# Parallel Distributed ML Implementation—

# **Course:** ML System Optimization

**Assignment:** Distributed Mini-Batch Neural Network Training with PyTorch and Horovod (MNIST MLP)

**Author:**  **RAHUL AGARWAL[2024aa05676][100%]**

**KRITHIKA M[2024aa05421] [100%]**

**SAURAV BANSAL[2023aa05710] [100%]**

**PAYEL KARMAKAR[2024aa05423] [100%]**

**YARRAGONDLA RUGMANGADHAREDDY[2024aa05435] [100%]**

## Deliverables

* This design document (exportable to Word/PDF)
* Companion Jupyter Notebook containing:
  + Runnable PyTorch torch.distributed (DDP) code for distributed MLP training on MNIST
  + Single-GPU simulation of multiple processes
  + Output logs, loss/accuracy plots
  + Quick experiments exploring batch size, throughput, and mixed precision trade-offs

## Table of Contents

1. Design Overview
2. System Architecture Diagram
3. Parallelization Strategy
4. Development Environment
5. Execution Platform & Implementation
6. Initial Challenges Identified
7. Assumptions
8. Experiments, Profiling & Expected Results
9. Security, Reproducibility & Submission Checklist

# 1. Design Overview

### 1.1 Proposed Approach

* **Primary strategy:** **Data Parallelism** using **PyTorch** **DistributedDataParallel (DDP)** in single-GPU simulation mode.
* Each worker process holds a replica of the MLP model and receives disjoint mini-batches from a DistributedSampler. After computing gradients locally, processes synchronize parameters using **gradient all-reduce**.
* **Horovod** is **not** part of the runnable Colab code but is included in the design document for conceptual completeness and for portability to multi-GPU environments.
* **Scaling:** Simulated multi-process DDP with torch.multiprocessing.spawn on Colab T4 (single GPU)
* **Profiling**: Per-epoch timing, per-batch timing, estimate of communication cost (timed all-reduce windows), GPU memory snapshots.

### 1.2 Justification

* MNIST’s MLP model is small enough to fit on a single GPU with ease.
* Data parallelism improves throughput by splitting data across processes/workers, even when simulated on one GPU.
* Task parallelism and hybrid strategies (model+data parallelism) are unnecessary for MNIST due to the small model size and limited complexity.
* Data Parallelism aligns with homogeneous training workloads where identical model copies can process disjoint portions of the dataset in parallel. MNIST classification with an MLP fits this perfectly: the model is small, data plentiful, and the computational pattern is regular.
* DDP is preferred due to PyTorch-native optimizations: gradient bucketing and overlapping compute and communication when possible, mature NCCL backend for GPU collectives, and simpler integration with torch.utils.data.DistributedSampler.

### 1.3 Alternatives Considered

* **Task Parallelism:** Not chosen because MNIST is a single-task classification problem and splitting the model into tasks (e.g., layer-wise parallelism) adds complexity without clear benefit.
* **Model Parallelism / Hybrid:** Not necessary for a small MLP. Useful for very large models (transformers) that exceed single-GPU memory.

# 2. System Architecture Diagram

### 2.1 Architecture Components

1. **Data Ingestion & Preprocessing**
   * Fetch MNIST via torchvision.datasets.MNIST.
   * DistributedSampler shards dataset per worker.
2. **Worker Process (Per GPU / Simulated Worker)**
   * Model instantiation.
   * DDP wrapping.
   * Local DataLoader using DistributedSampler.
   * Training loop (forward -> loss -> backward -> optimizer.step()).
3. **Communication Backend**
   * **NCCL (NVIDIA Collective Communications library)** used by DDP for GPU collectives.
   * For Horovod, **MPI / Gloo / NCCL** options exist depending on build.

