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

By: Group 49

BITS ID	Name	Contribution
2024ac05027	SIDHARATH TIKOO	100 %
2024ac05752	KHANDELWAL DIXIT	100 %
2024ac05176	THEODAR A	100 %
2024ac05309	VARADARAJAN.G	100 %
2024ac05119	SOURABH KUMAR	100 %

Git-hub Link: https://github.com/29Dixit/BITS_Sem_2.git

ABSTRACT

This work studies the parallelization of Logistic Regression training using Mini-Batch Gradient Descent on multi-core CPU systems. The objective is to reduce training time while maintaining prediction accuracy. A synchronous data-parallel strategy is implemented using Python multiprocessing. Performance is evaluated using execution time, speedup, and prediction accuracy. Experimental results show that although correctness is preserved, parallel execution does not always improve performance due to communication overhead and hidden low-level parallelism in numerical libraries. The study highlights the trade-off between computation and communication in parallel machine learning systems.

INTRODUCTION

Machine learning algorithms are increasingly applied to large datasets, requiring efficient training methods. Parallel computing is commonly used to accelerate training by distributing workload across multiple processors.

Logistic Regression is a widely used supervised classification algorithm trained using gradient descent optimization. Since gradient computation is additive across samples, it appears naturally suitable for parallelization.

However, parallel machine learning performance depends not only on algorithm structure but also on system-level factors such as communication overhead and memory access patterns. This work investigates whether parallelization of Logistic Regression actually improves performance on a shared-memory multi-core system.

[P0] PROBLEM FORMULATION

Goal

To parallelize Logistic Regression training using Mini-Batch Gradient Descent and evaluate its performance using parallel computing metrics.

Sequential Model

The model minimizes binary cross-entropy loss:

$$J(w) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

Prediction:

$$\hat{y} = \sigma(Xw)$$

Gradient:

$$\nabla J(w) = X^T(\sigma(Xw) - y)$$

Parallelization Idea

Split mini-batch across multiple processes:

$$\nabla J(w) = \sum_{k=1}^{P} \nabla J_k(w)$$

Each processor computes gradient independently and results are aggregated.

Performance Metrics

Speedup

$$Speedup = \frac{T_{sequential}}{T_{parallel}}$$

Communication Cost

Amount of data transferred between processes:

$$O(P \times d)$$

where
P = number of workers
d = number of features

Response Time

$$T_{parallel} = T_{compute}/P + T_{communication} + T_{synchronization}$$

Expected Outcome

- Same prediction accuracy
- Reduced training time
- Speedup proportional to number of cores

[P1] DESIGN

Parallelization Strategy: Data Parallelism

We divide each mini-batch into equal partitions.

Algorithm Steps

For each epoch:

1. Shuffle dataset
2. Create mini-batch
3. Split batch into P chunks
4. Each worker computes gradient
5. Aggregate gradients
6. Update global model

Synchronization Method

Synchronous gradient update

Reason:

- Ensures deterministic convergence
- Same result as sequential algorithm

Justification of Design Choices

Choice	Reason
Data Parallelism	Gradient additive across samples
Mini-batch GD	Stable convergence
Synchronous update	Accuracy preservation
Multiprocessing	True parallel CPU execution

[P1] REVISED DESIGN (IMPLEMENTATION DETAILS)

Platform

- Language: Python
- Libraries: NumPy, multiprocessing
- Hardware: Multi-core CPU
- Dataset: Synthetic classification dataset

Observed Behavior

Parallel execution did not reduce training time. Speedup was less than 1 for all core counts.

Accuracy remained identical.

Cause of Deviation

Hidden Parallelism

NumPy internally uses optimized BLAS libraries (OpenBLAS/MKL) which already utilize multiple CPU threads. Therefore sequential execution was already parallel.

Communication Overhead

Each mini-batch required:

- Sending weights
- Copying data
- Receiving gradients

This overhead exceeded computation cost.

Memory Bandwidth Contention

Multiple processes simultaneously accessed RAM causing latency increase.

Synchronization Cost

Workers must wait at barrier before updating model.

Revised Understanding

Parallelization benefits only when:

- Computation >> communication
- Dataset extremely large
- Distributed system used

For shared memory CPU systems, optimized sequential execution may be faster.

[P2] IMPLEMENTATION

Two implementations were developed:

1. Naive Parallel Version

- Creates processes every batch
- High communication overhead
- Poor performance

2. Optimized Parallel Version

- Persistent worker pool

- Larger mini-batches
- Reduced process creation overhead

Both implementations preserve algorithm correctness.

