TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs

Weiyang Wang[*] *Moein Khazraee*[*] *Zhizhen Zhong*[*] *Manya Ghobadi*[*]
Zhihao Jia[†,‡] *Dheevatsa Mudigere*[†] *Ying Zhang*[†] *Anthony Kewitsch*[§]

[*]*Massachusetts Institute of Technology* [†]*Meta* [‡]*CMU* [§]*Telescent*

Abstract

We propose TopoOpt, a novel direct-connect fabric for deep neural network (DNN) training workloads. TopoOpt co-optimizes the distributed training process across three dimensions: computation, communication, and network topology. We demonstrate the mutability of AllReduce traffic, and leverage this property to construct efficient network topologies for DNN training jobs. TopoOpt then uses an alternating optimization technique and a group theory-inspired algorithm called TotientPerms to find the best network topology and routing plan, together with a parallelization strategy. We build a fully functional 12-node direct-connect prototype with remote direct memory access (RDMA) forwarding at 100 Gbps. Large-scale simulations on real distributed training models show that compared to similar-cost Fat-tree interconnects, TopoOpt reduces DNN training time by up to 3.4×.

1 Introduction

Our society is rapidly becoming reliant on deep neural networks (DNNs). New datasets and models are invented frequently, increasing the memory and computational requirements for training. This explosive growth has created an urgent demand for efficient distributed DNN training systems.

Today's DNN training systems are built on top of traditional datacenter clusters, with electrical packet switches arranged in a multi-tier Fat-tree topology [47]. Fat-tree topologies are traffic-oblivious fabrics, allowing uniform bandwidth and latency between server pairs. They are ideal when the workload is unpredictable and consists mostly of short transfers–two inherent properties of legacy datacenter workloads [49, 50, 54, 67, 68]. But Fat-tree networks are becoming a bottleneck for distributed DNN training workloads [58, 69, 77, 85, 102, 105, 136].

Previous work has addressed this challenge by reducing the size of parameters to transmit through the network [48, 58, 59, 69, 73, 79, 82, 83, 94, 105, 123, 139] and developing techniques to discover faster parallelization strategies while considering the available network bandwidth [46, 48, 85, 105, 129]. These proposals co-optimize computation and communication as two important dimensions of distributed DNN training, but they do not consider the *physical layer topology* as an optimization dimension.

In this paper, we analyze DNN training jobs from production clusters of Meta. We demonstrate that training workloads do not satisfy common assumptions about datacenter traffic that underlie the design of Fat-tree interconnects. Specifically, we show that (*i*) the communication overhead of large DNN training jobs increases dramatically as we increase the number of workers; and (*ii*) the traffic pattern of a DNN training job depends on its parallelization strategies.

Motivated by these observations, we propose TopoOpt, a direct-connect DNN training system that co-optimizes network topology and parallelization strategy. TopoOpt creates dedicated partitions for each training job using reconfigurable optical switches and patch panels, and jointly optimizes the topology and parallelization strategy within each partition. To achieve our goal, we grapple with the *algorithmic* challenges of finding the best topology, such as how to navigate the large search space across computation, communication, and topology dimensions, and also with various *operational* challenges, such as which optical switching technologies match well with the traffic patterns of DNN models.

We cast the topology and parallelization strategy co-optimization problem as an off-line alternating optimization framework. Our optimization technique *alternates* between optimizing the parallelization strategy and optimizing the network topology. It searches over the parallelization strategy space assuming a fixed topology, and feeds the traffic demand to a TopologyFinder algorithm. The updated topology is then fed back into the parallelization strategy search algorithm. This alternating process repeats until the system converges to an optimized parallelization strategy and topology.

We demonstrate that finding an optimized network topology for DNNs is challenging because the ideal network topology needs to meet two goals simultaneously: (*i*) to complete large AllReduce transfers efficiently, and (*ii*) to ensure a small hop-count for Model Parallel transfers. To meet these goals, we propose a novel *group theory-based technique*, called TotientPerms, that exploits the *mutability* of AllReduce transfers. Our TotientPerms approach builds a series of *AllReduce permutations* that not only carry AllReduce transfers efficiently, but are also well-positioned to carry Model Parallel transfers and, hence, improve the overall training performance.

Optical circuit-switched networks traditionally support point-to-point traffic across hosts with direct circuits between them. As a result, for a given set of circuits, only directly connected hosts can communicate leaving the rest of the hosts wait for new circuits to be established. To support arbitrary communication across all hosts participating in a job, we enable TOPOOPT's hosts to act as relays and forward the traffic that does not belong to them. Host-based forwarding introduces a new challenge for RDMA flows since RDMA NICs drop packets that do not belong to them. To enable host-based RDMA forwarding, we exploit the network partition (NPAR) function of modern NICs, creating an efficient logical overlay network for RDMA packet forwarding (§6).

To evaluate TOPOOPT, we build a 12-server prototype with NVIDIA A100 GPUs [37], 100 Gbps NICs and a Telescent reconfigurable optical patch panel [43]. Moreover, we integrate our TotientPerms AllReduce permutations into NCCL and enable it to load-balance parameter synchronization across multiple ring-AllReduce sub-topologies. Our evaluations with six representative DNN models (DLRM [20], CANDLE [4], BERT [134], NCF [75], ResNet50 [74], and VGG [126]) show that TOPOOPT reduces the training iteration time by up to 3.4× compared to a similar-cost Fat-tree. Moreover, we demonstrate that TOPOOPT is, on average, 3.2× cheaper than an ideal full bisection bandwidth Fat-tree. TOPOOPT is the first system that co-optimizes topology and parallelization strategy for ML workloads and is currently being evaluated for deployment at Meta. The source code and scripts of TOPOOPT are available at https://topoopt.csail.mit.edu.

2 Motivation

Prior research has illustrated that *demand-aware* network fabrics are flexible and cost-efficient solutions for building efficient datacenter-scale networks [64, 68, 113]. However, predicting the upcoming traffic distribution is challenging in a traditional datacenter setting. This section demonstrates that DNN training workloads present a unique opportunity for demand-aware networks, as the jobs are long-lasting, and the traffic distribution can be *pre-computed* before the jobs start to run. First, we provide the necessary background to understand distributed DNN training and introduce three types of data dependencies between accelerator nodes in training jobs (§2.1). Then, we present measurements from production clusters in Meta (§2.2), and discuss the important properties of DNN training traffic.

2.1 Background on Distributed DNN training

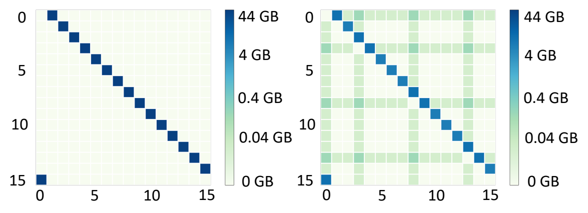
Training iteration. A common approach to training DNNs is stochastic gradient descent (SGD) [90]. Each SGD *iteration* involves selecting a random batch of labeled training data, computing the error of the model with respect to the labeled data, and calculating gradients for the model's weights through backpropagation. The SGD algorithm seeks to update the model weights so that the next evaluation reduces the error [55]. Training iterations are repeated with new batch of data until the model converges to the target accuracy.

Data parallelism. Data parallelism is a popular parallelization strategy, whereby a batch of training samples is distributed across training accelerators. Each accelerator holds a replica of the DNN model and executes the forward and backpropagation steps locally. In data parallelism, all accelerators synchronize their model weights during each training iteration. This step is commonly referred to as *AllReduce* and can be performed using various techniques, such as broadcasting [141], parameter servers [93], ring-AllReduce [3, 83, 130], tree-AllReduce [116], or hierarchical ring-AllReduce [131, 133].

Hybrid data and model parallelism. Large DNN models cannot fit in the memory of a single accelerator or even a single server with multiple accelerators. As a result, the model needs to be divided across multiple accelerators using *model parallelism* [84, 92]. Moreover, pure data parallelism is becoming suboptimal for large training jobs because of the increasing cost of synchronizing model parameters across accelerators [20, 78, 85, 104, 106, 125]. As a result, large DNNs are distributed using a hybrid of data and model parallelism, where different parts of a DNN and its dataset are processed on different accelerators in parallel.

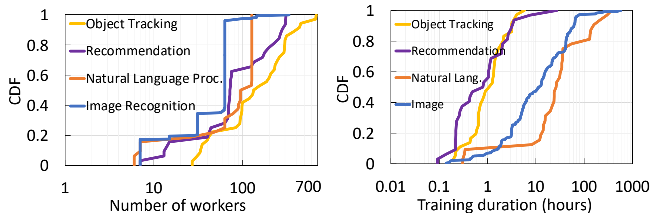
Types of data dependencies in DNN training. Each training iteration includes two major types of *data dependencies*. Type (1) refers to *activations* and *gradients* computed during the Forward and Backpropagation steps. This data dependency is required for each input sample. Type (2) refers to synchronizing the *model weights* across accelerators through the AllReduce step once a batch of samples is processed. Depending on the parallelization strategy, these data dependencies may result in local memory accesses or cross-accelerator traffic. For instance, in a hybrid data and model parallelization strategy, type (1) and (2) both result in cross-accelerator traffic, depending on how the model is distributed across accelerators. Given that type (1) is related to model parallelism, we refer to the network traffic created by type (1) as *MP transfers*. Similarly, we refer to the network traffic created by type (2) as *AllReduce transfers*. Note that AllReduce transfers do not strictly mean data parallelism traffic, as model parallelism can also create AllReduce transfers across a subset of nodes.

Example: DLRM traffic pattern. Deep Learning Recommendation Models (DLRMs) are a family of personalization and recommendation models based on embedding table lookups that capitalize on categorical user data [107]. DLRMs



(a) Data Parallelism (b) Hybrid Parallelism

Figure 1: DLRM traffic heatmaps for different parallelization strategies.



(a) Number of workers (b) Training job duration

Figure 2: Profiling distributed DNN training jobs in Meta.

are large, typically with 100s of billions of parameters, primarily because of their large embedding tables. Using pure data parallelism to distribute a DLRM results in massive AllReduce transfers. For instance, consider a DLRM architecture with four embedding tables $E_0, \cdots, E_3$, each with embedding dimensions of 512 columns and 10^7 rows (total size 22 GB for the model) distributed across 16 servers $S_0, \cdots, S_{15}$ with data parallelism. We compute the resulting traffic distribution, and Figure 1a illustrates the traffic pattern for a single training iteration. The rows and columns indicate source and destination servers, while the color encodes the amount of traffic between server pairs. The heatmap shows that using ring-AllReduce for synchronization, a pure data parallelism strategy results in 44 GB of AllReduce transfers.

Hence, a common parallelization strategy for DLRMs is to use a hybrid of data and model parallelism where the embedding tables are divided across nodes, while the rest of the model is replicated on all nodes [102]. Following the parallelization strategy used at Meta, we place E_0 on S_0, E_1 on S_3, E_2 on S_8, and E_3 on S_{13}, and replicate the rest of the model on all servers. This parallelization strategy creates a mix of MP and AllReduce traffic, shown in Figure 1b. It reduces the maximum transfer size from 44 GB to 4 GB.

Note that MP transfers in DLRM form one-to-many broadcast and many-to-one incast patterns to transfer the activation and gradients to all nodes because the servers handling embedding tables must communicate with *all other servers*. In this example, the size of each AllReduce transfer is 4 GB, whereas the size of MP transfers is 32 MB, as shown by darker green elements in the heatmap.

2.2 Production Measurements

We study traffic traces from hundreds of production DNN training jobs running on multiple clusters at Meta. We instrument each job to log its training duration, number of workers, and the total amount of data transferred across its workers during training.

Number of workers and job duration. Figure 2a shows the cumulative distribution function (CDF) of the number of workers for different models in Meta's clusters. Most jobs are distributed across 32 to 700 workers, agreeing with recent announcements by other major players in the industry [45, 104], where each worker is a single GPU. Figure 2b demonstrates the CDF of total training job duration; as the figure shows, most jobs last over 10 hours. In fact, the top 10% of jobs take more than 96 hours (four days) to finish. This measurement shows production DNN jobs at Meta are long-lasting, and take up to weeks to finish.

Network overhead. Figure 3 illustrates the percentage of network overhead as the number of GPUs is increased from 8 to 128 for six DNN jobs in production. We use RDMA to transmit packets between servers and measure the percentage of time consumed by communication during training as network overhead. The figure shows that as the number of GPUs increases, the network quickly takes up a significant portion of training iteration time. In fact, the network overhead accounts for up to 60% of a DNN training iteration time in Meta's production environment. Similar observations have been made in prior work [59, 77, 89, 105, 110, 123]. Such bottleneck suggests the existing datacenter networks are insufficient for the emerging DNN training workloads.

Traffic heatmaps. Figure 4 shows the heatmap of server-to-server traffic for four training jobs running in Meta's production GPU clusters. The values on the colormap and the exact names of DNN models are not shown for confidentiality reasons. All heatmaps in the figure contain diagonal squares (in dark blue), indicating a ring communication pattern between servers. This is expected, as ring-AllReduce is the common AllReduce communication collective at Meta. But the MP transfers (light blue and green squares) are *model-dependent* because MP transfers depend on the parallelization strategy and device placement of a training job. Moreover, we find that the traffic patterns of training jobs do not change between iterations *for the entire training duration*, resulting in the same per-iteration heatmap throughout the training. Once a training job starts, the same parallelization strategy and synchronization method are used across training iterations, resulting in a periodic and predictable traffic pattern. Similar observations have been made in previous work [140]. In particular, the traffic heatmap is identical *across* training iterations. Note that the traffic pattern changes *within* a training iteration during forward, backward, and AllReduce phases.

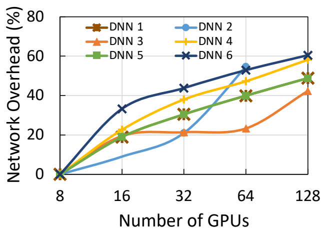


Figure 3: Network overhead measurements in Meta.

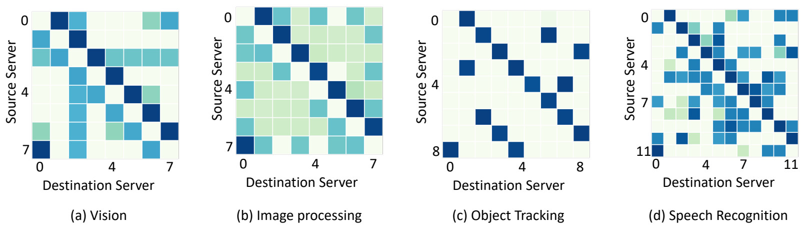


Figure 4: Traffic heatmaps of production jobs in Meta.

3 TopoOpt System Design

The observations in the previous section suggest that demand-aware fabrics are excellent candidates for a DNN training cluster. In this section, we seek to answer the following question: *"Can we build a demand-aware network to best support distributed training?"* To answer this question, we propose TopoOpt, a novel system based on optical devices that jointly optimizes DNN parallelization strategy and topology to accelerate today's training jobs.

