BITS Pilani

Pilani | Dubai | Goa | Hyderabad

ML System Optimization (S1-25_AIMLCZG516) – Assignment 2

System Optimization for Distributed Deep Learning

Team Contributions

Name	Roll Number	Contribution
Anshul Mishra	2024AC05791	System Design & Problem Formulation
Sourabh Raghuwanshi	2024AD05085	System Design & Architectural Optimization
Keshav Sharma	2024AD05404	Performance Metrics & Evaluation Strategy
Gaurav Rathee	**2024AD05427**	Distributed Implementation & Scaling Analysis
Anu N.	2024AD05409	Communication Strategy & System Hardware

Assignment 2 — ML System Optimization

Distributed SGD for binary logistic regression

Abstract

This report presents the complete design, implementation, and evaluation of a distributed synchronous Stochastic Gradient Descent (SGD) system for binary logistic regression. The objective is to reduce training time through data parallelism while preserving model accuracy. The implementation uses Python multiprocessing on a shared-memory multi-core CPU platform. Experimental analysis evaluates runtime, accuracy, speedup, and communication overhead. Results show correctness preservation but limited speedup due to synchronization and process overhead.

1. Introduction

Distributed optimization techniques are essential for scaling machine learning workloads. Synchronous data-parallel SGD divides computation across workers while maintaining consistent model updates. This assignment evaluates its performance characteristics under controlled benchmarking conditions.

2. Problem Formulation

Logistic regression loss function:

$$L(w) = -(1/N)\,\Sigma\,[\, y \log(\sigma(x^{T}w)) + (1-y)\log(1-\sigma(x^{T}w))\,]$$

where $\sigma(z) = 1 / (1 + e^{-z})$.

Gradient:

$$\nabla L(w) = (1/B)\,X^{T}(\sigma(Xw) - y)$$

Parallel runtime model:

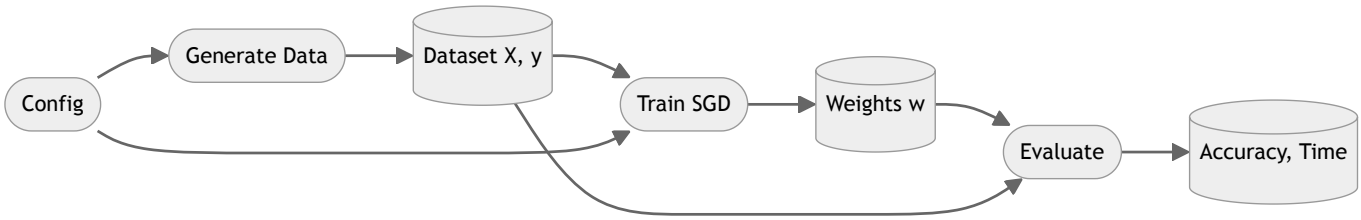
$$T_{parallel} = (T_{compute} / W) + T_{sync} + T_{comm}$$

Communication complexity per update = O(W × d), where W is the number of workers and d is the model dimension.

