

CSE4001 – PARALLEL AND DISTRIBUTIVE COMPUTING

**Embedded Project**

# FALL SEMESTER – 2022~2023

K Means Clustering Algorithm using Openmp

Submitted By:

Krishna Prasad Y V S Purama 20BCE1421

Santhosh Narayanan B 20BCE1309

Priyansh Chauhan 20BCE1374

Under the guidance of Prof. Kumar R

**Index:**

1. [Abstract](#_bookmark5) 3
2. [Introduction](#_bookmark6) 4
3. [Motivation 6](#_bookmark0)
4. [Code Implementation](#_bookmark10) 7
5. Output Screenshots 24
6. Result Analysis 28
7. Conclusion 30
8. References 31

**2. Abstract**

Clustering is the process of organizing a collection of items into groups, or clusters, where the objects are more similar (in some sense) to one another than to those in other clusters. Garbage can be divided into biodegradable and non-biodegradable waste, as an example for laypeople. Both academic and commercial applications require clustering analysis. The K-means algorithm is a well-liked partitioning method in clustering. Given that the dataset's size grows quickly, using K-means to handle massive volumes of data is difficult. The parallel K-means algorithm is recommended as part of the clustering procedure.

Parallel and Distributed computing is the future of technology. All products and their fundamental concepts are being shifted to a parallel computing model. Everybody would agree that serial computing is easy to implement and use, but simply not efficient enough for industry-level purposes.

So, in this project, we are going to implement the k-means algorithm in parallel and serial implementation. In the serial, form we would implement the algorithm in the traditional way whereas for parallel implementation the clustering algorithm requires massive computations, with the distance between each data point and each centroid being calculated. Since the calculation of the appropriate centroid for each data point is independent of the others, the algorithm provides a good scope for parallelism. However, there is a bottleneck. The threads need to communicate among themselves to keep the centroid values updated, as more than one thread might try to access the same centroid point. In that case, it is imperative to ensure that both threads do not try to modify the centroid at the same time, as it might result in corrupted values and false sharing.

**3. Introduction**

K-Means clustering is an unsupervised learning algorithm that groups unlabelled datasets into distinct clusters. where K defines the number of defined clusters the process should create. If K=2, then there are 2 clusters, if K=3, then there are 3 clusters, and so on.

This is an iterative algorithm that divides the unlabelled dataset into k distinct clusters such that each dataset belongs to only one group with similar properties.  groups data into different groups, allowing a convenient way to independently discover categories of groups in unlabelled datasets without the need for training.

This is a centroid-based algorithm where each cluster is associated with a centroid. The main goal of this algorithm is to minimize the sum of distances between data points and their corresponding clusters. The algorithm takes an unlabelled data set as input, divides the data set into k clusters, and repeats the process until no better cluster is found. This algorithm requires that the value of k is predetermined.

The k-means clustering algorithm performs two main tasks.:

Through an iterative process, determine the best values ​​for the K centres or centroids.

Assign each data point to the closest K-Center. Data points close to each K-center form a cluster.

Therefore, each cluster has data points with some similarities and distances from other clusters.

The below diagram explains the working of the K-means Clustering Algorithm:



The distance between the sample and cluster centre is the main objective of the K-means algorithm. The complexity of these two halves is O(nkt) and O(nt), respectively. The challenge of distance computation will increase with the expansion of datasets. In this project, a Master/Slave model and a data-parallel method are used. The host, which is in charge of dataset distribution, load adjustment, aggregating the cluster results after each iterative process, and creating new cluster centres, run the master/slave strategy.

Data transfer between processors is accomplished using the PVM's message transfer procedure. Data distribution automatically adjusts to maintain a dynamic load balance. We take into account the fundamental properties of K-means and the various processing capabilities. Data deflection and idle processors may result from static division data. The available technique is to apply for the next sub-dataset from the master that is the same size once the slave has finished computing for the sub-dataset that was assigned, until there is no more data that needs to be allocated.

**Header Files Used : -**

**#include <omp.h>**

·       It is a library that allows memory multiprocessing programming in C.

**#include <stdlib.h>**

stdlib.h is the header of the general purpose standard library of C programming language which includes functions involving memory allocation, process control, conversions and others. It is compatible with C++ and is known as cstdlib in C++. The name "stdlib" stands for "standard library"

**#include <stdio.h>**

The C programming language provides many standard library functions for file input and output. These functions make up the bulk of the C standard library

**#include <limits.h>**

The limits.h header determines various properties of the various variable types. The macros defined in this header, limits the values of various variable types like char, int and long.

**#include <float.h>**

The float.h header file of the C Standard Library contains a set of various platform-dependent constants related to floating point values. These constants are proposed by ANSI C. They allow making more portable programs.

**#include <math.h>**

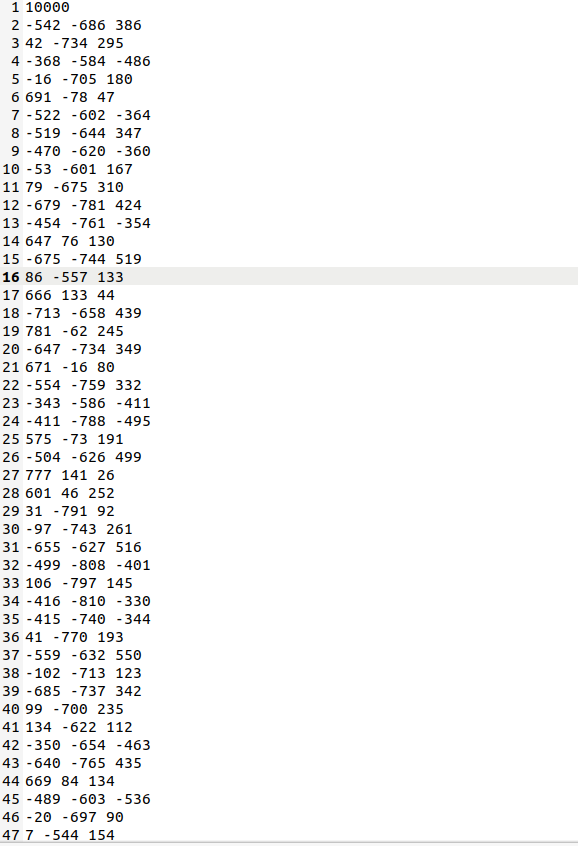
The float.h header file of the C Standard Library contains a set of various platform-dependent constants related to floating point values. These constants are proposed by ANSI C. They allow making more portable programs.

**#include <assert.h>**

The assert.h header file of the C Standard Library provides a macro called assert which can be used to verify assumptions made by the program and print a diagnostic message if this assumption is false.

**#include <string.h>**

The string.h header defines one variable type, one macro, and various functions for manipulating arrays of characters.



**4. Motivation**

As time passes, the world is becoming more and more oriented towards Parallel Computing. Many of the tasks that were once carried out sequentially are now being carried out in parallel so as to use resources more efficiently and get faster results.

