Better Together: Jointly Optimizing ML Collective Scheduling and Execution Planning using SYNDICATE

Kshiteej Mahajan*, Ching-Hsiang Chu†, Srinivas Sridharan†, Aditya Akella$

University of Wisconsin - Madison, Facebook†, UT Austin$*

Abstract: Emerging ML training deployments are trending towards larger models, and hybrid-parallel training that is not just dominated by compute-intensive all-reduce for gradient aggregation but also bandwidth-intensive collectives (e.g., all-to-all). These emerging collectives exacerbate the communication bottlenecks despite heterogeneous network interconnects with ample multipath opportunities. In this work, we propose SYNDICATE, a systematic, general framework to minimize communication bottlenecks and speed up training for both state-of-the-art and future large-scale models and interconnects. SYNDICATE proposes a novel abstraction, the motif, to break large communication work as smaller pieces as part of execution planning. SYNDICATE also does joint optimization of scheduling and execution planning by rethinking the interfaces in the networking systems stacks used for ML training. Motifs afford greater flexibility during scheduling and the joint optimizer exploits this flexibility by packing and ordering communication work so as to maximize both network utilization and overlap with compute. This improves the speed of training state-of-the-art large models by 21-74%.

1 Introduction

Training machine learning (ML) models is a common-case workload at any data-driven enterprise. To keep up with evolving data and maintain a competitive edge, enterprises are employing more sophisticated features and more complex model architectures, and attempting to train faster at ever larger scales and to deploy high-quality models frequently.

These trends are exemplified by the deep learning recommendation model (DLRM). DLRM is used in recommendation systems at several large organizations. These models use a mixture of continuous and categorical features obtained from user data. The model architectures, which are themselves rapidly evolving, uses a mixture of multi-layer perceptrons and embedding table lookups. The model capacity and compute is increasing exponentially year-on-year [24].

At production scale, such state-of-the-art models use a mixture of data and model parallelism to efficiently scale-out to a large number of machines in the training cluster. This induces rich communication collectives such as all-reduce, all-to-all, collective-permute, and all-gather [21,24]. The resulting communication operations (comm-ops) are a major bottleneck to end-to-end training performance [24].

Evolution in networking infrastructure in training clusters [32, 35] (to include faster interconnects such as NVLink/NVSwitch, RoCE, Infiniband and support faster transports such as Remote Direct Memory Access (RDMA)) does not in itself help address these bottlenecks. These advancements need to be coupled with effective computation-communication scheduling and execution planning optimizations. These optimizations hide communication by maximizing overlap with compute and help maximize utilization of the networking infrastructure.

Unfortunately, existing scheduling optimizations [16, 18, 29] and execution planners [10,11,20,33,34] fall short. These works make several restrictive assumptions limiting their applicability to simplistic models, training settings, and networks. Communication schedulers make assumptions about the model and training architecture (simple layer-by-layer models [29] with data-parallel training) or deployment mode (Parameter Server-based [16, 18]), and the execution planners make simplifying assumptions about the nature of comm-ops (only all-reduce [10, 11, 20, 33, 34] or only push-pull [20]).

Moreover, existing solutions are point solutions in the optimization space and fail to *jointly* optimize scheduling and execution planning concerns. Schedulers today are unaware of the optionality during execution planning, such as parallel execution of two comm-ops over non-overlapping network communication channels, and might impose orders that fail to leverage such opportunities during execution planning. As a result, they leave significant room for optimization.

We seek a comm-op optimization framework that jointly optimizes planning and scheduling, applies to state-of-the-art large models with complex communication patterns, is generalizable to future large models and arbitrary network interconnects. Our framework should also encapsulates all possible optimization axes, and enables a systematic, thorough, automatic search through the space for optimal strategies.

Enabling systematic joint optimization of scheduling and execution planning is challenging. First, the communication systems stacks used for ML training today place scheduling and execution planning concerns in two different layers. The scheduler is co-located with ML training frameworks (such as PyTorch, TensorFlow) and the execution planner is co-located with communication libraries (such as NCCL, MPI). These are governed by two different developer communities and the scheduler interacts with the execution planner via a narrow, one-way API to just submit comm-ops. Moreover, the scheduler and the execution planner only accommodate fast, deterministic procedures so as to enable tight co-ordination across worker processes that peer with each other using parallel programming frameworks (such as MPI) during training.

Second, scheduling happens at the very coarse granularity of collectives submitted by the training application which limits scheduling flexibility as it leads to fewer opportunities to reorder communication work in time and efficiently pack communication work in space, i.e., over the heterogeneous mix of communication channels and bandwidth available in the networking infrastructure.

We propose SYNDICATE, a system for joint optimization of scheduling and execution planning with several innovations:

- SYNDICATE proposes a novel abstraction, the motif, to break large communication work in comm-ops into smaller units of communication work. Motifs afford greater flexibility, by helping pack and order communication work so as to maximize network multipath utilization and to maximize overlap with compute.
- Similar to query optimization backed by a well-defined relational algebra, we present a novel algebra atop motifs that systematically codifies the search space of correct, composable motif operators used to transform comm-ops into motifs and enables comm-op optimization.
- SYNDICATE rethinks the interfaces in the commmunication stack and enables joint optimization via the joint action of a control plane and a data plane. The former executes a time-intensive, non-deterministic joint optimization out-of-band without blocking the latter which enables fast execution of tightly co-ordinated motifs.
- We blend techniques used for optimal tensor operator fragmentation [19], DAG scheduling [15] and query replanning [22] to probabilistically search the joint optimization space to yield near-optimal comm-op optimization plans. We also introduces a novel shim-layer above existing communication libraries to enforce these plans.

We implement the enforcer atop existing communication libraries by extending the `torch-ucc` interface; the joint optimizer as a separate python process; and enable safe interaction between the central optimizer and the enforcer via a two phase commit protocol. We present the evaluation of several state-of-the-art models on a 128-GPU cluster with rich multipath opportunities. SYNDICATE demonstrates 21–74% faster training than the closest state-of-the-art [29] and is better than hand-optimized trainers.

2 Background

The compute and capacity of models has been increasing exponentially [1], with model training compute approaching 1000s of petaflop/s-days [9] and model capacity approaching trillions of parameters [24]. To train ever larger models, training clusters are scaling up to thousands of devices [21].

In this section, we give a short primer on the compute parallelization strategies used for ML training and the accompanying communication operations (comm-ops) that are issued. We also discuss how training network infrastructure is evolvong to deal with higher network loads.

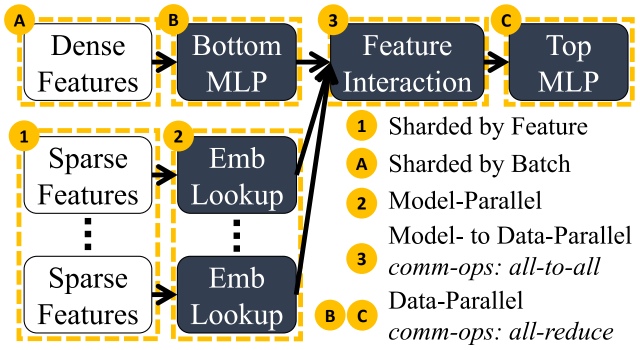


Figure 1: DLRM Model

2.1 Parallelization Strategies

We exemplify the different parallelization strategies via the Deep Learning Recommendation Model (DLRM). The largest DLRM used in production has trillions of parameters [12, 24, 26], making DLRM training especially challenging. DLRM uses a hybrid mix of parallelization strategies for different model parts (similar to BERT [13], Megatron [30], GPT [9]).

Figure 1 shows the DLRM model architecture. The training data comprises a mixture of dense continuous features and sparse categorical features (one-hot encoded or multi-hot encoded data), which are first mapped to a common embedding space using the bottom multi-layer perceptron (MLP) and the embedding table lookups respectively. The output embeddings go through a feature interaction phase and are then fed to the top MLP to get the recommender model output.

Data-Parallelism: With data-parallelism, all the model parameters are replicated across all the training devices and each device has a worker process computing parameter gradients in parallel. In the case of DLRM, the bottom MLP and top MLP use data-parallelism for training in production. These MLPs are compute intensive but not memory intensive and the MLP parameters fit within a single device memory.

Model-Parallelism: Data-Parallelism does not work for models with large capacity and with input datasets that cannot be trivially sharded into batches. With Model-Parallel training, the model is partitioned (and not replicated) across different devices. For DLRM, the embedding table lookup models and the input tables are large and memory-intensive, and as a result are paritioned across different devices during training resulting in model-parallel training.

Hybrid-Parallelism: As seen so far, different portions of DLRM training use different parallelization strategies. This is known as hybrid-parallelism. In the most general case, models can be replicated or partitioned in several different ways during training [19, 21, 25], resulting in hybrid-parallelism.

FSDP: Fully Sharded Data Parallelism [8] is a memory-efficient version of data-parallelism. It shards the model state (weights, gradients, optimizer state) for each layer of the model. During forward- or backward-propagation of a layer it enables data-parallel computation by first doing an all-gather of all the model state at all the devices and reshards the updated state post-computation by doing a scatter. This leads

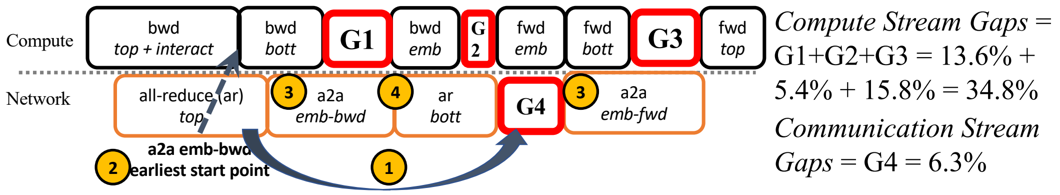


Figure 2: Gaps in DLRM Training Trace

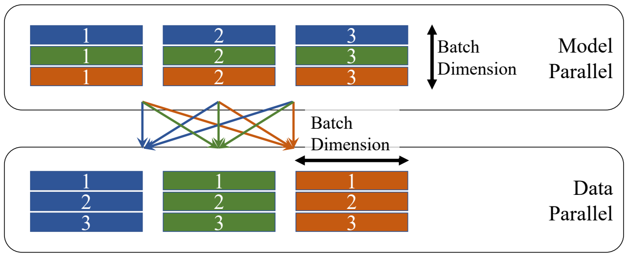


Figure 3: Illustration of all-to-all collective in DLRM

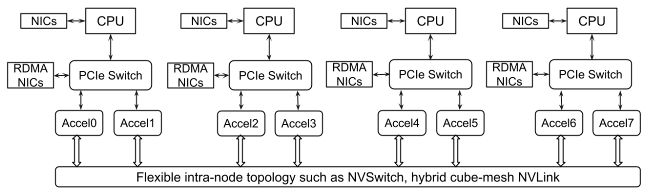


Figure 4: State-of-the-art system architecture of training cluster such as Nvidia DGX/HGX-like systems [24, 28]

to memory-efficiency and as a result allows to pack larger models in the same cluster resources.

