RESEARCH-ARTICLE

Chronos: Prescheduled circuit switching for LLM training

SUNDARARAJAN RENGANATHAN, Stanford University, Stanford, CA, United States

NICK MCKEOWN, Stanford University, Stanford, CA, United States

Published: 08 September 2025

Citation in BibTeX format

SIGCOMM '25: ACM SIGCOMM 2025 Conference
September 8 - 11, 2025
Coimbra, Portugal

Conference Sponsors:
SIGCOMM

https://doi.org/10.1145/3748273.3749210
ISBN: 9798400720826

Chronos: Prescheduled circuit switching for LLM training

Sundararajan Renganathan
rsundar@stanford.edu
Stanford University
Stanford, CA, USA

Nick McKeown
nickm@stanford.edu
Stanford University
Stanford, CA, USA

Abstract

Hundreds of thousands of accelerators are used to train LLMs, with accelerators connected by packet-switched AI fabrics. In this paper, we ask if the fabric can be built entirely from time-synchronized circuit switches instead. The goal would be to reduce power, increase switching capacity, or reduce the number of network tiers.

It appears to be possible, because traffic patterns are largely known *a priori*. We describe a tool that analyzes the training code and deduces a sequence of permutations that will correctly schedule a crossbar throughout the training run. Expert parallelism (used with mixture-of-experts models) is the only form of parallelism that cannot be pre-scheduled. For MoE traffic, we show how Birkhoff-von Neumann decomposition can be used to schedule the crossbar on demand.

CCS Concepts

• **Networks** → **Network design principles**; • **Hardware** → **Networking hardware**.

Keywords

AI Fabric, Network Architecture, Circuit Switch, Optical Switch, Scheduling Algorithm

ACM Reference Format:
Sundararajan Renganathan and Nick McKeown. 2025. Chronos: Prescheduled circuit switching for LLM training. In *2nd Workshop on Networks for AI Computing (NAIC '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3748273.3749210

1 Introduction

In large LLM training systems, accelerators are interconnected using a *packet-switched* fabric: a scale-up network connects accelerators within a rack (e.g., NVLink [35], TPU ICI [21], AMD Infinity Fabric [49], AWS NeuronLink [46]), and a scale-out network connects racks together (e.g., Infiniband [34], RoCE [14], UEC [52]). Packet switching works great when traffic is unpredictable. The super power of packet switching is its ability to efficiently carry heterogeneous applications with a best-effort service model.

In contrast, LLMs are trained on a purpose-built, carefully tuned system with software honed to carry as much data as possible, as quickly as possible, with minimal latency, in successive rounds of communication. Variable latency is considered bad because a round must wait for straggling data to complete. Training systems therefore rely on an over-provisioned fabric [14] to make it unlikely congestion will take place; an approach that is inefficient in terms of power and cost. And it will still have occasional hotspots that delay training cycles [39]. Essentially, training networks have unpredictable latency and throughput, which seems a bit counter-intuitive given the enormous investment in training systems to complete as fast as possible.

However, LLM training traffic is regular and predictable and is largely determined before the training run starts [24, 40]. The model developer decides when the messages are sent, how large they are, and who they are sent to. It is therefore interesting to ask whether packet switches are the optimal way to carry data in a training system.

In particular, we ask: *Would a circuit switched fabric, built from pre-scheduled crossbar switches, and cognizant of the traffic patterns and sequence of arrivals, allow LLM training systems to complete sooner and/or consume less power?*

Circuit switches usually add circuits one at a time, when a new call starts. We are going to see if we can pre-schedule the traffic for the *entire* training run upfront, using knowledge of the traffic pattern and the topology, and hence predetermine the configuration of the crossbar switches needed to transfer data in successive, fixed-length time slots. We envisage simple switches: no packet buffers, no forwarding tables, and no packet processing. Delays are predictable and minimized. We will assume that the fixed-length time slots time-synchronize the whole system, which is therefore essentially centrally controlled, albeit with fault-tolerant mechanisms to minimize downtime when switches, links, NICs, accelerators, CPUs and memories fail.

LLM training systems employ several parallelisms, such as data parallelism [43], tensor parallelism [48], pipeline parallelism [17, 30], sequence parallelism [22], and context parallelism [27]). Each parallelism requires communication between a specific set of accelerators in a specific order. An important exception is expert parallelism used with Mixture-of-Experts (MoE) [11, 28] where traffic patterns are input-dependent and not known until the routing layer decides which expert the tokens should be sent to. We show how to schedule MoE traffic (and any asynchronous traffic) on the fly using Birkhoff-von Neumann [8, 54] decomposition.

Our long-term goal is to design an AI fabric that maximizes the AI fabric's total capacity (in bits/s) divided by its overall power consumption. We conjecture that a pre-scheduled crossbar-based AI fabric will consume a fraction of the power of today's Ethernet and proprietary NVlink-based systems, and can reduce the network tiers by increasing switch fanout.

In this short paper, we describe a high-level design for a circuit-switched fabric built from simple crossbar switches. We show how

NAIC '25, Coimbra, Portugal

ACM ISBN 979-8-4007-2082-6/2025/09
https://doi.org/10.1145/3748273.3749210

to derive the switch schedules for the entire run, up front, directly from the training code. And we show how unpredictable traffic (in expert parallelism and control traffic) can be scheduled quickly and efficiently. We argue that the system is as at least as reliable as today's packet-switched systems.