TopoOpt interconnect. A TopoOpt cluster is a *shardable* direct-connect fabric where each server has d interfaces connected to a core layer of d optical switches, as shown in Figure 5. The optical switches enable TopoOpt to shard the cluster into dedicated partitions for each training job. The size of each shard depends on the number of servers the job requests. Given a DNN training job and a set of servers, TopoOpt first finds the best parallelization strategy and topology between the servers off-line (§4.1). Then, it reconfigures the optical switches to realize the target topology for the job. Appendix C provides details on how TopoOpt achieves sharding and dynamic job arrivals in shared clusters.

Degree of each server. We denote the number of interfaces on each server (i.e., the degree of the server) by d. Typically, d is the same as the number of NICs installed on the server. In cases where the number of NICs is limited, the degree can be increased using NICs that support break-out cables or the next generation of co-packaged optical NVLinks [11]. In our testbed, we use one 100 Gbps HPE NIC [29] with 4×25 Gbps interfaces to build a system with degree four ($d = 4$).

Direct-connect topology. In TopoOpt, optical switches connect the servers directly, forming a *direct-connect topology*. To further scale a TopoOpt cluster, we create a hierarchical interconnect by placing the servers under Top-of-Rack (ToR) switches and connecting ToR switches to the optical layer, creating a direct-connect topology at the ToR or spine layers, similar to previous work [53, 71, 72, 100, 114].

Host-based forwarding. In DNN training workloads, the degree of each server is typically smaller than the total number of neighbors with whom the server communicates during training. To ensure traffic is not blocked when there is no direct link between two servers, we use a technique called *host-based forwarding*, where hosts act as switches and forward incoming traffic toward the destination. Previous work used similar technique at the ToR switch level [53, 99, 100].

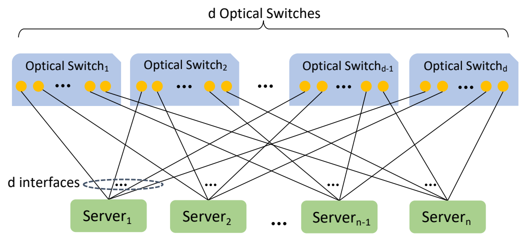


Figure 5: Illustration of TopoOpt's interconnect.

Optical switching technologies. A wide range of optical switches is suitable for TopoOpt, including commodity available optical patch panels [43] and 3D-MEMS [6, 41], as well as futuristic designs such as Mordia [113], MegaSwitch [57], and Sirius [53, 60]. Table 1 lists the characteristics of these devices. TopoOpt is compatible with any of these technologies. Appendix B provides details about these devices.

One-shot reconfiguration. Patch panel and OCS are both applicable for an immediate deployment of TopoOpt, as shown in Table 1. The choice of which technology to use depends on several factors, including scale of the cluster, iteration time of jobs, and frequency of job arrivals. For instance, OCSs can potentially be used to reconfigure the topology of a job *within* training iterations, whereas patch panels are only suitable when the topology remains intact throughout the entire training of a particular job. Our evaluations demonstrate that the reconfiguration latency of today's OCSs is too high for today's DNNs, leading to sub-optimal performance when the topology is reconfigured within iterations (§5). As a result, given that faster technologies are not yet available, TopoOpt uses a one-shot reconfiguration technique based on an offline co-optimization framework (§4) that jointly optimizes the parallelization strategy and topology. TopoOpt then reconfigures the interconnection between training servers of each job before the job starts and keeps the topology intact until the training is complete (or to recover from failures).

Technology	Port-count	Reconfig. latency	Insertion Loss (dB)	Cost /port
Optical Patch Panels [43]	1008	minutes	0.5	$100
3D MEMS [6, 41]	384	10 ms	1.5–2.7	$520
2D MEMS [57, 113]	300	11.5 μs	10–20	Not commercial
Silicon Photonics [89, 122]	256	900 ns	3.7	Not commercial
Tunable Lasers [53, 60]	128	3.8 ns	7-13	Not commercial
RotorNet [99, 100]	64	10 μs	2	Not commercial

Table 1: Comparison of optical switching technologies.

4 Co-optimizing Parallelization Strategy and Network Topology

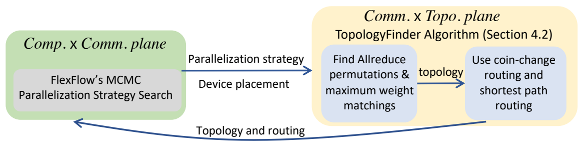
This section describes TopoOpt's co-optimization framework for finding a network topology and parallelization strategy for a given DNN training job.

4.1 Alternating Optimization

The search space is too large. Finding the optimal parallelization strategy alone is an NP-complete problem [85], and adding network topology and routing makes the problem even harder. An extreme solution is to jointly optimize compute, communication, and topology dimensions using a *cross-layer optimization formulation*. Theoretically, this approach finds the optimal solution, but the search space quickly explodes, even at modest scales (e.g., six nodes [129]).

Naive approach. The other extreme is to optimize the network topology *sequentially after* the parallelization strategy has been found. While this approach is able to reconfigure the network to match its traffic demand better, the eventual combination of topology and parallelization strategy is likely to be sub-optimal in the global configuration space.

TopoOpt's approach: alternating optimization. In TopoOpt, we seek to combine the best of both worlds. To make the problem tractable, we divide the search space into two planes: $Comp. \times Comm.$ and $Comm. \times Topo.$ We use an alternating optimization technique to iteratively search in one plane while keeping the result of the other plane constant. Figure 6 illustrates our alternating optimization framework. We use FlexFlow's MCMC (Markov Chain Monte Carlo) search algorithm [85] to find the best parallelization strategy for a given network topology while considering the communication cost. If the parallelization strategy improves the training iteration time, we feed it to the $Comm. \times Topo.$ plane to find the efficient network topology and routing using our TopologyFinder algorithm. The discovered topology is then fed back into the $Comp. \times Comm.$ plane, which further optimizes the parallelization strategy and device placement based on the new topology. This optimization loop repeats until convergence or after k iterations, where k is a configurable hyper-parameter.



Comp. x Comm. plane
FlexFlow's MCMC Parallelization Strategy Search
Parallelization strategy
Device placement
Comm. x Topo. plane
TopologyFinder Algorithm (Section 4.2)
Find Allreduce permutations & maximum weight matchings
topology
Use coin-change routing and shortest path routing
Topology and routing

Figure 6: TopoOpt searches for the best parallelization strategy, jointly with routing, and topology.

4.2 TopologyFinder Algorithm

TopologyFinder steps. Algorithm 1 presents the pseudocode of our TopologyFinder procedure. The algorithm takes the following inputs: n dedicated servers for the training job, each with degree d, as well a list of AllReduce and MP transfers ($T_{AllReduce}$ and T_{MP}) based on the parallelization strategy and device placement obtained from the $Comp. \times Comm.$ plane. The algorithm then finds the best topology (G) and routing rules (R) and returns them to the $Comp. \times Comm.$ plane for the next round of alternating optimization. Our algorithm consists of the following four steps.

Step 1: Distribute the degree. This step distributes the degree d between AllReduce and MP sub-topologies proportionally, based on their share of total traffic. We specifically start with AllReduce transfers and allocate at least one degree to the AllReduce sub-topology to ensure the network remains connected (line 2). The remaining degrees, if any, are allocated to the MP sub-topology (line 3).

Step 2: Construct the AllReduce sub-topology. To find the AllReduce sub-topology, the algorithm iterates over every AllReduce group k and allocates degree d_k to each group proportionally based on the amount of traffic (line 6). Note that in hybrid data and model parallelism strategies, the AllReduce step can be performed across a subset of servers when a DNN layer is replicated across a few servers instead of all servers. To efficiency serve both AllReduce and MP transfers, TopoOpt constructs the AllReduce sub-topology such that the diameter of the cluster is minimized. Section 4.3 explains two algorithms, called `TotientPerms` (line 8) and `SelectPermutations` (line 9) to construct the AllReduce sub-topology.

Step 3: Construct the MP sub-topology. We use the Blossom maximum weight matching algorithm [63] to find the best connectivity between servers with MP transfers (line 14). We repeat the matching algorithm until we run out of degrees. To increase the likelihood of more diverse connectivity across server pairs, we divide the magnitude of T_{MP} for pairs that already have an edge between them by two (line 17). In general, division by two can be replaced by a more sophisticated function with a diminishing return.

Step 4: Final topology and routing. Finally, we combine the MP and AllReduce sub-topologies to obtain the final topology (line 18). We then use a modified version of the coin-change algorithm [52] (details in Appendix E.1) to route

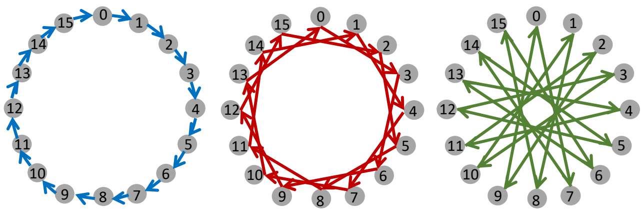
Algorithm 1 TopologyFinder pseudocode

```
1:  procedure TopologyFinder($n$, $d$, $T_{AllReduce}$, $T_{MP}$)
      ▷ Input $n$: Number of dedicated training servers for the job.
      ▷ Input $d$: Degree of each server.
      ▷ Input $T_{AllReduce}$: AllReduce transfers.
      ▷ Input $T_{MP}$: MP transfers.
      ▷ Output $G$: Topology to give back to the Comp. × Comm. plane.
      ▷ Output $R$: Routing rules to give back to the Comp. × Comm. plane.
        ▷ Distribute degree $d$ between AllReduce and MP sub-topologies
2:      $d_A = \max(1, \lceil d \times \frac{sum(T_{reduce})}{sum(T_{reduce})+sum(T_{MP})} \rceil)$
3:      $d_{MP} = d - d_{AllReduce}$
        ▷ Construct the AllReduce sub-topology $G_{AllReduce}$
4:      $G_{AllReduce} = \{\}$
5:      for each AllReduce group $k$ with set of transfers $T_k$ do
            ▷ Assign degree $d_k$ to group $k$ according to its total traffic
6:          $d_k = \lceil d_A \times \frac{sum(T_k)}{sum(T_{reduce})} \rceil$
7:          $d_A = d_A - d_k$
            ▷ Find all the permutations between servers in group $k$
8:          $P_k$ = TotientPerms($n$, $k$)   ▷ (Details in §4.3)
            ▷ Select $d_k$ permutations from $P_k$
9:          $G_{AllReduce} = G_{AllReduce} \cup$ SelectPermutations($n$, $d_k$, $P_k$)   ▷ (§4.3)
10:         if $d_{AllReduce} == 0$ then
11:             break
        ▷ Construct the MP sub-topology $G_{MP}$
12:     $G_{MP} = \{\}$
13:     for $i : i < d_{MP}$ do
            ▷ Find a maximum weight matching according to $T_{MP}$
14:         $g$ = BlossomMaximumWeightMatching($T_{MP}$)
15:         $G_{MP} = G_{MP} \cup g$
            ▷ Reduce the amount of demand for each link $l$ in graph $g$
16:         for $l \in g$ do
17:             $T_{MP}[l] = T_{MP}[l] / 2$
        ▷ Combine the AllReduce and MP topologies
18:     $G = G_{AllReduce} \cup G_{MP}$
        ▷ Compute routes on $G_{AllReduce}$ using the coin change algorithm [52]
19:     $R$ = CoinChangeMod($n$, $G_{AllReduce}$)   ▷ (Appendix §E.1)
        ▷ Compute routes on $G_{MP}$ with shortest path
20:     $R$ += ShortestPath($G$, $T_{MP}$)
21:     return $G, R$
```

AllReduce on the AllReduce sub-topology (line 19). Further, we use k-shortest path routing for the MP transfers to take advantage of the final combined topology (line 20).

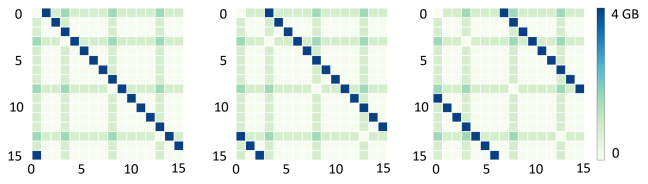
4.3 Traffic Mutability and AllReduce Topology

Finding an efficient AllReduce sub-topology. At first blush, finding an AllReduce sub-topology for a given DNN seems straightforward: we just need to translate the parallelization strategy and device placement from the *Comp.* × *Comm.* plane into a traffic matrix and map the traffic matrix into circuit schedules. Several papers have used this technique for datacenter networks [57, 64, 68, 72, 89, 95–97, 113, 137]. However, the conventional wisdom in prior work is to allocate as many direct parallel links as possible to *elephant flows* and leave *mice flows* to take multiple hops across the network. In principle, this approach works well for datacenters but it leads to sub-optimal topologies for distributed DNN training. While the size of AllReduce transfers is larger than MP transfers, MP transfers have a higher communication degree than AllReduce (Appendix D). Hence, the conventional approach creates parallel direct links for carrying AllReduce traffic and forces MP flows to have a large hop-count, thereby degrading the training performance.



(a) "+1" Permutation (b) "+3" Permutation (c) "+7" Permutation

Figure 7: Ring-AllReduce permutations.



(a) Traffic Heatmap 1 (b) Traffic Heatmap 2 (c) Traffic Heatmap 3

Figure 8: DLRM traffic heatmaps.

TopoOpt's novel technique. In TopoOpt, we seek to meet two goals simultaneously: (*i*) allocate ample bandwidth for AllReduce transfers, as the bulk of the traffic belongs to them, but (*ii*) ensure a small hop-count for MP transfers. We meet both goals by demonstrating a unique property of DNN training traffic – the AllReduce traffic is *mutable*.

Mutability of AllReduce transfers. We define traffic mutability as the ability to change the traffic pattern without altering parallelization strategy or device placement while maintaining correctness, and demonstrate that AllReduce transfers are *mutable* whereas MP transfers are not. Intuitively, this is because MP traffic is composed of network flows among nodes that contain *different* parts of a DNN model thus creating immutable data dependencies, while AllReduce transfers contain network flows among nodes that handle the *same* part of the model, providing flexibility in the order of nodes participating in AllReduce. For instance, consider a DLRM distributed across 16 servers each with three NICs. The common AllReduce pattern is shown as a ring with consecutive node IDs, as shown in Figure 7a. However, *this is not the only possible permutation*. Each heatmap in 8a, 8b, and 8c corresponds to a different ring-AllReduce permutation, shown in Figures 7a, 7b, and 7c. We denote each of these permutations as $+p$, where server S_i connects to server $S_{(i+p)\%n}$, and n is the number of servers, as shown in Figure 7. Although all three heatmaps correspond to the *exact same parallelization strategy and device placement*, the blue diagonal lines appear at different parts of the heatmaps, depending on the order of servers in the ring-AllReduce permutation. But MP transfers (green vertical and horizontal lines in each heatmap) are dictated by the parallelization strategy and device placement; thus, they remain at the same spot in all three heatmaps.

