ATP traffic. We launch a single VGG16 job (8 workers) using ATP for 10 iterations. We reserve only 50% of the switch aggregators needed to achieve peak training throughput with complete in-network aggregation (to emulate contention from background ATP jobs). Figure 13 shows the aggregation goodput (total size of parameters aggregated in a second) from a single worker against time with and without congestion control (CC). We see that the congestion control of ATP kicks in, and the aggregation goodput stabilizes around 7.5Gbps for the worker. Note that this goodput is lower than the peak goodput (∼20Gbps) from the previous experiment because here we have an 8->1 (8 workers to 1 PS) incast, only 50% of traffic is reduced in the network, and the PS in our implementation saturates at 60Gbps (§E.1).

Without CC, the goodput fluctuates dramatically between 4Gbps and 0; this is because incast causes frequent packet loss without congestion control, and introduces high packet loss recovery overhead. The training goodput of ATP with congestion control is 66.8 img/s while that without congestion control is 23.2 img/s, a decrease of 65%. These results show that the congestion control for ATP can effectively maintain high goodput and is effective in avoiding packets loss.

In summary, ATP's congestion control helps it co-exist with other tenants (both ATP and non-ATP).

6 Other Related Work

We discuss prior works that propose advances in hardware, software, offloads, and algorithms to accelerate DNN training.

Speedup Network Transmission. Prior efforts propose to improve gradient aggregation time by (1) smarter network scheduling – increasing the overlap between GPU/CPU computation and network transmission via fine-grained tensors transmission scheduling (per layer instead of the whole gradient or parameter) [29, 34, 48, 65]; combining model- and data-parallelism via pipelining [28, 32]; using asynchronous IO [18, 22]. (2) reducing network traffic – using large batch size to reduce the communication frequency [9, 26, 35]; using quantization [53] or reducing redundancy in SGD [41] to reduce bytes sent over the network; optimizing mixture of local-global aggregation to adapt to network change at runtime [19, 59]. ATP can incorporate these optimization to further improve its performance.

In-network Aggregation. The idea of in-network aggregation has been explored in wireless networks [16, 55]; in big-data systems and distributed training systems using end-hosts [21], a specialized host [44], high performance middleboxes [45] or overlay networks [15,58]. DAIET [51] proposed a simple proof-of-concept design of in-network aggregation without a deployment. ShArP [27], supported by special Mellanox Infiniband switches, builds an overlay reduction tree to aggregate data going through it, but it does not apply the aggregation until the switch receives all the data. ATP is the first to provide a *dynamic, best-effort* in-network aggregation service for multi-tenant multi-switch clusters.

7 Conclusion

We build an in-network aggregation service, ATP, to accelerate DT jobs in multi-tenant multi-rack networks. ATP provides a *best effort in-network aggregation* primitive via careful co-design of switch logic (for aggregation) and the end-host networking stack (for reliability and congestion control). Testbed experiments show that ATP is able to outperform existing systems by up to 8.7*X* for a single job, and it is even slightly better than the current state-of-the-art ring all-reduce with RDMA. In a multi-tenant scenario, best-effort in-network aggregation with ATP enables efficient switch resource usage, and outperforms current state-of-the-art static allocation techniques by up to 38% in terms of training time when there is heavy contention for on-switch resources.

References

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

[2] Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[3] Barefoot Tofino Software Behavior Model. https://www.barefootnetworks.com/products/brief-p4-studio/.

[4] Google TPU. https://cloud.google.com/tpu/.

[5] IEEE 754-1985. https://en.wikipedia.org/wiki/IEEE_754-1985.

[6] Intel FlexPipe. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[7] LiquidIO Server Adapters. http://www.cavium.com/LiquidIO_Server_Adapters.html.

[8] Netronome NFP-6000 Intelligent Ethernet Controller Family. https://www.netronome.com/media/redactor_files/PB_NFP-6000.pdf.

[9] Now anyone can train imagenet in 18 minutes. https://www.fast.ai/2018/08/10/fastai-diu-imagenet/.

[10] Nvidia clocks world's fastest bert training time and largest transformer based model, paving path for advanced conversational ai. https://devblogs.nvidia.com/training-bert-with-gpus/.

[11] XPliant Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.

[13] Paszke Adam, Gross Sam, Chintala Soumith, Chanan Gregory, Yang Edward, DeVito Zachary, Lin Zeming, Desmaison Alban, Antiga Luca, and Lerer Adam. Automatic differentiation in pytorch. In *In NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques*.

