Case Study: Optimizing 5D Parallelism for Training a Financial Domain LLM

Section 1: Industry Context and Business Problem

Industry: Financial Services / AI Infrastructure

The financial services industry generates an extraordinary volume of unstructured text — SEC filings, earnings transcripts, analyst reports, regulatory documents, credit agreements, and real-time news. The global market for AI in financial services is projected to reach 64 billion USD by 2030, driven by demand for automated document understanding, compliance monitoring, and trading signal extraction. Yet most general-purpose LLMs perform poorly on financial tasks because they lack deep familiarity with domain-specific terminology, regulatory frameworks, and the precise numerical reasoning that finance demands.

Company Profile: Meridian AI

Meridian AI is a Series B fintech AI startup headquartered in New York, founded in 2022 by former quantitative researchers from Citadel and infrastructure engineers from Meta's FAIR lab. The company has raised USD 185M across two rounds (Seed: USD 15M from Sequoia; Series A: USD 50M led by Andreessen Horowitz; Series B: USD 120M led by Tiger Global). The team has grown to 124 employees, including a 38-person ML engineering team and a 12-person infrastructure team responsible for the training cluster.

Meridian's core product is **MeridianFin**, a suite of financial AI tools built on top of a proprietary domain-specific LLM:

- **DocAnalyst**: Automated extraction and summarization of SEC filings, credit agreements, and regulatory documents
- **ComplianceGuard**: Real-time monitoring of trading communications for regulatory violations (MiFID II, Dodd-Frank, SEC Rule 10b-5)
- **SignalStream**: Extraction of alpha signals from earnings calls, analyst reports, and macroeconomic publications

The company serves 23 enterprise clients, including 4 bulge-bracket investment banks and 7 mid-tier asset managers, generating USD 28M in ARR. The total addressable market for their product suite is estimated at USD 4.2 billion.

Business Challenge

Meridian's current production model is a fine-tuned Llama 3 70B, which achieves acceptable but not competitive performance on financial benchmarks. Internal evaluations show:

- **FinanceBench accuracy**: 71.3% (vs. 78.5% for GPT-4 Turbo on the same benchmark)
- **Regulatory clause extraction F1**: 0.82 (target: 0.92+)
- **128K-context document understanding**: Fails entirely — the fine-tuned model only supports 8K context, forcing chunking and lossy summarization of long documents like 300-page credit agreements

The executive team has decided to train **MeridianFin-72B**, a custom 72 billion parameter Mixture-of-Experts model with 128 experts and top-4 routing (~18B active parameters per token). The model will be pre-trained from scratch on 4 trillion tokens of curated financial text, then instruction-tuned for downstream tasks. Critically, the model must support **128K-token context windows** to process full-length regulatory filings without chunking.

Meridian has secured a 6-month lease on a 1,024-GPU NVIDIA H100 cluster through a partnership with CoreWeave. The cluster consists of 128 nodes, each containing 8 H100 GPUs connected via NVLink (900 GB/s bidirectional bandwidth). Nodes are interconnected via 8x NVIDIA ConnectX-7 InfiniBand adapters providing 400 Gb/s per adapter (3.2 Tb/s total per node).

The problem: The ML infrastructure team's initial attempt to train the model using naive Data Parallelism (FSDP) across all 1,024 GPUs achieved only **32% Model FLOPs Utilization (MFU)** and ran out of memory when the context window exceeded 4,096 tokens. At 32% MFU, the estimated training time is 47 days — well beyond the 30-day target that fits within the compute budget.

The infrastructure team needs to design an **optimal 5D parallelism configuration** — choosing the right combination of Data Parallelism, Tensor Parallelism, Pipeline Parallelism, Sequence Parallelism, and Expert Parallelism — to:

1. Fit the full model (72B total parameters, 128 experts) within the 80 GB per-GPU memory constraint
2. Support 128K-token context windows during training
3. Achieve at least 45% MFU to complete training within 30 days
4. Maximize training throughput while minimizing communication overhead

Why It Matters

- **Revenue impact**: Meridian's largest client, a USD 400B AUM asset manager, has set a 6-month evaluation window. If MeridianFin-72B does not demonstrate state-of-the-art financial document understanding by Q3, the client will switch to a competing vendor. The contract is worth USD 4.8M/year.

- **Competitive pressure**: Bloomberg has released BloombergGPT, and several well-funded startups (Writer, Cohere) are pursuing financial vertical LLMs. Meridian's window to establish a moat is narrowing.
- **Compute cost**: The 1,024-GPU H100 cluster costs approximately USD 3.2M/month. Every day of inefficient training wastes roughly USD 107K. The difference between 32% and 45% MFU translates to approximately USD 1.8M in saved compute costs.
- **Talent retention**: The ML infrastructure team has been struggling with the parallelism configuration for 3 weeks. If the problem is not solved soon, key engineers may lose confidence in the technical direction.

