In-Network Aggregation with Transport Transparency for Distributed Training

Shuo Liu[*]
Huawei
China

Qiaoling Wang[*]
Huawei
China

Junyi Zhang
Huawei
China

Wenfei Wu[†]
Peking University
China

Qinliang Lin
Huawei
China

Yao Liu
Sun Yat-sen University
China

Meng Xu
Huawei
China

Marco Canini
KAUST
Saudi Arabia

Ray C. C. Cheung
City University of Hong Kong
China

Jianfei He
City University of Hong Kong
China

ABSTRACT

Recent In-Network Aggregation (INA) solutions offload the all-reduce operation onto network switches to accelerate and scale distributed training (DT). On end hosts, these solutions build custom network stacks to replace the transport layer. The INA-oriented network stack cannot take advantage of the state-of-the-art performant transport layer implementation, and also causes complexity in system development and operation.

We design a *transport-transparent* INA primitive named NetReduce for modern multi-rack data centers. NetReduce runs beneath the transport layer. The switch performs aggregation operations but preserves data transmission connections. The host uses RoCE as its transport layer to deliver gradient messages and receive aggregation results. NetReduce achieves performance gains from both INA and RoCE: linear scalability, traffic reduction, and bandwidth freeing-up from INA — high throughput, low latency, and low CPU overhead from RoCE. For jobs spanning several multi-GPU machines, we also devise parallel all-reduce based on NetReduce to make use of intra-machine and inter-machine bandwidth efficiently. We prototype NetReduce on an FPGA board attached to an Ethernet switch. We compare NetReduce with existing programmable switch-based solutions and justify the FPGA-based design choice. We evaluate NetReduce's performance by training typical Deep Neural Network models on single-GPU and multi-GPU testbeds. NetReduce inter-operates with the existing Ethernet transport layer, is training-framework friendly, accelerates network-intensive DT jobs effectively (e.g., 70% for AlexNet), reduces CPU overheads (e.g., only one core for transmission), and is cost-effective (e.g., only 2.40% more capital expense and 0.68% more power consumption making 12.3-57.9% more performance acceleration).

CCS CONCEPTS

• **Networks → In-network processing**.

KEYWORDS

In-Network Aggregation, FPGA, RDMA, Distributed Training.

ACM Reference Format:
Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray C. C. Cheung, and Jianfei He. 2023. In-Network Aggregation with Transport Transparency for Distributed Training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582037

1 INTRODUCTION

Recently, a class of *In-Network Aggregation (INA)* solutions is proposed to promote distributed training (DT) [8, 14, 17, 35, 38, 58]. INA solutions perform the *all-reduce* operation among DT workers; these solutions offload the gradient aggregation onto programmable switches, which accelerates network transmission, scales out training jobs, and frees network bandwidth. As INA (as a case of In-Network Computation, INC) "blurs the division between computation and networking" [3], INA is typically designed as a custom transport protocol [64]. Existing practices and prototypes implement INA as a clean-slate network stack (called INA stack). The INA stack needs to combine both functions of the traditional transport layer (i.e., (de)packetization, connection management, flow control, reliability, and congestion control) and of the application (i.e., tensor aggregation, fallback, floating-point quantization).

[*]Co-primary Author
[†]Corresponding Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582037

The tight *coupling of INA and the transport layer* has shortcomings in performance, development, and operation, respectively. First, the INA stack cannot benefit from the technical progress in the transport layer. Modern data centers employ various acceleration techniques on the transport layer, e.g., Segmentation Offload [44]. And especially, RDMA over Converged Ethernet — RoCE, which is a message transport protocol implemented on hardware NICs, has been gradually deployed at a large scale due to its good properties of low latency, low CPU overhead, and high throughput [18]. But the mismatched abstractions (INA's hierarchical data flow v.s. end-to-end transport connections) and the cost of re-implementing INA's transport functions prevent existing INA solutions from taking advantage of these new features and acquiring the consequent performance gains.

Second, the INA stack causes development complexity to Machine Learning (ML) system developers. The system developer needs to spare two kinds of efforts to build an INA DT system. (1) Traditional transport layer has matured interfaces to applications, e.g., TCP socket or RDMA verbs, and they have been widely applied in existing cloud applications [18, 22, 30, 31, 76] including distributed training [15, 25, 37, 57, 72, 73]. But switching to new INA primitives requires a learning curve, which could not be within the system developers' willingness and/or expertise. (2) The developer also needs to rebuild the basic transport layer functions (i.e., flow control, reliability, and congestion control); although these functions are already baked-in in existing transport protocols, they have to be re-invented in INA.

Third, the INA stack complicates network management. (1) Deploying the INA stack on a shared cluster requires either dedicating or virtualizing physical NICs (e.g., using SR-IOV [11, 12, 19, 34]) for INA-specific traffic. The deployment also crosses administrative boundaries [3], requiring longer business cycles to coordinate ML system developers and cluster operators. (2) In a shared cluster, the network also has to apply QoS rules to isolate INA traffic and other transport traffic variants; otherwise, bandwidth contention between them could lead to unexpected behaviors. For example, ECN-based INA (e.g., ATP [35]) would be more conservative or even starved when competing with non-ECN transport protocols.

Designing a transport-transparent INA is challenging. First, INA merges gradient data flows in an aggregation hierarchy, which violates the end-to-end byte stream abstraction of the transport connection. Our intuition is to preserve transport connections by creating a "man-in-the-middle" *data flow splice* in the switch. Second, the transport layer segments messages into packets without providing the flexibility to insert an INA header, but the INA header is critical to direct the aggregation behavior on switches. Our intuition is to track the mapping between the INA header and the connection in a *connection lookup table*, and populate the table along with data transmission (§2.3).

We design an in-Network All-Reduce primitive named NetReduce to achieve *transport transparent INA* for DT jobs. (1) On its northbound, NetReduce interfaces DT jobs with a *ring* abstraction for gradient aggregation. (2) Workers on the ring establish connections to neighbors. We use RoCEv2 for message transport, but do not exclude other transport protocols. (3) NetReduce designs an on-switch accelerator to perform connection-preserving INA. (4) For jobs spanning multi-GPUs on multi-machines, we also design a method to set up parallel NetReduce rings to make full usage of inter- and intra- machine bandwidth. In addition to transport transparency, NetReduce also supports other features as in existing INA solutions, such as floating-point calculation and hierarchical extension [8, 14, 17, 35, 38, 58, 74].

We prototype the NetReduce accelerator on an FPGA board attached to switches. We justify the FPGA-based design choice instead of P4 programmable switches due to the switch's hardware limitation to achieve full transport functionality, line rate, and low overhead together (§6). We evaluate NetReduce by training typical Deep Neural Network (DNN) models in ImageNet [33] classification and Transformer models [10] in NLP. NetReduce provides INA with compatibility with the RoCE network and it provides full bandwidth saturation (93%) in 100 Gbps networks with low RTT latency (6.3 μs) and low CPU overhead (only one CPU core is occupied). NetReduce can accelerate typical DT jobs by 5% - 45%, especially in the multi-GPU multi-machine scenario. NetReduce outperforms Flat Ring AllReduce and Tencent AllReduce [26] significantly (15.1% - 68.8% and 12.3% - 57.9%).

We make the following contributions.

- A *transport transparent* INA primitive NetReduce, achieving INA's communication acceleration and RoCE's high throughput, low latency, and low CPU overhead.
- A parallel all-reduce method based on NetReduce for training on multi-GPU multi-machines, where the inter- and intra- machine bandwidth is more efficiently utilized.
- A system prototype and extensive evaluation to demonstrate NetReduce's feasibility, performance gain, compatibility, and low overhead.

2 BACKGROUND AND MOTIVATION

INA could significantly accelerate DT jobs but is not compatible with the state-of-the-art performant transport layer, which introduces extra system complexity in deployment. In production networks, operators can hardly trade the system complexity for INA's performance gain.

2.1 Training with In-Network Aggregation

All-Reduce in DT. In ML model training, the algorithm takes iterations to compute a gradient and update the model. For a data-parallel DT job, in each iteration, multiple workers compute their gradients, aggregate the gradients, and use the aggregation result to update the model. There are two main architectures to support gradient aggregation.

In a Parameter Server (PS) architecture, one or several dedicated physical servers are assigned for gradient aggregation. Workers send gradients to the PS(s), and the PS(s) compute the aggregation result and send the result back to the workers. Each worker needs to establish one *connection* with the PS (Figure 1A). In a Ring AllReduce (RAR) architecture, workers form rings to aggregate gradients. A gradient message travels along the ring for two rounds: in the first round, each worker accumulates its gradient on the message, and in the second round, the aggregated result is delivered to each worker hop by hop. Each worker establishes one *connection* to its successor (Figure 1B).

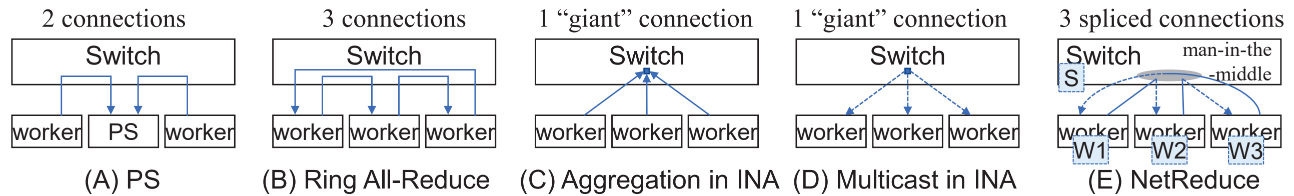

Figure 1: All-Reduce solutions and their connections.

Table 1: Performance of RoCE, parallel DMA, and DPDK.

	CPU	Throughput (Gbps)	RTT (us)
RoCE	25.6%	84.2	5.7
Parallel DMA	100%	58.67	70.2
DPDK	200%	90.5	20.5

In-Network Aggregation (INA) Preliminaries. INA offloads gradient aggregation onto network switches equipped with a programmable data plane (e.g., Tofino [1] in ATP and SwitchML [58], FPGA in PANAMA[14]) or a fixed-function ASIC (e.g., Quantum [43] in SHARP). Workers and switches form an *aggregation hierarchy*: workers stream gradient flows into the hierarchy; each switch aggregates its incoming flows into one flow and sends it to its parent; the root node (could be a switch [58] or a server [35]) multicasts the complete aggregation result back to workers along the hierarchy.

INA could reduce network traffic volume, shorten data transmission time, and eliminate incast bottlenecks. Thus, INA frees network resources [14], accelerates training jobs, and improves training scalability. For example, SwitchML reports a speedup of 5.5× for single jobs, and ATP reports 38-68% in a shared cluster.