[14] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 503–514, New York, NY, USA, 2014. Association for Computing Machinery.

[15] D. C. Arnold and B. P. Miller. Scalable failure recovery for high-performance data aggregation. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[16] Raghav Bhaskar, Ragesh Jaiswal, and Sidharth Telang. Congestion lower bounds for secure in-network aggregation. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, 2012.

[17] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[18] 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, Broomfield, CO, October 2014. USENIX Association.

[19] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1–1, 2019.

[20] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. Appswitch: Application-layer load balancing within a software switch. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 64–70, 2017.

[21] Paolo Costa, Austin Donnelly, Antony Rowstron, and O'Shea Greg. Camdoop: Exploiting in-network aggregation for big data applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 29–42, San Jose, CA, 2012. USENIX.

[22] 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.

[23] Claudio DeSanti. IEEE 802.1: 802.1Qbb - Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html, 2009.

[24] 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.

[25] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20.

[26] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[27] 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 *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10. IEEE, 2016.

[28] 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.

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

[30] 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.

[31] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proc. ACM SIGCOMM*, 2015.

[32] 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.

[33] 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.

[34] 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.

[35] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[36] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, 2018.

[37] 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.

[38] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019.

[39] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–81, 2016.

[40] 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.

[41] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[42] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[43] 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, Boston, MA, February 2019. USENIX Association.

[44] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[45] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L Wolf. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 249–262, 2014.

[46] Mellanox Technologies. Mellanox OFED for Linux User Manual. Rev 3.40. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v3.40.pdf.

[47] 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.

[48] 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.

[49] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[50] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the ai accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 20–25, 2018.

[51] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, 2017.

[52] 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.

[53] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, 2014.

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

[55] Mohamed A. Sharaf, Jonathan Beaver, Alexandros Labrinidis, and Panos K. Chrysanthis. Tina: A scheme for temporal coherency-aware in-network aggregation. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDe '03, 2003.

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

[57] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[58] Raajay Viswanathan and Aditya Akella. Network-accelerated distributed machine learning using mlfabric. *arXiv preprint arXiv:1907.00434*, 2019.

[59] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd. *arXiv preprint arXiv:1810.08313*, 2018.

[60] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record*, 45(2):17–22, 2016.

[61] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[62] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[63] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 645–661, Carlsbad, CA, October 2018. USENIX Association.

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

[65] 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, Santa Clara, CA, July 2017. USENIX Association.

[66] 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.

A Additional Design Details

This section is a supplement to ATP's design description from §3. We first provide additional design details for clearer exposition and then outline design extensions to make ATP more general and robust.

A.1 Switch Memory Layout

Figure 15 shows switch memory layout for ATP aggregators. The descriptions for individual fields are provided in §3.3.

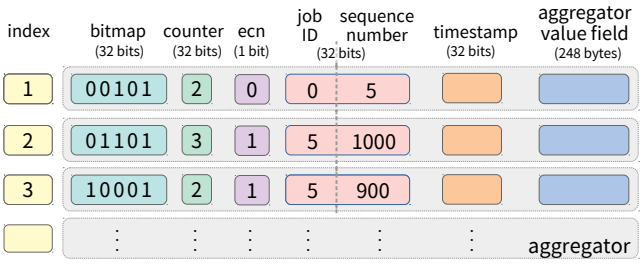


Figure 15: ATP switch memory layout.

