Parallel implementation of K-Means

Name: Ron Ginat  
Id: 209035450

K-Means Background:

*k*-means clustering is a method of [vector quantization](https://en.wikipedia.org/wiki/Vector_quantization), originally from [signal processing](https://en.wikipedia.org/wiki/Signal_processing), that is popular for [cluster analysis](https://en.wikipedia.org/wiki/Cluster_analysis) in [data mining](https://en.wikipedia.org/wiki/Data_mining).

K-Means is one of the most popular "clustering" algorithms. K-means stores k centroids that it uses to define clusters. A point is considered to be in a particular cluster if it is closer to that cluster's centroid than any other centroid.

The Problem:

Large datasets of points (two dimensions), each includes velocity that change their location every pulse. the goal is to find the best distribution of n points in k clusters by measure the quality (an average of diameters of the cluster divided by distance to other clusters).

Input data:

* **N** - number of points
* **K** - number of clusters to find
* **LIMIT** – the maximum number of iterations for K-MEAN algorithm.
* **QM** – quality measure to stop
* **T** – defines the end of time interval [0, T]
* **dT** – defines moments t = n\*dT, n = { 0, 1, 2, … , T/dT} for which calculate the clusters and the quality
* Coordinates and Velocities of all points

Solution:

The basis of the algorithm is a nested loop – the outer loop updates the current time of iteration (move the points by their velocity \* dT until stop terms are achieved OR max time – “T”), and the inner loop clustering “LIMIT” times the points (LIMIT times OR if there isn’t point that changed cluster from previous iteration).

After we get out from the inner loop, we check the quality – if the quality is below or equal to given QM, we finish the algorithm and we need to write the results to file.   
If the quality is above given QM, we continue to the outer loop, move the points by their velocity and go again.

Make the K-Means algorithm parallel instead of serial, thanks to calculates that not dependent each other’s.

The technologies to do so are:

1. MPI - Process Message Passing Interface.
2. Open MP - Open Multi-Processing - multi-threaded, shared memory parallelism.
3. CUDA - GPU Programming API created by Nvidia.

In my solution, I decided that every available process will make the inner loop calculations on his using Open MP and Cuda methods, process with id = 0 will handle the reminder.

In the outer loop, there is a diameter calculation that requires n2 iterations, so i divided the points between the processes at the optimal distribution, each process will iterate on as close as possible to equal number of iteration of the other processes (farther explaining in the implementation part and in the code, in a method called “calculateWorkForDiameterLoadBalancing”).

All the data managed by collecting (gatherv) and broadcasting by one process with MPI.

Pseudo Code:

If ( myid == 0 )

readFromFile();

MPI\_Bcast(info from file);

for ( double t = 0; t < T; t += dT ) //the outer loop

{

If ( its not the first iteration )

{

cudaRefreshPoints (every proc send his part of points);

MPI\_Gatherv(gather the points from all procs, id 0 has the remainder too);

MPI\_Bcast(broadcast the points to procs);

}

refreshClusters();//reset the clusters fields

for (int i = 0; i < LIMIT; i++)

{

cudaClassifiedPoints (every proc send his part of points and flag if point moved cluster);

refreshClusterByPoints();//master refresh the cluster by points

MPI\_Gather(gather the flags from all procs);

MPI\_Bcast(flags);

MPI\_Gather(gather the flags from all procs);

Id 0 merging the clusters from all procs;

MPI\_Bcast(clusters);

recalculateCenters(); // O(K) method so all procs doing it

all procs check the flags(break if true);

}

MPI\_Gatherv(gather the points from all procs);

MPI\_Bcast(points);

calculateDiameterOmp (every proc send his (calculated from the beginning) part of points);

MPI\_Gather(clusters);

Id 0 merging the clusters from all procs;

MPI\_Bcast(clusters);

quality = calculateQuality();

If ( quality <= QM )

{

If ( myid == 0)

writeToFile();

break();

}

}

Pre Information for Implemantation:

(**n** - number of points, **k** - number of clusters, **p** - number of proc, **tpp** - number of threads per proc, **tg** - number of threads in each GPU, **cuda penalty** - time to send to cuda including malloc and memcopy, **mpi penalty** - time to send with mpi)

Implementation:

Phase 1 – read from file

All the data store in text file. In this phase the master proc read the data from the file.  
 reading from file N points - **O(n)**.

Phase 2 – prepare for k-means algorithm, send the points and data

Using MPI:

1. The master proc broadcasting the information (N, K, T, dT, LIMIT, QM) to the other proc.
2. The master proc broadcasting all the points to the other proc.  
   As we measure in class, there isn’t much difference between sending a small or large amounts of data, so the runtime complexity here is **O(mpi penalty)**

Each proc malloc cluster array and initialize it with initiateClusters().

Each proc calculates everyone part of points and displacement in the large array of points – this info used in moving the points by dT and by classifying the points to clusters.

Each proc also calculates everyone part of points for the diameter calculation with calculateWorkForDiameterLoadBalancing().

Phase 3 – dynamic k-means algorithm

Phase 3.1 – refresh the points by their velocity (except the first iteration)

* Every proc send his part of points to cuda method cudaRefreshPoints() that refresh the points by their velocities - **O( + cuda penalty)**
* The master proc gather all the parts of the points array from other procs.
* The master proc broadcast the points.

In this point every proc have the updated array of points.

Phase 3.2 – classifying points (clustering the points by the closest clusters)

* Every proc send his part of points to Cuda method cudaClassifiedPoints() that clustering every point to the closest cluster, and change the boolean flag if there any change   
  (point moved from its last cluster).
* Each proc updates the clusters array by the updated points array with Open MP method UpdateClustersAfterCudaGroupOmp().
* The master proc gather all the booleans flags array and broadcast it.
* Master proc gathers the updated clusters of each proc.
* Master broadcast the large array of clusters and each proc merges the clusters to one array.
* All proc calculate the new center of each cluster with recalculateCenters().
* Each proc checks the Boolean flags for a true value that means we need to stay in the inner loop, when all flags are false – we can break from the inner loop.

Haviest methods in this phase are:  
cudaClassifiedPoints() - **O( + cuda penalty)**and UpdateClustersAfterCudaGroupOmp() - **O( + K\*tpp) ≈ O()**

Phase 3.3 – calculate clusters diameter

* The master proc gather all the parts of the points array from other procs, merge it and broadcast.
* Every proc using Open MP method calculateDiameterOmp() to calculate his part.   
  The method calculates the diameter of each cluster. The partition here for every proc is different then because here the partition is a nested loop that every outer loop variable is passing all the points from him and further on. So, if the partition was the first proc was iterate a lot more than the last proc.

I realized that if, for each point we need to go through all the points that follow in the array, then the sum of iterations is the sum of an arithmetic sequence (1+2+…+N).  
I calculated this sum, divided it to number of procs, and called it the optimalWork.  
then, I went with a loop and calculated the number of points that their work will be as close as possible to this optimum. Saved the number of points and displacement for each proc.

* Master proc gather the clusters parts from the other proc and broadcast it.
* Each proc merge the cluster array.  
  Heaviest operation is calculateDiameterOmp(). in average **O**

Phase 3.4 – calculate quality

* Using Open MP method calculateQuality(), every proc calculate the quality and check if the quality is under the given QM. – **O()**.

Phases 3.1,3.4 is in the outer loop – T times.

Phases 3.2,3.3 is in the inner loop- LIMIT times.

The complexity: