Scalable Distributed Training of Recommendation Models: An ASTRA-SIM + NS3 case-study with TCP/IP transport

Saeed Rashidi*, Pallavi Shurpali†, Srinivas Sridharan†, Naader Hassani†, Dheevatsa Mudigere†,
Krishnakumar Nair†, Misha Smelyanski† and Tushar Krishna*
*Georgia Institute of Technology, Atlanta, USA
†Facebook, Menlo Park, USA
saeed.rashidi@gatech.edu, ssrinivas@fb.com, tushar@ece.gatech.edu

***Abstract*—Recommendation model DNNs have gained significant attention due to their vital role in recommending the best content to the user. However, in order to further increase accuracy, DNNs are becoming more complex with more data to be trained, making them infeasible for training on a single node. Distributed training is a solution to tackle this problem by employing multiple nodes for training. The importance of recommendation models necessitates to design customized HW/SW platforms for training such networks in order to minimize the communication overheads among different nodes. However, exploring this design space is difficult due to the presence of many HW/SW parameters and the limitations to change the HW parameters in real systems.**

In this paper, we port the previously proposed ASTRA-SIM simulation platform on top of the versatile NS3 network simulator by introducing a portable network interface for ASTRA-SIM. Using NS3 enables modeling a wide variety of networks with much better accuracy. Furthermore, we enhance NS3 with detailed modeling of TCP/IP.

Finally, we study various HW/SW platforms for the DLRM recommendation model with TCP/IP as the network protocol and analyze the communication overheads in the presence of various interconnect configurations.

***Keywords*-distributed training, collective communication, recommendation models**

I. Introduction

Neural network-based recommendation models [4], [7], [13] have recently emerged as an important class of Deep Learning (DL) algorithms. These models are used extensively in ranking and click through rate (CTR) prediction tasks - a major contributor to revenue across numerous online services. Inputs to recommendation models comprise of both dense and sparse features. The dense or the continuous features are processed with multilayer perceptron (MLPs) while the sparse or the categorical features are processed using embeddings. A single embedding table contains tens of millions of vectors, each with hundreds of elements, requiring significant memory capacity, on the order of GBs. Therefore, training of recommendation models often requires the distribution of the model across multiple devices using a combination of data parallelism for MLPs and model parallelism for embedding tables [7]. In short, recommendation models differ significantly from MLP heavy Computer Vision and Natural Language Processing workloads, and require re-thinking how we design distributed training systems at scale [8].

This work is focused on addressing the challenges involved in designing highly scalable DL training platforms for recommendation models through SW/HW co-design. We use the MLPerf Deep Learning Recommendation Model (DLRM) [7] benchmark as a representative recommendation model workload. We make the following contributions.

- First, to enable exploration among a wide range of HW/SW platforms, we develop a comprehensive simulation methodology by combining the ASTRA-SIM[1] [10] training simulator with NS3 [12], a powerful network simulator capable of modeling data-center switches and NICs accurately. We accomplish this by introducing a portable network API that enables ASTRA-SIM to support different network simulators on the backend. ASTRA-SIM implements topology-aware collective algorithms and different parallelism approaches for training. It provides a high-level interface to the user to define new DNN models and helps simulate distributed training on different network topologies. On the other hand, NS3 enables modeling a wide variety of network hierarchies.
- We extend NS3 to include detailed modeling of TCP/IP which serves as the transport layer for distributed training in our evaluations.
- We capture the DLRM workload in the ASTRA-SIM + NS3 framework and quantify the impact of various SW (e.g. collective algorithms, levels of concurrency, chunk sizes) and HW choices (e.g. flat vs. hierarchical topology, size of switch buffer) on end workload performance for a 128 GPU system[2].
- We demonstrate the importance of simulation methodology as a tool for design of efficient DL training platforms that: (i) enables accurate modeling compared to the simple and inaccurate analytical models, and (ii) addresses the limitations of real system measurements.
- From our studies, we observe that for DLRM, the MLP

[1] https://github.com/astra-sim/astra-sim
[2] We use the term node and GPU device interchangeably.

Table I: Different parallelism approach communications

Parallelism	Activations during the forward pass	Weight gradients	Input gradients
Data		✓	
Model	✓		✓
Hybrid	partially	partially	partially

layer communication (allreduce) is better handled on hierarchical topology, while embedding communication (all-to-all) is better handled on flat topology. We also show how the combination of physical topology+collective algorith, changes the buffer requirements of network switches. For instance, with TCP running over a hierarchical topology, increasing buffer size from 64 MB to 1 GB reduces time per iteration time by 22.9×.

The rest of the paper is organized as follows: Section II provides background on distributed training. Section III describes the proposed simulation methodology followed by the experimental results in Section IV. Section V describes related work. We conclude the paper in Section VI.

II. Background on Distributed Training

When it comes to parallelizing the training task across multiple nodes (CPU/GPU/TPU), two main questions arise: (i) how to synchronize the weight updates? (ii) how to distribute the parameters (e.g., training data, model parameters) across different nodes? The most common approach used to address the first question is called synchronous training, where each node works on its own data and produces its local gradients, which are then accumulated/reduced across all or a certain number of nodes to update the weights before the next iteration can start.

The answer to the second question depends on the parallelization strategy employed. The common parallelization techniques for partitioning work across multiple nodes are data parallelism (replicating the entire model), model parallelism (splitting the model), pipelined parallelism, or some combination of these. In data parallel, each node is assigned a subset of samples and during each iteration, works on its minibatch (chosen from its own dataset) to produce the local gradients. In model parallel, the nodes have the same datasets and work on the same minibatch, but since the model is divided, each model is responsible for a portion of model gradients. In hybrid parallel, nodes are divided into different groups and the training within the group is data parallel/model parallel while between groups is model parallel/data parallel. The different parallelism approaches have different indications in terms of communication patterns between the nodes. Table I shows when data should be exchanged for different parallelism approaches [10].

