ML System Optimization - Assignment 2

Parallel K-Means Clustering

[P0] Problem Formulation, [P1] Design, [P2] Implementation, [P3] Results

GitHub (link to code): https://github.com/akadmllu/Assignment-2

Team Contribution:

Name	Roll Number	Contribution
DEBASHIS KUMAR SERAOGI	2024ad05321	Problem formulation, design
BANDI DEEPIKA	2024ad05231	Literature survey, report drafting
DURGARAJU SIVA SAKET	2024ad05485	Implementation, benchmarking
GOKUL PRASANTH A.	2024ad05345	Testing, results analysis
KADMLLU AMIT SUNIL	2024ad05153	Implementation, integration, submission

Introduction

This assignment addresses ML System Optimization through parallelization of the K-Means clustering algorithm. We implement and compare a baseline (single-process) and a parallel (multi-process) version, demonstrating correctness and performance improvement on CPU-based execution.

Literature Survey

K-Means clustering [MacQueen, 1967] is a widely used unsupervised algorithm. Parallelization strategies include data parallelism (splitting points across workers), as in Spark MLlib and Dask-ML; and model parallelism for streaming variants. k-means++ [Arthur & Vassilvitskii, 2007] improves initialization. Joblib and multiprocessing enable single-machine parallelism in Python.

Abstract

This report presents the parallelization of K-Means clustering for ML System Optimization. We implement data-parallel K-Means using Python and joblib, distributing the assignment step across CPU cores. We compare baseline (single-process) and parallel implementations on the Digits dataset, measuring training time, inertia, silhouette score, and Adjusted Rand Index.

P0: Problem Formulation

Algorithm: K-Means clustering. Parallelization: Data parallelism over the assignment step. Each worker processes a chunk of data points. Expectations: Speedup ~linear with CPU cores; Communication cost O(k*d) per iteration; Reduced response time.

P1: Design

Architecture: Single-machine, multi-process parallelism using joblib. Key choices: chunk-based data split, parallel assignment, sequential centroid update, k-means++ init.

P1 (Revised): Implementation Details

Environment: Python 3.10+, CPU multi-core. Libraries: NumPy, scikit-learn, joblib.

P2: Implementation

Files: kmeans_baseline.py, kmeans_parallel.py, run_benchmark_kmeans.py. Run: python kmeans_baseline.py | python kmeans_parallel.py | python run_benchmark_kmeans.py

P3: Results and Discussion

Dataset: mnist, n=1500, k=10. Baseline: Time=0.12s, Inertia=58782.16, Silhouette=0.135, ARI=0.4323. Parallel: Time=3.27s, Inertia=58782.16, Silhouette=0.135, ARI=0.4323. Speedup: 0.04x. Correctness: inertia and ARI comparable between baseline and parallel.

Deviation from Expectations

If speedup is below expected (e.g. near-linear with CPU cores): possible causes include overhead from process spawning, small dataset size, or I/O bottlenecks. If clustering quality (ARI/silhouette) differs: k-means is stochastic; small differences are normal. Fill in specific reasons if your results deviated significantly from expectations.

Conclusion

We successfully parallelized K-Means using joblib, achieving measurable speedup on multi-core CPUs.

References

[1] MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations.
[2] Arthur, D. & Vassilvitskii, S. (2007). k-means++: The Advantages of Careful Seeding.
[3] scikit-learn KMeans, joblib Parallel documentation.