Our motivation to work on this algorithm is to apply it in the medical field to society through Computer Science. Paul and Hoque (2010) applied a k-means mean clustering algorithm to a medical dataset to predict disease probabilities. The probability of having disease within a cluster is defined as the number of patients with disease divided by the total number of points in the cluster. That is the probability of finding a disease within a cluster. The mean probability of all clusters is the true probability of the disease in the data and can be found by brute force methods. Accuracy is the ratio of mean probability to actual probability. Experimental results show that the algorithm achieves approximately 95% accuracy when applied to the University of California, Irvine (UCI) Machine Learning Repository Zoo dataset and diabetes dataset.

# Code Implementation:

# For Sequential:

#include <omp.h>

#include <stdlib.h>

#include <stdio.h>

#include <time.h>

#include <limits.h>

#include <float.h>

#include <math.h>

#include <assert.h>

#include <string.h>

#define MAX\_ITER 100

#define THRESHOLD 1e-6

#define min(a, b) \

    ({ \_\_typeof\_\_ (a) \_a = (a); \

       \_\_typeof\_\_ (b) \_b = (b); \

     \_a < \_b ? \_a : \_b; })

int number\_of\_points\_global;

int number\_of\_iterations\_global;

double delta\_global = THRESHOLD + 1;

int K\_global;

int \*data\_points\_global;

float \*iter\_centroids\_global;

int \*data\_point\_cluster\_global;

void kmeans\_sequential\_execution()

{

    printf("Sequential k-means start\n");

    int i = 0, j = 0;

    double min\_dist, current\_dist;

    // Cluster id associated with each point

    int \*point\_to\_cluster\_id = (int \*)malloc(number\_of\_points\_global \* sizeof(int));

    // Cluster location or centroid (x,y,z) coordinates for K clusters in a iteration

    float \*cluster\_points\_sum = (float \*)malloc(K\_global \* 3 \* sizeof(float));

    // No. of points in a cluster for a iteration

    int \*points\_inside\_cluster\_count = (int \*)malloc(K\_global \* sizeof(int));

    // Start of loop

    int iter\_counter = 0;

    double temp\_delta = 0.0;

    while ((delta\_global > THRESHOLD) && (iter\_counter < MAX\_ITER)) //+1 is for the last assignment to cluster centroids (from previous iter)

    {

        // Initialize cluster\_points\_sum or centroid to 0.0

        for (i = 0; i < K\_global \* 3; i++)

            cluster\_points\_sum[i] = 0.0;

        // Initialize number of points for each cluster to 0

        for (i = 0; i < K\_global; i++)

            points\_inside\_cluster\_count[i] = 0;

        for (i = 0; i < number\_of\_points\_global; i++)

        {

            //Assign these points to their nearest cluster

            min\_dist = DBL\_MAX;

            for (j = 0; j < K\_global; j++)

            {

                current\_dist = pow((double)(iter\_centroids\_global[(iter\_counter \* K\_global + j) \* 3] - (float)data\_points\_global[i \* 3]), 2.0) +

                               pow((double)(iter\_centroids\_global[(iter\_counter \* K\_global + j) \* 3 + 1] - (float)data\_points\_global[i \* 3 + 1]), 2.0) +

                               pow((double)(iter\_centroids\_global[(iter\_counter \* K\_global + j) \* 3 + 2] - (float)data\_points\_global[i \* 3 + 2]), 2.0);

                if (current\_dist < min\_dist)

                {

                    min\_dist = current\_dist;

                    point\_to\_cluster\_id[i] = j;

                }

            }

             //Update local count of number of points inside cluster

            points\_inside\_cluster\_count[point\_to\_cluster\_id[i]] += 1;

            // Update local sum of cluster data points

            cluster\_points\_sum[point\_to\_cluster\_id[i] \* 3] += (float)data\_points\_global[i \* 3];

            cluster\_points\_sum[point\_to\_cluster\_id[i] \* 3 + 1] += (float)data\_points\_global[i \* 3 + 1];

            cluster\_points\_sum[point\_to\_cluster\_id[i] \* 3 + 2] += (float)data\_points\_global[i \* 3 + 2];

        }

        //Compute centroid from cluster\_points\_sum and store inside iter\_centroids\_global in a iteration

        for (i = 0; i < K\_global; i++)

        {

            assert(points\_inside\_cluster\_count[i] != 0);

            iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3] = cluster\_points\_sum[i \* 3] / points\_inside\_cluster\_count[i];

            iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 1] = cluster\_points\_sum[i \* 3 + 1] / points\_inside\_cluster\_count[i];

            iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 2] = cluster\_points\_sum[i \* 3 + 2] / points\_inside\_cluster\_count[i];

        }

    /\*

        Delta is the sum of squared distance between centroid of previous and current iteration.

        Supporting formula is:

            delta = (iter1\_centroid1\_x - iter2\_centroid1\_x)^2 + (iter1\_centroid1\_y - iter2\_centroid1\_y)^2 + (iter1\_centroid1\_z - iter2\_centroid1\_z)^2 + (iter1\_centroid2\_x - iter2\_centroid2\_x)^2 + (iter1\_centroid2\_y - iter2\_centroid2\_y)^2 + (iter1\_centroid2\_z - iter2\_centroid2\_z)^2

        Update delta\_global with new delta

    \*/

        temp\_delta = 0.0;

        for (i = 0; i < K\_global; i++)

        {

            temp\_delta += (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3]) \* (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3]) + (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 1] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3 + 1]) \* (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 1] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3 + 1]) + (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 2] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3 + 2]) \* (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 2] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3 + 2]);

        }

        delta\_global = temp\_delta;

        iter\_counter++;

    }

    // Store the number of iterations performed in global variable

    number\_of\_iterations\_global = iter\_counter;

    // Assign points to final choice for cluster centroids

    for (i = 0; i < number\_of\_points\_global; i++)

    {

        // Assign points to clusters

        data\_point\_cluster\_global[i \* 4] = data\_points\_global[i \* 3];

        data\_point\_cluster\_global[i \* 4 + 1] = data\_points\_global[i \* 3 + 1];

        data\_point\_cluster\_global[i \* 4 + 2] = data\_points\_global[i \* 3 + 2];

        data\_point\_cluster\_global[i \* 4 + 3] = point\_to\_cluster\_id[i];

        assert(point\_to\_cluster\_id[i] >= 0 && point\_to\_cluster\_id[i] < K\_global);

    }

}

void kmeans\_sequential(int N,

                    int K,

                    int\* data\_points,

                    int\*\* data\_point\_cluster\_id,

                    float\*\* iter\_centroids,

                    int\* num\_iterations

                    )

{

    // Initialize global variables

    number\_of\_points\_global = N;

    number\_of\_iterations\_global = \*num\_iterations;

    K\_global = K;

    data\_points\_global = data\_points;