As Table I indicates, different communications are initiated at different phases for different parallelism approaches. All these communications are handled by using some set of collective communication operations described in Fig. 1. Libraries like NCCL [9] provide efficient topology-aware

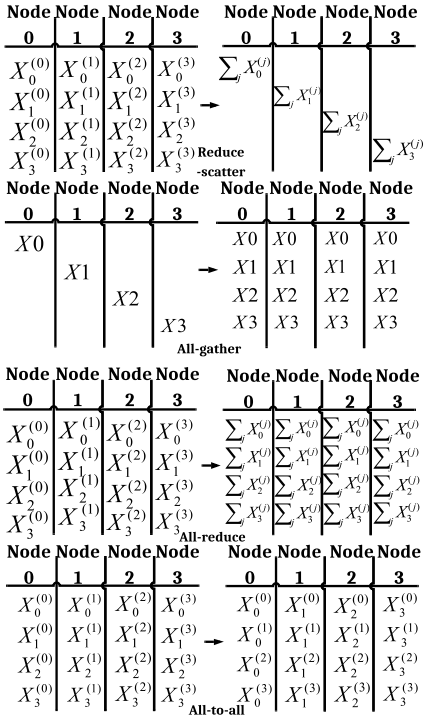

Figure 1: Overview of collective communication operation used in DNN training networks.

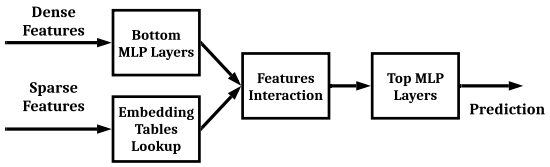

Figure 2: High level architecture of DLRM Model

implementations for these collectives for modern NVIDIA GPU-based training platforms.

III. Simulation Methodology: ASTRA-sim + NS3

A. Target Workload

For the real workload analysis section of this paper we mainly focus on the DLRM model. Fig. 2 shows the high-level architecture of the DLRM model. THe input features are divided into dense and sparse partitions where dense features are fed into the stack of Multilayer perceptron (MLP) layers called bottom MLP. Sparse features are used to look up the embedding tables and the output result, along with the output of bottom MLP, are interacted with each other (e.g. dot-product, concatenate) to generate the inputs for the top MLP layers. Finally, the output of the top MLP layers predicts the probability of a certain action (e.g. the probability that user x likes advertisement y).

Fig. 3 shows how the DLRM distributed training flow works and how it overlaps between the compute and communication. The forward pass begins by embedding

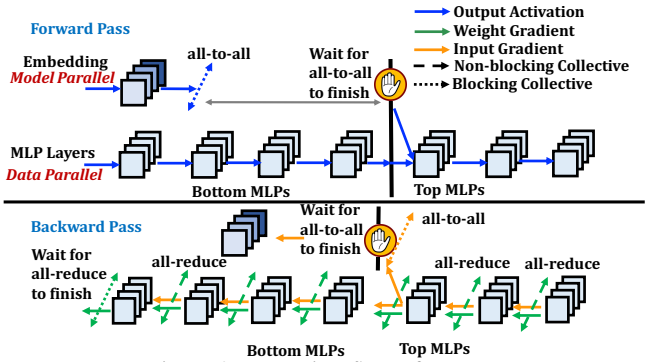

Figure 3: Execution flow of DLRM

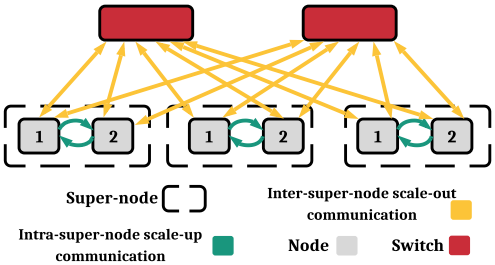

Figure 4: The AllToAll based training system topology

lookup followed by all-to-all communication. In parallel with all-to-all, the forward pass computation is started for bottom MLPs[3]. The forward pass for top MLP layers is done ONLY after the all-to-all is finished. During back propagation, the weight gradient communications of MLP layers are overlapped with the computation of input and weight gradients of next layers. As back propagation proceeds, it flows through both embedding and bottom MLP branches. In the embedding branch, it generates a blocking all-to-all, followed by embedding update. Bottom MLP branch flow is the same as top MLP and at end it stops at the first MLP layer, waiting to start the next forward pass after the first layer communication and embedding update are finished.

B. Target Training Platforms

Fig. 4 describes the target AllToAll based topology we use in our analysis that is similar to Facebook's Zion [2] and NVIDIA's DGX-2 [11] systems. Such systems are suitable for training recommendation models with all-to-all collective on their critical path [7]. It consists of supernodes where each supernode consists of one or multiple compute nodes (e.g. GPUs). There is a network within each supernode that enables local communicates (e.g. through NVLink) between the nodes belonging to the same supernode (i.e. local/scale-up dimension). Supernodes communicate (e.g. through TCP) with each other through global switches (i.e. alltoall/scale-out dimension). In our terminology, **flat topology** has only one node per supernode (local dimension=1) while **hierarchical topology** has more than one node per supernode (local dimension>1).

[3]Note that the forward pass for each layer is proceed ONLY after making sure that the weight update corresponding to the previous training iteration is finished.

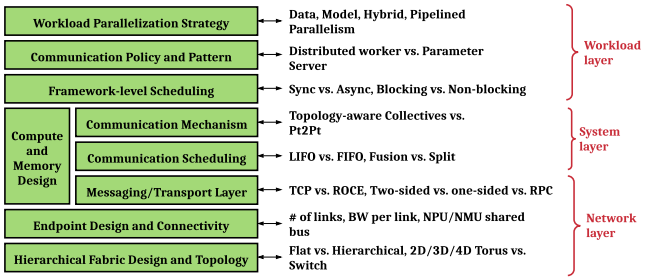

Figure 5: Training stack

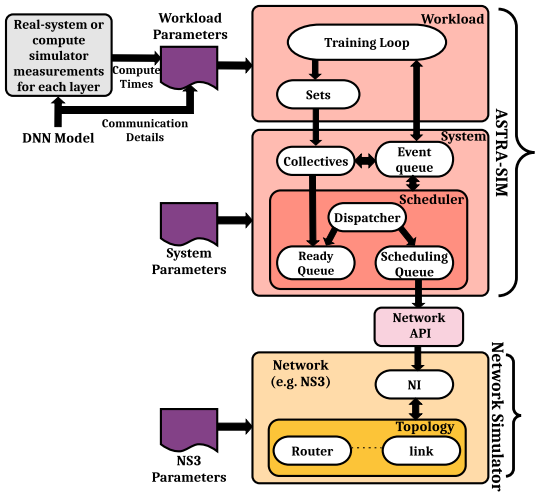

Figure 6: Simulator Architecture

Table II describes the terms used in this paper. This table describes the physical topologies and collective communication algorithms used for these topologies. In addition, it describes the hierarchy of links that form such topologies.

