ZORSE APPENDIX: OPTIMIZING LLM TRAINING EFFICIENCY ON HETEROGENEOUS GPU CLUSTERS

1 P2P COMMUNICATION IN ZORSE

1.1 Communication Plan

Algorithm 1 presents the pseudocode for the greedy algorithm used to compute an efficient communication plan between adjacent pipeline stages in a heterogeneous cluster. Note that each stage has $microbatchesPerRank$ defined based on the compute ratios of the GPUs. We compute this plan statically before execution begins based on compute latencies profiled from each GPU.

The algorithm takes as input the current pipeline stage configuration, the next pipeline stage configuration, and a mapping of GPU types to their compute latencies. It produces a communication plan that specifies:

1. For each microbatch in the current stage: the rank in the next stage to send outputs to
2. For each microbatch in the next stage: the rank in the current stage to receive inputs from
3. Unique tags for all forward and backward communications

1.2 NCCL Communication Challenges

We encountered challenges when using NCCL (NVIDIA Collective Communications Library (NVIDIA, 2024)) for pipeline parallel peer-to-peer communication. NCCL's communication primitives operate under strict ordering requirements that can lead to deadlocks in complex pipeline schedules.

The core issue stems from NCCL's blocking behavior in P2P communication (NVIDIA Corporation, 2020): any `ncclSend` operation blocks until the corresponding `ncclRecv` call is executed on the receiving rank, and similarly, any `ncclRecv` operation blocks waiting for the corresponding `ncclSend`. This will not block computation which can occur on a separate stream, but will block subsequent send and recv operations on the same NCCL communicator. This creates a strict global ordering requirement where communication operations must follow a consistent sequence across all GPU ranks.

Specifically, when implementing pipeline parallelism with multiple microbatches and interleaved pipeline stages of different sizes, there is significant potential for cyclic dependencies.

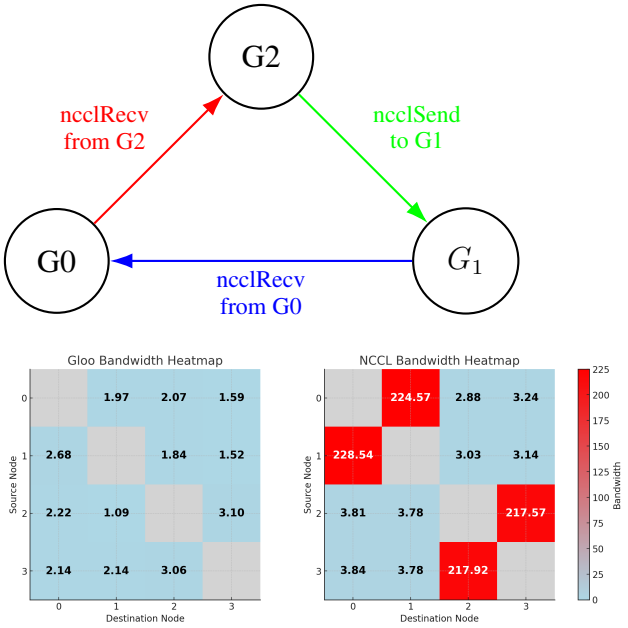

Figure 1. Bandwidth comparison heatmaps between Gloo (left) and NCCL (right) communication backends for 4 GPUs distributed across 2 machines. GPUs 0 and 1 are NVIDIA H100 GPUs within the same machine, while GPUs 2 and 3 are NVIDIA A100 GPUs within the same machine. The color intensity represents bandwidth in GB/s, with NCCL showing significantly higher bandwidth for intra-node communication between similar GPU types.

Due to differences in computation workload and stage execution timing, we could encounter a deadlock scenario like this:

GPU 0	GPU 1	GPU 2
`ncclRecv`(G_2)	`ncclRecv`(G_0)	`ncclSend`(G_1)
`ncclSend`(G_1)	`ncclRecv`(G_2)	`ncclSend`(G_0)

Here's why this creates a deadlock:

1. G_0 is waiting for data from G_2 before it can proceed to sending data to G_1
2. G_1 is waiting for data from G_0 before it can proceed to receiving data from G_2

Algorithm 1 Greedy Microbatch Distribution

Input: Current stage S_c, Next stage S_n, GPU compute time mapping L.
Output: Communication plan between stages
$C \leftarrow \emptyset$ {List to store microbatch completion times}
for each rank $r \in S_c$ **do**
  for each microbatch m from 0 to numMBs$(r) - 1$ **do**
    Append $(m \times L_r, r)$ to C {When this microbatch completes}
  end for
end for
$W \leftarrow \emptyset$ {List to store remaining work per GPU}
for each rank $r \in S_n$ **do**
  for each microbatch m from 0 to numMBs$(r) - 1$ **do**
    Append $((M_r - m) \times L_r, r)$ to W {Remaining work after m microbatches}
  end for
end for
Sort C by completion time (earliest first)
Sort W by remaining work (most work first)
Initialize microbatch indices for all ranks to 0
Initialize empty communication plan P
for each pair $(r_c, r_n) \in$ zip(C, W) **do**
  $i_c \leftarrow$ current index for rank r_c {Match by sorted order}
  $i_n \leftarrow$ current index for rank r_n
  $P[r_c][i_c]$.forward_send_to $\leftarrow r_n$ {Set forward path}
  $P[r_n][i_n]$.forward_recv_from $\leftarrow r_c$
  $P[r_c][i_c]$.backward_recv_from $\leftarrow r_n$ {Set backward path}
  $P[r_n][i_n]$.backward_send_to $\leftarrow r_c$
  Increment microbatch index for r_c and r_n
end for
return P

3. G_2 needs to send data to both G_1 and G_0, but its communication operations execute in order

The key insight is that the relative ordering of `ncclSend` and `ncclRecv` operations across different GPUs creates an implicit dependency graph. If this graph contains a cycle, a deadlock will occur. In pipeline parallelism with multiple microbatches, ensuring a globally deadlock-free ordering becomes extremely challenging as the number of stages and microbatches increases.

While NCCL provides `ncclGroupStart`/`ncclGroupEnd` primitives to group operations, these mechanisms are inadequate for complex pipeline parallelism scenarios with multiple microbatches. The key limitation is that operations within a group are batched and finish together, which prevents effective pipelining when different microbatches need to be processed in parallel. We leave the resolution of these communication ordering challenges in NCCL to future work.

1.3 NCCL Deadlock Example During Training

A common scenario where this NCCL deadlock can occur is when stages have different numbers of GPUs. For example, suppose there are 3 stages, with one GPU in the first stage (1a), two in the second stage (2a, 2b), and one in the third stage (3a). Then the following deadlock can occur:

1. GPU 1a sends an activation to GPU 2a.
2. GPU 1a is blocked as GPU 2a is sending an activation to GPU 3a.
3. GPU 2a is blocked as GPU 3a is receiving an activation from GPU 2b.
4. GPU 3a is blocked as GPU 2b is receiving an activation from GPU 1a.
5. A cycle forms since we are back at GPU 1a.

1.4 Gloo-Based Solution

To address these challenges, we implemented an alternative communication strategy using the Gloo communication library (Facebook Incubator, 2025). Gloo offers several advantages for pipeline communication:

- **Tag-based messaging**: Gloo's communication primitives support message tagging, allowing sends and receives to be matched based on tags rather than strict ordering
- **Non-blocking behavior**: Gloo's operations can be executed out of order without causing deadlocks
- **Flexibility**: This approach enables more dynamic communication patterns essential for efficient pipeline parallelism

The primary tradeoff with Gloo is that it communicates over CPU memory, meaning it cannot leverage high-bandwidth GPU interconnects like NVLink. This can lead to larger communication overheads when a lot of data is transferred. As the heatmaps in Figure 1 illustrate, while NCCL provides superior bandwidth for intra-node transfers, the gap narrows for inter-node communication where network bandwidth becomes the bottleneck rather than the communication backend. We limited our use of Gloo to PP communication, which is significantly less communication intensive than DP