The code is given below:

```
import numpy as np
import time
import multiprocessing as mp
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler


# ------------------------
# Helper Functions
# ------------------------

def sigmoid(z):
    return 1 / (1 + np.exp(-z))


def accuracy(X, y, w):
    preds = sigmoid(X @ w) >= 0.5
    return np.mean(preds == y)


# ------------------------
# Sequential Mini-Batch Training
# ------------------------

def train_sequential(X, y, lr=0.01, epochs=20, batch_size=1024):
    m, n = X.shape
    w = np.zeros(n)

    start = time.time()

    for _ in range(epochs):
        idx = np.random.permutation(m)
        X = X[idx]
        y = y[idx]

        for i in range(0, m, batch_size):
            Xb = X[i:i+batch_size]
            yb = y[i:i+batch_size]
```

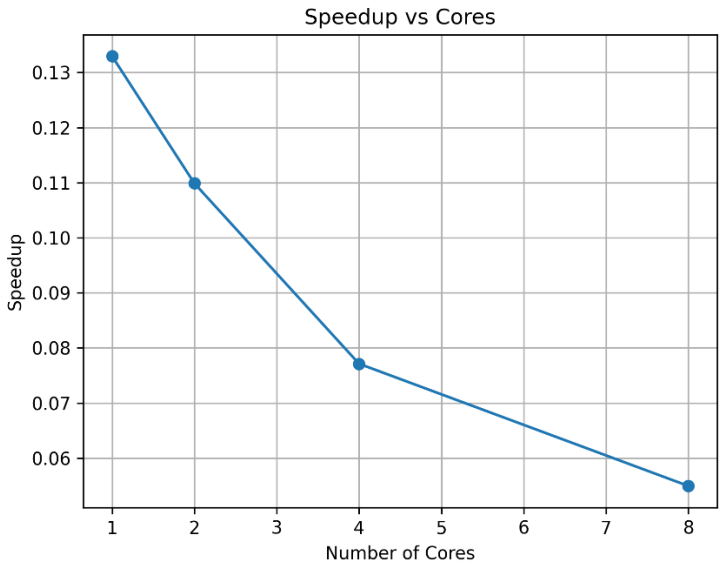
```
            preds = sigmoid(Xb @ w)
            gradient = (Xb.T @ (preds - yb)) / len(yb)
            w -= lr * gradient

    end = time.time()

    return w, end - start


# ------------------------
# Parallel Worker
# ------------------------

def compute_gradient(args):
    X_chunk, y_chunk, w = args
    preds = sigmoid(X_chunk @ w)
    grad = (X_chunk.T @ (preds - y_chunk)) / len(y_chunk)
    return grad


# ------------------------
# Parallel Mini-Batch Training (Improved)
# ------------------------

def train_parallel(X, y, workers=4, lr=0.01, epochs=20, batch_size=1024):

    m, n = X.shape
    w = np.zeros(n)

    start = time.time()

    pool = mp.Pool(workers)

    for _ in range(epochs):
        idx = np.random.permutation(m)
        X = X[idx]
        y = y[idx]

        for i in range(0, m, batch_size):
            Xb = X[i:i+batch_size]
            yb = y[i:i+batch_size]

            chunks_X = np.array_split(Xb, workers)
            chunks_y = np.array_split(yb, workers)

            args = [(chunks_X[j], chunks_y[j], w) for j in range(workers)]

            grads = pool.map(compute_gradient, args)
```