Topologies Under Test. We simulated two systems, both with 128 nodes:

- A flat system where all the nodes are connected via a single switch.
- A hierarchical system consisting of 16 super-nodes, each with 8 nodes. The 8 nodes within a super node are interconnected with a local NV switch equivalent. Each of the 128 nodes are also interconnected with a single switch.

C. Simulation Methodology

The design space of the large-scale deep learning systems can be organized along the layering stack shown in Fig. 5. Designing an efficient training platform requires consideration of the model type and parameters, logical communication algorithms (e.g. ring, tree, etc), pipelining, concurrency and scheduling mechanisms, congestion management/avoidance schemes, lossless versus lossy network protocols, physical network topology and hierarchy, link speeds, buffering and flow-control management, and so on.

Table II: Terminology

Term	Meaning
Flat Topology	A physical topology where all nodes are only connected to the global switches.
Hierarchical Topology	A physical topology where the nodes within a super-node locally communicate (e.g. via a local switch or rings) and they are also connected to nodes in other super nodes via the global switches.
AllToAll (aka A2A)	A family of collective communication algorithms.In all-reduce, The i'th node is responsible to globally reduce i'th segment, so all nodes send their i'th segment to node i for reduction (reduce-scatter phase). After reduction, each node simply broadcasts its reduced segment to all other nodes (all-gather), resulting in all-reduce. Another example of A2A is the personalized all-to-all operation (see description below in this table). To perform an A2A collective, all nodes simply exchange all of their data simultaneously and in a single step. Due to its nature, it is mostly suitable on switch based or fully connected network.
DBT	Double Binary Tree algorithm for performing an all-reduce collective. Two trees with roles of leafs and parents swapped. In each tree, data travels from 2 children to their parent which locally reduces the data. Parent then propagates this reduced data upward until we get to the root. The root will have the complete reduced data (reduce phase). The root then broadcast the reduced data down to its two children and they in turn, broadcast to their children until all leafs get the reduced data (broadcast phase).
all-to-all	We use this term to refer to "Personalized all-to-all" that is used to exchange embedding parameters between nodes. Usually, A2A algorithm is used to do this.
all-to-all hierarchical	Same as "personalized all to all" except all communication from one super node to the i'th node in any other super node happens thru that super node's i'th node.
Hierarchical all-reduce	All nodes in a super node do a full all-reduce. Then i'th nodes in all super nodes do all reduce together.
Hierarchical Optimal all-reduce (aka 3-phase all-reduce)	Nodes within a super node do reduce scatter, then i'th nodes in all super nodes do full all-reduce, then nodes within a super node do all gather.
Scale-up link	A link connecting a node to any other node within a super node
Scale-out link	A link connecting a node in one super node to a node in another super node. In a flat system, all links are scale-outs.

Fig. 6 shows the high-level structure of our simulator to model this stack. Our simulation methodology consists of three components:

- A front-end workload simulator, called ASTRA-SIM [10], that simulates the: (i) workload, the training loop that carries out various high-level parallelization strategies and (ii) the system software that provides different collective communication primitives and manages the concurrency and pipelining across different communication tasks.
- A back-end simulator built on top of NS3 [12] that simulates the transport protocol, the network hardware, and the physical topology. The back-end is customized to include much more communication protocols and more accurate modeling and it replaces the ASTRA-SIM default network simulator (i.e. Garnet [1]). We adopted NS3 as our back-end simulator due its capability/extendibility to model various levels of networks including the network of data-center platforms accurately [5].
- We develop a new *"Network API"* that is meant to provide decoupling and portability of the aforementioned two components. It provides a logical two-sided communication interface (e.g. send, receive) to the front-end while the actual physical implementation of the interface is back-end specific and depends on how the physical communication protocol is modeled inside the network simulator.

In order to maintain time causality between the two distinct pieces of software (ASTRA-sim workload layer and NS3), we treat NS3's event scheduler as a master and require the workload layer to schedule events in NS3's event queue while allowing the front-end to have its local event-queue for its internal events.

This simulation framework provides the ability to easily test the effects of changing hardware and software parameters and tune them for next generation training platforms. We focus on sync training with data parallelism for MLP and model parallelism for embedding tables in this paper — we will focus on other aspects like async training in future work.

*1) **Modeling Scheduling, Pipelining, and Concurrency**:* The scheduling discipline for the simulations is Last-In-First-Out (LIFO). That means that a later issued collective will take precedence over the earlier ones at the system layer. There is however one exception that are all-to-all collectives, since they are on the critical path in our training model, they get the highest priority among all collectives. In the network layer, the LIFO discipline is not the case (yet).

We define "chunk" as the smallest unit of data on which the various schedulers operate in the system layer. A chunk inherits its parent collective's priority. All chunks of a given collective have the same priority regardless of the phase they belong to. For instance, in a hierarchical optimal all-reduce, there are 3 phases: reduce scatter on local dimension, all reduce on global dimension, and all gather on local dimension. We apply priority at the collective level. The scheduling discipline we have used in the simulations here is as follows:

- All chunks of a collective inherit the priority of the parent collective
- Between the all-reduce collective, the scheduling in LIFO meaning all chunks of layer N-1 all-reduce take priority over all chunks of layer N all reduce (with the exception of the chunks at the head of the queue which are already committed to execution)
- The all-to-all collective get the highest priority. The back prop and forward all-to-alls have equal priority.

Chunking effectively enables pipelining as the network, DRAM, and compute functions operate on each chunk. We allow up to 64 concurrent chunks per dimension per node total to be outstanding at any given time.