2.2 Communication Operations (Comm-Ops)

Different parallelism strategies induce different comm-ops.

With data-parallel training, gradients computed at each worker process are aggregated layer-by-layer (during backward pass). Each aggregation yields an **all-reduce** collective.

After the embedding lookups in DLRM (② in Figure 1), each device has a vector for the table lookup models resident on those devices for all the samples in the batch, which needs to be reorganized and sharded along the batch dimension. This induces an **all-to-all** pattern of collective communication, as shown in Figure 3.

Collectives from the MPI standard [23]: In the general case, hybrid-parallel or FSDP model training [8, 14, 21] results in several types of comm-ops, ranging from all-reduce, all-to-all, collective-permute, all-gather, reduce-scatter to any collective defined in the MPI standard [23].

2.3 Evolving Network Infrastructure

The aforementioned comm-ops push increasing amounts of network traffic and the network infrastructure is adapting with fatter topologies and faster interconnects to ensure the needed throughput and latency.The network infrastructure in a state-of-the-art training cluster [24, 28] is shown in Figure 4. Each node has multiple CPU cores and accelerators such as GPUs, with frontend Network Interface Controllers (NICs) connected to the host CPUs and a dedicated RDMA NICs such as InfiniBand and RDMA over Converged Ethernet (RoCE) for each of the GPUs connected via PCIe switches. The RDMA NICs from across nodes can be connected with a dedicated network. The extensible design of this node allows to scale-out the network to interconnect thousands of nodes, forming a data-center scale training cluster. This cluster has heterogeneous mix of networking interconnects and protocols with varying throughput and latency guarantees. There are multiple communication channels between any two endpoints. At an intra-node level, a pair of GPUs can communicate via shared memory, NVLink, PCIe or the external network. At an inter-node level, any two GPUs can communicate via GPUDirect RDMA [27] or TCP/IP over Ethernet.

3 Motivation

Communication Bottleneck: Despite the networking infrastructure upgrades, the execution of comm-ops are a source of excessive delays in training. As an example of the issues that can arise in large model training, Figure 2 shows the execution of CUDA stream kernels on a randomly chosen GPU during a single iteration of production scale DLRM training [1]. The training creates a compute and a communication stream for serialized execution of tensor operator kernels and comm-op kernels, respectively. We note that there are several gaps during execution. A gap on a stream occurs when the stream is waiting for the result of kernel execution on the other stream. The compute stream gaps are wider (34.8%) and cumulatively larger than those in the communication stream (6.3%). This means that communication is a training bottleneck as it blocks compute for a third of the iteration. We now show that there are several opportunities to optimize comm-ops.

Better Scheduling Opportunity: Reordering of comm-ops improve compute/communication overlap. As shown in Figure 2 – ①: the top MLP all-reduce comm-op can be split judiciously and partially executed later to occupy the gap G4; ②: as a result the all-to-all backward comm-op can be pulled up to begin as soon as possible to reduce the gap G1.

Better Execution Planning Opportunity: Existing comm-ops do not efficiently utilize multiple communication channels available in heterogeneous network interconnects. We highlight this in Fig.2 – ③: both the all-to-all's can be broken up into smaller fragments of communication work and executed one fragment at a time to reduce incast and improve through-

[1]We show accurate percentages and hide low-level details.

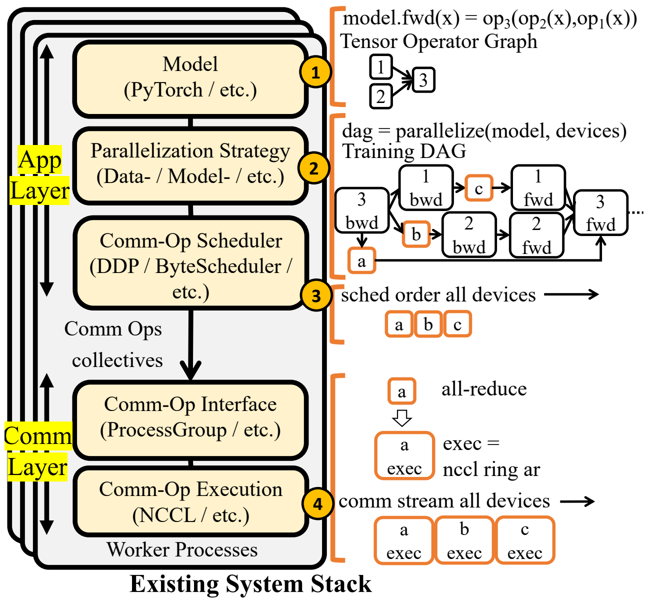
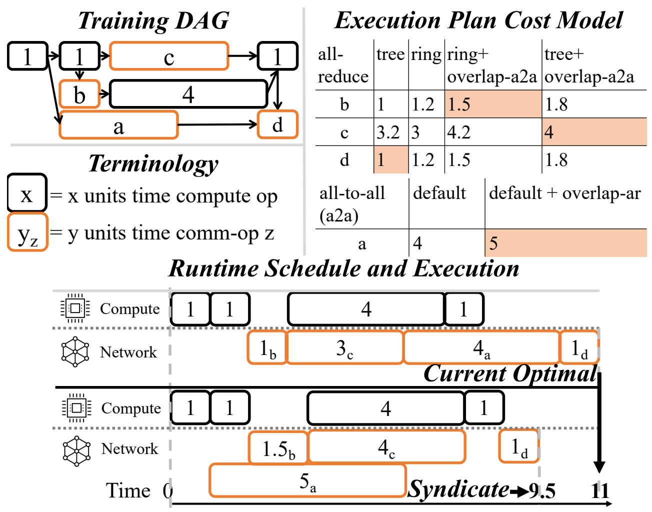

App Layer
Model (PyTorch / etc.)
Parallelization Strategy (Data- / Model- / etc.)
Comm-Op Scheduler (DDP / ByteScheduler / etc.)
Comm Ops collectives
Comm Layer
Comm-Op Interface (ProcessGroup / etc.)
Comm-Op Execution (NCCL / etc.)
Worker Processes
model.fwd(x) = op3(op2(x),op1(x))
Tensor Operator Graph
dag = parallelize(model, devices)
Training DAG
sched order all devices
all-reduce
exec = nccl ring ar
comm stream all devices
Existing System Stack

Figure 5: ML Training Communications Stack


Training DAG
Terminology
x = x units time compute op
y_z = y units time comm-op z
Execution Plan Cost Model
Runtime Schedule and Execution
Compute
Network
Current Optimal
Syndicate
Time

all-reduce	tree	ring	ring+ overlap-a2a	tree+ overlap-a2a
b	1	1.2	1.5	1.8
c	3.2	3	4.2	4
d	1	1.2	1.5	1.8

all-to-all (a2a)	default	default + overlap-ar
a	4	5

Figure 6: Motivating Example

put to reduce gaps G1 & G3; ④: all-to-all and all-reduce can be executed in parallel over communication channels with non-overlapping interconnects to start all-reduce sooner and drive higher network throughput to reduce gap G2.

Existing works to reduce communication overheads are optimal for specific training scenarios (PS architecture [16], layer-by-layer models [29], all-reduce collectives [20,34]; §7); and the scheduling and execution planning techniques proposed therein make restrictive assumptions, making it unclear as to how to compose and apply these different techniques towards hybrid-parallel training of large DLRM-like models.

A fundamental shortcoming of these works is that they do not explore *joint optimization* (§3.1) mainly because *existing interfaces in the communication stack used for ML training are not naturally amenable* (§3.2). We also note that *a collective is often too coarse-grained* to schedule communication work; breaking it up improves communication optimization flexibility (§3.3). We describe these issues next.

3.1 Disjoint Scheduling, Execution Planning

3.1.1 Communication Stack Overview

Figure 5 shows the two sets of layers in the communication stack used for ML training – the application (app) layer and the communication (comm) layer – and the four steps leading to execution of a comm-op over the network –

① **Model Definition:** The user defines a model by composing various tensor operations. The example shows a model declaration with three operators and its tensor operator graph.

② **Parallelization Strategy:** The parallelize module (e.g., `nn.DistributedDataParallel` in PyTorch) takes the model and the set of devices and converts the computation to a training DAG. The vertices are compute-ops or comm-ops and edges capture dependencies. Above we show the training DAG for a single iteration; the ops in the DAG are managed by spawning several worker processes on all the devices by using a parallel programming library such as MPI.

③ **Comm-Op Scheduling:** The communication scheduler takes the training DAG as input and decides a ordering for the comm-ops that maximizes compute/communication overlap. The scheduling procedure is deterministic and executes on all the worker processes so that the comm-ops are issued (and executed) every training iteration in *the same order on all the devices*. The default PyTorch order is FIFO.

④ **Comm-Op Execution Planning:** The comm layer at all the devices receive the comm-op from the app layer via an interface with a well-defined API (e.g., `ProcessGroup` in PyTorch). The execution planner on receiving each comm-op, assigns it an execution plan and queues it in the same order on the devices' i.e., GPU's communication stream for serialized execution. The example shows an all-reduce collective which binds to NCCL's ring all-reduce implementation during execution. Each collective typically has several options for its execution plan (e.g., ring or tree for all-reduce collective) and execution planners, such as NCCL, have network topology aware cost models that estimate the execution time for different options. Current execution planners are greedy and select the execution plan option with the least execution time. All worker processes bind to the same execution plan.