4. **Execution Pathways**

* + **CPU:** Data loading, preprocessing, process management, IPC for gradient synchronization
  + **GPU:** Forward & backward pass, gradient synchronization via NCCL, parameter updates

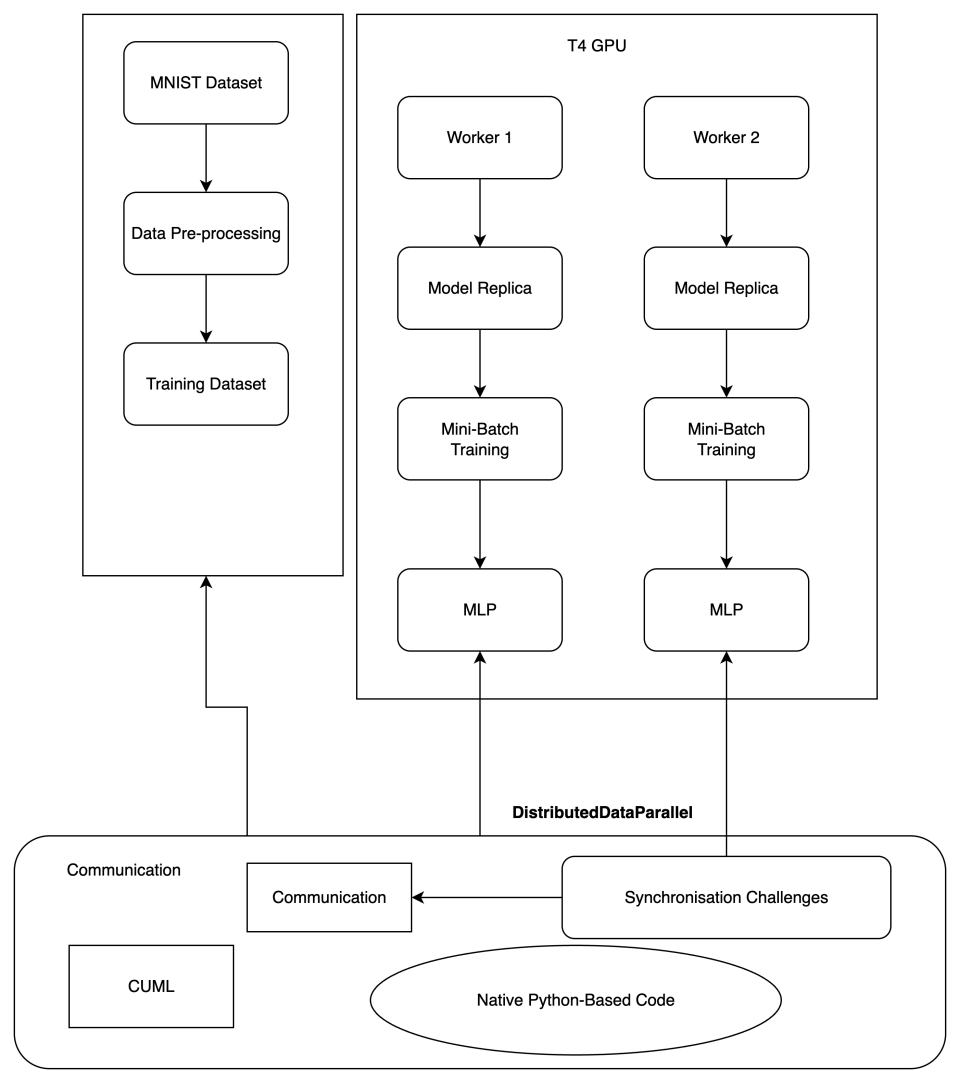
### 2.2 Workflow

[Dataset: MNIST] → [Transform: Normalize, ToTensor] → [DistributedSampler + DataLoader]

  ↳ Worker 0 (Process 0) — Model replica — Forward → Loss → Backward → Gradient AllReduce → Optimizer Step

  ↳ Worker 1 (Process 1) — Model replica — Forward → Loss → Backward → Gradient AllReduce → Optimizer Step

Evaluation → Metrics logging (rank 0 only) → Model checkpoints → Visualization of loss/accuracy & timing



# 3. Parallelization Strategy

### 3.1 Data vs Task Parallelism

* **Distribution method:** Data Parallelism. Each worker maintains a replica of the model and processes different minibatches.
* **Why not Task Parallelism:** Task parallelism would partition computation graph across devices (model parallelism). For a compact MLP on MNIST, overheads outweigh benefits.