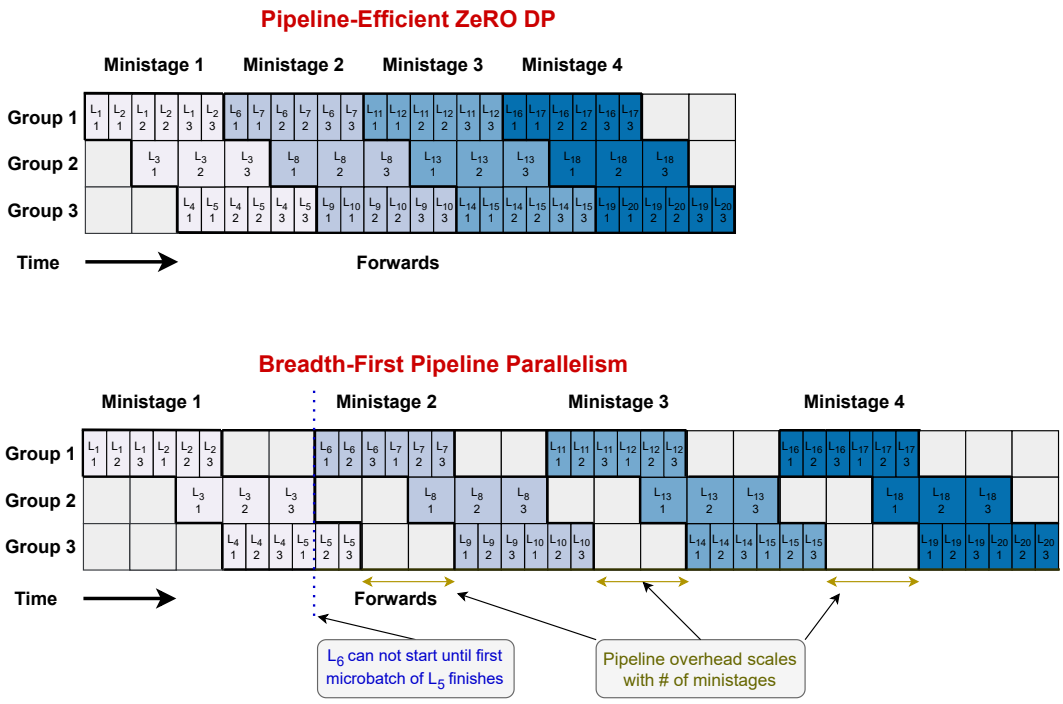

Figure 2. This diagram contrasts the forwards pass in Zorse compared to breadth-first pipeline parallelism. The model has 20 layers (L_i) and there are 3 microbatches. There are 3 GPU groups, each with 4 ministages (Same as Figure 4 of the paper). In breadth-first pipeline parallelism, communication cannot be effectively overlapped with computation efficiently since the subsequent stage cannot start until the current stage reaches the first microbatch of the last layer.

communication, and can still work efficiently over slower networking. Note that our PP communication is overlapped with computation, which is typically large enough to overlap GLOO P2P communication. Moreover, we still use NCCL for communication within GPU groups where we know there can be no deadlocks.

1.4.1 Hybrid Communication Strategy

To maximize performance, we implemented a hybrid communication strategy that selectively uses NCCL and Gloo based on the communication context:

- **Gloo for cross-stage pipeline communication**: We use Gloo for communication between pipeline stages of different sizes where deadlocks can arise from many-to-many communication schedules.
- **NCCL for data parallelism**: We use NCCL for data-parallel communication for maximum bandwidth, since it does not conflict with P2P communication across DP groups.
- **NCCL between same-sized pipeline stages**: For pipeline stages of equal size and containing identical sets of GPUs, we can establish a predictable communication one-to-one pattern that safely uses NCCL. This constraint ensures we avoid many-to-many communication schedules that could lead to deadlocks.

2 Pipeline-Efficient ZeRO DP vs Breadth-First PP

In this section we compare Pipeline-Efficient ZeRO DP and Breadth-First PP. While Zorse is similar to breadth-first PP in that multiple ministages are assigned to each GPU, Zorse is fundamentally different in that it processes all layers per-microbatch before moving to the next microbatch, whereas breadth-first PP processes all microbatches per layer before moving to the next layer. Breadth-first PP introduces a large pipelining overhead when ministages have more than 1 layer, which is necessary in heterogeneous clusters to balance computation. In Figure 2, we illustrate with an example how Zorse's pipeline compares with a breadth-first strategy.

3 Evaluation Details

In this section we provide additional details on the models and training configurations used by Zorse in the evaluation.

