CSci 5451, S'21 Programming Assignment # 2 Due Date: March 17, 2021

This assignment will familiarize you with MPI programming and with some of the basic issues of parallel algorithm design. The main task is to program a parallel *distributed* version of the `K-means` algorithm. This very simple algorithm was discussed a little at the beginning of the semester. Its goal is to cluster a set of m samples (feature vectors of size n each) into K clusters.

1 Background: The sequential algorithm

In the following the samples are stored in a matrix V of size $m \times n$ (m samples of n features each), K is the number of clusters we want. Upon exit, the routine should return an id list `idx` of length m that contains the cluster labels of each sample, and an array `ctrs` of size $K \times n$, that contains the centroids (the means) of the clusters: `ctrs(k,:)` contains the centroid of cluster k (note that this information is a little redundant – but this is for convenience). Also: $counter(k)$ counts the number of samples that belong to cluster k and $sum\_pt(k,:)$ is the sum of all samples that belong to cluster k. As you may recall each iteration of the the algorithm has 2 phases in addition to an initialization step. These are described below (it may be helpful to compare this with the matlab implementation).

Phase 0: initialization. The algorithm begins by selecting in a pseudo-random set of K 'centers' for the clusters. This is done by a function which will be supplied.

Phase 1: get cluster ids. Given a set of K centers find for, each item i, the closest center k. Set $idx(i) = k$. Update `counter(k)`, the number of nodes i that are closest to k. Accumulate feature $V(i,:)$ into $sum\_pt(k,:)$. At the end of phase one $counter(k)$ contains the number of samples that 'belong' to cluster k and $sum\_pt(k,:)$ contains the sum of the corresponding samples.

Phase 2: reset centroids. We will now update the centers. Each $center(k)$ is changed to $sum\_pt(k,:)/counter(k)$ unless $counter(k)$ is zero in which we draw a new random center to replace it. This process is stopped if either `max_its` is exceeded or each center has not changed much (within a tolerance) from the previous iteration.
A matlab version of the algorithm has been written to describe these steps [it was rewritten from last time to be closer to the C-version you will implement.]

2 The parallel algorithm

The idea in Lab2 is to adopt a distributed memory viewpoint of the algorithm just described. We now have p arrays of equal or roughly equal size. We will still call m the size of each data set (so the actual total size is $\approx m \times p$). The main idea is that *the algorithm is synchorized so that the k centers are kept identical across the processes at each iteration.* Here are some (but not all) the changes to be made, with reference to each of the 3 parts discussed above.

Phase 0: initialization. Each node selects a set of K centers as before. Then these centers are averaged across the p processors. This clearly calls for a certain use of `MPI_All_Reduce`.

Phase 1: get cluster ids. Given a set of K centers find for, each item i, we need to find the closest center k. This part is purely local operation. Next we need to update `counter(k)`, the number of nodes i that are closest to k. Each process has a value for counter(k) (k=1:K) and clearly these must be added [again a `MPI_All_Reduce`]. Similarly, each processor has its own version of $sum\_pt(k,:)$ and these must be added.

Phase 2: reset centroids. We are now ready to update the centers. Each $center(k)$ is changed to $sum\_pt(k,:)/counter(k)$ unless if $counter(k)$ is nonzero. Note that the centers computed in this way will be the same across the p processors. If $counter(k) == 0$ we draw a new random center to replace it and we again need to average the results across all nodes.
Clearly, the processors will stop at the same time since they have the same view of the K centers.

3 What is provided

You will find on canvas (see LABs module):

1. A main driver called `main.c`
2. Two data set (US pollution data from 2016) - a small one for testing and a second one for doing real runs and collecting statistics for the lab. These are located in a scratch partition so you wont need to copy it (The provided driver will do the reading).
3. A set of auxilliary functions in a file called auxil1.c. In this set you will find a token function called `MyKmeans_p` with comments. This is where most of your work will be focused.
4. A matlab section which contains a matlab implementation of K-means [this is a modified version of something shown in class - to make look similar to a C-implementation]

4 Tests

As stated above a main program along with some auxiliary functions will be provided. The goal is to cut down on coding time and allow you to focus on the main function which is the `MyKmeans_p` function. The main program will have one processor read the whole data from file, divide it into equal parts and distribute these parts to the other processors. Your function should find a clustering of the aggregate of all the samples using all processes. The main program writes the cluster information in files located in OUT/. make sure to create this folder.
You will first test your code with the smaller dataset and $np = 4$. The purpose of this is to show that your program does indeed work. Do not change the main program for this test.

Once your code has been tested you will need to analyze its performance and see, for a large number of items to cluster what happens to the execution time. You will try to get the best running time by varying the number of processors and slots, as well as by optimizing your code. You can provide a plot (or a table) of the timing achieved [1]. For this test you will need to modify `main.c` to remove the unnecessary print statements, and to add calls to timing functions (use MPI_Wtime()).

5 What to submit and grading criteria

5.1 Submit:

(1) All source codes. These should be submitted in Canvas. There should be two main programs. The one provided [unchanged]. Call this `main.c`. Then you will also need to provide another main with stats [timing and standard deviation for sizes.] Call this `main_Stats.c`. You may then provide makefiles for both executables. main.ex and mainStats.ex. A README file explaining how to make the executables and run them may help but is not required if the makefile is self-explanatory.
(2) Provide a file called `Report` (or `Report.pdf`). This can be a PDF file with plots. It can also be a plain text file with tables. Here are some of the items you need to comment on.

1. Any specific comments you have on your implementation. For example did you use any `MPI_Barrier` commands? If so why were they needed? Document the MPI commands you had to use.

2. Provide an analysis for the execution time. You should always the same number of iterations, e.g., 30 to make sure the stopping criterion is based on `max_its` only (you can set tol to zero and `max_its` to 30.) What does the model tell you (consider situations of large n and small n relative to processor numbers).

3. Comments on the statistics you see. Timing, performance in terms of load balancing, etc.

4. Finally, any comments on what you could do to improve performance. Think about what you would do if you had to write a real 'production' code.

5.2 Grading:

1. **15 %** Style and documentation.

2. **30 %** Overall correctness of your sorting routines

3. **30 %** Correctness of the specific approach [i.e., how does it conform to what is being asked.]

4. **25 %** Quality of your report: your comments, your suggestions for improvements, etc..

[1] It is not clear how this will work given the number of students working on the project at the same time. But your analysis of the results you get should be on what you see