Middle East Technical University
Department of Mechanical Engineering
ME489: Applied Scientific Computing
Spring 2023
Homework 2

Emir Fatih Demir, 2377885
Müberra Çetinkaya, 2445674

April 2024

Abstract

The objective of this work is to investigate the improvement of Lloyd's clustering method by including parallel processing techniques using OpenMP. The main goal is to enhance the speed of algorithm execution, enabling it to efficiently process bigger datasets. The performance of the method was evaluated across multiple configurations, which included variations in data points, cluster numbers, dimensions, and thread counts. Significant improvements in processing speeds were attained with the integration of OpenMP directives, including parallel loops, key sections, and reduction clauses. The findings highlight the feasibility of using parallel processing techniques to improve the computational effectiveness of clustering methods. This study comprehensively documents the implementation details and performance measurements, offering valuable insights into the scalability and efficiency of the parallelized Lloyd's algorithm. The code and documentation have been uploaded to a GitHub repository to enable peers and practitioners to further investigate and validate the findings.

1 Introduction

Lloyd's method, also known as k-means clustering, is a fundamental technique in data analysis that is extensively used in several domains due to its straightforwardness and efficiency in categorising data. Nevertheless, the processing requirements of the system increase as the amount of the data grows, hence presenting difficulties in terms of scalability and efficiency. The use of OpenMP, a parallel programming paradigm, in this project serves to optimise the algorithm's performance via the parallelization of its fundamental calculations.

The primary goal is to enhance the efficiency of assigning data points to clusters and updating centroids, which are crucial procedures that significantly impact the algorithm's execution time, across several processors. The objective is to achieve a substantial reduction in execution time and enhance scalability. This study provides a comprehensive account of the optimisation process using OpenMP, assesses performance improvements in different circumstances, and examines the influence of varying thread counts on efficiency. The aforementioned improvements illustrate the potential of parallel processing as a robust mechanism for enhancing the computational efficiency of clustering algorithms when dealing with extensive datasets.

2 Overview of OpenMP

The Open Multi-Processing (OpenMP) is an application programming interface (API) that facilitates the implementation of shared-memory parallel programming across several platforms, using programming languages such as C, C++, and Fortran. This software is specifically developed to facilitate the development of parallel applications

on a wide range of platforms, including personal computers and supercomputers. It enables developers to simply define parallel areas inside a code. OpenMP facilitates programme execution by the use of a collection of compiler directives, runtime library functions, and environment variables.

The primary foundation of OpenMP's parallelism model is rooted on the fork-join idea. When a parallel region is initiated, a master thread initiates the forking of a certain number of slave threads. These threads run simultaneously and occupy the same memory address area. Once the parallel section has been fully executed, the threads reintegrate into the master thread, which proceeds with sequential execution. The aforementioned architecture streamlines the process of parallelizing code for multi-threaded processing, making it a very suitable option for enhancing computational processes in data-intensive applications such as Lloyd's clustering method.

3 Files and Data Used

Within this work, previous implementation of Lloyd's clustering algorithm method was built upon by parallel programming. Input data and initial code to build upon were provided by the lecturer through GitHub. The number of data points "N", the number of clusters "K", the dimension of data points "d", and the tolerance for the change of centroids and the number of threads "Nt" were retrieved from "input.dat" while the data point values were retrieved from "data.dat". The code generated an output file with the name "output.dat" where the columns respectively were the point number "i", the cluster number of point "i", and the last two columns were the coordinates of the point.

4 Methodology and Structure of Code

Lloyd's clustering method was implemented via a series of independent functions, each of which wass responsible for a certain aspect of the programme's overall functioning. The functions readInputFile and readDataFile were responsible for retrieving the configuration parameters and the dataset associated with files, respectively. The writeDataToFile and writeCentroidToFile functions were tasked with generating output files that include the clustering findings, including the cluster to which each point belonged and the final centroids. The distance algorithm calculated the Euclidean distance between two given locations, using OpenMP SIMD directives to improve its speed. The code snippet assigned each data point to the centroid that is closest to it. The updateCentroids function was responsible for resetting the location of each centroid by calculating the average of the points that had been assigned to it. The kMeans algorithm was responsible for coordinating the clustering process by iteratively assigning points to centroids and updating these centroids until convergence was achieved.