*2) **Network Modeling and Approximations**:* We tune the network protocol differently for scale-up and scale-out network for accurate modeling.

Scale-up Network: We approximate the NV switch scale-up network within each supernode in a hierarchical topology with a generic switch model running TCP within NS3. We make sure that the TCP and switch parameters are tuned to eliminate packet drops in this scale-up network while utilizing the link fully. Across the supernodes (i.e., scale-out), regular TCP (with packet drops) is simulated. In the local dimension, we use 7 switches to interconnect the 8 nodes of a super node. We use this formula for load balancing traffic from a node: nvSwitch_id = (src_id + dst_id) mod 7 where src_id and dst_id are in (0..7) range.

Scale-out Network: We use "injection latency" as a proxy for memory copy effects: Injection latency is the time it takes for unit of data (i.e. a chunk) from network interface (NIC) to reach the compute and vice versa (e.g. for local reduction in the all-reduce).

We lump the TCP Ack delay into the link latency parameter. As such, the link latency includes both the light travel time on link as well as TCP turnaround time due to traversing the linux stack.

We use a simple generic switch model with following attributes:

- Total buffer is divided evenly among the ports
- Main buffering and queueing point is at the egress port
- Tail drop policy is implemented when an egress queue is completely full (i.e. the new packet would have overrun the queue)

We will replace these approximations with proper models in the follow-on work.

Table III: Design parameters

Parameter	Flat	Hierarchical
Scale-up link BW	N/A	1200 Gb/s (total)
Scale-up link latency	N/A	0.5us
Scale-up link communication protocol	N/A	TCP
NV Switch buffer size	N/A	64 MB
Scale-out link communication protocol	TCP	TCP
Allreduce algorithm	DBT (scale-out)	A2A (scale-up) + DBT (scale-out)
Alltoall algorithm	all-to-all	all-to-all hierarchical

D. Design Parameters

Table III shows the design parameters. Table IV shows the experiments setup and the parameters that are changed during our experiments. Table V describes DLRM we use which is a scaled version of MLPerf's [6] DLRM model to stress both compute and communications.

Table IV: Experiments setup

Variable Parameters	Default Value	Range
Global Batch Size	65536	
GPU Model	V100	
Number of GPUs	128	
Compute Power Per GPU	60 TFLOPS	
Per GPU batch size	512	
Scale-out link BW	100 Gb/s	{100, 800}
Scale-out switch buffer	1GB	{64MB, 1024MB}
Link latency + TCP Ack delay	0.5us	
Injection latency	0.01usec	{0.01, 1, 10, 100}
TCP window size	0.5MB	{0.5, 0.1}
all-reduce algorithm	DBT	{DBT, A2A}
Number of chunks	1024 (flat), 128 (hierarchical)	{1024, 128, 32}
Chunk parallelism	64	{64, 128}
Collective Scheduling	LIFO	

Table V: DLRM Model parameters

Model Parameters	Value	Unit
Size of embedding data-type	16	bits
Pooling factor	60	
Top MLP layers	10+2	
Bottom MLP layers	5+2	
Dense features	1600	
Top MLP layer size	2048	
Bottom MLP layer size	1024	
Sparse features	64	
Embedding dimension	64	

IV. Simulation Results

A. Simulation vs. Analytical Model for Single All-Reduce

The intent of this experiment is to use a single all-reduce collective to familiarize readers with the design concepts and simulation results, and to demonstrate how the front-end and back-end stats are used to explain observed behaviors. In addition, this experiment illustrates the gap that can exist between the results from a linear analytical model (that ignores the network congestion effects) and those from a detailed network simulator (as much as 20x in the example below).

We use a flat topology with 128 nodes with the following parameters:

- The flat physical topology consists of 128 NICs interconnected via a single switch
- Each NIC has a single 100 Gbps link to switch
- The transport protocol is TCP with 0.5 MB congestion window
- The switch has 64 MB, 1 GB of buffering space evenly divided amongst the 128 ports
- The algorithms used are DBT, AllToAll
- We vary the size of the matrix (in MB) to be reduced: 8, 16, 32, 64, 128, 256

1) Analytical Model: We know that the total amount of data that a node sends and receives for one all-reduce operation is roughly 2×D where D is the size of the matrix to be reduced. For instance, if D=8MB, and link speed =

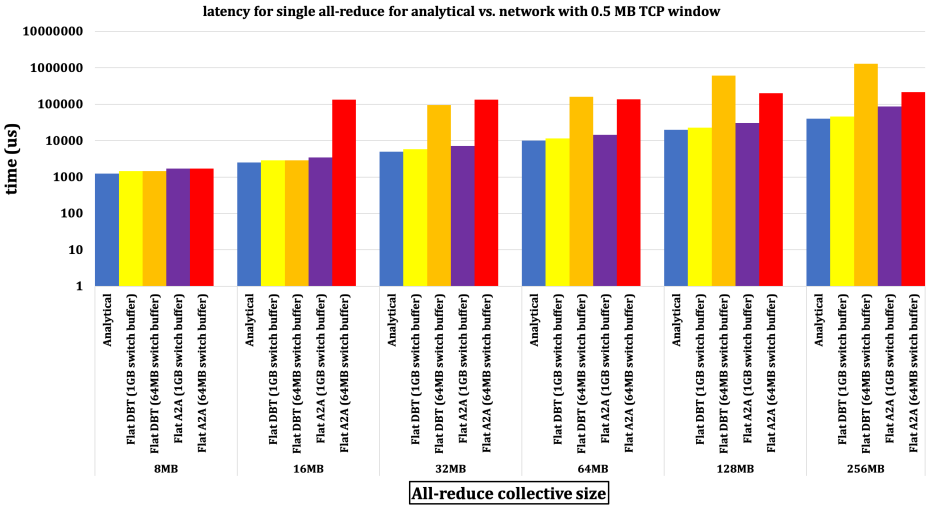

Figure 7: Total raw latency for a single all-reduce

100 Gbps, then the time it takes to transfer this much data on the link will be 2×8 MB / 100 Gbps = 1.28 msec. This is called the insertion delay. To this delay, we need to add the transit delay which includes, among other things, the switch queuing delay. The queuing delay is variable and depends on the buffer build up in the switch. For instance, if an egress queue has 512 KB buffering capacity and it is completely full, and the port is being drained at 100 Gbps, then the transit time thru this queue will be 512 KB / 100 Gbps = 40 μsec. Since in the analytical analysis we have no idea what the queuing delay might be (anywhere between 0 and 40 μsec in our example), we typically ignore it.