Thus, scheduling is an app-layer concern today, governed by schedulers in ML training frameworks as PyTorch, while execution planning is a comm-layer concern governed by execution planners in communication libraries as NCCL. As these are not jointly optimized, several inefficiencies arise, which we exemplify next.

3.1.2 Example to highlight suboptimality

Figure 6 illustrates the lost opportunities due to a lack of joint optimization of scheduling and execution planning. The network topology is similar to that illustrated in Figure 4. The training DAG in this example has four collectives: `a` is an all-to-all collective and `b`, `c`, `d` are all-reduce collectives.

There are several execution plan options for each collective. There is a cost associated with each option which measures the execution time over the network. For all-reduce, the execution plan options are tree all-reduce or the ring all-reduce, both using NVLink and GPUDirect RDMA. For all-to-all, there are two options: pairwise exchange between all processes either using NVLink and GPUDirect RDMA or using PCIe complex and TCP/IP over ethernet. With the latter option for all-to-all, all-to-all and all-reduce can be overlapped. We compare the iteration time of the current solution against SYNDICATE.

Current Solution: The execution planner greedily selects the fastest option for each collective resulting in a training iteration time of 11 units.

SYNDICATE Solution: SYNDICATE realizes that by jointly making changes to the scheduling order and execution plan choices there is opportunity to overlap all-to-all with all-reduce and speed-up communication by utilizing network heterogeneity. The current solution's scheduling order executes all-to-all last and does not allow overlap. SYNDICATE's scheduling order executes all-to-all collective at the very beginning and allows overlap. The execution plan choices made by SYNDICATE are shaded in the execution plan cost model in Figure 6. SYNDICATE's execution planner is not greedy and chooses a slower execution plan for all-to-all so as to allow for parallel execution of all-to-all and all-reduce over non-overlapping interconnects in the network. Overall, this results in a training iteration time of 9.5 units.

Joint optimization is beneficial but current interfaces are not amenable as we discuss next.

3.2 Interface constraints Joint Optimization

The training DAG scheduler in the ML processing frameworks is unaware of optionality (e.g., an all-reduce can be executed by as a ring all-reduce or a tree all-reduce) present lower down the stack during execution planning. A trivial extension of the existing interface is to expose the training DAG to the comm layer and push the scheduling concern down the stack to co-locate it with the execution planner. Exposing the training DAG down the stack is necessary to ensure that any reordering of comm-ops down the stack does not lead to *dependency violations*: a child comm-op cannot be ordered before a parent comm-op as otherwise it can lead to a *deadlock*. This enables joint optimization without dependency violations. However, the joint optimization problem is NP-hard and the joint optimization procedure requires a time-intensive, randomized algorithm (§4.3). As a result, this procedure can delay comm-op execution due to its time-intensive nature. To make matters worse, this randomized procedure, may lead to divergent scheduling orders across different processes. This can lead to *out-of-sync* issues, wherein if the collectives are not submitted in the same order across two different processes then it results in a *deadlock* where each process waits for the other process to issue the same collective as itself. As a result, the existing interfaces are unable to trivially accommodate

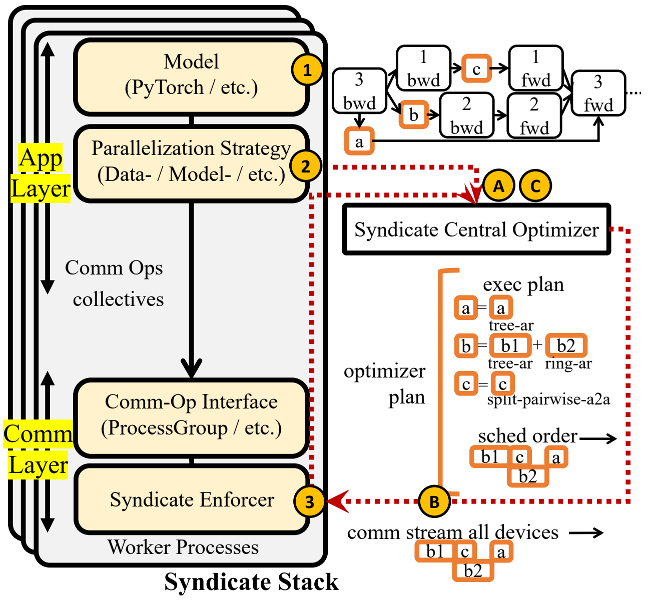


Figure 7: Overview of SYNDICATE's ML training Communication Stack

joint optimization of these concerns.

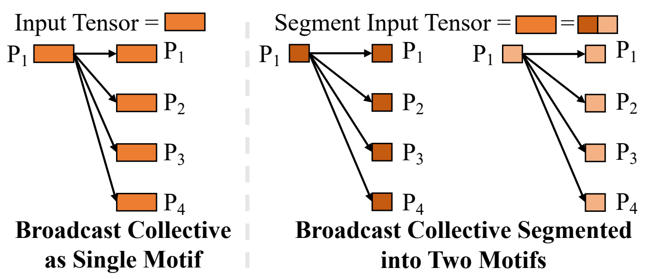
3.3 Issues with Coarse-Grained Scheduling

Scheduling today happens at the granularity of user-submitted comm-ops i.e., collectives. Communication libraries such as NCCL, submit each comm-op as a kernel on the GPU communication stream and a kernel cannot be context-switched during execution. This means that once a comm-op is scheduled for execution it cannot be stopped mid-execution. This leads to limited scheduling flexibility in space and time.

If the payload is very large then each network transfer in the comm-op, once scheduled for execution, occupies the network links for a long time. Likewise, if the pattern of network transfers is large (e.g., a clique of network transfers) then the comm-op gang schedules transfers on a large fraction of network interconnects. Comm-ops, if scheduled as-is, thus have large communication work orders and limit the ability to both context switch and efficiently pack communication work over available heterogeneous interconnects.

4 SYNDICATE Design

SYNDICATE changes the interfaces in the communication stack to enable joint optimization of scheduling and execution planning. It builds on the *motif* abstraction to enable deconstructing comm-ops into smaller work units along a few dimensions and allow finer-grained scheduling. In this section, we start with an overview of the new interfaces and the new modules in SYNDICATE's communication stack and how it enables joint optimization (§4.1). We then explain the motif abstraction (§4.2), the joint optimizer design (§4.3), and enforcement of the joint optimizer's decisions (§4.4).



Input Tensor =
P_1
P_1
P_2
P_3
P_4
Broadcast Collective as Single Motif
Segment Input Tensor = =
Broadcast Collective Segmented into Two Motifs

Figure 8: Example showing segmentation of broadcast collective. Left half shows the broadcast collective as a single motif. Right half shows broadcast collective segmented into two motifs, where each motif broadcasts one half of the bytes from the original tensor.

4.1 Overview

Figure 7 shows SYNDICATE's communication stack. Notably, we propose two new entities: a central optimizer and an enforcer. SYNDICATE co-locates scheduling and execution planning concerns in the centralized joint optimizer. The central optimizer generates an optimizer plan. This plan contains instructions on how to execute as well as how to schedule the comm-ops during training and is conveyed to the enforcer on each worker process. In this regards, *the central optimizer is the the control plane while the enforcer is the data plane*. We propose interfaces (A, B, C) between the central optimizer and the communication stack. These interfaces are out-of-band and asynchronous, meaning that the data plane does not, in any circumstances, block execution of a comm-op waiting for control plane instructions.

We now go over the workflow in SYNDICATE. We divide it into control plane workflow and the data plane workflow.

Stepping through the control plane workflow –

A **Joint Optimization:** The central optimizer pulls the training DAG from the app layer and the network topology from the comm layer. The optimizer uses these inputs to construct the execution plan cost model and does joint optimization to yield the optimizer plan (§4.3).

B **Optimizer Plan Distribution:** The joint optimizer plan is sent to the enforcer on all the worker processes (§4.3).

C **Feedback:** The central optimizer pulls comm-op performance statistics from the enforcer to help refine the execution plan cost model and potentially redo joint optimization (§4.4).

Stepping throught the data plane workflow –

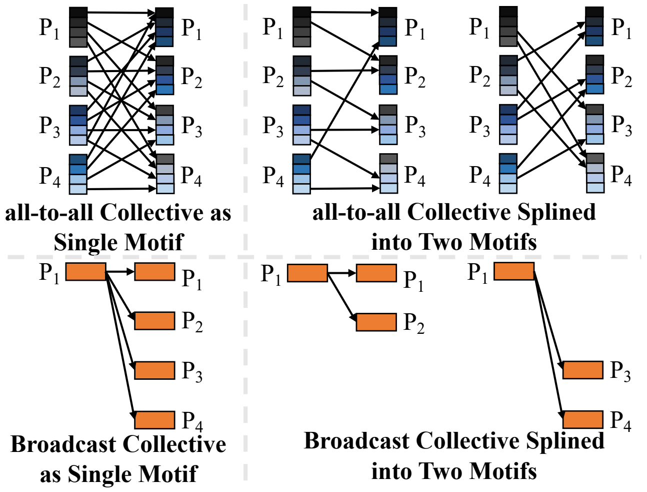
1 **Model Definition:** The user defines a model by composing tensor operators. This yields a computation graph (§3.1).

2 **Parallelization Strategy:** The computation graph is converted to a training DAG (§3.1). The comm-ops from the training DAG are submitted every training iteration as-is to the comm layer in the default FIFO order without applying any scheduling optimizations.

3 **Optimizer Plan Enforcer:** The comm-ops are executed as instructed by the central optimizer (§4.3).

4.2 Motif Abstraction

A motif is a logical grouping of several point-to-point transfers over the network. The enforcer schedules and executes



P_1
P_2
P_3
P_4
all-to-all Collective as Single Motif
all-to-all Collective Splined into Two Motifs
Broadcast Collective as Single Motif
Broadcast Collective Splined into Two Motifs

Figure 9: The left half shows all-to-all and broadcast collective as a single motif that bundles the transfers from all the source processes to all the destination processes. The right half shows both the collectives splined into two motifs, where each motif transfers the same payload from the source process to one half of the destination processes.

communication work at the granularity of motifs. A motif once issued to the device e.g., as a kernel on GPU communication stream is non-preemptible and occupies network resources until its communication work is completed.