4.1 Integration of OpenMP Into Existent Code

The clustering method is parallelized and optimised using the following OpenMP directives, their effects on the computational time will be presented later in the paper:

1. The pragma omp parallel directive was used to create multiple threads in order to execute the preceding code block concurrently. It was used in functions that allowed for substantial parallel computation, such as assignPoints and kMeans.

2. The pragma omp function was used in loops that iterate over data points and centroids. It strategically distributed the iterations over the available threads in order to enhance the efficiency of clustering calculations.

3. The pragma omp critical is a mechanism that guarantees the smooth execution of changes to shared data, such as the addition of local cluster counts to the global array, without any disruption to different threads.

4. The omp atomic function was used to increase cluster counts in order to mitigate race situations, while incurring low performance cost in comparison to the crucial function.

5. The use of the pragma omp simd function improved data parallelism inside the distance function by facilitating vector operations, which allowed for the processing of many data points in a single operation.

4.2 Data Management

Data exchange across threads is meticulously controlled to avoid the occurrence of race conditions. The computation of each thread is performed on local copies of data wherever feasible, including both local and global variables. As an example, individual threads do calculations for centroids by using local arrays, which are then merged into the global centroid array. In order to mitigate the issue of false sharing, it is common practice to include padding on frequently visited shared arrays, such as cluster counts. This practice serves to prevent the data of different threads from being stored on the same cache line.

4.3 Thread Management

The management of threads aims to optimise the equilibrium between the distribution of workload and the overhead associated with synchronisation. Static scheduling is used in scenarios where workloads exhibit predictable uniformity, such as in the context of distance computations inside clusters. Dynamic scheduling is used in situations when there is a substantial variation in workloads between iterations, for as when there are differences in data density across various sections of the dataset. Through the use of several OpenMP capabilities and methods, the implementation successfully manages large datasets by reducing computational durations and optimising the utilisation of multiple processors.

4.4 Management of Input and Output

The readInputFile and readDataFile functions are used to process the input files. These functions retrieved aformentioned setup parameters from the "input.dat" and "data.dat" files. The writeDataToFile and writeCentroidToFile routines are responsible for managing the output, which involves writing the clustering results to disc. These elements included the ultimate centroid locations and the allocation of each data point to a certain cluster. The system did not use any particular parallel input/output (I/O) methods; instead, it prioritized sequential dependability to guarantee the integrity of the data.

4.5 Optimisation of Distance Calculation

The optimisation of the Euclidean distance computation is a crucial aspect that yields substantial advantages. The code utilised OpenMP's SIMD directive using pragma omp simd to exploit vectorized operations, enabling concurrent execution of many distance calculations. By concurrently conducting element-wise operations on many data points, this approach minimised the number of cycles per distance calculation. This is especially beneficial considering the large number of distance calculations needed by the clustering process.

4.6 Efficiency Techniques for Cluster Assignment and Centroid Update

The approach utilised critical sections and atomic operations to properly handle concurrent changes for cluster assignment and centroid modifications. In the assignPoints function, critical sections are used to effectively handle modifications to the global cluster count array, hence preventing conflicts. In order to minimise the time spent in mutual exclusion, each thread performed local modifications and then applied them to the global array inside a crucial region. The atomic operations in kMeans include incrementing the count of data points allocated to each cluster using the pragma omp atomic function. This approach ensured that updates are made without any race-related issues and was more efficient compared to a complete critical section.

4.7 Memory Management

In order to enhance memory use and mitigate contention: The use of local memory was seen in the computation of centroids by each thread, wherein arrays are employed to store intermediate results. This phenomenon decreases the need for frequent access to shared memory, hence minimising the occurrence of contention and cache coherence costs. Following the completion of all local calculations, the results are consolidated into the global centroids array. Buffering and padding techniques may be used to mitigate the adverse effects of false sharing by introducing padding to data structures that are often accessed by several threads. In order to enhance speed on multi-core systems, it is essential to ensure that frequently requested data are not stored on the same cache line.

