ZeRO from Scratch: Evaluating Memory Savings, Communication Overhead, and Bandwidth Sensitivity

Thomason Zhao
Stanford University

Daniel Adkins
Stanford University

Abstract

Large language model training is increasingly constrained not only by compute, but also by the memory and communication costs of distributed optimization across many devices. While ZeRO reduces the memory redundancy of data-parallel training by partitioning optimizer states, gradients, and parameters, its practical efficiency depends on how these increasingly aggressive sharding strategies interact with limited network bandwidth. In this paper, we reimplement ZeRO stages 0-3 from scratch and study their memory savings, communication overhead, and bandwidth sensitivity in order to understand which stages remain effective outside high-bandwidth datacenter settings. We implement each stage in PyTorch using distributed communication primitives and evaluate them under controlled bandwidth throttling while measuring memory usage, communication behavior, and end-to-end training throughput. We find that all ZeRO stages provide the expected memory savings, and that unexpectedly ZeRO stage 3 delivers the best throughput across the tested bandwidth range, including under low-bandwidth constraints, despite its higher nominal communication cost. These results suggest that the effectiveness of distributed training strategies depends not only on raw communication volume, but also on the interaction between memory savings, batch scaling, and communication overlap, which can allow aggressively sharded methods to remain practical even in bandwidth-limited environments.

1 Introduction

Training large language models has become a core systems problem because model quality increasingly scales with parameter count, while the memory and communication demands of distributed training grow just as quickly [3, 4, 8, 9]. Prior work such as ZeRO [6] showed that standard data parallel training wastes substantial memory by replicating optimizer states, gradients, and parameters across devices, and that careful partitioning of these states can dramatically increase the size of models that fit on a fixed cluster. In the original setting, ZeRO was especially compelling because it preserved much of the computational efficiency of data parallelism while reducing memory redundancy enough to support models far beyond what conventional distributed data parallel training could handle.

However, the assumptions under which ZeRO was introduced do not fully capture the environment in which distributed LLM training may increasingly take place. The original paper emphasizes that communication, especially across node boundaries, can quickly become a limiting factor as bandwidth drops, and that model parallelism in particular suffers when it extends beyond a single high-bandwidth node. Our project asks how ZeRO's stage-wise memory-communication tradeoff changes when training moves from datacenter-style interconnects [5, 10] to more modest or unstable networks characteristic of commodity or widely distributed infrastructure. In that setting, the central question is not only whether later ZeRO stages introduce more communication, but whether that added communication actually outweighs the gains from lower memory footprint, improved batch sizing, and better hardware utilization.

In this paper, we revisit ZeRO from first principles and study the network-consumption behavior of ZeRO stages 0 through 3 under constrained bandwidth. Rather than proposing a new optimizer, we reimplement each stage ourselves using PyTorch distributed primitives [1, 2] and instrument the resulting system to measure memory usage, communication volume, and training throughput. This lets us move beyond the original paper's mostly high-bandwidth setting and directly ask how each ZeRO stage behaves as bandwidth is reduced, how the communication pattern shifts from bulk synchronization to more fine-grained layer-wise transfers, and whether more aggressively sharded training remains effective when the network is no longer abundant.

Our methodology combines faithful reproduction with controlled systems experimentation. We implement ZeRO stage 0 as standard data parallel training, stage 1 with optimizer-state partitioning, stage 2 with gradient partitioning via reduce-scatter plus parameter all-gather, and stage 3 with parameter sharding and layer-wise parameter materialization during forward and backward passes. We validate correctness by

matching loss trajectories across stages, then run a structured experiment matrix over model size, ZeRO stage, and available network bandwidth. To emulate lower-bandwidth distributed environments, we explicitly throttle inter-node communication and profile both end-to-end throughput and the underlying collective behavior, allowing us to connect measured performance back to each stage's theoretical communication pattern.

Our evaluation shows that the ZeRO memory-communication tradeoff is more subtle than the standard communication analysis alone would predict. As expected, higher ZeRO stages deliver substantially greater memory savings through progressively more aggressive state partitioning. Surprisingly, however, stage 3 achieves the best throughput across the tested bandwidth range, including in low-bandwidth settings, despite its higher nominal communication cost. This suggests that the practical efficiency of ZeRO depends not only on how much communication each stage performs, but also on how memory savings improve microbatch size, utilization, and communication-computation overlap. By sweeping bandwidth directly, we provide a more realistic picture of how ZeRO behaves once one leaves the assumptions of tightly coupled datacenter clusters.