Conversion of comm-op to motifs: Each comm-op i.e., a collective has two attributes: a payload (typically tensors) and a pattern of network transfers. We propose two transformation operators to slice a comm-op either along the payload dimension or the pattern dimension into one or more motifs. As compared to the original comm-op, each motif represents a smaller unit of communication work (with reduced payload size and/or smaller pattern). Since SYNDICATE does scheduling at the granularity of motifs, this enables finer-grained scheduling with increased opportunities for making more frequent scheduling decisions in time to enable better overlap of compute/communication and packing smaller units of communication work more efficiently over the network resources.

4.2.1 Motif Transformation Operators

Segmentation and Splining: SYNDICATE proposes two transformation operators: segmentation and splining. Segmentation splits the payload into smaller payload segments. Splining splits the pattern of network transfers into smaller patterns. Figure 8 and Figure 9 illustrates these operators.

Transformation Operator Algebra: We now formalize the algebra for the motif transformation operators. The goal of this algebra is to state concrete rules for transforming comm-ops into motifs. This formalization succinctly encodes: (1) correct and admissible motif transformations, (2) correct and admissible transformation combinations, and (3) a structured space for all possible operator compositions. We denote the segmentation operator by $\overset{s}{=}$ and the splining operator by $\overset{p}{=}$. These rules are by no means exhaustive and are extensible.

We first present the symbols used in the algebra.

$\Vert$	: Parallel Execution Permitted
$\overset{s}{=}$	: Segmentation Transformation
$\overset{p}{=}$	: Splining Transformation
$\mathtt{N}$	: Total number of Processes
$\mathtt{PG[IDs]}$	: Process $\mathtt{IDs}$ involved in a Collective
$\mathtt{T}_i\mathtt{[0:N,0:D]}$	: $\mathtt{N}$ Tensors of size $\mathtt{D}$ on Process $\mathtt{P}_i$ with $\mathtt{T}_i\mathtt{[j,0:D]}$ destined for Process $\mathtt{P}_j$
$\mathtt{M}_{AA}\mathtt{(T}_i\mathtt{[0:N,0:D],\ PG[0:N])}$	: collective with all-to-all pattern of transfers with tensor $\mathtt{T}_i$ as payload executing on each Process $\mathtt{P}_i$ for all i in $\mathtt{0:N}$
$\mathtt{M(T}_i\mathtt{[a}_i\mathtt{:b}_i\mathtt{,x}_i\mathtt{:y}_i\mathtt{],\ PG[IDs])}$	: Motif $\mathtt{M}$ executing on each Process $\mathtt{P}_i$ for all i in $\mathtt{IDs}$ with payload = tensor $\mathtt{T}_i\mathtt{[j,\ x}_i\mathtt{:y}_i\mathtt{]}$ from Process $\mathtt{P}_i$ destined to Process $\mathtt{P}_j$ for all j in $\mathtt{a}_i\mathtt{:b}_i$

Next, we present the algebraic rules that we use in the context of DLRM to transform all-to-all into motifs. The algebra for other comm-ops is in the Appendix §A.1.
Segmented All-To-All:

$$\mathtt{M}_{AA}\mathtt{(T}_i\mathtt{[0:N,0:D],\ PG[0:N])} \overset{s}{=} \Vert_{s=0}^{\frac{D}{d}-1} \mathtt{M(T}_i\mathtt{[0:N,\ s*d:(s+1)*d],\ PG[0:N])}$$

With segmentation, the payload to be sent from a source process to all the destination processes is split into segments of size $\mathtt{d}$ (= $\mathtt{T}_i\mathtt{[0:N,\ s*d:(s+1)*d]}$). Segmentation of all-to-all in this way, yields $\frac{D}{d}$ motifs where each motif sends a payload of size $\mathtt{d}$ from a source process keeping the set of destination processes the same. $\mathtt{d}$ is a parameter and controls the number of motifs associated with the input all-to-all.
Splined All-To-All:

$$\mathtt{M}_{AA}\mathtt{(T}_i\mathtt{[0:N,0:D],\ PG[0:N])} \overset{p}{=} \Vert_{c=0}^{\frac{N}{n}-1} \mathtt{M(T}_i\mathtt{[(i+c*n)\%N:(i+(c+1)*n)\%N,\ 0:D],}\ \mathtt{PG[0:N])}$$

With splining, the pattern of network transfers in the all-to-all with each source process sending the payload to all the $\mathtt{N}$ destination processes is broken down into smaller patterns, where each source process $\mathtt{P}_i$ sends the same payload as before to $\mathtt{n}$ destination processes (= $\mathtt{(i+c*n)\%N:(i+(c+1)*n)\%N}$). Splining of all-to-all in this way, yields $\frac{N}{n}$ motifs. Here, $\mathtt{n}$ parameterizes the all-to-all splining operator with larger $\mathtt{n}$ breaking the all-to-all into fewer motifs with larger sub-patterns.
Composition of Operators: Note that the all-to-all collective, $\mathtt{M}_{AA}\mathtt{(:)}$, is in fact a special case single motif (with $\mathtt{d} = \mathtt{D}$ and $\mathtt{n} = \mathtt{N}$). These operators can be composed and recursively break a motif into several more finer-grained motifs. While fine-grained motifs are beneficial for scheduling flexibility, there is a fixed overhead associated with dispatching a motif as a kernel on GPU communication stream and launching it during execution and too fine-grained motifs are not desirable as these overheads can slow-down communication.
Physical Plan for Motif: Each motif bundles together several network transfers. Physical plan determines the physical interconnects that each network transfer is assigned to. Figure 10 shows an example of a physical plan for the broadcast collective. The broadcast collective is first broken into three motifs. The physical plan maps the motif to point-to-point network transfers over various interconnects available in the network. The figure also shows a toy network topology where the GPU for process $P1$ connects to all other GPUs via both PCIe and NVLink interconnects. The three motifs can be multiplexed over different interconnects. The physical plan for the first motif does a `memcpy` on process $P1$. The physical plan for the remaining two motifs use PCIe and NVLink in a mutually exclusive manner. This allows the point-to-point transfers in the three motifs to execute in parallel, maximizing utilization of multipath opportunities available in the network.

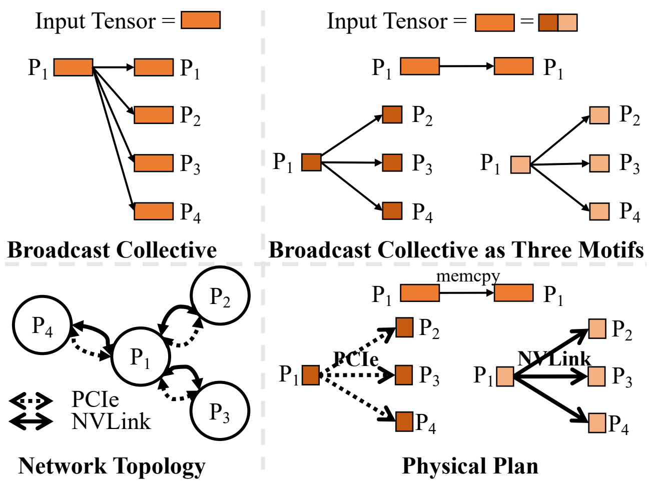


Figure 10: Physical Plan for Broadcast Collective

4.3 Central Optimizer

The central joint optimizer is responsible for minimizing training iteration time by minimizing communication overheads. The optimizer determines the optimizer plan by systematically navigating the vast space of potential schedules.

The optimizer plan has two pieces: the execution plan and the scheduling order, containing instructions regarding how to execute and how to schedule comm-ops respectively. The execution plan transforms each comm-op in the training DAG into one or more motifs. The scheduling order decides the order of execution of motifs.
Exponential Search Space: There is a lot of optionality in the execution plans for each comm-op. The transformation operators can be composed to break a comm-op into motifs in several different ways. Let us say that there are atleast $\mathtt{O}$ execution plan options for each comm-op and there are $\mathtt{C}$ comm-ops in the training DAG, then this results in $\mathtt{O}^{\mathtt{C}}$ unique execution plan options for all the comm-ops in a DAG.
Cost of each Execution Plan: For a particular execution plan, there is an optimal scheduling order for the motifs that maximizes overlap of compute/communication and minimizes training iteration time. This training iteration time with the optimal scheduling order is the cost of the execution plan.
Problem Statement: The centralized joint optimizer takes a

Pseudocode 1 Probabilistic Search

```
1: Training DAG with Greedy Execution Plan $G_*$

2: procedure MCMCSEARCH
3:     $C_*$, sched_order$_*$ = optSched($G_*$)
4:     while true do
5:         $G_{temp}$ = transform($G_*$)  ▷ change execution plan for a comm-op at random
6:         $C_{temp}$, sched_order$_{temp}$ = optSched($G_{temp}$)
7:         $\alpha(C_{temp} \mid C_*)$ = min(1, exp($\beta$ * ($C_*$ - $C_{temp}$)))
8:         $G_*$, $C_*$, sched_order$_*$ = $G_{temp}$, $C_{temp}$, sched_order$_{temp}$ with $\alpha$ prob.
9:     end while
10:    return $G_*$, sched_order$_*$
11: end procedure

12: procedure OPTSCHED
13:    comm_q                ▷ queue of ready communication motifs
14:    compute_q             ▷ queue of ready compute tasks
15:    comp_time = 0
16:    comm_time = 0
17:    sched_order
18:    while comp_time ≤ comm_time and comp_q != $\phi$ do
19:        comp_task = fifoDequeue(comp_q)
20:        comp_time += comp_task.time()
21:        sched_order.schedule(comp_task)  ▷ enqueue ready motifs, compute
22:    end while
23:    while comm_time ≤ comp_time and comm_q != $\phi$ do
24:        comm_motif, startTime = criticalPathDequeue(comm_q)
25:        comm_time = max(comm_time, startTime+comm_motif.time())
26:        sched_order.schedule(comm_motif)  ▷ queue ready motifs, compute
27:    end while
28:    return max(comm_time, comp_time), sched_order
29: end procedure
```

training DAG G and the network topology as inputs. We take a training DAG that unrolls compute-ops and comm-ops across two training iterations to enable cross-iteration optimizations. The aim of the joint optimizer is to take these inputs and find the execution plan with minimal cost. The joint optimizer outputs the optimizer plan, which has the execution plan and the scheduling order that minimizes overall cost.