Contributions.

1. We propose Chronos, which we believe is the first AI fabric design to use only pre-scheduled crossbar circuit switches, leveraging known *a priori* training traffic patterns.
2. A technique (and tool) to analyze the training source code, generating a sequence of permutations to schedule the crossbar switches for the training run.
3. Chronos handles unpredictable traffic (e.g., mixture-of-experts, management and control messages) by calculating the circuit switch permutations on-the-fly using Birkhoff-von Neumann decomposition.

2 Related Work

While prior work has investigated circuit switching for conventional datacenter traffic [6, 36, 37, 53], to our knowledge, we are the first to do so for the entire AI fabric during LLM training. Our work is largely inspired by Google's "lightwave" network [26] for dynamically creating clusters of TPU nodes prior to AI training, to exploit the pre-determined synchronous traffic patterns used by standard models of parallelism used by collectives. The authors provide an excellent survey of prior proposals for all-optical networks and pre-existing optical circuit switching techniques (Appendix C of [26]), as well as their development and deployment of the Palomar MEMS optical circuit switches, at scale. Our work differs in four ways: (1) Chronos takes circuit switching to the extreme by studying what happens if we replace *every* packet switch with a circuit switch, including at the top-of-rack. (2) Chronos is agnostic to the technology used to build the switches, whether they are optical or electrical; although of course the behavior will differ in terms of switching time, optical transparency and power. (3) Chronos supports unpredictable traffic that is not pre-determined, including the increasingly important MoE parallellism. And to be clear, (4) Chronos is a paper study; there is no prototype or deployment.

Previous authors have noted the predictability of communication patterns during LLM training [24, 40, 41, 55], but to our knowledge no-one has used it to completely pre-schedule the network ahead of time.

3 *A priori* knowledge of traffic patterns

Model developers use *collectives* to implement each type of parallelism (except MoE, which we study in Section 8). Collectives are patterns of communication between pre-determined groups of accelerators, rather than between individual pairs [32]. It helps to think of accelerators as arranged in an n-dimensional array, where n is the number of types of parallelism and each parallelism communicates in one dimension. Collectives, such as *all-reduce*, *reduce-scatter*, and *all-gather*, are used to communicate among accelerators within a parallelism, typically using logical rings.[1] Figure 1 shows a logical ring among four GPUs in the same parallelism group, and the corresponding permutation matrix.

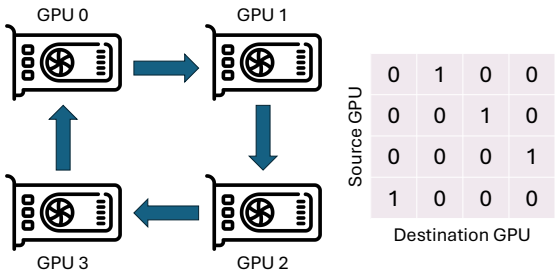

Figure 1: An example logical ring involving four GPUs (left) and the corresponding permutation (right).

In our approach, we determine the set (or "stack") of all permutations needed for a training run. Section 5 describes how we derive the permutations from the training source code. If all the accelerators are connected by one big circuit switch then the stack of permutations forms a schedule used to configure the central switch in successive rounds of communication, with one permutation per time slot. In big systems the fabric is a multistage hierarchy of circuit switches, and so we need to decompose the network-wide permutations into a local set of permutations for each switch. The decomposition is trivial if each tier has the same capacity (i.e. the fabric is not oversubscribed); we simply route all traffic via the top tier and back down again. If we want to switch locally (to reduce power, or because of oversubscription at higher tiers), we can use the method described in Section 6 and can potentially use shorter time slots locally than globally.

4 Circuit switches are simpler

A circuit switch can be built using a crossbar switch in which, during a time slot, each input is connected to at most one output and each output is connected to at most one input, as represented by the permutation matrix in Figure 2.[2]

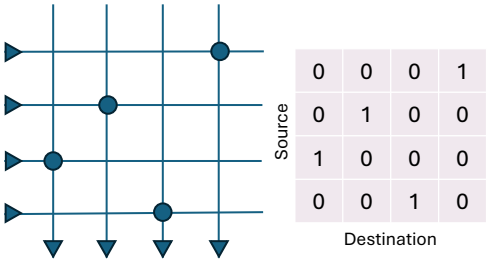

Figure 2: An example 4x4 crossbar switch (left) configured by a permutation matrix (right).

[1] Previous authors have used topology knowledge to accelerate collectives [10, 29, 45, 47, 60]. Our examples assume ring-based collectives but can in principle be extended to other implementations.

[2] In principle, multicast and broadcast can be useful to accelerate collectives, but we don't consider them in this paper.

Circuit switches are much simpler than packet switches because they have less to do. A typical single-chip Ethernet packet switch has 30% of its area dedicated to serial I/O, 50% to packet processing and 20% to packet buffers and the traffic manager [9]. An circuit switch does not need packet processing logic, lookup tables, or packet buffers, because arriving packets are immediately switched to a pre-determined output. This makes possible electronic circuit switches with 70% of the area freed up, which means (a) lower power, or (b) we can repurpose the area to add more I/O capacity (e.g., using area I/O [59]).

Circuit switches have deterministic, low latency; a few nanoseconds for an electrical or optical circuit switch, compared to variable, unpredictable latency in packet switches.