Our contributions are as follows:

- We present a from-scratch reimplementation of ZeRO stages 0-3 using PyTorch distributed primitives.
- We provide a controlled empirical study of how ZeRO stages trade off memory savings, communication overhead, and throughput as network bandwidth is reduced.
- We show that despite its higher nominal communication cost, ZeRO stage 3 achieves the best throughput across our tested bandwidth range.
- We use these results to evaluate the suitability of current distributed LLM training schemes for lower-bandwidth, commodity-style infrastructure.

2 Background & Related Work

2.1 Background

Training large language models across multiple GPUs requires balancing three resources at once: compute, memory, and communication. The most common baseline is data parallelism, where each device stores a full copy of the model, computes gradients on a different mini-batch shard, and synchronizes those gradients across devices. This approach is simple and often efficient, but it does not reduce per-device model-state memory, since parameters, gradients, and optimizer states are replicated on every worker. When models become too large to fit comfortably on one device, practitioners instead turn to model parallelism or pipeline parallelism, which partition the model itself across devices at the cost of additional communication and implementation complexity.

A useful way to understand the memory bottleneck is to separate training memory into *model states* and *residual states*. Model states include the parameters, gradients, and optimizer statistics; for adaptive optimizers such as Adam, optimizer states can dominate memory usage, especially under mixed-precision training where fp16 model copies coexist with fp32 master weights and moment estimates. Residual memory includes activations, temporary communication buffers, and fragmentation overhead. The key insight of ZeRO is that conventional data parallel training wastes memory by redundantly replicating model states across all workers even though not all of those states are needed on every device at every moment of execution.

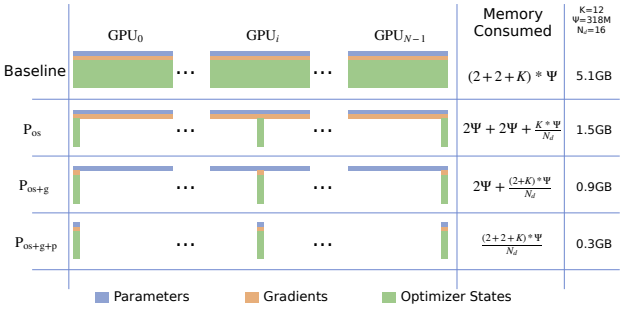


Figure 1: Per-GPU memory consumption under different partitioning strategies, from the baseline to full partitioning of parameters, gradients, and optimizer states. (adopted from [6])

ZeRO addresses this redundancy by progressively partitioning model states across data-parallel workers, as shown in Figure 1. In the terminology commonly used today, ZeRO stage 1 partitions optimizer states, stage 2 additionally partitions gradients, and stage 3 also partitions parameters. The appeal of this design is that it keeps the programming model close to data parallelism while reducing per-device memory substantially: the original ZeRO analysis argues that stage 1 and stage 2 preserve approximately the same communication volume as standard data parallel training, while stage 3 further reduces memory at the cost of extra parameter communication during forward and backward execution. This stage-wise structure is exactly what makes ZeRO a good target for a bandwidth-sensitivity study: each stage changes not only how much memory is saved, but also when and how communication is performed.

From a communication perspective, the distinction between stages is especially important. Standard data parallel training is typically implemented with all-reduce over gradients, which can be understood as a reduce-scatter followed by an all-gather. ZeRO stage 2 replaces replicated gradient synchronization with reduce-scatter of gradient shards plus an all-gather of updated parameters, while stage 3 goes further by materializing parameter shards on demand, typically through layer-wise or module-wise all-gather before computation and releasing them afterward. As a result, later stages may save more

memory but become more exposed to network bandwidth and communication scheduling effects, particularly once training moves beyond tightly coupled, high-bandwidth clusters.

2.2 Related Work

Prior work on distributed training has explored several ways to scale large models while managing memory and communication costs. Tensor/model parallel approaches such as Megatron-LM [9] partition computation within layers, while pipeline methods such as GPipe [4] and PipeDream [3] partition models across layers; both reduce per-device memory but introduce additional communication and scheduling complexity. ZeRO [6] takes a different approach by preserving the data-parallel training model while progressively partitioning optimizer states, gradients, and parameters to remove memory redundancy. Related techniques such as activation checkpointing, CPU offloading, and memory-efficient optimizers can further reduce memory usage and are often complementary to ZeRO. More recent systems such as PyTorch FSDP [10] extend the same sharded data-parallel idea into production training stacks. Our work builds on this line of research by revisiting ZeRO from scratch and focusing specifically on how its stage-wise memory savings interact with communication under constrained bandwidth.

3 System

We build a from-scratch distributed training system in PyTorch that reimplements ZeRO stages 0-3 and exposes their memory and communication behavior under controlled network constraints. Rather than reproducing the full DeepSpeed [7] software stack, our goal is to construct a minimal and transparent implementation that preserves the core semantics of each ZeRO stage while enabling direct measurement of memory usage, communication volume, and end-to-end training throughput. This design lets us study not only whether ZeRO reproduces its expected memory savings, but also how its communication patterns behave as bandwidth becomes constrained.

3.1 ZeRO Stage Reimplementation

Our implementation realizes ZeRO as a sequence of progressively more aggressive sharding strategies over model states, starting from a standard data-parallel baseline. Stage 0 corresponds to ordinary data parallel training, where every worker stores a full copy of the parameters, gradients, and optimizer states. Each rank computes forward and backward passes on its local minibatch shard, and gradients are synchronized across workers after backpropagation before the optimizer step is applied redundantly on every device. This stage serves as both the correctness baseline and the communication reference point for the rest of the system.

Stage 1: introduces optimizer-state partitioning while keeping parameters and gradients fully replicated. In our implementation, each rank owns and updates only the optimizer state associated with its assigned parameter shard, reducing persistent memory without significantly changing the structure of the training loop. Since gradients remain replicated, the communication pattern stays close to standard data parallel training, and the main benefit of this stage is memory reduction with relatively modest additional complexity. This makes stage 1 the simplest ZeRO variant that departs from full redundancy.

Stage 2: extends this design by partitioning gradients in addition to optimizer states. Rather than materializing full reduced gradients on every rank, our implementation uses reduce-scatter so that each worker receives only the gradient shard corresponding to the parameters it owns. Each rank then performs the optimizer update locally on its shard, after which updated parameters are synchronized so that the next iteration begins from a consistent model state. Compared with stage 1, this stage reduces both optimizer-state and gradient memory while still keeping the execution model close to data parallelism. In practice, stage 2 is an important middle ground because it achieves stronger memory savings without requiring parameter communication to enter the critical path of forward execution.

Stage 3: further partitions parameters, removing the need for each worker to keep a full model replica resident throughout training. In this stage, parameters are materialized on demand at module granularity: before a module executes, its parameter shards are all-gathered into a temporary full representation, and once computation finishes, the gathered copy is released. During backward propagation, the same parameters must be gathered again so gradients can be computed, after which gradients are reduce-scattered back to their owning ranks. This stage offers the greatest memory reduction, but it also changes the communication structure most dramatically by moving communication directly into both the forward and backward paths. As a result, stage 3 has the most communication-intensive execution pattern and is central to our study of how ZeRO behaves outside high-bandwidth datacenter environments.

Across these four stages, our reimplementation is designed to preserve the essential semantics of ZeRO while exposing the communication behavior as clearly as possible. Stages 0-2 primarily differ in how optimizer states and gradients are stored and synchronized after computation, whereas stage 3 additionally changes when communication occurs by requiring repeated parameter materialization during execution. This distinction is crucial for our experiments, since our goal is not only to compare memory savings across stages, but also to understand how increasingly aggressive sharding changes the timing, frequency, and performance impact of communication.

