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 21, 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

Dataset Size: For all the implementations below, the size of the data is the same: One Million Data Points

---

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;
}
```

Sequential Execution Time

1	3.307023
2	3.625141
3	3.394125
4	3.36934
5	3.326923
6	3.650981
7	3.360508
8	3.317143
9	3.384621
10	3.424207
AVG	3.4160012

2️⃣ 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.

Parallelized Execution time (Static Scheduling with no chunk size specified)

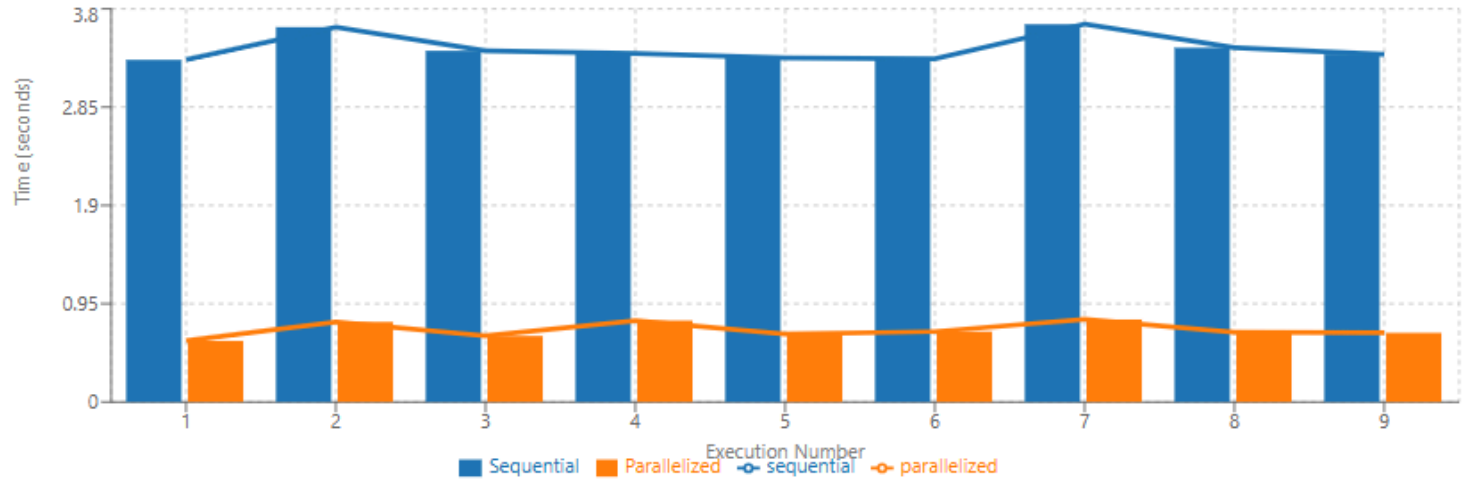
1	0.786763
2	0.773958
3	0.590179
4	0.654928
5	0.801457
6	0.796848
7	0.674847
8	0.667826
9	0.547029
10	0.638052
AVG	0.6931887