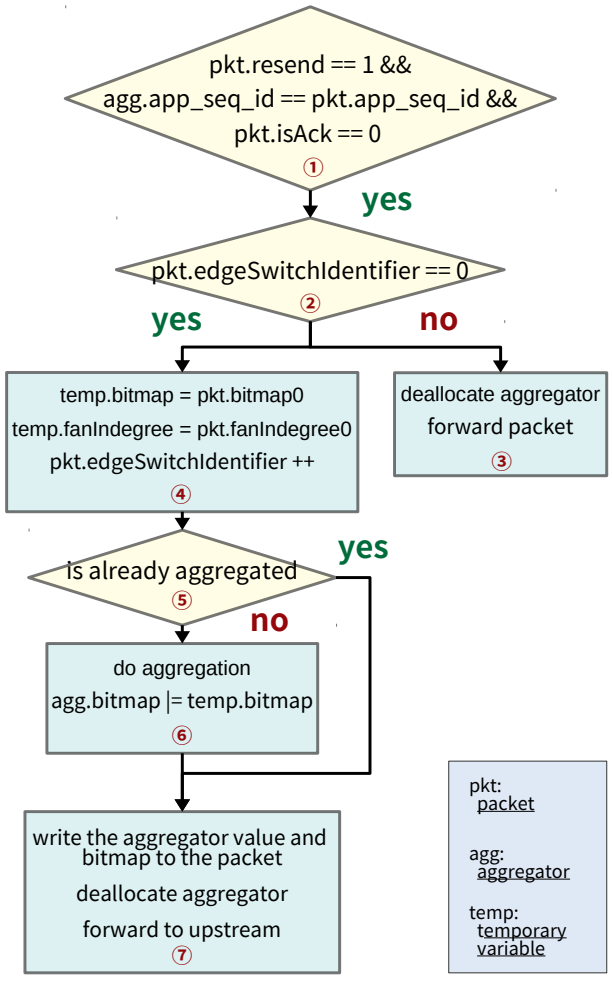


Figure 16: Pseudocode of the switch logic dealing with packet loss.

A.2 Switch Algorithm for Reliability

Figure 16 explains the details on how the switch handles the three issues discussed in Section 3.7. If the retransmitted gradient fragment packets have an aggregator at switch (① in Figure 16), ATP uses the bitmap in the gradient fragment packet header to check if bitmap field in the aggregator is set (⑤). If unset, the data field element is aggregated to the value field of the aggregator and the bitmap field in the aggregator is set (⑥). After aggregation or if the retransmitted gradient has already been seen, ATP copies the value field and the bitmap

field from the aggregator to the data field and the bitmap field of the packet, deallocates the aggregator, and sends the packet upstream (⑦). Given that only the first retransmitted gradient packet triggers aggregator deallocation, the retransmitted packets with the same sequence number from other workers do not hit the aggregators or reserve a new aggregator and thus, they will be directly forwarded to upstream devices. Note that at the second level of aggregation, ATP directly forwards a retransmitted gradient packet for simplicity (③). In all the cases, ATP deallocates the aggregator and sets all fields to null (③ and ⑦).

The parameter packet, whether it is retransmitted or not, always triggers the deallocation of the aggregator with the same <Job ID, Sequence Number>.

A.3 Aggregation at Multiple Hierarchy Levels

ATP can be extended to support aggregation at multiple hierarchy levels, i.e., beyond ToR switches to also enable aggregation at aggregation layer switches and core layer switches. The only pre-requisite is that packet routes for all gradient packets have to be deterministic (e.g., as in ECMP) and known ahead of time so that ATP knows the exact switch in the network at which gradient packets (or partially aggregated gradient packets) for a particular sequence number converge. This helps to precisely determine the fan-in degree at each intermediate switch from workers to the PS. This helps higher-level switches to aggregate partially aggregated results from lower-level switches and determine when aggregation at higher-level switches is complete. To enable multi-level aggregation, we need to add more `bitmap` and `fanInDegree` fields to ATP packet header (one for each additional in-network gradient aggregating switch) to track progress of aggregation at all the switches involved in gradient aggregation enroute from the workers to the PS.

Notably, for non-deterministic route load balancing schemes, such as Presto [31] and CONGA [14], ATP packets only deterministically route through TOR switches with non-deterministic routing at higher levels of the datacenter network hierarchy. Thus, to support such cases, ATP does aggregation only at the TOR-layer.

A.4 Recovering from Worker Failures

PS-based architectures deal with worker failures by respawning a worker process, perhaps on a different machine connected to a different ToR switch. This requires invalidating stale aggregators having incomplete aggregation results that are waiting on gradient packets from the failed worker. Such stale entries are invalidated as ATP has checks and balances to overcome switch memory leaks (described in §3.7). Also, the re-spawned worker might be on a different machine and connected to a different TOR switch. This might require changes to the `fanInDegree` field in ATP packet headers. ATP re-triggers dynamic job-setup phase (described in §3.2) after any such failure event and the PS re-initiates gradient aggregation by sending an ACK on the new multicast tree that includes the re-spawned worker for the earliest sequence number that is yet to be aggregated.