Constraints

Constraint	Specification
Compute budget	1,024 NVIDIA H100 GPUs (128 nodes x 8 GPUs/node) for 30 days
Memory per GPU	80 GB HBM3
Intra-node bandwidth	NVLink: 900 GB/s bidirectional
Inter-node bandwidth	InfiniBand: 400 Gb/s per adapter, 8 adapters per node
Target MFU	>= 45% (current: 32%)
Context window	128K tokens during training
Model architecture	72B total params, 128 experts, top-4 routing, 80 Transformer layers
Training data	4 trillion tokens of financial text (SEC filings, earnings calls, analyst reports, news)
Regulatory compliance	SOC 2 Type II, all training data must remain within US data centers
Team expertise	Strong in PyTorch and FSDP; limited experience with Megatron-LM or custom parallelism strategies

---

Section 2: Technical Problem Formulation

Problem Type: Distributed Systems Optimization for LLM Pre-Training

This is fundamentally an **optimization problem over a discrete configuration space**. The decision variables are the parallelism degrees along each of the five dimensions, and the objective is to maximize training throughput (tokens processed per second) subject to per-GPU memory constraints.

Why is this not a simple hyperparameter search? Because the five parallelism dimensions interact in non-trivial ways:

- Increasing Tensor Parallelism reduces per-GPU memory but adds intra-layer communication overhead
- Increasing Pipeline Parallelism reduces per-GPU layer count but introduces pipeline bubbles
- Increasing Data Parallelism improves throughput scaling but does not reduce per-GPU memory (unless combined with ZeRO)
- Expert Parallelism is only relevant for the MoE layers and introduces All-to-All communication patterns
- Sequence Parallelism shares resources with Tensor Parallelism and is not an independent axis

The right approach is to model the memory consumption and communication costs analytically, then search for the configuration that maximizes throughput within the memory budget. This is what real ML infrastructure teams at Meta, Google, and DeepSeek do.

Input Specification

The inputs to the optimization problem are:

Model Architecture Parameters: - Total parameters: $P_{\text{total}} = 72 \times 10^9$ - Number of Transformer layers: $L = 80$ - Hidden dimension: $d_{\text{model}} = 8192$ - Number of attention heads: $n_{\text{heads}} = 64$ - Head dimension: $d_{\text{head}} = 128$ - MLP intermediate dimension: $d_{\text{ffn}} = 28672$ (for non-expert layers) - Number of experts: $N_E = 128$ - Top-K routing: $K = 4$ - Expert MLP intermediate dimension: $d_{\text{expert}} = 4096$ - Vocabulary size: $V = 128{,}000$ - Maximum sequence length: $S = 128{,}000$

Hardware Parameters: - Total GPUs: $N_{\text{GPU}} = 1024$ - GPUs per node: $G_{\text{node}} = 8$ - Total nodes: $N_{\text{node}} = 128$ - Memory per GPU: $M_{\text{GPU}} = 80$ GB - NVLink bandwidth: $B_{\text{NV}} = 900$ GB/s - InfiniBand bandwidth per adapter: $B_{\text{IB}} = 50$ GB/s (400 Gb/s) - Number of IB adapters per node: $n_{\text{IB}} = 8$

Training Parameters: - Global batch size: B_{global} (to be determined) - Micro-batch size: b (to be determined) - Precision: BF16 mixed-precision training with FP32 optimizer states - Optimizer: AdamW

Output Specification

The output is a **parallelism configuration** — a tuple of five integers and associated scheduling decisions:

$$\mathcal{C} = (N_{\text{DP}}, N_{\text{TP}}, N_{\text{PP}}, N_{\text{SP}}, N_{\text{EP}})$$

Subject to:

$$N_{\text{DP}} \times N_{\text{TP}} \times N_{\text{PP}} \times N_{\text{EP}} = N_{\text{GPU}}$$

(Sequence Parallelism shares the TP group, so $N_{\text{SP}} = N_{\text{TP}}$.)

Along with:

- Pipeline schedule (1F1B or interleaved)
- Number of micro-batches per pipeline step: M
- ZeRO stage for DP (0, 1, 2, or 3)
- Activation checkpointing strategy (none, selective, full)
- Global batch size B_{global} and micro-batch size b

The output representation is a tuple rather than a single scalar because the parallelism dimensions are inherently multi-dimensional and interact with each other. A single "efficiency score" would hide the critical tradeoffs the engineer needs to understand.

Mathematical Foundation

Memory Model

The per-GPU memory consumption has four components. Understanding each one is essential for choosing the right parallelism configuration.

1. Model State Memory

For mixed-precision training with AdamW, each parameter requires: - 2 bytes for BF16 weights - 2 bytes for BF16 gradients - 4 bytes for FP32 master weights (AdamW maintains an FP32 copy) - 4 bytes for FP32 first moment (Adam m) - 4 bytes for FP32 second moment (Adam v)

Total: 16 bytes per parameter.

With parallelism, the model states are distributed:

$$M_{\text{model}} = \frac{P_{\text{total}} \times 16}{N_{\text{TP}} \times N_{\text{PP}} \times N_{\text{EP}} \times Z}$$

where Z is the ZeRO sharding factor: $Z = 1$ for ZeRO-0 (no sharding), $Z = N_{\text{DP}}$ for ZeRO-3. Intermediate stages have Z values between 1 and N_{DP} depending on what is sharded.

Let us build intuition for why ZeRO matters here. Without ZeRO, each GPU in the DP group holds a full copy of its assigned model shard. With 8-way TP, 16-way PP, and 8-way EP, each GPU holds $1/(8 \times 16 \times 8) = 1/1024$ of the parameters. For 72B parameters, that is 70.3M parameters per GPU, or about 1.1 GB of model states. This is comfortably small. But the picture changes dramatically when we consider activations.

