ARISTOTLE UNIVERSITY OF THESSALONIKI
FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Parallel and Distributed Systems
Semester 7th
"Finding the k-nearest neighbors in large datasets using OpenMPI"

Team: A
Student:
Liouliakis Nikolaos 10058

Thessaloniki
2022-2023

Main goal

The goal of this project was the creation of a serial program to find the k-nearest neighbors of a set of points (and their distances). Then to modify it using OpenMPI to make it able to solve problems for large datasets that would otherwise would not fit in the memory of a single computer.

Link to GitHub for the project:
https://github.com/Nick-Liou/Find_knn

Serial Approach - Algorithm Description

Regarding the serial implementation of the kNN algorithm, the code is written in C. Here are some important mentions:

Loading and reading the datasets

To run the program there needs to be a set of points, since there was not any specific format for the datasets, I used a txt file with a certain structure. Were in the first row there are 3 integers the number of points, the number of dimensions and the number of precomputed kNN. The number of precomputed kNN can be zero and is used for easily and automatic check the correctness of the code. Then there are n rows each with d numbers (they can be floating point numbers) the coordinates of the point and k indexes (zero-based) of the k-nearest neighbors separated by a comma. There are examples of this format in the GitHub repository for the code to run. I made a simple program in MATLAB that generates datasets, finds the kNN and stores them in the appropriate format.

Main algorithm

The main idea is to calculate the Euclidean distance matrix (EDM) of the of points and then for each point find the k smaller of their row (or column which are the same since it the EDM is symmetric). The EDM could be calculated using matrix multiplication and addition, but this would require storing all the EDM in memory which has a $O(n^2)$ space complexity. Instead, I chose to calculate one row of the EDM at a time, so this has a O(n) space complexity. This way the algorithm can be used for datasets where the number of points can be larger. It is worth noting that this could be optimized more by splitting the row in parts and after finding the kNN of each part then reduce them with the previous result. Another optimization that is used is that the square root was computed only in the result and not on each distance on the EDM (so to be precise it is not exactly the EDM that is computed).
The algorithm I used to find the k smallest elements and their indexes is not optimal but for small values of k the difference should not be very noticeable. It has an average time complexity of O(kn) but a best time complexity of O(n).

Checking the result with the precomputed kNN

After finding the kNN the result is checked with the precomputed kNN of the dataset (which in my datasets were calculated using MATLAB). If for the algorithm found more neighbors than the precomputed, then only the ones that were precomputed will be checked.

Asynchronous parallel - OpenMPI

Basic idea

Each process will have their part of the points to work on and find the kNN, a set of points that is receiving and a set of points that has already received that is working on and is sending in a ring topology. Each process in the first iteration sends and works on their part of the points. Since all processes to find the kNN need to compare their points with all of them there need to be p-1 steps on the ring, where p is the number of processes.

Main changes

The first main change is that each process must load its own points, so after one of the processes interacts with the user and gets the input about which file to load and how many kNN to find it sends that information to the rest of the processes. Then the processes check that all of them can find and open the file with the dataset. When the number of points is not divisible by p, the first p-1 processes get each $\lceil n/p \rceil$ points and the last process gets the rest $n - (p-1)\lceil n/p \rceil$ points. There are some edge cases where the number of points is small, and the number of processes is large where this would not work (when $n < (p-1)\lceil n/p \rceil$ it would not work but in the edge case where they are equal the last process would not have any points, but the program would work) but decreasing the numbers of processes solves the problem.

Send and Receive

Each process must calculate in which process it has to send and from which to receive data. Then to hide the communication cost it sends and receives data with non-blocking instructions and while the communication is taking place it starts working on its part of the problem then they wait of the communication to complete to be able to work with the next set of points. The processes must send and receive data p-1 times for the data to be propagated to every other process.

Reduce kNN

After each process calculates the kNN of its points with the received points it reduces the result with previous kNN it had found so far. The algorithm for the reduction is assuming that the reduced kNN as well as the two being reduced fit in memory (while the reduction happens). A different algorithm could had been implemented with a smaller memory footprint but there would be a significant tradeoff of the time complexity.

Checking the result with the precomputed kNN

After each process finds the kNN for their points they check them with the precomputed kNN of the dataset (if there were any).

Gather results

At the end after each process checks their results, they start sending their kNN to a specified process that gathers the result. Since there are two arrays to send the indexes and the distances first is a non-blocking send then calculates the square root of the distances and then sends them with a blocking send. The process that gathers the results is receiving with non-blocking instructions all of them and while it waits it find the square root of the distances and copies their result to the reduced result.

Improvements

Some improvements that could be made are:

- The processes could load more data than just their own. This would reduce the communications of the processes.
- Although a small improvement the processes could count the iterations and calculate which data they expect to receive.
- Lower memory footprint by combining the receiving and the already received/sending data and sending them in smaller pieces. This would make more complex the implementation, but it could solve bigger problems with fewer resources.
- Parallelize the program that each process is running.

