EECE5640 Final Project

Nicholas Fresneda

K-means Clustering with OpenMP and CUDA

Goals:

The main goal of my project was to implement two different versions (one using OpenMP and one using CUDA) of a straight forward k-means clustering algorithm and compare their performance for different configurations. A secondary goal was to graph a snapshot of the data set after every iteration using gnuplot, and to produce a video (gif) of the progression of the algorithm.

Algorithm:

My algorithm implementation follows the same sequence for both CUDA and OpenMP versions.

1. Generate data or read data from input file. Data consists of a list of 2D points.
2. Assign random centroids in the data's numeric range.
3. Iterate through dataset, assigning each 2d point to a centroid by finding the centroid with the closest Euclidean distance.
4. Output data and centroids to .dat files to be used by gnuplot to plot data.
5. Update centroids by taking the average 2d position for all 2d points for each centroid.

6. Repeat steps 3-5 for the given number of iterations, or until the centroids do not change in step 5 (convergence). The final output should be a .dat file for each iteration.
7. Finally, I use a gnuplot script to plot a png file for each .dat file.

Implementation:

1. OpenMP
    a. In my OpenMP implementation, I was able to use standard STL library template classes to store and do computations on my data set. For example, I stored a 2d point as a Coord class (with x and y member variables), and stored my data in a vector of pairs (Coord as the first value and assigned centroid number as the second value). I represented each centroid as a Coord stored in a vector, with its index serving as its identifier.
    b. My program consists of two big loops that were easily parallelizable: assigning each 2d point to a centroid and updating each centroid with the average of its assigned 2d points. I parallelized both these loops using OpenMP 'parallel for' pragma directives.
    c. The loop for assigning each 2d point to a centroid iterated through the whole dataset (up to 100,000,000 points in my benchmarks), so I used the maximum number of threads in my general Discovery Cluster partition (40 threads).
    d. The loop for updating each centroid had a max size of 16 iterations (one for each centroid - the max I used in my benchmarks was 16). So I simply directed omp to use only a thread per centroid (loop iteration).
    e. I did not parallelize writing to each .dat file.

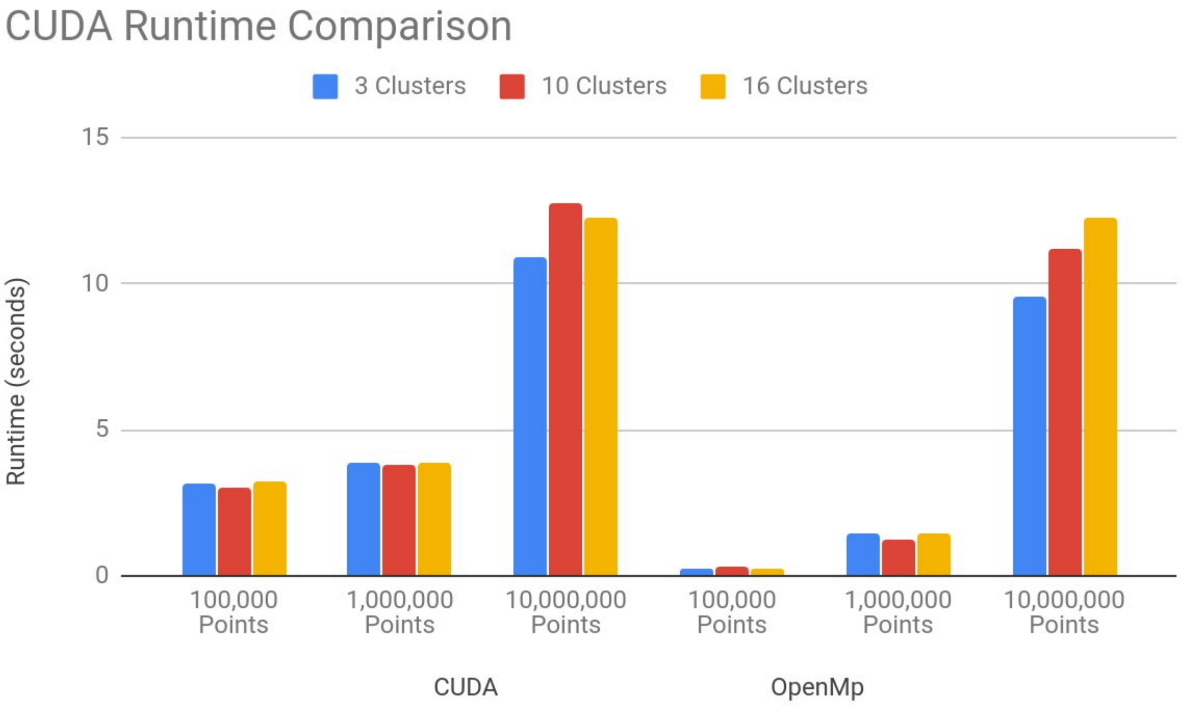
2. CUDA
    a. In my CUDA implementation, unlike my OpenMP implementation, I used more primitive data structures, as CUDA kernel code is not able to use the STL template library. To represent my data, I used arrays. I had an array for both the x and y values of each 2d data point, an array for both the x and y values for each cluster point, and an array of assigned centroids for each 2d data point. I used malloc() to allocate these arrays on the cpu, and cudaMalloc() to allocate them on the gpu. I generated / read the data on the cpu, and used cudaMemcpy() to copy the initialized data to the gpu.
    b. Similar to my OpenMP implementation, the bulk of my computation was in two for loops, one two update each 2d point's centroid and one to update the centroid coordinates. I created a CUDA kernel for updating each 2d point's centroid, as that operated over a very large dataset (100,000,000 in one of my benchmarks), but did not create a kernel to update the centroid coordinates. The loop to update the centroid coordinates only had up to 16 iterations, and while each of those iterations averaged the whole dataset, I could not optimally implement a CUDA kernel to average numbers in the dataset.