### 3.2 Distribution Type

* **Data Parallelism** — DistributedSampler ensures non-overlapping mini-batches between processes
* **DDP with NCCL backend** — AllReduce used for gradient synchronization

### 3.2 Mini-batch Handling & Parameter Synchronization

* **Mini-batch flow:** DistributedSampler ensures each worker receives unique indices per epoch. Batch size is specified per worker. Effective global batch size = batch\_size\_per\_worker \* world\_size.
* **Gradient sync:** PyTorch DDP performs an all-reduce over gradients after loss.backward() and before optimizer.step(). DDP implements gradient bucketing (grouping gradients into large buckets) to reduce communication overhead.
* **Overlap compute and communication:** DDP can start all-reduce for earlier buckets while later buckets are still computing gradients, enabling overlap and improved utilization.
* All processes perform forward, backward, and optimizer step in sync
* Rank 0 process handles logging and checkpoint saving

### 3.3 Horovod Note

* While **Horovod** can replace DDP in multi-node/multi-GPU setups, it is not used in the Colab notebook due to installation and environment constraints. The code can be adapted to Horovod with minor changes.

### cuML vs scikit-learn (note for RF tasks)

We did not use cuML in our assignment; however, it is important to highlight the difference between the two approaches

* **cuML**: GPU-accelerated implementations; parallelizes tree construction using CUDA kernels; suitable for large datasets and low-latency inference in GPU-heavy pipelines.
* **scikit-learn**: CPU-based, uses multi-threading to parallelize over trees (n\_jobs). GPU advantage is large when dataset and model complexity justify data transfer and kernel launch overhead.

# 4. Development Environment

**Table: Software & Libraries**

|  |  |
| --- | --- |
| **Item** | **Detail** |
| Programming Language | Python 3.10 (Colab default) |
| Framework | PyTorch ≥ 2.0.1 |
| Distributed API | torch.distributed DDP |
| CUDA Toolkit | CUDA 11.x (Colab default for T4 GPU) |
| cuDNN | Provided by Colab |
| Dataset | MNIST via torchvision.datasets |
| Visualization | Matplotlib |
| Platform | Google Colab (T4 GPU) |

| **Data Handling & Preprocessing** |  |
| --- | --- |
| Dataset | MNIST (torchvision) |
| Transform | ToTensor() combined with normalization |
| DataLoader | DistributedSampler used for training; standard DataLoader for evaluation |

**Assumptions**

* The grader will run the notebook on Colab T4 or similar with GPU enabled.
* If multi-GPU is not available, DDP will be simulated using multiple processes on the same GPU (note: real scaling performance requires multiple GPUs).

# 5. Execution Platform & Implementation

### 5.1 Hardware & Platform

* **Execution Environment:** Google Colab (free tier)
* **GPU:** NVIDIA Tesla T4
  + 2560 CUDA cores
  + 16 GB GDDR6 VRAM (≈ 15.9 GB usable in Colab)
* **CPU:** Colab virtual CPUs (2–8 vCPUs allocated dynamically)
* **RAM:** 12–25 GB (runtime dependent)
* **Disk:** Ephemeral VM storage (not persistent)

The Colab runtime was configured to **GPU** mode via *Runtime → Change runtime type → GPU*. Our implementation is single-GPU but **simulates multiple workers** using PyTorch DistributedDataParallel (torch.distributed + torch.multiprocessing.spawn).