2. Activation Memory

Activations are the intermediate tensors stored during the forward pass for use in the backward pass. For a Transformer layer with sequence length S and micro-batch size b:

$$M_{\text{act}} = S \times b \times d_{\text{model}} \times \alpha \times \frac{L_{\text{local}}}{N_{\text{SP}}} \times \beta$$

where: - $L_{\text{local}} = L/N_{\text{PP}}$ is the number of layers assigned to this GPU's pipeline stage - N_{SP} is the Sequence Parallelism degree (reduces activation memory along the sequence dimension) - α captures the number of intermediate activations per layer (attention scores, MLP intermediates, residual connections — typically $\alpha \approx 34$ bytes per element for a standard Transformer) - β accounts for the activation checkpointing strategy: $\beta = 1$ for no checkpointing, $\beta \approx 2/L_{\text{local}}$ for full checkpointing (only store inputs, recompute during backward)

This is where the memory problem becomes acute. With $S = 128{,}000$ tokens and $b = 1$:

$$M_{\text{act}} \approx 128{,}000 \times 1 \times 8192 \times 34 \times \frac{L_{\text{local}}}{N_{\text{SP}}} \approx 35.7 \text{ TB} \times \frac{L_{\text{local}}}{N_{\text{SP}} \times L}$$

Even with $L_{\text{local}} = 5$ layers (16-way PP) and $N_{\text{SP}} = 8$, the raw activation memory is about 27.9 GB per GPU — a significant fraction of the 80 GB budget. This is precisely why Sequence Parallelism is essential for long-context training.

Attention memory deserves special treatment. The self-attention mechanism stores the attention score matrix, which has size $O(S^2)$:

$$M_{\text{attn}} = \frac{n_{\text{heads}}}{N_{\text{TP}}} \times \frac{S}{N_{\text{SP}}} \times S \times 2 \text{ bytes}$$

Wait — why is only one S dimension divided by N_{SP}? Because standard Sequence Parallelism (Megatron-style) only shards the LayerNorm and Dropout activations along the sequence dimension. The attention scores themselves are not split by Megatron-SP. For truly splitting the attention computation across the sequence dimension, we need Ring Attention (Context Parallelism), where both dimensions can be split.

With Ring Attention at degree N_{CP}:

$$M_{\text{attn}} = \frac{n_{\text{heads}}}{N_{\text{TP}}} \times \frac{S}{N_{\text{CP}}} \times \frac{S}{N_{\text{CP}}} \times 2 \text{ bytes}$$

For $S = 128{,}000$, $n_{\text{heads}} = 64$, $N_{\text{TP}} = 8$, and $N_{\text{CP}} = 16$:

$$M_{\text{attn}} = 8 \times 8000 \times 8000 \times 2 = 1.02 \text{ GB}$$

That is manageable. Without Context Parallelism ($N_{\text{CP}} = 1$), the attention memory would be $8 \times 128{,}000 \times 128{,}000 \times 2 = 262$ GB — far exceeding a single GPU.

3. Communication Buffers

AllReduce, All-to-All, and point-to-point operations require temporary buffers. A reasonable estimate is:

$$M_{\text{comm}} \approx 2 \times \frac{P_{\text{shard}} \times 2}{N_{\text{TP}}}$$

where P_{shard} is the number of parameters in the largest TP-sharded layer. This is typically 1-3 GB for large models.

4. Total Memory Constraint

The total per-GPU memory must satisfy:

$$M_{\text{model}} + M_{\text{act}} + M_{\text{attn}} + M_{\text{comm}} + M_{\text{framework}} \leq M_{\text{GPU}}$$

where $M_{\text{framework}} \approx 2\text{-}4$ GB accounts for PyTorch framework overhead, CUDA context, and NCCL buffers.

Throughput Model

The training throughput in tokens per second is:

$$\text{Throughput} = \frac{B_{\text{global}} \times S}{T_{\text{step}}}$$

where T_{step} is the time for one training step, composed of:

$$T_{\text{step}} = T_{\text{compute}} + T_{\text{comm}} + T_{\text{bubble}}$$

Compute time depends on the FLOPs per token and the GPU's peak compute throughput:

$$T_{\text{compute}} = \frac{6 \times P_{\text{active}} \times B_{\text{global}} \times S}{\text{Peak FLOPS} \times N_{\text{GPU}} \times \text{MFU}}$$

The factor of 6 comes from the standard LLM training estimate: 2 FLOPs per parameter for the forward pass (one multiply, one add per parameter), and a factor of 3 for forward + backward (the backward pass costs approximately 2x the forward pass).

For MoE models, P_{active} is the number of active parameters per token:

$$P_{\text{active}} = P_{\text{non-expert}} + K \times P_{\text{per-expert}}$$

For MeridianFin-72B: $P_{\text{active}} \approx 18 \times 10^9$ (18B active out of 72B total).

Communication time is the sum of all communication operations:

$$T_{\text{comm}} = T_{\text{TP}} + T_{\text{PP}} + T_{\text{DP}} + T_{\text{EP}}$$

Each term depends on the message size divided by the effective bandwidth:

- T_{TP} : AllReduce within the TP group, over NVLink. Message size: $O(b \times S \times d_{\text{model}})$
- T_{PP} : Point-to-point activation transfer between pipeline stages, over InfiniBand. Message size: $O(b \times S \times d_{\text{model}})$
- T_{DP} : AllReduce of gradients across DP groups, over InfiniBand. Message size: $O(P_{\text{shard}} \times 2)$ (BF16 gradients)
- T_{EP} : All-to-All token dispatch, over InfiniBand. Message size: $O(b \times S \times d_{\text{model}} \times K/N_E)$ per expert

Pipeline bubble time:

$$T_{\text{bubble}} = \frac{N_{\text{PP}} - 1}{N_{\text{PP}} - 1 + M} \times (T_{\text{compute}} + T_{\text{comm, non-PP}})$$

where M is the number of micro-batches. The bubble fraction decreases as M increases, which is why large global batch sizes are preferred for pipeline-parallel training.