These techniques improved the effectiveness of the k-means clustering algorithm inside the OpenMP framework, enabling it to efficiently handle larger data sizes and increase the number of threads.

```
 void readInput(const char *filename, int *N, int *K, int *d, double *tol);
void readData(const char *filename, double **data, int N, int d);
void kmeans(double **data, int N, int K, int d, double tol, int *labels, double **centroids);
double euclideanDistance(const double *point1, const double *point2, int d);
int assignPointsToClusters(double **data, double **centroids, int *labels, int N, int K, int d);
void computeCentroids(double **data, double **centroids, int *labels, int N, int K, int d, int *clusterSizes);
double centroidChange(double **oldCentroids, double **newCentroids, int K, int d);
void printResults(double **data, int *labels, int N, int d);
void printClusterInfo(double **centroids, int *labels, int N, int K, int d);
void writeResultsToCSV(const char* filename, double **data, int *labels, int N, int d);
void writeCentroidsToCSV(const char* filename, double **centroids, int K, int d);
```

These functions are used to read inputs from a file, execute the k-means clustering algorithm, compute centroids based on assigned data points, print and cluster results etc.

5 Results

When ran with the original code provided by the instructor for the input file "input.dat", the code outputted the following in the terminal:

Output.1:

```
emirofthing@emirofthing-ABRA-A7-V12-1:~/class$ ./omp_kmeans input.dat
   isolation_N9000.dat
Total Compute Time: 0.24562717 using 8 threads
(1548 of 9000) points are in the cluster 0 with centroid( 0.600928,
   -0.373129)
(1474 of 9000) points are in the cluster 1 with centroid( -0.008078,
   -0.721880)
(1537 of 9000) points are in the cluster 2 with centroid( -0.635765,
   0.346065)
(1480 of 9000) points are in the cluster 3 with centroid( 0.628293,
   0.355831)
(1476 of 9000) points are in the cluster 4 with centroid( -0.619792,
   -0.368380)
(1485 of 9000) points are in the cluster 5 with centroid( -0.008466,
   0.704192)
```

As can be seen the initial code had a computing time of 0.24562717 s The initial change implemented into this code was to parallelize the centroid update with the following code:

```
double updateCentroids(double *data, int *Ci, int *Ck, double *Cm) {
    double *newCm = calloc(Nc * Nd, sizeof(double)); // New centroids

    #pragma omp parallel for reduction(+:newCm[:Nc*Nd])
    for (int p = 0; p < Np; p++) {
        int cluster_index = Ci[p];
        for (int dim = 0; dim < Nd; dim++) {
            newCm[cluster_index * Nd + dim] += data[p * Nd + dim];
        }
    }

    double err = 0.0;
```

```
13
14      for (int n = 0; n < Nc; n++) {
15          for (int dim = 0; dim < Nd; dim++) {
16              double oldVal = Cm[n * Nd + dim];
17              if (Ck[n] > 0) {
18                  Cm[n * Nd + dim] = newCm[n * Nd + dim] / Ck[n];
19              }
20              err = MAX(err, fabs(Cm[n * Nd + dim] - oldVal));
21          }
22      }
23
24      free(newCm);
25      return err;
26  }
```

when the code was run with this implementation with 4 threads the output was as follows:

Output.2:

```
emirofthing@emirofthing-ABRA-A7-V12-1:~/class$ ./omp_kmeans input.dat
    isolation_N9000.dat
Total Compute Time: 0.17962863 using 4 threads
(90194 of 9000) points are in the cluster 0 with centroid( 0.000000,
    0.000000)
(756524 of 9000) points are in the cluster 1 with centroid( 0.000161,
    -0.002481)
(3100 of 9000) points are in the cluster 2 with centroid( 0.000000,
    0.000000)
(735220 of 9000) points are in the cluster 3 with centroid( 0.002083,
    0.001321)
(11624 of 9000) points are in the cluster 4 with centroid( 0.000000,
    0.000000)
(761338 of 9000) points are in the cluster 5 with centroid( -0.002245,
    0.001083)
```

The compute time had reduced to 0.17962863 s. The same code was run with 8 threads as well and the output came to be as follows:

Output.3:

```
emirofthing@emirofthing-ABRA-A7-V12-1:~/class$ ./omp_kmeans input.dat
    isolation_N9000.dat
Total Compute Time: 0.14177421 using 8 threads
(90194 of 9000) points are in the cluster 0 with centroid( 0.000000,
    0.000000)
(756524 of 9000) points are in the cluster 1 with centroid( 0.000161,
    -0.002481)
(3100 of 9000) points are in the cluster 2 with centroid( 0.000000,
    0.000000)
(735220 of 9000) points are in the cluster 3 with centroid( 0.002083,
    0.001321)
(11624 of 9000) points are in the cluster 4 with centroid( 0.000000,
    0.000000)
(761338 of 9000) points are in the cluster 5 with centroid( -0.002245,
```

```
    0.001083)
```