In INA, there is no end-to-end connection. Instead, the aggregation hierarchy and the multicast tree act as a "giant connection" with multiple endpoints (i.e., leaf servers and the root). All endpoints cooperate to perform transport functions: network (de)multiplex, pace synchronization, flow control, and congestion control (Figure 1C and 1D).

2.2 Need for Transport Transparency

Existing INA replaces the transport layer. On its northbound, INA provides a "message" delivery interface for DT applications to exchange gradient messages (tensors); on its southbound, INA stack prepares "packets" for programmable switches. Thus, the INA stack needs to (de)packetize messages to/from network packets, where the data operation granularity exactly overlaps that in the transport layer.

Since the data mutation in INA contradicts the (immutable) stream abstraction in transport protocols, existing practices [8, 14, 17, 35, 38, 58] and proposals [64] choose to replace the transport layer. In detail, the traffic volume change (i.e., packet aggregation) would confuse the receiver's sequence number computing, acknowledgment, and window moving, and the packet content change would interfere with the packet integrity validation.

Issues of INA and transport coupling. As introduced in Section 1, the coupling of INA and the transport layer causes issues in performance, development, and operation. We illustrate the issues with examples of existing INA solutions. First, the INA stack cannot benefit from the transport layer advances. Recent data centers are evolving the message transport to RoCEv2 [18, 31, 76]. RoCE offloads message packetization and connection management onto hardware NICs and achieves high throughput and low latency with near-zero CPU overhead. However, INA stacks cannot be built atop RoCE and have to be implemented in software. We test the network I/O modules in existing INA — parallel DMA in ATP [35] and DPDK in SwitchML [58] and compare them with RoCE. Results in Table 1 show that the existing INA stacks cost more CPU resources but cannot match RoCE in latency and throughput.

Second, the INA stack causes development complexity in ML system building. We dissect the code organization of ATP. Table 2 lists the functions in ATP and their lines of code in implementation. We observe that nearly 40.81% of development effort is spent on reinventing the basic functions (packetization, flow control, reliability, and congestion control) in the transport layer. Such efforts could have been saved, had the traditional transport layer been reused. In addition, according to the code repository records, ATP developers spent more than four months integrating ATP with BytePS with 12K lines of code changed, which is a non-trivial amount of labor.

Table 2: Lines of code of functions in ATP network stack.

Function	Lines of Code	Percentage
Packetization & IO	1090	32.12%
Flow Control	50	1.47%
Reliability	181	5.33%
Congestion Control	64	1.89%
Floating Point Support	220	6.48%
Fallback	100	2.95%
Others	1689	49.76%
Total	3394	100%

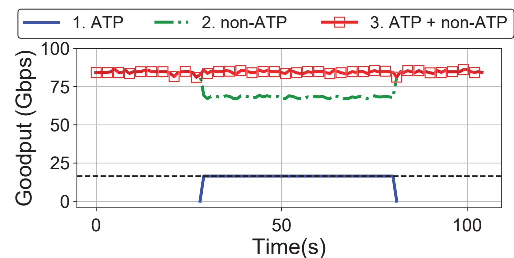

Figure 2: Bandwidth contention of VGG16 in ATP and background DCTCP, cited from [35].

Third, the INA stack causes network management complexity. When deploying INA in a legacy network, the operator has to make performance isolation between the INA traffic and existing traffic. Otherwise, there may be unexpected interaction behaviors between these transport variants. For example, Figure 2 shows the unfair bandwidth sharing of ATP and DCTCP due to their different congestion control algorithms. Making performance isolation involves configurations on tunneling (e.g., VLAN, VxLan) and QoS rules. Note that our goal is not to solve the fairness issue between transport variants, but to make INA "reuse" existing congestion control as so to reduce the complexity to manage bandwidth sharing.

2.3 Goal and Challenges

Goal. Our goal is to design a transport transparent INA primitive for DT jobs. The new primitive should achieve the communication acceleration and scalability from INA, inherit high throughput and low latency with low CPU overhead by reusing recent popular RoCE, and be development and management friendly. There are two challenges to this goal.

Challenge 1: Connection preservation. A data transmission connection delivers an end-to-end immutable byte stream, i.e., loss freedom, order-preserving, and content immutability; and the two endpoints maintain the connection states — sliding window — for the flow control. However, the INA aggregation hierarchy violates loss-freedom by consuming packets and content immutability by replacing packet payload with aggregation results.

Intuition. Existing solutions view the gradient aggregation and the result multicast as two separate processes. In each process, the data flow is indeed mutated. But if we put the two processes together, we can *splice data flows to form end-to-end connections*. Figure 1E illustrates an example: the gradient aggregation hierarchy has three data flows $W_i \rightarrow S$, and the multicast tree has three flows $S \rightarrow W_i$. We can splice the gradient data flow $W0 \rightarrow S$ with the result data flow $S \rightarrow W1$. The whole data flow $W0 \rightarrow W1$ would experience "man-in-the-middle" data manipulation but keep the volume unchanged, and (if the checksum is corrected) the two endpoints would not perceive the data mutation, keeping the connection states (e.g., sliding window) function correctly. Similarly, spliced data flows $W1 \rightarrow W2$ and $W2 \rightarrow W3$ would be end-to-end connections, and all connections form a ring.

Challenge 2: INA header recovery. Transport protocols segment a message into packets and add a layer-4 header, but INA solutions additionally require each packet to contain an *INA header* to direct the switch operation. For example, the switch needs an ID to distinguish simultaneous DT jobs, e.g., `JobID` in SwitchML/ATP and `aggregation group` in SHARP; the switch also needs a worker's position in the aggregation hierarchy to check the aggregation completion and deduplicate retransmitted packets, e.g., `bitmap` in ATP. However, existing transport protocols do not provide the flexibility to insert a custom header.

Intuition. We observe that a connection has a longer life than that of its packets. Thus, we can concatenate the INA header on the gradient, and hand the whole message to the transport layer. After packetization, the first packet would contain the INA header. The switch records the mapping between the connection and the INA header: the first packet would update the mapping, and the following packets would look up the mapping to recover the INA header.

3 DESIGN

We first introduce NetReduce's overall architecture and workflow (§3.1), and then describe its transport-transparent design (§3.2), multi-rack extension (§3.3), and parallel all-reduce method (§3.4).

3.1 Architecture and Workflow

Modules. Figure 3 shows the architecture of NetReduce. NetReduce consists of a message-level *flow control module* on hosts and an FPGA *accelerator* attached to the Ethernet switch. On the northbound of NetReduce, workers form a *ring* to perform the gradient all-reduce operation: each worker establishes a connection with its successor along the ring. The job worker passes the gradient messages to the flow control module, which further delivers the message to the connection of the ring.

NetReduce chooses to keep the server-to-server communication instead of building server-to-switch communication. Because existing applications use the server-to-server communication (i.e., TCP/RDMA connection in Ring AllReduce), NetReduce can keep this abstraction/interface unchanged so that the application does not need to be modified. If we choose to design a server-to-switch communication, we need to implement a network stack on the switch, it causes non-trivial development efforts and hardware resources to port a network stack onto the switch (e.g., commodity SHARP switch only implements partial network functions); the on-switch network stack can hardly be fully functional, and the server side still need to be modified to complement the missing functions (e.g., ATP and SwitchML need to change the server-side network stack for reliability).

Like existing INA solutions, NetReduce organizes an *aggregator array* for each ring in the accelerator. The aggregator works in the middle of connections: intercepting and aggregating gradient packets, and multicasting the aggregation result packets back to connections.

RoCEv2 as the transport layer. NetReduce chooses RoCEv2 as the transport layer but does not exclude other transport protocols. RoCE provides the following advantages: (1) applications have already applied standard RDMA interfaces (write/send/read verbs) for data transmission, and NetReduce could make minimum modifications to the legacy code; (2) the transport layer functions, e.g., reliability and congestion control, are offloaded onto the hardware NICs, which provides high performance and saves the effort of re-implementation.

NetReduce requires the *connection ID* and *packet sequence number (PSN)* of the data transmission to achieve transport transparency. In an RDMA connection, two hosts use *Queue Pair(QP)* as the connection endpoint. When a RoCE NIC packetizes a message, it inserts an IB BTH header between the UDP header and the data payload (in RoCEv2, see Figure 4). `dstQP` (destination QP) and `PSN` are in the IB BTH header, which can be recognized by the programmable data plane.

NetReduce can also adapt to other transport protocols with little modification — only the connection ID and sequence number are required. For example, in TCP, the connection ID is the five-tuple

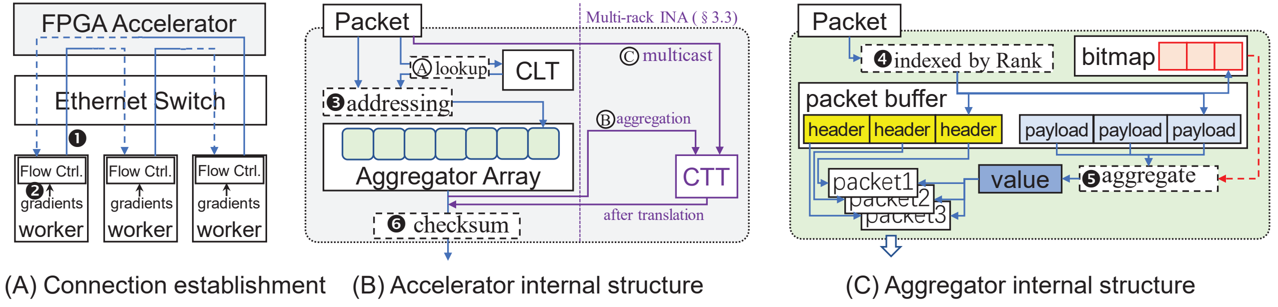

Figure 3: NetReduce Architecture and Workflow.

of IP addresses, port numbers, and the protocol, and TCP also has a sequence number field; as long as NetReduce can recognize the connection ID and the PSN, it would correctly achieve transport transparency.

NetReduce's RoCE-specific design also demonstrates its transport transparency. RoCE's transport layer is implemented in hardware on NICs, without providing any programming flexibility for INA, and our NetReduce prototype interacts with RoCE correctly and efficiently.

Initialization. When a DT job boots up, it sets up one or several rings for gradient aggregation, where each ring is assigned a `ringID` and each worker on the ring has a `Rank` ID. Each worker in a ring sets up one RDMA connection to its successor and also accepts one RDMA connection from its predecessor (❶). In the switch accelerator, an aggregator array is allocated for each ring. The aggregators in the array are accessed by index.

Worker Sending Gradient. Workers in the job synchronously send gradient tensors in the same order to the flow control module (❷). The flow control module chunks tensors to messages of the same fixed length `MsgLen`, and organizes the messages in a sequence. Each message is assigned a message ID `MsgID` whose value is its index in the message sequence.

The flow control module delivers messages sequentially to the RDMA connection. Each message is attached with an INA header (`MsgID`, `MsgLen`, `Rank`, and `RingID`) in front of the tensor values. The RDMA connection chunks a message into packets and sends packets to the network. The flow control module applies a *sliding window* to stream messages in concert with the aggregator array. The maximum window size is denoted as N messages, and the aggregator array size is set to be $2N$ messages to guarantee correctness (explained later).

Switch Aggregating Packets. When a gradient packet arrives at the accelerator, it is addressed to one aggregator in the aggregator array (❸). Packets with the same relative sequence number across workers are addressed to the same aggregator. A gradient packet's aggregator address is computed as

$$MsgID\%(2N) \times MsgLen + (PSN - PSN_0),$$

where $PSN - PSN_0$ is the relative sequence number of the current packet to the first packet of the message.

An aggregator has a bitmap and a packet buffer. Each gradient packet would set one bit in the bitmap and fill in a slot in the packet buffer, indexed by the packet's sender's `Rank` (❹). When an aggregator's bitmap is full, the aggregator adds up the payloads in its packet buffer, constructs packets using the original headers and the result, and outputs the packets (❺). The accelerator further re-computes the checksum of a packet and sends it back to the switch (❻).

The aggregator array is a circular array used by the gradient packet stream, it needs a way to deallocate aggregators which completes the aggregation. NetReduce does not immediately release an aggregator when the result packet is sent, because the result packet could be lost and further retransmission needs to fetch the result again. NetReduce makes a gradient packet to deallocate an aggregator *one window away* in the further, i.e.,

$$(MsgID + N)\%(2N) \times MsgLen + (PSN - PSN_0).$$

For a packet with PSN i, packets within the range of $(i - (MsgID - N) \times MsgLen, i + (MsgID + N) \times MsgLen)$ could all possibly be within the sliding window and in flight, thus, the aggregator array size is set to be $2N$ messages to avoid falsely deallocating an aggregator in use.

Worker Receiving Results. An aggregation result packet is forwarded back to its RDMA connections and the destination. NetReduce uses RDMA write[1], where the sender specifies a remote memory address at the receiver. The RDMA connection assembles packets at the destination memory address, and the receiver's flow control module fetches the result and hands the results to the training application.

3.2 Transport Transparency

NetReduce overcomes the two challenges to achieve transport transparency and is compatible with other transport layer functions.

Connection Preservation. As each gradient traverses the accelerator, it is temporarily held in the aggregator packet buffer. The packet payload is eventually replaced by the aggregation result. And the packet checksum is re-computed. So the receiver endpoint would not perceive the "man-in-the-middle" payload mutation and the connection endpoint would function normally.

INA Header Recovery. The flow control module concatenates an INA header (containing `RingID, MsgID, Rank,` and `MsgLen`) and tensor values as one whole message, and passes the message to the RoCE connection. The RoCE NIC chunks the message into a sequence of packets, where the first packet contains the INA header

[1]RDMA write is a one-side operation, where the receiver does not need to start a receive thread waiting for messages.

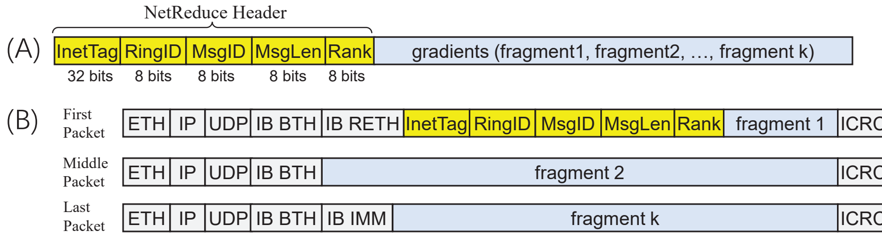

Figure 4: NetReduce Packet Format: (A) NetReduce message (INA header and tensor); (B) Packetizing a RoCE message on NIC.

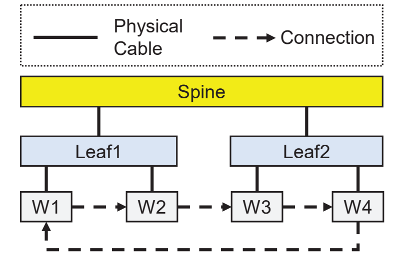

Figure 5: A Spine-Leaf Topology.

but the remaining packets do not (Figure 4). Since the switch accelerator needs the INA header in aggregator addressing, NetReduce introduces a *Connection Lookup Table (CLT)* to recover the INA header for non-first packets (Ⓐ in Figure 3).

CLT maps a connection to the aggregation direction state (INA header and other temporary states). For RoCE-specific connections, CLT has a format of

`<(SrcIP,DstIP,DstQP):(RingID,Rank,MsgID,PSN)>`.

The first packet of a message is identified by the `InetTag` (a magic number) in the INA header, and fills an entry in the CLT. The `PSN` field in CLT records the PSN of the first packet (denoted as $\texttt{PSN}_0$) of the message `MsgID`. The non-first packets would look up CLT and retrieve the aggregation direction state, which is further used in aggregator addressing (step ❸ in §3.1).

Reliability (and Correctness). The network could drop packets during transmission. NetReduce configures the host RDMA in Reliable Connection (RC) mode: whenever a packet is lost, the receiver finds the missing PSN and sends a NACK to the sender, and the sender retransmits packets since the missing PSN (go-back-N). The retransmission makes each packet eventually appear at the accelerator. NetReduce should guarantee correctness in various cases of loss and retransmission.

There could be computation errors in both the aggregator addressing process and the aggregation process. (1) If the first packet of a message is lost, the following packets cannot be correctly addressed. Because the CLT contains a stale `MsgID` and $\texttt{PSN}_0$ of the previous message. Since packets of the current message are out of the previous message's PSN range, i.e., $PSN - PSN_0 > MsgLen$, NetReduce identifies this case and drops packets whose first packet is lost. (2) A packet could appear more than once at the accelerator, e.g., the result packet loss triggering the gradient packet retransmission. Each packet is supposed to be aggregated "exactly one time" in the result [35, 58].

NetReduce's aggregation is *idempotent*, where the repeated execution of steps ❸ - ❻ always outputs the same aggregation value. Because NetReduce buffers packets, overwriting the buffer and computing on the buffer does not change the result. Note that this design is different from the "accumulation" method in existing solutions [35, 58], and the aggregation logic is simplified without the need to recognize and handle retransmitted packets specifically.

Flow Control. During the transmission and aggregation, each in-flight packet is addressed to one aggregator in the accelerator, thus, the sender needs to control the number of in-flight packets not to exceed the aggregator array size. Transport protocols adjust the window size to control the number of in-flight packets not to exceed the receiver's buffer, but NetReduce does not have control over the transport layer window size.

Instead, NetReduce's flow control module builds a message-level sliding window to control the in-flight traffic volume. On the worker, gradient messages are buffered in a FIFO queue with the sliding window. Messages within the window are passed to the RDMA connection; whenever a result message is received (from the predecessor), the window advances and the next message is sent. Bunching a window of messages and sending them together also improve the NIC bandwidth saturation. These two sliding windows (in the transport layer and NetReduce, respectively) cooperate in the flow control.

The window size N is decided by the network bandwidth-delay product. Assume each machine is dedicated to the DT job and could saturate the bandwidth $PortRate$; the switch can process (aggregating and switching) packets at the line rate. The in-flight bytes are $RTT \times PortRate$. The window should contain at least the same number of bytes to saturate the bandwidth. Thus, we have

$$N \times MsgLen \times pktSize \geq RTT \times PortRate,$$

and

$$N \geq \frac{RTT \times PortRate}{MsgLen \times pktSize}.$$

Congestion Control. NetReduce reuses the RoCEv2 Congestion Mechanism (RCM) on NICs. As NetReduce flow control bounds the window size not to exceed N messages, the congestion control would not falsely stream more packets (than the aggregator array size) into the network.

Packet Integrity. NetReduce re-computes the checksum before a result packet is sent out. So the receiver NIC would not misinterpret the result packet as a corrupted one. RoCE-specific packets have a checksum — Invariant CRC (ICRC) — appended after the payload. For a NetReduce RoCE connection, packet integrity is guaranteed: the sender NIC appends ICRC, the accelerator mutates the payload and re-computes the ICRC, and the receiver NIC verifies the ICRC.

3.3 Extension to Multi-racks

In modern data centers, a DT job may span multi-racks. Thus, we extend NetReduce to support hierarchical multi-rack aggregation. Figure 5 shows a spine-leaf topology with four workers across racks. Multi-rack INA makes two changes to the single-rack design: virtual switch-to-switch connection and connection translation.

Virtual Switch-to-Switch Connection. In hierarchical aggregation, traffic volume changes (decreased in aggregation and increased in multicast), thus, connections cannot be spliced. Instead,

Table 3: Connection Translation Table in Figure 5.

Switch	Direction	From_Conn	To_Conn
Spine (S)	Aggregation and Multicast	$L1 \rightarrow S$	$S \rightarrow L1$
		$L1 \rightarrow S$	$S \rightarrow L2$
		$L2 \rightarrow S$	$S \rightarrow L1$
		$L2 \rightarrow S$	$S \rightarrow L2$
Leaf (L1)	Aggregation	$W1 \rightarrow W2$	$L1 \rightarrow S$
		$W2 \rightarrow W3$	$L1 \rightarrow S$
	Multicast	$S \rightarrow L1$	$W1 \rightarrow W2$
		$S \rightarrow L1$	$W4 \rightarrow W1$

NetReduce translates connections. In an aggregation hierarchy, NetReduce builds virtual *switch-to-switch* connections for neighboring switches, e.g., $L1 \rightarrow S$, $L2 \rightarrow S$, $S \rightarrow L1$, and $S \rightarrow L2$ in Figure 5.

Connection Translation.

The switch maintains a *Connection Translation Table (CTT)* to translate packets along the aggregation hierarchy and the multicast tree. CTT entry's key is the connection that may trigger the translation, i.e., the child connections in aggregation and the parent connection in multicast; CTT entry's value is the connection that leaves the switch, i.e., the parent connection in aggregation and the child connection in multicast. Table 3 shows the CTTs for the example: the root switch has four entries (a combination of two gradient data flows and two result data flows), and the leaf switch has four entries (two for aggregation and two for multicast).