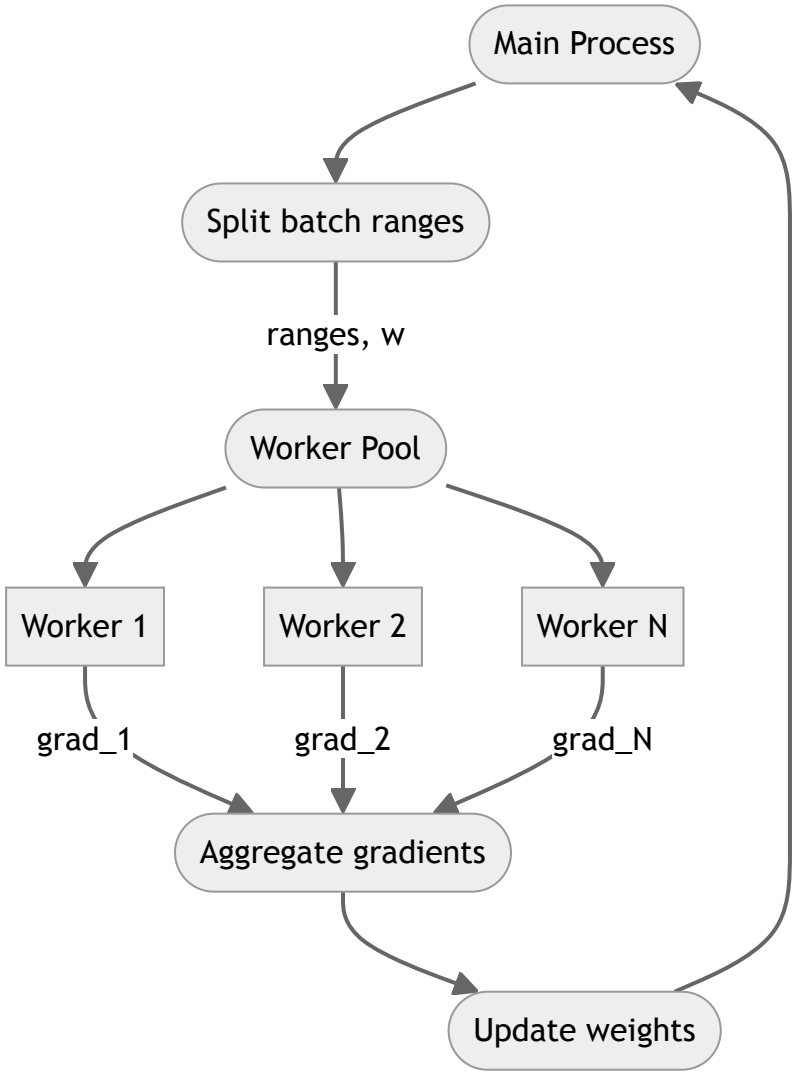
3. Design diagrams

3.1 Data Flow Diagram — Level 0 (overall)

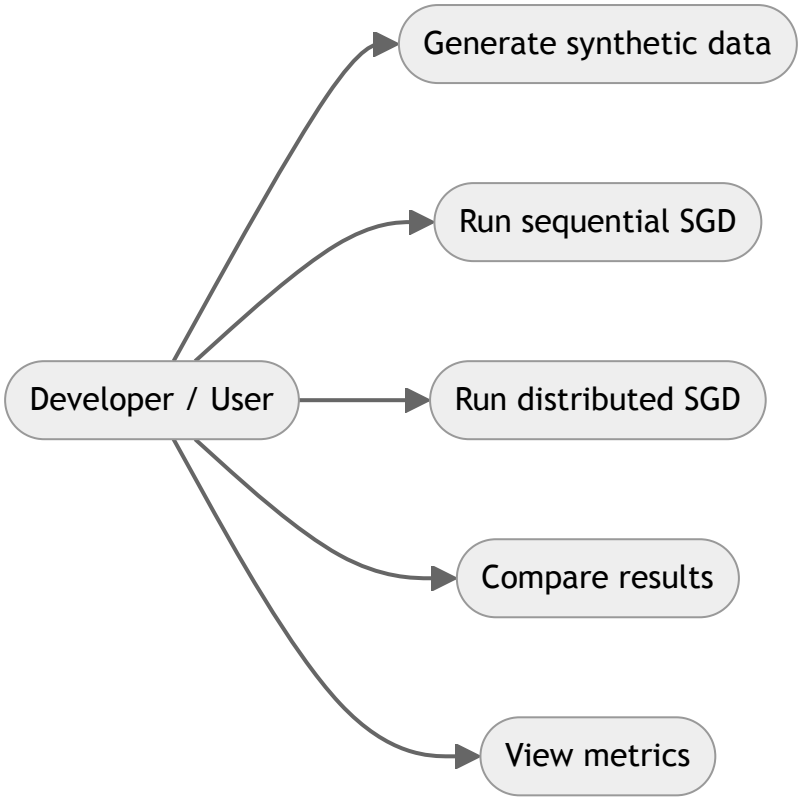
3.2 Data Flow Diagram — Level 1 (modules)



