CPSC 479

Dr. Bein

Project Assignment 2

Group members:

Huy Ho hho114@csu.fullerton.edu

Chandler Ebrahimi csebra@csu.fullerton.edu

Darren Vu vuchampion@csu.fullerton.edu

## I. Problem statement:

Sometimes we just want to see how the data is organized, and that's where clustering comes into play. The word 'clustering' means grouping similar things together, and the most commonly used clustering method is K-Means because of its simplicity. So in this Project, we implement k mean clustering with mpi to show how to apply a high performing computers application in data science.

## II. Implement

Step 1: Create random data point numbers from 0 to 1, and assign partial number points to each processor. (Use MPI_Scatter)

Step 2: Choose the first few K points as centroids and assign them to cluster.

Step 3: In each processor for each data point find its cluster by calculating its distance with centroids.(User MPI_bcast for centroid list and mean distance tracking)

Step 4: Calculate the mean distance each cluster, and update centroids for each cluster.(Use MPI_Reduce to get points distance from each process)

Step 5: Go to step (3) and repeat until the number of iterations > 10000 or mean distance has changed is less than 0.00001.

Step 6: Label all points with its cluster, print results.

Note: Code Implement at the end of document

## III. How to run the program

Go to the project directory, and use mpicc to compile the main.c file (mpicc main.c), then run the compile file with four input arguments.

mpirun -n "number process" a.out "k number or number of cluster" "number dimension" "number points"

For example: mpicc main.c && mpirun -n 6 a.out 2 2 100
//will use 6 process with 2 kmean and 2 dimension for 100 data points

For better understanding how k-means work, we develop a program to output image files which show how the centroids change. However this will only work for 2 dimensions and the k number limit to 3.

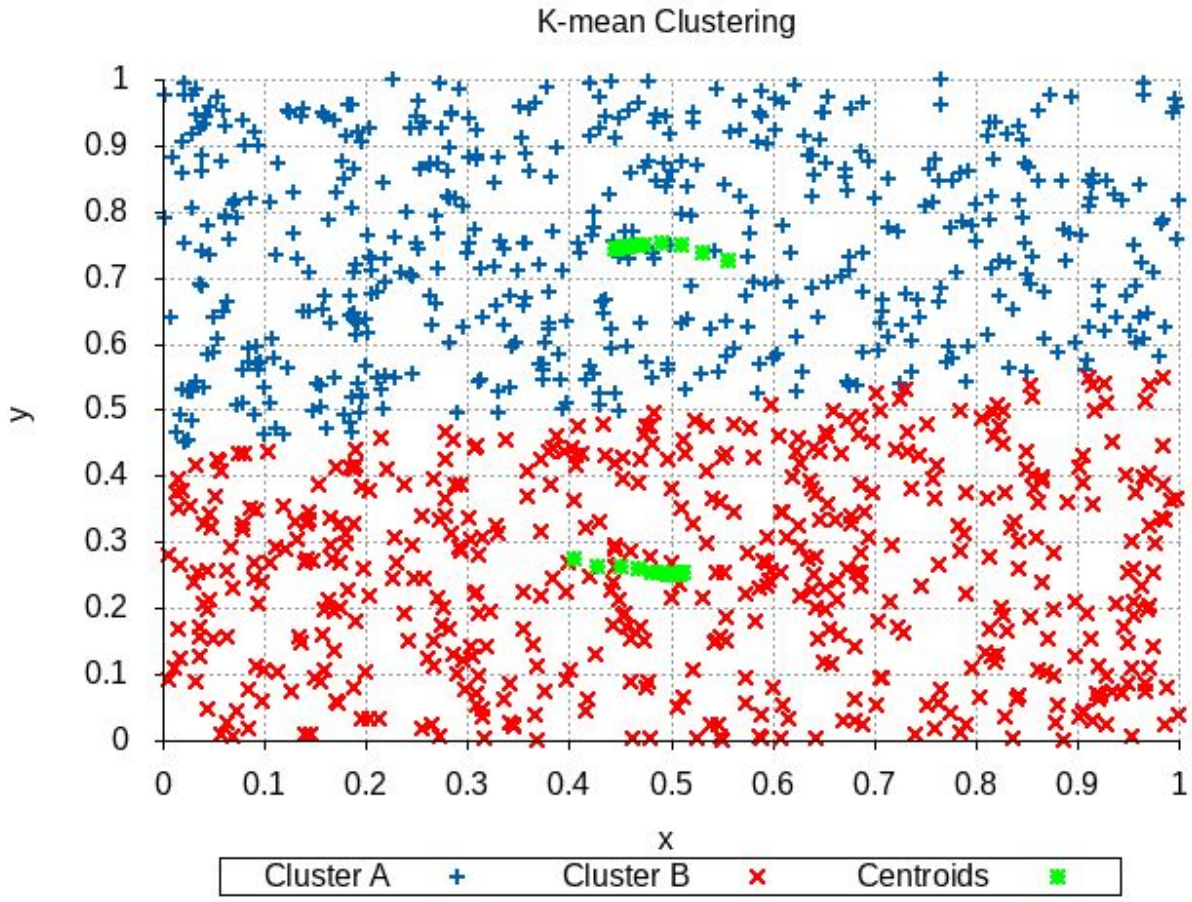
**Note**: This method require gnuplot program