In the aggregation process (traffic from leaf to root), each accelerator first aggregates packets as in §3.1, and then looks up the CTT before the checksum computation (Ⓑ in Figure 3). NetReduce duplicates the packet for each matched entry in CTT, and translates the packet header to the entry's target connection, i.e., its value field.

In the multicast process, the accelerator translates the packet based on CTT, bypasses the aggregator array (Ⓒ), and then updates the packet checksum and sends it back to the switch.

Table Populating. Non-leaf switches' CTTs only contain switch-to-switch connections, which are known before hosts start RDMA connections; thus, they are pre-computed and populated during the job initialization. Leaf switch CTTs contain end-to-end connections, which are dynamically decided when hosts initiate RDMA connections; thus, they are populated along with the RDMA connection establishment.

Each message's first packet not only participates in the aggregation but also (its header) is duplicated as a CTT populating notification and transmitted along its route. For an end-to-end connection, its source leaf switch adds a rule translating the connection to its parent switch-to-switch connection, and its destination leaf switch adds a rule doing a reverse translation. For example, the connection $W2 \rightarrow W3$ would populate two rules, one from $W2 \rightarrow W3$ to $L1 \rightarrow S$ on $L1$ (shown in Table 3) and one from $S \rightarrow L2$ to $W2 \rightarrow W3$ on $L2$.

3.4 Parallel All-Reduce using NetReduce

Modern ML training clusters are usually equipped with multi-GPUs on each machine. In a DT job, each GPU is dedicated to one worker. These workers/GPUs can communicate in different patterns [24].

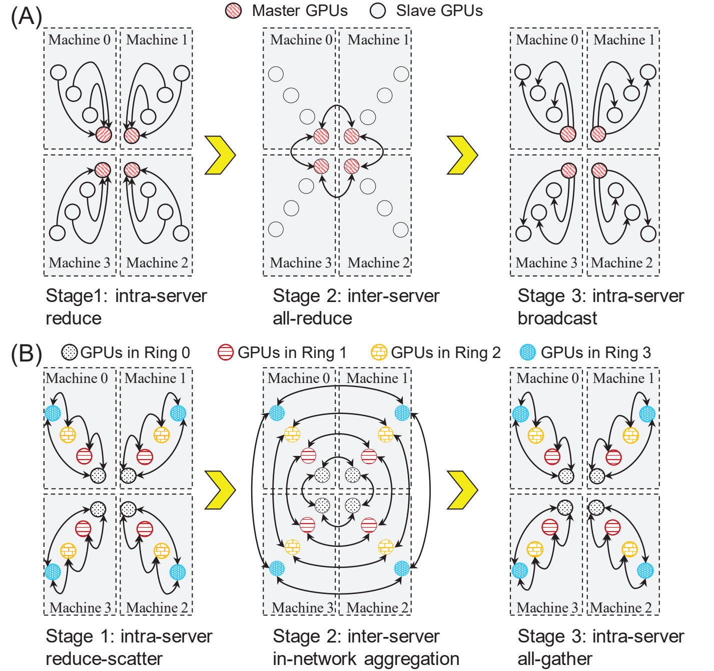

Figure 6: Communication Patterns: (A) Tencent AllReduce: master GPUs become bottleneck; (B) NetReduce: 4 arrows form a ring.

Constructing the communication pattern should take the *bandwidth gap* between intra- and inter-machines into consideration for better overall throughput. The NetReduce protocol provides the flexibility for a job to build multiple parallel rings to saturate the bandwidth. We list Flat Ring AllReduce (FR) and Tencent AllReduce (TA) as the baselines and derive the NetReduce-specific design — Parallel NetReduce (PN).

Flat Ring AllReduce (FR). All GPUs of all machines form one ring without hierarchical aggregation. This design ignores the bandwidth gap between intra-machine and inter-machine.

Tencent AllReduce (TA). It is a hierarchical all-reduce [26] as shown in Figure 6A. Each machine has one GPU as the master GPU. The aggregation of all GPUs' gradients is performed in three stages. First, all GPUs within one machine perform a `reduce` operation to aggregate the whole gradient to the master GPU. Second, all machines' master GPUs form a ring and perform `all-reduce` operation. By then, each master GPU would get the global aggregation result (the final complete aggregation result). Third, each master GPU `broadcasts` the global result to other local GPUs within the same machine. In this pattern, the master GPUs could undertake a heavier workload than slave GPUs, causing an imbalanced usage of GPU resources.

Parallel NetReduce (PN). NetReduce system fully balances the aggregation workload. In a NetReduce job with total P GPUs, each machine with n GPUs (P/n machines), the job sets up n rings during initialization. The i-th GPU on each machine belongs to the i-th ring.

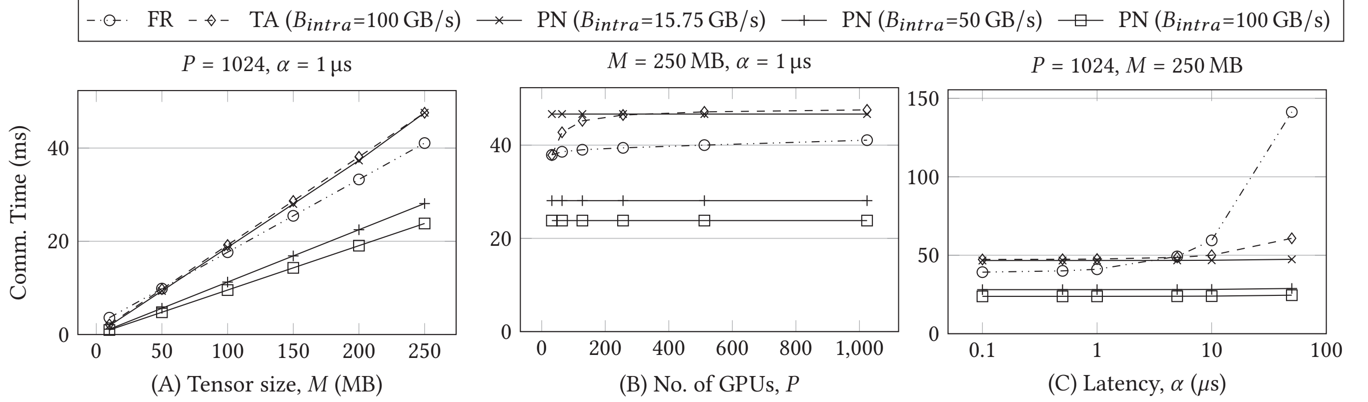

Figure 7: Communication cost taken by a single machine for parameter synchronization (n=8, B_{inter}=12.5 GB/s, varying B_{intra}).

It also has three stages shown in Figure 6B. First, each GPU chunks its gradient of size M into n pieces, and all GPUs within a machine perform the `reduce-scatter` operation. That is, within each machine, the i-th GPU would fetch the i-th piece of the gradient from all other local GPUs, and aggregate these pieces. Second, GPUs belonging to the same ring perform `NetReduce` aggregation, i.e., the i-th ring aggregates all machine's i-th GPU's i-th gradient piece. Then each machine's i-th GPU would have the global aggregation result of all gradient's i-th pieces. Third, all GPUs within a machine perform the `all-gather` operation to exchange the complete aggregated piece, i.e., the i-th GPU broadcasts its i-th piece to all other local GPUs and replaces their i-th local pieces.

It is worth noting that Ring All-Reduce can also apply parallel rings like NetReduce, i.e., replacing the NetReduce INA in Figure 6B with Ring All-Reduce, called Parallel RAR (PR). PR is also a feasible approach that can accelerate All-Reduce by parallelism. Comparing NetReduce with PR, NetReduce has the advantage of reducing traffic volume (and the transmission time) due to in-network aggregation.

Comparing Approaches. PN outperforms FR and TA in most scenarios. The intuitive reasons are as follows. First, for inter-machine traffic, all endpoints in PN send traffic concurrently, which meets simultaneously at the switch; but those in FR and TA have to wait and accumulate tensor values along a ring, suffering from hop-by-hop delay accumulation on the ring. Thus, FR and TA experience larger traffic relay latency. Especially for small messages, the latency is more obvious.

Second, for intra-machine traffic, it is reasonable to balance the aggregation workload as in PN. Because intra-machine data paths have more abundant bandwidth than inter-machine ones (e.g., NVLink with 1.2 Tbps internal all-to-all bandwidth), balancing data among GPUs does not cause obvious overhead; furthermore, with all GPUs participating in the aggregation, the in-network reduction process is parallelized and accelerated.

Appendix § A gives the mathematical modeling to compare approaches, and we list the results below. Table 4 defines the notions to describe the testbed settings. The sufficient conditions where PN outperforms FR and TA: $P > 3n$ and $\frac{B_{intra}}{B_{inter}} \geq \frac{2P}{P-2}$ $(P > n \geq 2)$. In a production network, the first is not hard to achieve, e.g., our testbed has $P = 32$ and $n = 8$; and the latter can be achieved with the recent progress of intra-machine GPU inter-connection: NVLink makes B_{intra}=150 GB/s and typical high-speed Ethernet is B_{inter}=100 Gbps.

Simulation. Figure 7 shows the simulation results of comparing PN with FR and TA in a multi-GPU multi-machine cluster. We conduct a flow-level simulation to understand the impact factors influencing communication time. We simulate a multi-GPU multi-machine cluster with n = 8 and B_{inter} =12.5 GB/s (Ethernet), compare PN with FR and TA, and tune the intra-machine bandwidth B_{intra} from 15.75 GB/s (16-lane PCIe 3.0) to 100 GB/s (NVLink), total number of GPUs P from 32 to 1024, and per-hop latency on a ring α from 0.1 µs to 100 µs.

First, FR's and TA's communication time varies with P and α, but NetReduce's does not (Figure 7B and 7C). The reason is that FR (TA) has a ring structure, and the total latency on a ring is decided by the number of hops P (P/n) and the per-hop latency α. But PN intra-machine reduce-scatter and all-gather is one hop, and the inter-machine aggregation re-organizes the (logical) ring into a physical aggregation hierarchy with limited hops.

Second, for typical tensor transmission, the data transmission time dominates over the latency; PN could reduce this dominant factor more significantly than FR and TA. A model with the size of 100 MB transmitted on a 10 GB/s link costs 10 ms; but a typical per-hop latency is 1-10 µs. Increasing B_{intra} to reduce the transmission time could effectively reduce the overall time. For example, in

Table 4: Symbols and their meaning.