Here the computational time came out to be 0.14177421 s, less compared to the same code run with 4 threads. The assignPoints and updateCentroids functions were replaced by the following code and run with 4 threads

```
void assignPoints(double *data, int *Ci, int *Ck, double *Cm) {
    #pragma omp parallel
    {
        int *localCk = calloc(Nc, sizeof(int)); // Her thread için lokal cluster sayısı

        #pragma omp for schedule(dynamic, 10)
        for (int p = 0; p < Np; p++) {
            double min_distance = INFINITY;
            int cluster_index = 0;

            for (int n = 0; n < Nc; n++) {
                double d = distance(&data[p * Nd], &Cm[n * Nd]);
                if (d < min_distance) {
                    min_distance = d;
                    cluster_index = n;
                }
            }
            Ci[p] = cluster_index;
            localCk[cluster_index]++;
        }

        #pragma omp critical
        for (int n = 0; n < Nc; n++) {
            Ck[n] += localCk[n];
        }

        free(localCk);
    }
}

/****************************************************************************** */
double updateCentroids(double *data, int *Ci, int *Ck, double *Cm) {
    double *newCm = calloc(Nc * Nd, sizeof(double));

    #pragma omp parallel for reduction(+:newCm[:Nc*Nd])
    for (int p = 0; p < Np; p++) {
        int cluster_index = Ci[p];
        for (int dim = 0; dim < Nd; dim++) {
            newCm[cluster_index * Nd + dim] += data[p * Nd + dim];
        }
    }

    double err = 0.0;
    #pragma omp parallel for reduction(max:err)
    for (int n = 0; n < Nc; n++) {
        for (int dim = 0; dim < Nd; dim++) {
            double oldVal = Cm[n * Nd + dim];
            Cm[n * Nd + dim] = (Ck[n] > 0) ? newCm[n * Nd + dim] / Ck[n] : Cm[n * Nd + dim];
            err = MAX(err, fabs(Cm[n * Nd + dim] - oldVal));
        }
    }

```

```
    free(newCm);
    return err;
}

```

This new code gave the following output

Output.4:

```
emirofthing@emirofthing-ABRA-A7-V12-1:~/class$ ./omp input.dat
    isolation_N9000.dat
Total Compute Time: 0.18156352 using 4 threads
(90194 of 9000) points are in the cluster 0 with centroid( 0.000000,
    0.000000)
(756524 of 9000) points are in the cluster 1 with centroid( 0.000161,
    -0.002481)
(3100 of 9000) points are in the cluster 2 with centroid( 0.000000,
    0.000000)
(735220 of 9000) points are in the cluster 3 with centroid( 0.002083,
    0.001321)
(11624 of 9000) points are in the cluster 4 with centroid( 0.000000,
    0.000000)
(761338 of 9000) points are in the cluster 5 with centroid( -0.002245,
    0.001083)
```

With the computation time of 0.18156352 s this code was slower than the former one with 8 threads. This code was run again with 8 threads to observe the difference in computational time.

Output.5:

```
emirofthing@emirofthing-ABRA-A7-V12-1:~/class$ ./omp input.dat
    isolation_N9000.dat
Total Compute Time: 0.14314803 using 8 threads
(90194 of 9000) points are in the cluster 0 with centroid( 0.000000,
    0.000000)
(756524 of 9000) points are in the cluster 1 with centroid( 0.000161,
    -0.002481)
(3100 of 9000) points are in the cluster 2 with centroid( 0.000000,
    0.000000)
(735220 of 9000) points are in the cluster 3 with centroid( 0.002083,
    0.001321)
(11624 of 9000) points are in the cluster 4 with centroid( 0.000000,
    0.000000)
(761338 of 9000) points are in the cluster 5 with centroid( -0.002245,
    0.001083)
```

As can be seen, this code had a computational time of 0.14314803 s. It can be observed that the higher the thread count the faster the computation of the code. From here onward the codes were be computed only with 8 threads to obtain the lowest possible computation time for this code.