- 2 Kmean Clustering 2 dimension with graph

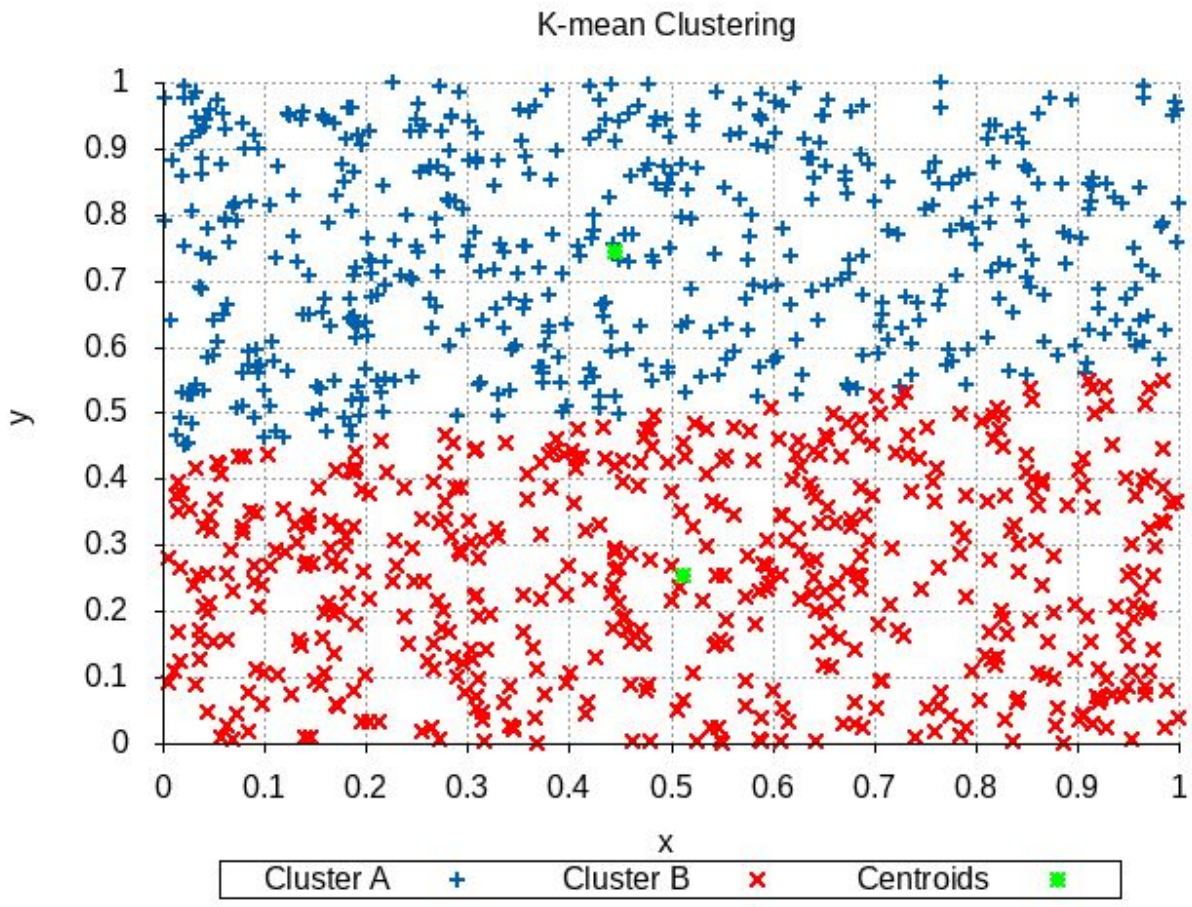
Use this command to create kmean cluster with 2 clusters and 100 data points per process graph:
Standard: mpicc main.c && mpirun -n "number process" a.out 2 2 "number points" && gnuplot graphs/2_kmean_graph.gp
Example: mpicc main.c && mpirun -n 10 a.out 2 2 1000 && gnuplot graphs/2_kmean_graph.gp