Results:

I tested my code on two data sets: a normal distribution by using the c standard rand() function to generate coordinates, and a high dimensional dataset I obtained from the website link cited below. I plotted these data sets using a gnuplot script to produce a series of PNG files, which I combined to make a gif video. The gif files are included in the final project submission. I used 16 centroids for the high dimensional dataset, and used 10 centroids for the normal distribution dataset. I have included final snapshots (as PNG files) of each configuration.

Below is a graph of the performance of my CUDA implementation compared with my OpenMP implementation. For very small data sets (100,000 and 1,000,000), the OpenMP version is much faster than the CUDA version. I believe this is a result of the overhead of allocating and transferring data to and from the GPU. In my CUDA implementation, I am sending the original data over to the GPU, and then for every kernel call, I copy over the new centroid coordinates, and copy back the new centroid assignments. This overhead can be seen greatly in small datasets. But, as the data size increases, the two versions become comparable. I only include the runtimes up to 10,000,000 input coordinates, but for input sizes much greater than that (100,000,000) the CUDA implementation is faster.

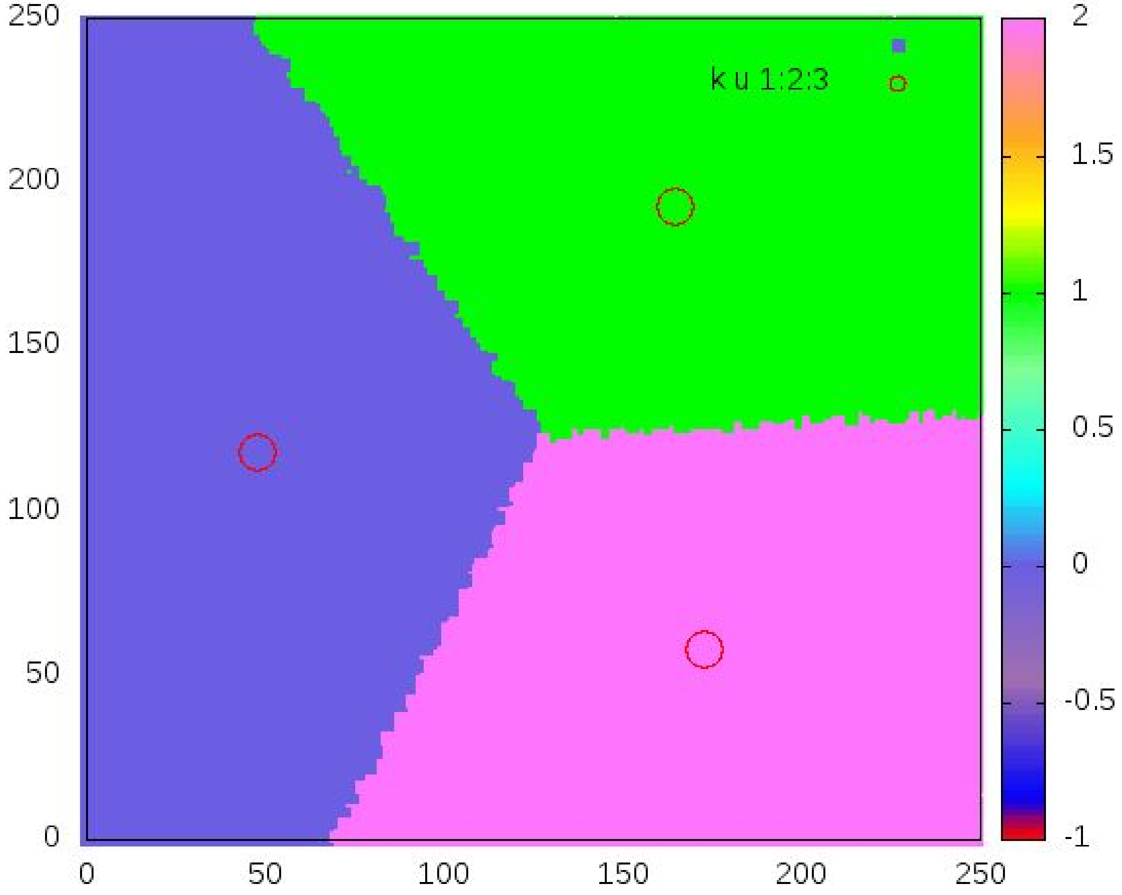
These results stress that a GPU needs a sufficient workload to attain speedup, and that allocating memory and copying data on the GPU greatly slows down computation.

