***K-Means Project Documentation***

***Tair Hassid, 301913505***

**Abstract:**

This solution is compatible for three processes or more.

Terminology:

n - number of points.

k - number of clusters.

numprocs- number of processes.

The master- process 0.

Using MPI, process 0 is the master that manages the whole program. All the processes run the K-Means Algorithm, including the master.

Functions implemented with CUDA use 512 threads per block because while testing the program on college computers using 1024 threads I ran into some problems.

Since n is a large number, I've decided to parallelize with CUDA functions that are supposed to iterate the n points, so that each thread works on one iteration- one point.

Simpler functions that iterate k clusters or simple other loops are executed using OMP.

**Steps:**

1. The master reads the input file.
2. The master packs the first line read from the file using MPI\_Pack, and broadcasts the package to all other processes using MPI\_Bcast. The information sent- n, k, t, dt, limit, qm.
3. Each process unpacks the package received.
4. Each process creates new MPI datatypes for the structs: Point, Velocity, Position and Cluster, using MPI\_Type\_create\_struct and MPI\_Type\_commit.
5. The master initializes k clusters with the k first points that was read from the file using OMP, and each cluster gets an id, also using OMP.
6. The master then divides the n points read from the file, equally to all processes. If n / numprocs leaves remainder, then the remainder, r, is a number between 1 and numprocs-1. Each process between 0 and r gets one more point.  
   The master sends the points to each process using MPI\_Send, and the processes using MPI\_Recv.  
   The master also broadcasts the clusters to the other processes using MPI\_Bcast.
7. At the beginning at each process all the points belong to cluster 0.
8. Each process (including the master) is running the K-Means algorithm, "for" loop running from 0 to t.  
   1. Update each point's position in time using CUDA- this function executes every iteration from 0 to t, and since n is a large number instead of iterating n points, using CUDA means each thread works on one point.
   2. "while" loop checks the 'termination' flag that indicates if there was at least one point that moved from one cluster to another on the last iteration and checks if number of iterations has reached the limit.  
      within the "while loop":
      1. From the second iteration- calculate the average between all points in each cluster- which is now the new center of the cluster. The master gathers the sums of points for each cluster from all other processes (using MPI\_Gather) and calculates the average.
      2. Find the closest center of cluster to each point and move the point to this cluster-using CUDA.
      3. Update the number of points in each cluster.
   3. The master receives the points from each process (using MPI\_Recv, other processes use MPI\_Send) and puts it in the array.
   4. The master evaluates the quality of the current situation. If the quality is less than qm we can finish the algorithm. The evaluation is made using OMP.

**Evaluation of Complexity:**

* + - 1. Updating the position of each point using CUDA:  
         This function iterates over the points and calculates the new location of each point. In the sequential solution the function is , with cuda .
      2. Defining the center for each point using CUDA:

This function iterates over the points and for each point finds the closest cluster and updates it to be this point's cluster. In the sequential solution the function takes , with CUDA .

* + - 1. Evaluating the quality using OMP:
* This function iterates over the clusters
* for each cluster uses findDiameter\*\*
* and iterates over the rest of the clusters to find the distance between all clusters .

\*\*The findDiameter function iterates over all the points in the specified cluster and for each point iterates all points to find the furthest one.   
  
The sequential solution gives complexity of

Say number of threads the CPU provides to OMP to be w, so:

The parallel solution using OMP gives complexity of

the outer loop iterates from 0 to t in intervals of dt, so overall the complexity of K-Means algorithm is