Optical circuit switches bring additional benefits, including optical transparency and hence can carry data independent of wavelength and data rate.

5 Deriving permutations from training code

We wrote a software tool Genstack to derive the switch permutations from the training code. It operates in two steps: First, it replaces collective calls with logging code and instruments how the collectives are called. Second, it combines collective calls into global permutations that capture communications among all accelerators. We describe the two steps in detail below. Our results are based on Megatron-LM [31], a popular and open framework for LLM training, and can be used with other LLM training frameworks [1, 2, 44].

5.1 Logging the collectives

Genstack logs all collective calls (e.g. `all_reduce`, `all_gather`, `broadcast`, etc.) invoked by a single iteration of training, along with their tensor shapes and participating GPUs.

But we cannot log the collective calls on the real full-scale system before we have picked the permutations, and so instead we emulate a single iteration of training running on a smaller system, without GPUs. This is still challenging because: (1) We can not manipulate the full-system high-dimensional model tensors without a large number of physical GPUs, (2) we can not spawn thousands of GPU processes, and (3) training code (such as Megatron-LM) conditionally invokes communication collectives based on pipeline stage, tensor-parallel group, and data-parallel group, requiring end-to-end code execution.

We avoid needing GPUs by using *meta tensors* [38], zero-overhead tensors that store only shape and dtype. Throughout the model construction, forward pass, backward pass, and optimizer step, only the shapes and dtypes are tracked, rather than actual floating-point data. Operations such as matrix multiplication and layer normalization become *no-ops* during runtime; they skip kernel execution but preserve shape propagation for the next layer. Then we monkey-patch `torch.distributed` to spoof the execution on GPUs, and to record the collectives rather than executing them. We use multiprocessing to spawn one CPU process for each GPU involved. This maintains correct control-flow logic without incurring the expense of real data transfers.

With these patches in place, we run exactly one training iteration (forward + backward + optimizer step). Each process logs the communication events it experiences locally:

```
[ {"op": "broadcast", "call_id": 1, "ranks": [0,1], "shape":
      [1024, 4096], "dtype": "float16"},
  {"op": "all_reduce", "call_id": 2, "ranks": [0,1,2,3], "
       shape": [1024, 4096], "dtype": "float16", "reduce_op
       ": "MIN"}, ... ]
```

This trace fully captures the GPUs involved and the tensor dimensions of each collective call in an iteration. The tool is described in more detail, with code examples, in Appendix A.

5.2 Deriving permutations from the logs

To derive the stack of global permutations we combine the permutations for each collective call and for each parallelism.

It helps to study a small example. Consider 16 GPUs arranged in a logical 4×4 array, and using data parallelism (x-axis communication) and tensor parallelism (y-axis communication). We need to produce a stack of 16×16 global permutation matrices to schedule the 16×16 crossbar switch.

We will start with data parallelism and consider the rows of the 4×4 array. The rows operate identically and in parallel, calling the same collective operations at the same time, and transferring the same amount of data. Specifically, they make n collective calls $(c_1, c_2, \ldots, c_n)$ causing data transfers represented by permutations $p_1, p_2, \ldots, p_n$. This allows us to generate global permutations $P_1, P_2, \ldots, P_n$ for the n collective calls across rows where P_j is generated by combining the 4×4 permutations for each row. We repeat the process along each column for tensor parallelism.[3]

If pipeline parallelism is used, there will be a different stack of global permutations per pipeline stage in the forward and backward pass. Each permutation is incomplete because the GPUs in each pipeline stage only communicate with GPUs in the previous/next stage (for pipeline parallelism) or with other GPUs in the same stage (for all the other parallelisms).

Once we have the global permutations, we generate the switch-local permutations.

6 Time Slots

The duration of a time slot, T, is determined by four attributes, (a) the smallest number of bytes, b, transferred by a collective communication call, (b) the link speed, c, (c) the maximum one-way latency between endpoints, t_{max}, and (d) the 'dead' time (*aka* the 'guard' time) while a crossbar changes permutation, t_x, < 10ns for an electronic switch or an optical switch with tunable lasers, and about 1ms for a MEMs-based optical switch. The time slot duration should be chosed so that $T \geq \frac{b}{c} + t_{max} + t_x$.

The fabric sends a global synchronization signal to announce the start of a new time slot (e.g., from a spine switch, with a leader selection and failover mechanism). This is straightforward in an electronic switch; optical circuit switches need a way to *broadcast* a time sync signal to all end nodes simultaneously. For example, we can generate the time signal electrically, then convert it to an optical signal. The signal is broadcast to the end points using a 1-to-N passive coupler at the switch to couple the time slot signal into the optical links connected to the end nodes.

[3]The number of global permutations is different for each parallelism.

The time slot signal is only used to announce a new time slot; it is independent of the local clock driving the sequential logic in the accelerators and NICs (at, say, 1GHz). In practice, for electrical and optical switches, the range of time slot durations $1\mu s < T < 10ms$ and therefore corresponds to $10^3 - 10^6$ clock cycles of the GPU.

Accelerators will be at different distances from the spine switch and will "hear" about the start of the new time slot at different times. If the variation in latencies is small compared to T, this does not matter. The most efficient systems will measure the round-trip time to every device so that its data arrives at the correct time to the spine switch. The system designer can decide whether or not to implement such a mechanism based on the switching efficiency, $\eta = \frac{b/c}{T} \leq \frac{b/c}{b/c+t_{max}+t_x}$, which is the fraction of time useful data is transferred.

GPUs	Tensor	Context	Pipeline	Data
8,192	8	1	16	64
16,384	8	1	16	128
16,384	8	16	16	8

Table 1: Three different configurations of GPUs used to train the 405B parameter Llama 3 model, using four types of parallelism. The entry in the table indicates the degree of each type of parallelism.

Example T for Llama 3. The largest Llama 3 [15] model has 405B parameters and is trained using tensor, context, pipeline and data parallelism, in the three configurations in Table 1. Data parallelism generally transfers the smallest units of data and dictates the slot time. The amount of data is determined by, (a) the number of parameters in each layer (because of compute-communication overlapping [33], the communications kick off after the backward pass of a bucket finishes, usually for a single layer for the largest models), (b) the degree of tensor parallelism (because it shards the parameters within a layer) and (c) the degree of data parallelism as ring-based collectives divide the transmitted tensor into chunks. If we denote the parameters per layer p_l, then $b = \frac{4p_l}{TP.DP}$, where TP is the degree of tensor parallelism (first column in the table) and DP is the degree of data parallelism. The factor of four is because the gradients are in `fp32` format. With $2.8B$ parameters per layer, and a link speed of 800Gb/s, the lowest $\frac{b}{c}$ is 109μs across the three configurations. If t_{max} is 10μs (2km links) and t_x =10ns, then we lose 8.4% throughput.

More examples. A circuit switch that connects accelerators over 200m links running at 400Gb/s, and b = 1Mbyte, should use a time slot $T \geq 22\mu + t_x$. For an electronic switch or an optical switch built with fast tunable lasers ($t_x = 10ns$) the efficiency exceeds 95%. If we replace the switch with an optical MEMS switch with $t_x = 1ms$, we need $b >$ 150Mbytes for the efficiency to exceed 75%.

7 Failures and Stragglers

It takes weeks to train a large model, and hardware failures are inevitable. Periodic checkpoints help, but checkpointing the training state is expensive and takes time. If the operator checkpoints too often then training takes too long to complete; not often enough and training has to be frequently repeated.

Example. Meta reported 419 unexpected interruptions during a 54-day training run of Llama 3 with an MTBF (mean time between failure) of about three hours [15]. 58.7% of failures were GPUs (HBM3 issues, SRAM issues, and faulty GPUs), 8.4% were switches and cables, and 1.7% were NICs.

Before adopting this extreme approach (i.e. using an AI fabric built entirely from circuit switches), an operator would need to understand the consequences of hard failures (e.g. components failing, links breaking) and soft failures (e.g. straggler GPUs that respond slowly or inconsistently, links that are flapping on and off, or causing many bit errors).

Answering these questions fully is beyond the reach of our initial paper study, and so we leave the comprehensive analysis of failures for further work. However, there is some reason to think the MTBF of individual components will be about the same: The system will have the same number of GPUs, CPUs, memories, and NICs as before. In principle, a circuit switch is likely more reliable than a packet switch because it is simpler, particularly if it is electronic. However, given that the system component count is dominated by other components, this is unlikely to affect the overall system MTBF.

If the AI fabric fails, or if GPUs miss their time slot, the recovery process will be different for a circuit switch than for a packet switch. Generally, packet switches will be more forgiving of timing errors; on the other hand, it is easier to determine an error in a circuit switch when data is expected to arrive at a pre-determined time. A full system design needs to consider the consequences of different types of error, and map in appropriate standby components as needed.

In future work, we plan to study if and how the whole system can be stopped and started synchronously, by controlling the flow of synchronization messages. If possible, this might allow the system to be frozen, interrogated, and possibly healed without losing system state.

The switch fabric also needs to handle variations in GPU processing time - for example, when a GPU takes longer to finish its calculation than expected. A simple solution is to delay the next time slot until all data has arrived. Delays would reduce efficiency and increase the overall training time, and so the system controller will need to decide how often this can be allowed, and by how much, before intervention or replacement is required.

Failure tolerance. There have been several proposals to handle failures during training [4, 13, 20, 51, 58]. We argue that the failure tolerance of our design is no worse than packet switching, and any additional traffic during failures can be scheduled using the BvN decomposition.

Stragglers. Stragglers in training have been investigated by prior work [23, 25, 56, 57]. While it is easier to identify stragglers in our design (due to their non-adherence to the time slots) than the status quo, a definitive answer to the straggler tolerance of our design requires more study.

8 Handling unpredictable traffic

The programmer does not always know about a particular communication in advance. For example, control and management messages,

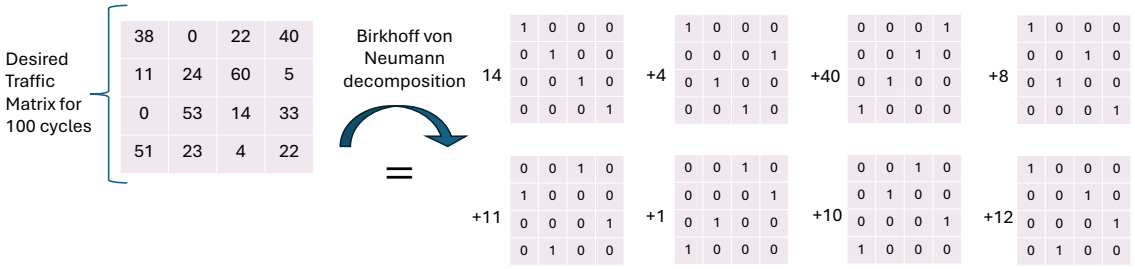

Figure 3: The Birkhoff-von Neumann decomposition of a traffic demand matrix into permutations. The weight corresponding to each permutation indicates how many times the permutation is held.

error messages, interrupts and reloading data after a checkpoint. A big and important example is when the system uses a mixture-of-experts (MoE) model. The routing of data to experts is implemented using the *all-to-all* collective and its input-dependence makes it hard to predict or bound the patterns and volume of communication.