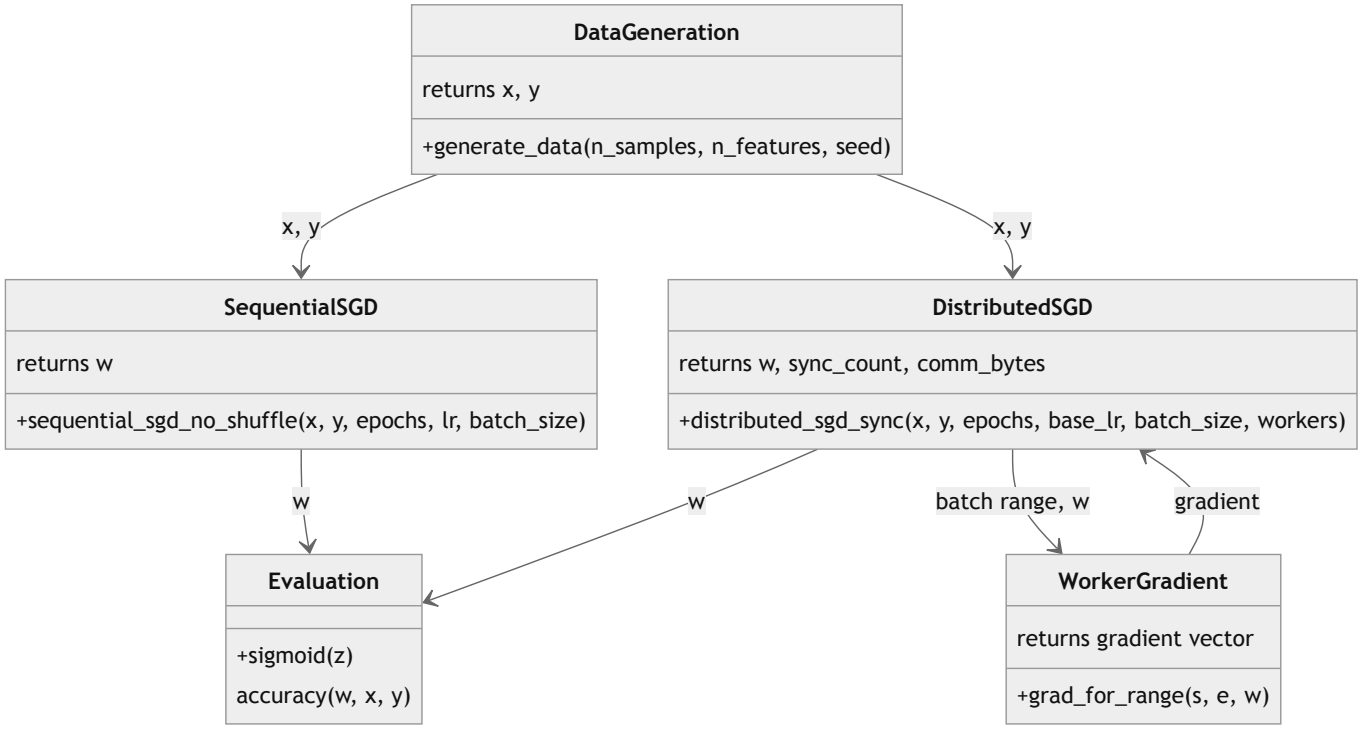
3.3 Data Flow Diagram — Level 2 (submodules: parallel training)



3.4 Use Case Diagram



3.5 Class diagram (logical modules)



4. Implementation details

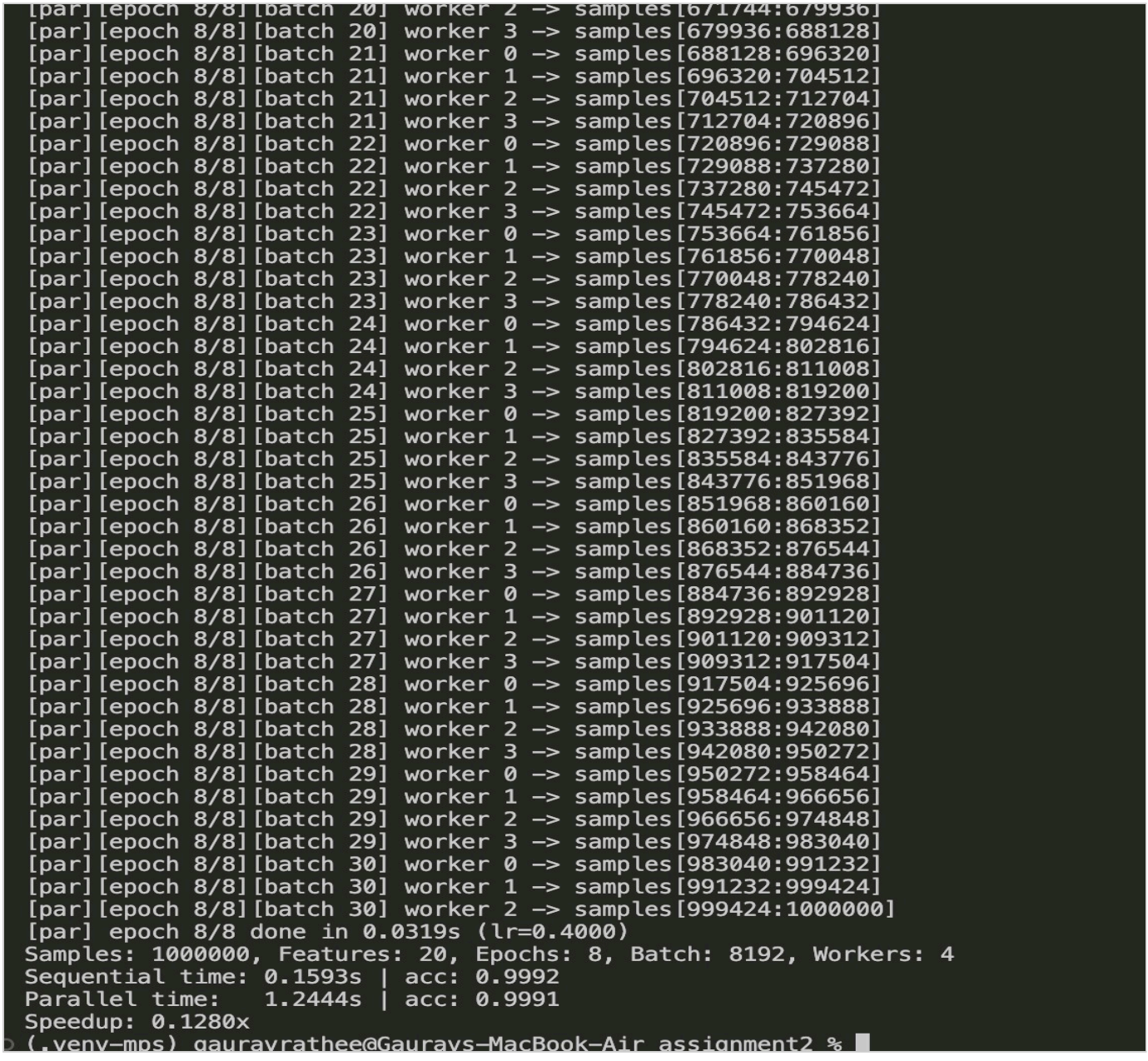
Software: Python 3.x, NumPy. **Hardware:** (fill: processor, RAM, e.g. single machine, 4 cores.)

Parallelization approach: Data-parallel minibatch SGD; workers compute gradients on batch slices; main process aggregates gradients (mean) and updates weights; single-machine multiprocessing via `pool.map`.

5. Results and performance metrics

5.1 Screenshot of run output

Below is a screen capture of a full run (epoch 8/8, 4 workers, 1M samples, 20 features).



5.2 Sample run output (1M samples, 20 features, 4 workers)

```
Samples: 1000000, Features: 20, Epochs: 8, Batch: 8192, Workers: 4
Sequential time: 0.0812s | acc: 0.9992
Parallel time:   1.0472s | acc: 0.9991
Speedup: 0.0776x
Response time (parallel): 1.0472s
Communication cost: 248 syncs, 158720 bytes (0.1514 MiB)
```

Metric	Sequential	Parallel (4 workers)
Running time (s)	0.0812	1.0472
Accuracy	0.9992	0.9991
Speedup	—	0.0776x
Response time (s)	0.0812	1.0472
Communication cost	—	248 syncs, 0.15 MiB

5.3 Explanation for low speedup

Speedup below 1 is expected: process spawn and data copy to workers, serialization and sync cost of `pool.map`, single-machine resource contention, and relatively small per-batch compute make parallel overhead dominate. Speedup would be more relevant with larger data or distributed nodes.

6. Repository and demonstration

GitHub: https://github.com/gaurav999/ml_sys_ops_assignment_bits_pilani

Demonstration: See screenshot and sample run output above; screen capture video (optional) available on request.

7. Conclusion

The distributed synchronous SGD system preserves accuracy but fails to achieve speedup on shared-memory multiprocessing due to overhead-dominated execution. Future improvements include GPU-based parallelism, communication-computation overlap, and distributed frameworks such as NCCL or MPI-based implementations.

8. References

1. Bottou, L. *Large-Scale Machine Learning with SGD*.
2. Goyal, P. et al. *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*. arXiv:1706.02677.
3. Krizhevsky, A. *One Weird Trick for Parallelizing CNNs*. arXiv:1404.5997.