Algorithm 2 `TotientPerms` pseudocode

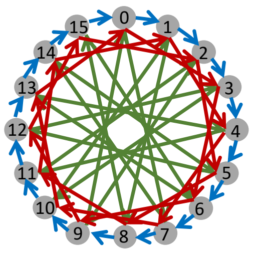
```
1: procedure TOTIENTPERMS($n$, $k$)
     ▷ Input $n$: Total number of nodes
     ▷ Input $k$: AllReduce group size
     ▷ Output $P_k$: Set of permutations for AllReduce group of size $k$
       ▷ Initially, $P_k$ is empty
2:     $P_k = \{\}$
         ▷ This loop runs $\phi(p)$ times, where
         ▷ $\phi$ is the Euler's totient function, $\phi(p) = |\{k < p : gcd(k, p) = 1\}|$

         ▷ one can also restrict $p$ to be prime only
3:     for $p \leq k$, $gcd(p, k) == 1$ do
4:         $one\_perm = []$
5:         for $i$ in 0 to $N/k$ do
6:             $one\_perm$ += $[i + j \times p$ for $j$ in 0 to $k]$
7:         $P_k$ += $one\_perm$
8:     return $P_k$
```

Algorithm 3 `SelectPermutations` pseudocode

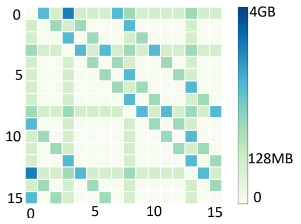
```
1: procedure SELECTPERMUTATIONS($n$, $d_k$, $P_k$)
     ▷ Input $n$: Total number of nodes
     ▷ Input $d_k$: Degree allocated for group this AllReduce group of size $k$
     ▷ Input $P_k$: Candidate permutations for this AllReduce group of size $k$
     ▷ Output $G_k$: Parameter synchronization topology, given as a set of permutations
       ▷ Initially, $G_k$ is empty
2:     $G_k = \{\}$
       ▷ $q$ now is the minimum candidate in $P_k$
3:     $q = P_k[0]$
       ▷ GetConn($q$) gives the connection described
       ▷ by the permutation corresponding to $q$
4:     $G_k = G_k \cup$ GetConn($q$)
       ▷ Ratio of the geometric sequence to fit
5:     $x = \sqrt[d_k]{N}$
6:     for $i \in \{1, \cdots, d_k - 1\}$ do
           ▷ Select the next candidate based on the ratio
7:         $q' = x \times q$
           ▷ Project $q'$ onto $P_k \setminus G_k$ with minimal distance (L1-norm)
8:         $q' = \text{argmin}_{r \in P_k \setminus G_k} |r - q'|$
           ▷ Add this candidate to final topology
9:         $G_k = G_k \cup$ GetConn($q'$)
10:        $q = q'$
11:    return $G_k$
```

Leveraging AllReduce traffic mutability. Traffic mutability implies that if a group of servers is connected in a certain order, simply permuting the label of the servers gives another ordering that will finish the AllReduce operation with the same latency while potentially providing a smaller hop-count for MP transfers. Instead of selecting just one AllReduce order, TopoOpt finds multiple permutations for each AllReduce group and overlaps their corresponding sub-topologies. In doing so, TopoOpt efficiently serves the AllReduce traffic while decreasing the hop-count for MP transfers.

TotientPerms algorithm. While overlapping multiple permutations sounds straightforward, navigating through the set of all possible AllReduce orderings is non-trivial since the number of possible permutations is $O(n!)$. To reduce the search space of all possible permutations, we design the TotientPerms algorithm to find the ring generation rule for all



(a) TopoOpt topology



(b) TopoOpt traffic pattern

Figure 9: TopoOpt's topology and traffic matrix.

regular rings, based on group theory. Regular rings are those where the distance between indices of consecutive servers is equal; i.e., server S_i is connected to server $S_{(i+p)\%n}$ for some p. Algorithm 2 presents the pseudocode of `TotientPerms`. Inspired by Euler's totient function [25], we find all integer numbers $p < n$, where p is co-prime with n (i.e. $gcd(p, n) = 1$, line 3, Algorithm 2), represent a valid ring-AllReduce permutation (§E.1). For instance, for $n = 12$ servers, the ring generation rule for $p = 1, 5, 7, 11$ will lead to four distinct ring-AllReduce permutations between the servers. Note that each p describes a unique regular permutation. To handle large-scale clusters, we restrict p to be a prime number, thereby reducing the search space size to only $O(\frac{n}{\ln n})$, as per the Prime Number Theorem [66].

SelectPermutations algorithm. For a group of n servers participating in AllReduce, `TotientPerms` finds a set of regular permutations $P_k = \cup_{p:gcd(p,n)=1}\{p\}$ across them. TopologyFinder then selects d_k permutations using a module called `SelectPermutations`, where d_k is the number of degree allocated to the group of nodes running AllReduce (line 6, Algorithm 1). Algorithm 3 presents the pseudocode of `SelectPermutations`. Several metrics can be used in the `SelectPermutations` module. In our implementation, `SelectPermutations` aims to reduce the cluster diameter to benefit the MP transfers. To this end, `SelectPermutations` chooses $\{p_1, \cdots, p_{d_k}\} \subset P_k$, such that $\{p_1, \cdots, p_{d_k}\}$ is close (in L1-norm) to a geometric sequence (line 7, Algorithm 3).

Theorem 1. *TopoOpt's `SelectPermutations` algorithm bounds the diameter of the AllReduce sub-topology to $O(d_A \cdot n^{1/d_A})$, under certain assumptions.*

We list the assumptions and proof of Theorem 1 in Appendix E.2. Intuitively, each server in the topology is able to reach a set of servers with a geometrically distributed hop-count distance (line 5, Algorithm 3), creating a topology similar to Chord [128].

Example. Consider the DLRM model in Figure 8. Instead of choosing one of the AllReduce permutations in Figure 7, TopoOpt combines the three ring-AllReduce permutations to load-balance the AllReduce transfers while providing a short hop-count for MP transfers. Figure 9 illustrates TopoOpt's topology and traffic matrix and shows a more balanced traffic matrix than Figure 8.

5 Large Scale Simulations

This section evaluates the performance of a large-scale TopoOpt interconnect. First, we explain our simulation software and methodology (§5.1). Then, we provide a cost analysis of TopoOpt to inform our simulations when comparing different interconnects (§5.2). Next, we demonstrate the performance of TopoOpt when a cluster is dedicated to a single distributed DNN training job (§5.3). We perform a sensitivity analysis to quantify the impact of all-to-all traffic (§5.4) and host-based forwarding (§5.5). We extend this setting to a case where a training cluster is shared among multiple jobs (§5.6). Finally, we evaluate the impact of reconfiguration latency (§5.7) on TopoOpt's performance.

5.1 Methodology & Setup

We implement two simulators to evaluate TopoOpt.

***FlexNet* simulator.** We augment FlexFlow's simulator [27] to be network-aware and call it *FlexNet*. Given a DNN model and a batch size, FlexFlow's simulator explores different parallelization strategies and device placements to minimize iteration training time. The output of this simulator is a *task graph* describing the set of computation and communication tasks on each GPU and their dependencies. The current implementation of FlexFlow ignores the network topology by assuming servers are connected in a *full-mesh* interconnect. Our *FlexNet* simulator extends the FlexFlow simulator and enables it to consider multiple networks, including Fat-trees, TopoOpt, and expander networks. Moreover, *FlexNet* implements our alternating optimization framework (§4) to find an optimized network topology and routing rules for TopoOpt.

***FlexNetPacket* simulator.** FlexFlow's simulator only provides course-grind estimation of training iteration time, because it does not simulate individual packets traversing through a network. Extending *FlexNet* to become a packet-level simulator is computationally infeasible, because FlexFlow generally requires thousands of MCMC iterations to converge. To faithfully simulate per-packet behavior of network switches, buffers, and multiple jobs sharing the same fabric, we build a second event-based packet simulator, called *FlexNetPacket*, on top of htsim [7]. *FlexNetPacket* takes the output of *FlexNet* (i.e., the optimized parallelization strategy, device placement of each operator, network topology, and routing rules) and simulates several training iterations. The link propagation delay is set to 1 μs throughout this section.

Simulated network architectures. We simulate distributed training clusters with n servers equipped with four NVIDIA A100 GPUs [37]. We vary n in different experiments and simulate the following network architectures:

• **TopoOpt.** A TopoOpt interconnect where each server is equipped with d NICs, each with bandwidth B connected via a flat layer of optical devices. At the beginning of each job, a shard of the network is selected, and the topology of the shard is reconfigured based on the output of our alternating optimization framework (§4) and remains unchanged throughout the entire training job. Both OCS and patch panels are suitable for this architecture.
• **OCS-reconfig.** To study the impact of changing the network topology within training iterations, we simulate a reconfigurable TopoOpt interconnect. We rely on commercially available Optical Circuit Switches (OCSs) for this design and assume the reconfiguration latency is 10 ms. Given that FlexFlow's parallelization strategy search is not aware of dynamically reconfigurable networks, following prior work [89], we measure the traffic demand every 50 ms and adjust the circuits based on a heuristic algorithm to satisfy the current traffic demand as much as possible. We also enable host-based forwarding such that the communication is not blocked even when a direct link is not available (Appendix E.4).
• **Ideal Switch.** An ideal electrical switch that scales to any number of servers, where each server is connected to the switch via a link with $d \times B$ bandwidth. For any pair of d and B, no network can communicate faster than this ideal case. In practice, the Ideal Switch can be approximated with a full-bisection bandwidth Fat-tree where the bandwidth of each link is $d \times B$.
• **Fat-tree.** To compare the performance of TopoOpt to that of a similar-cost Fat-tree architecture, we simulate a full bisection bandwidth Fat-tree where each server has one NIC and the bandwidth of each link is $d \times B'$, where B' is lower than B and is selected such that Fat-tree's cost is similar to TopoOpt (§5.2).
• **Oversub. Fat-tree.** This is a 2:1 oversubscribed Fat-tree interconnect, where the bandwidth of each link is $d \times B$ but half of the links in the ToR uplink layer are omitted.
• **SiP-ML [89].** SiP-ML is a futuristic DNN training cluster with Tbps of bandwidth per GPU. While having a Tbps network is beneficial, our goal is to compare the algorithmic contributions of TopoOpt and SiP-ML. Hence, to make a fair comparison, we allocate d wavelengths, each with bandwidth B, to each SiP-ML GPU and follow its SiP-Ring algorithm to find a topology with a reconfiguration latency of 25 μs. Appendix F elaborates on our modifications to SiP-ML.
• **Expander [127, 135].** Finally, we simulate a fabric where each server has d NICs with bandwidth B interconnected via an Expander topology.

DNN Workloads. We simulate six real-world DNN models: DLRM [20], CANDLE [4], BERT [62], NCF [75], ResNet50 [74] , and VGG [126]. List 1 (Appendix D) provides details about model configurations and batch sizes used in this paper.

Parallelization strategy. We use *FlexNet*'s topology-aware parallelization strategy search for Ideal Switch, Fat-tree, Over-

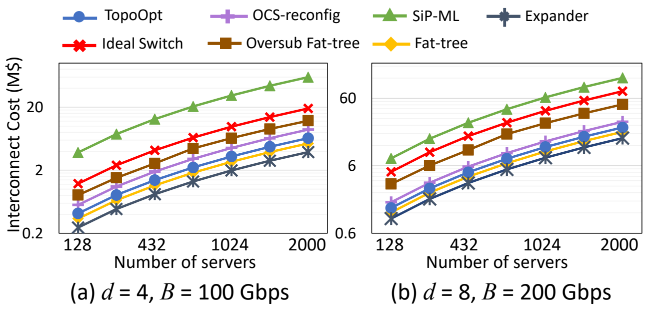


Figure 10: Interconnect cost comparison.

sub. Fat-tree, SiP-ML, and Expander networks. For TopoOpt, we use *FlexNet*'s alternating optimization framework to find the best parallelization strategy jointly with topology, where the final parallelization strategy is either hybrid or pure data-parallel. We use ring-AllReduce and distributed parameter server [93] as default AllReduce communication collectives between servers and within servers, respectively. Each data point averages 5–10 simulation runs.

5.2 Cost Analysis

We begin our evaluations by comparing the cost of various network architectures. Details about the cost of each component used in each architecture are given in Appendix G.

Figure 10 compares the interconnect cost across various network architectures as the number of servers is increased. We estimate the cost of Ideal Switch with a full-bisection Fat-tree of the same bandwidth. We make the following observations. First, using OCSs for TopoOpt is more expensive (1.33×, on average) than patch panels. Note that OCSs can be used in both TopoOpt and OCS-reconfig interconnects. Second, the cost of TopoOpt overlaps with that of the Fat-tree. This is intentional, because having a cost-equivalent architecture enables us to compare the performance of TopoOpt to a cluster at the same price point. Third, the ratio of Ideal Switch's cost to TopoOpt's cost is 3.2× on average. Finally, the most and least expensive fabrics are SiP-ML and Expander, respectively, and as this section shows, they both perform worse than TopoOpt for certain workloads.

We acknowledge that estimating the cost of networking hardware is challenging because prices are subject to significant discounts with bulk orders. Assuming all components in this analysis are subject to similar bulk order discounts, the relative comparison across architectures remains valid. As a point of comparison, we compute the cost of a cluster with 4,394 servers ($k = 26$ Fat-tree) by following the discounted cost trends in Sirius [53] and with 50% discounts for patch panels. For a cluster at this scale, the cost of full-bisection bandwidth Fat-tree (which approximates our Ideal Switch baseline) relative to the cost of TopoOpt changes from 3.0× to 3.6×, indicating our estimates are reasonable. Moreover, a TopoOpt cluster incurs lower energy cost than Fat-trees, as optical switches are passive.

5.3 Performance Comparison on Dedicated Clusters

This section compares the training iteration time of TopoOpt with that of other network architectures when the cluster is dedicated to serving one DNN training job.

Figure 11a compares the training iteration times of various architectures for CANDLE distributed on a dedicated cluster of 128 servers with a server degree of four ($d = 4$). We vary the link bandwidth (B) on the x-axis. The figure shows that Ideal Switch, TopoOpt, and SiP-ML architectures achieve similar performance because the best parallelization strategy for CANDLE at this scale is mostly data parallel, with few MP transfers. The OCS-reconfig architecture performs poorly because it uses the instantaneous demand as the baseline to estimate the future traffic to schedule circuits. This estimation becomes inaccurate during training, in particular when the current AllReduce traffic is about to finish but the next round of AllReduce has not started. The Expander architecture has the worst performance, as its topology is not optimized for DNN workloads. Averaging across all link bandwidths, compared to Fat-tree interconnect, TopoOpt improves the training iteration time of CANDLE by 2.8×; i.e., the ratio of CANDLE's iteration time on Fat-tree to TopoOpt is 2.8. TopoOpt's servers have more raw bandwidth, resulting in faster completion time.[1]

Figures 11b and 11c show the training iteration times for VGG and BERT. The trends are similar to CANDLE, as these models have similar degree requirements. Compared to Fat-tree, on average, TopoOpt improves the iteration time of VGG and BERT by 2.8× and 3×, respectively.

The cases of DLRM and NCF are more interesting, as they have more MP transfers than the other DNNs. As shown in Figures 11d and 11e, TopoOpt's performance starts to deviate from Ideal Switch, especially for NCF, because it uses host-based forwarding for the many-to-many MP transfers (§5.4 and §5.5). For DLRM (and NCF), TopoOpt is 2.8× (and 2.1×) faster than Fat-tree, while Ideal Switch further improves the training iteration time by 1.3× (and 1.7×) compared to TopoOpt. SiP-ML performs poorly, and even when we increase the link bandwidth, its training iteration time stays flat. This happens because MP transfers in DLRM and NCF require several circuit reconfigurations to meet the traffic demand.