3.2 Measurement Support

To study ZeRO under bandwidth-constrained conditions, we augment the implementation with measurement support that

makes communication and performance directly observable. For each experiment configuration, the system records end-to-end throughput, communication time, communication volume, and memory usage while running a fixed training workload. A key part of this support is controlled bandwidth throttling: by intentionally varying available network bandwidth, we move beyond the high-bandwidth assumptions typically associated with large datacenter clusters and can observe how the different ZeRO stages behave as communication becomes more constrained. This is especially important for stage 3, whose repeated parameter all-gathers make its runtime behavior more tightly coupled to the execution schedule than stages 0-2.

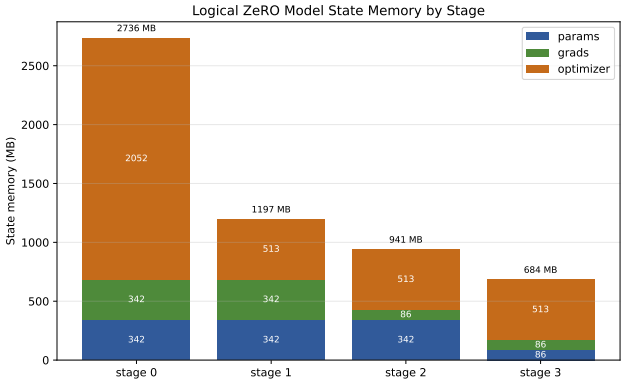
The measurement harness is also designed to distinguish persistent memory savings from runtime overhead. In addition to overall throughput, we track both long-lived model-state memory and transient execution-time overhead from temporary communication buffers or gathered parameter copies. These measurements allow us to explain not only which stage performs better under a given network regime, but also why, by relating observed performance back to the underlying memory and communication behavior. Although the system is intentionally minimal and does not attempt to reproduce every optimization of production frameworks, it is sufficient for our goal of exposing the core ZeRO memory-communication tradeoff in a controlled and interpretable way.

4 Evaluation

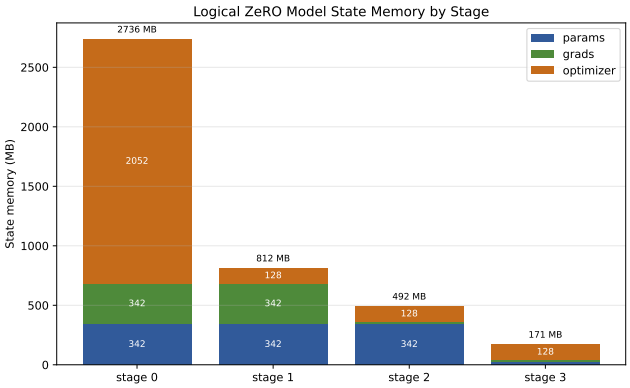
We evaluate our from-scratch ZeRO implementation on a single 16-GPU server with NVIDIA RTX 4090 GPUs (24 GB memory and 82 TFLOP/s each). Our experiments focus on three aspects of ZeRO behavior: memory consumption, throughput under different bandwidth conditions, and scaling as the number of GPUs increases. We use two LLaMA 3.1-based transformer models with 318M and 1.34B parameters, and run each setup for enough steps to verify decreasing loss and collect stable steady-state measurements rather than training to full convergence. We report model-state memory, residual memory, and training performance in tokens/s or TFLOP/s. Unlike the original ZeRO paper, which evaluates much larger DGX-2-scale systems, our setup is deliberately smaller and is intended to study the same tradeoffs under more modest hardware conditions.

4.1 Verifying Memory Consumption and Memory Access Behavior

We first verify that the measured model-state memory across ZeRO stages matches the expected partitioning behavior. Figure 2 shows the logical model-state memory for stages 0–3 on 4 GPUs and 16 GPUs. On 4 GPUs, stage 0 requires 2736 MB of model-state memory, stage 1 reduces this to 1197 MB, stage 2 further reduces it to 941 MB, and stage 3 reaches 684 MB. On 16 GPUs, the same trend continues more



(a) 4 GPUs



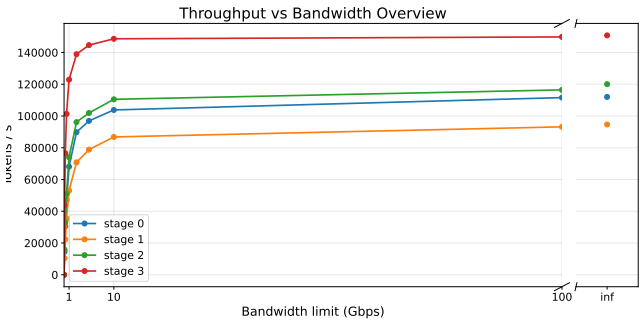
(b) 16 GPUs

Figure 2: Logical ZeRO model-state memory measured across stages for 4-GPU and 16-GPU runs under the same configuration.