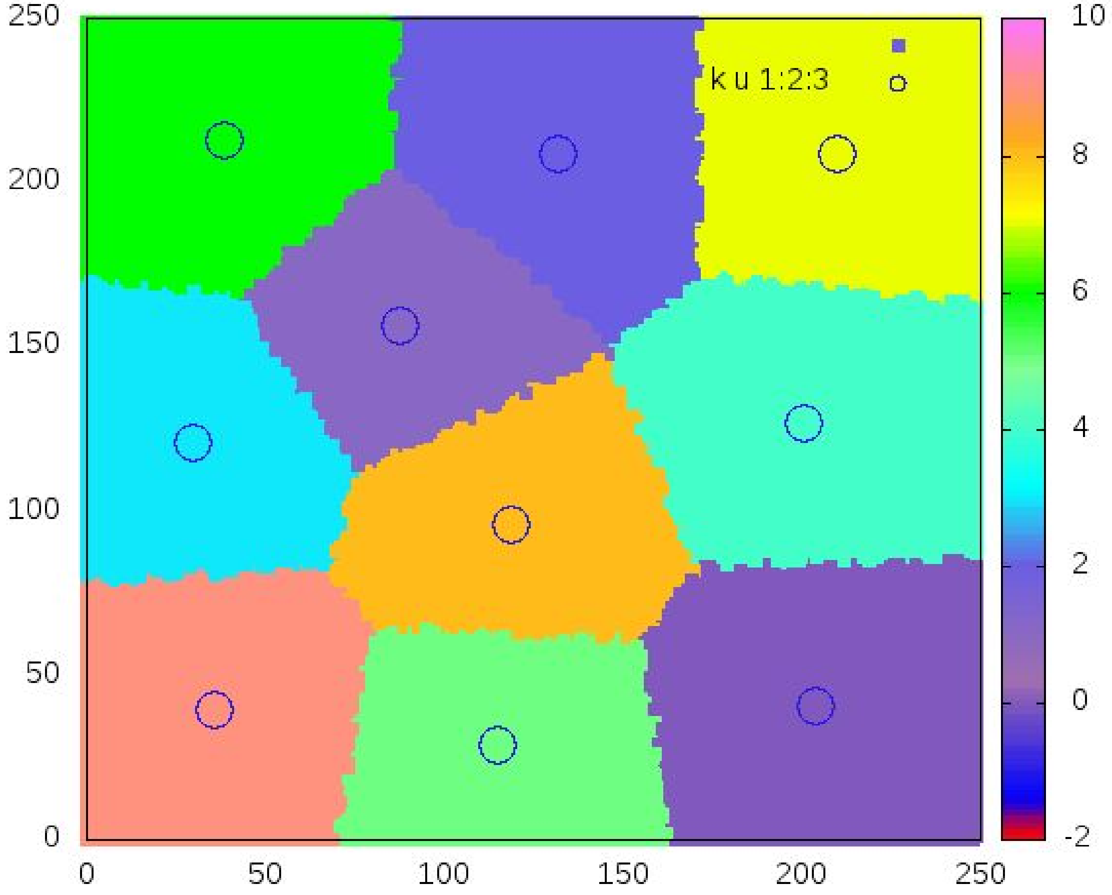

CUDA Runtime Comparison
3 Clusters
10 Clusters
16 Clusters
Runtime (seconds)
15
10
5
0
100,000 Points
1,000,000 Points
10,000,000 Points
100,000 Points
1,000,000 Points
10,000,000 Points
CUDA
OpenMp

Sources:

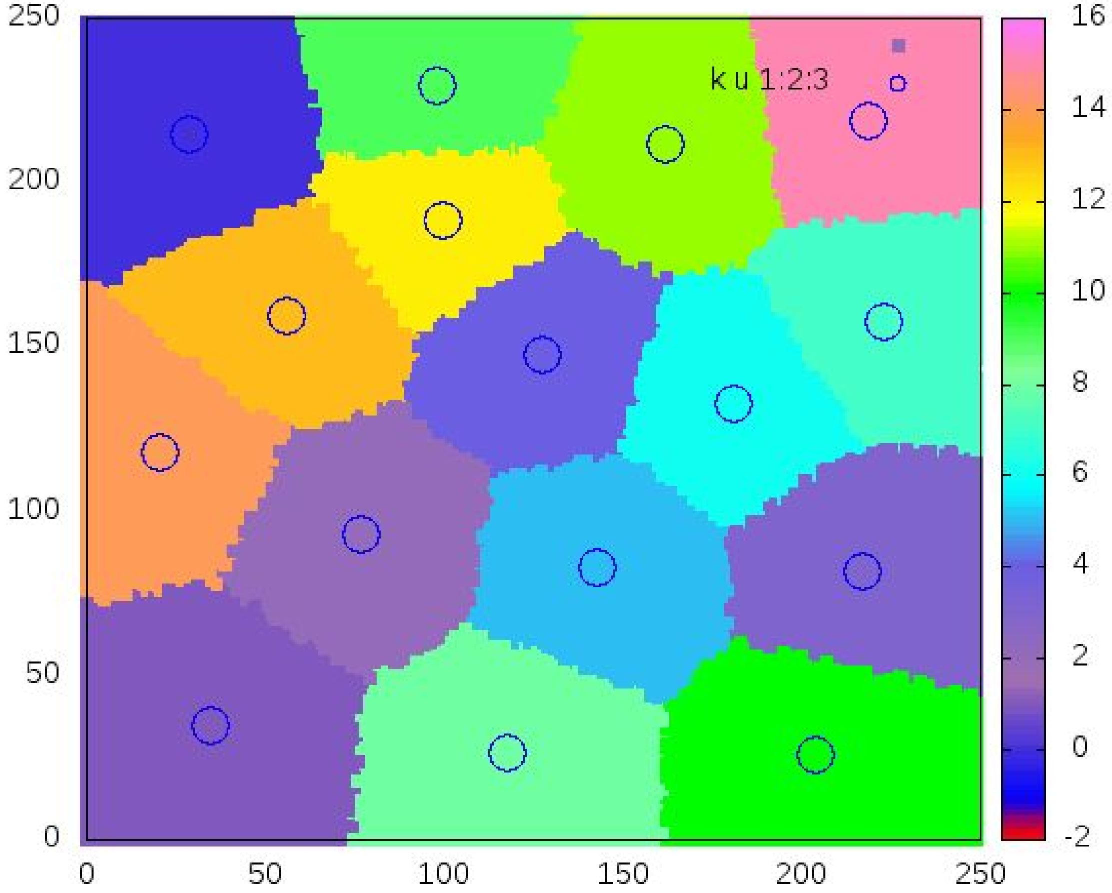
http://cs.joensuu.fi/sipu/datasets/

Final Images of Each Configuration (3, 10, and 16 centroids):


k u 1:2:3
0
50
100
150
200
250
0
50
100
150
200
250
2
1.5
1
0.5
0
-0.5
-1



250
200
150
100
50
0
0
50
100
150
200
250
k u 1:2:3
10
8
6
4
2
0
-2



Videos:

(Included in zip file)

Code (CUDA Implementation):

kmeans_cuda.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
#include <vector>
#include "Cluster.hpp"
#include <unordered_map>
#include <ctime>

using namespace  std;

#define COL_SZ 2
#define REQ_ARGS 4
#define MAX_K 20
#define MAX_PT 800
static int K_SZ;
static int ITER;
static long NUM_PTS;


void processInputs(int argc, char* argv[])
{
   if(argc != REQ_ARGS)
   {
       cout<<"Wrong inputs:\t\n"
```

```
        << "Usage: " << argv[0] << " <number of k clusters> <number of
points> <iterations>\n";
        exit(1);
    }

    K_SZ = atoi(argv[1]);
    if (K_SZ > MAX_K)
    {
        cout<<"Error: Only 20 or fewer k clusters supported\n";
    }

    NUM_PTS = atoi(argv[2]);
    ITER = atoi(argv[3]);
}

int main(int argc, char* argv[])
{
    //initialize inputs
    processInputs(argc, argv);

    Cluster cluster(K_SZ, NUM_PTS);
    //main loop
    for (int i = 0; i < ITER; i++)
    {
        cluster.updateCoordMap();
        cluster.writeToDatafile(i);
        cluster.updateClusterCoords();
        if (cluster.converged)
        {
            break;
        }
    }

}
```

Cluster.hpp

```
#ifndef CLUSTER_H
#define CLUSTER_H

#include <vector>
#include <unordered_map>
#include <string>
using namespace std;

class Cluster {
private:
  // vector<pair<Coord, int>> coordPairs;
   int* coordXVals;
   int* coordYVals;
   int* coordKMaps;
   int* clusterXVals;
   int* clusterYVals;
   int kSize;
   long coordSize;
   int* d_coordXVals;
   int* d_coordYVals;
   int* d_coordKMaps;
   int* d_clusterXVals;
   int* d_clusterYVals;
public:
   Cluster(int kSize, long coordSize);
   void updateCoordMap();
```

```
    void updateClusterCoords();
    void writeToDatafile(int fileNum);
    bool converged;
    ~Cluster();


};

#endif
```

Cluster.cu

```
#include "Cluster.hpp"
#include <math.h>
#include <fstream>
#include <string>
#include <iostream>
#include <omp.h>
#define MAX_PT 800
using namespace std;


__global__ void updateCoordKernel(long coordSize, int* coordXVals, int*
coordYVals,
                                  int* clusterXVals, int* clusterYVals, int
kSize,
                                  int* coordKMaps)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = tid; i < coordSize; i+= stride)
    {
        int coordX = coordXVals[i];
        int coordY = coordYVals[i];

        //set to first cluster's distance, will be updated in loop
        double p1 = (coordX - clusterXVals[0]) * (coordX -
clusterXVals[0]);
        double p2 = (coordY - clusterYVals[0]) * (coordY -
clusterYVals[0]);
        int minDist = (int) sqrt(p1 + p2);
        int index = 0;
        for (int j = 1; j < kSize; j++)
        {
            double p1 = (coordX - clusterXVals[j]) * (coordX -
clusterXVals[j]);
            double p2 = (coordY - clusterYVals[j]) * (coordY -
clusterYVals[j]);
            int newDist = (int) sqrt(p1 + p2);
            if (newDist < minDist)
            {
                minDist = newDist;
                index = j;
            }
        }

        coordKMaps[i] = index;
    }

}

Cluster::Cluster(int kSize, long coordSize)
{
    this->kSize = kSize;
    this->coordSize = coordSize;
    //allocate and initialize member variables
    coordXVals = (int*)malloc(coordSize * sizeof(int));
    coordYVals = (int*)malloc(coordSize * sizeof(int));
    coordKMaps = (int*)malloc(coordSize * sizeof(int));
    clusterXVals = (int*)malloc(kSize * sizeof(int));
    clusterYVals = (int*)malloc(kSize * sizeof(int));
}

