Distributed Training

Parallelism Strategies for Training Large Language Models

February 2026 • Technical Report

Table of Contents

1. Introduction

Training large language models requires computational resources that vastly exceed the capacity of any single accelerator. A model with hundreds of billions of parameters may require tens of terabytes of memory when accounting for weights, gradients, optimizer states, and activations, while the training computation itself may take millions of GPU-hours. Distributed training techniques split this workload across hundreds or thousands of accelerators, enabling the training of models that would otherwise be physically impossible to build.

This report examines the primary parallelism strategies used in distributed LLM training: data parallelism, tensor parallelism, pipeline parallelism, and the ZeRO family of memory optimizations. It also covers the communication patterns, failure handling, and systems engineering challenges that arise when training at scale.

2. Data Parallelism

2.1 Standard Data Parallelism

Data parallelism is the simplest and most widely used form of distributed training. Each accelerator holds a complete copy of the model and processes a different subset (micro-batch) of the training data. After each forward and backward pass, gradients are aggregated across all replicas using an all-reduce collective communication operation, ensuring that all model copies remain synchronized. The effective batch size scales linearly with the number of accelerators, and the per-step throughput scales nearly linearly for compute-bound workloads.

2.2 Limitations at Scale

Standard data parallelism requires each accelerator to store a full copy of the model weights, gradients, and optimizer states. For a 70-billion-parameter model in mixed precision with Adam optimizer, this amounts to roughly 840 GB per replica, which is far beyond the memory of any single GPU. Additionally, gradient all-reduce communication becomes a bottleneck as the number of accelerators grows, particularly when communication bandwidth does not scale proportionally with compute. These limitations motivated the development of memory-efficient variants and complementary parallelism strategies.

3. ZeRO: Memory-Efficient Data Parallelism

3.1 ZeRO Stages

The Zero Redundancy Optimizer (ZeRO), developed by Microsoft DeepSpeed, eliminates the memory redundancy inherent in standard data parallelism by partitioning model states across accelerators instead of replicating them. ZeRO operates in three

progressive stages. **ZeRO Stage 1** partitions optimizer states (e.g., Adam's momentum and variance), reducing memory by roughly 4x per GPU. **ZeRO Stage 2** additionally partitions gradients, achieving approximately 8x memory reduction. **ZeRO Stage 3** partitions the model weights themselves, enabling each GPU to store only a fraction of the total model. With ZeRO-3, the aggregate memory across all GPUs can hold the full model state, removing the per-GPU memory constraint.

3.2 FSDP (Fully Sharded Data Parallelism)

PyTorch's FSDP is a native implementation of ZeRO-3-style sharding integrated into the PyTorch framework. FSDP shards parameters, gradients, and optimizer states across data-parallel workers, gathering parameters on demand for computation and releasing them afterward. FSDP supports mixed precision training, activation checkpointing, and flexible sharding strategies that allow users to control the granularity of sharding. It has become the standard approach for memory-efficient data-parallel training in the PyTorch ecosystem.

4. Tensor Parallelism

Tensor parallelism splits individual layers of the model across multiple accelerators, with each device computing a portion of each layer's output. For transformer models, the most common approach splits the attention and feed-forward layers along the hidden dimension. The self-attention layer's query, key, and value projections are partitioned column-wise, and the output projection is partitioned row-wise, requiring only a single all-reduce per layer. This achieves high utilization because each device performs a meaningful fraction of each layer's computation.

Tensor parallelism is most effective within a single node where accelerators are connected by high-bandwidth interconnects (such as NVLink or NVSwitch, providing 400 to 900 GB/s between GPUs). The frequent all-reduce operations per layer make tensor parallelism sensitive to communication latency and bandwidth, making it impractical across nodes connected by slower network fabrics. Typical tensor parallel degrees range from 2 to 8, corresponding to the GPUs within a single node.

5. Pipeline Parallelism

5.1 Basic Pipeline Parallelism

Pipeline parallelism assigns different groups of consecutive layers to different accelerators, creating a pipeline through which micro-batches flow sequentially. Accelerator 1 processes layers 1 through N, accelerator 2 processes layers N+1 through 2N, and so on. Only inter-stage activations (the output of one stage and input to the next) need to be communicated between devices, making pipeline parallelism suitable for cross-node distribution where bandwidth is more limited.