Parallel(static with no chunk size specified) VS Sequential Results

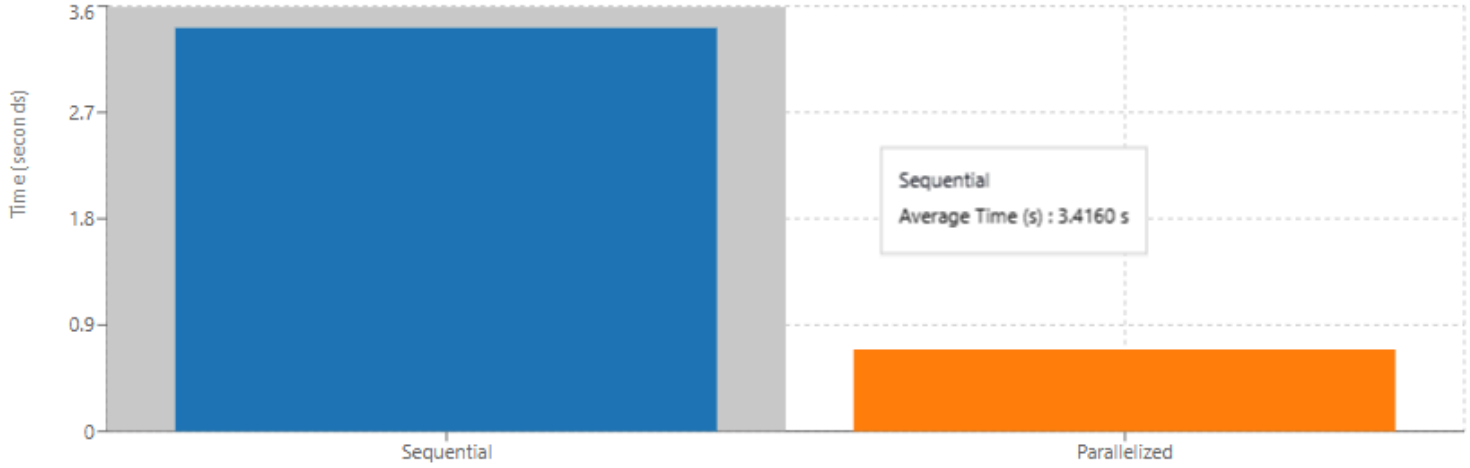
K-means Clustering Performance Analysis

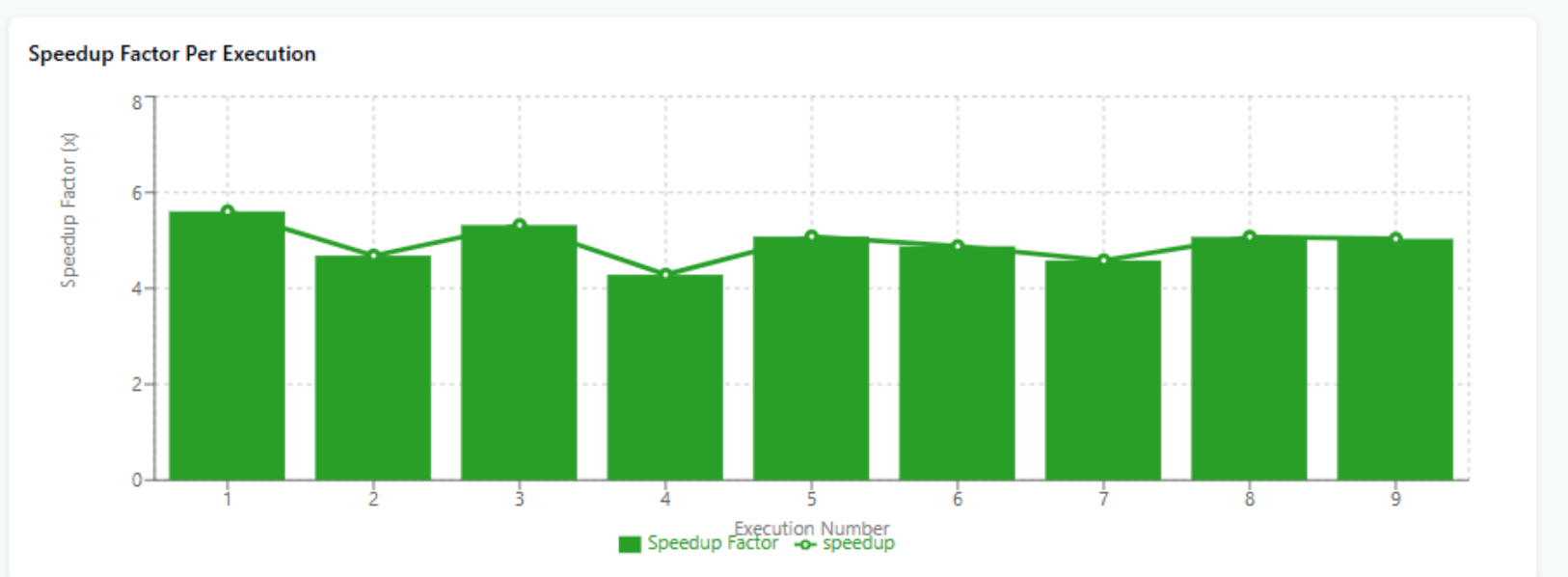
OpenMP Parallelization Results: Average Speedup = 4.93x

Execution Time Comparison



Average Execution Time



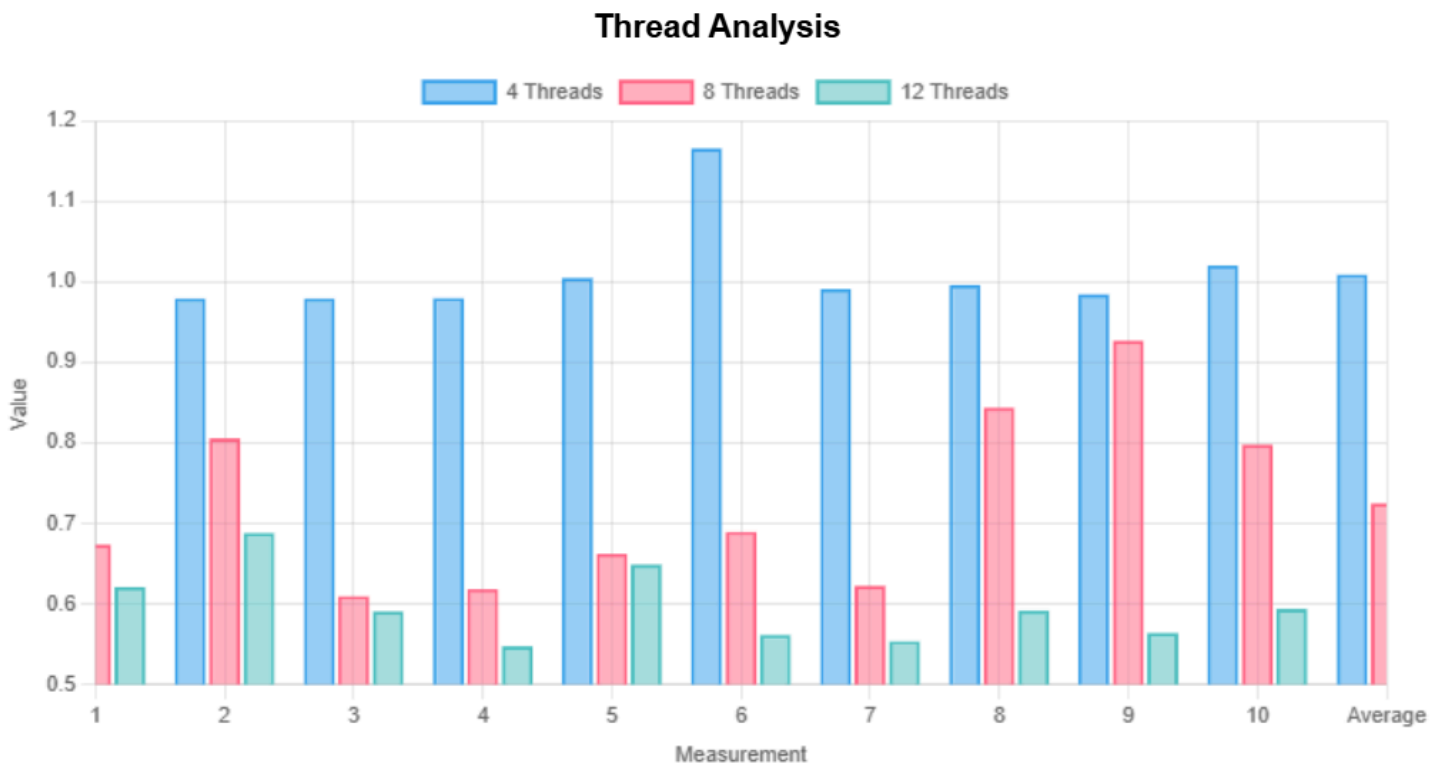


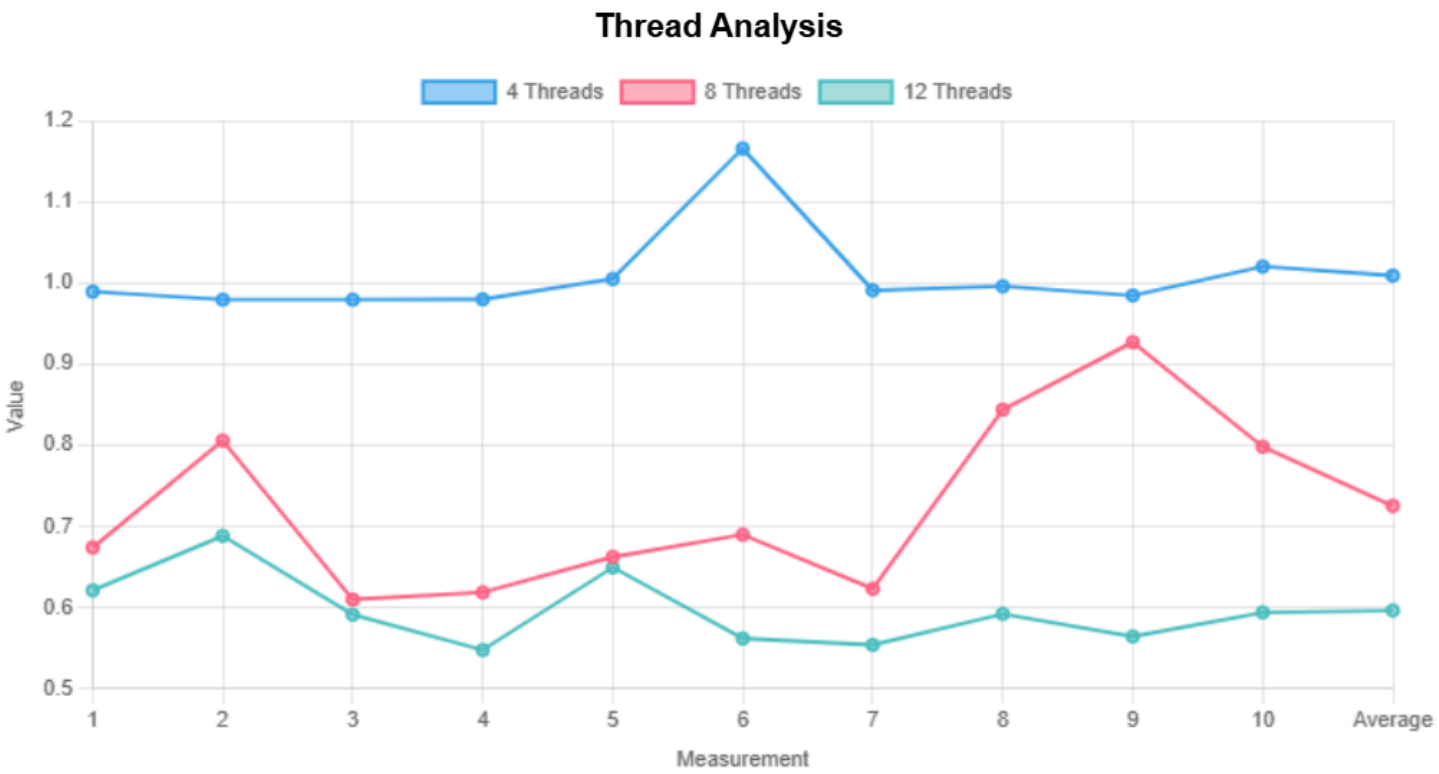
---