4.3.1 Joint Optimization Procedure

The key idea in SYNDICATE is to do *probablistic search* over the exponentially large search space. We use MCMC search as outlined in Pseudocode 1.

MCMC Search: The joint optimizer starts with the default execution plan for the training DAG (denoted by G_*), wherein all the comm-ops are greedily assigned the execution plan choice with the minimum possible execution time. Thereafter, a comm-op is chosen at random and it is assigned a random execution plan option. This changes the motifs associated with this particular comm-op, keeping all the other motifs constant and yields a temporary execution plan for the training DAG (denoted by G_{temp}). The cost i.e., the execution time of this training DAG is calculated using the optSched (line 11 in Pseudocode 1) procedure which is a greedy scheduling heuristic to always dequeue motifs on the critical path in the DAG to maximize overlap of communication motifs with compute tasks or other communication motifs (in case the two communication motifs have non-overlapping physical plans). This update to the execution plan is probablistically sampled using the Metropolis-Hastings algorithm [17] and retained in G_* (line 8 in Pseudocode 1). This tends to behave as a greedy search over the search space with an ability to escape local minimas [17, 19].

Pseudocode 2 Distributed Optimizer Plan Enforcer

```
1: Exec Plan $\mathbb{E}_{colls} = \{..., coll_i^{in} : \{motif_{i,j}^{out}\}, ...\}$   ▷ optimal execution plan
2: Scheduling Order $\mathbb{S} = \{..., motif_{i,j}^{out} : seq_{i,j}^{num}, ...\}$   ▷ optimizer scheduling order
3: Progress Queue $pq$   ▷ thread-safe priority queue containing ready motifs

4: procedure ENFORCEEXECPLAN($coll^{in}$)   ▷ app submits comm-op to comm layer
5:     $\{motif^{out}\} = \mathbb{E}_{colls}[coll^{in}]$   ▷ comm-op is deconstructed into motifs
6:     for all $motif^{out} \in \{motif^{out}\}$ do
7:         $seq^{num} = \mathbb{S}[motif^{out}]$
8:         $pq$.INSERT($priority$=$seq^{num}$, $motif^{out}$)
9:     end for
10: end procedure

11: procedure ENFORCEORDER   ▷ runs in a separate thread and enforces order
12:     $nextMotifSeqNum = 0$
13:     while true do
14:         while $pq$.TOP().$priority$ != $nextMotifSeqNum$ do
15:                         ▷ busy loop until next in order motif is ready
16:         end while
17:         $nextMotifSeqNum$ += 1
18:         $\{motif\} = pq$.POP()
19:         $\{motif_{tensors}\} = \{motif\}$.EXECUTE()
20:         REPACK($\{motif_{tensors}\}$)
21:     end while
22: end procedure
```

Search Termination: MCMC search is terminated if the search procedure exceeds the time budget assigned for search or if the search procedure does not find a better joint optimizer plan for more than half of the total elapsed search time.

4.4 Enforcer

The central optimizer commits the same joint optimizer plan, comprising of the execution plan and the scheduling order to the enforcer on each worker process. The enforcer is responsible for *tightly co-ordinating this plan across all the worker processes during training* so as to avoid deadlocks and out-of-sync issues (§3.2). The application thread spawned by the ML processing framework at each worker process submits comm-ops to the comm layer every training iteration. With SYNDICATE, these comm-ops are submitted one-at-a-time in FIFO order. These comm-ops are intercepted by the enforcer. The enforcer is responsible for execution of these comm-ops and preparing the result of these comm-ops (tensors) to unblock the next application thread operation (compute-op or comm-op typically waiting on a CUDA stream) that is waiting on these tensors.

The enforcer takes the responsibility of executing these comm-ops as per the instructions of the optimizer plan and preparing the output tensors once ready. It does so in three steps. First, on intercepting a comm-op, it enforces the execution plan by breaking it into motifs. Second, it enforces the desired scheduling order of execution of motifs. Third, as and when motifs complete, it checks for completion of comm-ops and packages the output of individual motifs into the comm-ops output tensors. Pseudocode 2 shows the procedure to enforce the execution plan and the scheduling order contained in the joint optimizer plan. The repacking of tensors to comm-op output tensors happens after successful execution of each motif (line 20 in Pseudocode 2).

Enforcing Execution Plan: The enforcer is layered as a shim on top of existing comm-op execution layer i.e., differ-

Compute (TFLOPS)	120 (FP32)/ 1000 (FP16)
HBM	256 GB, 7.2 TB/s
DDR	1.5 TB, 200 GB/s
Scale-up bandwidth	1.2 TB/s (uni-directional)
Scale-out bandwidth	8 x 100 Gbps (uni-directional)
Host NW	2 × 100 Gbps

Table 1: Configuration of each node in our cluster

ent communication libraries such as NCCL, MPI, UCX. The app layer submits comm-ops to the comm layer using the interface between them and is immediately trapped by the `enforceExecPlan` procedure (line 4 in Pseudocode 2). This procedure deconstructs the comm-op to one or more motifs, each assigned a sequence number that captures the priority of this motif in the current training iteration. This sequence number is contained in the scheduling order of the joint optimizer plan. These motifs are enqueued into a priority queue using the sequence number as the priority.

Enforcing Scheduling Order: The `enforceOrder` procedure (line 11 in Pseudocode 2) enforces the scheduling order and runs in a thread separate from the `enforceExecPlan` procedure. This procedure maintains a priority counter that is incremented sequentially and is reset at the end of each training iteration. This counter maintains the priority of the next expected motif(s). In case of overlapping motifs, two or more motifs can be assigned the same priority. The `enforceOrder` procedure busy loops until the priority of the motif at the top of the priority queue matches the value in the priority counter. It busy loops until the `enforceExecPlan` enqueues the next expected motif. This ensures that the enforcer on all the worker processes executes motifs in the same order.

Replanning: We measure the wait time in the busy loop and send it as feedback to the central optimizer. If wait times in every iteration consistently add up to more than a threshold (= 5% of iteration time), then we redo joint optimization at the optimizer to explore a different optimizer plan.

5 Implementation

We implement the central optimizer as a separate module in python. The central optimizer simulates the execution of different execution plan choices as part of the MCMC search procedure until the search procedure terminates and yields a joint optimizer plan. The central optimizer runs on one of the machines in the training cluster and interacts with the various enforcers on the worker processes via RPCs. We build a two-phase commit (2PC) protocol using RPCs so that the same joint optimizer plan is safely committed by the central optimizer to all the enforcers. After the 2PC protocol is complete, the enforcers switch to the new joint optimizer plan from the subsequent training iteration. This ensures that out-of-sync issues are avoided whenever transitioning to a new joint optimizer plan.

We implement the enforcer in the Unified Collective Communication (UCC) library interface [6]. We implement the enforcer routines: `enforceExecPlan` routine in the `main thread` and the `enforceOrder` routine in the `progressLoop thread` in the `torch-ucc` interface [5].

Model	A1	A2	A3	A4
Num parameters	95B	793B	845B	332B
MFLOPS per sample	89	638	784	60
Num of emb tables	∼ 100s	∼ 1000s	∼ 1000s	∼ 1000s
Emb table dim ([min, max], avg)	[4, 192] avg: 68	[4, 384] avg: 93	[4, 960] avg: 231	[32, 128] avg: 72
Avg pooling size	27	15	17	49
Num MLP layers	26	20	26	43
Avg MLP size	914	3375	3210	682
Batch Size	512	1024	512	4096
Parallel Paradigm	Hybrid	Hybrid	Hybrid	FSDP [8]

Table 2: Models in our workload. Model A5 and A6 descriptions are in §6.2.

6 Evaluation

6.1 Testbed

We experimented with our prototype on a production-scale cluster using off-the-shelf NVIDIA HGX-2 based systems. Specifically, each node hosts dual-socket CPUs, 8 NVIDIA V100 GPUs that are fully-connected using NvSwitch, 2 front-end host NICs, and 8 back-end RoCE NICs to allow RDMA communication between GPUs across nodes. Table 1 summarizes the node configuration; we deployed 16 such nodes.

The testbed runs CentOS-8 and CUDA 11.4 with NVIDIA driver 470.57.02. For distributed training of DLRM models, we used PyTorch 1.11 (nightly) with the extension of Process Group UCC [5] and the latest UCC library [6], which can take advantage of various transports such as NCCL 2.10.3 [2] and UCX-based collectives [31] for dynamically selecting optimal execution planing of collective operations.

6.2 Workloads

We tested SYNDICATE in production across a breadth of scenarios; see Table 2.

Vary Model Architectures: We experiment with three different model architecture families. A1-A4 are Recommendation Models (DLRM [24]), A5 is an NLP Model (XLM-R-XL [14]), A6 is a CV Model (RegNetZ [7]).

Vary Model Sizes: We have progressively wider MLPs and higher number of embedding tables from model A1 to A3.

Vary Parallelization Strategies: Models A1-A3 are Hybrid Parallel. Model A4 is Hybrid Parallel with Fully Sharded Data Parallel (FSDP) for data parallelism [8]. Model A5 is Model Parallel. Model A6 is Data Parallel.

Vary Topologies: We vary the number of nodes (and hence GPUs) used from our testbed.

6.3 Metrics

We measure the following metrics

(1) **Training Throughput:** We measure the training throughput in terms of recommendation queries per second (A1-A4) or words per sec (A5) or images per sec (A6). Higher throughput is desirable.

(2) **Compute Idling:** We measure the idle gaps in the

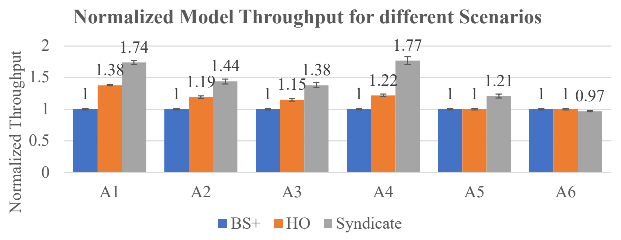


Figure 11: Training performance for SYNDICATE compared against baselines

GPU's compute stream and report it as a percentage of the total iteration time. Lower compute idling is desirable.