The output will look like this:

Changing centroids Process




Final centroid when finish




- 3 Kmean Clustering 2 dimension with graph

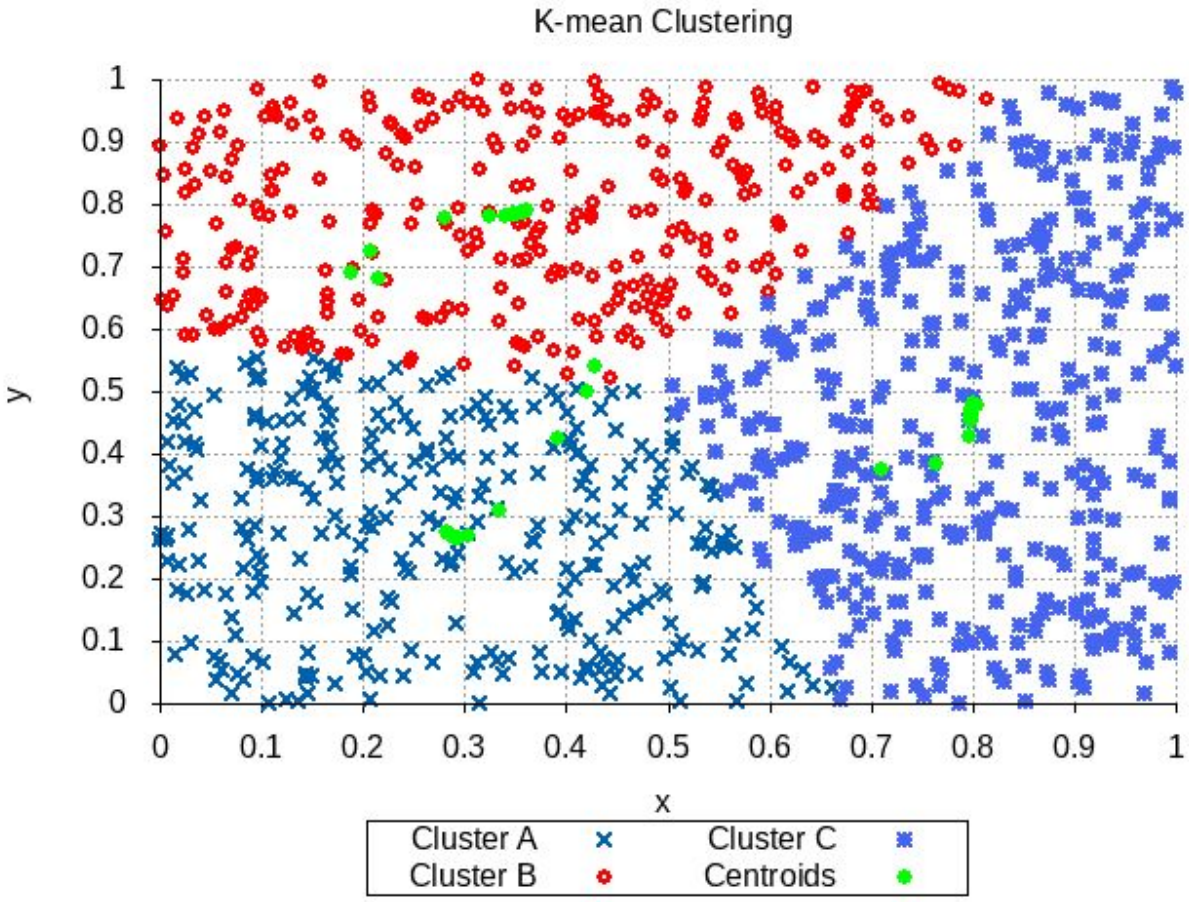
Use this command to create kmean cluster with 3 clusters and 100 data points per process graph:

Standard: mpicc main.c && mpirun -n "number process" a.out 3 2 "number point" && gnuplot graphs/3_kmean_graph.gp