A.5 Dealing with Stragglers

A slow worker or a slow link can slow down the whole training process. However, this is an artifact of synchronous training and not an issue with ATP. Note that because ATP alleviates network bottlenecks as it aggregates gradients in the network, it reduces the likelihood of network-induced stragglers.

A.6 Comparison to Ring All-reduce Approach

Ring all-reduce uses one-to-one communication in each training iteration/round. With this communication pattern, ATP does not have any opportunity to kick in. A drawback of ring all-reduce is that the amount of data that each worker sends and receives is higher and is $4(n-1)\frac{|U|}{n}$, where n is the number of workers, $|U|$ is the total number of bytes to be aggregated. With ATP the amount of data that each worker sends and receives is $2|U|$; the amount of data that each PS sends and receives is $2|U|/m$ where m is the number of PSes. Quantitatively, this indicates that ring all-reduce generates more network traffic than ATP, which may congest the network for other running applications in the cluster. We compare ATP against the ring all-reduce approach (**Horovod RDMA** and **Horovod TCP**) in Figure 7, demonstrating that ATP is better than ring all-reduce for training popular models.

B Addtional details on dealing with floating point

ATP converts floating point values to integers by multiplying with a scaling factor to enable gradient aggregation on programmable switches.

B.1 ATP's Choice of Scaling Factor

The choice of the scaling factor is crucial. A smaller scaling factor can lead to rounding-off a lot of digits after the decimal point and a greater loss in precision. Theoretically, the precision loss due to this conversion is bounded by $\frac{n}{s}$ (Theorem 1 in [52]), where n is the number of workers and s is the scaling factor. A large scaling factor can lead to aggregation of large integers and may cause overflow at the switches. Theoretically, there is no overflow if gradient values are less than an upper bound, $B = \frac{2^{31}-n}{ns}$ (Theorem 2 in [52]).

Prior work [52] relies on empirical measurements to find this upper bound B for gradient values of a popular suite of ML training jobs and chooses a scaling factor $s = \frac{2^{31}-n}{nB}$. The precision loss with this choice of scaling factor is bounded by $\frac{n^2B}{2^{31}-n}$.

There are two drawbacks with this approach. First, the guarantee of no overflows with this approach strictly relies on obtaining an accurate estimate of the upper bound B on gradient values. This makes the approach expensive to accom-

Job	False Positive Packets
ResNet50 - 8 Workers	37,002
ResNet50 - 16 Workers	231,528
ResNet50 - 32 Workers	1,156,126
ResNet50 - 64 Workers	14,602,998

Table 1: Average false positive packets per epoch increase as workers increase.

modate jobs that train new and unseen models as empirically determining the value of B for a new model requires an end-to-end training run. Second, the scaling factor decreases as the value of B and the number of workers n increas, which leads to increasing loss of precision. With 100 workers and $B = 200$, the value of the scaling factor is $\sim 10^5$ and the maximum loss of precision is $\sim 10^{-5}$ per gradient value. We sample gradient values below 10^{-5} in some epochs across models and find that there are 18% such gradient values for ResNet50, 42.8% for VGG16 and 39.8% for AlexNet. These values will experience completed loss of precision hurting model accuracy upon training.

With ATP, we choose a high scaling factor so as to minimize loss of precision. We note that 32-bit floating point representation provides the precision of 7 decimal digits [5]. Thus, to provide an equivalent precision, ATP chooses the scaling factor of 10^8. The decoupling of scaling factor from B completely eliminates the first drawback and partially eliminates the second drawback described above. However, a large scaling factor can lead to overflow of aggregated gradients.

There are broadly two mechanisms to overcome overflows: proactive and reactive. ATP chooses a reactive mechanism (§3.8) because there are drawbacks to using a proactive mechanism that we discuss next.

B.2 Pitfalls of Proactive Overflow Mechanism

The proactive mechanism prevents overflows from ever occurring at the switch. It determines a maximum gradient value threshold B such that, as long as gradient values aggregated in the switch are less than B, any overflow is avoided. The value of B ($= \frac{2^{31}-n}{ns}$) is computed at each worker during job initialization. Packets with gradient values $\leq B$ are aggregated in the switch, while workers flag packets with gradient values $> B$. Packets with this flag are not aggregated in the switch and are eventually aggregated at the PS. It is worth noting that this mechanism naturally fits with the best-effort aggregation service provided by ATP and cannot be implemented in the prior SwitchML work.

