Project Report

PARALLEL K-MEANS CLUSTERING WITH FORK/JOIN

Course Name : Parallel Processing and High Performance Computing

Prepared by:

Salma Kamal

Toka Abdelaziz

Mariam Saeed

Adham Zineldin

Mohamed Ali

Mohamed Maged

Table of Contents

1. Introduction

K-Means clustering is a widely used unsupervised learning algorithm for partitioning data into K groups. In this project, we implemented both Sequential and Parallel (Fork/Join) versions of K-Means and evaluated their performance on different datasets.

A key enhancement we used is K-Means++ initialization, which significantly improves clustering quality by choosing smarter initial centroids.

The goal is to:

- Compare performance (runtime, speedup)
- Compare quality (SSE)
- Explain convergence behavior
- Discuss parallel design and trade-offs

2. K-Means Algorithm with K-Means++ Initialization

K-Means works in four main steps:
(1) initialization, (2) assignment, (3) update, (4) convergence.

2.1 Initialization with K-Means++

Instead of choosing random initial centroids, we used **K-Means++**, which selects initial centroids using a probabilistic distance-based method:

1. Choose one centroid uniformly at random.
2. For each remaining centroid:
   - Compute distance of each point to its nearest selected centroid.
   - Choose the next centroid with probability proportional to distance^2.

Why K-Means++?

- Produces **better SSE** than random initialization
- Reduces variability between runs
- Leads to **faster convergence**

Using K-Means++ improved stability in both sequential and parallel versions.

3. Parallel Design Using Fork/Join

The parallel implementation divides work among multiple threads using Java's Fork/Join framework.

3.1 Parallel Assignment

- Dataset is partitioned into chunks.
- Each chunk processes distance calculations independently.
- Threads assign points to their nearest centroid in parallel.

3.2 Parallel Centroid Recalculation

Each thread computes partial sums for its chunk. Results are later reduced (merged) to form new centroids.

3.3 Synchronization

Fork/Join ensures:

- Efficient task splitting
- Work stealing for load balancing
- Minimal overhead compared to manual thread handling

4.Experimental Results

4.1 Mall Customers Dataset Results

Method	K	SSE	Runtime (ms)	Iterations	Initialization
Sequential	5	98101.5015	53 ms	11	Random
Sequential	5	75493.8446	20 ms	11	k-means++
Parallel	5	97234.2977	34 ms	11	Random
MultiStart Parallel	5	75479.7643	29 ms	12	k-means++

4.1.1 Results(Sequential vs Parallel Comparison)

Sequential Results:

SSE: 75479.7643

Runtime: 7 ms

Iterations: 4

Parallel Results:

SSE: 75493.8446

Runtime: 9 ms

Iterations: 8

Performance:

Speedup: 0.78x

Time Saved: -2 ms (-28.6%)

Quality:

SSE Difference: 14.0803

Results differ slightly (random initialization)

4.2 Bank Customers Dataset Results

Method	K	SSE	Runtime (ms)	Iterations	Initialization
Sequential	3	274825720494.39	198 ms	26	Random
Sequential	3	274825720494.39	127 ms	22	k-means++
Parallel	3	274825720494.39	247 ms	27	Random
MultiStart Parallel	3	274825720494.39	490 ms	12	k-means++

4.2.1 Results(Sequential vs Parallel Comparison)

Sequential Results:	**Parallel Results:**
SSE: 274825758968.857	SSE: 274825758968.8580
Runtime: 69 ms	Runtime: 70 ms
Iterations: 18	Iterations: 15

Performance:

Speedup: 0.99x

Time Saved: -1 ms (-1.4%)

Quality:

SSE Difference: 0.0002

Results are identical (within tolerance)

5. Interpretation Of The Results

- Parallel runtime was slightly slower because:

    - Dataset sizes were small
    - Fork/Join overhead > actual computation time
- SSE values between sequential and parallel were very close, showing good stability with K-Means++.
- Parallel version converged in fewer iterations due to:
    - Better centroid spread from K-Means++
    - Minor floating-point differences in parallel reduction
- Speedup < 1 indicates that parallelization is not beneficial for small datasets.

6.Trade-offs

6.1 Speed vs Quality

- Sequential is faster on small data (7 ms vs 9 ms).
- Parallel can be slightly slower but sometimes needs fewer iterations on larger data.

6.2 Accuracy vs Performance

- SSE is almost identical in both versions (very small differences).
- Any slight variation comes from the random nature of KMeans++.

6.3 Resource Usage vs Benefit

- Parallel uses multiple CPU threads, but the dataset size is not large enough to show real speedup.

6.4 Stability vs Randomness

- KMeans++ improves consistency, but results can still vary slightly between runs.

6.5 Dataset Size Impact

- The datasets used are not large →
  Parallel processing does not provide a noticeable speed advantage due to thread overhead.