Mixture-of-experts splits the transformer's feedforward network (FFN) into multiple smaller FFNs (from 8-128 *experts*) and in *expert parallelism* the experts are placed on different accelerators. A routing layer computes the data (i.e., tokens) to send to each expert, which tells us the traffic demand matrix between the GPUs.

Whenever the system needs to carry unpredictable traffic, we need to configure the crossbar switches on demand. We can not use packet switching, because the switches contain no buffers or forwarding logic. Instead, we can calculate new permutations on the fly to carry the traffic pent up in the network interfaces' VOQ.

Luckily, there is a clever way to turn a traffic matrix into a sequence of permutations called a Birkhoff-von Neumann (BvN) decomposition [8, 54]. Figure 3 shows a simple example. Decomposing an $N \times N$ matrix produces $O(N^2)$ unique permutations; each permutation is found using a bipartite matching algorithm. Each permutation is, in turn, subtracted from the traffic matrix to create a residual matrix. The permutation is held for consecutive time slots equal to the minimum entry along the permutation.

If we run a maximum cardinality matching algorithm at every step (e.g., the Hopcroft-Karp algorithm [16]), the overall time complexity of a BvN decomposition is $O(N^{4.5})$. We can instead use a much faster *maximal* (greedy) algorithm [50], which is close to optimal (maximum size). The times to compute the BvN decompositions on an Apple M1 Max CPU are reported in Table 2. Performance can be improved with specialized hardware, in particular using the Wrapped Wave Front Arbiter (WWFA) algorithm [50], which is known to be the most hardware-efficient maximal matching algorithm.

We invented a novel hardware algorithm (*BhaVaN*) that is the fastest known BvN decomposition. BhaVaN is implemented in Verilog and uses the WWFA and completes a BvN decomposition in less than 1μs for a 64-port switch on a 16nm ASIC process. The design takes up less than $1mm^2$ and hence would consume less than 1% of an electronic circuit switch ASIC. If BhaVaN is used with an optical circuit switch, the algorithm can run on a modified NIC chip. BhaVaN will be described in more detail in a separate paper.

Time slots. Unpredictable traffic can use a number of successive time slots dictated by the permutation weights in the BvN decomposition. For example, if the weight is two, then the permutation is held steady for two time slots.

	N=16	N=32	N=64	N=128	N=256
Maximum	0.5ms	3ms	25ms	291ms	4.45s
Maximal	0.2ms	0.8ms	4.6ms	52ms	0.65s

Table 2: Time to compute BvN decomposition in single-threaded software for $N \times N$ matrices, different N.

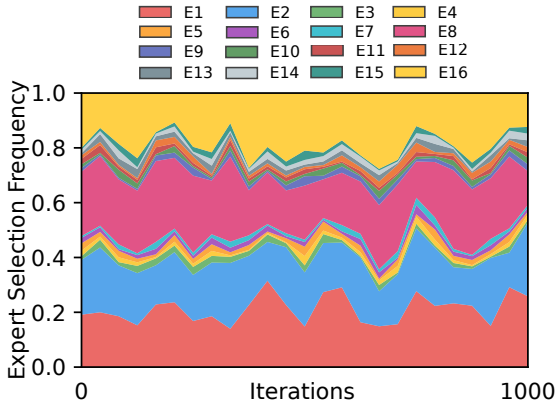

Figure 4: The fraction of tokens received by each expert across successive training iterations for a GPT-MoE model. E1 refers to expert 1. Figure taken from [12].

8.1 Performance comparison

We compare the completion time of MoE traffic for our circuit-switched approach against conventional packet switching. We do this by generating 1,000 different MoE traffic matrices and simulate how long the packet-switched network takes to transfer data, and how long the BvN decomposition takes to run for circuit switching, plus the transfer time.

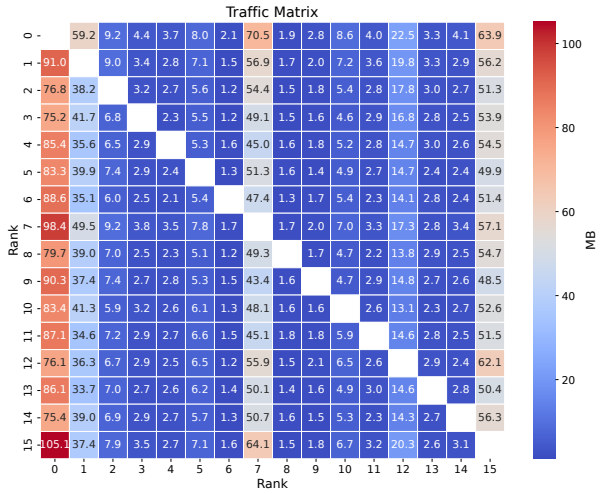

Figure 5: Example generated traffic matrix.