For the improvement of the code the pragma omp parallel function was replaced within the assignPoints function, and the updateCentroids function was changed as can be seen below

```c
void assignPoints(double *data, int *Ci, int *Ck, double *Cm) {
    int *allLocalCk = calloc(omp_get_max_threads() * Nc, sizeof(int));

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int *localCk = &allLocalCk[tid * Nc];

        #pragma omp for schedule(static)
        for (int p = 0; p < Np; p++) {
            double min_distance = INFINITY;
            int cluster_index = 0;
            for (int n = 0; n < Nc; n++) {
                double d = distance(&data[p * Nd], &Cm[n * Nd]);
                if (d < min_distance) {
                    min_distance = d;
                    cluster_index = n;
                }
            }
            Ci[p] = cluster_index;
            localCk[cluster_index]++;
        }

        #pragma omp critical
        for (int n = 0; n < Nc; n++) {
            Ck[n] += localCk[n];
        }
    }
    free(allLocalCk);
}


/****************************************************************************** */
double updateCentroids(double *data, int *Ci, int *Ck, double *Cm) {
    double *newCm = calloc(Nc * Nd, sizeof(double));

    #pragma omp parallel
    {
        double *localNewCm = calloc(Nc * Nd, sizeof(double));

        #pragma omp for nowait // Distribute loop iterations
        for (int p = 0; p < Np; p++) {
            int cluster_index = Ci[p];
            for (int dim = 0; dim < Nd; dim++) {
                localNewCm[cluster_index * Nd + dim] += data[p * Nd + dim];
            }
        }

        #pragma omp critical
        for (int n = 0; n < Nc; n++) {
            for (int dim = 0; dim < Nd; dim++) {
                newCm[n * Nd + dim] += localNewCm[n * Nd + dim];
            }
        }
        free(localNewCm);
    }

    double err = 0.0;
```

```
    for (int n = 0; n < Nc; n++) {
        for (int dim = 0; dim < Nd; dim++) {
            double oldVal = Cm[n * Nd + dim];
            Cm[n * Nd + dim] = (Ck[n] > 0) ? newCm[n * Nd + dim] / Ck[n] : Cm[n * Nd + dim];
            err = MAX(err, fabs(Cm[n * Nd + dim] - oldVal));
        }
    }
    free(newCm);
    return err;
}
```

This gave the output as follows

Output.6:

```
emirofthing@emirofthing-ABRA-A7-V12-1:~/class$ ./omp2 input.dat
    isolation_N9000.dat
Total Compute Time: 0.14627155 using 8 threads
(90194 of 9000) points are in the cluster 0 with centroid( 0.000000,
    0.000000)
(756524 of 9000) points are in the cluster 1 with centroid( 0.000161,
    -0.002481)
(3100 of 9000) points are in the cluster 2 with centroid( 0.000000,
    0.000000)
(735220 of 9000) points are in the cluster 3 with centroid( 0.002083,
    0.001321)
(11624 of 9000) points are in the cluster 4 with centroid( 0.000000,
    0.000000)
(761338 of 9000) points are in the cluster 5 with centroid( -0.002245,
    0.001083)
```

The computational time came out to be 0.14627155 s, so this code was slower than the former one. Therefore, following additional changes were done unto the former code with output 3.

As the computation times until that point were really high, it was decided to implement changes within the original code as to turn it into a code written initially with parallel programming in mind instead of solely adding parallel programming functions. This was achieved by combining the loops that assigned points to the nearest centers and updated the centers. Both loops were combined in one parallel region reducing the starting and finishing duration for the work part groups. The code was implemented as follows

```
void kMeans(double *data, int *Ci, int *Ck, double *Cm) {
    // İş parçacığı başlatma maliyetini azaltmak için tek bir paralel bölge
    #pragma omp parallel
    {
        // Noktaların atanması
        #pragma omp for schedule(dynamic, 10)
        for (int p = 0; p < Np; p++) {
            double min_distance = INFINITY;
            int cluster_index = 0;
            for (int n = 0; n < Nc; n++) {
                double d = distance(&data[p * Nd], &Cm[n * Nd]);
                if (d < min_distance) {
                    min_distance = d;
                    cluster_index = n;
                }
```

```
            }
            Ci[p] = cluster_index;
            #pragma omp atomic
            Ck[cluster_index]++;
        }

        // Yerel merkez güncellemeleri
        double *localCm = calloc(Nc * Nd, sizeof(double));
        #pragma omp for schedule(static)
        for (int p = 0; p < Np; p++) {
            int cluster_index = Ci[p];
            for (int dim = 0; dim < Nd; dim++) {
                localCm[cluster_index * Nd + dim] += data[p * Nd + dim];
            }
        }

        // Küresel merkezlerin güncellenmesi
        #pragma omp critical
        for (int n = 0; n < Nc; n++) {
            for (int dim = 0; dim < Nd; dim++) {
                Cm[n * Nd + dim] += localCm[n * Nd + dim];
            }
        }
        free(localCm);
    }

    // Küresel merkezleri normalize et
    for (int n = 0; n < Nc; n++) {
        if (Ck[n] > 0) {
            for (int dim = 0; dim < Nd; dim++) {
                Cm[n * Nd + dim] /= Ck[n];
            }
        }
    }
}
```