(3) **Normalized Metric:** We normalize the metric (such as throuput) against baseline using the formula – $\frac{\text{Metric with SYNDICATE}}{\text{Metric with Baseline}}$.

We run each experiment 5 times and plot the mean and standard deviation.

6.4 Baselines

We compare SYNDICATE against the following baselines.

- ByteScheduler+ (BS+): ByteScheduler [29] proposes LIFO scheduling policy for maximizing overlap of compute and communication. Their implementation only supports all-reduce in layer-by-layer models and does not have support for all-to-all collectives. We emulate ByteScheduler (BS) via our own implementation that is co-located with PyTorch framework. To emulate the bayesian optimizer used in ByteScheduler for tensor partitioning, we aid our BS scheduler with an oracle (BS+) that optimally segments tensors in all collectives.
- Hand Optimized Model (HO): Existing execution planners do not optimize all-to-all collectives and existing schedulers do not have support for DLRM-like models. We hand optimize both the scheduling policy in PyTorch (and also provide it the benefit of the segment oracle) and the choice of execution plan for each collective (including all-to-all) in the UCC library. In this regards, HO models the best possible solution with today's placement of scheduling and execution planning concerns in exiting stacks.
- SYNDICATE-Exec (S-Exec): We disable scheduling optimizations in SYNDICATE. We do so by using PyTorch's default scheduler that does FIFO scheduling to estimate the cost of each execution plan during joint optimization.
- SYNDICATE-Sched (S-Sched): We disable execution planning optimizations in SYNDICATE. We do so by disabling the MCMC search procedure and find the optimal scheduling order using SYNDICATE's scheduling heuristic (and also provide it the benefit of the segment oracle to optimally segment collectives).

6.5 Evaluation on Testbed

Figure 11 compares training performance for SYNDICATE against the BS+ and HO baselines for all the models. The Y-axis measures the model throughput normalized against that of BS+.

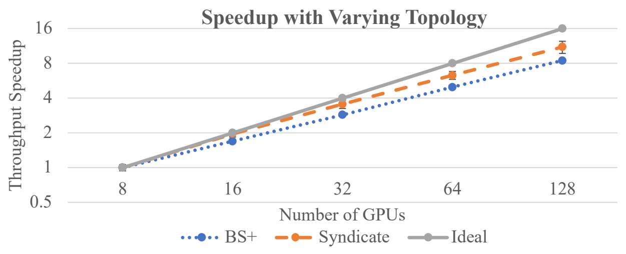


Figure 12: Speedup with varying topology sizes

Vary Model Sizes: SYNDICATE outperforms BS+ by a factor of 1.74x, 1.44x, 1.38x for Models A1, A2, A3, respectively. SYNDICATE outperforms HO baseline by a factor of 1.26x, 1.21x, 1.2x for Models A1, A2, A3, respectively. Note that gains diminish with increased model sizes. The embedding tables are not compute-intensive and do not contribute to increasing the MFLOPs per sample but have a high memory footprint (and contribute to higher number of parameters) and induce progressively more communication bandwidth-hungry all-to-all's to transfer a large number of embeddings. On the other hand, MLPs are compute intensive and increase the model compute (MFLOPS per sample). On detailed analysis, we found that the larger embedding table sizes amplify the amount of time spent in all-to-all in a training iteration to a higher degree than the contribution of increased MLP compute time; which skewed the overall ratio of communication to compute and diminished the opportunity to overlap communication and compute with SYNDICATE.

Vary Model Architectures and Parallelization Strategies: SYNDICATE is effective across a range of parallelization strategies and outperforms the BS+ baselines for A2 (hybrid-parallel, recommendation model) by 1.44x, A4 (FSDP, recommendation model) by 1.77x, and A5 (model-parallel, NLP model XLM-R-XL) by 1.21x. SYNDICATE is slightly worse-off for A6 (data-parallel, RegNetZ) by 0.97x. For this model there are no opportunities to overlap comm-ops as they have linear dependency and BS+ solution (LIFO with oracle tensor partitions) is the optimal solution (similar to other CV model families, e.g., ResNet [29]). SYNDICATE is slightly worse-off due to the overheads of SYNDICATE's enforcer. The gains are for A4 are significantly higher than that for A2 despite both the models having similar model sizes and model architecture. We found that FSDP parallel strategy for A4 offers a richer set of collectives: reduce-scatter and all-gather in addition to all-to-all and all-reduce. This allows SYNDICATE to find a schedule and an execution plan that overlaps atmost three comm-ops for A4 at the same time (compared to atmost two for A2). For Model A5, we find that BS+'s LIFO schedule is optimal and hence HO does not yield any improvements. For A5, the 1.21x gains with SYNDICATE are due to better execution plan with overlap of allgather and reduce-scatter during backward pass. For Model A6, we observe no improvements with SYNDICATE. This is primarily because A6 is data

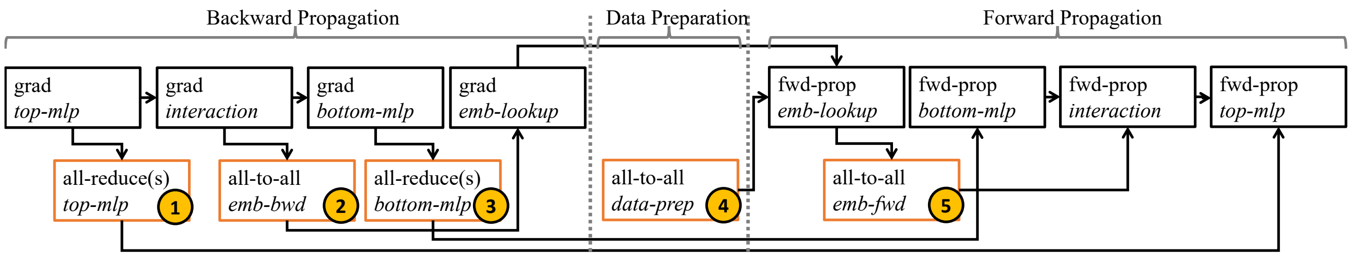


Figure 13: DLRM Training DAG. The numbers represent the order in which the PyTorch modules (`nn.DistributedDataParallel` and `nn.EmbeddingBag`) submit these collectives.

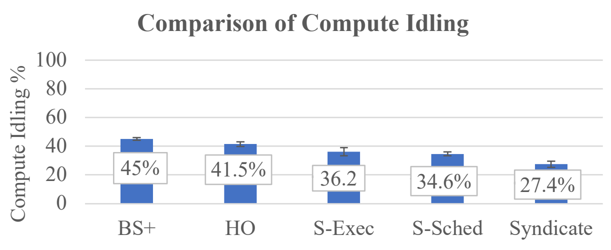


Figure 14: Comparison of Compute Idling

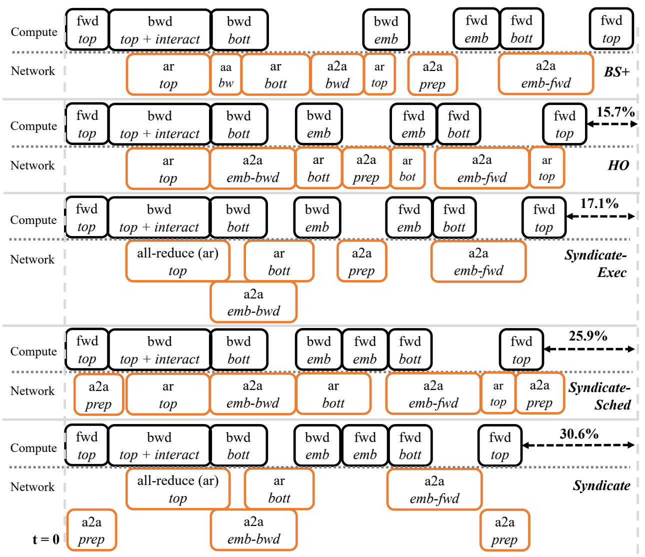


Figure 15: Zooming in on every DLRM iterations for different systems

parallel and BS+ LIFO schedule is optimal for data parallel models. Furthermore, the collectives in training DAG for A6 have serial dependencies across themselves with no room to optimize execution plan by overlapping collectives.

Vary Topologies: Figure 12 compares throughput speedup for Model A2 with SYNDICATE against BS+. We report the speedup relative to throughput on a single node. We note that SYNDICATE scales better than BS+ and is closer to ideal speedup line. SYNDICATE is better at opportunistically utilizing the increasing communication bandwidth as the cluster size scales out.

6.5.1 Sources of Improvement

Compute Idling:

Figure 14 compares the compute idling metric for SYNDICATE against baselines for Model A2. We observe that compute idling with SYNDICATE is 27.4% and is 1.64x, 1.51x, 1.32x, 1.26x less than the BS+, HO, S-Sched, S-Exec baselines, respectively. This shows that SYNDICATE is better at overcoming communication bottlenecks and achieves higher overlap of compute and communication than any other baseline. It also highlights that joint optimization is beneficial as it outperforms the S-Sched and S-Exec baselines. Next, we zoom-in on each training iteration to better understand the reasons for lesser compute idling.

Zooming in on an Iteration: We collect traces for execution of DLRM with different systems using PyTorch Kineto [4]. We illustrate these traces[2] to zoom-in on the execution of compute and communication events on the GPU streams for a single training iteration for Model A2 in Figure 15. We also show the training DAG in Figure 13 for reference. We explain these traces one system at a time.

BS+: BS+ prioritizes the execution of the most recently submitted collective (LIFO). For reference, Figure 13 shows the order of submission of collectives by DLRM PyTorch trainer. To achieve LIFO, tensors in collectives need to be optimally segmented and as explained before, we use a segment oracle to do so. BS+ is the worst-performing baseline. BS+ prioritizes execution of a2a-emb-bwd over ar-top-mlp[3], which is beneficial. However, to its detriment, it also prioritizes execution of ar-bottom-mlp over a2a-emb-bwd despite a2a-emb-bwd being on the critical path. Delaying a2a-emb-bwd also delays bwd-emb compute, which delays a2a-data-prep.