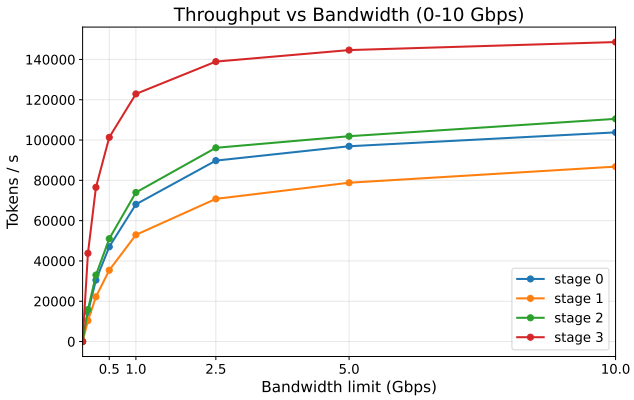
aggressively: stage 0 remains at 2736 MB, while stage 1 drops to 812 MB, stage 2 to 492 MB, and stage 3 to 171 MB. These numbers match the expected ZeRO progression: stage 0 fully replicates parameters, gradients, and optimizer states; stage 1 shards optimizer states only; stage 2 shards optimizer states and gradients; and stage 3 shards parameters as well. The reduction is especially strong when moving from 4 to 16 GPUs, confirming that the memory savings scale with data-parallel degree in the way predicted by ZeRO's model-state analysis.

The breakdown in Figure 2 also helps verify that the memory *access scheme* of our implementation matches the intended semantics of each stage, as shown in Figure 1. In both the 4-GPU and 16-GPU runs, stages 0 and 1 keep full parameter and gradient replicas, while stage 1 alone reduces optimizer-state memory; stage 2 then reduces gradient residency by switching from replicated gradients to partitioned gradient ownership; and stage 3 reduces parameter residency as well, consistent with on-demand parameter materialization. In other words, the measured memory layout is not merely smaller overall, but smaller in exactly the components that

should be affected by each stage. This gives us confidence that the implementation correctly realizes the state-partitioning design of ZeRO rather than relying on incidental differences in runtime allocation behavior, and overall the measured memory behavior matches our hypothesis closely.



(a) Full bandwidth range.



(b) Zoomed view for 0-10 Gbps.

Figure 3: Throughput under different bandwidth limits for ZeRO stages 0–3. The overview plot shows the full trend, and the zoomed-in plot highlights the low-bandwidth regime from 0 to 10 Gbps.

4.2 Throughput Under Bandwidth Constraints

We next evaluate how training throughput changes as available bandwidth is reduced. This is the main extension beyond the original ZeRO paper: rather than only asking whether ZeRO reduces memory, we ask when the communication introduced by each stage becomes the limiting factor. Figure 3 shows throughput across bandwidth limits for ZeRO stages 0–3, including a zoomed view of the low-bandwidth regime from 0 to 10 Gbps. As expected, throughput decreases for all stages as bandwidth is reduced. However, the most notable result is that stage 3 remains the best-performing stage across the full bandwidth range, including at low bandwidth.

At first glance, this seems counterintuitive. The original ZeRO analysis argues that stage 1 and stage 2 preserve communication volume close to standard data parallelism, while stage 3 introduces additional parameter communication and therefore carries higher nominal communication cost. If communication volume alone determined performance, we would expect stage 3 to degrade more sharply as bandwidth falls. Our measurements show that this is not the whole story in our setting. For the transformer models we study, the memory savings of stage 3 allow larger microbatch sizes and more favorable operating points, and those gains are large enough to outweigh the nominal communication penalty. Stage 3 also uses a finer-grained communication schedule, materializing parameters at module granularity and reduce-scattering gradients after use, so its practical behavior is shaped not only by how much communication it performs, but also by when that communication occurs.