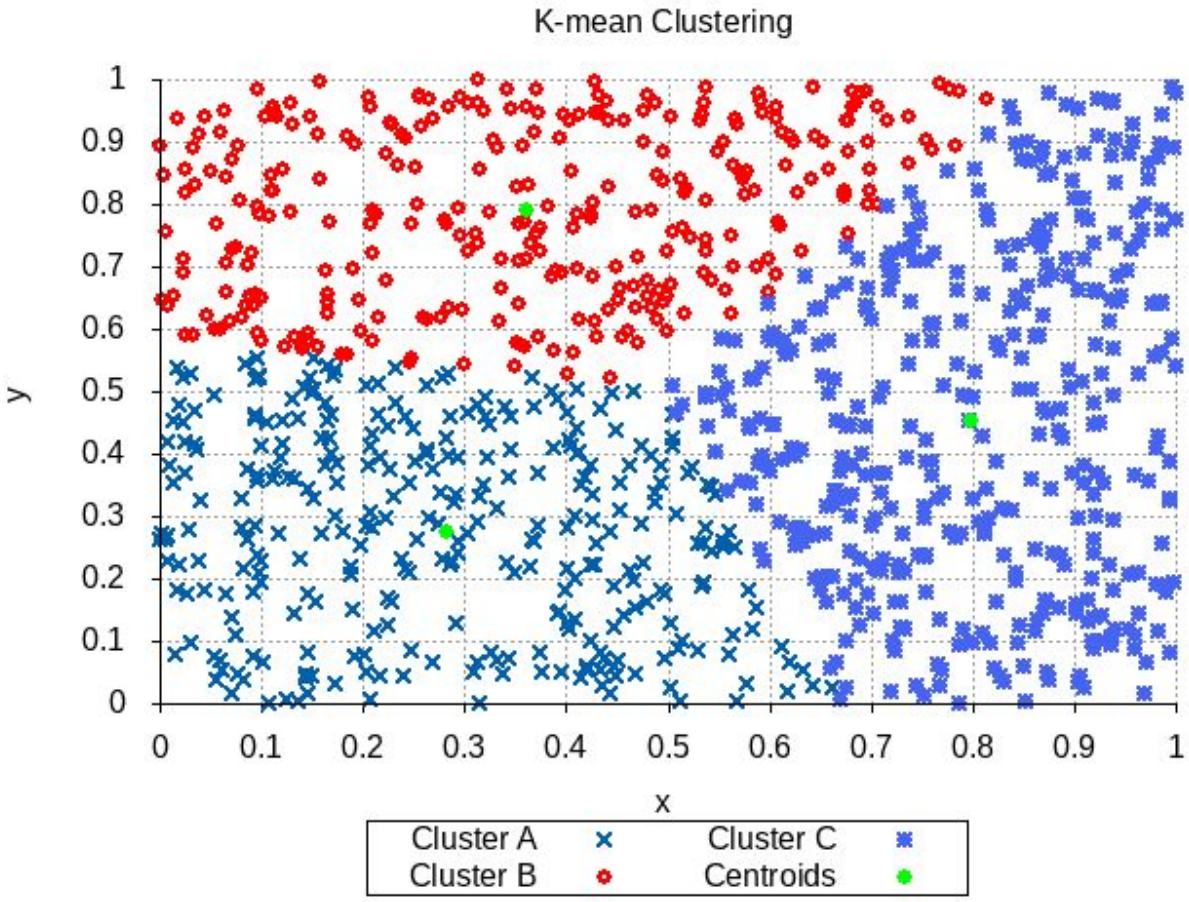
Example:  mpicc main.c && mpirun -n 10 a.out 3 2 1000 && gnuplot graphs/3_kmean_graph.gp

The output will look like this:

Changing centroids Process




Final centroid when finish

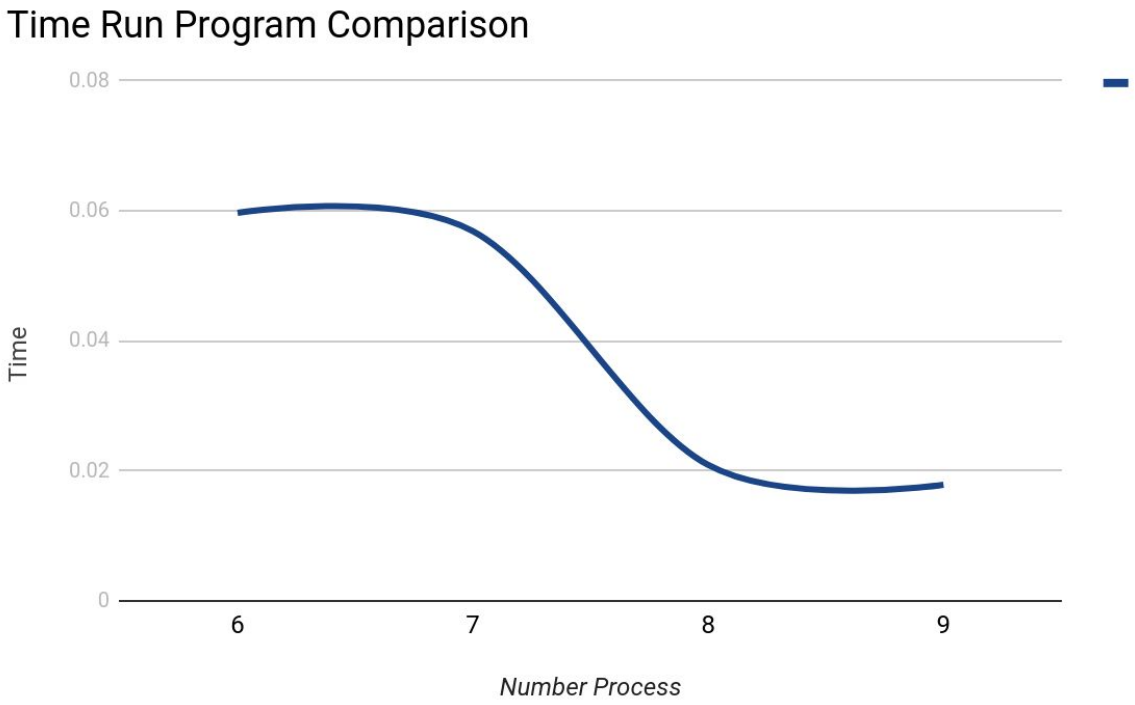



Command to run both method above “make”

## IV. Result From Run Program

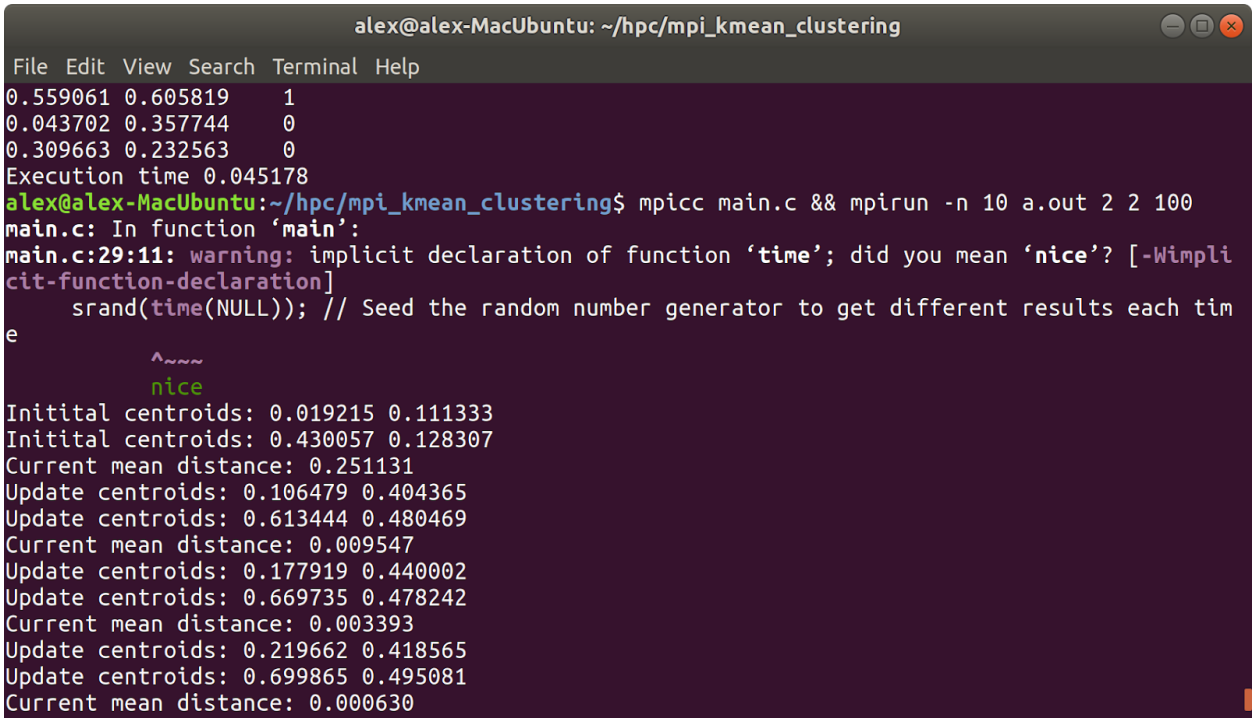
Table time run program with k number is 2, dimension is 2 and total point 1000 but N number process increase