Results

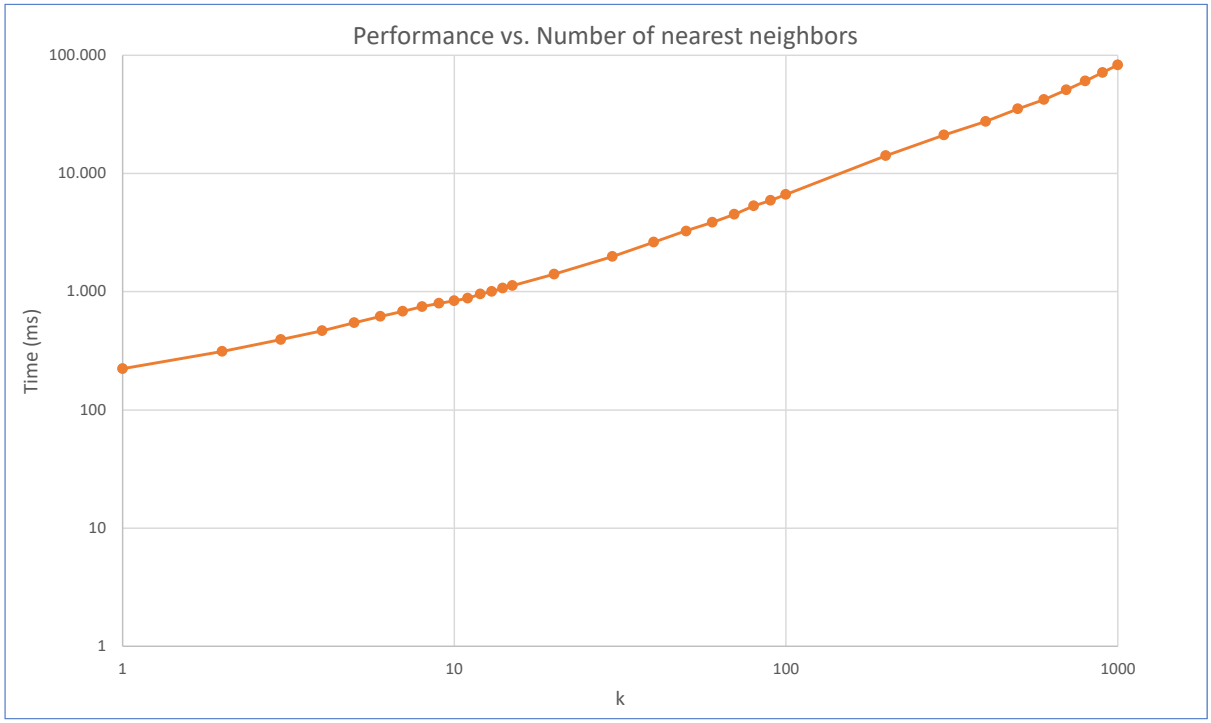

Figure 1 For the dataset11 (n=10000, d=2)

The algorithm implemented has a $O(n^2(k+d))$ time complexity and as expected we can see in Figure 1 the linear relation of the execution time in respect to k. In Figure 2 is shown an interpolation for the expected time given some sample points (both for the sequential and OpenMPI version), with more points and a better model the interpolation could be get more accurate. As is shown in Figure 3 OpenMPI is also reducing the execution time, it is expected since each process uses only a single thread, and the communication costs are hidden. If it was running on the same machine, it would be a way to parallelize the program but quite inefficient compared to other methods such as OpenCilk, OpenMP or pthreads. Note that the graphs use logarithmic scale on most of the axis. The execution times for those graphs have been produced using the Aristotle University of Thessaloniki (AUTh) High Performance Computing Infrastructure and Resources.