Table 1. Model Statistics

Model	Layers	Embd. Size	Attn. Heads	Parameters
Llama 1B (Zhang et al., 2024a)	22	2048	32	1B
Llama 3B (Touvron et al., 2023)	26	3200	32	3B
Llama 7B (Touvron et al., 2023)	32	4096	32	7B
Llama 13B (Touvron et al., 2023)	40	5120	40	13B
Llama 33B (Touvron et al., 2023)	60	6656	52	33B
Llama 65B (Touvron et al., 2023)	80	8192	64	65B
GPT 6.7B (Brown et al., 2020)	32	4096	32	6.7B
GPT 13B (Brown et al., 2020)	40	5120	40	13B

3.1 Models

We use the larger LLama 7B - 65B (Touvron et al., 2023) models for our evaluation in Section 5, and smaller variants (Zhang et al., 2024a) for benchmarking PP + DP in the introduction. We also report additional experimental results for GPT (Brown et al., 2020) models in this section. Model specifications are provided in Table 1.

3.2 Training Configurations

In Table 2, we provide more details on the training configurations that Zorse uses for experiments in the main paper.

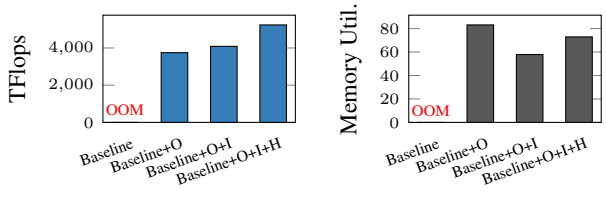

Figure 3. TFlops and memory utilization of Zorse optimizations (Cluster A, LLama 65B). Baseline runs out of memory.

3.3 Ablation Study

We conduct an ablation study to evaluate the impact of Zorse's key components on training throughput and memory utilization. Starting with a baseline PP+ZeRO-2 implementation in TorchTitan, we incrementally add: (1) activation offloading (O), (2) Pipeline-Efficient ZeRO DP and interleaved optimizer updates (I), and (3) Heterogeneous Pipeline Parallelism (H). We evaluate these components using Llama 65B on Cluster A, with results shown in Figure 3.

The PP+ZeRO-2 baseline cannot train the model due to excessive activation memory requirements. Activation offloading (O) enables training with PP degree 5 and ZeRO-2 degree 4, though H100 GPUs remain underutilized despite 3× higher speed than A100s, lacking memory for additional layers. Pipeline-Efficient ZeRO DP with interleaved updates (I) decreases memory usage by 26% through parameter offloading and interleaved optimizer updates. Heterogeneous Pipeline Parallelism (H) reduces pipeline stages to 4 by grouping node-local GPUs, achieving balanced computation and trading modest memory increases for substantial performance gains.

Note that the main benefit of Pipeline-Efficient ZeRO DP is in reducing memory utilization compared to standard PP+DP. With unchanged parallelism strategy and partitioning, this yields marginal throughput gains. However, when GPU memory is limited, Pipeline-Efficient ZeRO DP can enable training configurations that use more memory but balance computation more effectively or improve computation-communication overlap. For instance, in this ablation study, the gain from adding heterogeneous pipeline parallelism is achievable only because Pipeline-Efficient ZeRO DP reduces memory usage sufficiently to support this configuration; without it, the model would not fit in memory.

4 Cluster Profiling

4.1 Network Bandwidth Profiling

Accurate measurement of inter-GPU communication bandwidth is essential for optimizing distributed training configurations in our planner.

4.1.1 Methodology

Our profiling framework systematically measures both intranode and internode bandwidth using controlled point-to-point communication tests.

1. **Cluster Information Collection**: We gather hardware details from each machine in the cluster, including GPU types, counts, and connectivity interfaces.

2. **Bandwidth Measurement**: We conduct point-to-point communication tests that measure both bandwidth (Gb/s) and latency (ms) between logical GPU pairs using PyTorch's distributed communication primitives.

3. **Backend Selection**: For intranode communication, we use NVIDIA's NCCL backend which leverages high-speed NVLink where available. For internode communication, we use the GLOO backend for P2P bandwidth and latency estimates.