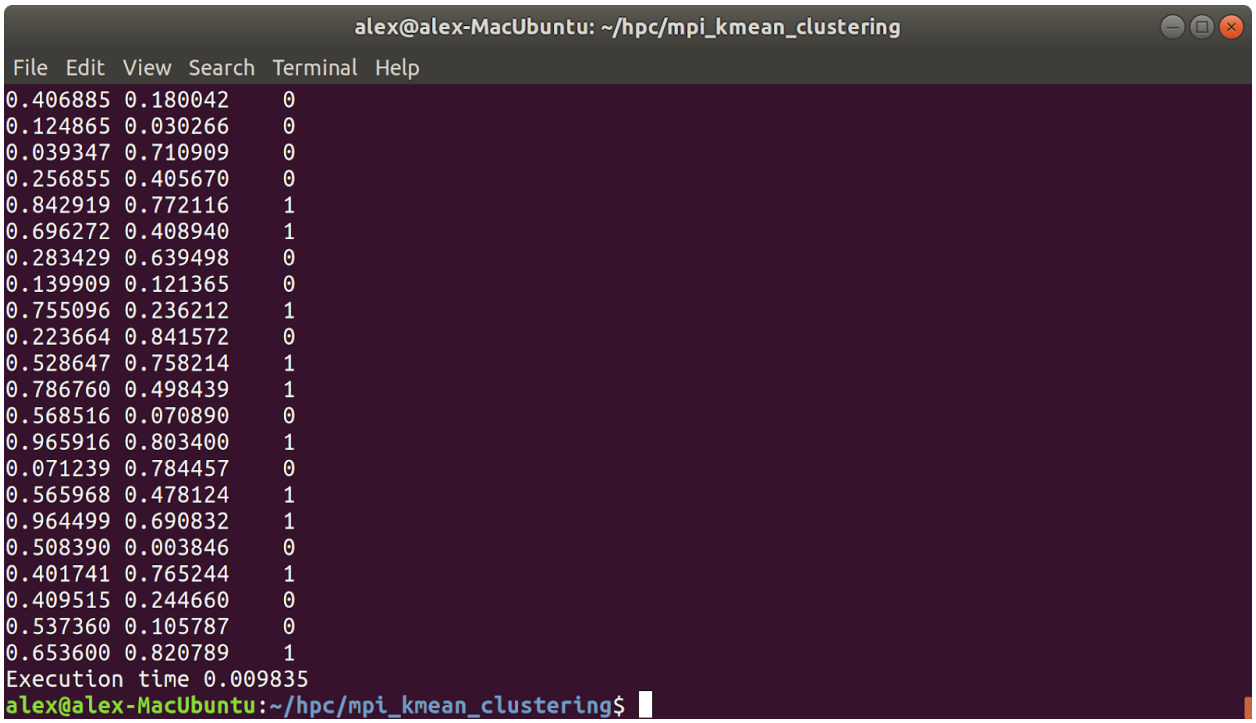
| N | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| Time | 0.059706 | 0.056879 | 0.020904 | 0.017869 |




Conclusion: The more process use to clustering, the faster program will be

Screenshots:



Code:

**Implement in C (main.c)**

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi/mpi.h>
#include <unistd.h>
#include <string.h>
#include "functions.h"

#define MAX_ITERATIONS 10000

int main(int argc, char **argv)
{
   if (argc != 4)
   {
       perror("Possible wrong command input\nStandard: mpirun -n <number process> <compile file> <k-mean number> <number dimensions> <number seeds> \n");
       exit(1);
   }

   int k = atoi(argv[1]);          // number of clusters.
   int dimension = atoi(argv[2]);  // dimension of data.
   int totalPoint = atoi(argv[3]); // total point input
   double start, end;              //time start and end
   // Initial MPI and find process rank and number of processes.
```

```
   MPI_Init(NULL, NULL);
   int rank, sizeRank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &sizeRank);
   MPI_Barrier(MPI_COMM_WORLD);


   srand(time(NULL)); // Seed the random number generator to get different
results each time
   // int totalPoint = 1000;
   int pointsPerProcess = totalPoint / sizeRank; //number points per
process
   start = MPI_Wtime();                          //start count the time
   int counter = 0;                              //counter until 10000
   // - The points assigned to each cluster by each process.
   // - The points get back from each process.
   // - The current centroids.
   float *recvPoints, *points, *centroids;


   // - The number of point assigned to each cluster by each process.
   // - The labels for each cluster.
   int *counts, *labels;


   recvPoints = malloc(pointsPerProcess * dimension * sizeof(float)); //
All points for all the processes.
   points = malloc(k * dimension * sizeof(float));                    //
Sum of points assigned to each cluster by all processes.
```