The Optimization Problem

Formally, the problem is:

$$\max_{\mathcal{C}} \quad \text{Throughput}(\mathcal{C})$$

subject to:

$$M_{\text{total}}(\mathcal{C}) \leq 80 \text{ GB per GPU}$$

$N_{\text{DP}} \times N_{\text{TP}} \times N_{\text{PP}} \times N_{\text{EP}} = 1024$

$$N_{\text{TP}} \leq G_{\text{node}} = 8$$

$N_{\text{TP}} \in \{1, 2, 4, 8\}$

$$N_{\text{PP}} \in \{1, 2, 4, 8, 16, 20, 40, 80\}$$

$N_{\text{EP}} \in \{1, 2, 4, 8, 16, 32, 64, 128\}$

$$L \mod N_{\text{PP}} = 0$$

$N_E \mod N_{\text{EP}} = 0$

The constraint $N_{\text{TP}} \leq 8$ reflects the physical reality that Tensor Parallelism must stay within a node (NVLink boundary). The divisibility constraints ensure layers and experts can be evenly distributed.

Loss Function: Pre-Training Objective

The model is pre-trained with the standard causal language modeling objective:

$$\mathcal{L}_{\text{LM}} = -\frac{1}{T} \sum_{t=1}^{T} \log P(x_t \mid x_1, \ldots, x_{t-1}; \theta)$$

For the MoE architecture, an auxiliary load-balancing loss is added:

$$\mathcal{L}_{\text{aux}} = \alpha_{\text{bal}} \times N_E \times \sum_{i=1}^{N_E} f_i \times p_i$$

where: - $f_i = \frac{\text{number of tokens routed to expert } i}{\text{total tokens}}$ is the fraction of tokens dispatched to expert i - $p_i = \frac{1}{T} \sum_{t=1}^{T} g_i(x_t)$ is the average gate probability for expert i - α_{bal} is a hyperparameter (typically 0.01)

Why the auxiliary loss? Without it, the router tends to collapse — sending all tokens to a small number of "popular" experts while the rest go unused. This wastes the capacity of the

unused experts and creates severe load imbalance in Expert Parallelism (some GPUs are overloaded while others are idle). The $f_i \times p_i$ product penalizes the situation where both the routing fraction and the gate probability are high for the same expert, encouraging uniform distribution.

What happens if α_{bal} is too large? The router is forced to distribute tokens uniformly regardless of content, losing the specialization benefit of MoE. If too small, load imbalance degrades training throughput. The optimal value balances model quality against training efficiency — a direct tradeoff between the ML objective and the systems objective.

The total loss is:

$$\mathcal{L} = \mathcal{L}_{\text{LM}} + \mathcal{L}_{\text{aux}}$$

Evaluation Metrics

Metric	Target	Rationale
Model FLOPs Utilization (MFU)	>= 45%	Primary efficiency metric; measures what fraction of theoretical peak FLOPs are used for useful computation
Tokens per second	>= 1.5M tokens/sec	Directly determines training time: 4T tokens / 1.5M tok/s = 31 days
Peak per-GPU memory	<= 76 GB	Must fit within 80 GB with 4 GB safety margin for framework overhead
Pipeline bubble fraction	<= 15%	Measures wasted compute in pipeline stages
Expert load imbalance	<= 10%	Ratio of max-to-mean tokens per expert; indicates routing quality
FinanceBench accuracy	>= 78%	Post-training evaluation on financial QA benchmark
128K context perplexity	<= 8.5	Validates long-context capability on held-out financial documents

Baseline: Naive FSDP (ZeRO Stage 3)

The baseline approach uses **FSDP (ZeRO Stage 3) across all 1,024 GPUs** with no Tensor, Pipeline, Sequence, or Expert Parallelism.

What this means: Every GPU participates in a single Data Parallel group. The model weights, gradients, and optimizer states are sharded across all 1,024 GPUs. Each GPU stores only $1/1024$ of the model states but must gather full weight tensors before each forward/backward computation and scatter gradients afterward.

Baseline performance: - MFU: 32% (measured) - Memory: OOM at sequence length > 4,096 tokens (activation memory exceeds budget) - Training time estimate: 47 days at 32% MFU (exceeds 30-day target) - Communication bottleneck: Frequent AllGather and ReduceScatter

operations across the entire 1,024-GPU cluster, most of which traverse the slower InfiniBand interconnect

Why the baseline fails: 1. **No Tensor Parallelism**: Individual MoE expert layers are large enough that their activations consume significant memory, but FSDP does not reduce activation memory — it only shards weights and optimizer states. 2. **No Pipeline Parallelism**: All 80 layers are "logically" on every GPU, requiring frequent full-model AllGather operations. 3. **No Sequence Parallelism**: 128K-token sequences produce quadratic attention memory that a single GPU cannot hold. 4. **No Expert Parallelism**: Each GPU must handle all 128 experts, requiring enormous activation memory and wasting capacity on experts that are not activated for the current token. 5. **Cross-node communication**: FSDP's AllGather spans the entire cluster, hitting the InfiniBand bottleneck for every single layer computation.