The key point is that practical throughput depends on more than communication volume. It depends on the interaction between memory savings, batch-size scaling, and communication scheduling. In our experiments, stage 3 benefits enough from its reduced model-state footprint that it remains throughput-optimal even when bandwidth is constrained. This does not contradict the original ZeRO analysis. Instead, it sharpens the systems lesson: the best ZeRO stage is determined by the full memory-communication tradeoff, not by communication volume in isolation.

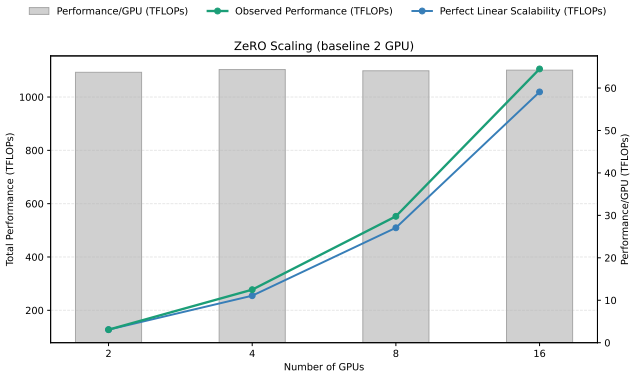


Figure 4: Scaling performance of ZeRO from a 2-GPU baseline, showing per-GPU throughput, observed total throughput, and ideal linear scaling.

4.3 Scaling Behavior and Superlinear Speedup

Finally, we evaluate how ZeRO scales as the number of GPUs increases. Figure 4 plots both per-GPU performance and total observed performance, using the 2-GPU configuration as the baseline and comparing against the ideal linear scaling trend. We observe a clear superlinear scaling effect: as the number of GPUs increases from 2 to 4, 8, and 16, total throughput grows faster than the ideal linear baseline, and per-GPU throughput also increases rather than remaining constant. In other words,

adding more GPUs not only provides more aggregate compute, but also improves the efficiency of each individual GPU.

This behavior is consistent with the mechanism described in the original ZeRO paper. As the data-parallel degree increases, ZeRO reduces per-GPU model-state memory, which improves the effective operating point of training and raises arithmetic intensity. The original paper emphasizes that this effect becomes especially strong when the saved memory is converted into larger per-GPU microbatches. Our current scaling figure already shows the same qualitative trend, but it likely understates the full magnitude of the effect. If we ran the scaling experiment using memory-fit batches at each GPU count, rather than holding the workload more fixed, we would expect the superlinear behavior to become even more pronounced and to more closely resemble the original ZeRO result. That is exactly the mechanism highlighted in the original paper: larger GPU counts reduce per-device memory pressure, which in turn enables larger per-GPU batches and better hardware utilization.

More broadly, this result reinforces a central theme of our evaluation. ZeRO's benefits are not limited to fitting larger models in memory. By lowering persistent model-state memory, it can also improve the operating regime of training and yield higher realized throughput as the system scales out. In our experiments, the superlinear trend therefore provides additional evidence that ZeRO's memory savings translate into concrete performance gains, rather than serving only as a capacity advantage.

5 Discussion & Limitation

Our results reinforce the central insight of ZeRO: reducing memory redundancy is highly effective, but the practical impact of sharding depends on more than nominal communication volume alone. As expected, higher ZeRO stages achieve progressively stronger memory savings, with stage 3 providing the smallest model-state footprint. Surprisingly, however, stage 3 also delivers the best throughput across our tested bandwidth range, including under low-bandwidth constraints. This suggests that in our setting, the gains from lower memory pressure, larger microbatch sizes, and better communication-computation overlap outweigh the additional communication introduced by more aggressive sharding. Rather than showing that later stages necessarily lose their advantage as bandwidth drops, our results indicate that the realized efficiency of ZeRO depends on the full interaction between memory savings, execution schedule, and network conditions.