Symbols	Meaning
M	The size of the gradient
n	Number of GPUs per machine
P	Total number of GPUs in a job
α	Per-hop latency in a ring for data preparation.
N	The window size
B_{intra}	Intra-machine bandwidth
B_{inter}	Inter-machine bandwidth, i.e., network bandwidth
T_{fr}	Comm. time of one iteration in Flat Ring
T_{ta}	Comm. time of one iteration in Tencent AllReduce
T_{pn}	Comm. time of one iteration in Parallel NetReduce

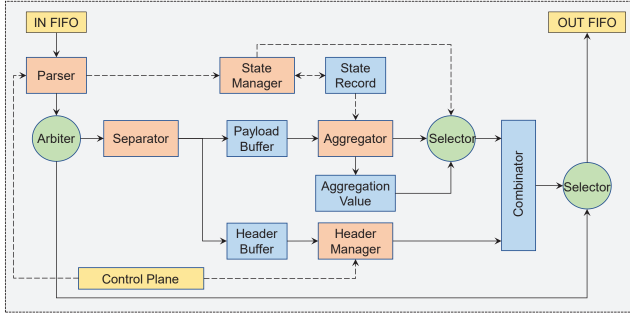

Figure 8: The architecture of NetReduce FPGA accelerator (dash and solid arrow lines refer to control and data flows, respectively).

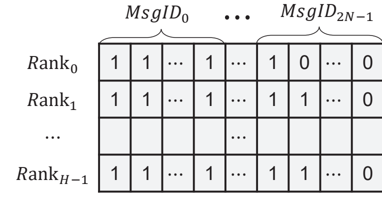

Figure 9: Arrival states (bitmap) of packets for each ring (H and N refer to no. of hosts and no. of message in a window, respectively).

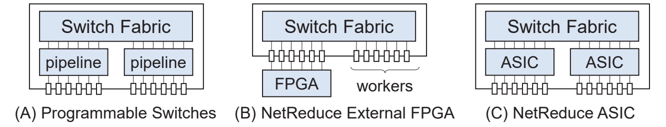

Figure 10: Accelerator with an Ethernet Switch.

Figure 7A and 7B, with B_{intra} larger than 15.75 GB/s, PN always outperforms FR and TA.

4 IMPLEMENTATION

FPGA NetReduce Accelerator (and Justifying the Choice). We prototype the INA accelerator on an FPGA board. The FPGA board is equipped with a Xilinx Virtex Ultrascale chip [71] which supports six ports at a line rate of 100 Gbps. The implementation of NetReduce consumes small portions of the whole FPGA resources: 10.31% (109025), 7.91% (167554), and 26.27% (993) for LUTs, Flip-Flops, and BRAM, respectively.

The internal architecture of the FPGA accelerator is shown in Figure 8. The bitmap, packet buffer, and value in aggregators are implemented as separate arrays — `State Record`, `Header Buffer`, `Payload Buffer`, and `Aggregation Value`. When a packet arrives, a `Parser` identifies the aggregation packet or directs other kinds of packets to the output port directly. The `Parser` further feeds the NetReduce header to a `State Manager` which tracks the arrival states of the packets, i.e., the bitmaps of aggregators. Figure 9 shows the data structure to track the arrival states: it is a matrix of bits with the row index indicating a host and the column index indicating packet sequence number (each column is a bitmap). The `Aggregation Value` is the array of tensor values field of aggregators. `Header Manager` is in charge of packet address translation. `Combinator` merges the header with the payload and sends the final packet out.

The INA accelerator is deployed as an external middlebox attached to a commodity Ethernet switch; both sides spare six ports to connect. The INA accelerator is fully compatible with legacy Ethernet switches. The switch is configured with rules to route DT traffic into the accelerator. The wiring with the Ethernet switch is shown in Figure 10B. If the NetReduce accelerator tapes out to ASIC, it can be integrated into the switch in the same way as the programmable switch (Figure 10C).

We argue that FPGA is a reasonable approach for NetReduce compared with programmable switches. First, FPGA has better programmability to completely implement transport functions, e.g., ICRC. Second, implementing CLT and CTT on programmable switches is costly: one or two pipeline stages and out-of-band table management. Third, programmable switch-based solutions either suffer from small packet issues and consequent low goodput (e.g., SwitchML and ATP) or spare too much switch internal bandwidth to complement the low goodput (e.g., SwitchML RDMA); the extra switch bandwidth overhead is even larger than that of NetReduce's wiring. Section 6 has a quantitative analysis of the bandwidth overhead.

Host. NetReduce reuses RoCE (its high-speed I/O and reliability) and the ring abstraction in NVIDIA Collective Communication Library (NCCL) [51], whose communication primitives are optimized for NVIDIA GPUs and networking. And the host is made a minimum modification, including incorporating NetReduce header with gradients, adding a sliding window, and modifying enqueue/dequeue behaviors.

Switch. NetReduce uses standard commodity Ethernet switches. They are configured with Access Control List (ACL) rules to direct DT traffic to FPGA, and each worker would install one extra rule in the switch.

Training Framework. We implement NetReduce protocol as a new primitive *GenericOp* in NCCL-2.4.7 [48]. And we use the primitive in PyTorch-1.5.1 [65] and Horovod-0.16.0 [59] supported TensorFlow-1.12.0 [66] for training.

Comparison of implementation complexity. Table 5 shows the lines of code (LoC) of ATP, SwitchML, and NetReduce. NetReduce host reuses RoCE (interfaces and transport functionalities) and the ring in Ring AllReduce, so it has the lowest complexity on hosts (only ~850 LoC v.s. ~3400 and ~3100 in ATP and SwitchML).

5 EVALUATION

The evaluation demonstrates NetReduce's superior properties.

Table 5: Lines of code in solutions.

	SwitchML	ATP	NetReduce
Host (C/C++)	~3100	~3400	~850
Switch	~3700 P4	~5200 P4	~8600 FPGA

Table 6: [6 machines] Micro benchmark.

	Tput/Gput	CPU	RTT	Switch Overhead
ATP	50.3/39.7 Gbps	128%	751 μs	50% bandwidth
SwitchML	46.8/32.7 Gbps	130%	812 μs	N/A
SwitchML RDMA	89.4/82.7 Gbps	100%	8.1 μs	75% bandwidth
NetReduce	92.5/86.8 Gbps	100%	6.3 μs	50% switch ports

(1) NetReduce inter-operates with the transport layer correctly, and inherits RoCE's low overhead, high throughput, and low latency. (§5.2)
(2) NetReduce's overhead of switch ports is not more significant than programmable switch-based solutions. (§5.2)
(3) NetReduce accelerates and scales DT jobs, and the parallel all-reduce based on NetReduce efficiently utilizes both intra- and inter-machine bandwidth compared with the state of the art. (§5.3, §5.4)
(4) NetReduce is cost-effective, achieving performance gain with a lower capital expense and power. (§5.5)

5.1 Experiment Settings

Environments. We test NetReduce on two testbeds. (1) Testbed 1 is a single-GPU multi-machine one that can validate the NetReduce protocol. There are six machines, each equipped with two 10-core CPUs (Intel Xeon E5-2064 2.4 GHz), 32 GB*3 DDR4 memory, one NVIDIA Geforce RTX 2080 8 GB GPU [52], and a Mellanox ConnectX-5 [42] 100 GbE NIC. (2) Testbed 2 is a multi-GPU multi-machine one that we use for a limited time to validate parallel NetReduce. There are four machines, each with two 18-core CPUs (Intel Xeon Gold 6154 3.00 GHz), 1 TB (64 GB *16) DDR4 memory, eight NVIDIA Tesla V100 SXM2 32 GB GPUs [50], and a Mellanox ConnectX-5 100 GbE NIC. Within each machine, a hybrid cube-mesh topology [47] is used for 8-GPU interconnection via NVLink [49].

Baselines. We compare NetReduce with Ring AllReduce, SwitchML, and Tencent AllReduce. Ring AllReduce and Tencent AllReduce are implemented by NCCL-2.4.7, while SwitchML is implemented by using a programming switch equipped with a Tofino chip. We also use a microbenchmark to compare NetReduce, ATP, and SwitchML.

Workloads. We evaluate the systems in typical image classification training workload ImageNet [9]. Three representative Convolutional Neural Network (CNN) models are chosen: AlexNet [33], VGG16 [63], and ResNet50 [21]. We also evaluate NetReduce on typical NLP Transformer models, including BERT[10] and GPT [55] pertaining, and GLUE [62, 69] and SQuAD [23, 56] fine-tuning.

Parameters. In the experiments, the sliding window size $N = 2$, each message has 170 packets, and the packet payload carries 1 KB of data. In the experiment, we also tune the parameters of batch size and value precision to observe their impact on the result.

Metrics. We measure two metrics: the *training throughput* — the average number of samples that each worker/GPU can train in one second — to measure the system efficiency; the *loss* — the objective function of training — to measure the training quality. The experimental results are obtained by using float-point number arithmetic for aggregation.

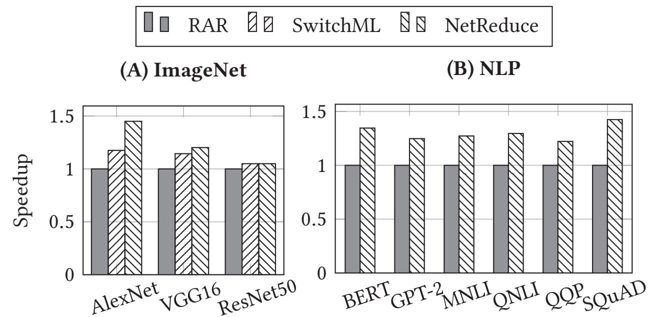

Figure 11: [Testbed 1] Speedup of NetReduce and SwitchML v.s. RAR for deep learning models.

5.2 Micro Benchmark

We run micro-benchmarks to make all-reduce communication on ATP, SwitchML, SwitchML RDMA, and NetReduce. There are six workers in the experiment. In the benchmarking, all workers send traffic with their best effort. Table 6 shows the results.

Transport Transparency. NetReduce inter-operates with the underlying RoCEv2 correctly. Together with the LoC comparison in Table 5, the transport transparent design of NetReduce makes it friendly for development and deployment.

Performance Acceleration. ATP and SwitchML consume more CPU than NetReduce, have large latency, but only achieve half of NetReduce's throughput. ATP and SwitchML run in software multi-thread mode, which causes the CPU and latency overhead and low performance. The latency increased by software packetization in ATP and SwitchML even exceeds the latency introduced by two extra hops (between FPGA accelerator and switch) in NetReduce. ATP and SwitchML also suffer from the small packet issue, and their goodput is lower (less than half of NetReduce).

