Distributed kNN with MPI

Assignment for the course of Parallel & Distributed Systems, ECE AUTh
Odysseas Sofikitis, 10130, sodyssea@ece.auth.gr, GitHub Repo

I. Introduction

Goal of the assignment is to implement a distributed brute-force *all-kNN* (k-nearest neighbors) algorithm with *MPI* (Message Passing Interface) in a ring-fashion manner of communication.

For the calculation of the distance between two sets of points the *EDM* [1] (Euclidean Distance Matrix) is used, which can be calculated by:

$$D = (X \odot X)ee^T - 2XY^T + ee^T(Y \odot Y)^T,$$

where X denotes the set of query points and Y the set of corpus. This equation can be translated in matlab code as:

$$D = sum(X.\hat{}2,\ 2) - 2 * X * Y.' + sum(Y.\hat{}2,\ 2).'$$

which is basically the code used by our implementation but in *C++*. Functions necessary for these calculations are in *utilities.hpp* file, while *knn.h/knn.cpp* contains only functions regarding the actual finding of the neighbors, *e.g. result struct, k-select [2], communication between processes and comparison of results*. A short script was also developed for the testing of the program, that generates a random set of points -the amount and dimensions of which are specified by the user- and produces a file in the format acceptable by the program.

II. Sequential Implementation

The set of query points X is given as command line argument through a file, whose first line contains the number of points and dimensions and each next one contains the value of each dimension of one point. The points are stored in a row-major format and are assigned arbitrary, zero-based indexes. The number of the desired neighbors, k, is also given as a command line argument. Having acquired this information, the finding of *kNN* is a simple calculation of the *EDM* and execution of *k-select* k times. The results are allocated to a *Knnresult struct*, which holds the indexes of the k-neighbors and the distance of the query point from these.

A special case must be noted in which a point has multiple neighbors with equal distance or the same point exists multiple times in the set. As it was already mentioned, the points are indexed so it is crucial to find the right index of these different points with

the same distance and not present them as one. This is fairly easy to achieve, as the finding of the neighbors is by default sorted and we only have to check if the distance of the new neighbor is equal to the previous one. If it is, we find the next point with the same distance and different index.

The results are printed in *seqOutput.txt* from the main function.

III. MPI Implementation

A) Distributing the points

The format of the data file does not change. Each process finds the lines of the file that it should read by knowing the total number of corpus points, *n*, and the number of processes used, *p*. That way, each process is given *n / p* points, except for the last one that also receives *n % p*. When the acquisition of data is complete, all processes call the *distrallkNN* function that uses the sequential *kNN* algorithm and also implements communication between processes.

Aside from the query points, the number of these, the number of dimensions and the number of neighbors, the total number of points is also given as an argument. This is used to calculate the maximum amount of points a process can receive at the communications that will follow, as the “last” process has not an equal amount of points as every other one. Therefore, while the average size of the corpus set in each communication is *n / p*, one process always receives the maximum size of *n / p + n % p*.

B) Calculating *kNN*

Each process uses *p* times the *kNN* algorithm for each own query points and the set of corpus it received from the “previous process”. For the complete calculation of *kNN* there are totally needed *p - 1* communications between processes. The first iteration of the algorithm is done independently, as the query and corpus points are the same for all processes.

The steps of the next iterations are the following and each has its own trivial and non-trivial problems to solve:

a) Wait for previous communications to end.
   i) Although non-blocking communication is used, we cannot be sure if the communication has ended while the *kNN* was being calculated.
   ii) Swapping of pointers is used for the new sets instead of memory copying mechanisms.

b) Initialize new communication.

   i) There is no need for a new transfer of points if it is the last iteration.

   ii) Each process pretends that sends and receives *maximumSize * d + 1*, but at *incoming[maximumSize * d]* there is stored the actual number of points of this corpus set. Obviously, if *n % p == 0*, this is redundant and we just allocate one more space for practically no reason.

c) Calculate *kNN* using the new corpus set.

d) Compare results of the new *kNN* with the already existing one.

   i) If at least one new neighbor is further than the current furthest neighbor, there is no need to check the rest of them.

   ii) If there is a new neighbor with distance less than the current furthest, it is placed in the correct order, so that the results are kept sorted.

   iii) In order to calculate the global index of the new point, an index converter is used that practically shows the starting process of this corpus set. Then, it is multiplied with the average number of points of each process and added to the local index value.

e. g. *If there is a total number of 23 points and 4 processes, then each process would have 5 points except for the last one that would have 8. The average size is 5, while the maximum is 8. If process 0 finds a new neighbor in the corpus set acquired during the second iteration, then this corpus set came originally from process 2 and represents the points with indexes 10 through 14. That means, to find the global index, process 0 would have to add 10 to the indexes of this particular corpus set. This can be calculated by multiplying 2, the id of the original process of this set, with 5, the average size. The finding of the id of the original process is achieved by this line of code:*

```
indexConverter = (selfTID - i + numTasks) % numTasks;
```

After the calculation of *kNN*, the results are gathered one by one in the main process and are outsourced to a file, *mpiOutput.txt*.

IV. Results

On *Figure 1* and *2*, we can see the results of the algorithm on random data sets created by *fileProducer.cpp*. The implementation was tested for a total number of points 20-10k and dimensions 10-1k. The *MPI* implementation performs almost the same as the sequential for small data sets, with communication taking up most of the time during the execution. However, starting with data sets of 10k points and more, it becomes clearly better, speeding up almost 3 times in comparison with the sequential.