4.1.2 Optimization Techniques

Profiling all pairs of GPUs in a large cluster would require $O(n^2)$ measurements, where n is the number of GPUs in the cluster, making the profiling process very expensive. We implement several optimizations to reduce profiling time:

Table 2. Zorse's training configurations used in the evaluation.

Cluster	Model	Batch Size	Sequence Length	Total Tokens	Microbatch Size	Ministages / GPU Group	GPU Groups
A	Llama 7B	256	4096	1M	4	1	2H100,2H100,8V100,8V100
A	Llama 13B	256	4096	1M	2	1	2H100,2H100,8V100,8V100
A	Llama 33B	256	4096	1M	2	4	2H100,2H100,8V100,8V100
A	Llama 65B	256	4096	1M	2	4	2H100,2H100,8V100,8V100
B	Llama 7B	1024	1024	1M	4	1	8A100,16V100,16A10G,24T4
B	Llama 13B	1024	1024	1M	4	4	8A100,16V100,16A10G,24T4
B	Llama 33B	1024	1024	1M	1	6	8A100,16V100,16A10G,24T4
C	Llama 7B	2048	512	1M	4	1	8V100,8V100,24T4,24T4,16A10G,24T4,24T4
C	Llama 13B	2048	512	1M	2	1	8V100,8V100,24T4,24T4,16A10G,24T4,24T4
C	Llama 33B	2048	512	1M	1	3	4V100,4V100,4V100,4V100,24T4,24T4,16A10G,24T4,24T4

Logical Pair Reduction for Intranode Testing Instead of measuring bandwidth between all possible GPU pairs within a node, we identify the set of logical GPU pairs for each machine with unique GPU and networking characteristics. For a machine with x unique GPU types, we measure bandwidth between $\frac{x(x-1)}{2}$ pairs, which represents all unique combinations of GPU types within the node.

For example:

- In a homogeneous machine with 8 identical GPUs, we measure only 1 pair since all GPUs have identical interconnect characteristics
- In a heterogeneous machine with 4 A100 GPUs and 4 H100 GPUs, we measure 3 pairs: (A100→A100), (H100→H100), (A100→H100)

The communication test uses non-blocking operations (isend/irecv) with sufficient warmup iterations. Furthermore, these intranode tests are conducted in parallel across all machines in the cluster.

Logical Pair Reduction for Internode Testing For internode communication, we apply a similar logical pair reduction approach:

1. **GPU Type Representation**: For each machine pair, we select one representative GPU of each unique type from each machine. For a pair of machines with x and y unique GPU types respectively, this reduces testing complexity from $n \times m$ pairs to $x \times y$ pairs, where n and m are the total number of GPUs on each machine. In cloud environments where GPUs within a machine are typically homogeneous, this often reduces to measuring just a single pair of GPUs.
2. **Machine Type Grouping**: We further optimize by grouping machines based on:
   - Geographic region/datacenter
   - Hardware configuration (exact GPU types and counts)

For each group of identical machines, we select one representative machine. For each pair of machine groups, we measure bandwidth between their representatives and propagate these measurements to all machine pairs between these groups. This approach reduces the testing complexity from $O(M^2)$ to $O(G^2)$, where M is the total number of machines and G is the number of unique machine configurations (typically $G \ll M$).

5 Performance Models

5.1 Latency Model.

We model the total training latency as:

$$L_{total} = (L_{forwards} + L_{backwards}) \cdot N_{ministages} + L_{startup} \quad (1)$$

where $L_{forwards}$, $L_{backwards}$ are the latencies to process one ministage in the forward and backward passes, respectively, $N_{ministages}$ is the number of ministages per GPU group, and $L_{startup}$ is the latency to initialize the pipeline.

Ministage Forward and Backward Pass Latency. Let $F_{i,g}$ be the forward pass latency for running a ministage for all microbatches on GPU g in GPU group i, and L_{send} be the latency required to send all microbatches between neighboring GPU groups. The compute latency for processing one ministage in the forward pass of the pipeline is:

$$C_{forwards} = \max_{G_i} \max_{g \in G_i} \max(F_{i,g}, L_{send}) \quad (2)$$

The inner max accounts for the overlap between computation and communication of microbatches, while the outer max terms model the forwards pass being bottlenecked by the slowest ministage across all GPU groups. To account for the overlap of the forward computation with the AllGather of parameters, the overall latency for a ministage in the forward pass is:

$$L_{forwards} = \max(C_{forwards}, L_{AG}, L_{offload}) \quad (3)$$

where L_{AG} is the latency of the AllGather for the ministage parameters, and $L_{offload}$ is the latency for offloading and reloading both ministage parameters and activations. We model the AllGather latency based on profiled network bandwidths, assuming a ring-based AllGather implementation (Jiang et al., 2020; Strati et al., 2024). Given a ministage size of P GB and $|G_i|$ GPUs in the group, each GPU will transmit a total of $\frac{(|G_i|-1) \cdot P}{|G_i|}$ GB of data. This data exchange

occurs in parallel and is bottlenecked by the slowest network bandwidth b_{min} between any two GPUs in the group. Thus, the communication time for AllGather is:

$$L_{AG} = \frac{(|G_i| - 1) \cdot P}{|G_i| \cdot b_{min}} \tag{4}$$

Offloading latency can be estimated by dividing the total bytes offloaded and loaded for model parameters and activations by the PCIe bandwidth.

The backward computation (which includes the forwards recomputation of activations) is overlapped with another AllGather of parameters along with ReduceScatter to average the gradients:

$$L_{backwards} = \max(C_{backwards}, L_{AG} + L_{RS}) \tag{5}$$

where $C_{backwards}$ is the latency for processing a ministage in the backwards pass, L_{RS} the latency for ReduceScatter, modeled like $C_{forwards}$ and L_{AG}.

Pipeline Startup Latency. The pipeline startup latency $L_{startup}$ is computed by summing the AllGather latency of the first ministage and the send latencies for the first $K - 1$ microbatches, which are not overlappable with any computation:

$$L_{startup} = L_{AG} + L_{send} \cdot (K - 1) \tag{6}$$

where K is the number of GPU groups.

5.2 Memory Model.

We model the total per-GPU memory consumption as:

$$M_{total} = M_{params} + M_{grads} + M_{optim} + M_{activations} \tag{7}$$

where M_{params} represents memory for model parameters, M_{grads} gradients, M_{optim} optimizer states, and $M_{activations}$ activations. Let N_{TL} and N_{ML} be the total number of layers assigned to a GPU group G, and the number of layers per ministage, respectively. In FP16 mixed-precision training, a master copy of the model parameters are kept sharded in full precision, while at most 2 layers in half precision are materialized in memory. Then, total memory (bytes) for the model parameters is

$$M_{params} = 4 \cdot \left(\frac{N_{TL} \cdot P_{layer}}{|G|} \right) + 2 \cdot (2 \cdot N_{ML} \cdot P_{layer}) \ .$$

With Pipeline-Efficient ZeRO DP, there are gradients for at most one ministage in memory at a time. The gradient memory is

$$M_{grads} = 2 \cdot (N_{ML} \cdot P_{layer}) \ .$$

With a standard optimizer like Adam (Kingma & Ba, 2015), the first and second moments of the gradient are stored as optimizer state in full precision, and so the sharded optimizer state requires

$$M_{optim} = 4 \cdot \left(2 \cdot \frac{N_{TL} \cdot P_{layer}}{|G|} \right) \ .$$

Finally, with offloading, only two sets of boundary activations are kept in memory, which amount to

$$M_{activations} = 2 \cdot (b \cdot s \cdot h)$$

where b is the microbatch size, s is the sequence length, and h is the model hidden dimension.

Table 3. Datacenter-class GPU Specifications

Performance	GPU	Memory	TFlops (FP16)
High	H100-NVL	94 GB	989
	A100	40/80 GB	312
Middle	V100	16 GB	125
	A10G	24 GB	125
Low	T4	16 GB	65

6 GPU Specifications

In Table 3, we provide the specifications of the GPUs used in the evaluation.

7 Tensor Parallelism Integration

The current design of Zorse combines PP and DP for three practical reasons. First, in a large majority of heterogeneous training use cases, using sharded DP is more communication-efficient than TP while being equally memory-efficient. As shown in Figure 2 of the paper, TP suffers from poor GPU utilization on most such VMs due to limited interconnect bandwidth, with only the 8×A100 with NVSwitch VM achieving acceptable TP performance. Second, for our targeted model sizes and cluster scales, Pipeline-Efficient ZeRO DP already reduces memory use efficiently without TP's communication overhead. The batch size constraints that TP resolves (global_batch_size $<$ num_GPUs) are only relevant for very large clusters with 1024+ GPUs, while heterogeneous clusters are typically much smaller due to resource constraints (Zhang et al., 2024b; Guo et al., 2024; Jiang et al., 2025; Yan et al., 2025; Um et al., 2024; Ding et al., 2021; Nie et al., 2024).