Finally, Figure 11f shows most architectures achieve similar training iteration times for ResNet50 since it is not a communication-heavy model. The Expander architecture performs poorly when the link bandwidth is lower than 100 Gbps, as the topology does not match the AllReduce traffic pattern.

[1]It is possible to improve the performance of the Expander fabric by augmenting Blink's approach [136] to a cluster-level solution.

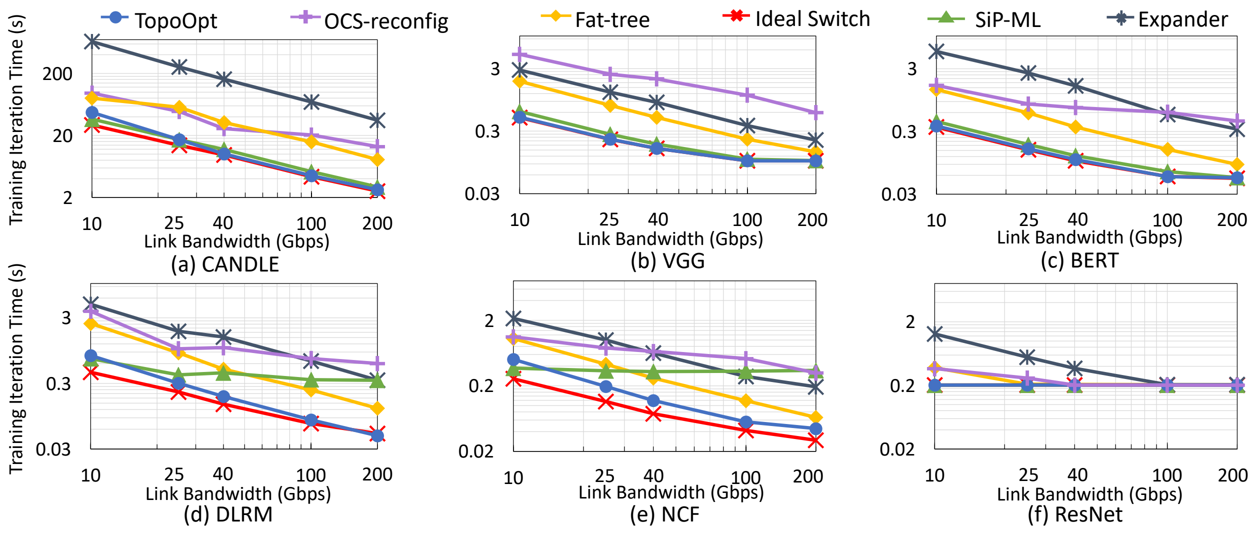


Figure 11: Dedicated cluster of 128 servers ($d = 4$).

We repeat this simulation with $d = 8$ and observe a similar performance trend (Appendix H).

5.4 Impact of All-to-all Traffic

This section evaluates the impact of all-to-all traffic patterns on TOPOOPT's performance. In particular, TOPOOPT's host-based forwarding approach incurs bandwidth tax [99] exacerbated by *all-to-all* and *many-to-many* communication patterns. This tax is defined as the ratio of the traffic volume in the network (including forwarded traffic) to the volume of logical communication demand. Hence, the bandwidth tax for a full bisection bandwidth Fat-tree topology is always one, because hosts do not act as relays for each other.

Consider a DNN model with R bytes of AllReduce traffic and A bytes of all-to-all traffic, distributed on a full bisection bandwidth topology with total network bandwidth $N \cdot B_F$ (i.e., number of servers multiplied by the bisection bandwidth). The training iteration time of this DNN is: $T_F = \frac{R}{N \cdot B_F} + \frac{A}{N \cdot B_F} + C_{bs}$, where C_{bs} is the computation time of the model with batch size bs.[2]

Now suppose the same DNN is distributed on a TOPOOPT topology with total network bandwidth NB_T. In this case, assuming the entire AllReduce traffic is carried on TotientPerms with direct links, the training iteration time becomes $T_T = \frac{R}{N \cdot B_T} + \frac{\alpha \cdot A}{N \cdot B_T} + C_{bs}$ (Eq. 1), where α represents the slowdown factor that all-to-all transfers create in the network, due to host-based forwarding. The value of α depends on the amount of bandwidth tax and routing strategy (§5.5).

Increasing the amount of all-to-all traffic (A) increases the iteration time for both T_F and T_T. But when $N \cdot B_F$ and $N \cdot B_T$ are equal, TOPOOPT's performance degrades faster because of the α factor in the numerator. To quantify this behavior concretely, we distribute a DLRM training task with 128 embedding tables on a cluster with 128 servers. We choose large embedding tables and distribute each table on each server, creating worst-case all-to-all traffic.

Figure 12 compares the training iteration times of TOPOOPT, Ideal Switch, and Fat-tree as the batch size is increased. The top x-axis lists the ratio of all-to-all to AllReduce traffic for each batch size value given on the bottom x-axis. As shown in Figure 12a, when the batch size is 128 and $d = 4$, TOPOOPT's performance matches that of Ideal Switch, while Fat-tree is a factor of 2.7 slower. This result agrees with the performance gains in Figure 11d, as the batch sizes are the same.

Increasing the batch size increases A, and this, in turn, increases the training iteration times in all three architectures. As predicted by Eq. (1), TOPOOPT's iteration time increases faster. Specifically, when the batch size is 2048 and all-to-all traffic is 80% of AllReduce traffic, TOPOOPT performs poorly, and the iteration time is a factor of 1.1 higher than that of the Fat-tree architecture. Increasing the server degree d mitigates the problem, as shown in Figure 12b. Note that increasing the batch size does not always result in faster training time [89, 109, 124]. Moreover, publicly available data suggest 2048 is the largest batch size for training DLRM [102]. The number of columns in the embedding tables and the number of servers are smaller in their workload: (92, 16) vs. (128, 128), respectively. Hence, the DLRM workload we evaluate contains more all-to-all traffic than the state-of-the-art model used in industry.

[2]For clarify of presentation, this formulation assumes no overlap between communication and computation stages and no competing traffic.

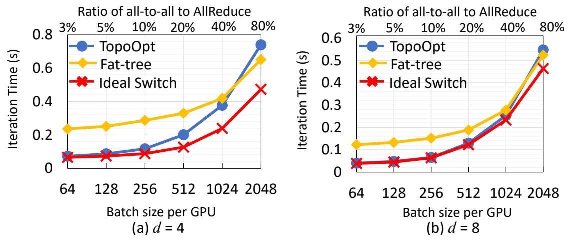


Figure 12: Impact of all-to-all traffic on a dedicated cluster of 128 servers ($B = 100$ Gbps).

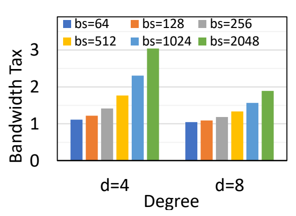


Figure 13: Bandwidth tax.

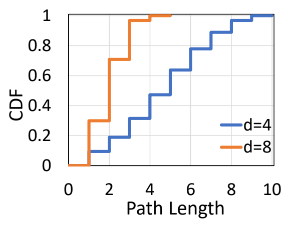


Figure 14: Path length CDF.

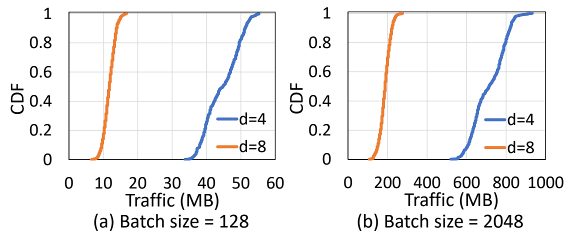


Figure 15: Traffic distribution.

5.5 Impact of Host-based Forwarding

Two factors impact the performance of host-based forwarding in TopoOpt: bandwidth tax and routing strategy.

Bandwidth tax. Figure 13 shows the amount of bandwidth tax experienced by the DLRM job in the previous section. Each bar represents a different batch size. At batch size 64 with $d = 4$, TopoOpt experiences a bandwidth tax of 1.11, indicating that host-based forwarding creates 11% extra traffic in the network. Increasing the degree to $d = 8$ further improves this number to 1.05. In the worst-case scenario with batch size 2048, TopoOpt pays a bandwidth tax of 3.03 when $d = 4$, causing it to perform worse than Fat-tree, as shown in Figure 12a. Determining the value of tolerable bandwidth tax is challenging for a TopoOpt cluster, as it depends on the compute time and the amount of compute-communication overlap, and this varies for different DNN models.

Impact of path length. Intuitively, the amount of bandwidth tax grows with the path length [99]. Figure 14 shows the CDF of path length across all server pairs. When $d = 4$, the average path length is 5.7, resulting in at least 5.7× overhead of host-based forwarding relative to Ideal Switch for all-to-all traffic. Based on Eq. (1), and since the total network bandwidth in TopoOpt is higher than Fat-tree ($NB_T > NB_F$), the overhead of host-based forwarding becomes at least 1.4× for the Fat-tree architecture. Increasing the server degree to 8 reduces the average path length to 3, thereby reducing the overhead bound. Appendix H evaluates the impact of increasing node degree on performance for other models.

Routing strategy. Building a topology with a small path length is necessary but not sufficient to reduce the impact of host-based forwarding. To handle forwarded traffic with minimum performance impact, the routing strategy also needs to be efficient. The best routing strategy *minimizes the maximum link utilization* for a given network topology, similar to WAN traffic engineering solutions [91]. However, finding the optimal routing strategy requires solving a set of linear equations with a centralized controller [76, 81]. To quantify the load imbalance in TopoOpt, Figure 15 illustrates the CDF of the amount of traffic carried by each physical link for an all-to-all traffic matrix. When the batch size is 128 (Figure 15a), the link with the least traffic carries 39% and 59% less traffic than the link with the most traffic, for $d = 4$ and $d = 8$, respectively. This imbalance in load suggests further opportunities to improve the performance of TopoOpt. Achieving optimal routing makes α (Eq. (1)) equal to the average path length. Without a centralized controller, however, the link utilization becomes non-uniform, and the average path length only serves as a lower bound. We leave optimizing the routing strategy in TopoOpt to future work.

5.6 Performance on Shared Clusters

We now compare the performance of different network architectures when the cluster is shared across multiple DNN jobs. Following prior work [98, 115], we run a series of simulations where 40% of the jobs are DLRM, 30% are BERT, 20% are CANDLE, and 10% are VGG16. We change the number of active jobs to represent the load on the cluster. Assuming each job requests 16 servers (64 GPUs), we execute 5, 10, 15, 20, and 27 jobs on the cluster to represent 20%, 40%, 60%, 80% and 100% load, respectively.

Figure 16 compares the average and 99%-tile iteration time at different loads for a cluster with 432 servers where $d = 8$ and $B = 100$ Gbps. SiP-ML does not support multiple jobs; hence, we omit it in this experiment. We omit OCS-reconfig and Expander networks, as they both show poor performance in this setting. Instead, we add the Oversub. Fat-tree interconnect to demonstrate the impact of congestion on Fat-tree topologies. Figure 16a shows that TopoOpt improves the average iteration time by 1.7× and 1.15×, compared to the Fat-tree and Oversub. Fat-tree architectures, respectively. We observe a similar trend for the tail iteration completion times, depicted in Figure 16b. At the extreme case when all servers are loaded, TopoOpt's tail training iteration time is 3.4×

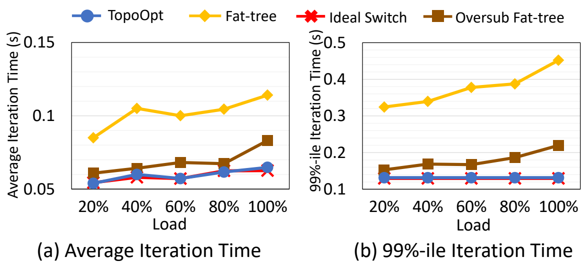


Figure 16: Shared cluster of 432 servers ($d = 8, B = 100$ Gbps).

faster compared to Fat-tree architecture. Averaging across all load values on the x-axis, TopoOpt improves the tail training iteration time by 3× and 1.4× compared to Fat-tree and Oversub. Fat-tree architectures.

5.7 Impact of Reconfiguration Latency

Figure 17 shows the training iteration time of DLRM and BERT in the same setting as Figure 11, while sweeping the reconfiguration latency of OCSs in OCS-reconfig from 1 μs to 10 ms. The horizontal blue line corresponds to TopoOpt's iteration time; it remains constant as it does not reconfigure the network topology. We find host-based forwarding is challenging when the network is reconfigurable, as the circuit schedules need to account for forwarding the traffic while the topology reconfigures. Therefore, we evaluate the performance of OCS-reconfig with and without host-based forwarding. The purple line corresponds to OCS-reconfig with host-based forwarding (same as OCS-reconfig evaluated in Figure 11), denoted by OCS-reconfig-FW. For the orange line, we disable host-based forwarding (similar to SiP-ML) and call it OCS-reconfig-noFW.

We find enabling host-based forwarding when the topologies reconfigures within a training iteration is not always beneficial. For DLRM (Figure 17a), OCS-reconfig-FW achieves better performance than OCS-reconfig-noFW, as DLRM has all-to-all MP transfers which benefit from host-based forwarding. However, for BERT (Figure 17b), enabling forwarding increases the chance of inaccurate demand estimation and imposes extra bandwidth tax, therefore increasing the iteration time of OCS-reconfig-FW by a factor of 1.4 compared to OCS-reconfig-noFW.

Reducing the reconfiguration latency all the way to 1 μs enables OCS-reconfig-noFW to match the performance of TopoOpt. However, OCS-reconfig-FW still suffers from inaccurate demand estimations. Although fast reconfigurable switches are not yet commercially available, they are going to be essential in elastic scenarios where the cluster is shared across multiple jobs and servers join and leave different jobs unexpectedly, or when large, high-degree communication dominates the workload. We believe futuristic fast reconfigurable switches, such as Sirius [53], are well-suited for this

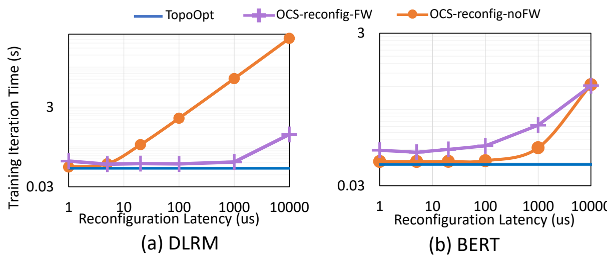


Figure 17: Impact of reconfiguration latency (d=8, B=100 Gbps).

setting. Finding a parallelization algorithm that is aware of reconfigurability within training iterations is a challenging and exciting future research problem.