Switch Overhead. NetReduce attaches the external FPGA accelerator to the switch, consuming 50% of switch ports. But for solutions using programmable switches (mainly Tofino), the hardware limitation also causes equal or even larger bandwidth overhead. In Tofino switches, a pipeline can only connect 16 switch ports and process four 32-bit tensor values in the payload (detailed discussion in §6). ATP uses internal recirculation to process packets with a larger payload, the recirculation costs 50% switch internal bandwidth; SwitchML does not handle the small-payload issue, its goodput upper bound is 63% line rate (128-byte payload) [58]; SwitchML RDMA chains four pipelines of a 6.4 Tbps switch to get a huge pipeline for a larger payload, but the recirculation costs 75% switch internal bandwidth, leaving only 16 ports for machines. INA solutions usually need to make a tradeoff between extra hardware costs and high performance. NetReduce costs reasonable switch ports/bandwidth compared with state-of-the-art solutions.

5.3 Performance on Single-GPU Machines

Training Throughput. We run CNN model training on Testbed 1 and show the results in Figure 11A. First, NetReduce and SwitchML both accelerate the training speed compared with Ring AllReduce (RAR), e.g., the training speed ratio of the three is 1.45:1.18:1 for

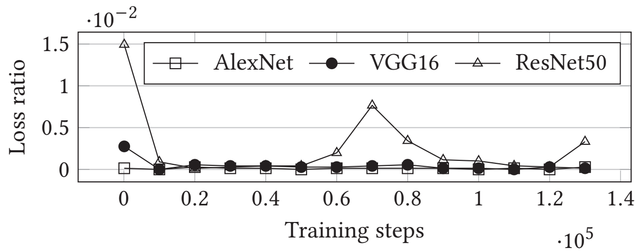

Figure 12: [Testbed 1] Loss difference ratio between NetReduce and ring all-reduce.

AlexNet because INA reduces the gradient transmission time. Second, NetReduce outperforms SwitchML by a larger speedup for the reason that NetReduce has full-length Ethernet frames and segmentation is offloaded to RoCE NIC. Third, NetReduce and SwitchML have similar performance on ResNet50. The reason is that ResNet50 is a computation-intensive model, and the communication time improvement from INA is marginal in this case.

Figure 11B shows the performance gain of NetReduce compared with RAR in NLP model training. We get similar observations with the CNN models. Compared with RAR, NetReduce improves BERT pretraining, GPT-2 pretraining, GLUE-MNLI, GLUE-QUNLI, GLUE-QQP, and SQuAD by 34.6%, 24.8%, 27.3%, 29.6%, 22.2%, 42.5%, respectively.

Training Quality. Figure 12 shows the loss difference between NetReduce and RAR in training iterations, and the metric is $\frac{|LOSS_{inet}-LOSS_{ring}|}{LOSS_{ring}}$. Both solutions progress at the same pace (the same loss value) with the training iterations: the loss ratio is usually smaller than 0.2% (ResNet50 occasionally shows a larger difference of 1.5%, but is still negligible). Thus, NetReduce does not pay the price of losing training accuracy to get a higher speed (Figure 11).

5.4 Performance on Multi-GPU Machines

We train CNN models on testbed 2 with two settings: one GPU on each machine involved (Figure 13) and all GPUs involved (Figure 14). Ring AllReduce and NetReduce are compared in the single-GPU setting, and Flat Ring (FR), Tencent AllReduce (TA), and Parallel NetReduce (PN) are compared in the multi-GPU setting. The batch size (BS) and precision, i.e., 16bit and 32bit floating-point (FP), are tuned. The training throughput is measured.

Scaling across machines. NetReduce could scale linearly with the number of machines, which outperforms other solutions. In both figures, NetReduce's average throughput of 4 machines is the same as the throughput of a single machine. (Parallel) NetReduce has performance gains of 68.8%, 57.9%, and 35.6% for AlexNet compared with FR, TA, and RAR with BS=32 and FP16. The experiment result complies with the analysis in §3.4: NetReduce achieving better cross-machine scalability by coordinating concurrent worker sending and balancing GPU workload.

Impact on different models. NetReduce impacts model training differently due to their communication-to-computation ratio, and benefits the communication-intensive models. Table 7 further specifies the case of BS=32 and FP16 in Figure 13. In Table 7, NetReduce improves AlexNet on the throughput by 35.6%, which is

Table 7: Training performance in Figure 13 (4 GPUs) with BS=32 and FP16.

Model		Throughput (images/s)	Iteration (ms)	Communication (ms)
AlexNet	Ring all-reduce	527.9	60.62	47.12(77.7%)
(236 MB)	NetReduce	716.0	44.69	31.10(69.6%)
	↑	35.6%	26.3%	34.0%
VGG16	Ring all-reduce	172.9	185.08	111.98(60.5%)
(528 MB)	NetReduce	215.3	148.63	74.64(50.2%)
	↑	24.5%	19.7%	33.3%
ResNet50	Ring all-reduce	358.8	89.19	23.04(25.8%)
(98 MB)	NetReduce	383.6	83.42	19.29(23.1%)
	↑	6.9%	6.5%	16.3%

Table 8: Throughput scaling-down for AlexNet, FP=32.

Batch Size	NetReduce (images/s) 32 GPUs	4 GPUs	↓	Flat Ring (images/s) 32 GPUs	4 GPUs	↓
64	938.4	1372.2	31.6%	585.2	1051.1	44.3%
128	1555.5	2225.3	30.1%	1037.2	1912	45.8%

Table 9: Training performance in Figure 14 (32 GPUs) with with BS=32 and FP16.

Model		Flat ring all-reduce	Tencent all-reduce	Parallel NetReduce
AlexNet	Images/s	307.5	328.8	519.2
(236 MB)	↑	68.8%	57.9%	-
VGG-16	Images/s	115.2	122.2	173.6
(528 MB)	↑	50.7%	42.1%	-
ResNet-50	Images/s	276.0	282.8	317.6
(98 MB)	↑	15.1%	12.3%	-

the most. This is because when using the ring all-reduce algorithm to train AlexNet, the time taken for communication occupies 77.7% (=47.12/60.62 as shown in the 5th column in Table 7) of the whole iteration time, which has a significant potential to improve. Indeed, NetReduce improves AlexNet in communication by 34.0%. On the contrary, although VGG16 is improved on communication by 33.3%, which is similar to AlexNet, the communication part occupies 60.5%, which is smaller than AlexNet, resulting in a smaller improvement in total training throughput (24.5%). Especially for ResNet50, which is a computation-intensive model, with 16.3% improvement on the communication part, which accounts for only 25.8% of the iteration time, we only have 6.9% improvement on the training throughput.

Scaling across GPUs on multi-machines. FR, TA, and PN show sub-linear scaling in the multi-GPU multi-machine setting, but PN degrades less. Table 8 shows the average GPU throughput of AlexNet in Figure 13 and Figure 14. NetReduce decreases from 1372.2 to 938.4 (31.6%) and FR/RAR from 1051.1 to 585.2 (44.3%). All solutions cannot handle the inter-machine and intra-machine bandwidth gap without throughput loss, because moving data between the network and GPU memory is expensive (especially for multi-GPUs with one NIC); NetReduce shows less degradation, as its parallelization method (GPU balanced) and INA acceleration complements the loss.

Table 9 details Figure 14 when BS=32 and FP16. Parallel NetReduce outperforms Flat Ring by 68.8%, 50.7%, and 15.1% for AlexNet,

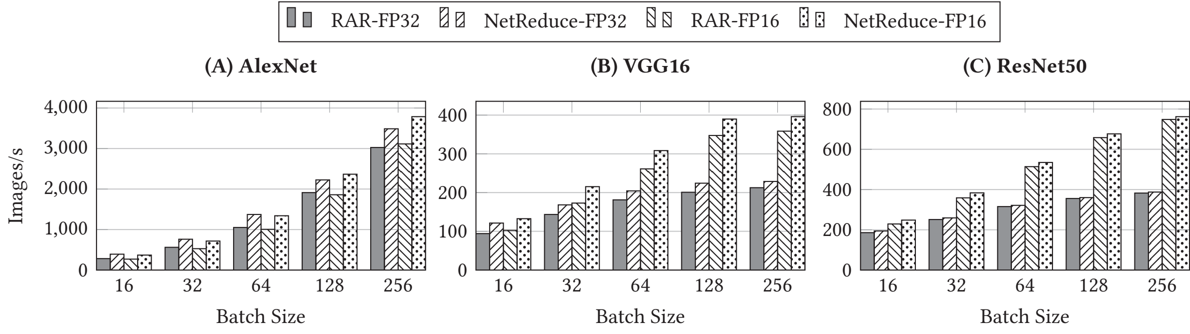

Figure 13: [Testbed 2: 1 GPU/machine] Training throughput, varying batch size (BS) and precision.

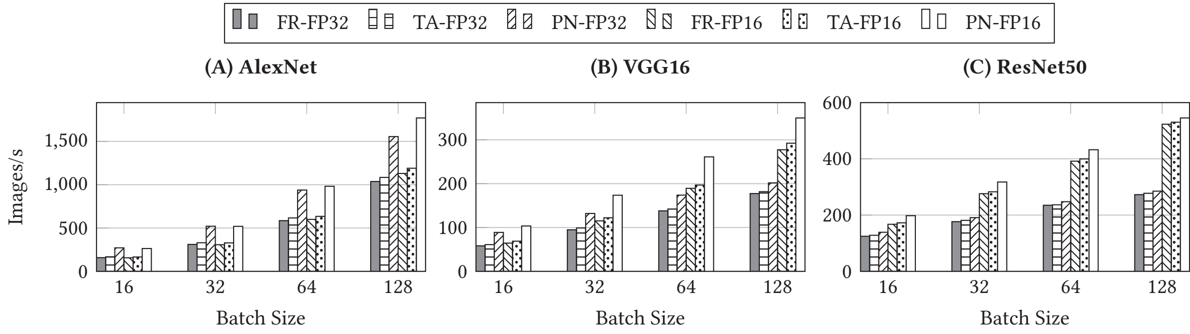

Figure 14: [Testbed 2: 8 GPUs/machine] Training throughput, varying batch size (BS) and precision.

VGG-16, and ResNet-50, respectively. Compared with Tencent All-Reduce, parallel NetReduce speeds up training by 57.9%, 42.1%, and 12.3% for the three models, respectively.

Validating Communication Modeling (in §3.4). Our experiment validates the report in [26] that TA brings performance gain for tensors with smaller sizes, but for relatively larger tensors, FR still outperforms TA. In Appendix § A, we model the communication cost of each algorithm as a combination consisting of two items: message processing latency item with α and tensor transmission item with M. The α item is mostly affected by the number of GPUs participating in training P. Therefore, for small tensors where the α item accounts for most communication costs, TA gives superior performance. However, for big tensors where the M item accounts for most communication costs and the system becomes less sensitive to P, TA brings less gain.