2) Network Simulator: Now let us study the results from our ASTRA-sim + NS3 simulator. Fig. 7 illustrates the completion times, and deviation from the analytical model for a single all-reduce for different test configurations. The total latency stats are produced by the front-end ASTRA-sim.

The network stats for this experiment, produced by the back-end NS3 simulator are presented next. Fig. 8 shows the average link utilization across all ports for the duration of 2 iterations on the left axis, and the total number of RTOs (TCP retransmission timeout) events on the right axis. Fig. 9 shows the average packet round trip time (RTT) for percentile 50 and percentile 99 of packets. The RTT value for each flow is sampled every 1 ms of simulation time. These RTT samples are averaged per flow (i.e. source/destination pair of nodes) at the end of the simulation. These per-flow averages are used to create the average, p50, and p99 plots. Fig. 10 shows the average packet RTT and total number of retransmitted packets across all flows.

3) Discussion: In order to show the difference between the simulator results and the analytical model, we make few observations based on Fig. 7 and further elaborate them using network simulator results in the explanations.

Observation 1. Consider "Flat A2A (64 MB switch buffer)": For 8 MB collective size, the difference between the

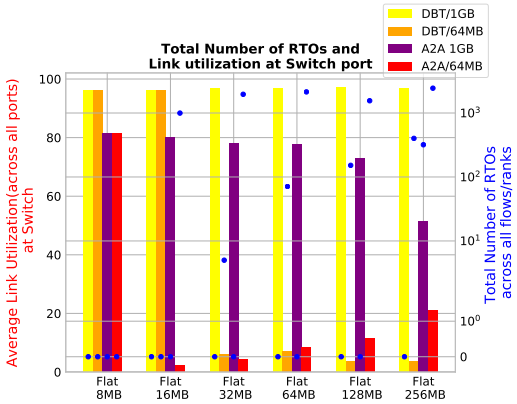

Figure 8: Average link utilization and total number of packet retransmission timeout (RTO) corresponding to the systems simulated in Fig. 7

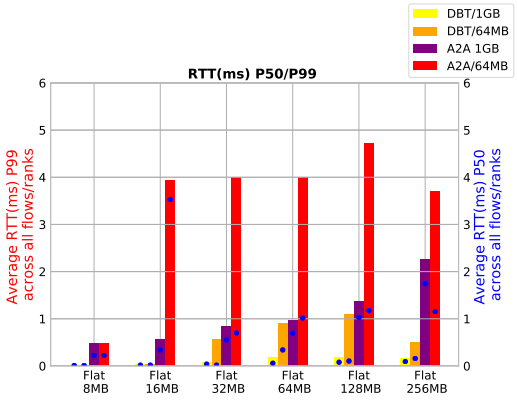

Figure 9: Average packet round trip time (RTT) for percentile 50 and percentile 99 of packets corresponding to the systems simulated in Fig. 7

analytical model (bytes/link_speed) and the network simulator results is around 37%.

Explanation. The average link utilization was 81% (Fig. 8) and the total number of retransmission events was 907

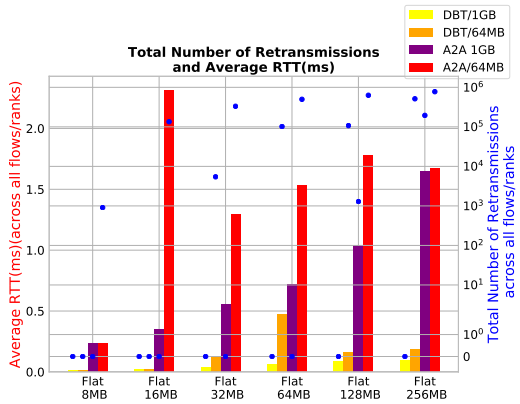

Figure 10: Average packet round trip time (RTT) and total number of retransmitted packets corresponding to the systems simulated in Fig. 7

(Fig. 10). There were no RTO events. This explains the 37% deviation from the analytical model prediction.

In addition, based on Fig. 7, when going from 8 MB to 16 MB size, the difference between the linear model and the network simulator jumps to over 50x. The reason is that at 16 MB size, there are 997 RTO events, the link utilization collapses (Fig. 8), and both the number of retransmissions and the average RTT shoot up (Fig. 9 and Fig. 10). The RTO timer was set to 50 ms.

Observation 2. As data size is increased, the small switch buffer configurations severely deviate from the analytical model with DBT being more optimal for smaller collectives and A2A being more optimal for larger collectives.

Explanation. This observation is supported by the link utilization and RTO event count (Fig. 8). The real interesting observation here is that the DBT algorithm is much more susceptible to packet drops and RTO events than the A2A one. Note that in the 32 MB collective size case, a handful of RTO events (5 to be exact) bring down the average link utilization in the DBT configuration, while the A2A counterpart has 2000 RTO events with slightly less average link utilization. To explain this, one needs to consider the synchronous nature of the tree algorithm where the entire collective could be at mercy of a single point to point transfer. The parent node in the tree that fails to receive the data from one of its children will stall the rest of the tree and if this occurs close to the leaf layer, the subsequent layers of the tree are stalled.

Observation 3. For all collective sizes up to 256 MB, the 1GB switch buffer configurations stay reasonably close to the analytical model with DBT being better than the A2A. The DBT with 1GB buffer stays within 16% of the analytical for all collective sizes.

Explanation. Note that DBT configurations maintain close to 100% link utilization and zero retransmits/RTOs across all collective sizes (Fig. 8). This should not be surprising because of the organized nature of the tree and the 2*log(N) sequential communication steps. In contrast, in the A2A all nodes simultaneously blast their data into the switch. The retransmit events occur more frequently as collective size increases, and the link utilization drops significantly (to 52%) when the RTO events occur at 256MB collective size.