6 Prototype

Testbed setup. We build a prototype to demonstrate the feasibility of TopoOpt. Our prototype includes 12 ASUS ESC4000A-E10 servers and a G4 NMT patch panel [43]. Each server is equipped with one A100 Nvidia GPU [37] (40 GB of HBM2 memory), one 100 Gbps HP NIC [29], and one 100 Gbps Mellanox ConnectX5 NIC. Our HP NICs are capable of supporting 4×25 Gbps interfaces using a PSM4 transceiver with four breakout fibers [8], enabling us to build a TopoOpt system with degree $d = 4$ and $B = 25$ Gbps. We use RoCEv2 for communication, and enable DCB [19] and PFC on these interfaces to support a lossless fabric for RDMA. We build a completely functional TopoOpt prototype with our patch panel (Figure 18). We compare TopoOpt's performance with two baselines: (i) Switch 100Gbps, where the servers are connected via 100 Gbps links to a switch, and (ii) Switch 25Gbps, where the servers are connected via 25 Gbps links to a switch. The Switch 100Gbps baseline corresponds to the Ideal Switch case in our simulations.

Distributed training framework. We use FlexFlow's training engine [26], based on Legion's parallel programming system [30], to train four DNN models: ResNet50 [74], BERT [62], VGG16 [126], and CANDLE [4]. For DLRM, we use Facebook's implementation from [20]. Since our prototype is an order of magnitude smaller in scale than our simulation setup, we use smaller model and batch sizes.

Modifications to NCCL. By default, the NCCL communication library [36] assumes all network interfaces are routable from other interfaces. This assumption is not ideal for TopoOpt because we have a specific routing strategy to optimize training time. We modify NCCL to understand TopoOpt's topology and respect its routing preferences. Moreover, we integrate our TotientPerms AllReduce permutations into NCCL and enable it to load-balance parameter synchronization across multiple ring-AllReduce permutations.

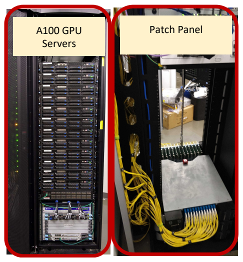


Figure 18: Testbed photo.

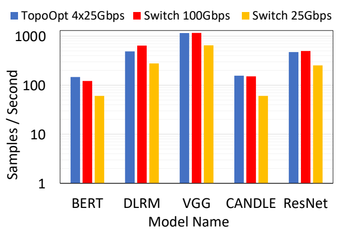


Figure 19: Training throughput (samples/second).

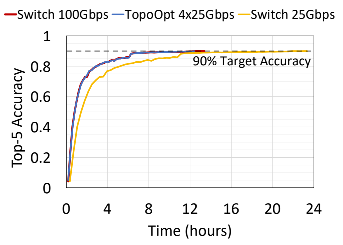


Figure 20: Time-to-accuracy of VGG19 with ImageNet.

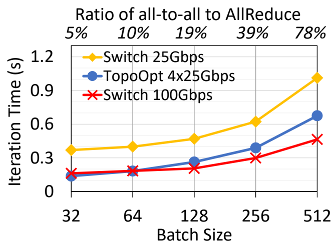


Figure 21: Impact of all-to-all traffic in our testbed.

RDMA forwarding. Implementing TopoOpt with today's RDMA NICs requires solving an engineering challenge, because the RDMA protocol assumes a switch-based network. Packet processing and memory access in RDMA protocol are offloaded to the NIC, and a RoCEv2 packet whose destination IP address is different from that of the host is assumed to be corrupted. Therefore, the NIC silently drops forwarded packets. To address this issue, we collaborated with engineers at Marvell who developed the firmware and driver of our HP NICs. Our solution uses a feature called network partitioning (NPAR) which enables the NIC to separate host-based forwarding traffic from direct traffic, and uses the Linux kernel to route them (details in Appendix I). Our conversations with Marvell indicate that updating the firmware and the driver enables the NIC to route forwarded RoCEv2 packets, thereby bypassing the kernel entirely.

Training performance. Figure 19 demonstrates that TopoOpt's training throughput (samples/second) is similar to our Switch 100 Gbps baseline for all models. The performance of Switch 25Gbps baseline is lower because its available bandwidth is lower than TopoOpt. Figure 20 shows the time-to-accuracy plot of training VGG19 on the ImageNet [61] dataset. As the figure indicates, TopoOpt reaches the target accuracy of 90% 2.0× faster than the Switch 25Gbps baseline. TopoOpt achieves similar performance to the Switch 100Gbps baseline, as the blue and red lines overlap in Figure 20.

Impact of all-to-all traffic. Similar to Section 5.4, we evaluate the impact of all-to-all MP traffic on our RDMA-forwarding enabled testbed by measuring the average iteration time across 320 iterations of a DLRM job distributed in our testbed. We vary the amount of all-to-all traffic by changing the batch size. To create worst-case traffic, we increase the embedding dimensions by 128× relative to the state-of-the-art [20] (model details are in List 1, Appendix D). Figure 21 shows the training iteration time for various batch sizes. The results are consistent with Figure 12, but since the bandwidth tax in our 12-server testbed is much smaller than a 128-server cluster in simulations, TopoOpt performs better relative to the switch-based architectures for a given all-to-all to AllReduce traffic ratio. For instance, for batch size 512, the ratio of all-to-all traffic to AllReduce is 78%, and the training iteration time with TopoOpt is 1.6× better than the Switch 25Gbps baseline.

7 Discussion

Target workload. The most suitable workload for a TopoOpt cluster is a set of large DNN training jobs with hybrid data and model parallelism (or simply data parallelism). We assume the set of servers assigned to each job remains the same throughout the lifetime of the job, and the GPUs are not shared across multiple jobs.

Storage and control plane traffic. Meta's training clusters consist of custom-designed servers, each with eight GPUs, eight dedicated NICs for training traffic (GPU NICs), and four additional NICs for storage and other traffic (CPU NICs) [102]. Other companies, such as NVIDIA, have similar architectures [10]. TopoOpt only considers GPU NICs as server degree and partitions the network dedicated for training traffic. The CPU NICs are connected through a separate fabric to carry storage and other control plane traffic.

Supporting dynamic scheduling and elasticity. Others have demonstrated the benefits of dynamically choosing the training servers for elastic training jobs [98, 115]. Our target use case in Meta is to leverage TopoOpt for the vast number of long-lasting training jobs that do not change dynamically. In cases where elasticity is required, instead of using patch panels, we use OCSs (or other fast reconfigurable optical switches) to change the servers participating in a job quickly. Note that dynamically changing the set of servers participating in a job while keeping both the topology and the parallelization strategy optimal requires augmenting the optimization space with an additional dimension, making the problem even more challenging. We leave this to future work.

Handling failures. Unlike SiP-ML's single ring topology [89], a single link failure does not disconnect the graph in TopoOpt. When a fiber fails, TopoOpt can temporarily use a link dedicated to MP traffic to recover an AllReduce ring. In case of permanent failures, TopoOpt reconfigures to swap ports and recover the failed connection.

Supporting multi-tenancy. To support multi-tenancy [142, 143], TopoOpt can leverage NVIDIA's MIG [39] to treat one physical server as multiple logical servers in its topology.

TotientPerms in Fat-trees. Although our TotientPerms technique is well-suited for reconfigurable optical interconnects, it may be of independent interest for Fat-tree interconnects as well since load-balancing the AllReduce traffic across multiple permutations can help with network congestion.

TopoOpt's limitations. TopoOpt's approach assumes the traffic pattern does not change between iterations. However, this assumption may not hold for Graphic Neural Network (GNN) models [121] or Mixture-of-Expert (MoE) models [80]. In addition, we plan to extend TopoOpt by bringing its demand-awareness design within training iterations. This is an open research question, and as shown in Section 5.7, we will need fast-reconfigurable optical switches, as well as a more sophisticated scheduling algorithm. Another limitation of TopoOpt is that a single link failure within a AllReduce ring causes the full ring to become inefficient for AllReduce traffic. A fast optical switch addresses this problem by quickly reconfiguring the topology.

8 Related Work

Optimizing DNN training. To address the increasing computation and network bandwidth requirements of large training jobs, a plethora of frameworks have been proposed [5, 46, 58, 69, 77, 79, 85, 86, 105, 108, 111, 117, 118, 123, 129, 136, 146]. These frameworks distribute the dataset and/or DNN model across accelerators while considering the available network bandwidth, but unlike TopoOpt, they do not consider optimizing the *physical layer topology*. Specifically, Blink [136] builds collectives for distributed ML, but it needs a physical topology to generate spanning trees. Zhao et al. [147] study the optimal topology for collective communication operations, but this does not apply for general MP traffic. In addition, several methods have been proposed to quantize and compress the gradients to reduce the amount of communication data across servers [48, 56, 144]. While these approaches are effective, they are designed for data parallel strategies and do not consider the large amount of data transfers caused by model parallel training. Wang et al. [138] compare the performance of Fat-trees and BCube topologies for distributed training workloads and highlight several inefficiencies in Fat-trees. SiP-ML [89] demonstrates the benefits of 8 Tbps silicon photonics-based networks for distributed training. However, unlike TopoOpt, these proposed approaches do not *co-optimize* topology and parallelization strategy.

DNN parallelization strategies. Data and model parallelism are widely used by today's DNN frameworks (e.g., TensorFlow [44], PyTorch [42], MXNet [17]) to parallelize training across multiple devices. Recent work has also proposed *automated frameworks* (e.g., FlexFlow [85], ColocRL [101], MERLIN [38]) that find efficient parallelization strategies by searching over a comprehensive space of potential strategies. These frameworks rely on and are optimized for the conventional Fat-tree interconnects. TopoOpt proposes a new approach to building DNN training systems by jointly optimizing network topology and parallelization strategy.

DNN training infrastructures and schedulers. Several training infrastructures have been proposed recently, including NVIDIA DGX SuperPOD [10], TPU cluster [9], and supercomputers [1]. All these systems assume non-reconfigurable network topologies, such as Fat-tree, Torus, and other traffic-oblivious interconnects. TopoOpt is the first DNN system to use commodity reconfigurable interconnects to accelerate DNN jobs.Gandiva [140], Themis [98], Tiresias [70], BytePS [86, 111], and Pollux [115] seek to improve the utilization of GPU clusters through scheduling algorithms. These approaches are complementary to ours, and many of their techniques can be applied to a TopoOpt cluster.

Optical Interconnects. Several papers have demonstrated the benefits of optically reconfigurable interconnects for datacenters [51, 53, 57, 60, 64, 68, 95–97, 99, 100, 113]. These designs lead to sub-optimal topologies for distributed DNN traffic. Similarly, *traffic oblivious* interconnects, such as RotorNet [99, 100], are a great fit for datacenter workloads, but they are not suitable for DNN training jobs characterized by repetitive traffic demands. Hybrid electrical/optical datacenter proposals [64, 137] can be used to route AllReduce traffic through the optical fabric and MP flows through a standard electrical Fat-tree network. But hybrid clusters are not cost effective and suffer from many problems, including TCP ramp-up inefficiencies [103], segregated routing issues [65], and uncertainty in terms of how to divide the cluster between electrical and optical fabrics [68, 72].

9 Conclusion

We present TopoOpt, a novel system based on optical devices that jointly optimizes DNN parallelization strategy and topology to accelerate training jobs. We design an alternating optimization algorithm to explore the large space of *Computation* $\times$ *Communication* $\times$ *Topology* strategies for a DNN workload, and demonstrate TopoOpt obtains up to 3.4$\times$ faster training iteration time than Fat-tree.

10 Acknowledgments

We thank our shepherd Sangeetha Abdu Jyothi and anonymous reviewers for their valuable feedback. We also acknowledge Meta for supporting this research. In particular, we thank Omar Baldonado, Gaya Pradeep Sindhu, and Jahangir Hasan. In addition, we thank Alan Gibbemeyer, Bob Shine, Karl Kuhn and Ramiro Voicu from Telescent for their support on the Telescent NTM-G4. We also thank Arial Elior, Karl Erickson, and Nishant Lodha from Marvell for their help on RDMA forwarding. The MIT-affiliated authors are supported by ARPA-E ENLITENED PINE DE-AR0000843, DARPA FastNICs 4202290027, NSF grants CNS-2008624, SHF-2107244, ASCENT-2023468, CAREER-2144766, PPoSS-2217099, CNS-2211382, Meta faculty award, Google faculty award, and Sloan fellowship FG-2022-18504.

References

[1] Summit Supercomputer, 2014. https://www.olcf.ornl.gov/summit/.

[2] Datasheet for Single Mode Network Optical Switch up to 384x384 ports, 2016. https://www.hubersuhner.com/en/documents-repository/technologies/pdf/data-sheets-optical-switches/polatis-series-7000n.

[3] Baidu, 2017. https://github.com/baidu-research/baidu-allreduce.

[4] CANDLE Uno: Predicting Tumor Dose Response across Multiple Data Sources, 2017. https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/Uno.

[5] Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow, 2017. https://eng.uber.com/horovod.

[6] CALIENT Edge 640™ Optical Circuit Switch, 2018. https://www.calient.net/2018/03/calient-edge640-optical-circuit-switch-offers-industrys-highest-density-fiber-optic-cross-connect/.

[7] htsim packet simulator, 2018. https://github.com/nets-cs-pub-ro/NDP/wiki/NDP-Simulator.

[8] AOI 100G PSM4 Transceiver, 2020. https://www.ebay.com/itm/234092018446?hash=item3680f8bb0e:g:WoMAAOSwLFJg8dKF.

[9] Google TPU, 2020. https://cloud.google.com/tpu.

[10] Nvidia DGX SuperPOD, 2020. https://www.nvidia.com/en-us/data-center/dgx-superpod/.

[11] NVIDIA is Preparing Co-Packaged Photonics for NVLink, Dec. 2020. https://www.techpowerup.com/276139/nvidia-is-preparing-co-packaged-photonics-for-nvlink.

[12] 100GBASE-SR4 QSFP28 850nm 100m DOM MTP/MPO MMF Optical Transceiver Module, 2022. https://www.fs.com/products/48354.html.

[13] 10GBASE-SR SFP+ 850nm 300m DOM LC MMF Transceiver Module, 2022. https://www.fs.com/products/11552.html.

[14] 1x2 PLC Fiber Splitter, Splice/Pigtailed ABS Module, 2.0mm, SC/APC, Singlemode, 2022. https://www.fs.com/products/11615.html.

[15] 25GBASE-SR SFP28 850nm 100m DOM LC MMF Optical Transceiver Module, 2022. https://www.fs.com/products/67991.html.

[16] 40GBASE-SR4 QSFP+ 850nm 150m DOM MTP/MPO MMF Optical Transceiver Module, 2022. https://www.fs.com/products/36143.html.

[17] Apache MXNet, 2022. https://mxnet.apache.org/.

[18] Colfax Direct, HPC and Date Center Gear, 2022. https://www.colfaxdirect.com/.

[19] Data Center Bridging eXchange (DCBX), 2022. https://man7.org/linux/man-pages/man8/dcb-dcbx.8.html.

[20] Deep Learning Recommendation Model for Personalization and Recommendation Systems, 2022. https://github.com/facebookresearch/dlrm.

[21] Edgecore AS5812-54X 48-Port 10GbE Bare Metal Switch with ONIE - Part ID: 5812-54X-O-12V-F, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3614.

[22] Edgecore AS6812-32X 32-Port 40GbE Bare Metal Switch with ONIE - Part ID: 6812-32X-O-AC-F-US, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3078.