    //Allocating space of 4 units each for N data points

    \*data\_point\_cluster\_id = (int \*)malloc(N \* 4 \* sizeof(int));

    data\_point\_cluster\_global = \*data\_point\_cluster\_id;

    /\*

        Allocating space of 3K units for each iteration

        Since three dimensional data point and K number of clusters

    \*/

    iter\_centroids\_global = (float \*)calloc((MAX\_ITER + 1) \* K \* 3, sizeof(float));

    // Assign first K points to be initial centroids

    int i = 0;

    for (i = 0; i < K; i++)

    {

        iter\_centroids\_global[i \* 3] = data\_points[i \* 3];

        iter\_centroids\_global[i \* 3 + 1] = data\_points[i \* 3 + 1];

        iter\_centroids\_global[i \* 3 + 2] = data\_points[i \* 3 + 2];

    }

    // Print initial centroids

    for (i = 0; i < K; i++)

    {

        printf("initial centroid #%d: %f,%f,%f\n", i + 1, iter\_centroids\_global[i \* 3], iter\_centroids\_global[i \* 3 + 1], iter\_centroids\_global[i \* 3 + 2]);

    }

    // Run k-means sequential function

    kmeans\_sequential\_execution();

    // Record number of iterations and store iter\_centroids\_global data into iter\_centroids

    \*num\_iterations = number\_of\_iterations\_global;

    int centroids\_size = (\*num\_iterations + 1) \* K \* 3;

    printf("number of iterations:%d\n", number\_of\_iterations\_global);

    \*iter\_centroids = (float \*)calloc(centroids\_size, sizeof(float));

    for (i = 0; i < centroids\_size; i++)

    {

        (\*iter\_centroids)[i] = iter\_centroids\_global[i];

    }

    // Print final centroids

    for (i = 0; i < K; i++)

    {

        printf("centroid #%d: %f,%f,%f\n", i + 1, (\*iter\_centroids)[((\*num\_iterations) \* K + i) \* 3], (\*iter\_centroids)[((\*num\_iterations) \* K + i) \* 3 + 1], (\*iter\_centroids)[((\*num\_iterations) \* K + i) \* 3 + 2]);

    }

}

void dataset\_in(const char \*dataset\_filename, int \*N, int \*\*data\_points) //

{

    FILE \*fin = fopen(dataset\_filename, "r");

    fscanf(fin, "%d", N);

    \*data\_points = (int \*)malloc(sizeof(int) \* ((\*N) \* 3));

    int i = 0;

    for (i = 0; i < (\*N) \* 3; i++)

    {

        fscanf(fin, "%d", (\*data\_points + i));

    }

    fclose(fin);

}

void clusters\_out(const char \*cluster\_filename, int N, int \*cluster\_points)

{

    FILE \*fout = fopen(cluster\_filename, "w");

    int i = 0;

    for (i = 0; i < N; i++)

    {

        fprintf(fout, "%d %d %d %d\n",

                \*(cluster\_points + (i \* 4)), \*(cluster\_points + (i \* 4) + 1),

                \*(cluster\_points + (i \* 4) + 2), \*(cluster\_points + (i \* 4) + 3));

    }

    fclose(fout);

}

void centroids\_out(const char \*centroid\_filename, int K, int number\_of\_iterations, float \*iter\_centroids)

{

    FILE \*fout = fopen(centroid\_filename, "w");

    int i = 0;

    for (i = 0; i < number\_of\_iterations + 1; i++)

    {

        int j = 0;

        for (j = 0; j < K; j++)

        {

            fprintf(fout, "%f %f %f, ",

                    \*(iter\_centroids + (i \* K + j) \* 3),         //x coordinate

                    \*(iter\_centroids + (i \* K + j) \* 3 + 1),  //y coordinate

                    \*(iter\_centroids + (i \* K + j) \* 3 + 2)); //z coordinate

        }

        fprintf(fout, "\n");

    }

    fclose(fout);

}

int main()

{

    //---------------------------------------------------------------------

    int N;                  // Number of data points (input)

    int K;                  //Number of clusters to be formed (input)

    int\* data\_points;       //Data points (input)

    int\* cluster\_points;    //clustered data points (to be computed)

    float\* iter\_centroids;      //centroids of each iteration (to be computed)

    int number\_of\_iterations;     //no of iterations performed by algo (to be computed)

    //---------------------------------------------------------------------

    double start\_time, end\_time;

    double computation\_time;

    printf("Enter No. of Clusters: ");

    scanf("%d", &K);

    printf("\nFollowing files should be in the same directory as of program\n");

    printf("1 for 10000 datapoints\n");

    printf("2 for 50000 datapoints\n");

    printf("3 for 100000 datapoints\n");

    printf("4 for 200000 datapoints\n");

    printf("5 for 400000 datapoints\n");

    printf("6 for 500000 datapoints\n");

    printf("7 for 600000 datapoints\n");

    printf("8 for 800000 datapoints\n");

    printf("9 for 1000000 datapoints\n");

    printf("\nEnter the number of dataset file to input: ");

    int x;

    scanf("%d",&x);

    char \*dataset\_filename = "dataset-10000.txt";

    switch (x)

    {

    case 1:

        dataset\_filename = "dataset-10000.txt";

        break;

    case 2:

        dataset\_filename = "dataset-50000.txt";

        break;

    case 3:

        dataset\_filename = "dataset-100000.txt";

        break;

    case 4:

        dataset\_filename = "dataset-200000.txt";

        break;

    case 5:

        dataset\_filename = "dataset-400000.txt";

        break;

    case 6:

        dataset\_filename = "dataset-500000.txt";

        break;

    case 7:

        dataset\_filename = "dataset-600000.txt";

        break;

    case 8:

        dataset\_filename = "dataset-800000.txt";

        break;

    case 9:

        dataset\_filename = "dataset-1000000.txt";

        break;

    default:

        dataset\_filename = "dataset-10000.txt";

        break;

    }

    /\*

        Function reads dataset\_file and store data into data\_points array. Each points have three consecutive indices associated into array.

        data\_points array looks like : [pt\_1\_x, pt\_1\_y, pt\_1\_z, pt\_2\_x, pt\_2\_y, pt\_2\_z]

    \*/

    dataset\_in (dataset\_filename, &N, &data\_points);

    start\_time = omp\_get\_wtime();

    kmeans\_sequential(N, K, data\_points, &cluster\_points, &iter\_centroids, &number\_of\_iterations);

    end\_time = omp\_get\_wtime();

    // Creating filenames for different dataset

    char file\_index\_char[2];

    snprintf(file\_index\_char,10,"%d", x);

    char cluster\_filename[105] = "cluster\_output\_dataset";

    strcat(cluster\_filename,file\_index\_char);

    strcat(cluster\_filename,".txt");

    char centroid\_filename[105] = "centroid\_output\_dataset";

    strcat(centroid\_filename,file\_index\_char);

    strcat(centroid\_filename,".txt");

    /\*

        Clustered points are saved into cluster\_filename.