Why 5D Parallelism Is the Right Approach

The 5D parallelism framework is the principled solution because each dimension directly addresses a different bottleneck:

1. **Tensor Parallelism** (intra-node) reduces per-layer memory and computation, staying within the NVLink boundary where bandwidth is highest.
2. **Pipeline Parallelism** (inter-node) splits the 80-layer stack across multiple nodes, reducing the number of layers each GPU must handle.
3. **Data Parallelism** (cluster-wide) scales throughput by processing more data in parallel, with ZeRO sharding to reduce redundancy.
4. **Sequence Parallelism / Context Parallelism** addresses the 128K-token memory challenge by distributing activation memory and attention computation along the sequence dimension.
5. **Expert Parallelism** distributes the 128 experts across GPUs, so each GPU only hosts and computes a subset of experts.

No single parallelism strategy can solve all five bottlenecks simultaneously. The art is in finding the right combination.

Technical Constraints Summary

Component	Constraint	Impact on Configuration
NVLink boundary	TP must be within a node ($N_{\text{TP}} \leq 8$)	TP degree is 1, 2, 4, or 8
80 layers, divisibility	$80 \mod N_{\text{PP}} = 0$	PP degree is 1, 2, 4, 5, 8, 10, 16, 20, 40, or 80
128 experts, divisibility	$128 \mod N_{\text{EP}} = 0$	EP degree is 1, 2, 4, 8, 16, 32, 64, or 128
GPU count	$N_{\text{DP}} \times N_{\text{TP}} \times N_{\text{PP}} \times N_{\text{EP}} = 1024$	Configuration must multiply to 1024
Memory budget	80 GB per GPU (76 GB usable)	

Component	Constraint	Impact on Configuration
		Determines which configurations are feasible
Activation checkpointing	Reduces activation memory by ~60% but adds ~33% compute overhead	Tradeoff between memory and speed

---

Section 3: Implementation Notebook Structure

3.1 Data Acquisition Strategy

Dataset: We use two publicly available financial datasets:

1. **SEC-EDGAR Full-Text Filings** — A corpus of 10-K, 10-Q, 8-K, and proxy statement filings downloaded from the SEC's EDGAR system. These documents range from 5,000 to 300,000+ tokens and represent the core document type MeridianFin must process.

2. **FinanceBench** — A curated dataset of 150 financial question-answer pairs grounded in SEC filings, used for evaluating the model after training. Available at: https://huggingface.co/datasets/PatronusAI/financebench

For the implementation notebook, we work at a reduced scale: a single-node 8-GPU simulation. Students configure parallelism for a smaller (1.3B parameter, 8 experts) model, which captures all five parallelism dimensions while being trainable on accessible hardware.

Data loading and preprocessing: - Tokenize SEC filings with a SentencePiece tokenizer (BPE, 32K vocab) - Pack sequences to the maximum context length with document boundary markers - Shard the dataset across Data Parallel ranks using a deterministic sampler

TODO: Students implement a memory-efficient data pipeline with proper DP sharding and sequence packing.

3.2 Exploratory Data Analysis

Before diving into parallelism configuration, students must understand the computational profile of the training workload:

- **Memory profiling**: Measure the per-GPU memory consumption of model states, activations, and communication buffers for different parallelism configurations
- **Communication profiling**: Measure the time spent in AllReduce, AllGather, ReduceScatter, and All-to-All operations
- **Compute profiling**: Measure the actual FLOPs achieved vs. theoretical peak

Students will: - Profile memory usage for the baseline (pure FSDP) configuration - Identify which component (model states, activations, attention) is the memory bottleneck - Measure communication overhead as a fraction of total step time

TODO: Students write profiling code using PyTorch's memory profiler and CUDA event timing, then answer guided questions about the bottlenecks.

3.3 Baseline Approach: Pure Data Parallelism (FSDP)

The baseline uses PyTorch FSDP (Fully Sharded Data Parallel) across all available GPUs with no other parallelism strategies.

Students will: - Configure FSDP with ZeRO Stage 3 sharding - Run a short training loop (100 steps) on the small model - Measure MFU, memory usage, and throughput - Identify why this approach fails for long sequences and MoE models

TODO: Students implement the FSDP baseline, measure its performance, and write a short analysis of its limitations.

3.4 Model Design: Configuring 5D Parallelism

This is the core of the case study. Students will progressively add each parallelism dimension and measure the impact on memory and throughput.

3.4.1 Tensor Parallelism

Students implement column-wise and row-wise matrix splitting for a linear layer, then apply Megatron-style partitioning to the MLP block.

Key concepts: - Why column-wise for the first linear layer and row-wise for the second - Why GeLU between the two layers requires no communication - The cost of AllReduce within each TP group

TODO: Students implement `ColumnParallelLinear` and `RowParallelLinear` classes, then build a `TensorParallelMLP` module.

3.4.2 Pipeline Parallelism

Students implement a basic pipeline schedule with micro-batching.

Key concepts: - How to partition layers across pipeline stages - The 1F1B schedule and why it reduces peak memory vs. GPipe - The bubble fraction formula and how to minimize it

TODO: Students implement a `PipelineStage` class and a `OneFOneBSchedule` function that orchestrates forward and backward micro-batches.

3.4.3 Sequence Parallelism (Context Parallelism)

Students implement Ring Attention for distributing the attention computation across GPUs along the sequence dimension.