Cluster::~Cluster()
{
}

void Cluster::writeToDatafile(int fileNum)
{
    //iterate through map
    //writing each coordinates x and y and then cluster num to file
    ofstream myfile;
    std::string file("data" + to_string(fileNum) + ".dat" );

    myfile.open (file);
    for(unsigned int i = 0; i < coordSize; i++)
    {
        myfile << coordXVals[i] << "\t" << coordYVals[i] << "\t"
         << coordKMaps[i] << "\n";
    }

    myfile.close();
    std::string kfile("data" + to_string(fileNum) + "a.dat");
    myfile.open(kfile);
    for (unsigned int i = 0; i < kSize; i++)
    {
        //output each cluster with -1 as third number, to tell gnuplot to
color the points the same
        myfile << clusterXVals[i] << "\t" << clusterYVals[i] << "\t" << -1
<< "\n";
    }

    myfile.close();
}

void Cluster::updateCoordMap()
```

```
{
    //iterate through map
    //calculate closest cluster for each coordinate and update k num in map


    static bool firstTime = 1;

    if (firstTime)
    {
        cudaMalloc(&d_coordXVals, sizeof(int) * coordSize);
        cudaMalloc(&d_coordYVals, sizeof(int) * coordSize);
        cudaMalloc(&d_coordKMaps, sizeof(int) * coordSize);
        cudaMalloc(&d_clusterXVals, sizeof(int) * kSize);
        cudaMalloc(&d_clusterYVals, sizeof(int) * kSize);
        firstTime = 0;
        srand(time(NULL));
        for (int i = 0; i < coordSize; i++)
        {
            coordXVals[i] = rand() % MAX_PT;
            coordYVals[i] = rand() % MAX_PT;

            //set k cluster for each coord to 0, will be changed later
            coordKMaps[i] = 2;
        }

        for(int i = 0; i < kSize; i++)
        {
            clusterXVals[i] = rand() % MAX_PT;
            clusterYVals[i] = rand() % MAX_PT;
        }

        cudaMemcpy(d_coordXVals, coordXVals, coordSize * sizeof(int),
cudaMemcpyHostToDevice);
        cudaMemcpy(d_coordYVals, coordYVals, coordSize * sizeof(int),
cudaMemcpyHostToDevice);
        cudaMemcpy(d_coordKMaps, coordKMaps, coordSize * sizeof(int),
cudaMemcpyHostToDevice);
```

```
    }

    cudaMemcpy(d_clusterXVals, clusterXVals, kSize * sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_clusterYVals, clusterYVals, kSize * sizeof(int),
cudaMemcpyHostToDevice);

    int blockSize = 128;
    int numBlocks = (coordSize + blockSize - 1) / blockSize;
    updateCoordKernel<<<numBlocks, blockSize>>>(coordSize, d_coordXVals,
d_coordYVals,
                                           d_clusterXVals, d_clusterYVals,
kSize, d_coordKMaps);

    cudaDeviceSynchronize();
    cudaMemcpy(coordKMaps, d_coordKMaps, coordSize * sizeof(int),
cudaMemcpyDeviceToHost);


}

void Cluster::updateClusterCoords()
{
    //iterate through k cluster vec
    //update cluster with average of each point in maps

    //copy cluster to compare for later
    int copyXCluster[kSize];
    int copyYCluster[kSize];
    for (int i = 0; i < kSize; i++)
    {
        copyXCluster[i] = clusterXVals[i];
        copyYCluster[i] = clusterYVals[i];
    }
    for (unsigned int i = 0; i < kSize; i++)
    {
        int sumX = 0;
        int sumY = 0;
        int count = 0;
        for (unsigned int j = 0; j < coordSize; j++)
        {
            if ((unsigned int) coordKMaps[j] == i)
            {
                count++;
                sumX += coordXVals[j];
                sumY += coordYVals[j];
            }
        }

        int newXVal = 0;
        int newYVal = 0;
        if (count != 0)
        {
           newXVal = sumX / count;
           newYVal = sumY / count;
        }

        clusterXVals[i] = newXVal;
        clusterYVals[i] = newYVal;
    }

    bool converged = true;
    for (unsigned int i = 0; i < kSize; i++)
    {
        if (copyXCluster[i] != clusterXVals[i]
         || copyYCluster[i] != clusterYVals[i])
        {
            converged = false;
            break;
        }
    }