At the same time, these findings should be interpreted in the context of a deliberate reproduction-oriented design. We implement ZeRO stages 0-3 from scratch using PyTorch distributed primitives and controlled bandwidth sweeps to expose the core memory-communication tradeoff clearly, rather than to match the full optimization level of DeepSpeed or the original ZeRO implementation. Our experiments are conducted on a 16-GPU cluster built from RTX 4090 GPUs, which is far smaller in scale than the hardware used in the original paper. As a result, our measurements are most useful for understanding the relative behavior of ZeRO stages and the mechanisms behind their performance, rather than for claiming absolute parity with the throughput or scalability numbers reported in the original large-scale evaluation.

Our study also has several practical limitations. First, because our experiments are performed at a much smaller scale, the behavior we observe, including stage 3's strong performance under constrained bandwidth, may shift on larger clusters, different interconnects, or workloads with substantially larger models. Second, our low-bandwidth setting is created through controlled communication throttling, which is useful for isolating bandwidth as an experimental variable but does not fully capture the noise, contention, and heterogeneity of real commodity or wide-area deployments. Third, to keep the comparison interpretable, we focus on the core ZeRO stages and do not fully explore complementary optimizations such as activation checkpointing, offloading, overlap strategies, or hybrid combinations with model and pipeline parallelism. Future work could extend this study to larger clusters, more realistic heterogeneous networks, and richer training stacks to better understand how sharded training schemes behave outside datacenter-scale infrastructure.

6 Conclusion

In this paper, we present a from-scratch reimplementation of ZeRO stages 0-3 and evaluate their memory savings, communication overhead, and bandwidth sensitivity under controlled network constraints. We confirm the expected memory savings of progressively more aggressive sharding, but find that practical performance is more nuanced than communication volume alone would predict: despite its higher nominal communication cost, stage 3 achieves the best throughput across our tested bandwidth range, including at low bandwidth. This suggests that the benefits of reduced memory pressure, larger microbatch sizes, and communication-computation overlap can outweigh the extra communication introduced by deeper sharding. As distributed LLM training moves beyond high-bandwidth datacenter clusters, understanding this full memory-communication tradeoff will be critical for choosing effective training strategies.

Acknowledgments

We would like to express our sincere gratitude to Dr. David Mazières and Dr. Keith Winstein for their valuable guidance and instruction throughout this course. We also appreciate their effort in organizing and offering this course, which made this class project possible.

AI Usage: To implement this paper, we used AI agent tooling such as Claude Code. The overall goals of our study study,

experiments, and the high-level architecture were designed by us. Implementation was a roughly 60 percent AI, 40 percent human effort by lines of code. The process involved us iteratively deciding how to further improve our ZeRO design based on the original paper and the goals of our study; we found that AI agents frequently exhibited "reward hacking" behavior where they would find erroneous ways to implement systems that yielded results which did not hold up under further scientific scrutiny. One example of this was consistent false justifications surrounding a result that we were investigating surrounding the behavior of Stage 3 vs. Stage 2 of the algorithm under extremely low bandwidth, which was sufficiently complex that AI coding tools failed to produce a scientifically-justifiable answer. As a result, the majority of our AI usage was focused on commodity infrastructure, rapid debugging tools, and monitoring of large logs of experimental data we found, as opposed to the design or interpretation of the experiments themselves and the implementation of critical pieces of the algorithms and infrastructure. We did find it helpful in finding improvements to existing infrastructure which could be objectively benchmarked in a clearly interpretable way, such as discovering blocking ‘torch.cuda.synchronize()‘ calls that we recognized were causing unexpected results in certain timing measurements.

Availability

References

[1] NVIDIA Collective Communications Library (NCCL).

[2] PyTorch Distributed Overview — PyTorch Tutorials 2.11.0+cu130 documentation.

[3] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. PipeDream: Fast and Efficient Pipeline Parallel DNN Training, June 2018. arXiv:1806.03377 [cs].

[4] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Yonghui Wu. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[5] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, St. Louis Missouri, November 2021. ACM.

[6] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pages 1–16. IEEE, 2020.

[7] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, Virtual Event CA USA, August 2020. ACM.

[8] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow, February 2018. arXiv:1802.05799 [cs].

[9] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, March 2020. arXiv:1909.08053 [cs].

[10] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel, September 2023. arXiv:2304.11277 [cs].