Unfortunately, a proactive mechanism severely limits the opportunities for in-network aggregation. With 100 workers and a scaling factor of 10^8, the value of B is 0.2. To highlight this, in Table 1, we measure the number of false positives in a single epoch when training a ResNet50 model. We classify a gradient packet with a particular sequence number as a false positive if the gradient value in that packet is $> B$ but the aggregated gradient value for that sequence number from all the workers does not overflow integer bounds (2^{31}). Each false positive packet could have been aggregated in the switch and consumed in the network, but instead with the proactive approach ends up traversing the network from the worker all the way to the PS. As seen in Table 1, the number of false positive packets gets worse as we increase the number of workers in a ResNet50 job. This trend applies to all models, although the magnitude of false positive packets might change. This is not desirable because a higher number of false positive packets means that we lose out on in-network aggregation opportunities and that the number of packets traversing the link to the PS increases. Thus, increasing false positive packets leads to increased likelihood of incast which only snowballs as the number of workers in the job increases.

For these reasons, in ATP, we avoid using a proactive mechanism and use a reactive mechanim for overflow correction (§3.8).

C Dynamic Scaling Factor

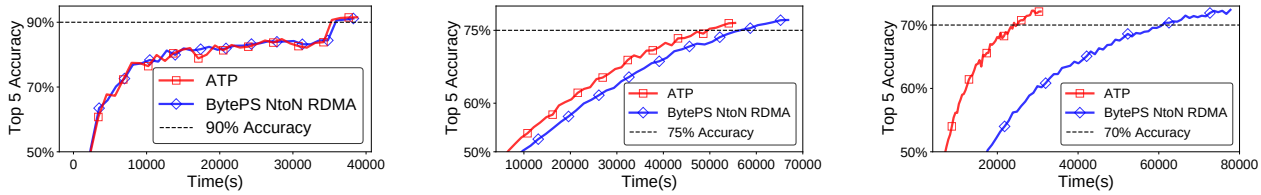
A static scaling factor can, in the worst case, lead to a very high overflow correction frequency when the magnitude of gradient values from all workers is large. In such a case, reducing the value of the scaling factor will reduce the overflow correction frequency and will also not lead to a loss of precision (as the gradient values are large). Thus, a possible optimization would be to do dynamic adjustments to the scaling factor in reaction to the current range of gradient values in recent iterations, especially for the case when the number of workers n is large. In our evaluations with popular models and the scale at which these distributed models are trained today, we do not see the need for such dynamic scaling. However, dynamic scaling adjustments may be beneficial in the future as new large models emerge and training jobs scale-out even further.

D Additional Implementation Details

This section supplements the description of ATP's implementation in §4.

To increase the packet size processed by the switch, ATP programs the switch to process each ATP packet twice, which we refer to as **two-pass**. The first half of the gradients in the packet are aggregated in the first pass and the second half are aggregated in the second pass. Instead of using two end-to-end (all the way from ingress on a port to egress to a port) pipelines, which requires two ports to recirculate the packets, ATP uses the *resubmit* and *recirculate* features together, which allows to re-process the same packet (except the last packet that completes gradient aggregation) only in the ingress pipeline of the switch. This avoids using an additional port at the egress pipeline.

Recall that ATP drops the first $n-1$ packets (where, n is number of workers) after aggregation at the switch, and



(a) ResNet101 (b) VGG19 (c) AlexNet

Figure 17: Time to accuracy

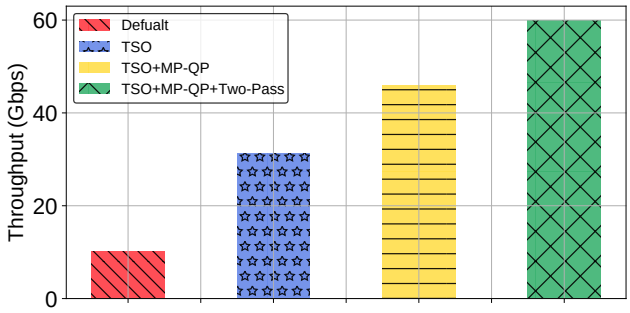


Figure 18: Throughput with different hardware acceleration.