HO: To amend the drawbacks of BS+, we hand optimize the scheduling order to prioritize the execution of a2a-emb-bwd as well as a2a-data-prep before ar-bottom-mlp. We also add support for greedy execution planning for a2a collective (ar greedy optimization is available out-of-the-box). We observe that HO improves the iteration time by 15.7% as compared to BS+. We observe that the key reason for this improvement is that HO unblocks bwd-emb and fwd-emb compute sooner

[2]We hide low-level events and absolute timing information for confidentiality and legal reasons.

[3]We use a2a and ar as short hand for all-to-all and all-reduce, respectively.

and enables better overlapping of ar-bottom-mlp with these compute blocks.

S-Exec: With S-Exec, we observe that the iteration time is further improved and is 17.1% better as compared to BS+. We observe that despite placing limiting constraints on scheduling (default FIFO scheduling), the joint optimizer in SYNDICATE finds an execution plan that assigns two different communication channels to a2a and ar and enables better communication-communication overlap by leveraging heterogeneity in the network. The ar's use a communication channel over NVLink (for intra-node) and GPUDirect RDMA (for inter-node). The a2a's use a non-intersecting communication channel over PCIe (for intra-node) and TCP/IP over Ethernet (for inter-node). Such communication-communication overlap is not possible with HO and BS+ as they use traditional communication stack and interfaces therein only permit one-at-a-time execution of comm-ops with greedy execution plan.

S-Sched: With S-Sched, we observe that iteration time is further improved and is 21.9% faster than BS+, despite the constraints on execution planning (we also handicap S-Sched with choosing the default execution plan option, which is sub-optimal, for a2a). The primary reason for the improvement is that SYNDICATE's scheduler finds a superior comm-op scheduling order and SYNDICATE's enforcer enables enforcing of this order. SYNDICATE's scheduling order moves a2a-data-prep from the i^{th} iteration and moves it back in time as to overlap it with the fwd-top-mlp and bwd-top-mlp compute blocks in the $(i\text{-}1)^{th}$ iteration. The enforcer design enables this ordering due to the presence of busy loop in the `enforceOrder` procedure in Pseudocode 2. The enforcer blocks execution of all the comm-ops in the very first training iteration until a2a-data-prep collective for the next batch is submitted by the application layer. This increases the training iteration time only for the first iteration but significantly improves the training iteration time for all the subsequent iterations by unlocking pipelining.

SYNDICATE*:* With SYNDICATE, we observe that iteration time is faster than all the baselines and is 30.6% faster than BS+. We observe that as compared to S-Sched, SYNDICATE is able to hide the overheads of ar-top-mlp by completely overlapping it with compute. SYNDICATE enables this by leveraging network heterogeneity and enabling communication-communication overlap of ar-top-mlp and a2a-emb-bwd. S-Sched, due to its execution planning constraints is unable to do so and in its scheduling order has to partially push ar-top-mlp to the very end where it cannot be overlapped with compute. In this way, SYNDICATE's joint optimizer maximizes compute-communication overlap by leveraging the benefits of communication-communication overlap.

SYNDICATE's Optimizer Plan for DLRM: Here, we summarize the key highlights of the optimizer plan that SYNDICATE finds for DLRM Model A2. In our study, we find that these observations also hold for Model A1 and Model A3.

Data Prefetch The scheduling order proposed by the optimizer

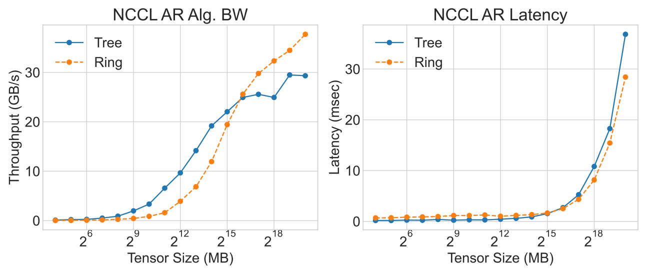


Figure 16: Effect of different execution plans on all-reduce performance

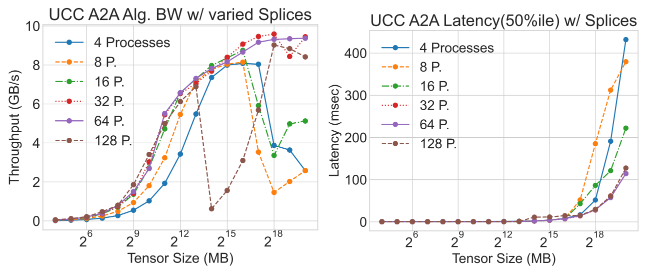


Figure 17: Effect of different execution plans on all-to-all performance

moves a2a-data-prep back in time from the i^{th} iteration to the $(i\text{-}1)^{th}$ iteration. As mentioned before, this is made possible by SYNDICATE's enforcer.

a2a-ar Overlap The execution plan proposed by the optimizer overlaps all-to-all collective with all-reduce collective over two separate communication channels as explained before. SYNDICATE binds both the a2a's to the 4-way splined execution plan, the ar-bottom-mlp to the ring all-reduce execution plan and the ar-top-mlp to the tree all-reduce execution plan. This maximizes multipath network utilization and also enables greater communication-compute overlap.

6.6 Microbenchmarks

We use the communication microbenchmark, PARAM [3] to systematically understand the space of execution plans for different collectives to better understand the choices made by SYNDICATE's optimizer plan. SYNDICATE uses these microbenchmarks as a cost model for its joint optimizer. We highlight a subset of these microbenchmarks and explain the various choices made by SYNDICATE for Model A2.

Execution planning options for all-reduce: Figure 16 shows the effect of different all-reduce execution planning options in our testbed. We see that the optimal execution plan depends on the input message size. The optimal plan at small message sizes is tree all-reduce motif whereas the optimal plan at large message sizes is ring all-reduce motif. Bottom MLP is wider and induces larger (O(100's of MB) vs. top MLPs O(MB)) all-reduce collectives and explains choice of ring all-reduce and tree all-reduce for ar-bottom-mlp and ar-top-mlp, respectively.

Execution planning options for all-to-all:

Figure 17 shows the effect of different all-to-all execution planning options. We note that the optimal plan at small mes-

		TicTac [16]	P3 [18]	Blink [34]	ByteScheduler [29]	Syndicate
Execution	Network Throughput	×	×	✓	✓	✓
	Network Heterogeneity	×	×	✓	×	✓
	Network Ops	Push-Pull	Push-Pull	All-Reduce	Push-Pull; All-Reduce	Send-Recv; Collectives
Scheduling	Preemptible	✓	✓	—	✓	✓
	Models	General DAGs	Layer-by-Layer	—	Layer-by-Layer	General DAGs
	Frameworks	PS	PS	—	PS; $\sim$ P2P	PS; P2P
	Policy	DAG Optimal	LIFO	—	LIFO	DAG Optimal
Joint Optimization		×	×	—	×	✓

Table 3: Comparison of systems optimizing communication operations for training workloads

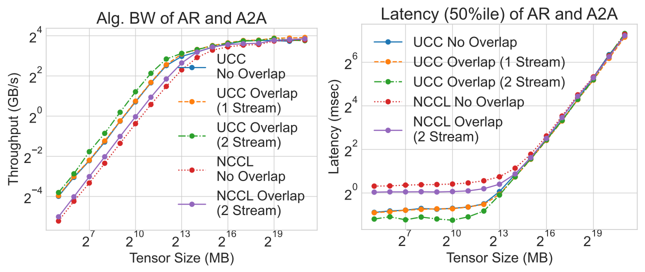


Figure 18: Effect of different execution plans for all-to-all and all-reduce overlap

sage sizes is the 1-way splined motif (simultaneous transfers to 128 destination processes from all source processes), at intermediate message sizes is the 2-way splined motif (64 Processes), and at large message sizes is the 4-way splined motif (32 Processes). The effects of incast are significant as we increase the message sizes and more splining helps reduce incast. DLRM training induces large a2a's (O(GB) message size) and SYNDICATE chooses the 4-way splined execution plan.

Execution planning options for overlap of all-reduce and all-to-all: Figure 18 shows that the optimal execution plan is the one where all-reduce uses NCCL implementation and all-to-all use UCC implementation over non-overlapping communication channels (i.e., 2 streams). With this implementation all-to-all is CPU-driven and uses the PCIe complex and TCP/IP over Ethernet, while all-reduce is GPU-driven and uses NVLink complex and GPUDirect RDMA. This choice is optimal (as opposed to vice versa) as all-reduce also does compute (gradient aggregation) which is faster with GPUs. SYNDICATE uses this execution plan for overlap of all-to-all and all-reduce.

7 Other Related Work

Several works speed-up training by optimizing two main concerns of communication operations: scheduling and execution. Table 3 shows comparison of SYNDICATE against several state-of-the-art systems.

Scheduling concerns looks at reordering communication operations to maximize overlap of compute and communication. The optimal scheduling policy is dependent on factors such as the model architecture, and the parallelization strategy/framework.

Execution concerns look at accelerating individual communication operations through efficient transport over all communication links. These optimizations propose optimal batching to improve link utilization, propose multipath in collectives to make better use of heterogeneous links in the network, and enable preemption to enable scheduling optimizations.

Existing scheduling works fall short in generalizing optimally to all scenarios and they exercise only a subset of optimizations as highlighted in Table 3. Crucially, unlike SYNDICATE, existing works do not jointly optimize both these concerns.

8 Conclusion

We propose SYNDICATE that rethinks communication scheduling granularity and the interfaces in the communication stack for ML training to enable joint optimization of scheduling and execution planning. Using the novel notion of motifs and a split control/data plane architecture SYNDICATE achieves improvements of 21-74% for production scale large-model training as it better utilizes the network multipath opportunities in emerging training clusters.

References

[1] Ai and compute. https://openai.com/blog/ai-and-compute. Accessed: 2021-08-26.

[2] Nvidia collective communication library: Optimized primitives for collective multi-gpu communication. https://github.com/NVIDIA/nccl. Accessed: October 3, 2022.

[3] Parametrized recommendation and ai model benchmark. https://github.com/facebookresearch/param. Accessed: October 3, 2022.