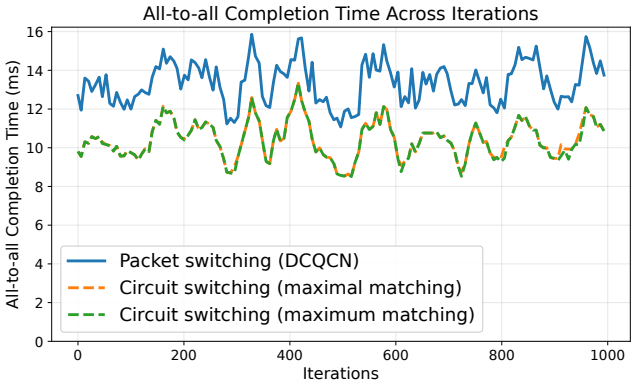

Figure 6: Comparison of the all-to-all completion times for 16 experts, for 1,000 iterations.

Generating MoE traffic matrices. We generate sample MoE traffic matrices using the data measured by 1,000 successive iterations of training of a 16-expert GPT-MOE system [12]. The matrices are quite non-uniform, with 4 out of 16 experts receiving up to 80% of the tokens (Figure 4). To generate a 16×16 traffic matrix, we use the fraction of tokens received by each expert and randomly spread them across the source GPUs from which the tokens arrive.[4] We repeat this for every iteration in the training data, to generate 1,000 traffic matrices.[5] We use a hidden size of 16,384 and a sequence length of 8,192 as in Llama 3 405B [15]. Figure 5 shows an example traffic matrix and the amount of data transferred (in megabytes).

Topology and performance metric. The GPUs that host the experts are typically connected to the same switch. Our simulations assume a link speed of 800 Gb/s and an RTT of 1 μs. Our performance metric is the all-to-all completion time.

Simulations. The packet switch is simulated using ns-3 from the HPCC ns-3 repo [3], with DCQCN [5, 62] as the congestion control. The circuit switch completion time is calculated by adding the permutation weights of the BvN decomposition and dividing by the link speed. For a fair comparison, we tune DCQCN parameters (K_{min}, K_{max}, P_{max}, R_{AI}, R_{HAI} and g) using a hyperparameter optimization framework [7, 19].

Figure 6 shows the time to transfer all data to 16 experts, over 1,000 runs, each with a different traffic matrix. The average completion time for packet switching is approximately 13ms, with clear peaks during times of congestion. The circuit-switched network finishes 22.04% faster on average (max 32% faster) because there are no packet buffers and data is transferred in an orderly fashion.[6]

[4]If multiple experts are placed on a single GPU (e.g., 4 experts per GPU [18, 42]), the traffic matrix is smaller and easier to decompose compared to when all the experts are placed on different GPUs.
[5]While not strictly independent samples (they are from the same training run), we will assume they are, and we use them for 1,000 Monte Carlo simulation runs.
[6]The performance improvement would be different with a different congestion control algorithm, which will be the subject of further work.

Furthermore, once the relatively short flows to a given destination depart, the packet switching control loop is slow to grab the available bandwidth for the remaining large flows due to uplink contention at the senders.

Note that the time to decompose the traffic matrix (< 0.5ms for 16 GPUs) is small compared to the transfer time, and very efficient in this case, with maximal or maximum matching. If MoE starts to use 256 or more GPUs in the future, we will need to implement it in hardware, as described above.

9 Discussion

We set out to understand whether it is feasible to build a complete training system using a circuit-switched fabric, and we started with three open questions. First, is the majority of traffic predictable? We conclude that it is and can be deduced from the training code. Second, can unpredictable traffic be scheduled on the fly fast enough? We conclude that it can be, and we invented a very fast parallel algorithm that can schedule it in less than 1μs. Third, how does our design handle failures and stragglers? We hypothesize that it is no worse than packet switching, but this needs further study. A big unresolved question is understanding the power, cost, and area benefits of our design compared to packet switching. This is the topic of further work as it requires prototyping, and we will embark on this next.

Our proposal focuses on large single-job training runs, but it can be applied to multi-tenant training clusters by leveraging the insights in prior work [40, 61], where training jobs are interleaved so that their communications do not interfere.

Acknowledgements

This work was funded by the UK Advanced Research and Invention Agency (ARIA) and by the VMware/Broadcom University Research Fund. We thank Noa Zilberman for her suggestions.

References

[1] Deepspeed AI. Fetched May 6th, 2025. DeepSpeed repository. https://github.com/deepspeedai/DeepSpeed.
[2] Lightning AI. Fetched May 6th, 2025. PyTorch Lightning repository. https://github.com/Lightning-AI/pytorch-lightning.

[3] Alibaba. Fetched May 6th, 2025. High-Precision-Congestion-Control repository. https://github.com/alibaba-edu/High-Precision-Congestion-Control.