PN would always outperform FR if condition (6) holds, regardless of tensor size. Considering our hardware prototype, substituting P=32 and n=8 into (6) gives $\frac{B_{intra}}{B_{inter}} \geq 2.3$. Indeed, intra and inter nodes being connected via NVLink and 100GbE gives B_{intra} = 150 GB/s and B_{inter} = 12.5 GB/s, respectively. Therefore, in our hardware prototype, $\frac{B_{intra}}{B_{inter}}$=12 > 2.3. With increased P, the α item accounts for a larger proportion in FR, resulting in poor scalability. PN reduces the impact of α item by dividing a big ring into multiple small parallel rings, improving the scalability.

Table 10: Cost effect comparison, with price in 2019.

	CapEx($)	Power (W)
One GPU (V100)	9075.4	300
Unit Switch Port (3.2 Tbps)	245.5	9.4
One FPGA Board	6000	28.3
Testbed 2 without NetReduce	291394.6	9637.4
Testbed 2 with NetReduce	298376.5	9703.2

5.5 Cost Effectiveness

Table 10 shows the price and power of the unit component and the whole Testbed 2. When the testbed is not equipped with NetReduce, each machine consumes one switch port; when equipped with NetReduce, there is one extra FPGA board that consumes six extra ports. NetReduce spends 2.40% more capital expense and 0.68% more power than the original testbed. According to the experiment in Figure 14, the extra cost promotes the cluster efficiency by 12.3%-57.9% in model training. Thus, NetReduce is a cost-effective solution to promote existing GPU clusters.

6 DISCUSSION

Compared with Programmable Switch-based Solutions. Switch-based INA solutions are difficult to achieve full functionality, line rate, and low overhead together. First, NetReduce achieves transport transparency essentially by maintaining the connection states in the switch. Implementing NetReduce on the programmable switch

incurs two kinds of overhead: one stage for CLT and one for CTT, and, more critically, an out-of-band process to populate the table for each connection/message, which delays each message. In addition, some functions like ICRC computation are not available on switches, and the endpoint has to turn the feature off, which, however, could lead to the endpoint misbehaving in the case of packet corruption.

Second, programmable switches have hardware limitations that lead to small payload sizes and consequent low bandwidth efficiency, and existing solutions make complicated tradeoffs to complement the throughput. The programmable switches process a packet in a pipeline of stages, with each stage able to process four 32-bit integers and each pipeline can connect 16×100 Gbps ports. One pass of a packet can only process values of at most 192 bytes (12 stages $\times$ 4 registers/stage $\times$ 4 bytes/register). (1) SwitchML does not handle the small payload issue specifically, thus, its goodput upper bound is 63% line rate (reported in [58]). (2) ATP runs recirculation for each packet, so that the packet has a second pass on the switch pipeline, and the goodput can be 40 Gbps with 256-byte payload. However, recirculating packets once costs 50% internal bandwidth. (3) SwitchML RDMA recirculates packets on all four switch pipelines to get a huge pipeline; it achieves 89.4 Gbps throughput (line rate), but it costs 75% switch ports saturated in loopback mode.

NetReduce achieves full transport functions and transparency at line rate, and the overhead is no larger than switch-based solutions.

Further Comparison with SwitchML RDMA. SwitchML RDMA uses RDMA Write Immediate in Unreliable Connection (UC) transport mode (like UDP) and (logically) chains pipelines to achieve a line rate packet processing. This design is not fully transport transparent. In UC mode, unordered and duplicated packets are detected by the responder but the requester is not informed. To achieve reliability when packet loss happens, the application has to be involved in loss detection, notification, and retransmission.

SwitchML RDMA also encodes aggregation information in the RDMA header, which could affect the flexibility of setting up RDMA connections. SwtichML INA metadata such as pool index is encoded in RDMA Extended Transport Header (RETH) which position originally contains Virtual Address and R_Key for remote DMA. RETH only appears in the first packet in a message, and SwitchML RDMA links non-first packets to first packets by using a sequence number as NetReduce does. Once the switch completes a packet aggregation (metadata is consumed), it reloads Virtual Address and R_Key back to packet RETH. This requires endpoints to install such information to the switch out of the band at the job initialization time.

SwtichML RDMA turns off the ICRC on the NIC because the programmable switch mutates the packet content but cannot compute this checksum. Thus, the endpoint is not able to detect packet corruption.

Feasibility on SmartNIC/DPU. Recently, SmartNIC/DPU has gained attention. They provide programmability on NICs. Thus, it provides the potential to offload the INA stack to SmartNICs/DPUs.

This approach can save the host CPU, but may not get development simplicity or good performance. (1) For SmartNICs with FPGA to program, it is not trivial to re-design and re-implement the entire INA stack on FPGA, especially the transport layer. (2) For DPUs with ARM cores, the programmability is sufficient, but dumping a network stack on ARM cores makes its software architecture to be the same as on the host, which could suffer from similar latency and low throughput issues as existing INA solutions.

Multi-Tenancy. NetReduce supports multi-tenancy. Different jobs can declare disjoint ring IDs and accelerator memory regions for their gradient aggregation.

7 RELATED WORK

INA Solutions. ATP [35], SwitchML [58], iSwitch [38], PANAMA [14], and NVIDIA's accelerator-centric network [32] build dedicated network stack. SHARP [16] has hardware support on NIC and switches. Flare [8] only proposes the switch logic. OmniReduce [13] does not fully elaborate its INA realization in a switch, which derives from SwitchML and inherits some of its limitations. SwitchML RDMA is implemented in RDMA Unreliable Connection mode (like UDP) as discussed in Section 6. To the best of our knowledge, NetReduce is the first solution that is compatible with the existing transport layer, especially RDMA.

DT Acceleration with High-Speed Networks. The class of solutions that directly accelerate DT jobs by improving the network throughput can be complementary with NetReduce. For example, GossipGraD [7] uses InfiniBand, MG-WFBP [60] merges smaller messages into a large one, SiP-ML [29] applies optical network to transfer gradients, and Horovod can be accelerated by RDMA [59].

Other Network-centric DT Acceleration Solutions. A class of DT acceleration solutions focuses on the control plane. By optimizing job scheduling and job placement, the infrastructure can carry more jobs and complete them more quickly. NetReduce is orthogonal with this class of solutions. For example, Gavel, and THEMIS schedule jobs to shorten average JCT [41, 46]; Blink, AFS, and PLink allocate network resources (e.g., routing, bandwidth) to jobs [40, 61, 70]; TicTac, ByteScheduler, and MLfabric schedules gradient tensors' order [20, 53, 68].

Other In-Network Computation Solutions. Offloading computation to programmable network devices is an effective approach to accelerate systems. There are system acceleration solutions in the field of storage [27, 28, 30, 39, 75], network functions [5, 6], and data query [4, 36, 45, 67]. And NetReduce offloads gradient aggregation specifically.

8 CONCLUSION

We built NetReduce, a transport transparent INA primitive for distributed training. NetReduce provides a ring abstraction to DT jobs, and workers establish RDMA connections along the ring. NetReduce implements an FPGA accelerator attached to Ethernet switches. The accelerator provides a connection-preserving in-network aggregation. We also devised parallel all-reduce with NetReduce to efficiently utilize the inter- and intra- machine bandwidth. Our prototype and evaluation demonstrated the feasibility, low CPU overhead, high throughput, low latency, scalability, and cost-effectiveness of NetReduce.

ACKNOWLEDGMENTS

We thank our shepherd and all the anonymous reviewers for their helpful feedback. Part of the evaluation in this work is done by using the resources of Peng Cheng Laboratory.

A COMMUNICATION TIME MODELING

Modeling. Using the symbols in Table 4, we model the communication time in FR, TA, and PN. In the multi-GPU multi-machine scenario, the communication time taken by using the flat ring all-reduce algorithm is modeled as

$$T_{fr} = 2(P-1)\alpha + 2\frac{P-1}{P}\frac{M}{B_{inter}} \quad (1)$$

where B_{inter} refers to the inter-machine bandwidth where machines are connected via computer networks such as Ethernet or InfiniBand.

For Tencent all-reduce, consider Rabenseifner's reduce algorithm [54] and Van de Geijn's broadcast algorithm [2], and assume n is a power of 2, the communication cost can be modeled as

$$\begin{aligned} T_{ta} &= T_{ta1} + T_{ta2} + T_{ta3} \\ &= \left[2\alpha\log_2(n) + \frac{2(n-1)}{n}\frac{M}{nB_{intra}}\right] \\ &\quad + \left[2(\frac{P}{n}-1)\alpha + 2\frac{P/n-1}{P/n}\frac{M}{B_{inter}}\right] \\ &\quad + \left[(\log_2(n)+n-1)\alpha + 2\frac{n-1}{n}\frac{M}{B_{intra}}\right] \\ &= \frac{n^2+3n\log_2(n)-3n+2P}{n}\alpha \\ &\quad + \frac{4(n-1)PB_{inter}+2(P-n)nB_{intra}}{nPB_{intra}B_{inter}}M \end{aligned} \quad (2)$$

where B_{intra} refers to the intra-machine bandwidth where GPUs are connected via expansion bus such as PCIe or NVLinks.

The communication cost of parallel NetReduce is given as

$$\begin{aligned} T_{pn} &= T_{pn1} + T_{pn2} + T_{pn3} \\ &= \left[(n-1)\alpha + (n-1)\frac{M}{nB_{intra}}\right] + \left(\alpha + \frac{M}{B_{inter}}\right) \\ &\quad + \left[(n-1)\alpha + (n-1)\frac{M}{nB_{intra}}\right] \\ &= (2n-1)\alpha + \frac{2(n-1)B_{inter}+nB_{intra}}{nB_{intra}B_{inter}}M \end{aligned} \quad (3)$$

When $n = 1$, $B_{intra} = B_{inter} = B$, Eq.(3) reduces to the single-GPU case as $T_{inet} = \alpha + \frac{M}{B}$.

Comparison. Eq.(2) subtracting Eq.(3) gives

$$\begin{aligned} \Delta T_{ta-pn} &= T_{ta} - T_{pn} \\ &= (2P/n + 3\log_2(n) - n - 2)\alpha \\ &\quad + \frac{(P-2n)nB_{intra}+2(n-1)PB_{inter}}{nPB_{intra}B_{inter}}M \end{aligned} \quad (4)$$

When $P > 3n$, (4) is always larger than 0, considering n is usually no larger than 16.