    this->converged = converged;
}
```

Code (OpenMP Implementation)

Kmeans_omp.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
#include <vector>
#include "Cluster.hpp"
#include <unordered_map>
#include <ctime>

using namespace  std;

#define COL_SZ 2
#define REQ_ARGS 5
#define MAX_K 20
#define MAX_PT 250
static int K_SZ;
static int ITER;
static int NUM_PTS;
static bool inputFile;

void processInputs(int argc, char* argv[])
{
   if(argc != REQ_ARGS)
```

```
    {
        cout<<"Wrong inputs:\t\n"
        << "Usage: " << argv[0] << " <number of k clusters> <number of
points> <iterations> <inputFile>\n";
        exit(1);
    }

    K_SZ = atoi(argv[1]);
    if (K_SZ > MAX_K)
    {
        cout<<"Error: Only 20 or fewer k clusters supported\n";
    }

    NUM_PTS = atoi(argv[2]);
    ITER = atoi(argv[3]);
    inputFile = atoi(argv[4]);
}

int main(int argc, char* argv[])
{
    //initialize inputs
    processInputs(argc, argv);

    //initialize k clusters
    srand(time(NULL));
    vector<pair<Coord, int>> clusterInit;
    vector<Coord> kClusters;
    for(int i = 0; i < K_SZ; i++)
    {
        Coord newCoord(rand() % MAX_PT, rand() % MAX_PT);
        kClusters.push_back(newCoord);
    }

    //check to see whether to take data from input file or generate it
    if (inputFile)
    {
        ifstream inFile;
        inFile.open("dim032.txt");
        if (!inFile) {
            cout << "Unable to open file";
            exit(1); // terminate with error
        }

        int coordNum = 0;
        int input;
        int temp;
        while (inFile >> input) {
            if (coordNum == 0)
            {
                temp = input;
                coordNum++;
            }
            else
            {
                coordNum = 0;
                Coord newCoord(temp, input);
                clusterInit.push_back(std::make_pair(newCoord, 0));
            }
        }

        inFile.close();

    }
    else
    {
       //initialize coordinates and write to data file
        for (int i = 0; i < NUM_PTS; i++)
        {
            int xVal = rand() % MAX_PT;
            int yVal = rand() % MAX_PT;

            //initialize all coordinates with zero as cluster (will be set
later)
```

```
            Coord newCoord(xVal, yVal);
            // pair<Coord, int> newMapVal(newCoord, 0);
            clusterInit.push_back(std::make_pair(newCoord, 0));
        }
    }




    Cluster cluster(clusterInit, kClusters);
    clock_t start, end;
    double cpu_time;
    start = clock();
    //main loop
    for (int i = 0; i < ITER; i++)
    {
        cluster.updateCoordMap();
        cluster.writeToDatafile(i);
        cluster.updateClusterCoords();
        if (cluster.converged)
        {
            break;
        }
    }

    end = clock();
    cpu_time = (double)(end - start);
    cout <<"Kmeans time " << cpu_time / CLOCKS_PER_SEC <<endl;

}
```

Cluster.hpp

```
#ifndef CLUSTER_H
#define CLUSTER_H

#include <vector>
#include <unordered_map>
#include <string>
using namespace std;


class Coord {
private:
   int x;
   int y;

public:
   Coord(int xVal, int yVal);
   int getX();
   int getY();
   void setX(int x);
   void setY(int y);
   int getEuclideanDistance(Coord coord);

};

class Cluster {
private:
   vector<pair<Coord, int>> coordPairs;
   std::vector<Coord> kClusters;

public:
   Cluster(vector<pair<Coord, int>> clusterInit, vector<Coord> kClusters);
   void updateCoordMap();
   void updateClusterCoords();
   void writeToDatafile(int fileNum);
   bool converged;
   ~Cluster();


};

#endif
```