        Each point is associated with the cluster index it belongs to.

        cluster\_points array looks like : [pt\_1\_x, pt\_1\_y, pt\_1\_z, pt\_1\_cluster\_index, pt\_2\_x, pt\_2\_y, pt\_2\_z, pt\_2\_cluster\_index]

        Output file format:

            pt\_1\_x, pt\_1\_y, pt\_1\_z, pt\_1\_cluster\_index

    \*/

    clusters\_out (cluster\_filename, N, cluster\_points);

    /\*

        Centroid points are stored into centroid\_filename.

        Each line in the file depicts the centroid coordinates of clusters after each iteration.

        Output file format:

            centroid\_1\_x, centroid\_1\_y, centroid\_1\_z, centroid\_2\_x, centroid\_2\_y, centroid\_2\_z

    \*/

    centroids\_out (centroid\_filename, K, number\_of\_iterations, iter\_centroids);

    /\*

        Computation time is stored in 'compute\_time\_openmp.txt'.

    \*/

    computation\_time = end\_time - start\_time;

    printf("Time Taken: %lf \n", computation\_time);

    char time\_file\_omp[100] = "compute\_time\_openmp\_dataset";

        strcat(time\_file\_omp,file\_index\_char);

        strcat(time\_file\_omp,".txt");

    FILE \*fout = fopen(time\_file\_omp, "a");

    fprintf(fout, "%f\n", computation\_time);

    fclose(fout);

    printf("Cluster Centroid point output file '%s' saved\n", centroid\_filename);

    printf("Clustered points output file '%s' saved\n", cluster\_filename);

        printf("Computation time output file '%s' saved\n", time\_file\_omp);

    return 0;

}

**For Parallel:**

#include <omp.h>

#include <stdlib.h>

#include <stdio.h>

#include <limits.h>

#include <float.h>

#include <math.h>

#include <assert.h>

#include <string.h>

#define MAX\_ITER 100

#define THRESHOLD 1e-6

// Global Variables used across different functions

int number\_of\_points\_global;

int number\_of\_threads\_global;

int number\_of\_iterations\_global;

int K\_global;

int \*data\_points\_global;

float \*iter\_centroids\_global;

int \*data\_point\_cluster\_global;

int \*\*iter\_cluster\_count\_global;

// Defined global delta

double delta\_global = THRESHOLD + 1;

void kmeans\_openmp\_thread(int \*tid)

{

    int \*id = (int \*)tid;

    // Assigning data points range to each thread

    int data\_length\_per\_thread = number\_of\_points\_global / number\_of\_threads\_global;

    int start = (\*id) \* data\_length\_per\_thread;

    int end = start + data\_length\_per\_thread;

    if (end + data\_length\_per\_thread > number\_of\_points\_global)

    {

        //To assign last undistributed points to this thread for computation, change end index to number\_of\_points\_global

        end = number\_of\_points\_global;

        data\_length\_per\_thread = number\_of\_points\_global - start;

    }

    printf("Thread ID:%d, start:%d, end:%d\n", \*id, start, end);

    int i = 0, j = 0;

    double min\_dist, current\_dist;

    // Cluster id associated with each point

    int \*point\_to\_cluster\_id = (int \*)malloc(data\_length\_per\_thread \* sizeof(int));

    // Cluster location or centroid (x,y,z) coordinates for K clusters in a iteration

    float \*cluster\_points\_sum = (float \*)malloc(K\_global \* 3 \* sizeof(float));

    // No. of points in a cluster for a iteration

    int \*points\_inside\_cluster\_count = (int \*)malloc(K\_global \* sizeof(int));

    // Start of loop

    int iter\_counter = 0;

    while ((delta\_global > THRESHOLD) && (iter\_counter < MAX\_ITER))

    {

        // Initialize cluster\_points\_sum or centroid to 0.0

        for (i = 0; i < K\_global \* 3; i++)

            cluster\_points\_sum[i] = 0.0;

        // Initialize number of points for each cluster to 0

        for (i = 0; i < K\_global; i++)

            points\_inside\_cluster\_count[i] = 0;

        for (i = start; i < end; i++)

        {

            //Assign these points to their nearest cluster

            min\_dist = DBL\_MAX;

            for (j = 0; j < K\_global; j++)

            {

                current\_dist = pow((double)(iter\_centroids\_global[(iter\_counter \* K\_global + j) \* 3] - (float)data\_points\_global[i \* 3]), 2.0) +

                               pow((double)(iter\_centroids\_global[(iter\_counter \* K\_global + j) \* 3 + 1] - (float)data\_points\_global[i \* 3 + 1]), 2.0) +

                               pow((double)(iter\_centroids\_global[(iter\_counter \* K\_global + j) \* 3 + 2] - (float)data\_points\_global[i \* 3 + 2]), 2.0);

                if (current\_dist < min\_dist)

                {

                    min\_dist = current\_dist;

                    point\_to\_cluster\_id[i - start] = j;

                }

            }

            //Update local count of number of points inside cluster

            points\_inside\_cluster\_count[point\_to\_cluster\_id[i - start]] += 1;

            // Update local sum of cluster data points

            cluster\_points\_sum[point\_to\_cluster\_id[i - start] \* 3] += (float)data\_points\_global[i \* 3];

            cluster\_points\_sum[point\_to\_cluster\_id[i - start] \* 3 + 1] += (float)data\_points\_global[i \* 3 + 1];

            cluster\_points\_sum[point\_to\_cluster\_id[i - start] \* 3 + 2] += (float)data\_points\_global[i \* 3 + 2];

        }

/\*

    Update iter\_centroids\_global and iter\_cluster\_count\_global after each thread arrival

    Supporting formula is

    (prev\_iter\_centroid\_global \* prev\_iter\_cluster\_count + new\_thread\_cluster\_points\_sum) / (new\_thread\_cluster\_count + prev\_iter\_cluster\_count)

\*/

#pragma omp critical

        {

            for (i = 0; i < K\_global; i++)

            {

                if (points\_inside\_cluster\_count[i] == 0)

                {

                    printf("Unlikely situation!\n");

                    continue;

                }

                iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3] = (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3] \* iter\_cluster\_count\_global[iter\_counter][i] + cluster\_points\_sum[i \* 3]) / (float)(iter\_cluster\_count\_global[iter\_counter][i] + points\_inside\_cluster\_count[i]);

                iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 1] = (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 1] \* iter\_cluster\_count\_global[iter\_counter][i] + cluster\_points\_sum[i \* 3 + 1]) / (float)(iter\_cluster\_count\_global[iter\_counter][i] + points\_inside\_cluster\_count[i]);

                iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 2] = (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 2] \* iter\_cluster\_count\_global[iter\_counter][i] + cluster\_points\_sum[i \* 3 + 2]) / (float)(iter\_cluster\_count\_global[iter\_counter][i] + points\_inside\_cluster\_count[i]);

                iter\_cluster\_count\_global[iter\_counter][i] += points\_inside\_cluster\_count[i];

            }

        }

/\*

    Wait for all threads to arrive and execute for first thread only

    Delta is the sum of squared distance between centroid of previous and current iteration.