### 5.2 CUDA/CuDNN, NCCL compatibility.

* **CUDA Toolkit:** 11.x (default for Colab’s T4 runtime)
* **cuDNN:** Provided by Colab, matching installed CUDA version
* CUDA compatibility verified via:

torch.version.cuda

torch.backends.cudnn.version()

torch.cuda.is\_available()

* **cuDNN enabled** for GPU-accelerated kernels during training

### 5.3 NCCL compatibility

* **Backend:** NCCL (NVIDIA Collective Communications Library)
* NCCL handles **gradient synchronization** across processes in DDP
* On a single GPU, NCCL still runs but actual data transfer is intra-device
* init\_process\_group(backend='nccl') ensures optimal performance for multi-GPU setups, making the code portable beyond Colab
* Communication time is still measured in our code, though minimal on single GPU

### 5.4 Execution strategy & hyperparameters

* **Parallelism Type:** Data Parallelism using DistributedSampler
* **Processes per Node:** 2–4 simulated workers (adjustable)
* **Batch Size:**
  + Per-process: 64
  + Global = Per-process × World Size
* **Model:** MLP with 3 fully connected hidden layers, ReLU activations, dropout
* **Optimizer:** SGD with momentum
* **Learning Rate:** 0.01 (StepLR scheduler optional)
* **Epochs:** 5–10 for normal runs; adjustable for extended training
* **Metrics per Epoch:**
  + Train/value loss & accuracy
  + Epoch wall-clock time
  + Compute time vs communication time
  + GPU memory allocated & peak memory usage

### 5.4 Implementation notes (code & outputs)

* **Environment Setup:**
  + torch, torchvision, matplotlib
  + pip freeze output captured in notebook

A close up of a date

AI-generated content may be incorrect.

* **Data Loading**:
  + MNIST dataset from torchvision.datasets
  + Preprocessing: transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))
  + DistributedSampler ensures each process gets unique mini-batches
* **Training Loop:**
  + Rank 0 handles printing/logging/checkpointing
  + Time measurements around forward, backward, and sync steps

**A graph of a graph and a graph of a graph

AI-generated content may be incorrect.**

Epoch Time vs Batch Size Without AMP

**A graph of two people

AI-generated content may be incorrect.**

Epoch Time vs Batch Size With AMP

* **Outputs:**
  + Per-epoch logs (loss, accuracy, timings) +Matplotlib plots for training/validation loss and accuracy
    - Single Process Training results:

A close-up of a graph

AI-generated content may be incorrect.

Single Process Training Results

* + - DDP Training Results:

A close-up of a graph

AI-generated content may be incorrect.

DDP Training Result

* + - Single vs DDP results:

A graph of a line

AI-generated content may be incorrect.

Single vs DDP results

* + Communication vs computation timing table

On single GPU, NCCL comm overhead is minimal

* + GPU memory stats table

**A graph with green lines

AI-generated content may be incorrect.**

Single Process Max GPU Memory

**A graph with a green line

AI-generated content may be incorrect.**

DDP Max GPU Memory

* **Quick Experiments:**
  + Effect of batch size on throughput

**A graph with a line and a dotted line

AI-generated content may be incorrect.**

**Batch size and Epoch Time are inversely proportional**

* + Scaling with simulated worker count

With the Colab environment, this simulation wasn’t available

* + Optional mixed precision training impact

**A graph of a graph and a graph of a graph

AI-generated content may be incorrect.**

Epoch Time vs Batch Size Without AMP

**A graph of two people

AI-generated content may be incorrect.**

Epoch Time vs Batch Size With AMP

# 6. Initial Challenges Identified

* **Colab limitation:** Only 1 GPU available; multi-process DDP runs are simulated, not physically parallel across GPUs
* **Horovod installation:** Complex in Colab, omitted in runnable code
* **Data loading bottlenecks:** Mitigated with num\_workers > 0 and pin\_memory=True
* **Communication time measurement:** On single GPU, NCCL comm overhead is minimal

# 7. Assumptions

* All runs will be on Colab T4 GPU
* MNIST dataset is preprocessed with normalization to mean=0.1307, std=0.3081
* Model fits fully in GPU memory
* Notebook export will include code, plots, and logs in PDF form