National Textile University, Faisalabad

Department of Computer Science

Name	Syed Junaid Jaffery
Class	BSCS-A-6
Registration No.	22-NTU-CS-1167
Assignment	Assignment-1-OpenMP
Course Code	CSC-3075
Course Name	Parallel and Distributed Computing
Submitted To	Sir Nasir Mahmood
Submission Date	March 10, 2025

K-Means Clustering: Parallelization and Performance Analysis

📌 Introduction to K-Means Clustering

K-Means clustering is an **unsupervised machine learning algorithm** used for grouping similar data points into **K distinct clusters**. It aims to minimize intra-cluster variance and iteratively refines cluster centroids.

🔄 Steps in K-Means Clustering

1. **Initialization Step:** Select K initial cluster centroids randomly from the dataset.
2. **Assignment Step:** Assign each data point to the nearest cluster centroid.
3. **Summation Step:** Compute the sum of all points assigned to each cluster.
4. **Mean Calculation Step:** Update each centroid as the mean of all assigned points.
5. **Repeat Until Convergence:** Repeat steps 2-4 until centroids stop changing significantly.

🚀 Challenges in K-Means

- **Choosing K:** Finding the optimal number of clusters can be tricky.
- **Computational Cost:** It has a complexity of O(n k i d).
  - n → Number of data points.
  - k → Number of clusters.
  - i → Number of iterations until convergence.
  - d → Dimensionality of the data.
- **Initialization Sensitivity:** Poor initial centroids may lead to suboptimal clusters.

💡 Why Parallelize K-Means?

K-Means involves **intensive computations**, making it slow for large datasets. Parallelization using **OpenMP** helps distribute workload across multiple cores, significantly reducing execution time.

🖥️ Implementation Overview

1️⃣ Sequential Implementation

We first implemented a sequential version of K-Means as a baseline.

```
%%file K_means_seq.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define NUM_POINTS 1000000   // A much larger dataset (adjust as needed)
#define DIM 2                // 2D points (x and y)
#define K 3                  // Number of clusters
#define MAX_ITER 100         // Maximum iterations
// Function to compute squared Euclidean distance between two points.
double distance_sq(double p1[], double p2[]) {
    double sum = 0.0;
    for (int d = 0; d < DIM; d++) {
        double diff = p1[d] - p2[d];    // the X2 - X1 step
        sum += diff * diff;             // the sqauring
    }
    return sum;
}
int main() {
    int i, j, iter;
    // Allocate memory for our dataset: an array of pointers to each point.
    double **data = malloc(NUM_POINTS * sizeof(double *));
    for (i = 0; i < NUM_POINTS; i++) {
        data[i] = malloc(DIM * sizeof(double));
    }
    for (i = 0; i < NUM_POINTS; i++) {
        for (j = 0; j < DIM; j++) {
```

```
            data[i][j] = (double)rand() / RAND_MAX;
        }
    }
    int *labels = calloc(NUM_POINTS, sizeof(int));  // Initializes to 0.
    double centroids[K][DIM];
    for (i = 0; i < K; i++) {
        for (j = 0; j < DIM; j++) {
            centroids[i][j] = data[i][j];
        }
    }
    double start_time = omp_get_wtime();
    int changed = 1;  // Flag to check if any point changes its cluster.
    for (iter = 0; iter < MAX_ITER && changed; iter++) {
        changed = 0;
        // Assignment Step: assign each point to the nearest centroid.
        for (i = 0; i < NUM_POINTS; i++) {
            int best_cluster = 0;
            double best_dist = distance_sq(data[i], centroids[0]);
            for (j = 1; j < K; j++) {
                double d = distance_sq(data[i], centroids[j]);
                if (d < best_dist) {
                    best_dist = d;
                    best_cluster = j;
                }
            }
            if (labels[i] != best_cluster) {
                labels[i] = best_cluster;
                changed = 1;
            }
        }

        // Update Step: recompute centroids as the mean of points in each cluster.
        double new_centroids[K][DIM] = {0};  // Temporary sums for each centroid.
        int counts[K] = {0};                 // Number of points in each cluster.

        // Sum the coordinates for each cluster.
        for (i = 0; i < NUM_POINTS; i++) {
            int cluster = labels[i];
```

```
            counts[cluster]++;
            for (j = 0; j < DIM; j++) {
                new_centroids[cluster][j] += data[i][j];
            }
        }
        // Calculate the mean (average) for each centroid.
        for (i = 0; i < K; i++) {
            if (counts[i] > 0) {  // Avoid division by zero.
                for (j = 0; j < DIM; j++) {
                    centroids[i][j] = new_centroids[i][j] / counts[i];
                }
            }
        }
    }

    // Timing: end the clock.
    double end_time = omp_get_wtime();
    double elapsed = end_time - start_time;
    // Print out the results.
    printf("K-Means converged in %d iterations.\n", iter);
    printf("Elapsed time (sequential): %f seconds\n", elapsed);
    printf("Final centroids:\n");
    for (i = 0; i < K; i++) {
        printf("Cluster %d: ", i);
        for (j = 0; j < DIM; j++) {
            printf("%f ", centroids[i][j]);
        }
        printf("\n");
    }
    // Free allocated memory.
    for (i = 0; i < NUM_POINTS; i++) {
        free(data[i]);
    }
    free(data);
    free(labels);

    return 0;
}
```