3 Thread Analysis (on parallel version of the code with static scheduling)

Thread Analysis			
Index	**4**	**8**	**12**
1	0.98955	0.674092	0.621129
2	0.979596	0.805838	0.688407
3	0.979596	0.609953	0.591169
4	0.980048	0.618639	0.547644
5	1.005169	0.662349	0.649396
6	1.165956	0.689789	0.561849
7	0.991148	0.62303	0.553966
8	0.996056	0.844069	0.592145
9	0.984832	0.92725	0.56433

10	1.020416	0.798239	0.593982
Average	**1.0092367**	**0.7253248**	**0.5964017**

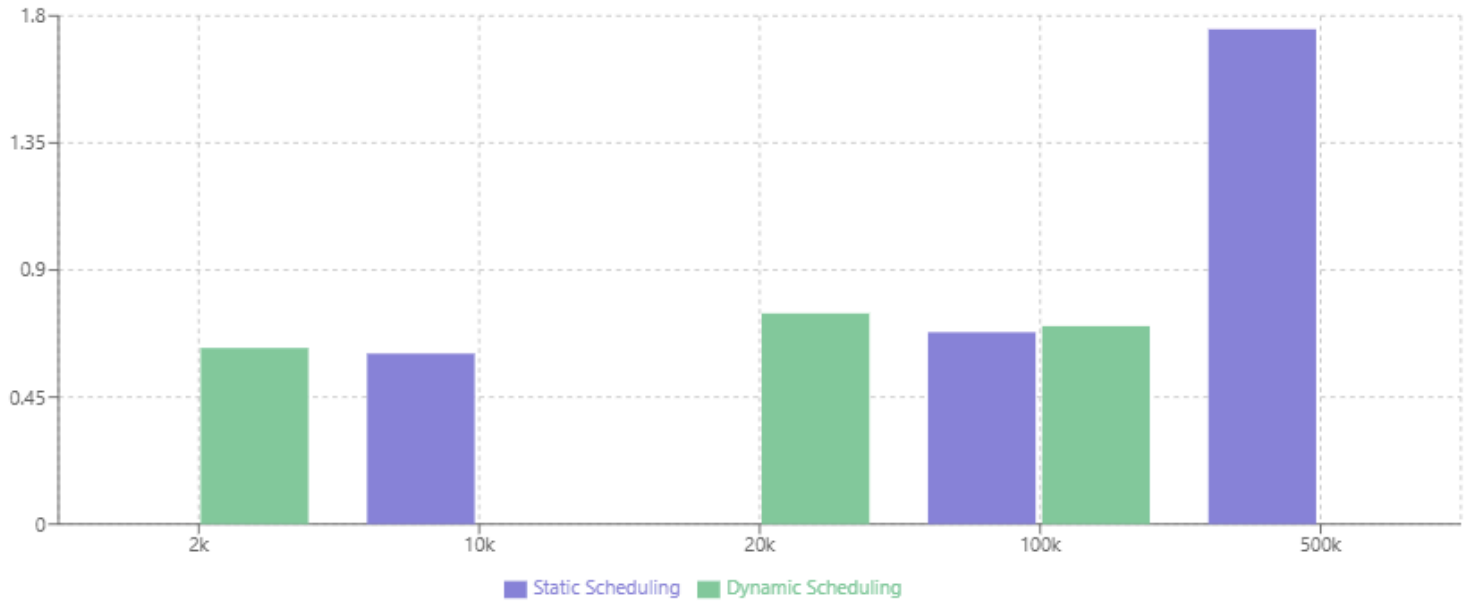

Thread Analysis
4 Threads
8 Threads
12 Threads
1.2
1.1
1.0
0.9
0.8
0.7
0.6
0.5
Value
1
2
3
4
5
6
7
8
9
10
Average
Measurement

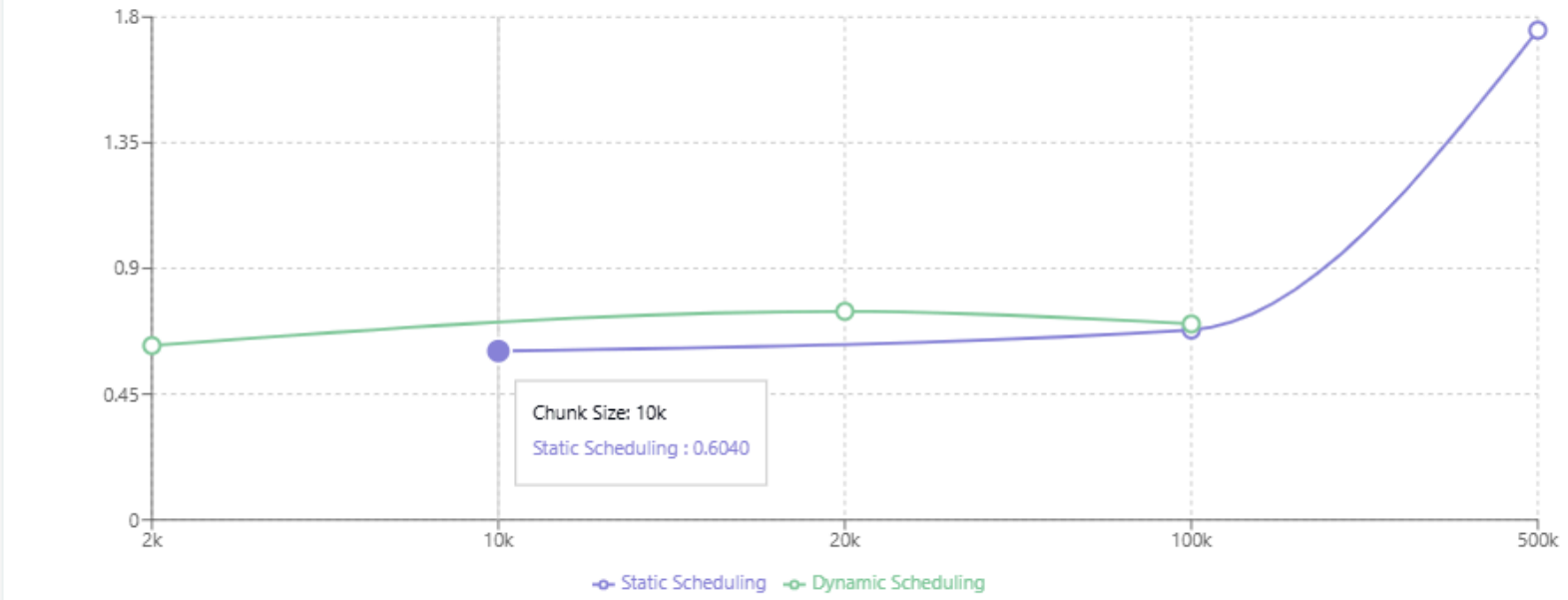


Thread Analysis Performance Trends:

- **Inverse Relationship**: As thread count increases, performance values generally decrease.