[23] Edgecore AS7312-54XS 48-Port 25GbE + 6-Port 100GbE Bare Metal Switch with ONIE - Part ID: 7312-54XS-O-AC-F-US, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3598.

[24] Edgecore AS7816-64X 64-Port 100GbE Bare Metal Switch with ONIE - Part ID: 7816-64X-O-AC-B-US, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3483.

[25] Euler's totient function, 2022. https://en.wikipedia.org/wiki/Euler%27s_totient_function.

[26] Flex Flow's Training Engine, 2022. https://flexflow.ai/.

[27] FlexFlow source code, 2022. https://github.com/flexflow/FlexFlow.

[28] FS.COM, 2022. https://www.fs.com/.

[29] HPE Ethernet 4x25Gb 1-port 620QSFP28 Adapter, 2022. https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c05220334.

[30] Legion Programming System, 2022. https://legion.stanford.edu/overview/.

[31] Managing edge data centers through automation and remote diagnostics, 2022. https://www.telescent.com/blog/2021/11/11/managing-edge-data-centers-through-automation-and-remote-diagnostics.

[32] Mellanox ConnectX-4 Single Port 25 Gigabit Ethernet Adapter Card, PCIe 3.0 x8 - Part ID: MCX4111A-ACAT, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=2814.

[33] Mellanox ConnectX-4 Single Port 40 Gigabit Ethernet Adapter Card, PCIe 3.0 x8 - Part ID: MCX4131A-BCAT, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=2817.

[34] Mellanox ConnectX-5 EN Single Port 100 Gigabit Ethernet Adapter Card, PCIe 3.0 x16 - Part ID: MCX515A-CCAT, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3150.

[35] Mellanox ConnectX-5 VPI Adapter Card with Multi-Host Socket Direct, Dual PCIe 3.0 x8 - Part ID: MCX556M-ECAT-S25, 2022. https://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3209.

[36] NCCL, 2022. https://github.com/NVIDIA/nccl-tests.

[37] NVIDIA A100 Tensor Core GPU, 2022. https://www.nvidia.com/en-us/data-center/a100/.

[38] NVIDIA MERLIN, 2022. https://developer.nvidia.com/nvidia-merlin.

[39] NVIDIA MULTI-INSTANCE GPU, 2022. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[40] Patch Panel Wiki, 2022. https://en.wikipedia.org/wiki/Patch_panel.

[41] Polatis Optical Circuit Switch, 2022. https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabled.asp.

[42] PyTorch, 2022. https://pytorch.org.

[43] Telescent G4 Network Topology Manager, 2022. https://www.telescent.com/products.

[44] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[45] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale, 2020.

[46] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Learning generalizable device placement algorithms for distributed machine learning. In *Advances in Neural Information Processing Systems*, volume 32, pages 3981–3991. Curran Associates, Inc., 2019.

[47] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[48] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 1709–1720. Curran Associates, Inc., 2017.

[49] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[50] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM.

[51] Daniel Amir, Tegan Wilson, Vishal Shrivastav, Hakim Weatherspoon, Robert Kleinberg, and Rachit Agarwal. Optimal oblivious reconfigurable networks, 2021.

[52] Javed A. Aslam. Dynamic Programming Solution to the Coin Changing Problem, 2004. https://www.ccs.neu.edu/home/jaa/CSG713.04F/Information/Handouts/dyn_prog.pdf.

[53] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[54] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[55] J. Brownlee. *Better Deep Learning: Train Faster, Reduce Overfitting, and Make Better Predictions*. Machine Learning Mastery, 2018.

[56] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp : Adaptive residual gradient compression for data-parallel distributed training. *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[57] Li Chen, Kai Chen, Zhonghua Zhu, Minlan Yu, George Porter, Chunming Qiao, and Shan Zhong. Enabling wide-spread communications on optical fabric with megaswitch. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 577–593, Boston, MA, 2017. USENIX Association.

[58] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. SysML Conference, 2019.

[59] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, and Michael Haselman. Accelerating persistent neural networks at datacenter scale. In *Hot Chips*, volume 29, 2017.

[60] K. Clark, H. Ballani, P. Bayvel, D. Cletheroe, T. Gerard, I. Haller, K. Jozwik, K. Shi, B. Thomsen, P. Watts, H. Williams, G. Zervas, P. Costa, and Z. Liu. Sub-nanosecond clock and data recovery in an optically-switched data centre network. In *2018 European Conference on Optical Communication (ECOC)*, pages 1–3, 2018.

[61] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[62] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[63] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[64] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. *SIGCOMM'10*, pages 339–350.

[65] K. Foerster, M. Ghobadi, and S. Schmid. Characterizing the algorithmic complexity of reconfigurable data center architectures. In *Proc. ANCS '18*, pages 89–96, 2018.

[66] Everest G. and Ward Thomas. An introduction to number theory, 2005.

[67] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 249–264, Berkeley, CA, USA, 2016. USENIX Association.

[68] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 216–229, New York, NY, USA, 2016. Association for Computing Machinery.

[69] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[70] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[71] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 63–74, New York, NY, USA, 2009. Association for Computing Machinery.

[72] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. *SIGCOMM'14*, pages 319–330.

[73] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 418–430, 2019.

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

[75] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering, 2017.

[76] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[77] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS*, 2019.

[78] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[79] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR*, abs/1511.00175, 2015.

[80] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87, 1991.

[81] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[82] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed DNN training. *CoRR*, abs/1905.03960, 2019.

[83] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *CoRR*, abs/1807.11205, 2018.

[84] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2274–2283, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[85] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *SysML*, 2019.

[86] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[87] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. Optimizing bulk transfers with software-defined optical wan. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 87–100, New York, NY, USA, 2016. Association for Computing Machinery.

[88] A. S. Kewitsch. Large scale, all-fiber optical cross-connect switches for automated patch-panels. *Journal of Lightwave Technology*, 27(15):3107–3115, 2009.

[89] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. Sip-ml: High-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 657–675, New York, NY, USA, 2021. Association for Computing Machinery.

[90] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952.

[91] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering with smore. In *Proceedings of the Applied Networking Research Workshop*, ANRW '18, page 21, New York, NY, USA, 2018. Association for Computing Machinery.

[92] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. On model parallelization and scheduling strategies for distributed machine learning. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 2834–2842. Curran Associates, Inc., 2014.

[93] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. OSDI'14, pages 583–598. USENIX Association, 2014.

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

[95] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with REACToR. *NSDI'14*, pages 1–15.

[96] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[97] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: A new design element for low-latency dcns. *SIGCOMM'14*, pages 283–294.

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

[99] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. *NSDI'20*, 2020.

[100] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. *SIGCOMM '17*, pages 267–280, 2017.

[101] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2430–2439, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[102] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021.

[103] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. Adapting TCP for reconfigurable datacenter networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651–666, Santa Clara, CA, February 2020. USENIX Association.

[104] Shar Narasimhan. NVIDIA Clocks World's Fastest BERT Training Time and Largest Transformer Based Model, Paving Path For Advanced Conversational AI, Aug. 2019. https://devblogs.nvidia.com/training-bert-with-gpus/.

[105] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[106] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. Deep learning training in facebook data centers: Design of scale-up and scale-out systems, 2020.

[107] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019.

[108] T. T. Nguyen, M. Wahib, and R. Takano. Topology-aware sparse allreduce for large-scale deep learning. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2019.

[109] Yosuke Oyama, Naoya Maruyama, Nikoli Dryden, Erin McCarthy, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Peter Nugent, and Brian Van Essen. The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[110] Heng Pan, Zhenyu Li, JianBo Dong, Zheng Cao, Tao Lan, Di Zhang, Gareth Tyson, and Gaogang Xie. Dissecting the communication latency in distributed deep sparse learning. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 528–534, New York, NY, USA, 2020. Association for Computing Machinery.

[111] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[112] Genzhi Photonics. 1x2 Mechanical Optical Switch, 2022. https://www.gezhiphotonics.com/1x2-optical-switch.html.

[113] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. *SIGCOMM'13*, pages 447–458.

[114] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of ACM SIGCOMM 2022*, 2022.

[115] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[116] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[117] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale, 2022.

[118] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning, 2021.

[119] Leslie Reid. MOX Announces New Telescent Automation Technology on Its Latest Hillsboro to Portland Fiber Route, Sept. 2020. https://www.businesswire.com/news/home/20200915005391/en/MOX-Announces-New-Telescent-Automation-Technology-on-Its-Latest-Hillsboro-to-Portland-Fiber-Route.

[120] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[121] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[122] Tae Joon Seok, Niels Quack, Sangyoon Han, Richard S. Muller, and Ming C. Wu. Large-scale broadband digital silicon photonic switches with vertical adiabatic couplers. *Optica*, 3(1):64–70, Jan 2016.

[123] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[124] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research*, 20(112):1–49, 2019.

[125] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[126] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[127] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 17–17, Berkeley, CA, USA, 2012. USENIX Association.

[128] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[129] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of DNN graph operators. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[130] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[131] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

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

[133] Yuichiro Ueno and Rio Yokota. Exhaustive study of hierarchical allreduce patterns for large messages between gpus. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 430–439, 2019.

[134] Jakob Uszkorei. Transformer: A Novel Neural Network Architecture for Language Understanding, Aug. 2017. https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html.

[135] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 205–219, New York, NY, USA, 2016. ACM.

[136] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In *Conference on Machine Learning and Systems (MLSys 2020)*, March 2020.

[137] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-Through: Part-time optics in data centers. *SIGCOMM'10*, pages 327–338.

[138] S. Wang, D. Li, J. Geng, Y. Gu, and Y. Cheng. Impact of Network Topology on the Performance of DML: Theoretical Analysis and Practical Factors. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1729–1737, 2019.

[139] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. *SoCC '16*, 2016.

[140] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[141] Pengtao Xie, Jin Kyu Kim, Yi Zhou, Qirong Ho, Abhimanu Kumar, Yaoliang Yu, and Eric Xing. Lighter-communication distributed machine learning via sufficient factor broadcasting. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*, pages 795–804, Arlington, Virginia, USA, 2016. AUAI Press.

[142] Peifeng Yu and Mosharaf Chowdhury. Fine-grained GPU sharing primitives for deep learning applications. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[143] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. Fluid: Resource-aware hyperparameter tuning engine. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 502–516, 2021.

[144] Yue Yu, Jiaxiang Wu, and Longbo Huang. Double quantization for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, volume 32, pages 4438–4449. Curran Associates, Inc., 2019.

[145] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding lifecycle management complexity of datacenter topologies. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 235–254, Boston, MA, February 2019. USENIX Association.

[146] H. Zhao and J. Canny. Kylix: A sparse allreduce for commodity clusters. In *2014 43rd International Conference on Parallel Processing*, pages 273–282, 2014.

[147] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Optimal direct-connect topologies for collective communications, 2022.

[148] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. RAIL: A case for redundant arrays of inexpensive links in data center networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 561–576, Boston, MA, March 2017. USENIX Association.

A Tree-AllReduce and Other AllReduce Permutations

Section 2 established that we can manipulate the traffic of a ring-AllReduce collective by permuting the labeling of servers in the AllReduce group. Here, we illustrate how to use the same technique on another AllReduce algorithm, called tree-AllReduce.

In the tree-AllReduce algorithm, the servers are connected logically to form a tree topology. The AllReduce operation starts by running a reduce operation to the root node with recursive halving, followed by a broadcast to the rest of the cluster with recursive doubling [132].

A common instantiation of tree-AllReduce is the double binary tree (DBT) algorithm [120]. In this algorithm, the first step is to create a balanced binary tree for the nodes. The properties of balanced binary trees guarantee that one half of the nodes will be leaf-nodes, and the other half will be in-tree; thus, a second binary tree is constructed by flipping the labeling of the leaf and in-tree nodes. This way, each node (except the root in both trees) has the same communication requirements for the AllReduce operation, as described in the last paragraph, and bandwidth-optimally is achieved. Figure 23a shows an example where in the first binary tree, the in-tree nodes are even, and the leaf nodes are odd, while the second tree flips the labeling.

The DBT itself is essentially an example of permuting the node labeling to achieve an AllReduce operation with balanced communication load. We also note that we can permute the labeling *for the entire set of nodes* for a pair of DBTs to create a new pair of trees that can perform the AllReduce operation at the same speed. Figures 23b and 23c illustrate two other possible double binary trees, and their corresponding traffic demand matrix for the DLRM and CANDLE example shown in Figures 22 and 24 (§2). Arbitrary permutations can be used, and to limit the cases, we could simply consider the cyclic permutations in the modular space as described in TotientPerms.

In general, all AllReduce operations can be described as a directed graph $G = (V,E)$ where V is the set of nodes in the cluster, and E denotes data dependencies. The *permutable* property says every graph $G' = (V,E')$ that is isomorphic to G can perform the AllReduce operation equally well, where the homomorphism between G and G' is described by the symmetric group on V (generally denoted by $Sym(V)$ in group theory).

B Commercially Available Patch Panels and Optical Circuit Switches

Optical patch panels. A patch panel is a device to facilitate connecting different parts of a system. For instance, electrical patch panels are used in recording studios and concert sound systems to connect microphones and electronic instruments on demand [40]. Fiber optic patch panels are commonly used for cable management, and have been proposed in recent datacenter topology designs [145]. Reconfigurable optical patch panels are a new class of software-controlled patch panels and are already commercialized at scale [119]. For instance, Telescent offers 1008 duplex ports with insertion loss less than 0.5 dB and cost ≈\$100K (\$100/port) [88, 119]. Reconfiguration is performed using a robotic arm that grabs a fiber on the transmit side and connects it to a fiber on the receive side [88]. However, the reconfiguration latency of optical patch panels is several minutes [43]. Note that reliability is of utmost concern for operation in unmanned locations; for example, Telescent NTM patch panels have been certified to NEBS Level 3 and have over 1 billion port hours in operation [31].

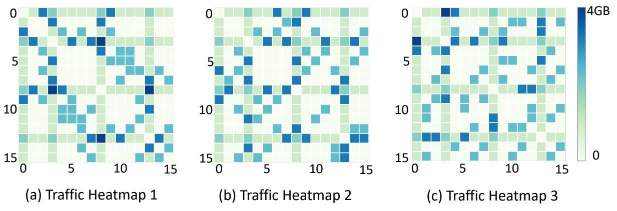
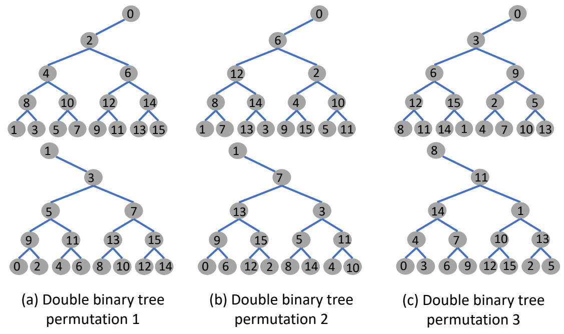


Figure 22: DLRM traffic heatmaps with double binary tree AllReduce.



(a) Double binary tree permutation 1
(b) Double binary tree permutation 2
(c) Double binary tree permutation 3

Figure 23: Double binary tree (DBT) permutations.