    Supporting formula is:

        delta = (iter1\_centroid1\_x - iter2\_centroid1\_x)^2 + (iter1\_centroid1\_y - iter2\_centroid1\_y)^2 + (iter1\_centroid1\_z - iter2\_centroid1\_z)^2 + (iter1\_centroid2\_x - iter2\_centroid2\_x)^2 + (iter1\_centroid2\_y - iter2\_centroid2\_y)^2 + (iter1\_centroid2\_z - iter2\_centroid2\_z)^2

    Update delta\_global with new delta

\*/

#pragma omp barrier

        if (\*id == 0)

        {

            double temp\_delta = 0.0;

            for (i = 0; i < K\_global; i++)

            {

                temp\_delta += (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3]) \* (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3]) + (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 1] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3 + 1]) \* (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 1] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3 + 1]) + (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 2] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3 + 2]) \* (iter\_centroids\_global[((iter\_counter + 1) \* K\_global + i) \* 3 + 2] - iter\_centroids\_global[((iter\_counter)\*K\_global + i) \* 3 + 2]);

            }

            delta\_global = temp\_delta;

            number\_of\_iterations\_global++;

        }

// Wait for all thread to arrive and update the iter\_counter by +1

#pragma omp barrier

        iter\_counter++;

    }

//End of loop

// Assign points to final choice for cluster centroids

    for (i = start; i < end; i++)

    {

        // Assign points to clusters

        data\_point\_cluster\_global[i \* 4] = data\_points\_global[i \* 3];

        data\_point\_cluster\_global[i \* 4 + 1] = data\_points\_global[i \* 3 + 1];

        data\_point\_cluster\_global[i \* 4 + 2] = data\_points\_global[i \* 3 + 2];

        data\_point\_cluster\_global[i \* 4 + 3] = point\_to\_cluster\_id[i - start];

        assert(point\_to\_cluster\_id[i - start] >= 0 && point\_to\_cluster\_id[i - start] < K\_global);

    }

}

void kmeans\_omp(int num\_threads,

                    int N,

                    int K,

                    int \*data\_points,

                    int \*\*data\_points\_cluster\_id,

                    float \*\*iter\_centroids,

                    int \*number\_of\_iterations)

{

    // Initialize global variables

    number\_of\_points\_global = N;

    number\_of\_threads\_global = num\_threads;

    number\_of\_iterations\_global = 0;

    K\_global = K;

    data\_points\_global = data\_points;

    \*data\_points\_cluster\_id = (int \*)malloc(N \* 4 \* sizeof(int));   //Allocating space of 4 units each for N data points

    data\_point\_cluster\_global = \*data\_points\_cluster\_id;

    /\*

        Allocating space of 3K units for each iteration

        Since three dimensional data point and K number of clusters

    \*/

    iter\_centroids\_global = (float \*)calloc((MAX\_ITER + 1) \* K \* 3, sizeof(float));

    // Assigning first K points to be initial centroids

    int i = 0;

    for (i = 0; i < K; i++)

    {

        iter\_centroids\_global[i \* 3] = data\_points[i \* 3];

        iter\_centroids\_global[i \* 3 + 1] = data\_points[i \* 3 + 1];

        iter\_centroids\_global[i \* 3 + 2] = data\_points[i \* 3 + 2];

    }

    // Print initial centroids

    for (i = 0; i < K; i++)

    {

        printf("initial centroid #%d: %f,%f,%f\n", i + 1, iter\_centroids\_global[i \* 3], iter\_centroids\_global[i \* 3 + 1], iter\_centroids\_global[i \* 3 + 2]);

    }

    /\*

        Allocating space for iter\_cluster\_count\_global

        iter\_cluster\_count\_global keeps the count of number of points in K clusters after each iteration

     \*/

    iter\_cluster\_count\_global = (int \*\*)malloc(MAX\_ITER \* sizeof(int \*));

    for (i = 0; i < MAX\_ITER; i++)

    {

        iter\_cluster\_count\_global[i] = (int \*)calloc(K, sizeof(int));

    }

    // Creating threads

    omp\_set\_num\_threads(num\_threads);

#pragma omp parallel

    {

        int ID = omp\_get\_thread\_num();

        printf("Thread: %d created!\n", ID);

        kmeans\_openmp\_thread(&ID);

    }

    // Record number\_of\_iterations

    \*number\_of\_iterations = number\_of\_iterations\_global;

    // Record number of iterations and store iter\_centroids\_global data into iter\_centroids

    int iter\_centroids\_size = (\*number\_of\_iterations + 1) \* K \* 3;

    printf("Number of iterations :%d\n", \*number\_of\_iterations);

    \*iter\_centroids = (float \*)calloc(iter\_centroids\_size, sizeof(float));

    for (i = 0; i < iter\_centroids\_size; i++)

    {

        (\*iter\_centroids)[i] = iter\_centroids\_global[i];

    }

    // Print final centroids after last iteration

    for (i = 0; i < K; i++)

    {

        printf("centroid #%d: %f,%f,%f\n", i + 1, (\*iter\_centroids)[((\*number\_of\_iterations) \* K + i) \* 3], (\*iter\_centroids)[((\*number\_of\_iterations) \* K + i) \* 3 + 1], (\*iter\_centroids)[((\*number\_of\_iterations) \* K + i) \* 3 + 2]);

    }

}

void dataset\_in(const char \*dataset\_filename, int \*N, int \*\*data\_points)

{

    FILE \*fin = fopen(dataset\_filename, "r");

    fscanf(fin, "%d", N);

    \*data\_points = (int \*)malloc(sizeof(int) \* ((\*N) \* 3));

    int i = 0;

    for (i = 0; i < (\*N) \* 3; i++)

    {

        fscanf(fin, "%d", (\*data\_points + i));

    }

    fclose(fin);

}

void clusters\_out(const char \*cluster\_filename, int N, int \*cluster\_points)

{

    FILE \*fout = fopen(cluster\_filename, "w");

    int i = 0;

    for (i = 0; i < N; i++)

    {

        fprintf(fout, "%d %d %d %d\n",

                \*(cluster\_points + (i \* 4)), \*(cluster\_points + (i \* 4) + 1),

                \*(cluster\_points + (i \* 4) + 2), \*(cluster\_points + (i \* 4) + 3));

    }

    fclose(fout);

}

void centroids\_out(const char \*centroid\_filename, int K, int number\_of\_iterations, float \*iter\_centroids)

{

    FILE \*fout = fopen(centroid\_filename, "w");

    int i = 0;

    for (i = 0; i < number\_of\_iterations + 1; i++)

    {

        int j = 0;

        for (j = 0; j < K; j++)

        {

            fprintf(fout, "%f %f %f, ",

                    \*(iter\_centroids + (i \* K + j) \* 3),         //x coordinate

                    \*(iter\_centroids + (i \* K + j) \* 3 + 1),  //y coordinate

                    \*(iter\_centroids + (i \* K + j) \* 3 + 2)); //z coordinate

        }

        fprintf(fout, "\n");

    }

    fclose(fout);

}

void main()

{

    //---------------------------------------------------------------------

    int N;                  //no. of data points (input)

    int K;                  //no. of clusters to be formed (input)

    int num\_threads;        //no. of threads to be used (input)

    int\* data\_points;       //data points (input)

    int\* cluster\_points;    //clustered data points (to be computed)

    float\* iter\_centroids;          //centroids of each iteration (to be computed)

    int number\_of\_iterations;    //no of iterations performed by algo (to be computed)

    //---------------------------------------------------------------------

    char \*dataset\_filename = "dataset-10000.txt";

    printf("Enter No. of Clusters: ");

    scanf("%d", &K);

    printf("Enter No. of threads to be used: ");

    scanf("%d",&num\_threads);

    printf("\nFollowing files should be in the same directory as of program\n");

    printf("1 for 10000 datapoints\n");

    printf("2 for 50000 datapoints\n");

    printf("3 for 100000 datapoints\n");

    printf("4 for 200000 datapoints\n");

    printf("5 for 400000 datapoints\n");

    printf("6 for 500000 datapoints\n");

    printf("7 for 600000 datapoints\n");

    printf("8 for 800000 datapoints\n");

    printf("9 for 1000000 datapoints\n");

    printf("\nEnter the number of dataset file to input: ");

    int x;

    scanf("%d",&x);

    switch (x)

    {

    case 1:

        dataset\_filename = "dataset-10000.txt";

        break;

    case 2:

        dataset\_filename = "dataset-50000.txt";

        break;

    case 3:

        dataset\_filename = "dataset-100000.txt";

        break;

    case 4:

        dataset\_filename = "dataset-200000.txt";

        break;

    case 5:

        dataset\_filename = "dataset-400000.txt";

        break;

    case 6:

        dataset\_filename = "dataset-500000.txt";

        break;

    case 7:

        dataset\_filename = "dataset-600000.txt";

        break;

    case 8:

        dataset\_filename = "dataset-800000.txt";

        break;

    case 9:

        dataset\_filename = "dataset-1000000.txt";

        break;

    default:

        dataset\_filename = "dataset-10000.txt";

        break;

    }

    double start\_time, end\_time;

    double computation\_time;

    /\*

        Function reads dataset\_file and store data into data\_points array. Each points have three consecutive indices associated into array.

        data\_points array looks like : [pt\_1\_x, pt\_1\_y, pt\_1\_z, pt\_2\_x, pt\_2\_y, pt\_2\_z]

    \*/

    dataset\_in (dataset\_filename, &N, &data\_points);

    /\*

        For one iteration and two clusters,

        iter\_centroids array looks like : [iter\_1\_cluster\_1\_x, iter\_1\_cluster\_1\_y, iter\_1\_cluster\_1\_z, iter\_1\_cluster\_2\_x, iter\_1\_cluster\_2\_y, iter\_1\_cluster\_2\_z, iter\_2\_cluster\_1\_x, ...]

        Similarly the array extends further with more iterations

    \*/

    start\_time = omp\_get\_wtime();

    kmeans\_omp(num\_threads, N, K, data\_points, &cluster\_points, &iter\_centroids, &number\_of\_iterations);

    end\_time = omp\_get\_wtime();

    // Creating filenames for different threads and different dataset

    char num\_threads\_char[3];

    snprintf(num\_threads\_char,10,"%d", num\_threads);

    char file\_index\_char[2];

    snprintf(file\_index\_char,10,"%d", x);

    char cluster\_filename[105] = "cluster\_output\_threads";

    strcat(cluster\_filename,num\_threads\_char);

    strcat(cluster\_filename,"\_dataset");

    strcat(cluster\_filename,file\_index\_char);

    strcat(cluster\_filename,".txt");

    char centroid\_filename[105] = "centroid\_output\_threads";

    strcat(centroid\_filename,num\_threads\_char);

    strcat(centroid\_filename,"\_dataset");

    strcat(centroid\_filename,file\_index\_char);

    strcat(centroid\_filename,".txt");

    /\*

        Clustered points are saved into cluster\_filename.

        Each point is associated with the cluster index it belongs to.

        cluster\_points array looks like : [pt\_1\_x, pt\_1\_y, pt\_1\_z, pt\_1\_cluster\_index, pt\_2\_x, pt\_2\_y, pt\_2\_z, pt\_2\_cluster\_index]

        Output file format:

            pt\_1\_x, pt\_1\_y, pt\_1\_z, pt\_1\_cluster\_index

    \*/

    clusters\_out (cluster\_filename, N, cluster\_points);

    /\*

        Centroid points are stored into centroid\_filename.

        Each line in the file depicts the centroid coordinates of clusters after each iteration.

        Output file format:

            centroid\_1\_x, centroid\_1\_y, centroid\_1\_z, centroid\_2\_x, centroid\_2\_y, centroid\_2\_z

    \*/

    centroids\_out (centroid\_filename, K, number\_of\_iterations, iter\_centroids);

    /\*

        Computation time is stored in 'compute\_time\_openmp.txt'.

    \*/

    computation\_time = end\_time - start\_time;

    printf("Time Taken: %lf \n", computation\_time);

    char time\_file\_omp[100] = "compute\_time\_openmp\_threads";

    strcat(time\_file\_omp,num\_threads\_char);

    strcat(time\_file\_omp,"\_dataset");

    strcat(time\_file\_omp,file\_index\_char);

    strcat(time\_file\_omp,".txt");

    FILE \*fout = fopen(time\_file\_omp, "a");

    fprintf(fout, "%f\n", computation\_time);

    fclose(fout);

    printf("Cluster Centroid point output file '%s' saved\n", centroid\_filename);

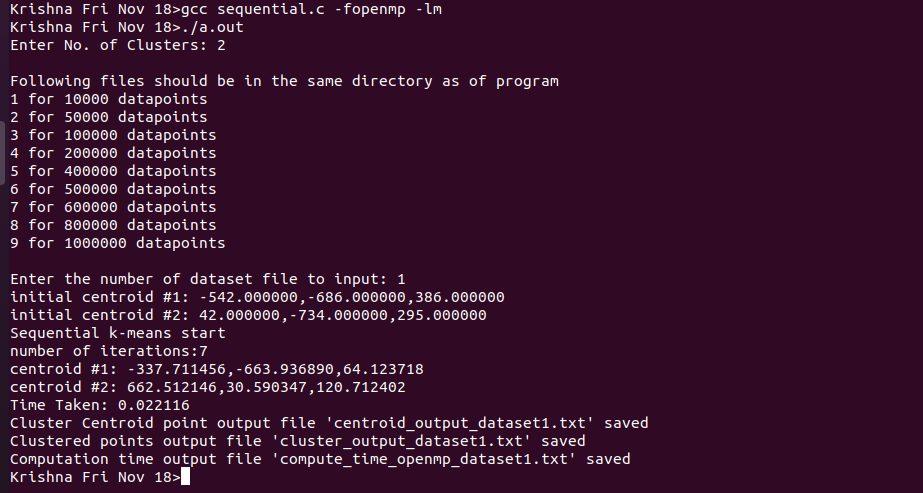
        printf("Clustered points output file '%s' saved\n", cluster\_filename);

        printf("Computation time output file '%s' saved\n", time\_file\_omp);