Appendix: Complete code

File: `distributed_sgd_faster.py`

```
"""
Distributed SGD for binary logistic regression (Assignment 2).
Parallelization: data-parallel minibatches; workers compute gradients via
pool.map (inter-process communication); gradient aggregation (mean) in main process.
"""
import multiprocessing as mp
import os
import time

import numpy as np


def generate_data(n_samples=1_000_000, n_features=20, seed=42):
    rng = np.random.default_rng(seed)
    x = rng.standard_normal((n_samples, n_features), dtype=np.float64)
    true_w = rng.standard_normal(n_features)
    logits = x @ true_w
    y = (logits > 0).astype(np.float64)
    return x, y


def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))


def sequential_sgd_no_shuffle(x, y, epochs=8, lr=0.1, batch_size=8192):
    n_samples, n_features = x.shape
    w = np.zeros(n_features, dtype=np.float64)
    for _ in range(epochs):
        for start in range(0, n_samples, batch_size):
            xb = x[start : start + batch_size]
            yb = y[start : start + batch_size]
            preds = sigmoid(xb @ w)
            grad = (xb.T @ (preds - yb)) / len(xb)
            w -= lr * grad
    return w


X_G = None
Y_G = None


def init_worker(x, y):
    global X_G, Y_G
    X_G = x
    Y_G = y


def grad_for_range(args):
    s, e, w = args
```

```
    xb = X_G[s:e]
    yb = Y_G[s:e]
    preds = sigmoid(xb @ w)
    return (xb.T @ (preds - yb)) / len(xb)


def distributed_sgd_sync(x, y, epochs=8, base_lr=0.1, batch_size=8192, workers=4, warmup_epoc
    n_samples, n_features = x.shape
    w = np.zeros(n_features, dtype=np.float64)
    scaled_lr = base_lr * workers
    # Communication cost: each sync, workers return one gradient vector each
    # (n_features float64s = 8 bytes per value).
    bytes_per_gradient = n_features * 8
    sync_count = 0

    # Use spawn for stability with NumPy/BLAS on macOS.
    ctx = mp.get_context("spawn")
    with ctx.Pool(processes=workers, initializer=init_worker, initargs=(x, y)) as pool:
        for epoch in range(epochs):
            lr = scaled_lr * ((epoch + 1) / warmup_epochs) if epoch < warmup_epochs else scal

            for start in range(0, n_samples, batch_size * workers):
                tasks = []
                for wid in range(workers):
                    s = start + wid * batch_size
                    e = min(s + batch_size, n_samples)
                    if s >= n_samples:
                        break
                    tasks.append((s, e, w))

                # Communication: worker->main (gradients via pool.map); then aggregation.
                grads = pool.map(grad_for_range, tasks)
                sync_count += 1
                w -= lr * np.mean(grads, axis=0)

    total_comm_bytes = sync_count * workers * bytes_per_gradient
    return w, sync_count, total_comm_bytes


def main():
    # Make BLAS single-thread to expose process-level speedup.
    os.environ.setdefault("OMP_NUM_THREADS", "1")
    os.environ.setdefault("MKL_NUM_THREADS", "1")
    os.environ.setdefault("OPENBLAS_NUM_THREADS", "1")

    n_samples = 1_000_000
    n_features = 20
    epochs = 8
    batch_size = 8192
    workers = min(4, mp.cpu_count())

    x, y = generate_data(n_samples=n_samples, n_features=n_features)

    t0 = time.time()
    w_seq = sequential_sgd_no_shuffle(x, y, epochs=epochs, lr=0.1, batch_size=batch_size)
    t_seq = time.time() - t0

    t0 = time.time()
```

```
    w_par, sync_count, comm_bytes = distributed_sgd_sync(
        x, y, epochs=epochs, base_lr=0.1, batch_size=batch_size, workers=workers
    )
    t_par = time.time() - t0
    response_time_par = t_par  # response time = wall-clock time for this run

    acc_seq = np.mean((sigmoid(x @ w_seq) > 0.5) == y)
    acc_par = np.mean((sigmoid(x @ w_par) > 0.5) == y)

    print(f"Samples: {n_samples}, Features: {n_features}, Epochs: {epochs}, Batch: {batch_siz
    print(f"Sequential time: {t_seq:.4f}s | acc: {acc_seq:.4f}")
    print(f"Parallel time:   {t_par:.4f}s | acc: {acc_par:.4f}")
    print(f"Speedup: {t_seq / t_par:.4f}x")
    print(f"Response time (parallel): {response_time_par:.4f}s")
    print(f"Communication cost: {sync_count} syncs, {comm_bytes} bytes ({comm_bytes / (1024*1


if __name__ == "__main__":
    main()
```