Key concepts: - Why standard Megatron-SP is insufficient for 128K contexts - How Ring Attention passes K, V blocks around a ring - The memory savings from distributing the attention score matrix

TODO: Students implement a simplified `RingAttention` module that distributes Q, K, V across a ring of GPUs and computes the correct output.

3.4.4 Expert Parallelism

Students implement All-to-All token dispatch for a Mixture-of-Experts layer.

Key concepts: - How the router assigns tokens to experts - Why All-to-All communication is needed to move tokens between GPUs - The load-balancing loss and its impact on training dynamics

TODO: Students implement an `ExpertParallelMoE` layer with a router, All-to-All dispatch, and load-balancing loss.

3.4.5 Combining All Five Dimensions

Students build a configuration optimizer that takes the model architecture and hardware specifications as input and searches for the optimal parallelism configuration.

Key concepts: - The multiplicative constraint: $N_{\text{DP}} \times N_{\text{TP}} \times N_{\text{PP}} \times N_{\text{EP}} = N_{\text{GPU}}$ - The communication hierarchy: TP within nodes, PP across nearby nodes, DP across the cluster - Tradeoff between pipeline bubbles and memory savings

TODO: Students implement a `find_optimal_config` function that enumerates valid configurations, estimates memory and throughput for each, and returns the best one.

3.5 Training Strategy

Optimizer: AdamW with the following schedule: - Learning rate: 3×10^{-4} with linear warmup over 2,000 steps and cosine decay to 3×10^{-5} - Weight decay: 0.1 - Adam betas: $\beta_1 = 0.9$, $\beta_2 = 0.95$ - Gradient clipping: max norm 1.0

Why AdamW over SGD? For large Transformer models, the adaptive learning rates in Adam are critical. Different parameters (attention weights, MLP weights, embeddings, router weights) have vastly different gradient magnitudes and curvatures. SGD with a single learning rate would either under-train some components or destabilize others. The decoupled weight decay in AdamW provides regularization without interfering with the adaptive learning rates.

Why cosine schedule? The cosine schedule provides a smooth decay that avoids the sharp drops of step-wise schedules. For long training runs (4T tokens), this gives the optimizer a chance to explore the loss landscape during the warm-up phase and then gradually converge.

The minimum learning rate (3×10^{-5} , 10x lower than peak) prevents the model from "going to sleep" in the final phase.

Training loop structure: 1. Sample micro-batches from the data pipeline 2. Execute the pipeline schedule (1F1B) across pipeline stages 3. Within each micro-batch: forward pass with TP and EP, backward pass with activation checkpointing 4. After all micro-batches: gradient synchronization via AllReduce across DP group 5. Optimizer step with gradient clipping

TODO: Students implement the full distributed training loop integrating all five parallelism dimensions, with proper gradient synchronization and mixed-precision handling.

3.6 Evaluation

Quantitative evaluation: - Measure MFU over the last 50 training steps (should be >= 45%) - Measure peak per-GPU memory usage (should be <= 76 GB) - Measure pipeline bubble fraction (should be <= 15%) - Measure expert load imbalance (max/mean tokens per expert, should be <= 1.10) - Record wall-clock time per training step and extrapolate to full training time

Comparison against baseline: - Create a table comparing baseline (FSDP-only) vs. optimized 5D configuration on all metrics - Visualize the throughput improvement as a bar chart - Plot memory usage breakdown (model states, activations, attention, communication) for both configurations

TODO: Students generate evaluation plots, fill in the comparison table, and write a short analysis explaining why the 5D configuration outperforms the baseline.

3.7 Error Analysis

Common failure modes in distributed training:

1. **Memory OOM**: A configuration that fits in the memory model but fails in practice due to fragmentation, communication buffer spikes, or activation checkpointing overhead being underestimated.
2. **Gradient explosion with Pipeline Parallelism**: When pipeline stages have different gradient magnitudes, gradient clipping applied globally may not prevent local instabilities.
3. **Expert collapse**: The router converges to using only a few experts despite the load-balancing loss, leading to wasted parameters and load imbalance.
4. **Communication deadlock**: Incorrect ordering of AllReduce, All-to-All, and point-to-point operations can cause hangs.
5. **Numerical divergence from mixed-precision**: BF16 has limited dynamic range; certain parameter combinations can cause loss spikes.

TODO: Students inject controlled failures (disable load-balancing loss, set TP degree higher than intra-node GPUs, use insufficient micro-batches for PP) and analyze the resulting errors. They must identify and categorize the top 3 failure modes they observe.

3.8 Scalability and Deployment Considerations

Scaling analysis: - Plot throughput vs. number of GPUs for the optimal configuration (128, 256, 512, 1024 GPUs) - Identify the scaling efficiency (linear scaling = 100%) - Determine the main bottleneck to further scaling

Inference deployment: - During inference, the model only needs weights (no gradients, no optimizer states), and Tensor Parallelism is the primary strategy - Discuss how the 128 experts are distributed at inference time - Profile inference latency for different TP degrees (1, 2, 4, 8)

TODO: Students write a scaling analysis script that simulates throughput at different GPU counts using the memory/throughput model, and an inference benchmarking script that measures latency for different TP configurations.

