Parallel & Distributed Computer Systems

kNN Algorithm using MPI

Task

Implement in MPI a distributed brute-force all-kNN search algorithm for the k nearest neighbors of each point $x \in \boldsymbol{X}$.

The set of X points will be passed to you as an input array along with the number of points n, the number of dimensions d and the number of neighbors k.

Each MPI process *Pi* will calculate the distance of its own points from all (other) points and record the distances and global indices of the k nearest for each of its own points.

What to implement:

- A **sequential** version which finds for each point in a query set X the k nearest neighbors in the corpus set Y (X=Y in our version).
- An **asynchronous parallel** that will use the sequential code in p **MPI** processes and find all kk-NN of the points that are distributed in disjoint blocks to all processes.

Implementation

The purpose of the algorithm is to find a solution to the **limited memory** problem. This problem occurs cause of the big data base that make us unable to store in our system the matrix of the distances **D**, which has $\boldsymbol{size\ of\ X * size\ of\ X}$ elements.

Instead we break the data and **distribute** them in the different processes, and so every process will have a set of query points $\boldsymbol{xi}$ and a set of corpus $\boldsymbol{yi}$.

Afterwards, we **move the data along a ring**(receive new yi from previous send yours to next process) and update k nearest every time, until every process has receive **every available corpus set** $\boldsymbol{yi}$.

A. Sequential

In the sequential approach we are breaking the set of X, by using different indices to point in matrix X. Two nested while loops implement the code, one to point to the query set xi and the other one to yi. Keep on mind that, because in the sequential code we have only one **real** process, we will need $number\ of\ processes * number\ of\ processes$ to find the kNN of the X.

In matter of memory allocation, we use $size\ of\ X * dimensions$ for X matrix, $2 * size\ of\ X * k$ for the results, both ids and the distances of nearest neighbors, and a matrix of distances, which has size of $size/processes * size/processes$ instead of the initial $size^2$.

B. Parallel - MPI

In the parallel approach we are dealing with the problem in the same way, but we have real separate processes, that results to each process run the routine of finding distances and update kNN, only $number\ of\ processes\ times$. That is the main reason of the acceleration in results.

In matter of memory allocation, the set of X is distributed to the process in matrices xi of $size/processes * size/processes$. There also matrices of same size corpus yi and a "copy" of corpus zi, which we use to send and receive the data safely without package loss. We are keeping the results in the separate processes until the ring transfer ends, and afterwards we send them to process 0, which prints the results in a file.

***Note**: A problem that occurs and probably is not so obvious from the look of this algorithm is to save the right indices of the neighbors of its point, because of the data distribution and the differentiation between global and local indices. The problem is solved, but the way implementing that is not optimized and slows down the execution time.*

Results

I tested my program with small X and some regular grids 2 and 3 dimensional. It looked to works well and the results was right, but when I tried to run with big datasets like mnist or bigger regular grids my program crashed, because I exceeded the available RAM. I assume that maybe its my systems' limitations and tried it run it on AUTh HPC infrastructure and "killed" for big data bases. So, I probably have some mistakes in my code, that I couldn't find in time. The following results are for a 2D regular grid with 10k elements and 9 neighbors and a 3D regular grid with 1000 elements and 27 neighbors, the first one was the biggest data set I could run.

Dataset	Sequential	MPI (n = 2)	MPI (n = 3)	MPI (n = 4)
2D 10k	10.72	20.98	15.24	13.83
3D 1k	0.05	0.44	0.45	0.43

It's obvious that the sequential is faster, because the databases are pretty small and the parallelism doesn't make sense for that amount of data.

Run

You can find the code here: https://github.com/sandreadis/MPI

- Clone repository
- make clean
- make
- sequential ./bin/knn_serial "use 2 or 3 for 2d or 3d"
- parallel mpirun -n 4 ./bin/knn_mpi "use 2 or 3 for 2d or 3d"

Stylianos Andreadis (10022)