4) Conclusions: The intent of this section was to dissect the simple case and demonstrate key concepts and tools and show why simulation is necessary for accurate modeling. However, additionally there are some takeaway messages from the previous simulations that are briefly presented here.

Granted, we are simulating a lossy TCP network here, but if we consider the 1GB buffer cases in the zero drop regime (8, 16, 32, 64 MB collectives), it is evident that the DBT algorithm outperforms the A2A both in terms of collective completion time and link utilization for the all-reduce collective. This indicates that the sequential and network friendly nature (e.g. much lower in-cast effect) of the DBT outweighs its multi-step aspect.

On the other hand, when packet drops do occur in a lossy network, the A2A is much more resilient than the DBT. The reason for this is the way the DBT algorithm moves in lock step. The tree cannot make further progress until data from both children are received by parent and reduced.

Next, we investigate the various system/network configurations when running real workloads. In all the experiments of subsequent sections, we simulated 2 iterations of the DLRM loop. All parameters except for the variable under study are set to the default values specified in Table III, and Table IV. The largest weight matrix is for top MLP layer 0 = 16.2 MB. The second largest matrices are for the top MLP layers = 8 MB. In the first experiment, we show the per-layer and the aggregate stats but in the subsequent experiments, for brevity, we only show the aggregate compute versus exposed communication.

B. DLRM Experiment 1: Effect of Physical Topology

We fix all parameter to default and focus on comparison between 4 systems hierarchical (hier in short), hierarchicalOptimal (hierOpt in short), flat100, flat800. The switch buffer size is set to 1GB. The 2 flavors of the flat systems refer to the scale-out link bandwidth (i.e. 100 Gbps vs. 800 Gbps) while in both hierarchical and hierarchicalOptimal the scale-out link bandwidth is 100 Gbps (refer to Table II for more information regarding each configuration).

1) Raw Latencies: Fig. 11 shows the layer-wise raw collective communication time. The raw communication latency is measured from the time collective is created until it is completely finished. The collectives of different layers can run in parallel and are mostly overlapped with computation (discussed later).

Observation 1: The LIFO Scheduling Effect. As can be seen in Fig. 11, the all-reduce operation for the last layer of the top MLP (layer 11) completes in very short time. One reason is the smaller communication size for this layer. Additionally, it practically encounters an empty network as

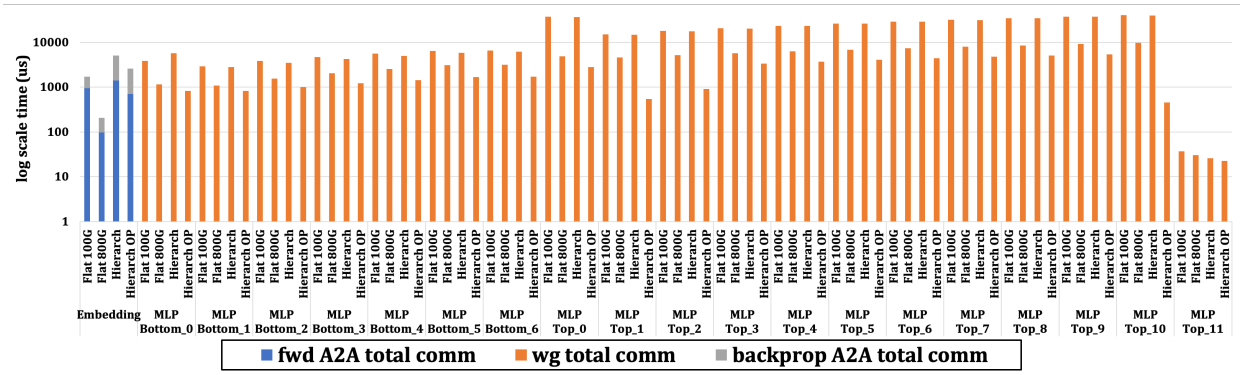

Figure 11: Total raw communication delay for two training iterations of DLRM

it is the very first collective that is issued and it is finished before the arrival of the next collectives.

Moreover, for layers 10, 9, ..., 1 — all with equal weight matrix size — since the later issued collectives (e.g. top layer 1) are inferred with the earlier issued ones (e.g. top layer 10), LIFO scheduling makes the later issued collectives to complete faster.

Observation 2: The MLP Size. In total, the top MLP layer 0 has the largest compute time and communication size. The reason is it has a large input dimension, creating a large weight matrix. Due to its high computation latency, most of the previous top layer communications are already finished before the collective of this layer is issued. This explains the lower raw communication latency of many other top layers in the presence of LIFO scheduling.

Observation 3: Forward and Back Prop All-to-all. Looking at the embedding part in Fig. 11, we note that the flat800G is the best topology for all to all type collective. Flat performs all-to-all in a single phase while hierarchical needs 2 steps one for each dimension (see the Table II for all-to-all hierarchical). Also note that all-to-all hierarchical does not reduce the total traffic injected into the scale-out dimension, hence in this case flat shows better performance. Back prop all-to-all takes longer than forward pass all-to-all, since back prop all-to-all sees more in-execution chunks ahead of itself (and hence more queuing delay) compared to forward pass all-to-all.

Observation 4: All-reduce Latency: In terms of all-reduce latency hierOpt is working the best. Compared to flat, in hierarchical less number of steps are needed to finish a collective, since trees in scale-out dimension are shallower in hierarchical (4 in hierarchical vs. 7 in flat) and the A2A in local dimension only requires 2 steps (one for reduce-scatter and one for all-gather). HierarchicalOptimal also reduces the amount of data to be sent to the scale-out dimension, resulting in better all-reduce performance.

2) EndToEnd Latency: Fig. 11 shows the layer-wise compute times and exposed communication times. Exposed communication latency is the amount of latency that the algorithm is forced to stop because it is waiting for that communication to be finished. This is the *effective* communication latency that actually increases the training time. So just a portion of raw communication latency will translate into exposed latency and the rest will be overlapped with compute times.