5.2 The Pipeline Bubble Problem

Naive pipeline parallelism suffers from pipeline bubbles: idle time at the start and end of each training step when some stages have no micro-batches to process. The bubble fraction is approximately (P-1)/M where P is the number of pipeline stages and M is the number of micro-batches. GPipe addresses this by splitting the mini-batch into many micro-batches, reducing the bubble fraction. PipeDream and its variants introduce asynchronous pipeline schedules (1F1B scheduling) that start backward passes before all forward passes complete, further reducing idle time. State-of-the-art schedulers can reduce bubble overhead to a few percent of total training time.

6. Combining Parallelism Strategies

Large-scale training systems combine multiple parallelism strategies in what is called **3D parallelism** or **hybrid parallelism**. A typical configuration uses tensor parallelism within a node (across 8 GPUs connected by NVLink), pipeline parallelism across nodes within a rack or cluster, and data parallelism across groups of pipeline-parallel replicas. This layered approach allows each parallelism strategy to operate at the communication scale where it is most efficient, maximizing hardware utilization while minimizing communication overhead.

For MoE models, expert parallelism adds a fourth dimension, distributing different experts across different devices and using all-to-all communication to route tokens to the appropriate expert. The combination of expert parallelism with the three standard parallelism dimensions creates complex communication patterns that require careful orchestration to achieve high efficiency.

7. Parallelism Strategy Comparison

Strategy	What is Distributed	Communication Pattern	Best Suited For
Data Parallel	Data (batches)	All-reduce (gradients)	Scaling batch size
ZeRO / FSDP	Model state + data	All-gather + reduce-scatter	Memory-limited training
Tensor Parallel	Individual layers	All-reduce per layer	Intra-node (high BW)
Pipeline Parallel	Groups of layers	Point-to-point activations	Inter-node (lower BW)
Expert Parallel	MoE experts	All-to-all (token dispatch)	MoE architectures

8. Systems Engineering Challenges

8.1 Communication Optimization

At scale, communication efficiency is as important as compute efficiency. Techniques include overlapping communication with computation (pipelining gradient all-reduce with backward pass computation), gradient compression to reduce communication volume, hierarchical all-reduce that exploits the topology of the network (intra-node reduction before inter-node), and careful placement of communication primitives to avoid serialization bottlenecks.

8.2 Fault Tolerance and Checkpointing

When training runs across thousands of GPUs for weeks or months, hardware failures are not exceptional events but a regular occurrence. Training systems must implement robust checkpointing to save model state periodically and resume seamlessly after failures. Asynchronous checkpointing avoids blocking training for the duration of the checkpoint write. Elastic training frameworks can adapt to changing numbers of available accelerators without restarting from scratch. At the largest scales, mean time between failures for the cluster can be measured in hours, making fast recovery essential for training efficiency.

8.3 Determinism and Reproducibility

Achieving deterministic training results across distributed configurations is difficult due to non-associative floating-point arithmetic in collective operations, non-deterministic GPU kernel execution, and variations in communication ordering. While perfect bit-level reproducibility is often impractical at scale, approximate reproducibility through careful control of random seeds, communication ordering, and numerical precision is important for debugging and scientific rigor.

9. Future Directions

Distributed training continues to evolve as model scales push toward trillions of parameters and training clusters grow to tens of thousands of accelerators. Sequence parallelism, which distributes the sequence dimension for long-context training, is becoming essential as context windows extend beyond what a single device can hold. Context parallelism specifically distributes the attention computation across devices for very long sequences. Improved interconnects like NVLink 5.0, CXL, and custom network topologies are reducing communication bottlenecks. Compiler-driven automatic parallelism tools like XLA GSPMD and Alpa aim to discover optimal parallelism strategies automatically given a model and hardware configuration, reducing the need for manual parallel configuration.

10. Conclusion

Distributed training is the engineering backbone that makes large language models possible. The combination of data, tensor, pipeline, and expert parallelism enables models to scale to hundreds of billions of parameters across thousands of accelerators, while memory optimizations like ZeRO and FSDP ensure that the memory requirements of these enormous models can be met. As the field pushes toward ever-larger models and training runs, the systems challenges of communication efficiency, fault tolerance, and automated parallelism configuration will only grow in importance, making distributed training one of the most consequential areas of systems research in modern AI.