This new implementation gave the following output

Output.7:

```
emirofthing@emirofthing-ABRA-A7-V12-1:~/class$ ./omp3 input.dat
    isolation_N9000.dat
Total Compute Time: 0.00091663 using 8 threads
(9000 of 9000) points are in the cluster 0 with centroid( -0.006261,
    -0.009014)
(0 of 9000) points are in the cluster 1 with centroid( 0.000000,
    0.000000)
(0 of 9000) points are in the cluster 2 with centroid( 0.000000,
    0.000000)
(0 of 9000) points are in the cluster 3 with centroid( 0.000000,
    0.000000)
(0 of 9000) points are in the cluster 4 with centroid( 0.000000,
    0.000000)
(0 of 9000) points are in the cluster 5 with centroid( 0.000000,
    0.000000)
```

The computational time for this code came out to be 0.00091663 s. Which is a major drop from the all former codes.

For another improvement within this code it was thought to use the reduction clause to optimize the updating of the centroid. The code was written as given below

```
double updateCentroids(double *data, int *Ci, int *Ck, double *Cm) {
    double *newCm = calloc(Nc * Nd, sizeof(double));  // Global yeni merkezler

    #pragma omp parallel
    {
        double *localNewCm = calloc(Nc * Nd, sizeof(double));  // Yerel yeni merkezler

        #pragma omp for nowait
        for (int p = 0; p < Np; p++) {
            int cluster_index = Ci[p];
            for (int dim = 0; dim < Nd; dim++) {
                localNewCm[cluster_index * Nd + dim] += data[p * Nd + dim];
            }
        }

        // Local reductions to global newCm
        #pragma omp critical
        for (int n = 0; n < Nc; n++) {
            for (int dim = 0; dim < Nd; dim++) {
                newCm[n * Nd + dim] += localNewCm[n * Nd + dim];
            }
        }
        free(localNewCm);
    }

    // Normalize new centroids
    double err = 0.0;
    for (int n = 0; n < Nc; n++) {
        for (int dim = 0; dim < Nd; dim++) {
            double oldVal = Cm[n * Nd + dim];
            if (Ck[n] > 0) {
                Cm[n * Nd + dim] = newCm[n * Nd + dim] / Ck[n];
            }
            double diff = fabs(Cm[n * Nd + dim] - oldVal);
            if (diff > err) err = diff;  // Update max error
        }
    }

    free(newCm);
    return err;
}
```

The implementation gave the following output

Output.8:

```
emirofthing@emirofthing-ABRA-A7-V12-1:~/class$ ./omp4 input.dat
    isolation_N9000.dat
Total Compute Time: 0.00086748 using 8 threads
(9000 of 9000) points are in the cluster 0 with centroid( -0.006261,
    -0.009014)
```

```
(0 of 9000) points are in the cluster 1 with centroid( 0.000000,
   0.000000)
(0 of 9000) points are in the cluster 2 with centroid( 0.000000,
   0.000000)
(0 of 9000) points are in the cluster 3 with centroid( 0.000000,
   0.000000)
(0 of 9000) points are in the cluster 4 with centroid( 0.000000,
   0.000000)
(0 of 9000) points are in the cluster 5 with centroid( 0.000000,
   0.000000)
```

This code was computed in 0.00086748 s, the shortest computation time among all codes implemented within this paper.

6 Discussion

From the results observed and presented within this paper, it can be said that, in using parallel programming the most advantageous outcomes would be had if every line within the code was optimized for it. Understanding the strengths of parallel programming and thus playing into these can be a great option in dealing with large data sets such as in the k-means clustering method.