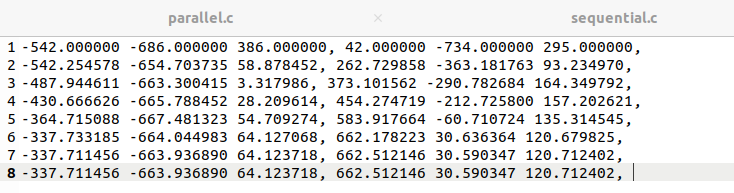
}

**Output Screenshots:**

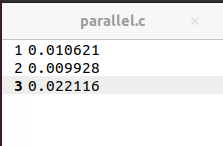
**For Shared**

****

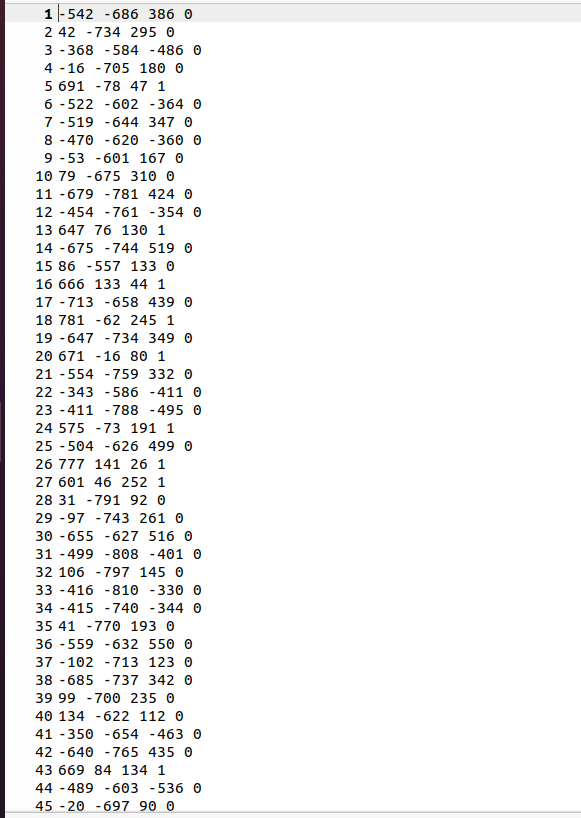
**Centroid Output:**

****

**Time Taken:**

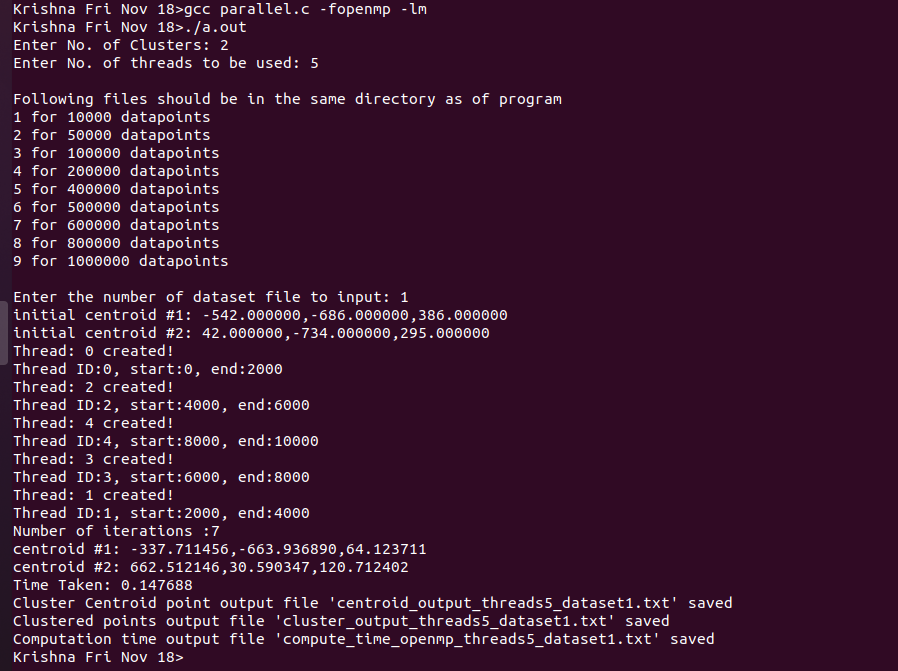
****

**Cluster Division:**

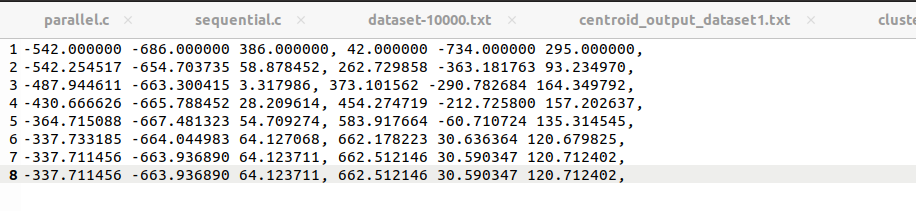
****

**Up to given Input.**

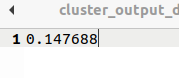
**For Parallel**



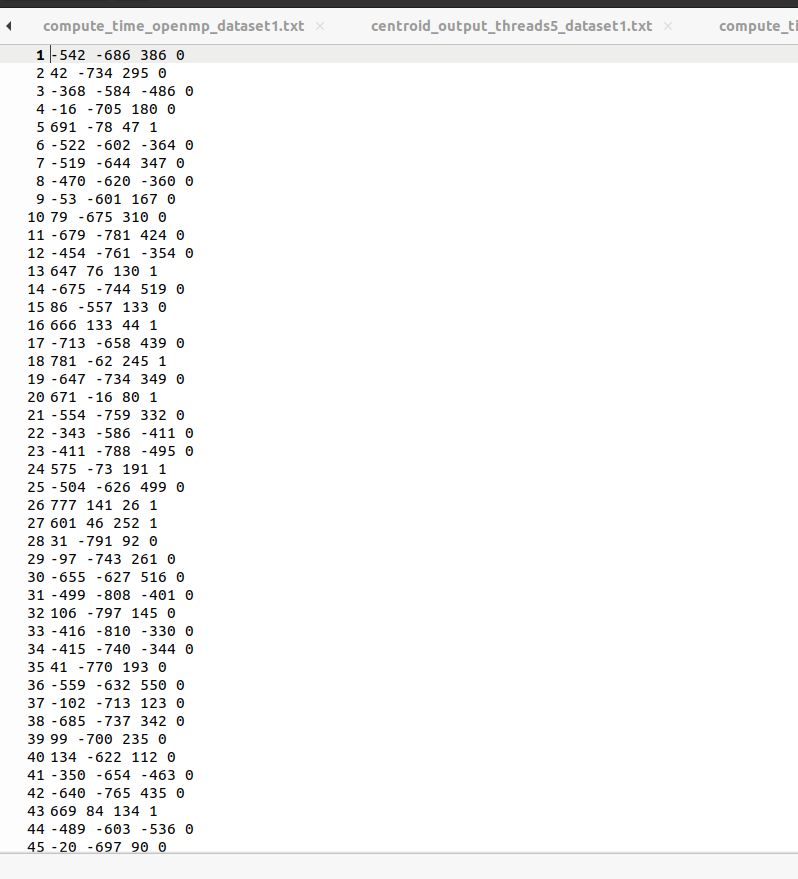
**Centroid Output:**



**Time Taken:**

****

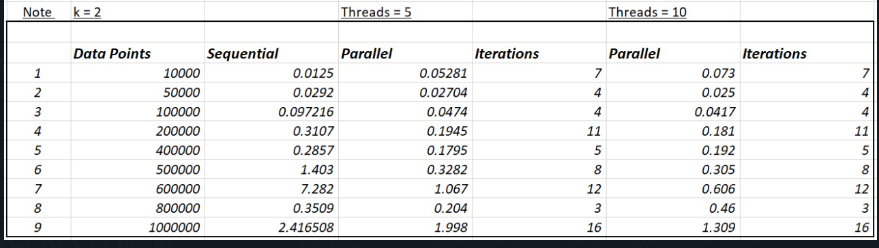
**Cluster Division:**

****

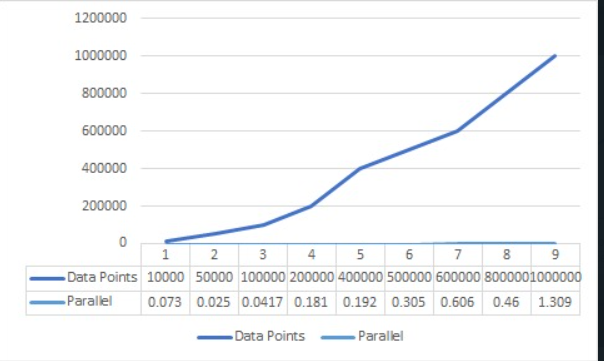
**Up to given input.**

# 14. Result Analysis:

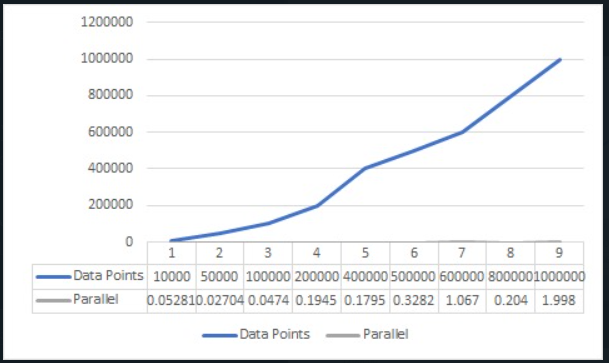
For Cluster 2 in Sequential and Parallel



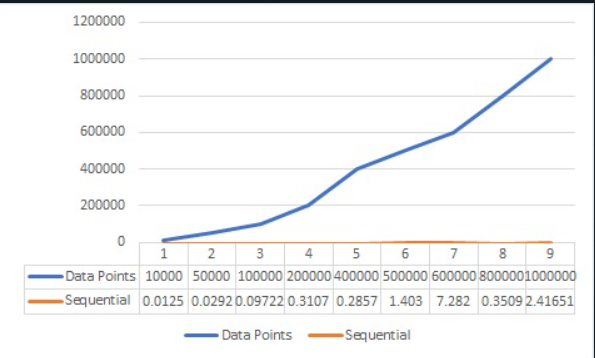
For Threads = 10 Parallel



For threads = 5 Parallel



For Sequential



* We can observe that when data points are fewer in number sequential is better than parallel.
* In parallel for fewer data points with a smaller number of threads it is implemented efficiently compared to more threads.
* Conversely, we observe the indirect relation for a greater number of data, so we observe sequential << parallel with fewer threads << parallel with more threads.

We can conclude from the results that for more data going for sequential is better than parallel even though the margin is less. But for more data, we need to go for parallel, which takes less time than sequential with a high margin.

# Conclusion:

The Parallel K-Mean algorithms solve the problems of the Simple algorithm, and the parallel algorithms always produce the same results, which improves the quality of the cluster, the number of iterations, and the amount of time that has passed. Additionally, the