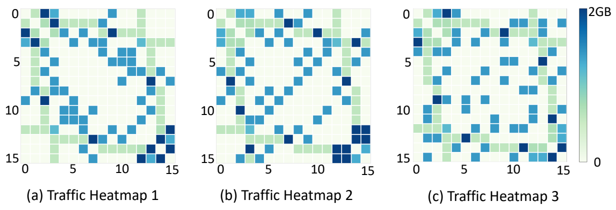


Figure 24: CANDLE traffic heatmaps with double binary tree AllReduce.

3D MEMS-based Optical Circuit Switches (OCSs). An OCS uses tiny mirrors to change the direction of light, thereby

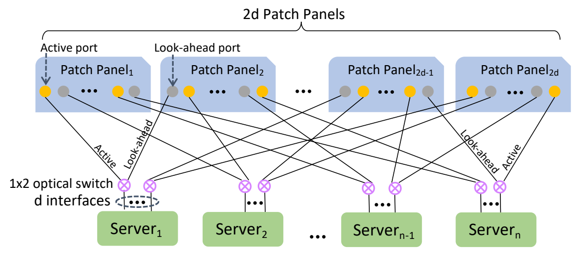


Figure 25: Active & Look-ahead ports for high reconfiguration latency.

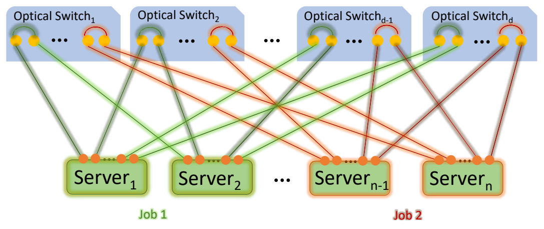


Figure 26: Sharding TopoOpt cluster for two jobs.

reconfiguring optical links. The largest optical circuit switch on the market has 384 duplex ports with ≈10 ms reconfiguration latency and is available for \$200K (\$520/port) [41]. However, the optical loss of these switches is 1.5–2.7 dB [2]. Compared to patch panels, OCSs have the following disadvantages: (*i*) each port is five times more expensive; (*ii*) their insertion loss is higher; and (*iii*) their port-count is three times lower. The main advantage of OCSs is that their reconfiguration latency is *four orders of magnitude* faster than patch panels.

C Handling Sharding and Dynamic Job Arrivals in Shared Clusters

Section 3 explained how TopoOpt can support multiple job sharing the cluster through sharding; here we provide a detailed explanation of how sharding works. Figure 26 shows how a TopoOpt cluster is sharded to train two jobs together. In this scenario, the optical switches are configured in a way such that the green part (Server 1, 2 and their corresponding links) is completely disjointed from the red part (Server $n-1$, server n). The complete isolation ensures each job gets its dedicated resources, and benefits the performance (especially the tail latency) as shown in Section 5.6.

To start a job with k servers, we need to reconfigure the interconnection between these k servers before the job starts. This can be done quickly when OCSs are used, but when patch panels are used, there could be several minutes of delay before the job can start. To address this challenge, we use a look-ahead approach to pre-provision the next topology while current jobs are running. More specifically, we use a simple 1×2 mechanical optical switch [112] at each server's interface to choose between *Active* and *Look-ahead* ports. These 1×2 switches are inexpensive (\$25) and have 0.73 dB optical loss measured in our prototype. Unlike optical splitters [14], that incur 3 dB loss, these switches choose where to send light between their two output ports. We then connect the two ends of each 1×2 switch to different patch panels, as shown in Figure 25. As a result, a TopoOpt cluster with n servers, each with d interfaces, has $2d$ patch panels where each interface is split into two parts: Active and Look-ahead. At any point in time, only one end of each 1×2 switch is participating in the active topology; the other end is pre-provisioning the topology for the next job. Since the topology and parallelization strategy are calculated off-line, we already know the sequence of job arrivals and the number of servers required by each job. This design allows each server to participate in two independent topologies. Hence, when a set of servers uses one topology for a training job, TopoOpt pre-provisions the next topology, optimized for the next task by reconfiguring Look-ahead ports. Once all the servers for the new job are ready, TopoOpt immediately *flips* to the new topology by reconfiguring the corresponding 1×2 switches.

D Model Configurations and Transfer Sizes

List 1 summarizes the parameters we used in our simulation and testbed. Model parameters and batch sizes are selected based on common values used in Meta for simulations. For the prototype, we reduce parameter values and batch sizes to fit the models in our 12-node cluster.

In most workloads observed in Meta, the size of AllReduce transfers is larger than the size of MP transfers for each iteration, because in most cases, it would not be worthwhile if MP transfers were as large as AllReduce transfers. Consider the DLRM example in Section 4.3 with 20 GB embedding tables with double-precision floating parameters. If we were to distribute this embedding table using data parallelism, each server would need to send and receive 37.5 GB of data for the AllReduce operation. On a 100 Gbps fabric, this would take 3 seconds by itself, but if we put it on one server, it would only need to transfer 32 MB/server (assume we have a per-server batch size of 8192; then, MP traffic is calculated as 16 servers $\times$ 8192 samples/server $\times$ 512 activation per sample $\times$ 8 bytes per activation / 16 servers = 32 MB). We note that adding *pipeline parallelism* can increase the amount of MP traffic as it overlaps forward and backward passes. Efficient ways to pipeline batches remains an active research area [77, 105] especially when hybrid parallelism is employed. Pure model parallelism creates another type of sparse traffic pattern where only accelerators with inter-layer dependencies need to communicate. Our TopologyFinder algorithm can support such communication patterns.

Conceptually, however, when the network bandwidth goes

VGG:
Batch/GPU: 64 (§5.3, §5.6), 32 (§6)
ResNet50:
Batch/GPU: 128(§5.3), 20 (§6)
BERT:
Batch/GPU: 16 (§5.3, §5.6), 2 (§6)
#Trans. blks: 12 (§5.3), 6 (§5.6, §6)
Hidden layer: 1024 (§5.3), 768 (§5.6), 1024(§6)
Seq. length: 64 (§5.3), 256 (§5.6), 1024(§6)
#Attn. heads: 16 (§5.3), 6 (§5.6), 16(§6)
Embed. size: 512 (§5.3, §5.6, §6)
DLRM:
Batch/GPU: 128 (§5.3),[32, $\cdots$,2048] (§5.4), 256 (§5.6), [64, $\cdots$,512] (§6)
#Dense layer: 8 (§5.3, §5.6), 4 (§6)
Dense layer size: 2048 (§5.3), 1024 (§5.6, §6)
#Dense feat. layer: 16 (§5.3, §5.6), 8 (§6)
Feat. layer size: 4096 (§5.3), 2048 (§5.6, §6)
Embed.: 128×10^7 (§5.3), 256×10^7 (§5.6), 32768×10^5 (§6)
#Embed. tables: 64 (§5.3), 16 (§5.6), 128 (§5.4) , 12 (§6)
CANDLE:
Batch/GPU: 256 (§5.3, §5.6), 10 (§6)
#Dense layer: 8 (§5.3, §5.6), 4 (§6)
Dense layer size: 16384 (§5.3), 4096 (§5.6, §6)
#Dense feat. layer: 16 (§5.3, §5.6), 8 (§6)
Feat. layer size: 16384 (§5.3), 4096 (§5.6, §6)
NCF:
Batch/GPU: 128 (§5.3)
#Dense layer: 8 (§5.3)
Dense layer size: 4096 (§5.3)
#User embedding table (MF, MLP): 32, 32 (§5.3)
#User per table: 10^6 (§5.3)
#Item embedding table (MF, MLP): 32, 32 (§5.3)
#Item per table: 10^6 (§5.3)
MF embedding dimension: 64 (§5.3)
MLP embedding dimension: 128 (§5.3)

List 1: DNN models used in our simulations and testbed.

to infinity, other overheads in the system (e.g. CUDA kernel launch) will dominate the latency. In such cases, it might be beneficial to choose model parallelism instead of data parallelism, to reduce the amount of system overheads. In particular, prior work shows 10 Tbps Silicon Photonics links enable more aggressive model parallelism where the size of MP traffic is significant [89]. TOPOOPT's approach to distribute the degree between the MP and AllReduce sub-topologies enables us to accommodate this case as well.

E Algorithm Details

E.1 TOPOLOGYFINDER

Using group theory to find AllReduce permutations. For a ring-AllReduce group with n servers labeled $S_0, \ldots, S_{n-1}$, a straightforward permutation is $(S_0 \to S_1 \to S_2 \cdots \to S_{n-1} \to S_0)$. We denote this permutation by a ring generation rule as: $S_i \to S_{(i+1)\ mod\ n}$. Since the servers form a ring, the index of the starting server does not matter. For instance, these two rings are equivalent: $(S_0 \to S_1 \to S_2 \to S_3 \to S_0)$ and $(S_1 \to S_2 \to S_3 \to S_0 \to S_1)$.[3]

We first provide the mathematical foundation of the ring permutation rule.

Theorem 2 (Ring Generation). *For a cluster of n nodes $V = \{S_0, S_1, \cdots, S_{N-1}\}$, all integer numbers $p < n$, where p is co-prime with n (i.e. $gcd(p,n) = 1$) represent a unique ring-AllReduce permutation rule.*

Proof. Consider the integer modulo n group with addition $\mathbb{Z}_n^+ = \{0, 1, \cdots, (n-1)\}$. $\mathbb{Z}_n^+$ is a cyclic group. By the fundamental theorem of cyclic groups, p is a generator of $\mathbb{Z}_n^+$ if and only if $gcd(p,n) = 1$. Hence we can cover the entire $\mathbb{Z}_n^+$ by repeatedly adding p to itself.

Now consider the graph $G_{\mathbb{Z}_n^+,p} = (V_{\mathbb{Z}_n^+}, E_p)$ where the set of vertices $V_{\mathbb{Z}_n^+} = \mathbb{Z}_n^+$ and $E_p = \{(a \times p, (a+1) \times p) \in V_{\mathbb{Z}_n^+}^2, a \in \mathbb{Z}_n^+\}$. The set E_p forms a cycle on $G_{\mathbb{Z}_n^+,p}$. Now denote our cluster as $G = (V, E)$ where V is defined as above and E represents a set of directed links. Then $G_{\mathbb{Z}_n^+,p}$ is isomorphic to G, hence following the rule in E_p we can define a valid ring in G. Furthermore, since $\forall p_i \neq p_j$ we can guarantee that $(0, p_i) \in E_{p_i}$ and $(0, p_j) \notin E_{p_i}$, and each p_i is guaranteed to describe a unique ring. □

One way to extend our approach to other AllReduce algorithms is to generalize TotientPerms (Algorithm 2) so that the E_p described in theorem 2 simply represents a *permutation* which we apply to the original node labeling, while keeping the edge relation, to create an isomorphic graph that describes the new AllReduce topology.

E.2 Bounding maximum hop count with TotientPerms

In this section, we argue that fitting a geometric sequence for choosing permutation provides an approximately $O(d\sqrt[d]{n})$ bound for the maximum diameter of a cluster with n nodes and degree d. Denote $x \equiv \sqrt[d]{n}$. We simplify the question to the following: given a contiguous set of numbers $\mathcal{N} = \{1, \ldots, n\}$ and a set of numbers from the geometric sequence $S = \{x^0, x^1, \ldots, x^{d-1}\}$, choose h numbers (allow repetition) $s_1, \cdot, s_k$ from S so that $m = \sum_{i=1}^h s_i$ for some $m \in \mathcal{N}$. Let $h = \kappa(m)$, find $\min_{m \in \mathcal{N}} \kappa(m)$.

Again for simplicity, assume $x \in \mathbb{Z}$. Then for a given $m \in \mathcal{N}$, we get the recursive relation $\kappa(m) = 1 + \kappa(m - x^i)$ where $i = \text{argmax}_{i \le d, x^i \le m}$. $m = N - 1$ gives the maximum $\kappa(N-1) = dx$.

The problem above is simpler than the one in TOPOOPT. In TOPOOPT, x is rarely an integer, and S is a projection of the geometric sequence $S = \{x^0, x^1, \ldots, x^{d-1}\}$ onto the

[3]Given that ring-AllReduce is the dominant AllReduce collective, we describe our algorithms based on ring-AllReduce. Appendix E.1 explains how to extend our algorithm to other AllReduce communication collectives.

Algorithm 4 `CoinChangeMod` pseudocode

```
1:  procedure CoinChangeMod($n$, $G$)
      ▷ Input $n$: Total number of nodes
      ▷ Input $G$: Network Topology
      ▷ Output $R$: Routings
        ▷ R is the routing result
2:      $R = \{\}$
        ▷ Acquire the set of "coins" from the topology,
        ▷ which are the choices of Algorithm 3
3:      $C =$ GetCoins($G$)
4:      for $i \in [1, N-1]$ do
          ▷ curr_dist denotes the "distance" of a value
          ▷ (node distance) counted by number of "coins"
5:          $curr\_dist[i] = \infty$
          ▷ curr_bt record a back-trace of "coins" to
          ▷ get to a value (node distance)
6:          $curr\_bt[i] = \infty$
7:      for $c \in C$ do
8:          $curr\_dist[c] = 0$
9:          $curr\_bt[c] = c$
10:     while $curr\_dist$ has at least one $\infty$ in it do
11:         for $i \in [1, N-1]$ do
12:             $new\_dist[i] = curr\_dist[i]$
13:             $new\_bt[i] = curr\_bt[i]$
14:             for $c \in C$ do
15:                 if $curr\_dist[(i-c)\ mod\ N] < new\_dist[i]$ then
16:                     $new\_dist[i] = cur\_dist[(i-c)\ mod\ N] + 1$
17:                     $new\_bt[i] = c$
18:         $curr\_dist = new\_dist$
19:         $curr\_bt = new\_bt$
         ▷ Construct the routing for each node distance from the back-trace
20:     $R =$ GetRouteSeq($curr\_bt$)
21:     return $R$
```

candidates (co-prime numbers with the size of a subset of node participating in AllReduce). The intuition still holds.

Note that when $\sqrt[d]{n} < 2$, it is advantageous to choose $x = 2$ and spend less degree to create a geometric sequence with a ratio of at least 2. In this case, the d factor becomes the actually used degree $d = \log_2 n$, and the bound holds at $O(\log_2\ n)$.

E.3 Coin Change Routing

Consider servers S_i and S_j that need to exchange AllReduce transfers but do not have a direct edge between them. We use a modified version of the classical coin change problem [52] to find an efficient routing path (line 19). In classical coin change, the goal is to find the minimum number of *coins* that would sum to a certain *total value*. Our ring generation rules enable us to treat the routing problem similarly. In particular, the p values of AllReduce permutations that have been selected in the AllReduce sub-topology are the coin values, and the difference between server i and j indices ($(j-i)\ mod\ n$) is the target total value that we want to achieve. The only difference is that our problem runs with *modulo n* arithmetic, as the server IDs wrap around in the ring structure. Algorithm 4 lists the pseudocode of `CoinChangeMod`.

E.4 OCS-reconfig Heuristic

Algorithm 5 describes the heuristic we use for OCS-reconfig. As mentioned in Section 4, our goals are (*i*) to have enough bandwidth for large transfer demands, (*ii*) while also minimizing the latency of indirect routing for nodes that do not have a direct link between them.