[4] Daiyaan Arfeen, Dheevatsa Mudigere, Ankit More, Bhargava Gopireddy, Ahmet Inci, and Gregory R. Ganger. 2025. Nonuniform-Tensor-Parallelism: Mitigating GPU failure impact for Scaled-up LLM Training. arXiv:2504.06095 [cs.DC] https://arxiv.org/abs/2504.06095

[5] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. 2023. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 49–67. https://www.usenix.org/conference/nsdi23/presentation/bai

[6] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. 2020. 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 782–797. doi:10.1145/3387514.3406221

[7] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems* 24 (2011).

[8] Garrett Birkhoff. 1946. Tres observaciones sobre el algebra lineal. *Univ. Nac. Tucuman, Ser. A* 5 (1946), 147–154.

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[10] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 62–75. doi:10.1145/3437801.3441620

[11] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. arXiv:2101.03961 [cs.LG] https://arxiv.org/abs/2101.03961

[12] Swapnil Gandhi and Christos Kozyrakis. 2024. MoEtion: Efficient and Reliable Checkpointing for Mixture-of-Experts Models at Scale. arXiv:2412.15411v1 [cs.DC] https://arxiv.org/abs/2412.15411v1

[13] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 211–228. doi:10.1145/3694715.3695960

[14] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 57–70. doi:10.1145/3651890.3672233

[15] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[16] John E Hopcroft and Richard M Karp. 1973. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing* 2, 4 (1973), 225–231.

[17] 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. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. arXiv:1811.06965 [cs.CV] https://arxiv.org/abs/1811.06965

[18] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. 2023. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems* 5 (2023), 269–287.

[19] Hyperopt. Fetched May 21st, 2025. Hyperopt: Distributed Hyperparameter Optimization. https://github.com/hyperopt/hyperopt.

[20] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 382–395. doi:10.1145/3600006.3613152

[21] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. doi:10.1145/3579371.3589350

[22] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.

[23] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Yujie Wang, Hailin Zhang, Xiaonan Nie, and Bin Cui. 2025. Malleus: Straggler-Resilient Hybrid Parallel Training of Large-scale Models via Malleable Data and Model Parallelization. arXiv:2410.13333 [cs.DC] https://arxiv.org/abs/2410.13333

[24] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. 2024. Understanding Communication Characteristics of Distributed Training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking* (Sydney, Australia) *(APNet '24)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3663408.3663409

[25] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, and Jinyang Li. 2025. Understanding Stragglers in Large Model Training Using What-if Analysis. arXiv:2505.05713 [cs.DC] https://arxiv.org/abs/2505.05713

[26] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, Erji Mao, Daniel Nelson, George Papen, Mukarram Tariq, and Amin Vahdat. 2023. Lightwave Fabrics: At-Scale Optical Circuit Switching for Datacenter and Machine Learning Systems. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 499–515. doi:10.1145/3603269.3604836

[27] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. arXiv:2310.01889 [cs.CL] https://arxiv.org/abs/2310.01889

[28] Meta. Fetched May 21st, 2025. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

[29] Microsoft. Fetched May 21st, 2025. Microsoft Collective Communication Library. https://github.com/microsoft/msccl.

[30] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. arXiv:2104.04473 [cs.CL] https://arxiv.org/abs/2104.04473

[31] NVIDIA. Fetched April 16th, 2025. Megatron-LM repository. https://github.com/NVIDIA/Megatron-LM.

[32] NVIDIA. Fetched March 31st, 2025. Collective operations. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html.

[33] NVIDIA. Fetched March 31st, 2025. NVIDIA NeMo Framework User Guide: Communication Overlap. https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/optimizations/communication_overlap.html.

[34] NVIDIA. Fetched May 6th, 2025. NVIDIA Quantum InfiniBand Switches. https://www.nvidia.com/en-us/networking/infiniband-switching/.

[35] NVIDIA. Fetched May 6th, 2025. NVLink and NVLink Switch. https://www.nvidia.com/en-us/data-center/nvlink/.

[36] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2013. Integrating microsecond circuit switching into the data center. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 447–458. doi:10.1145/2534169.2486007

[37] 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. 2022. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 66–85. doi:10.1145/3544216.3544265

[38] PyTorch. Fetched April 16th, 2025. Meta device. https://pytorch.org/docs/stable/meta.html.

[39] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. 2024. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 691–706. doi:10.1145/3651890.3672265

[40] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. 2024. {CASSINI}:{Network-Aware} Job Scheduling in Machine Learning Clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1403–1420.

[41] Sudarsanan Rajasekaran, Sanjoli Narang, Anton A. Zabreyko, and Manya Ghobadi. 2024. MLTCP: A Distributed Technique to Approximate Centralized Flow Scheduling For Machine Learning. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks* (Irvine, CA, USA) *(HotNets '24)*. Association for Computing Machinery, New York, NY, USA, 167–176. doi:10.1145/3696348.3696878

[42] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*. PMLR, 18332–18346.

[43] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. arXiv:1910.02054 [cs.LG] https://arxiv.org/abs/1910.02054

[44] Databricks Mosaic Research. Fetched May 6th, 2025. Composer repository. https://github.com/mosaicml/composer.

[45] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. 2024. Swing: Short-cutting Rings for Higher Bandwidth Allreduce. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1445–1462. https://www.usenix.org/conference/nsdi24/presentation/de-sensi

[46] Amazon Web Services. Fetched May 6th, 2025. AWS Trainium. https://aws.amazon.com/ai/machine-learning/trainium/.