- **Diminishing Returns**: The performance enhancement from 4→8 threads (28%) is larger than from 8→12 threads (18%).
- **Scaling Inefficiency**: Eventually, as you keep increasing the threads, you will start getting diminishing returns.

4️⃣ Chunk Size Analysis


1.8
1.35
0.9
0.45
0
2k
10k
20k
100k
500k
Static Scheduling
Dynamic Scheduling


1.8
1.35
0.9
0.45
0
2k
10k
20k
100k
500k
Chunk Size: 10k
Static Scheduling : 0.6040
Static Scheduling
Dynamic Scheduling

Static Chunk Size Analysis:

- Smaller Chunks (10k): The execution time is the lowest (0.5531 seconds). This is likely due to more balanced distribution, allowing for better load balancing between threads. Smaller chunks reduce the waiting time for threads, ensuring each one gets work to do more quickly.

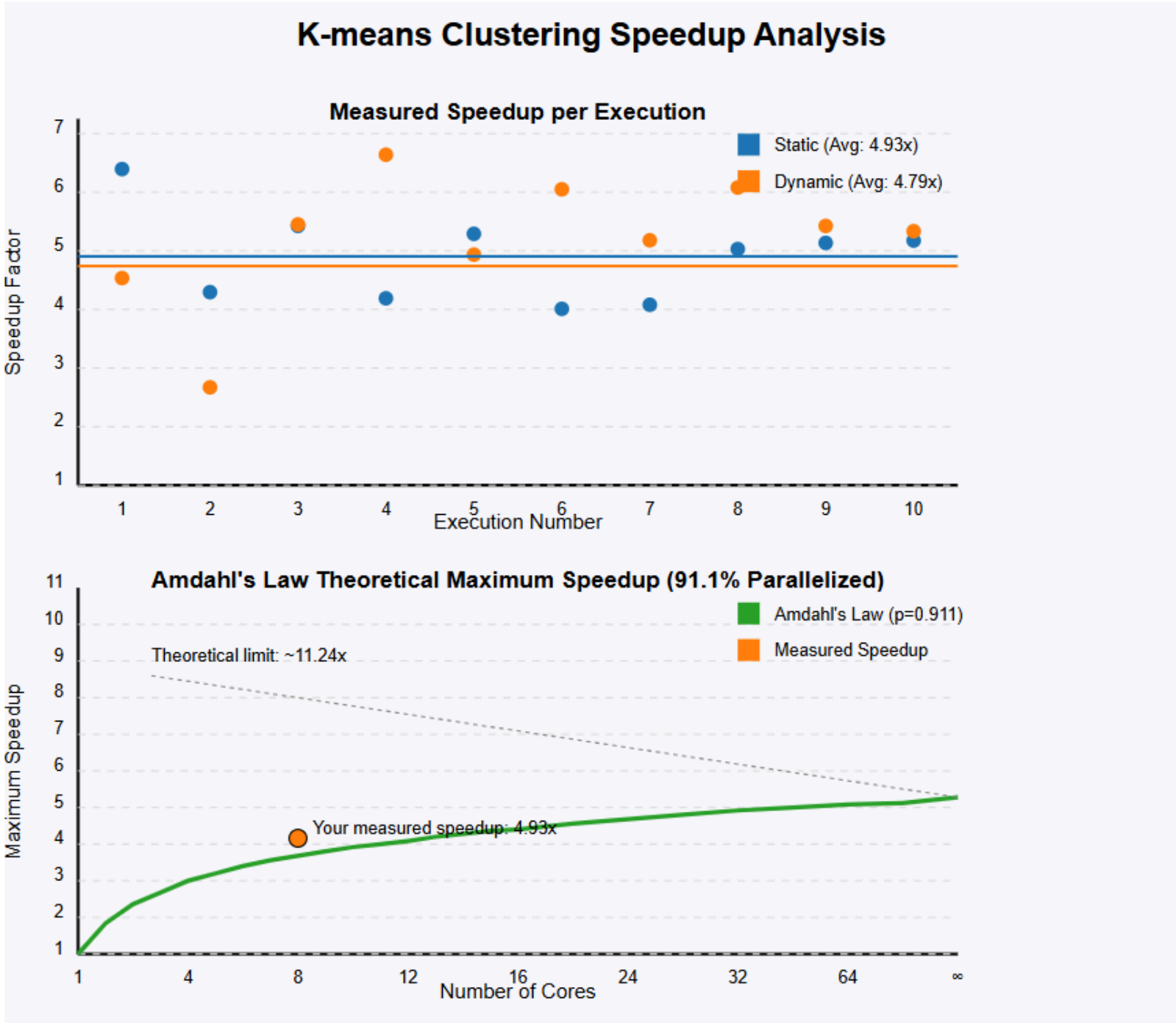
- Medium Chunks (100k): Performance is slightly slower (0.6516 seconds). While chunking is still reasonable, there may be a small overhead due to larger work units being assigned, which can lead to a bit of inefficiency.
- Larger Chunks (500k): The execution time increases significantly (2.6363 seconds). This is expected because, with larger chunks, the threads might not be as efficiently utilized, and some threads might finish earlier while others are still working on their chunks, leading to idle time.