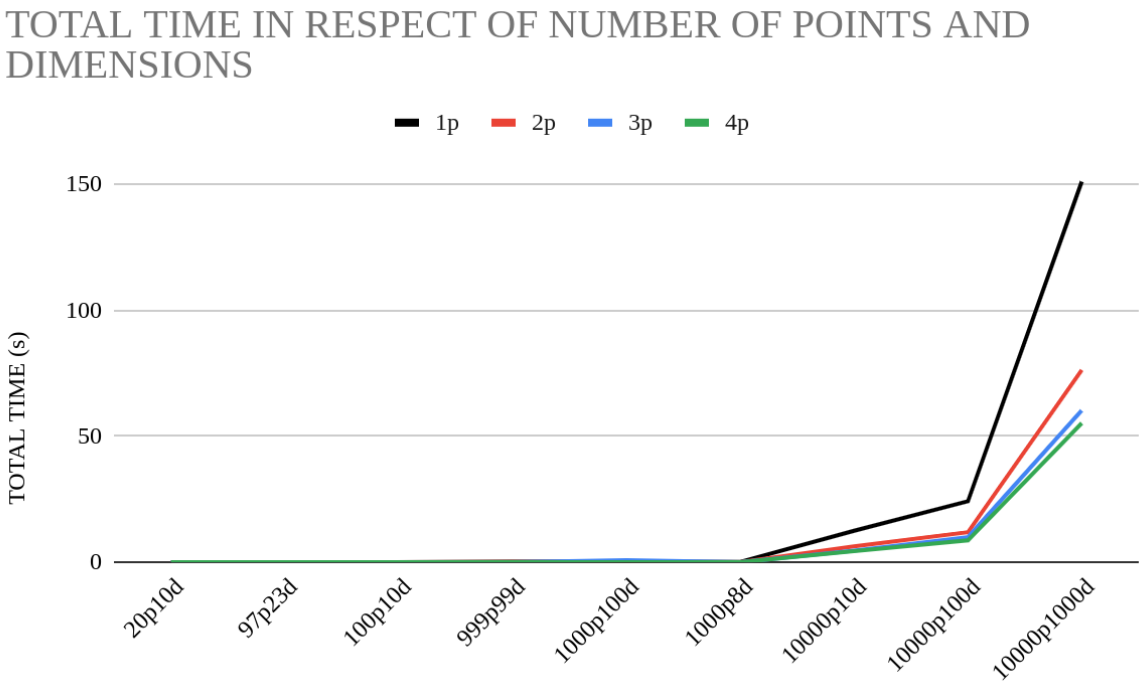

Figure 1. Total time of execution for p processes and data set of X points, Y dimensions

Although *MPI* can provide the optimization of a parallel algorithm on distributed systems, diminishing returns quickly show, as there is little speed up using 4 processes instead of 3, while the communication cost starts getting bigger.

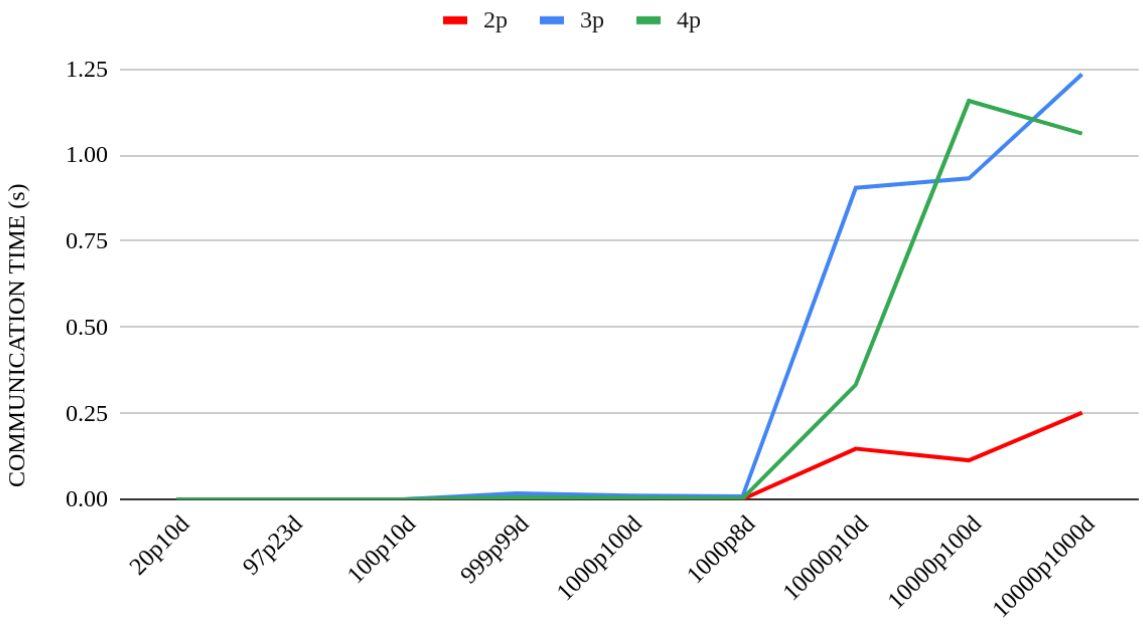

Figure 2. Communication time between processes

V. References

[1] Euclidean Distance Matrix

[2] K-select / Quickselect

[3] Lots of ChatGPT instead of googling