Future work may explore TP integration in settings where it becomes advantageous. One setting is in large clusters with widespread high-bandwidth interconnects or for models with single layers exceeding single-GPU memory capacity. TP in heterogeneous clusters would require *cross-mesh resharding*, since transitions between GPU groups with different TP configurations (e.g., 4 to 2) or between TP and

FSDP demand tensor layout conversions and careful coordination to limit communication overhead (Zhuang et al., 2023).

The following additions to Zorse would support TP without changing Zorse's core design:

1. Layout-aware resharding routines that overlap communication with computation.
2. Updating planner latency and memory models to accurately account for TP's communication and memory overheads.
3. Considering TP when searching for the parallelization strategy in the planner.

REFERENCES

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Ding, Y., Botzer, N., and Weninger, T. Hetseq: Distributed gpu training on heterogeneous infrastructure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 15432–15438. AAAI Press, May 2021. doi: 10.1609/aaai.v35i17.17813.

Facebook Incubator. Gloo: Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo, 2025. Accessed: 2025-04-17.

Guo, R. B., Anand, U., Chen, A., and Daudjee, K. Cephalo: Harnessing heterogeneous gpu clusters for training transformer models, 2024. URL https://arxiv.org/abs/2411.01075.

Jiang, Y., Zhu, Y., Lan, C., Yi, B., Cui, Y., and Guo, C. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association. ISBN 978-1-939133-19-9.

Jiang, Y., Fu, F., Yao, X., He, G., Miao, X., Klimovic, A., Cui, B., Yuan, B., and Yoneki, E. Demystifying cost-efficiency in llm serving over heterogeneous gpus, 2025. URL https://arxiv.org/abs/2502.00722.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Nie, C., Maghakian, J., and Liu, Z. Cannikin: Optimal adaptive distributed dnn training over heterogeneous clusters. In *Proceedings of the 25th International Middleware Conference*, Middleware '24, pp. 299–312, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706233. doi: 10.1145/3652892.3700767. URL https://doi.org/10.1145/3652892.3700767.

NVIDIA. NCCL: NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl, 2024.

NVIDIA Corporation. Point to point communication functions. NVIDIA NCCL User Guide API documentation, version 2.26.2, 2020. URL https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/api/p2p.html. Accessed: 2025-04-17.

Strati, F., Elvinger, P., Kerimoglu, T., and Klimovic, A. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, pp. 107–116, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705410. doi: 10.1145/3642970.3655843. URL https://doi.org/10.1145/3642970.3655843.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Um, T., Oh, B., Kang, M., Lee, W.-Y., Kim, G., Kim, D., Kim, Y., Muzzammil, M., and Jeon, M. Metis: Fast automatic distributed training on heterogeneous {GPUs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 563–578, 2024.

Yan, R., Jiang, Y., Nie, X., Fu, F., Cui, B., and Yuan, B. Hexiscale: Accommodating large language model training over heterogeneous environment, 2025. URL https://arxiv.org/abs/2409.01143.

Zhang, P., Zeng, G., Wang, T., and Lu, W. Tinyllama: An open-source small language model, 2024a.

Zhang, S., Diao, L., Wu, C., Cao, Z., Wang, S., and Lin, W. HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis. In *Proceedings of the European Conference on Computer Systems (EuroSys '24)*, pp. 18, New York, NY, USA, 2024b. ACM. doi: 10.1145/3627703.3629580. URL https://doi.org/10.1145/3627703.3629580.

Zhuang, Y., Zhao, H., Zheng, L., Li, Z., Xing, E. P., Ho, Q., Gonzalez, J. E., Stoica, I., and Zhang, H. On optimizing the communication of model parallelism. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 526–540. Curan, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/a42cbafcabb6dc7ce77bfe2e80f5c772-Paper-mlsys2023.pdf.