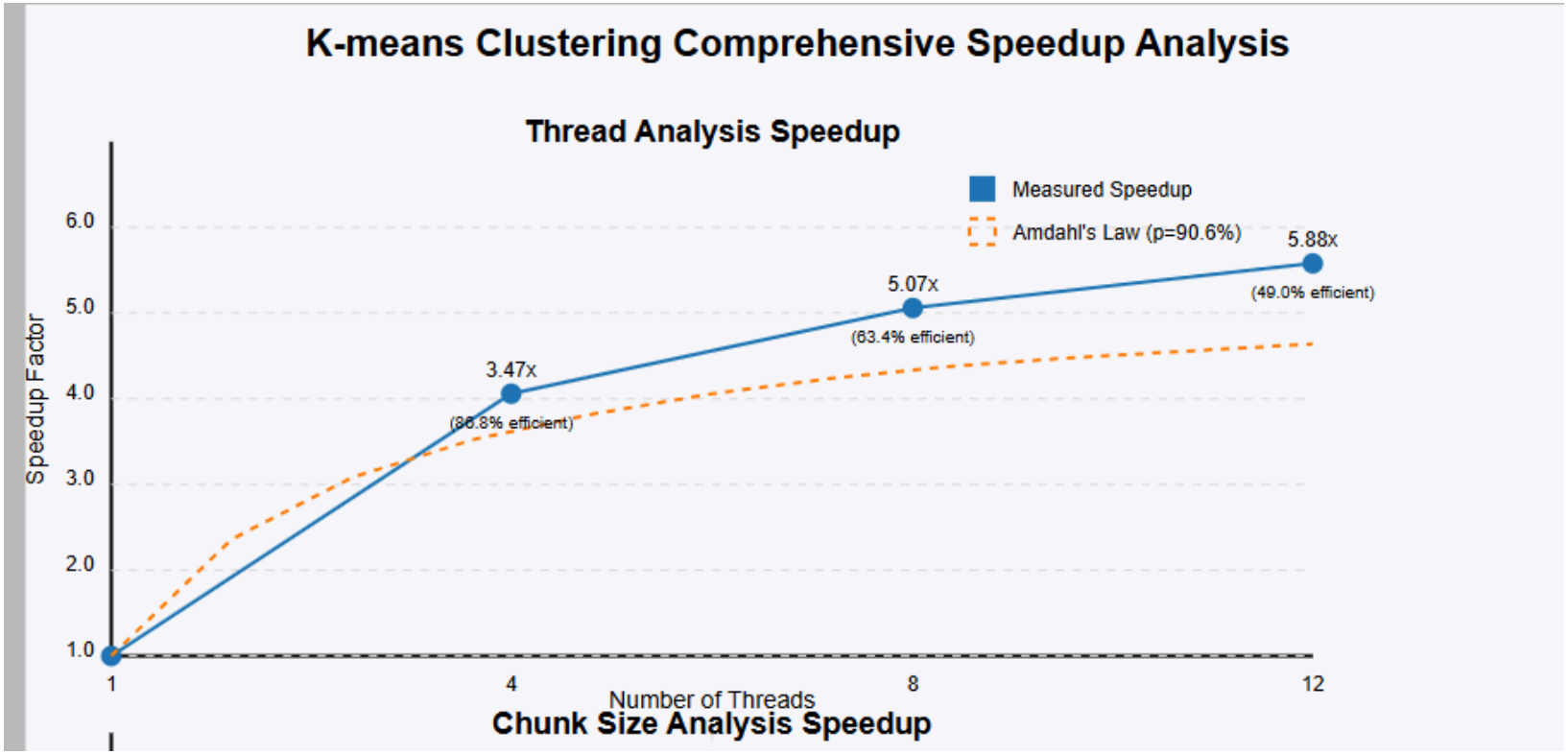
Dynamic Chunk Size Analysis:

- Smaller Chunks (2k): The performance is significantly good (0.5640 seconds). Dynamic scheduling excels with smaller chunks, as threads are able to grab new chunks as soon as they finish, keeping them busy and reducing idle times.
- Medium Chunks (20k): The performance starts to degrade slightly (1.0464 seconds). While dynamic scheduling still works well, the chunk sizes increase the overhead of task management, as the scheduler has to keep distributing tasks dynamically.
- Larger Chunks (100k): The execution time (0.6596 seconds) starts to improve, but the overhead of dynamically managing larger chunks may negate some of the benefits of parallelism.

Chunk Analysis (Static)			Chunk Analysis (Dynamic)		
10k	100k	500k	2k	20k	100k
0.5531	0.6516	2.6635	0.5640	1.0464	0.6396
0.6679	0.6909	1.7059	0.4796	0.8802	0.7058
0.6866	0.6931	1.6074	0.6343	1.0346	0.7944
0.5984	0.6873	1.6337	0.5301	0.5570	0.6891
0.7287	0.6759	1.6748	0.6494	0.5700	0.7544
0.7407	0.7501	1.6673	0.8296	0.7318	0.6231
0.2614	0.7043	1.6377	0.5631	0.5779	0.7194
0.6121	0.6302	1.5190	0.8492	0.6351	0.6939
0.5874	0.6317	1.6647	0.5133	0.6835	0.6911

5 Speedup Calculations