3.9 Ethical and Regulatory Analysis

Bias considerations specific to financial AI: - **Temporal bias**: Training data skews toward recent filings; the model may not understand historical financial instruments or regulations - **Sector bias**: If training data overrepresents certain sectors (e.g., tech), the model may perform poorly on underrepresented sectors (e.g., utilities, mining) - **Size bias**: Large-cap company filings are more plentiful and higher quality; the model may underperform on small-cap or micro-cap filings - **Linguistic bias**: SEC filings follow US English conventions; the model may not handle international financial documents well

Regulatory compliance: - SOC 2 Type II: All training data and model weights must remain within certified US data centers - SEC regulations: The model's outputs cannot be construed as investment advice without appropriate disclaimers - GDPR: If the model is deployed for European clients, personal information in filings must be handled according to GDPR requirements - Model risk management (SR 11-7): Bank clients must validate model outputs independently

TODO: Students write a 500-word ethical impact assessment covering data bias, fairness across market sectors, and regulatory compliance requirements specific to financial AI.

Section 4: Production and System Design Extension

Architecture Overview

The production system for MeridianFin-72B consists of five major components:

```
[Client Apps: DocAnalyst, ComplianceGuard, SignalStream]
                    |
```

```
            [API Gateway (Kong)]
                    |
        [Load Balancer (NVIDIA Triton)]
                    |
    +-------+-------+-------+-------+
    |       |       |       |       |
[vLLM   [vLLM   [vLLM   [vLLM   [vLLM
 Pod 0]  Pod 1]  Pod 2]  Pod 3]  Pod N]
    |       |       |       |       |
    +-------+-------+-------+-------+
                    |
        [Model Registry (MLflow)]
                    |
    [Object Storage (S3) — checkpoints, datasets]
```

Each vLLM pod runs on a node with 8 H100 GPUs using 8-way Tensor Parallelism. The 128 MoE experts are distributed across 16 pods (8 experts per pod) using Expert Parallelism at inference time.

API Design

Primary endpoint: Document Analysis

```
POST /v1/analyze
Content-Type: application/json

Request:
{
    "document": "<base64 encoded document or text>",
    "document_type": "10-K" | "10-Q" | "8-K" | "credit_agreement" | "other",
    "tasks": ["summarize", "extract_financials", "compliance_check", "qa"],
    "questions": ["What is the total revenue?", ...],  // for qa task
    "max_tokens": 4096,
    "temperature": 0.1,
    "stream": true
}

Response (streaming):
{
    "id": "req_abc123",
    "status": "completed",
    "results": {
        "summary": "...",
        "financials": {...},
        "compliance_flags": [...],
        "answers": [...]
    },
    "usage": {
        "input_tokens": 85432,
        "output_tokens": 2048,
        "latency_ms": 12400,
        "cost_usd": 0.42
    }
}
```

Health and monitoring endpoints:

```
GET /v1/health          — Cluster health status
GET /v1/metrics         — Prometheus-compatible metrics
POST /v1/batch/submit   — Batch processing for large document sets
GET /v1/batch/{id}      — Batch job status
```

Serving Infrastructure

Framework: vLLM with PagedAttention for efficient KV-cache management. vLLM supports Tensor Parallelism out of the box and has emerging support for Expert Parallelism.

Scaling strategy: - **Horizontal scaling**: Add more vLLM pods to handle increased request volume. Each pod handles one request at a time for maximum throughput per request. - **Vertical scaling**: For documents exceeding 128K tokens, route to dedicated "long-context" pods with 16-way Context Parallelism across 2 nodes. - **Autoscaling**: Kubernetes HPA based on GPU utilization and request queue depth. Scale up when average GPU utilization exceeds 70% or queue depth exceeds 10. Scale down when utilization drops below 30% for 10 minutes.

Expert Parallelism at inference: The 128 experts total approximately 54B parameters (out of 72B). Each inference node (8 GPUs) can host 8 experts. A full expert-parallel deployment requires 16 nodes (128 GPUs). For cost efficiency, Meridian uses expert offloading: only the top-16 most frequently activated experts are kept in GPU memory, and the rest are loaded on-demand from CPU memory with prefetching based on router predictions.

Latency Budget

For a typical 50K-token input document with 2K-token output:

Component	Budget (ms)	Notes
API Gateway + Auth	50	Token validation, rate limiting
Document preprocessing	200	Tokenization, chunking if needed
Prefill (input processing)	3,500	50K tokens at ~70 tokens/ms per pod
Decode (output generation)	5,000	2K tokens at ~2.5 tokens/step
Expert routing overhead	500	All-to-All for MoE dispatch
Post-processing	200	JSON formatting, compliance filtering
Network overhead	150	Client-server round trips
Total	**~9,600**	Target: < 15,000 ms

For real-time compliance monitoring (ComplianceGuard), the latency budget is tighter: < 2,000 ms for short messages (< 1K tokens), which is achievable because the input is much shorter.

Monitoring

Model-level metrics (Prometheus + Grafana): - Tokens per second (throughput) - Time to First Token (TTFT) — p50, p95, p99 - Inter-Token Latency (ITL) — p50, p95, p99 - GPU memory utilization per device - GPU compute utilization per device - KV-cache hit rate (for repeated/similar documents) - Expert utilization distribution (are all experts being used at inference time?)