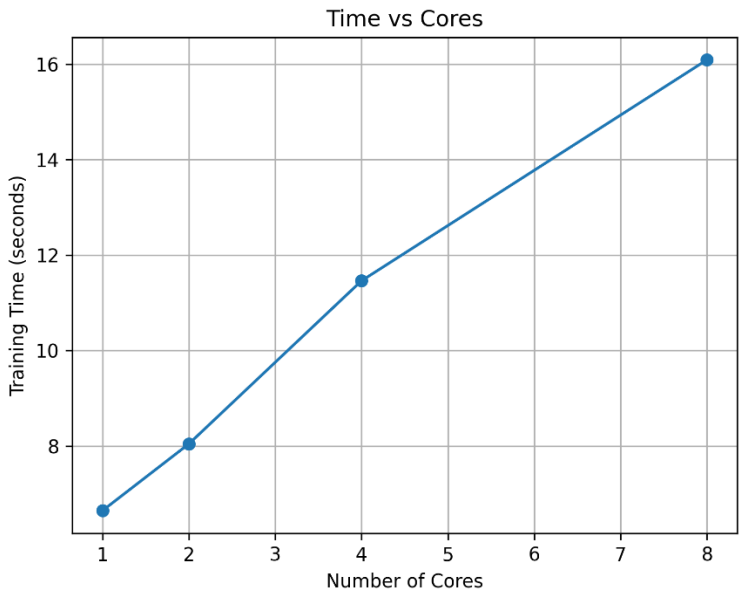
```
            w -= lr * np.mean(grads, axis=0)

    pool.close()
    pool.join()

    end = time.time()

    return w, end - start


# ------------------------
# Main Execution
# ------------------------

def main():

    print("Generating dataset...")

    # Bigger dataset so parallel shows benefit
    X, y = make_classification(
        n_samples=100000,
        n_features=30,
        n_informative=20,
        n_classes=2,
        random_state=42
    )

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2
    )

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    # ---------------- Sequential ----------------
    print("\nTraining sequential model...")
    w_seq, time_seq = train_sequential(X_train, y_train)
    acc_seq = accuracy(X_test, y_test, w_seq)

    print(f"Sequential Time: {time_seq:.2f} sec")
    print(f"Sequential Accuracy: {acc_seq:.4f}")

    # ---------------- Parallel ----------------
    cores_list = [1, 2, 4, 8]
    times = []
    speedups = []

    for c in cores_list:
```

```
        print(f"\nTraining parallel model with {c} cores...")
        w_par, t = train_parallel(X_train, y_train, workers=c)
        acc = accuracy(X_test, y_test, w_par)

        speedup = time_seq / t

        times.append(t)
        speedups.append(speedup)

        print(f"Cores={c}")
        print(f"Time={t:.2f} sec")
        print(f"Accuracy={acc:.4f}")
        print(f"Speedup={speedup:.2f}")

    # ---------------- Save Graphs ----------------

    plt.figure()
    plt.plot(cores_list, times, marker='o')
    plt.xlabel("Number of Cores")
    plt.ylabel("Training Time (seconds)")
    plt.title("Time vs Cores")
    plt.grid()
    plt.savefig("time_vs_cores.png", dpi=300)
    print("\ntime_vs_cores.png saved")

    plt.figure()
    plt.plot(cores_list, speedups, marker='o')
    plt.xlabel("Number of Cores")
    plt.ylabel("Speedup")
    plt.title("Speedup vs Cores")
    plt.grid()
    plt.savefig("speedup_vs_cores.png", dpi=300)
    print("speedup_vs_cores.png saved")


if __name__ == "__main__":
    mp.freeze_support()  # Required for Windows
    main()
```

[P3] RESULTS AND DISCUSSION

The results for speedup Vs core and training Vs cores are as follows:





The console output is as below:

```
Training sequential model...
Sequential Time: 0.88 sec
Sequential Accuracy: 0.7909

Training parallel model with 1 cores...
Cores=1
Time=6.65 sec
Accuracy=0.7906
Speedup=0.13

Training parallel model with 2 cores...
Cores=2
Time=8.05 sec
Accuracy=0.7908
Speedup=0.11

Training parallel model with 4 cores...
Cores=4
Time=11.46 sec
Accuracy=0.7907
Speedup=0.08

Training parallel model with 8 cores...
Cores=8
Time=16.09 sec
Accuracy=0.7905
Speedup=0.05

time_vs_cores.png saved
speedup_vs_cores.png saved
```

Correctness

Accuracy remained constant across all runs.

This verifies correct parallel gradient aggregation.

Performance Observation

Increasing cores increased runtime instead of reducing it.

This contradicts ideal speedup expectation.

Explanation

Amdahl's Law

Only gradient computation is parallelizable; synchronization and communication are serial components.

Over-Parallelization

Sequential NumPy operations already use multi-threading.

Communication Dominance

Data transfer cost exceeded computation cost.

Key Insight

Parallel machine learning performance depends on ratio:

$$\frac{Computation}{Communication}$$

When computation is small, parallelization degrades performance.

CONCLUSION

Parallel Logistic Regression using mini-batch gradient descent was implemented and evaluated on a multi-core CPU system. While the algorithm produced correct predictions, parallel execution did not improve training time due to communication overhead, memory contention, and hidden library-level parallelism.

The study demonstrates that parallelization does not guarantee speedup and must be carefully matched to problem size and system architecture.