To achieve this goal in a reconfigurable interconnect, we propose a utility function that finds a balance between the two goals by maximizing the number of parallel links between high demand nodes but with a *diminishing return*. More formally, assume a network topology is represented by graph $G = (V, E)$ and each node has degree d. We define $L(i, j)$ to be the number of parallel links between node-pair (i, j). Let $T(i, j)$ be the amount of unsatisfied traffic demand. We define a topology G's utility function as follows:

$$Utility(G) = \sum_{\{i,j\} \in E} T(i,j) \times Discount(L(i,j)) \tag{1}$$

The *Discount* function can be defined in different ways; in Algorithm 5 and Algorithm 1's MP construction, we use

$$Discount(l) = \sum_{x=1}^{l} 2^{-x} \tag{2}$$

to reduce the utility of additional links exponentially. We can also explore other discount scaling, such as linear or factorial functions.

When the fabric is reconfigurable (as in OCS-reconfig), we collect the unsatisfied traffic demand every 50 ms and run Algorithm 5 to decide the new network topology. After the new topology is computed, we pause all the flows for 10 ms representing the reconfiguration delay of the OCS, apply the new topology, and then resume the flows with one or more corresponding physical links across the flow source and destination. The two-edge replacement algorithm from OWAN [87] in line 21 ensures the topology is connected, when we enable host-based forwarding.

F Modifications to SiP-ML

Since SiP-ML's SiP-Ring proposal is based on a physical ring topology, its reconfiguration algorithm has several constraints on wavelength allocation for adjacent nodes. Given that TopoOpt's physical topology is not a ring, directly applying SiP-Ring's optimization using the original C++ code causes SiP-ML to perform extremely poorly in our setup. To give SiP-ML a leg up, we observe that its formulation tries to optimize a utility function very similar to Equation 1 without the *Discount* part (i.e. $Discount = 1$), but with an integer liner program (ILP). While an ILP gives the optimal solution, its runtime makes it prohibitive for the amount of simulation parameters we explore. Therefore, we substitute the ILP

Algorithm 5 OCS-reconfig pseudocode

```
1:  procedure OCS-RECONFIG($V$, $T$, $d$, $L$)
        ▷ Input $V$: Nodes in the network
        ▷ Input $T$: Unsatisfied traffic demand matrix
        ▷ Input $d$: Node degree limit
        ▷ Input $L$: Number of links between ordered node-pair, initially zero
        ▷ Output $E$: Allocated links, initially empty
          ▷ Initially, E is empty
2:        $E = \{\}$
          ▷ Initially, each node has d available tx and rx interfaces
3:        for $v \in V$ do
4:            $available_{tx}[v] = d$
5:            $available_{rx}[v] = d$
          ▷ Create new links according to the demand list
6:        while $\exists i, j < |V| : i \neq j, available_{tx}[v_i] > 0, available_{rx}[v_j] > 0$ do
              ▷ allocate a direct connection for the highest demand pair
7:            $(v_1, v_2)$ = node-pair with highest demand in $T$
8:            $e$ = NewLink($v_1$, $v_2$)
9:            $E = E \cup \{e\}$
              ▷ Increment the number of parallel links from $v_1$ to $v_2$
10:           $L(v_1, v_2)$ += 1
              ▷ Scale the demand down by the number of links
11:           $T(v_1, v_2)$ ×= 1/2
              ▷ Update available interfaces
12:           for $v \in (v_1, v_2)$ do
13:               $available_{tx}[v_1]$ −= 1
14:               $available_{rx}[v_2]$ −= 1
                      ▷ Stop considering nodes with zero available interfaces
15:               if $available_{tx}[v_1] == 0$ then
16:                   for $u \in V$ do
17:                       Remove $(v_1, u)$'s entry from $T$
18:               if $available_{rx}[v_2] == 0$ then
19:                   for $u \in V$ do
20:                       Remove $(u, v_2)$'s entry from $T$
          ▷ Ensure the network graph is connected
21:       2-EdgeReplacement($E$, $T$)
          ▷ Updte route for host-based forwarding
22:       UpdateRoute($E$)
23:       return $E$
```

with Algorithm 5 with *Discount* $= 1$, a heuristic that tries to achieve a similar goal.

Note that the SiP-ML paper has another design called SiP-OCS, which is similar architecturally to TopoOpt. In the paper, SiP-OCS is proposed as a one-shot reconfiguration approach due to the long reconfiguration latency of 3D-MEMS based OCSs.

G Cost of Network Components

Table 2 lists the cost of network components we use in Section 5.2, namely NICs, transceivers, fibers, electrical switches, patch panels, and optical switches. The cost of transceivers, NICs, and electrical switch ports is based on the lowest available prices in official retailer websites [18, 28]. Note that for 200 Gbps, we use more 100 Gbps ports and fibers, because they were less expensive than high-end 200 Gbps and 400 Gbps components, or their price was not available. To estimate the cost of electrical switch ports, we consider Edgecore

[4] 200 G transceivers and switch ports are estimated as $2\times$ 100G cost.

Link bandwidth	Transceiver ($)	NIC ($)	Electrical switch port ($)	Patch panel port ($)	OCS port ($)	1×2 switch ($)
10 Gbps	20 [13]	185 [32]	94 [21]	100 [43]	520 [41]	25 [112]
25 Gbps	39 [15]	185 [32]	144 [23]	100 [43]	520 [41]	25 [112]
40 Gbps	39 [16]	354 [33]	144 [22]	100 [43]	520 [41]	25 [112]
100 Gbps	99 [12]	678 [34]	187 [24]	100 [43]	520 [41]	25 [112]
200 Gbps[4]	198 [12]	815 [35]	374 [24]	100 [43]	520 [41]	25 [112]

Table 2: Cost of network components.

bare metal switches with L3 switching and maximum number of ports to amortize the per port cost. The cost of NICs is taken from the Mellanox ConnectX series, and we consider two 2-port NICs as one 4-port NIC. We obtain the cost of the patch panel, OCS, and 1×2 optical switch directly from their suppliers, Telescent [43] and Polatis [41] (with 40% discount). The cost of transceivers matches that reported in Sirius [53].

To compute the network cost of Fat-tree and Ideal Switch, we consider number of nodes in a full bisection bandwidth Fat-tree. For example, a standard $k = 8$ Fat-tree has 80 switches with 64 ports, or 640 switch ports in total, in addition to 1 NIC per host and one transceiver per NIC and switch port. A TopoOpt system of 128 nodes with degree d uses $128 \times d$ NICs and transceivers, but $128 \times 2 \times d$ patch panel ports because of the look-ahead design. Note that the cost of optical components stays constant as link bandwidth increases, an inherent advantage of optics. Following prior work, we estimate the cost of fiber optics cables as 30 cents per meter [68] and select each fiber's length from a uniform distribution between 0 and 1000 meters [148]. We calculate the cost of TopoOpt based on $2d$ patch panels and 1×2 switches at each link to support its look-ahead design (§C). OCS-reconfig's cost is based on d OCSs connected to all servers in a flat topology.

H Impact of Server Degree on TopoOpt's Performance

Figure 27 shows the same setting as Figure 11 except that each server has a degree of eight ($d = 8$). The results show a similar trend: even though per server bandwidth has increased, the behavior of different network architectures remains consistent.

Next we do a sensitivity analysis of impact of server degree d on TopoOpt's performance. Specifically, we vary the degree of each server in TopoOpt for two link bandwidths: 40 Gbps and 100 Gbps. Figure 28 shows the trend for different DNN models. Both DLRM and CANDLE are network-heavy; therefore, they benefit more from the additional bandwidth obtained by increasing d. CANDLE's improvement is almost linear as degree goes up, as the strategy is closer to data parallel and the amount of bandwidth available to AllReduce operation increases linearly as well. In the case of DLRM, we observe a super-liner scaling when B = 100 Gbp because DLRM has one-to-many and many-to-one

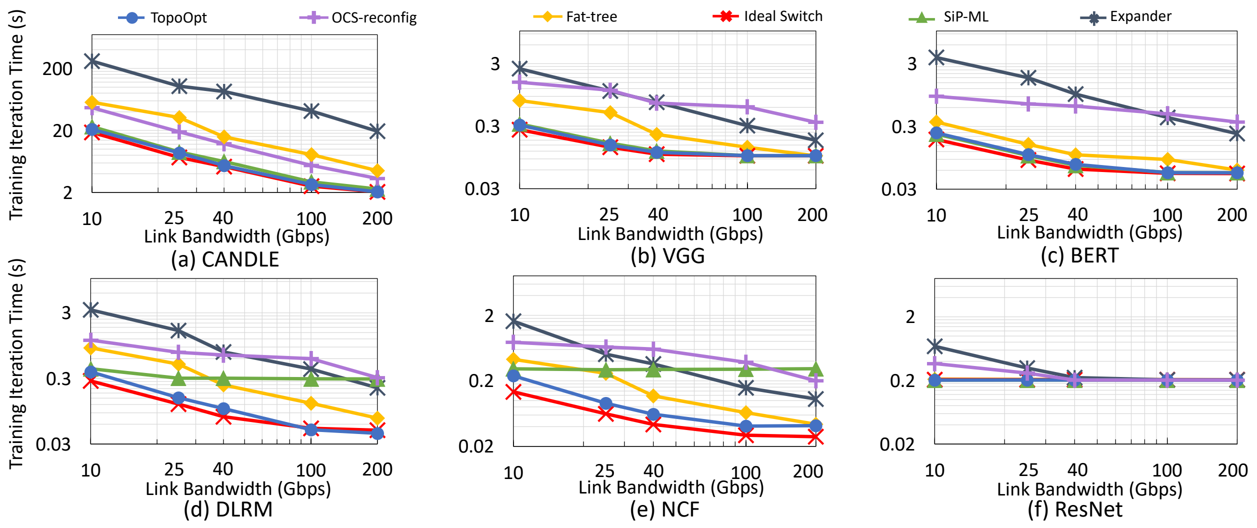


Figure 27: Dedicated cluster of 128 servers (d = 8).

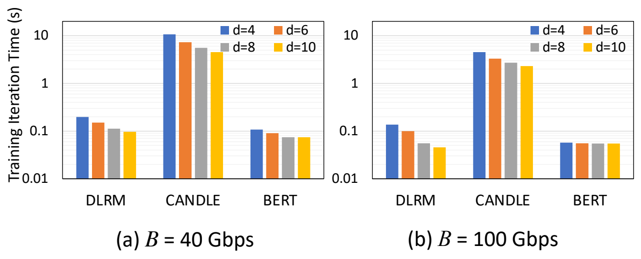


Figure 28: Impact of server degree (d) on performance.

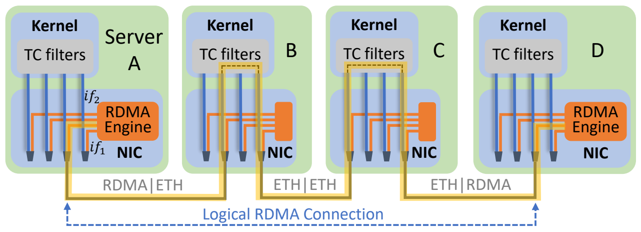


Figure 29: Host-based RDMA forwarding to create a logical RDMA connection between end hosts.

MP transfers which require a low hop count in the topology. As we increase d, TopologyFinder is able to find network topologies with much lower diameter, consequently benefiting the performance by both increasing bandwidth and reducing hop-count for MP transfers. Finally, BERT is mostly compute bound at higher bandwidth; hence, increasing the server degree and bandwidth per node has marginal impact on its iteration time.

I Enabling Host-based Forwarding in RDMA

To support a multihop TopoOpt interconnect using host-based forwarding, we enable RDMA RoCEv2 forwarding on all our HP NICs. RoCEv2 is an implementation of RDMA on top of UDP/IP protocol, by utilizing a particular UDP port (4791) and encapsulating an InfiniBand (IB) data packet. Hence, each RoCEv2 packet can be routed with its source and destination IP addresses. However, host-based forwarding is challenging in RDMA protocol, as the packet processing and memory access are offloaded to the NIC, and the host does not have access to individual packets. More precisely, if a packet's IP destination IP address does not match the NIC's IP address, the RDMA engine silently drops the packet.

To address this issue, we collaborated with engineers from Marvell, the provider of the firmware and driver for our HP NICs. The solution that came out of our collaboration does not require proprietary software or firmware, and is applicable to commodity NICs with the same ASIC. We will release our scripts publicly. At a high-level, we use a feature called *NPAR*, or *network partitioning*. It allows us to split each 25 Gbps physical interface into two *logical interfaces* in the hardware level: if_1 and if_2, as shown in the right-most port of server A in Figure 29. if_1 is a normal RDMA interface, where the RDMA engine of the NIC bypasses the kernel, and it has an IP address. This enables the upper layer software to consider if_1 as a normal RDMA interface. However, if_2 does not have an IP address and RDMA is disabled. if_2 has a different MAC address from if_1, and we use this address to split the traffic across if_1 and if_2. The traffic that needs to be forwarded uses the MAC address of if_2 and hence is delivered to the host networking stack instead of NIC's RDMA engine.

Furthermore, we establish a set of `iproute`, `arp`, and `tc flower` rules in Linux to enable the proper forwarding of packets. If two servers are directly connected, such as the third port of server A and the second port of Server B in Figure 29, we only need to indicate the outgoing interface on each of these servers. RDMA engines will handle the communication. However, for the connection between server A and D, we set the `iproute` and `arp` tables on server A and server D to dictate which port the traffic should go out, as well as the proper MAC address of the next server in the forwarding chain. In this case, the packets are delivered to the kernel. Then, on servers B and C, we set the `tc flower` rules to forward the packets to the next server with the proper MAC address. In these `tc flower` rules, we look-up the final destination IP and assert the routing that was computed by our algorithm.

Walk-through of an example of a packet going from server A to server D. In Figure 29, the RDMA engine of server A assumes server D is connected on the third port. It uses the kernel's routing tables for the destination MAC address, which is set to the MAC address of if_2 of the second port on server B. Therefore, a packet which starts as an RDMA packet of server A is treated as an Ethernet packet when it arrives at server B, and goes to server B's kernel. In the kernel, based on the packet's final destination IP of server D, server B redirects the packet to the fourth port, with destination MAC address set to if_2 of server C. In this connection, the packet is treated as a normal Ethernet packet. Finally, on server C, the kernel rewrites the destination MAC address to that of if_1 on the third port of server D, and redirects it to that port. In this connection, the outgoing Ethernet packet is considered an RDMA packet because of the destination MAC address. For the reverse connection from server D to A, the same process happens in reverse, to support a bidirectional connection.

With these forwarding rules, we construct logical RDMA connections between all pairs of servers. Upper layer communication libraries such as NCCL requires all-to-all connectivity, and they will utilize these connections. We also modify NCCL to be topology-aware, as certain pairs of servers are only connected through specific ports.

Compared to native point-to-point RDMA, this approach takes a performance penalty. Our experiments indicate the overhead is negligible when the amount of forwarded traffic is small. Our NICs currently support TCP forwarding offload. With firmware and driver modifications or future versions of the NICs, they will also support RDMA forwarding offload. This will further reduce the overhead of our approach.