2 Parallel Implementation

The parallelized code is as follows:

```
%%file K_means_para.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define NUM_POINTS 1000000   // A much larger dataset (adjust as needed)
#define DIM 2                // 2D points (x and y)
#define K 3                  // Number of clusters
#define MAX_ITER 100         // Maximum iterations
double distance_sq(double p1[], double p2[]) {
    double sum = 0.0;
    for (int d = 0; d < DIM; d++) {
        double diff = p1[d] - p2[d];    // the X2 - X1 step
        sum += diff * diff;             // the sqauring
    }
    return sum;
}
int main() {

    int i, j, iter; // we can initialize these inside the loop too as usual bt
    double **data = malloc(NUM_POINTS * sizeof(double *));  // I don't want to give a size to the array, i want it
dynamic so thats why its a pointer array.
    for (i = 0; i < NUM_POINTS; i++) {  // now inside each index of the data pointer array, we allocate space for
each individual entry. we go through each pointer in our data array and allocate an array of DIM doubles
        data[i] = malloc(DIM * sizeof(double));
    }

    // Generate random data points in the range [0, 1] for each dimension.
    for (i = 0; i < NUM_POINTS; i++) {
        for (j = 0; j < DIM; j++) {
            data[i][j] = (double)rand() / RAND_MAX;
        }
    }

    // Array for cluster assignments (labels) for each point.
```

```
    int *labels = calloc(NUM_POINTS, sizeof(int));  // Initializes to 0.

    // Initialize centroids (K x DIM). We'll use the first K points as our initial centroids.
    double centroids[K][DIM];
    for (i = 0; i < K; i++) {
        for (j = 0; j < DIM; j++) {
            centroids[i][j] = data[i][j];
        }
    }
    double start_time = omp_get_wtime();
    int changed = 1;  // Flag to check if any point changes its cluster.

 // Each iteration depends on the results of the previous iteration, so you cannot fully parallelize across iter.
We won't try to parallelize this loop
    for (iter = 0; iter < MAX_ITER && changed; iter++) {
        changed = 0;
        // Assignment Step: assign each point to the nearest centroid.
        // in the following loop, we are accessing the data points (rows of the 2D array you can say) and hence we
can use diff threads to access these data points separately, so parallel for can be used here:
        #pragma omp parallel for private(j) reduction(|:changed)
        for (i = 0; i < NUM_POINTS; i++) {  // ASSIGNMENT STEP
            int best_cluster = 0;
            double best_dist = distance_sq(data[i], centroids[0]);  // no race condition here since its local, and
no race condition on i either since the for directive takes care of that on its own.
                for (j = 1; j < K; j++) {   // there will be a race condition on j since it is initialized outside
this region so its not local, this is why we made it private in the directive above
                double d = distance_sq(data[i], centroids[j]);
                if (d < best_dist) {
                    best_dist = d;
                    best_cluster = j;
                }
            }
            if (labels[i] != best_cluster) {
                labels[i] = best_cluster;
                changed |= 1;
            }
        }
```

```
        // Update Step: recompute centroids as the mean of points in each cluster.
        double new_centroids[K][DIM] = {0};  // Temporary sums for each centroid.
        int counts[K] = {0};                 // Number of points in each cluster.
#pragma omp parallel
{
    // Allocate thread-local arrays for partial sums and counts
    double local_new_centroids[K][DIM] = {0};
    int local_counts[K] = {0};

    // Each thread processes a portion of the data:
    #pragma omp for nowait  // the threads and their iterations don't depend on each other and they can merge and
end when ever they wish so using nowait here is acceptable
    for (int i = 0; i < NUM_POINTS; i++) {
        int cluster = labels[i];
        local_counts[cluster]++;    // before, we had a race condition here. lets walk through it:
        for (int d = 0; d < DIM; d++) {
            local_new_centroids[cluster][d] += data[i][d];
        }
    }
    #pragma omp critical
    {
        for (int c = 0; c < K; c++) {
            counts[c] += local_counts[c];
            for (int d = 0; d < DIM; d++) {
                new_centroids[c][d] += local_new_centroids[c][d];
            }
        }
    }
}

        for (i = 0; i < K; i++) {   //  MEAN CALCULATIN STEP
            if (counts[i] > 0) {  // Avoid division by zero.
                for (j = 0; j < DIM; j++) {
                    centroids[i][j] = new_centroids[i][j] / counts[i];
                }
            }
        }
    }
```

```
    // Timing: end the clock.
    double end_time = omp_get_wtime();
    double elapsed = end_time - start_time;
    printf("K-Means converged in %d iterations.\n", iter);
    printf("Elapsed time (sequential): %f seconds\n", elapsed);
    printf("Final centroids:\n");
    for (i = 0; i < K; i++) {
        printf("Cluster %d: ", i);
        for (j = 0; j < DIM; j++) {
            printf("%f ", centroids[i][j]);
        }
        printf("\n");
    }
    for (i = 0; i < NUM_POINTS; i++) {
        free(data[i]);
    }
    free(data);
    free(labels);

    return 0;

}
```