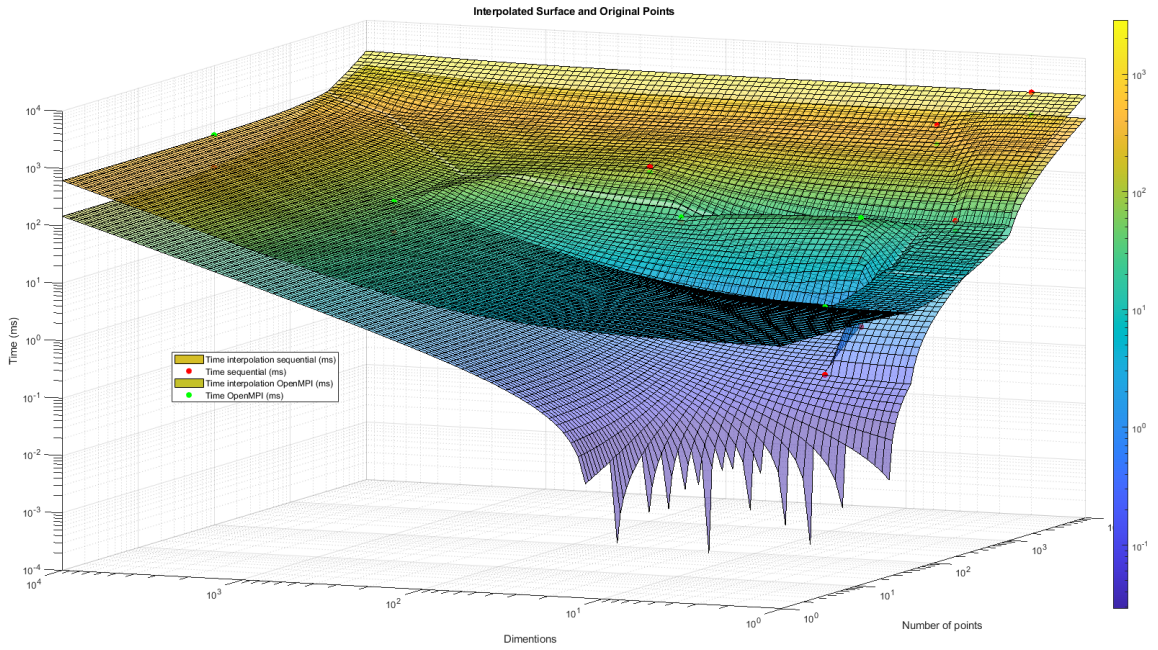

Figure 2 Time (ms) for various number of points and dimensions (for k=10)

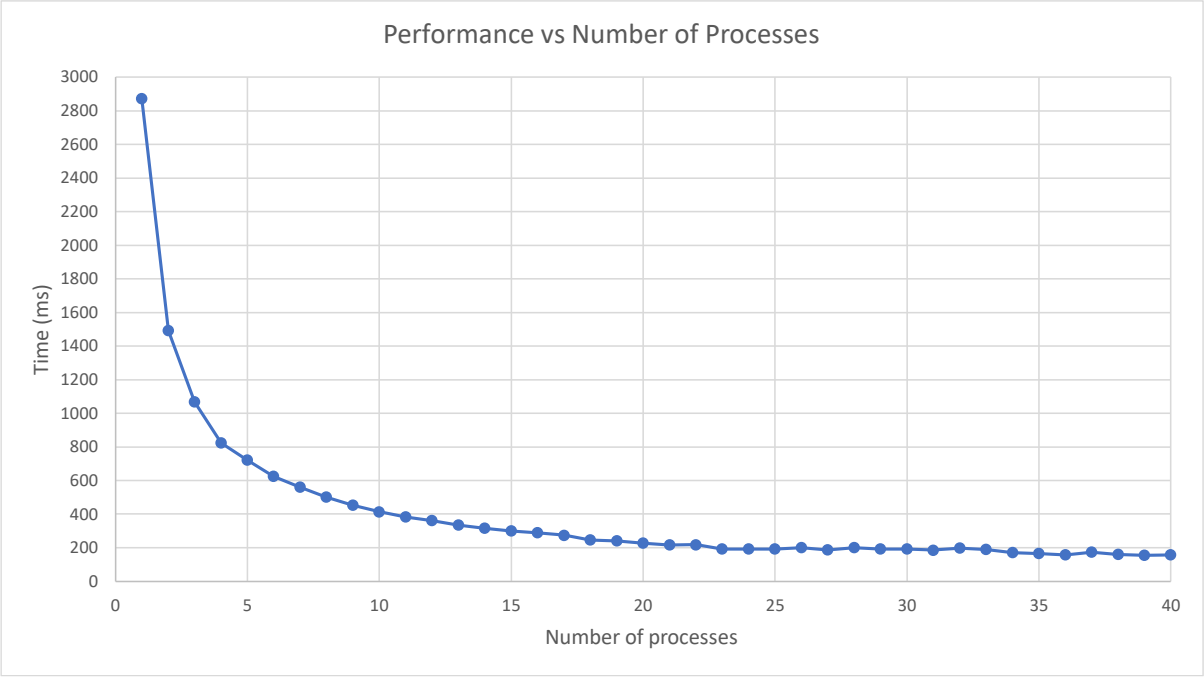

Figure 3 Execution time for different number of processes (for dataset11 n=10000, d=2)

Challenges / Difficulties

The most challenging part of this assignment was installing OpenMPI to successfully compile and execute a simple hello word example, I took 3 installations of Linux and a lot of help.

Resources

- ChatGPT
- Wikipedia
- OpenMPI documentation