Observation 5: Top MLP Layer 0. Recall that Fig. 11 showed huge raw latency in completion time of MLP Layer 0 communication. Fig. 12 shows that when we arrive at the Top MLP layer 0 in the current iteration, we have to stall because the weight gradient update from the back prop stage of the previous iteration has not completed. Hence the "exposed communication" is large. This is due to Top 0 large size, LIFO scheduling, and the strict priority of the all-to-alls.

Observation 6: All-to-all Collective. Looking at the Embedding section, we note again, how the flat800G has no exposed all-to-alls. For the other systems, we note that both the back prop and forward all-to-alls have different degrees of exposure:

- Back prop all-to-all takes longer than the partially overlapping back prop bottom MLP
- Forward all-to-all takes longer than the partially overlapping forward bottom MLP

3) Total exposed communication latency to compute latency (per training iteration): Fig. 13 shows the ratio of exposed communication to total computation time per training iteration. Our goal should be minimizing the amount of exposed communication as it means that our compute resources would be idle. This diagram conveys the information in the previous figure in an aggregate way: Sum of all exposed communications across all layers versus sum of all compute times across all layers in the previous figure would yield the figure below. As mentioned before, the exposed communication time is the time during which the compute is idle only because it is waiting for network operations to complete.

Observation 7: Comparison of Different Configurations. The net performance delta between an optimal topology (flat800G or hierOpt) and a not-so-optimal one (flat100G, hier) is around 3x. More specifically, flat800G reduces the

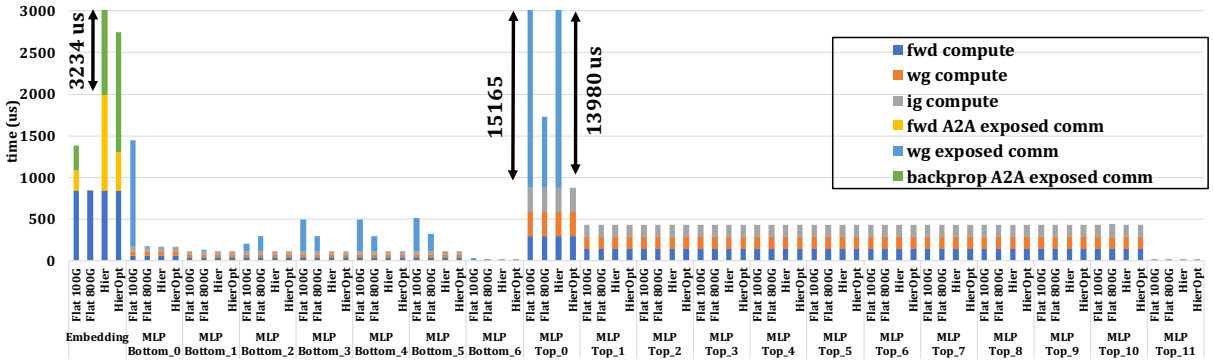

Figure 12: Total end-to-end latency (compute+exposed comm) latency for 2 iterations

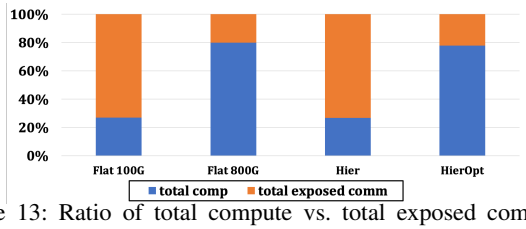

Figure 13: Ratio of total compute vs. total exposed comm per iteration

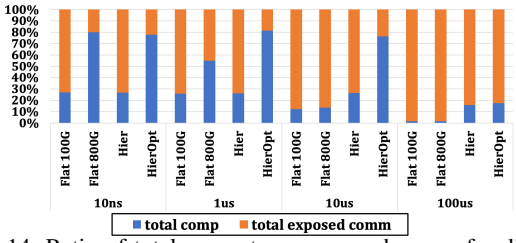

Figure 14: Ratio of total compute vs. exposed comm for different injection latencies

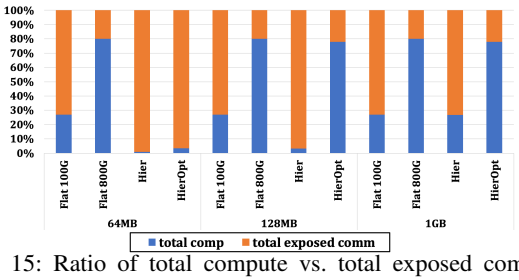

Figure 15: Ratio of total compute vs. total exposed comm for different global switch buffer size

percentage of exposed communication from 73.1% in the case of flat100G to 20%, resulting in 2.97× better iteration time. On the other hand, hierOpt reduces the percentage of exposed communication from 73.2% in the case of hier to 22.1%, making the iteration time 2.91× better.

Observation 8: Optimal Configurations Tradeoff. The two optimal topologies — flat800G and hierOpt — represent the tradeoff between network capacity and algorithm complexity. The flat800G needs 8x more network capacity in the global (scale-out) dimension but the communication algorithm (DBT) has a single phase (simpler to implement). On the other hand, the hierOpt achieves same performance with 100 Gbps scale out links, but the 3-phase all-reduce algorithm is more complex as it requires multiple phases (reduce scatter local, all reduce global, all gather local).

C. DLRM Experiment 2: Effect of memory copies

As stated earlier, we model the network-to-compute latency with a fixed parameter called "injection latency". This is the latency required by the node to process and handle each message it receives from other nodes. Most of this delay corresponds to modeling the effect of memory copies (e.g. (i) writing to receiver node's memory address and (ii) then reading again doing local reduction and then (iii) sending the message out to the network again). In this experiment, we fix all parameters and show the effect of injection latency 10 ns, 1 μs, 10 μs, 100 μs for 4 topologies hier100, hierOpt100, flat100, flat800. The switch buffer size is set to 1 GB. Fig. 14 shows the ratio of exposed communication to total computation time per training iteration.

Observation 9: Tolerance to Injection Latency. As the injection latency increases, the flat topologies performance degrades far more than the hierarchical ones. The main reason is that flat all-reduce requires more steps compared to hierarchical topology (14 steps vs. 10). Hence, increased node response has more severe effect in hierarchical. When injection latency is increased from 10 ns to 100 μs, flat800G compute ratio is reduced from 79.9% to 1.5%, while HierarchOpt's compute ratio is reduced from 77.8% to 17.5%. This results in degradation of overall iteration time by around 50× and 4.53× for flat800G and HierarchOpt, respectively.