Business-level metrics: - Requests per minute by client and task type - Error rate by error category (OOM, timeout, invalid input) - Cost per request - FinanceBench accuracy on weekly eval runs (regression detection)

Alerting thresholds: | Alert | Condition | Severity | |---|---|---| | High latency | p95 TTFT > 5s for 5 min | Warning | | Very high latency | p95 TTFT > 10s for 5 min | Critical | | GPU OOM | Any GPU OOM event | Critical | | Expert collapse | Top-1 expert handles > 25% of tokens | Warning | | Model quality regression | FinanceBench accuracy drops > 2% week-over-week | Critical | | Cluster health | > 2 nodes unresponsive | Critical |

Model Drift Detection

Distribution shift sources in financial AI: - New regulatory frameworks (e.g., Basel IV implementation changes filing structures) - Market regime changes (e.g., high-volatility periods produce different language patterns) - New financial instruments (e.g., crypto-related filings, SPACs) - Evolving corporate disclosure practices

Detection strategy: 1. **Input drift**: Monitor token distribution entropy on incoming documents. If entropy shifts by > 2 standard deviations from the training distribution baseline over a 7-day rolling window, flag for review. 2. **Output drift**: Track the distribution of compliance flags, extraction confidence scores, and summary lengths. Sudden shifts indicate potential model degradation. 3. **Performance drift**: Run FinanceBench evaluation weekly. If accuracy drops > 1% from baseline, trigger a detailed investigation. 4. **Expert routing drift**: Monitor the expert utilization distribution at inference time. If it diverges significantly from the training distribution, the model may be encountering out-of-distribution inputs.

Response protocol: - Minor drift (< 1% accuracy drop): Continue monitoring, increase eval frequency - Moderate drift (1-3% accuracy drop): Trigger fine-tuning on recent data (last 90 days of filings) - Severe drift (> 3% accuracy drop): Pause affected product features, initiate full retraining evaluation

Model Versioning

Versioning scheme: `meridianfin-{param_count}-v{major}.{minor}.{patch}` - Example: `meridianfin-72b-v1.2.0` - Major: Architecture changes or full retraining - Minor: Fine-tuning or significant data updates - Patch: Bug fixes, prompt template changes

Storage: All checkpoints stored in S3 with MLflow tracking. Each checkpoint includes: - Model weights (sharded by the original training parallelism config) - Optimizer states (for continued training) - Training config (parallelism degrees, hyperparameters) - Data provenance (which data shards were used, their hashes) - Evaluation results on the standard benchmark suite

Rollback strategy: Blue-green deployment. The previous model version always remains loaded on a standby set of pods. Rollback is a DNS switch (< 60 seconds to take effect). Rollback is triggered automatically if the new model's p95 latency exceeds 2x the previous model's within the first hour of deployment.

A/B Testing

Framework: Custom A/B testing built on top of the API gateway. Requests are routed to model A or model B based on a deterministic hash of the client ID.

Statistical requirements: - Minimum sample size: 1,000 requests per variant per task type - Significance level: $\alpha = 0.05$ - Minimum detectable effect: 2% absolute improvement in primary metric - Test duration: minimum 7 days (to capture weekly patterns in filing volumes)

Guardrail metrics: Metrics that must not regress during an A/B test: - p99 latency must not increase by more than 20% - Error rate must not increase by more than 0.5 percentage points - Compliance flag false-positive rate must not increase by more than 1 percentage point

If any guardrail is violated, the test is automatically stopped and traffic is routed 100% to the control model.

CI/CD for ML

Training pipeline (Airflow-orchestrated): 1. **Data preparation**: Download new filings from EDGAR, deduplicate, tokenize, pack sequences 2. **Data validation**: Check for PII, verify token distributions, ensure no data leakage between train/eval splits 3. **Training**: Launch distributed training with the optimized 5D parallelism config 4. **Checkpoint selection**: Choose the checkpoint with lowest eval loss (evaluated every 1,000 steps) 5. **Evaluation gate**: Run the full benchmark suite (FinanceBench, internal compliance test, latency test). All metrics must meet thresholds defined in the model card. 6. **Model registration**: Register the validated checkpoint in MLflow with all metadata 7. **Staging deployment**: Deploy to the staging cluster for manual review by the ML team 8. **Production deployment**: Blue-green deployment with automatic rollback

Validation gates (must all pass): | Gate | Criteria | |---|---| | FinanceBench accuracy | >= 78% | | Compliance extraction F1 | >= 0.90 | | 128K context perplexity | <= 8.5 | | p95 TTFT (50K input) | <= 5,000 ms | | No expert collapse | Max expert load <= 2x mean |

Cost Analysis

Training costs (one full pre-training run):

Item	Cost
1,024 H100 GPUs x 30 days (CoreWeave spot)	USD 2.4M
Storage (4T tokens tokenized + checkpoints)	USD 12K
Data acquisition and preprocessing	USD 35K
Engineering time (2 ML infra engineers x 2 months)	USD 120K
Total training cost	**~USD 2.57M**

Inference costs (monthly, at steady state):

Item	Cost/month
128 H100 GPUs (16 nodes) for serving	USD 384K
API gateway and load balancing	USD 5K
Monitoring and logging infrastructure	USD 8K
Storage (model checkpoints, logs)	USD 3K
Total inference cost	**~USD 400K/month**

Cost per request: At an estimated 2M requests/month, the cost per request is approximately USD 0.20. For a typical 50K-token document analysis, this is competitive with API-based alternatives (USD 0.30-0.50 for equivalent quality with GPT-4).

Break-even analysis: With USD 28M ARR and USD 4.8M/year in compute costs (training amortized over 12 months + inference), the gross margin on the AI infrastructure is approximately 83%. The training cost pays for itself if it retains even one major client.