1. Parallelizing the Assignment Step

I used:

```
#pragma omp parallel for private(j) reduction(|:changed)
```

- **Why private(j)?**
  - j is used inside the nested loop, and each thread must have its own version of j to avoid race conditions.
- **Why reduction(|:changed)?**
  - changed is a shared variable and is updated by multiple threads. The reduction ensures all thread updates are combined correctly.

2. Parallelizing the Summation Step

- **Race conditions on counts[]:** Instead of making counts[] global, we create local copies (local_counts[]).
- **Race conditions on new_centroids[][]:** Again, we use local copies (local_new_centroids[][]) and merge them using #pragma omp critical.

#pragma omp parallel

- Explanation
  - This directive creates a parallel region, meaning that multiple threads will now execute the following block concurrently.
  - Inside this block, each thread has its own local copies of local_counts[] and local_new_centroids[][], which prevents race conditions.
  - The creation of thread-local storage ensures that no two threads simultaneously modify the same memory locations, avoiding conflicts.

#pragma omp for nowait

- Explanation
  - This distributes the loop iterations across multiple threads.
  - Without this, even though we are inside a parallel region, the loop would still execute sequentially.
  - By default, OpenMP synchronizes all threads at the end of a parallel loop.
  - Using **nowait** removes this synchronization, allowing threads to finish their work independently and continue execution without waiting for others.

#pragma omp critical

- Explanation
  - This ensures that only one thread at a time can execute the enclosed block.
  - Since each thread has its own local copies, they must be combined into the global counts[] and new_centroids[][] arrays.
  - If multiple threads tried to update these shared variables at the same time, race conditions would occur.
  - Why not atomic instead?
    - atomic only works on single variable operations, like counts[cluster]++.
    - Here, we are updating entire arrays, which makes critical a better option.