Eq.(1) subtracting Eq.(3) gives

$$\begin{aligned} \Delta T_{fr-pn} &= T_{fr} - T_{pn} \\ &= (2P - 2n - 1)\alpha \\ &\quad + \frac{(P-2)nB_{intra}-2(n-1)PB_{inter}}{nPB_{intra}B_{inter}}M \end{aligned} \quad (5)$$

Similarly, we can obtain a relaxed sufficient condition from (5) that parallel NetReduce outperforms flat ring all-reduce on communication as follows

$$\frac{B_{intra}}{B_{inter}} \geq \frac{2P}{P-2} \qquad (P > n \geq 2) \quad (6)$$

We get the sufficient conditions where PN outperforms FR and TA: $P > 3n$ and $\frac{B_{intra}}{B_{inter}} \geq \frac{2P}{P-2}$ $(P > n \geq 2)$. In a production network, the first is not hard to achieve, e.g., our testbed has $P = 32$ and $n = 8$; and the latter can be achieved with the recent progress of intra-machine GPU inter-connection: NVLink makes B_{intra} >=100 GB/s and typical high-speed Ethernet is B_{inter} = 100 Gbps.

REFERENCES

[1] Barefoot. 2019. TOFINO: World's fastest P4-programmable Ethernet switch ASICs. (2019). https://barefootnetworks.com/products/brief-tofino/.

[2] Mike Barnett, Lance Shuler, Robert van De Geijn, Satya Gupta, David G Payne, and Jerrell Watts. 1994. Interprocessor collective communication library (InterCom). In *Proceedings of IEEE Scalable High Performance Computing Conference*. IEEE, 357–364. https://ieeexplore.ieee.org/abstract/document/296665.

[3] Theophilus A Benson. 2019. In-network compute: Considered armed and dangerous. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 216–224.

[4] Li Chen, Ge Chen, Justinas Lingys, and Kai Chen. 2018. Programmable switch as a parallel computing device. *arXiv preprint arXiv:1803.01491* (2018).

[5] Xiang Chen, Qun Huang, Peiqiao Wang, Zili Meng, Hongyan Liu, Yuxin Chen, Dong Zhang, Haifeng Zhou, Boyang Zhou, and Chunming Wu. 2021. LightNF: Simplifying Network Function Offloading in Programmable Networks. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 1–10.

[6] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. 2017. AppSwitch: Application-Layer Load Balancing within a Software Switch. In *Proceedings of the First Asia-Pacific Workshop on Networking* (Hong Kong, China) *(APNet'17)*. Association for Computing Machinery, New York, NY, USA, 64–70. https://doi.org/10.1145/3106989.3106998

[7] Jeff Daily, Abhinav Vishnu, Charles Siegel, Thomas Warfel, and Vinay Amatya. 2018. Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent. *arXiv preprint arXiv:1803.05880* (2018). https://arxiv.org/pdf/1803.05880.pdf.

[8] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255. https://www.image-net.org/papers/imagenet_cvpr09.pdf.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018). https://arxiv.org/abs/1810.04805.

[11] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 11 (2012), 1471–1480.

[12] Yaozu Dong, Zhao Yu, and Greg Rose. 2008. SR-IOV Networking in Xen: Architecture, Design and Implementation.. In *Workshop on I/O Virtualization*, Vol. 2.

[13] Jiawei Fei, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient Sparse Collective Communication and its application to Accelerate Distributed Deep Learning. In *Proceedings of SIGCOMM*.

[14] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. 2021. In-network Aggregation for Shared Machine Learning Clusters. *Proceedings of Machine Learning and Systems* 3 (2021), 829–844.

[15] Jinkun Geng, Dan Li, and Shuai Wang. 2019. Rima: an RDMA-accelerated model-parallelized solution to large-scale matrix factorization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 100–111.

[16] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. 2016. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. IEEE, 1–10. https://ieeexplore.ieee.org/abstract/document/7830486/.

[17] Richard L Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. 2020. Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) Streaming-Aggregation

Hardware Design and Evaluation. In *International Conference on High Performance Computing*. Springer, 41–59. https://link.springer.com/chapter/10.1007/978-3-030-50743-5_3.

[18] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity Ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 202–215. https://dl.acm.org/doi/pdf/10.1145/2934872.2934908.

[19] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A software NIC to augment hardware. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155* (2015).

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

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778. http://openaccess.thecvf.com/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf.

[22] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. 2020. MasQ: RDMA for Virtual Private Cloud *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3387514.3405849 https://doi.org/10.1145/3387514.3405849.

[23] Huggingface. 2020. Transformers:State-of-the-art Natural Language Processing for PyTorch and TensorFlow 2.0. (2020). https://github.com/huggingface/transformers.

[24] Sylvain Jeaugey. 2017. NCCL 2.0. (2017). http://on-demand.gputechconf.com/gtc/2017/presentation/s7155-jeaugey-nccl.pdf.

[25] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, and Mengxian Chi. 2018. Improving the performance of distributed tensorflow with RDMA. *International Journal of Parallel Programming* 46, 4 (2018), 674–685.

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

[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[28] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[29] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. 2021. SiP-ML: high-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 657–675.

[30] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 297–312. https://doi.org/10.1145/3230543.3230572

[31] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 113–126. https://www.usenix.org/conference/nsdi19/presentation/kim.

[32] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 996–1009.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[34] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, et al. 2019. PicNIC: predictable virtualized NIC. In *Proceedings of the ACM Special Interest Group on Data Communication*. 351–366.

[35] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 741–761. https://www.usenix.org/conference/nsdi21/presentation/lao.

[36] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. 2019. The Case for Network Accelerated Query Processing.. In *CIDR*.

[37] Mingfan Li, Ke Wen, Han Lin, Xu Jin, Zheng Wu, Hong An, and Mengxian Chi. 2019. Improving the performance of distributed mxnet with rdma. *International Journal of Parallel Programming* 47, 3 (2019), 467–480.

[38] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating distributed reinforcement learning with in-switch computing. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 279–291. https://ieeexplore.ieee.org/abstract/document/8980345.

[39] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 143–157. https://www.usenix.org/conference/fast19/presentation/liu

[40] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. 2020. PLink: Discovering and Exploiting Datacenter Network Locality for Efficient Cloud-based Distributed Training. *Proc. of MLSys* (2020).

[41] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient {GPU} cluster scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 289–304.

[42] Mellanox. 2022. ConnectX-5 EN Single/Dual-Port Adapter Supporting 100Gb/s Ethernet. (2022). https://www.mellanox.com/products/ethernet-adapters/connectx-5-en.

[43] Mellanox. 2022. InfiniBand Switch Silicon: Mellanox Quantum. (2022). https://www.mellanox.com/products/infiniband-switches-ic/quantum.

[44] Jeffrey C Mogul. 2003. TCP Offload Is a Dumb Idea Whose Time Has Come.. In *HotOS*. 25–30.

[45] Craig Mustard, Fabian Ruffy, Anny Gakhokidze, Ivan Beschastnikh, and Alexandra Fedorova. 2019. Jumpgate: In-network processing as a service for data analytics. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[46] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 481–498.

[47] NVIDIA. 2017. NVIDIA DGX-1 with Tesla V100 System Architecture. (2017). https://www.nvidia.com/en-us/data-center/resources/dgx-1-system-architecture-whitepaper/.

[48] NVIDIA. 2019. NCCL: Optimized primitives for collective multi-GPU communication. (2019). https://github.com/NVIDIA/nccl.

[49] NVIDIA. 2019. NVIDIA NVLink Fabric. (2019). https://www.nvidia.com/en-sg/data-center/nvlink/.

[50] NVIDIA. 2020. NVIDIA V100: The First Tensor Core GPU. (2020). https://www.nvidia.com/en-sg/data-center/v100/.

[51] NVIDIA. 2021. NVIDIA Collective Communication Library (NCCL). (2021). https://developer.nvidia.com/nccl.

[52] NVIDIA. 2023. GeForce RTX 2080. (2023). https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080/.

[53] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29. https://dl.acm.org/doi/abs/10.1145/3341301.3359642.

[54] Rolf Rabenseifner. 1997. A new optimized MPI reduce algorithm. (1997). https://fs.hlrs.de/projects/par/mpi//myreduce.html.

[55] Alec Radford, Jeffrey Wu, Dario Amodei, Daniela Amodei, Jack Clark, Miles Brundage, and Ilya Sutskever. 2019. Better language models and their implications. *OpenAI Blog* (2019). https://openai.com/blog/better-language-models.

[56] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQUAD: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016). https://arxiv.org/abs/1606.05250.

[57] Yufei Ren, Xingbo Wu, Li Zhang, Yandong Wang, Wei Zhang, Zijun Wang, Michel Hack, and Song Jiang. 2017. irdma: Efficient use of rdma in distributed deep learning systems. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 231–238.

[58] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio.

[59] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018). https://arxiv.org/pdf/1802.05799.

[60] Shaohuai Shi, Xiaowen Chu, and Bo Li. 2019. MG-WFBP: Efficient data communication for distributed synchronous SGD algorithms. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 172–180. https://arxiv.org/pdf/1811.11141.pdf.

[61] Jinwoo Shin and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. (2021).

[62] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053* (2019). https://arxiv.org/abs/1909.08053.

[63] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014). https://arxiv.org/pdf/1409.1556.

[64] Brent E Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. 2021. TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP). In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 61–68.

[65] PyTorch Team. 2023. PyTorch. (2023). https://github.com/pytorch/pytorch.

[66] TensorFlow. 2019. A benchmark framework for Tensorflow. (2019). https://github.com/tensorflow/benchmarks.

[67] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2407–2422.

[68] Raajay Viswanathan, Arjun Balasubramanian, and Aditya Akella. 2020. Network-accelerated distributed machine learning for multi-tenant settings. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 447–461.

[69] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018). https://arxiv.org/abs/1804.07461.

[70] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. 2019. Blink: Fast and generic collectives for distributed ml. *arXiv preprint arXiv:1910.04940* (2019).

[71] Xilinx. 2023. Virtex UltraScale - Xilinx. (2023). https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html#productAdvantages.

[72] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2019. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[73] Weihong Yang, Yang Qin, Zukai Jiang, and Xiaowen Chu. 2021. Traffic Management for Distributed Machine Learning in RDMA-enabled Data Center Networks. In *ICC 2021-IEEE International Conference on Communications*. IEEE, 1–6.

[74] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. 2022. Unlocking the Power of Inline Floating-Point Operations on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*.

[75] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 376–389. https://doi.org/10.14778/3368289.3368301

[76] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2785956.2787484 https://doi.org/10.1145/2785956.2787484.

Received 2022-10-20; accepted 2023-01-19