D. DLRM Experiment 3: Effect of the switch buffer size

In this experiment, we vary the buffer size of the switch in the global dimension (the scale-out). We fix all parameters and show effect of switch buffer 64 MB, 128 MB, 1 GB for 4 topologies hier100, hierOpt100, flat100, flat800. Fig. 15 shows the ratio of exposed communication to total computation time per training iteration.

Observation 10: Switch Buffer Size Requirements. Note that both flat100G and flat800G systems are insensitive to switch buffer size. Recall that the largest weight matrix is 16 MB in the model we simulated. The single all-reduce

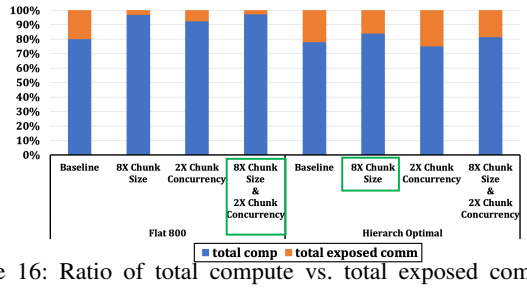

Figure 16: Ratio of total compute vs. total exposed comm for different scheduling policies

results earlier also indicated that flat100G with DBT shows completion time within 16% of the analytical model for both 64 MB and 1 GB buffer sizes for 8 MB and 16 MB collectives (with no packet drops and close to 100% link utilization). It is logical that the flat800G system would also be insensitive to switch buffer size. In the case of HierarchOpt, increasing the global switch buffer size from 64 MB to 1 GB increases the ratio of compute time from 3.4% to 77.8%, enhancing the iteration time by 22.9×. The main reason for this enhancement is that compared to flat, hierarchical topology execute on multiple shallower depth DBTs that inject more traffic-per-time compared to single in-depth DBT of flat, hence, requiring more buffer size.

E. DLRM Experiment 4: Effect of Pipelining and Scheduling

In the last experiment, we explore the effects of optimizations in the system layer of the front-end simulator. We vary the chunk size and degree of the concurrency for flat800G and hierOpt topologies. The switch buffer size is set to 1 GB. Fig. 16 shows the ratio of exposed communication to total computation time per training iteration.

Observation 11: Optimal Concurrency Degree. There is as much as 8.89× variation in exposed communication time (21.5% variation in total iteration time) for the flat800G topology and 1.54× difference in exposed communication time (7.4% variation in total iteration time) for the hierOpt as we change the degree of chunking and concurrency. This variation is primarily due to moving towards and away from the optimal point of operation between the link utilization and degree of network congestion. As long as network is not too congested, we would like to increase the link utilization to reduce the training iteration time. However, as the network becomes more and more congested, increasing the link utilization will have the inverse effect as retransmission and RTO events come into play. Varying the chunk size and degree of concurrency effectively control the offered load into the network.

V. Related Work

Due to growing importance of training, several works studied the training design space exploration on various platforms during the recent years. For instance, authors in [3] and [8] explored the design choices for training of ResNet-50 and DLRM networks, respectively. However, since their methodology relies on real systems, their hardware parameter exploration is limited to whatever hardware is available. The original ASTRA-SIM paper [10] studied the impact of topology and algorithm for Resnet-50 using the ASTRA-SIM frontend with the GARNET network simulator [1] as the backend. However, unlike this work, the analysis was limited to scale-up networks. Also, as described in this paper, Resnet-50 has very different characteristics compared to recommendation models such as DLRM.

VI. Conclusion

Our work is focused on addressing the challenges involved in designing highly scalable DL training platforms for recommendation models through SW/HW co-design. We presented an end-to-end simulation infrastructure and evaluated the impact of Hierarchical vs. Flat topologies for scaling DLRM. Further, we studied the impact of various SW (e.g. collective algorithms, levels of concurrency, chunk sizes) and HW choices (e.g. size of switch buffer) on end workload performance. We plan to extend this work to larger system sizes and different transport protocols (e.g. RDMA over Converged Ethernet) as part of future work.

References

[1] N. Agarwal *et al.*, "Garnet: A detailed on-chip network model inside a full-system simulator," in *ISPASS*, 2009. [Online]. Available: http://synergy.ece.gatech.edu/tools/garnet/

[2] K. Arnold, "Application-specific hardware accelerators," Mar 2019. [Online]. Available: https://engineering.fb.com/data-center-engineering/accelerating-infrastructure/

[3] J. Dong *et al.*, "Eflops: Algorithm and system co-design for a high performance distributed training platform," in *HPCA*, 2020.

[4] Y. Gong *et al.*, "Hashtag recommendation using attention-based convolutional neural network," in *IJCAI*, 2016.

[5] A. R. A. Kumar, S. V. Rao, and D. Goswami, "Ns3 simulator for a study of data center networks," in *ISPDC*, 2013.

[6] P. Mattson *et al.*, "Mlperf training benchmark," *arXiv preprint arXiv:1910.01500*, 2019.

[7] M. Naumov *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[8] ——, "Deep learning training in facebook data centers: Design of scale-up and scale-out systems," *arXiv preprint arXiv:2003.09518*, 2020.

[9] NVIDIA, "Nvidia collective communications library (nccl)," 2018. [Online]. Available: https://developer.nvidia.com/nccl

[10] S. Rashidi *et al.*, "ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms," in *ISPASS*, 2020. [Online]. Available: https://github.com/astra-sim/astra-sim

[11] Y. Ren, S. Yoo, and A. Hoisie, "Performance analysis of deep learning workloads on leading-edge systems," 2019. [Online]. Available: http://arxiv.org/abs/1905.08764

[12] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*. Springer Berlin Heidelberg, 2010, pp. 15–34.

[13] C. Wu, J. Wang, J. Liu, and W. Liu, "Recurrent neural network based recommendation for time heterogeneous feedback," *Knowledge-Based Systems*, vol. 109, pp. 90 – 103, 2016.