```
   counts = malloc(k * sizeof(int));                                    //
Sisze of each cluster
   centroids = malloc(k * dimension * sizeof(float));
   labels = malloc(pointsPerProcess * sizeof(int)); // The labels for each
cluster.


   float *allPoints = NULL;    // All points for all the processes
   float *pointSums = NULL;    // Sum of points assigned to each cluster by
all processes.
   int *clusterCounts = NULL; // Size of each cluster
   int *allLabels;             // Result of program: The labels for each
cluster.


   if (rank == 0)
   {


       allPoints = createRandomNums(dimension * totalPoint); //create
random number from 0 to 1
       // Take the first few k points as the initial cluster centroids.
       for (int i = 0; i < k * dimension; i++)
       {
           centroids[i] = allPoints[i];
       }


       inittialCentroids(centroids, k, dimension); //print centroids
       counter++;
       pointSums = malloc(k * dimension * sizeof(float));
```

```
        clusterCounts = malloc(k * sizeof(int));
        allLabels = malloc(sizeRank * pointsPerProcess * sizeof(int));
    }

    // Root sends each process its share of clusters.
    MPI_Scatter(allPoints, dimension * pointsPerProcess, MPI_FLOAT,
recvPoints,
                dimension * pointsPerProcess, MPI_FLOAT, 0,
MPI_COMM_WORLD);
    // MPI_Scatter(allPoints, dimension * pointsPerProcess, MPI_FLOAT,
recvPoints,
    //             dimension * pointsPerProcess, MPI_FLOAT, 0,
MPI_COMM_WORLD);

    float distance = 1;

    while (distance > 0.00001 && counter < MAX_ITERATIONS) //while counter
less than 10000 or distance greater than 0.0001 do prcess n work
    {

        // Broadcast the current cluster centroids to all processes.
        MPI_Bcast(centroids, k * dimension, MPI_FLOAT, 0, MPI_COMM_WORLD);

        // Each process reinitializes its cluster accumulators.
        for (int i = 0; i < k * dimension; i++)
        {
            points[i] = 0.0;
```

```
        }


        for (int i = 0; i < k; i++)
        {
            counts[i] = 0;
        }


        // Find the closest centroid to each point and assign to cluster.
        float *pointsAssign = recvPoints;
        for (int i = 0; i < pointsPerProcess; i++, pointsAssign += dimension)
        {
            int clusterNum = assignLabel(pointsAssign, centroids, k, dimension);
            // Record the assignment of the site to the cluster.
            counts[clusterNum]++;
//increase size of point in this cluster
            addPoint(pointsAssign, &points[clusterNum * dimension], dimension); // add point into its cluster
        }


        // Gather and sum at root all cluster sums for individual processes.
        MPI_Reduce(points, pointSums, k * dimension, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
        // Gather and sum count of point in each cluster
        MPI_Reduce(counts, clusterCounts, k, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
        if (rank == 0)
        {
            // Root process computes new centroids by dividing sums per
cluster by count per cluster.
            for (int i = 0; i < k; i++)
            {
                for (int j = 0; j < dimension; j++)
                {

                    pointSums[dimension * i + j] /= clusterCounts[i];
                }
            }
            // Get mean distance in cluster.
            distance = distanceBetween(pointSums, centroids, dimension *
k);
            printf("Current mean distance: %f\n", distance); //If mean
distance is zero, it mean the distance have not change and convergence
progess is done
            // Copy new centroids from pointSums into centroids.
            for (int i = 0; i < k * dimension; i++)
            {
                centroids[i] = pointSums[i];
            }
            notifyUpdateCentroids(centroids, k, dimension, &counter);
        }
        // Broadcast a better distance to all processes.
```

```
        MPI_Bcast(&distance, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
        counter++;
    }

    // Now centroids are fixed, so compute a final label for each cluster.
    float *pointsAssign = recvPoints;
    for (int i = 0; i < pointsPerProcess; i++, pointsAssign += dimension)
    {
        labels[i] = assignLabel(pointsAssign, centroids, k, dimension);
    }

    // Gather all labels into root process.
    MPI_Gather(labels, pointsPerProcess, MPI_INT,
               allLabels, pointsPerProcess, MPI_INT, 0, MPI_COMM_WORLD);

    // Root can print out all points and labels.
    if (rank == 0)
    {
        printf("\nFinal clustering result with centroid tag:\n"
               "x\ty\ttag\n");
        FILE *fp = fopen("./data/output.dat", "w");
        FILE *fp1 = fopen("./data/output1.dat", "w");
        FILE *fp2 = fopen("./data/output2.dat", "w");
        FILE *fpc = fopen("./data/final_centroids.dat", "w");
        for (size_t i = 0; i < k; i++)
        {
```