In [1]:
```
import pandas as pd
df = pd.read_excel("PDC-A1-Analysis.xlsx")
```

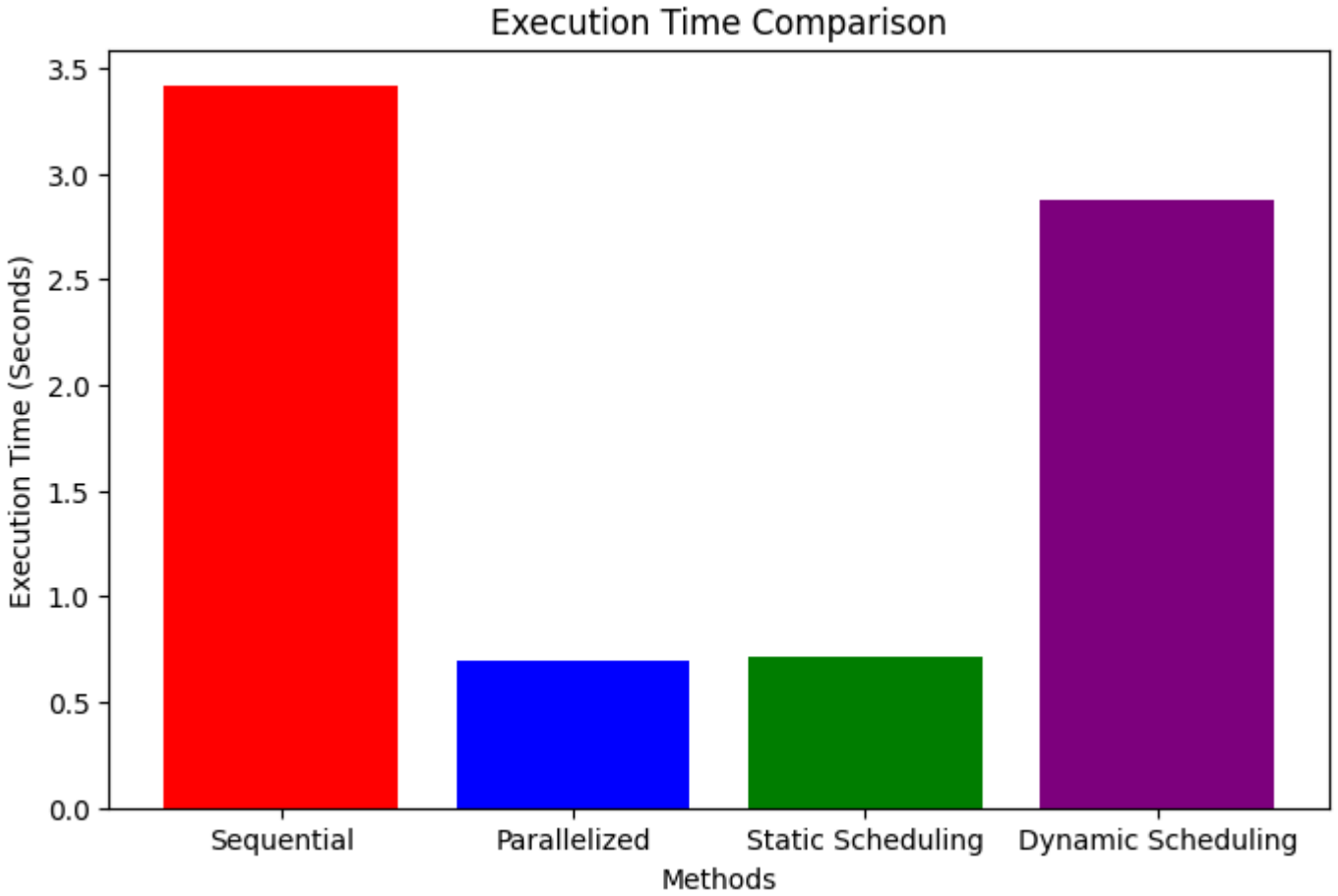
```
df.head(12)
```

Out[1]:

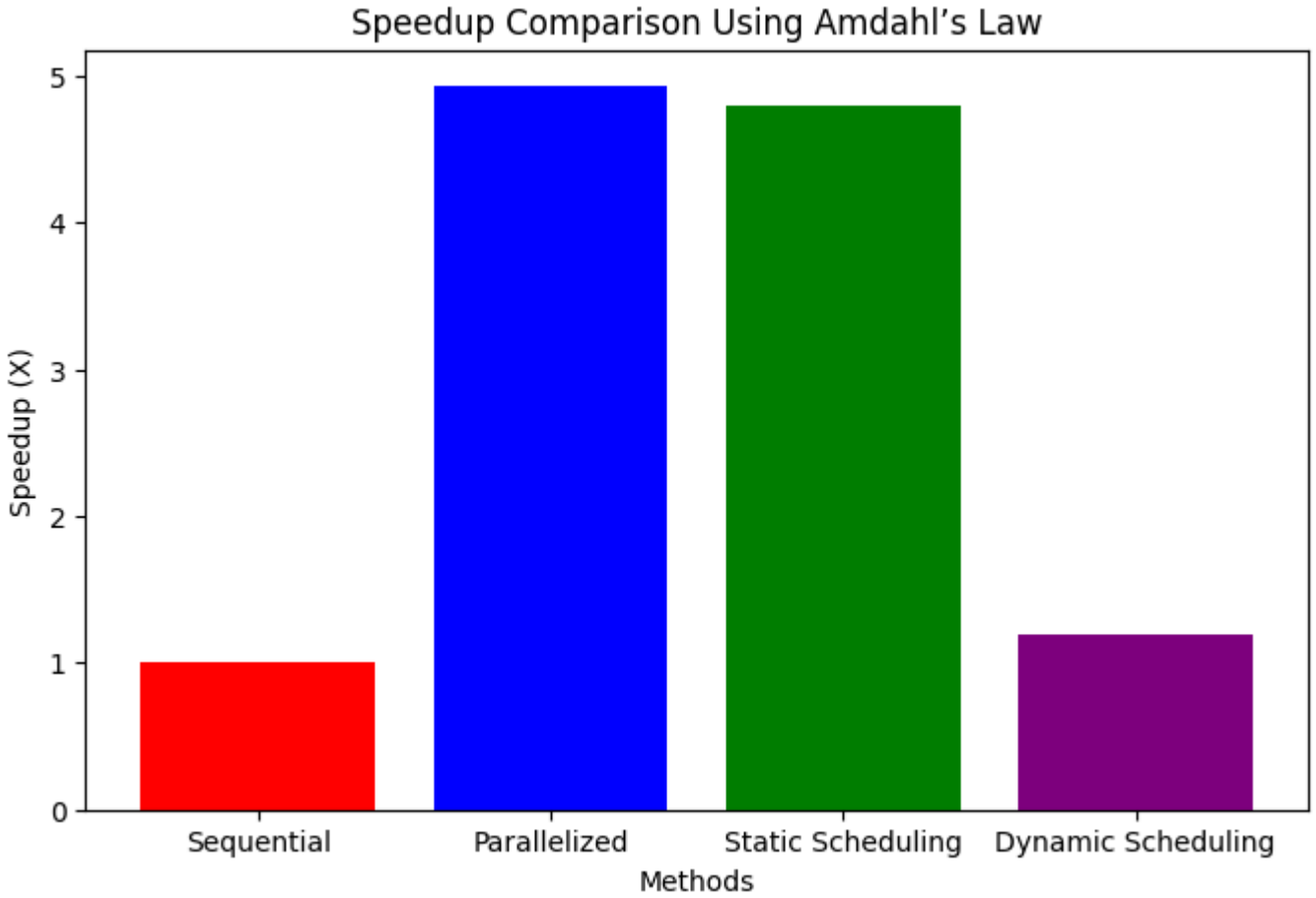
	Execution No.	Sequential	Parallelized	Static Scheduling	Dynamic scheduling
0	1	3.307023	0.590179	0.747647	2.866114
1	2	3.625141	0.773958	1.284712	2.546029
2	3	3.394125	0.638052	0.652273	2.859618
3	4	3.369340	0.786763	0.580888	2.872311
4	5	3.326923	0.654928	0.701048	3.043738
5	6	3.317143	0.801457	0.609893	3.029168
6	7	3.650981	0.796848	0.664077	3.020177
7	8	3.424207	0.674847	0.603265	2.915599
8	9	3.360508	0.667826	0.638367	2.878635
9	10	3.384621	0.547029	0.648159	2.761101
10	AVERAGE	3.416001	0.693189	0.713033	2.879249
11	Speedup	1.000000	4.927953	4.790804	1.186421

In [2]:

```
import matplotlib.pyplot as plt
methods = ["Sequential", "Parallelized", "Static Scheduling", "Dynamic Scheduling"]
execution_times = [3.4160, 0.6931, 0.7130, 2.8792]
plt.figure(figsize=(8, 5))
plt.bar(methods, execution_times, color=['red', 'blue', 'green', 'purple'])
plt.xlabel("Methods")
plt.ylabel("Execution Time (Seconds)")
plt.title("Execution Time Comparison")
plt.show()
```



In [3]:
```
speedup = [3.4160 / t for t in execution_times]
plt.figure(figsize=(8, 5))
plt.bar(methods, speedup, color=['red', 'blue', 'green', 'purple'])
plt.xlabel("Methods")
plt.ylabel("Speedup (X)")
plt.title("Speedup Comparison Using Amdahl's Law")
plt.show()
```


Speedup Comparison Using Amdahl’s Law
Speedup (X)
0
1
2
3
4
5
Sequential
Parallelized
Static Scheduling
Dynamic Scheduling
Methods