[4] Pytorch kineto. https://github.com/pytorch/kineto. Accessed: 2021-08-26.

[5] Pytorch process group third-party plugin for ucc. https://github.com/facebookresearch/torch_ucc. Accessed: October 3, 2022.

[6] Unified collective communication (ucc). https://ucfconsortium.org/projects/ucc/. Accessed: October 3, 2022.

[7] A. Adcock, V. Reis, M. Singh, Z. Yan, L. van der Maaten, K. Zhang, S. Motwani, J. Guerin, N. Goyal, I. Misra, L. Gustafson, C. Changhan, and P. Goyal. Classy vision. https://github.com/facebookresearch/ClassyVision, 2019.

[8] M. Baines, S. Bhosale, V. Caggiano, N. Goyal, S. Goyal, M. Ott, B. Lefaudeux, V. Liptchinsky, M. Rabbat, S. Sheiffer, A. Sridhar, and M. Xu. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale, 2021.

[9] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[10] M. Cho, U. Finkler, D. Kung, and H. Hunter. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In *Proceedings of the 2nd SysML Conference*, 2019.

[11] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. Panda. NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems. In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, 2020.

[12] A. Desai, L. Chou, and A. Shrivastava. Random Offset Block Embedding Array (ROBE) for CriteoTB Benchmark MLPerf DLRM Model : 1000$\times$ Compression and 2.7$\times$ Faster Inference, 2021.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[14] N. Goyal, J. Du, M. Ott, G. Anantharaman, and A. Conneau. Larger-scale transformers for multilingual masked language modeling. *arXiv preprint arXiv:2105.00572*, 2021.

[15] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 81–97, 2016.

[16] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[17] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[18] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko. Priority-based parameter propagation for distributed dnn training. *arXiv preprint arXiv:1905.03960*, 2019.

[19] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *SysML 2019*, 2019.

[20] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. 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, 2020.

[21] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[22] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic query re-planning using {QOOP}. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 253–267, 2018.

[23] Message Passing Interface Forum. http://www.mpi-forum.org/. Accessed: October 3, 2022.

[24] D. Mudigere, Y. Hao, J. Huang, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. R. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao. High-performance, distributed training of large-scale deep learning recommendation models. *CoRR*, abs/2104.05158, 2021.

[25] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[26] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[27] NVIDIA. NVIDIA GPUDirect. `https://developer.nvidia.com/gpudirect`, 2011. Accessed: October 3, 2022.

[28] NVIDIA. DGX A100 System User Guide. `https://docs.nvidia.com/dgx/pdf/dgxa100-user-guide.pdf`, 2021. Accessed: October 3, 2022.

[29] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[30] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

[31] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, et al. Ucx: an open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[32] M. Smelyanskiy. Zion: Facebook next-generation large memory training platform. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–22. IEEE Computer Society, 2019.

[33] Y. Ueno and R. 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, May 2019.

[34] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica. Blink: Fast and generic collectives for distributed ml. *arXiv preprint arXiv:1910.04940*, 2019.

[35] J. Yin, S. Gahlot, N. Laanait, K. Maheshwari, J. Morrison, S. Dash, and M. Shankar. Strategies to deploy and scale deep learning on the summit supercomputer. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 84–94, 2019.

A Appendix

A.1 Transformation Operator Algebra

We now present the algebra for the motif transformation operators. We denote the segmentation operator by $\overset{s}{=}$ and the splining operator by $\overset{p}{=}$. Note that the algebraic rules presented below are not exhaustive and are extensible. Here, we present the algebraic rules that we use in the context of DLRM to transform all-reduce and all-to-all collectives into motifs. We first go over the various symbols used in the algebra.

`N` :	Total number of Processes
`PG[0:N]` :	Process `IDs` involved in a Motif
$\mathtt{T}_i\mathtt{[0:D]}$:	Tensor of size `D` on Process $\mathtt{P}_i$
$\mathtt{T}_i\mathtt{[0:N,0:D]}$:	`N` Tensors of size `D` on Process $\mathtt{P}_i$ with first dimension indicating destination Process `ID`
$\|$:	Parallel Execution
$\rightarrow$:	Sequential Execution
$\overset{s}{=}$:	Segmentation Transformation
$\overset{p}{=}$:	Splining Transformation
`AR` :	`all-reduce motif`
`AA` :	`all-to-all motif`
$\mathtt{RE}_r$:	`reduce motif` with root Process $\mathtt{P}_r$
`RS` :	`reduce-scatter motif`
$\mathtt{BC}_r$:	`broadcast motif` with root Process $\mathtt{P}_r$
`AG` :	`all-gather motif`
$\mathtt{COLL(T}_i\mathtt{[:],\ PG[IDs])}$:	Motif `COLL` with input tensor $\mathtt{T}_i$ executing on each Process $\mathtt{P}_i$ for all i in `IDs`

We now present the algebraic rules for transforming the all-reduce motif using the segment and spline operators.

Segmented All-Reduce: First, we show application of the segment operator which splits the input tensor at all the processes and converts an all-reduce motif into smaller all-reduce motifs over the splits. Each smaller all-reduce motif are independent and can execute at the same time in parallel.

$$\mathtt{AR(T}_i\mathtt{[0:D],\ PG[0:N])} \overset{s}{=} \|_{s=0}^{\frac{D}{d}-1} \mathtt{AR(T}_i\mathtt{[s*d:(s+1)*d],\ PG[0:N])}$$

Ring All-Reduce: Next, we show an instance of the spline operator that divides the pattern in original all-reduce into two sub-patterns: reduce-scatter motif followed by the all-gather motif. The reduce-scatter motif does aggregation and the all-gather motif broadcasts the aggregated result. The reduce-scatter and all-gather motifs induce a pattern of communication over a ring, where the processes are arranged in a ring and the tensor is divided into `N` pieces. Each process $\mathtt{P}_i$ does a point-to-point transfer of the `(i+r)%N` piece to its neighboring process in the ring in the $\mathtt{r}^{th}$ round for `N` rounds.

$$\begin{aligned}
\mathtt{AR(T}_i\mathtt{[0:D],\ PG[0:N])} &\overset{c}{=} \mathtt{RS(T}_i\mathtt{[0:D],\ PG[0:N])} \\
&\quad \rightarrow \mathtt{AG(T}_i\mathtt{[0:D],\ PG[0:N])} \\
\mathtt{RS(T}_i\mathtt{[0:D],\ PG[0:N])} &= \mathtt{ring\ pattern\ of\ communication} \\
\mathtt{AG(T}_i\mathtt{[0:D],\ PG[0:N])} &= \mathtt{ring\ pattern\ of\ communication}
\end{aligned}$$

Tree All-Reduce: Next, we show an instance of the spline operator that divides the pattern in the original all-reduce into three smaller sub-patterns: reduce motif followed by a smaller all-reduce motif followed by broadcast motif. The same spline operator algebraic can be recursively applied to the smaller all-reduce motif. Recursive application results in a hierarchical tree pattern of communication where several reduce motifs first aggregate results in a tree like fashion at a single root process and several broadcast motifs broadcast the aggregated result from the root process in a tree like fashion until it is updated at all the processes. Each reduce motif results in a convergent pattern of communication where all the processes involved in the reduce send their tensors to the root process where it is aggregated. Each broadcast motif results in a divergent pattern of communication where the root process sends its tensor to all the processes involved in the broadcast motif.

$$\begin{aligned}
\mathtt{AR(T}_i\mathtt{[0:D],\ PG[0:N])} &\overset{c}{=} \|_{c=0}^{\frac{N}{n}-1} \mathtt{RE}_{c*n}\mathtt{(T}_i\mathtt{[0:D],\ PG[c*n:(c+1)*n])} \\
&\quad \rightarrow \mathtt{AR(T}_i\mathtt{[0:D],\ PG[}\cup_{c=0}^{\frac{N}{n}-1}\mathtt{\ c*n])} \\
&\quad \rightarrow \|_{c=0}^{\frac{N}{n}-1} \mathtt{BC}_{c*n}\mathtt{(T}_i\mathtt{[0:D],\ PG[c*n:(c+1)*n])} \\
\mathtt{RE}_j\mathtt{(T}_i\mathtt{[0:D],\ PG[j:j+n])} &= \mathtt{convergent\ pattern\ of\ communication} \\
\mathtt{BC}_j\mathtt{(T}_i\mathtt{[0:D],\ PG[j:j+n])} &= \mathtt{divergent\ pattern\ of\ communication} \\
\mathtt{AR(T}_i\mathtt{[0:D],\ PG[}\cup_{c=0}^{\frac{N}{n}-1}\mathtt{\ c*n])} &= \text{recursive application of } \overset{c}{=} \text{ induces} \\
&\quad\ \mathtt{tree\ pattern\ of\ communication}
\end{aligned}$$

Segmented and Splined All-To-All: Next, we show examples of segmenting and splining an all-to-all collective into smaller motifs. With segmentation, the tensor at all the processes is split and the original all-to-all is deconstructed into several smaller all-to-all motifs over the split tensors. With splining, the pattern of communication in the original all-to-all motif with a clique of point-to-point transfers between all the processes is broken down into smaller all-to-all motifs with smaller patterns where each process $\mathtt{P}_i$ initiates point-to-point transfers to a subset of destination processes (with ids in the range `(i+c*n)%N:(i+(c+1)*n)%N`). Here, `n` parameterizes the all-to-all splining operator with larger `n` resulting in breaking the original all-to-all into fewer all-to-all motifs with larger sub-patterns.

$$\begin{aligned}
\mathtt{AA(T}_i\mathtt{[0:N,0:D],\ PG[0:N])} &\overset{s}{=} \|_{s=0}^{\frac{D}{d}-1} \mathtt{AA(T}_i\mathtt{[0:N,\ s*d:(s+1)*d],\ PG[0:N])} \\
\mathtt{AA(T}_i\mathtt{[0:N,0:D],\ PG[0:N])} &\overset{c}{=} \|_{c=0}^{\frac{N}{n}-1} \mathtt{AA(T}_i\mathtt{[(i+c*n)\%N:(i+(c+1)*n)\%N,\ 0:D],} \\
&\quad \mathtt{PG[0:N])}
\end{aligned}$$