```
            for (size_t j = 0; j < dimension; j++, centroids++)
            {
                fprintf(fpc, "%f\t", *centroids);
            }
            fprintf(fpc, "\n");
        }

        float *allLabelPoints = allPoints;
        for (int i = 0; i < sizeRank * pointsPerProcess; i++,
allLabelPoints += dimension)
        {

            for (int j = 0; j < dimension; j++)
            {
                printf("%f ", allLabelPoints[j]);

                //Only for 2k and 3k with 2 dimension
                if (allLabels[i] == 0)
                {
                    fprintf(fp, "%f\t", allLabelPoints[j]);
                }
                else if (allLabels[i] == 1)
                {
                    fprintf(fp1, "%f\t", allLabelPoints[j]);
                }
                else if (allLabels[i] == 2)
```

```
            {
                fprintf(fp2, "%f\t", allLabelPoints[j]);
            }
        }
        //Only for 2k and 3k with 2 dimension
        if (allLabels[i] == 0)
        {
            fprintf(fp, "%4d\n", allLabels[i]);
        }
        else if (allLabels[i] == 1)
        {
            fprintf(fp1, "%4d\n", allLabels[i]);
        }
        else if (allLabels[i] == 2)
        {
            fprintf(fp2, "%4d\n", allLabels[i]);
        }

        printf("%4d\n", allLabels[i]);
    }

    fclose(fp);
    fclose(fp1);
    fclose(fpc);
}
MPI_Barrier(MPI_COMM_WORLD);
```

```
    end = MPI_Wtime();

    MPI_Finalize();

    if (rank == 0) /* use time on master node */

    {


        printf("Execution time %f \n", end - start);

    }

}
```

Implement in C (functions.h)

```
// Calculate distance between point and centroid using the euclidean
distance
float distanceBetween(const float *point, const float *centroid, const int
dimension)
{
    float dist = 0.0;
    for (int i = 0; i < dimension; i++)
    {
        float diff = point[i] - centroid[i];
        dist += diff * diff;
    }
    return dist;
}

// Assign label to the correct cluster by computing its distances to each
cluster centroid.
int assignLabel(const float *points, float *centroids,
                const int k, const int dimension)
{
    int bestCluster = 0;
```

```
    float bestDist = distanceBetween(points, centroids, dimension);
//calculate first point and first centroid distance
    float *centroid = centroids + dimension;                    //skip
first centroid
    for (int c = 1; c < k; c++, centroid += dimension)
    {
        float dist = distanceBetween(points, centroid, dimension);
        if (dist < bestDist)
        {
            bestCluster = c;
            bestDist = dist;
        }
    }
    return bestCluster;
}

// Add points into a sum of points.
void addPoint(const float *points, float *sums, const int dimension)
{
    for (int i = 0; i < dimension; i++)
    {
        sums[i] += points[i];
    }
}

// Print the centroids one per line.
void notifyUpdateCentroids(float *centroids, const int k, const int
dimension, int *counter)
{
    FILE *fpo;
    if (*counter == 1)
    {
        fpo = fopen("./data/old_centroids.dat", "w");
    }
    else
    {
```

```
        fpo = fopen("./data/old_centroids.dat", "a");
    }

    float *p = centroids;

    for (int i = 0; i < k; i++)
    {
        printf("Update centroids: ");
        for (int j = 0; j < dimension; j++, p++)
        {
            printf("%f ", *p);
            fprintf(fpo, "%f\t", *p);
        }
        fprintf(fpo, "\n");
        printf("\n");
    }

    fclose(fpo);

    *counter = *counter + 1;
}

void inittialCentroids(float *centroids, const int k, const int dimension)
{
    FILE *fpo = fopen("./data/init_centroids.dat", "w");

    float *p = centroids;

    for (int i = 0; i < k; i++)
    {
        printf("Inititial centroids: ");
        for (int j = 0; j < dimension; j++, p++)
        {
            printf("%f ", *p);
            fprintf(fpo, "%f\t", *p);
        }
        // fprintf(fpo, "\n");
```

```
        printf("\n");
    }

    fclose(fpo);
}

// Creates an array of random floats. Each number has a value from 0 - 1
float *createRandomNums(const int totalPoints)
{
    float *rand_nums = (float *)malloc(sizeof(float) * totalPoints);
    rand_nums != NULL;
    for (int i = 0; i < totalPoints; i++)
    {
        rand_nums[i] = (rand() / (float)RAND_MAX);

    }
    return rand_nums;
}
```