K-means Clustering Speedup Analysis
Measured Speedup per Execution
Static (Avg: 4.93x)
Dynamic (Avg: 4.79x)
Speedup Factor
Execution Number
Amdahl's Law Theoretical Maximum Speedup (91.1% Parallelized)
Amdahl's Law (p=0.911)
Measured Speedup
Theoretical limit: ~11.24x
Your measured speedup: 4.93x
Maximum Speedup
Number of Cores



Parallelization Percentage: It is approximately 90.6% parallelizable based on thread scaling analysis. Optimal Configuration: The best performance was achieved with dynamic scheduling and 2k chunk size (5.51x speedup). Thread Scaling: Increasing from 8 to 12 threads provided diminishing returns, improving speedup by only 16% while efficiency dropped from 63.4% to 49.0%. Chunk Size Impact:

For static scheduling: 100k chunks performed 8.6% better than 10k chunks For dynamic scheduling: 2k chunks performed 24.1% better than 20k chunks

Static vs. Dynamic: With optimal chunk sizes, dynamic scheduling outperforms static scheduling by approximately 4.4%. Theoretical Limit: According to Amdahl's Law, your implementation has a theoretical maximum speedup of approximately 10.64x with infinite cores. Efficiency Trade-off: As you scale to more threads, you gain speedup but lose efficiency, with 12 threads dropping below 50% efficiency.