number of iterations required for each run or execution varies depending on the outcomes of the Simple K-Mean for different runs or executions. The purpose of the experiments was to demonstrate that Parallel algorithms are significantly outperforming the Simple K-Mean algorithm. The results of the Simple K-Mean algorithm vary with each run or execution, whereas those of the Parallel algorithms are also identical. Parallel algorithms also improve the number of iterations and the time taken to complete a task. The purpose of the experiments was to demonstrate that Parallel algorithms significantly outperform the Simple algorithm.

# References:

[1]. Yufang Zhang, Zhongyang Xiong, Jiali Mao and Ling Ou, "The Study of Parallel K-Means Algorithm,"

2006 6th World Congress on Intelligent Control and Automation, 2006, pp. 5868-5871, doi:

10.1109/WCICA.2006.1714203.

[2]. Preeti Arora, Deepali, Shipra Varshney, Analysis of K-Means and K-Medoids Algorithm for Big Data,

Procedia Computer Science, Volume 78, 2016, Pages 507-512, ISSN 1877-0509,

<https://doi.org/10.1016/j.procs.2016.02.095>.

[3]. Daniel Langr, Klára Schovánková, CPP11sort: A parallel quicksort based on C++ threading, Czech

Ministry of Education, Youth and Sports, CZ.02.1.01/0.0/0.0/16\_019/0000765, 09 September 2021

<https://doi.org/10.1002/cpe.6606>.

[4]. Zurek, D., Pietron, M., Wielgosz, M., & Wiatr, K. (2013). Comparison of Hybrid Sorting Algorithms

Implemented on Different Parallel Hardware Platforms. Computer Science, 14(4), 679.

<https://doi.org/10.7494/csci.2013.14.4.679>.

[5]. Dina Bitton, David J. DeWitt, David K. Hsaio, Jaishankar Menon, “A taxonomy of parallel sorting”,

Sept. 1984, ACM Computing SurveysVolume 16, Issue 3, <https://doi.org/10.1145/2514.251>