[47] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 593–612. https://www.usenix.org/conference/nsdi23/presentation/shah

[48] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[49] Alan Smith and Vamsi Krishna Alla. 2025. AMD Instinct™ MI300X: A Generative AI Accelerator and Platform Architecture. *IEEE Micro* (2025), 1–9. doi:10.1109/MM.2025.3552324

[50] Yuval Tamir and H-C Chi. 1993. Symmetric crossbar arbiters for VLSI communication switches. *IEEE Transactions on Parallel and Distributed Systems* 4, 1 (1993), 13–27.

[51] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 497–513. https://www.usenix.org/conference/nsdi23/presentation/thorpe

[52] UEC. Fetched May 6th, 2025. Ultra Ethernet Consortium. https://ultraethernet.org/.

[53] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. 2012. Practical TDMA for datacenter ethernet. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 225–238. doi:10.1145/2168836.2168859

[54] John Von Neumann. 1953. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games* 2, 0 (1953), 5–12.

[55] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. 2024. Towards Domain-Specific Network Transport for Distributed DNN Training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1421–1443. https://www.usenix.org/conference/nsdi24/presentation/wang-hao

[56] Ertza Warraich, Omer Shabtai, Khalid Manaa, Shay Vargaftik, Yonatan Piasetzky, Matty Kadosh, Lalith Suresh, and Muhammad Shahbaz. 2025. OptiReduce: Resilient and Tail-Optimal AllReduce for Distributed Deep Learning in the Cloud. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 685–703. https://www.usenix.org/conference/nsdi25/presentation/warraich

[57] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. 2024. FALCON: Pinpointing and Mitigating Stragglers for Large-Scale Hybrid-Parallel Training. arXiv:2410.12588 [cs.DC] https://arxiv.org/abs/2410.12588

[58] Yongji Wu, Wenjie Qu, Tianyang Tao, Zhuang Wang, Wei Bai, Zhuohao Li, Yuan Tian, Jiaheng Zhang, Matthew Lentz, and Danyang Zhuo. 2024. Lazarus: Resilient and Elastic Training of Mixture-of-Experts Models with Adaptive Expert Placement. arXiv:2407.04656 [cs.DC] https://arxiv.org/abs/2407.04656

[59] Jinjun Xiong, Yiu-Chung Wong, Egino Sarto, and Lei He. 2006. Constraint driven I/O planning and placement for chip-package co-design. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*. 207–212.

[60] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. 2025. Efficient Direct-Connect Topologies for Collective Communications. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 705–737. https://www.usenix.org/conference/nsdi25/presentation/zhao-liangyu

[61] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 428–440. doi:10.1145/3544216.3544224

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

A Genstack: A software tool to log collectives and generate the global stack of permutations

Meta-Tensor Execution: Since PyTorch 1.9, users can instantiate *meta tensors* [38] via:

```
x = torch.empty(16, 1024, device='meta')
```

The tensor has no backing memory, but still participates in shape inference. We intercept all tensor creation routines (e.g., `torch.zeros`, `torch.empty`, `torch.ones`, or model parameter initializations) so that they produce meta tensors by default:

```
_orig_empty = torch.empty

def meta_empty(*shape, **kwargs):
    return _orig_empty(*shape, device='meta', **kwargs)

torch.empty = meta_empty
# Similarly for zeros, ones, etc.
```

Spoofing Distributed Processes: We emulate the full training system on a small CPU-based system in two steps. First, we spawn N Python processes (each simulating one rank). Then, each process assigns ranks in the range $[0, N-1]$ by calling

```
torch.distributed.init_process_group(backend='gloo',
    world_size=N, rank=LOCAL_RANK, store=...)
```

Because we only use meta tensors, no actual GPU memory is allocated. PyTorch's distributed initialization runs normally, but subsequent collective calls are intercepted.

Intercepting collective calls: All collective calls are *monkey-patched*, for example:

- `torch.distributed.all_reduce`
- `torch.distributed.all_gather`
- `torch.distributed.broadcast`
- `torch.distributed.reduce_scatter`
- `torch.distributed.send/recv`
- `torch.distributed.barrier`

We store the original PyTorch functions, but replace them with wrapped versions to log the call by gathering the call type (e.g., `all_reduce`), tensor shape, dtype, operation type (e.g., `SUM`), and rank inferred from the `ProcessGroup` handle. The calls return immediately with a no-op result, for example:

```
orig_allreduce = torch.distributed.all_reduce

def wrapped_allreduce(tensor, op, group=None, async_op=False
    ):
    # 1. Identify ranks in 'group' from a stored lookup
        table
    ranks = group_to_ranks[group]
    # 2. Record shape, dtype, etc.
    # reduce_op is only logged for all_reduce and
        reduce_scatter
    log_event({"op": "all_reduce", "call_id":
        get_next_call_id(), "ranks": ranks, "shape": list(
        tensor.shape), "dtype": str(tensor.dtype), "
        reduce_op": "SUM"})
    # 3. Return a no-op; skip real comm
    return
```

The wrappers capture exactly which collective calls would be issued during forward/backward passes and the optimizer step.

Process Groups and Rank Membership. Megatron-LM often creates logical subgroups (data-parallel, tensor-parallel, pipeline-parallel) via:

```
data_parallel_group = torch.distributed.new_group(ranks=
    dp_ranks)
```

We intercept `new_group` to record which ranks are in a group. When a collective references a group, our wrapper retrieves the associated set of ranks.