Cluster.cpp

```cpp
#include "Cluster.hpp"
#include <math.h>
#include <fstream>
#include <string>
#include <iostream>
#include <omp.h>

#define MAX_K_THREADS 2
using namespace std;

Coord::Coord(int xVal, int yVal)
{
   this->x = xVal;
   this->y = yVal;
}

int Coord::getEuclideanDistance(Coord coord)
{
   double p1 = pow(coord.x - this->x, 2);
   double p2 = pow(coord.y - this->y, 2);
   return (int) sqrt(p1 + p2);
}

int Coord::getX()
{
   return x;
}

int Coord::getY()
{
   return y;
}

void Coord::setX(int x)
{
   this->x = x;
}

void Coord::setY(int y)
{
   this->y = y;
}

Cluster::Cluster(vector<pair<Coord, int>> coordPairs, vector<Coord> kClusters)
{
   this->coordPairs = coordPairs;
   this->kClusters = kClusters;
   converged = false;
}

Cluster::~Cluster()
{
}

void Cluster::writeToDatafile(int fileNum)
{
   //iterate through map
   //writing each coordinates x and y and then cluster num to file
   ofstream myfile;
   std::string file("data" + to_string(fileNum) + ".dat" );
```

```
    myfile.open (file);
    for(unsigned int i = 0; i < coordPairs.size(); i++)
    {
        myfile << coordPairs.at(i).first.getX() << "\t" <<
coordPairs.at(i).first.getY() << "\t"
         << coordPairs.at(i).second << "\n";
    }

    myfile.close();
    std::string kfile("data" + to_string(fileNum) + "a.dat");
    myfile.open(kfile);
    for (unsigned int i = 0; i < kClusters.size(); i++)
    {
        //output each cluster with -1 as third number, to tell gnuplot to
color the points the same
        myfile << kClusters.at(i).getX() << "\t" << kClusters.at(i).getY()
<< "\t" << -1 << "\n";
    }

    myfile.close();
}

void Cluster::updateCoordMap()
{
    //iterate through map
    //calculate closest cluster for each coordinate and update k num in map
    #pragma omp parallel
    {
        omp_set_num_threads(omp_get_max_threads());
        #pragma omp for
        for(unsigned int i = 0; i < coordPairs.size(); i++)
        {
            Coord coord = coordPairs[i].first;
            int minDist = coord.getEuclideanDistance(kClusters.at(0));
            int count = 0;
            int index = 0;
            vector<int> distances();
```

```
           for(Coord cluster : kClusters)
           {
               int newDist = coord.getEuclideanDistance(cluster);
               if(newDist < minDist)
               {
                   minDist = newDist;
                   index = count;
               }
               count++;
           }

           coordPairs[i].second = index;
       }

   }

}

void Cluster::updateClusterCoords()
{
   //iterate through k cluster vec
   //update cluster with average of each point in maps

   //copy cluster to compare for later
   vector<Coord> copyCluster = kClusters;
   #pragma omp parallel
   {
       omp_set_num_threads(kClusters.size());

       #pragma omp for
       for (unsigned int i = 0; i < kClusters.size(); i++)
       {
           int sumX = 0;
           int sumY = 0;
           int count = 0;
           for (unsigned int j = 0; j < coordPairs.size(); j++)
           {
               if ((unsigned int) coordPairs[j].second == i)
               {
                   count++;
                   sumX += coordPairs[j].first.getX();
                   sumY += coordPairs[j].first.getY();
               }
           }

           Coord newCoord(0,0);
           if (count != 0)
           {
           newCoord.setX(sumX / count);
           newCoord.setY(sumY / count);
           }

           kClusters[i] = newCoord;
       }
   }

   bool converged = true;
   for (unsigned int i = 0; i < kClusters.size(); i++)
   {
       if (copyCluster.at(i).getX() != kClusters.at(i).getX()
        && copyCluster.at(i).getY() != kClusters.at(i).getY())
       {
           converged = false;
           break;
       }
   }

   this->converged = converged;
}
```

Gnuplot Script:

```
set palette model RGB maxcolors 10

set palette model RGB defined ( -1 "#FE0000", 0 "#0000FE", 3 "#9F6EAF", 6 "#6C5EE2", 8
"#00FEFE", 11 "#80FE96", 13 "#00FE00", 15 "#FEFE00", 17 "#FEA820", 20 "#FE74FE")

set terminal png

do for [i=0:8] {

    output_file = sprintf('file%d.png', i)

    data = sprintf('data%d.dat', i)

    k = sprintf('data%da.dat', i)

    set output output_file

    plot data u 1:2:3 w points pt 5 ps 1 palette, \

    k u 1:2:3 with circles palette

}
```