writes the aggregated results from the aggregator to the n-th worker's gradient packet, where n is the number of workers. ATP uses *resubmit* for the first $n-1$ gradient packets after the first pass. As the name suggests, this takes the packet from the end of the ingress pipeline after the first processing pass, left shifts the packet using the 'force_shift ingress' feature on the parser to drop the data part that has been aggregated in the first pass, and immediately resubmits this packet to the ingress pipeline for a second pass to process the second half of the packet. However, we can not apply the resubmit feature to the last gradient packet because ATP writes the first half of the aggregated results to the last packet during the first pass and a resubmit with left shift will lead to a loss of this aggregated result. To deal with this, ATP avoids using the resubmit feature for the last packet. Instead, we use 'recirculate' to enable a second end-to-end pass for the second half of the last packet. This approach requires only one additional port to recirculate the last packet. If the next generation of programmable switches is able to process larger packet sizes or the NIC supports higher packet processing rate, ATP will not require a two pass implementation at the switch.

E Additional Evaluation Details

This section provides additional evaluation results (§E.1, §E.3) and supplements some existing results (§E.2).

E.1 Small Packets Optimizations

ATP applies multiple hardware acceleration techniques to boost throughput with small-packets as mentioned in Section 4. We demonstrate the effectiveness of each optimization by measuring the throughput of the microbenchmark using configurations that incrementally add more optimizations. We launch two hosts attached to a single switch, one as a worker, and the other as PS. the switch logic of ATP, and then are forwarded to the PS. The PS copies the received data from the receiver memory region to the sender memory region and then sends the data back to the worker.

Figure 18 shows the network throughput gains as we progressively add optimizations. TSO provides the biggest benefit, and increases performance by $3X$. Incorporating MP-QP (§4) at the end hosts further increases performance $1.47X$. Finally, adding aggregation based on two-pass implementation at the switches increases the throughput a final $1.3X$. Overall, ATP's **TSO+MP-QP+Two-Pass** optimizations provide a $6X$ throughput improvement over no hardware acceleration (**Default**). We also notice that throughput does not double between **TSO+MP-QP** and **TSO+MP-QP+Two-Pass** when packet size doubles. This occurs because throughput is bottlenecked by the memory copy at the PS and workers [3].

In summary, ATP is able to achieve high throughput with small packets by utilizing recent advances in hardware acceleration.

E.2 Switch Resource Sharing (Supplemental)

In this section, we study ATP's dynamic aggregator allocation approach and observe the impact of the number of workers and the model size on the performance of contending jobs. We launch two DT jobs on ATP under various settings.

First, we launch two VGG16 jobs each with 3 workers. Figure 19 shows the training throughput normalized against the job in isolation. We see equal throughput, across identical jobs (same model and number of workers), indicating equal sharing of switch resources. Also, the throughput reduction for each of the two jobs is only 10%.

Second, we launch two VGG16 jobs with 5 and 2 workers each. The job with more workers performs slightly worse than that with fewer workers (145.6 image/s for 5 v.s. 159.3 image/s for 2 workers).

Third, we launch a VGG16 and a ResNet50 job and observe only 5% and 10% throughput reduction respectively; reduction for the large model (5% in VGG16) is less than that for the small model (10% in ResNet50).

In summary, ATP's switch resource sharing is equitable for similar workloads. In case of different workloads it tends to favor jobs using fewer workers and larger models.

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

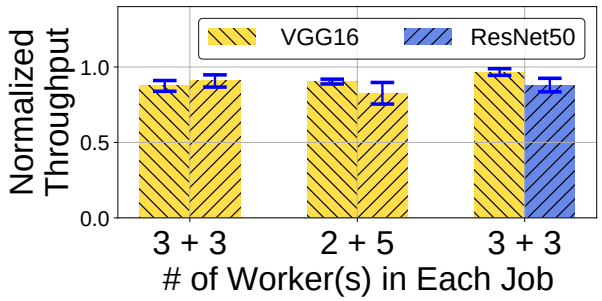


Figure 19: Multiple jobs.

E.3 Time To Accuracy (Supplemental)

Figure 17 illustrates the TTA training curve for ResNet101, VGG19 and AlexNet with the same setting from §5.2.2. ResNet101 (Figure 17a) does not see a noticeable speed up (1.01X in ATP) as it is a compute-intensive model similar to our measurements in §5.2.2 (ResNet50 and ResNet152). The results for VGG19 (Figure 17b) and AlexNet (Figure 17c) reflect 1.2X and 2.39X speed up, which is consistent with the training throughput presented in §5.2.1 for communication-intensive models.