Exercise II

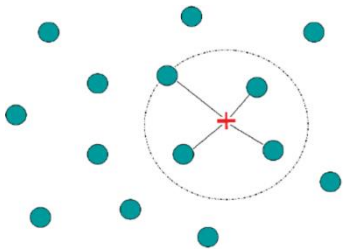
Parallel & Distributed Systems

An MPI implementation of KNN Search

Evripidis Chondromatidis 8527 eurichon1996@gmail.com

Problem

For a given data set X and a query set Y in the d-th Euclidian space we want to calculate the k nearest neighbors X for every point in the set Y using as metric the Euclidian distance.



Note: For every implementation that follows there is a further detail explanation which can be found in the code documentation.

Github: https://github.com/eurichon/KNN-MPI-Implementation

Sequential Solution

For this assignment a brute force implementation is chosen. Having that limitation in mind there two ways to speed up the calculations:

- By calculating efficiently, the distances between every point of Y to every point of X,
- And by efficiently select the nearest of all the neighbors for each point Y.

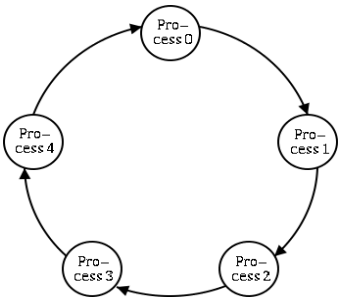
To tackle the first issue, we employ the highly optimized routines of OpenBLAS, so as to calculate the distances in the vectorized form shown below. More specifically we use cblas_dgemm for the mixed product X*Y calculation and cblas_ddot for the X.^2, Y.^2 calculation which we stored in two auxiliary tables Xsum and Ysum. That resulted in a speed up 10 - 70 times faster depending on the size of the problem. More dimensions produce better results.

$$dist(A,B) = \sqrt{\sum A^2 - 2\sum(A \times B) + \sum B^2}$$

Concerning now the second part, we will use the quickselect algorithm for every row of each point Y in the distance table to find the k-th nearest neighbor. However, luckily quick-select has the attribute to split the original set two subsets the left L and right R were Li <= k <= Rj for every i, j that belongs to the set. So, then we only need to perform quick-sort in the first k elements of every row reducing the overall complexity. Furthermore, to keep track the

indexes of the elements of X we perform all the previous calculation in the structure <<data>> which has the distance value and the index of the of the X element that gave us that distance. To accomplice this we overload the comparator of quick-sort and quick-select so as to perform the calculation with respect to the <<data.dist>> value.

MPI Synchronous solution



In this implementation the master process splits the data in other p-1 processes and then all the processes perform a p-1 series of data rotations and knn updates around a “ring” structure. To ameliorate the data rotation in this blocking implementation we can use the handy MPI function MPI_Sendrecv which performs send and receive of a buffer simultaneously and hides in the background the process of data transfer according to which we first transmitting the data from the even to the odd processes and vice versa. To accomplish that, we also use two extra auxiliary buffers: one to receive the data and one to transmit the previously received data while we keep the local unchanged. This will help to compute while exchanging data in the next section. For the calculation of the distances between the Y/p points and the X/p points we keep another k positions in each row which are initialized with a huge number. In every iteration each row’s positions from k to n + k is captured with the new data and in every update the smaller distances are pushed in the first k places due to the employment of quick-select / quick-sort. This structure overall reduces the computational complexity of these calculations and the memory usage of its process which is very import as the size of the problem many times cannot been hosted by a single machine.

MPI Asynchronous solution

In this section instead of using blocking communications which interrupt the program’s execution we employ non-blocking communications by using the MPI_Isend & MPI_Irecv functions. The purpose of this is to hide the communication cost, which in most cases is the bottleneck of an MPI program, behind the computational cost. To do this we create the function <<rotateData>>, which each time is called it can identify from its flags if the communication has begun, if it is finished of if it is still happening. Also, in the last case it can find in which stage we are OddToEven of EvenToOdd transmission by performing an MPI_Test. Including this function in the iterative calculation parts of the code we can simplify the process of hiding the communicating cost with a rather elegant way.

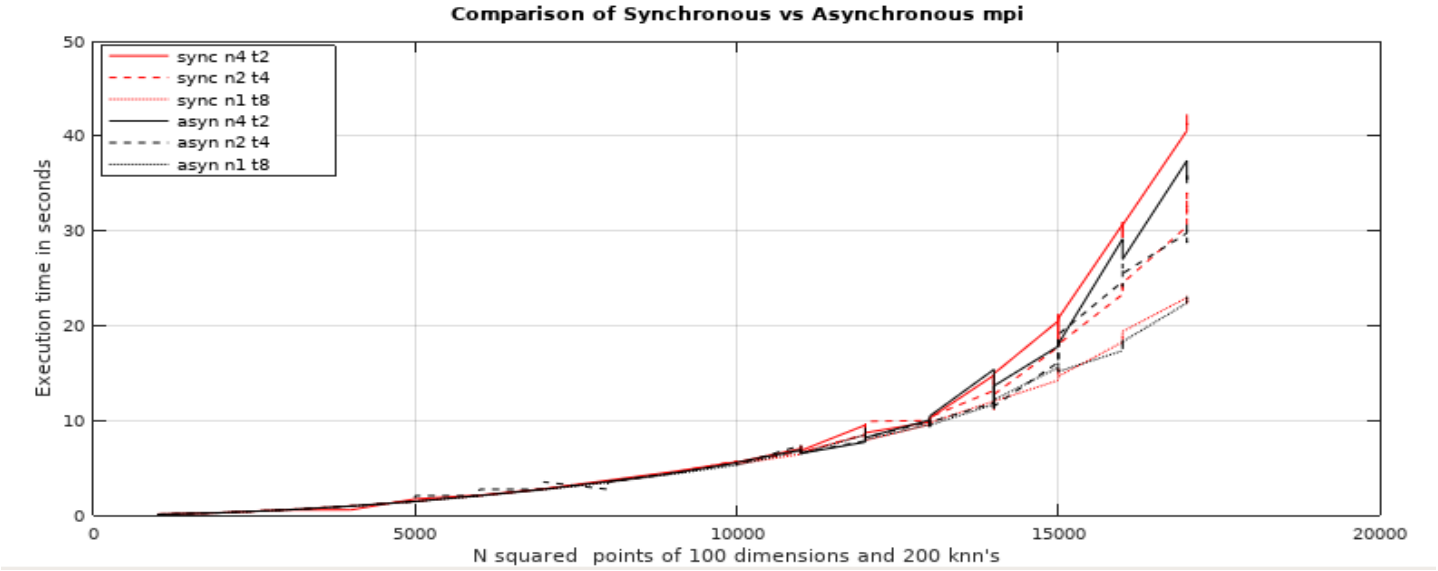
Global Reduction

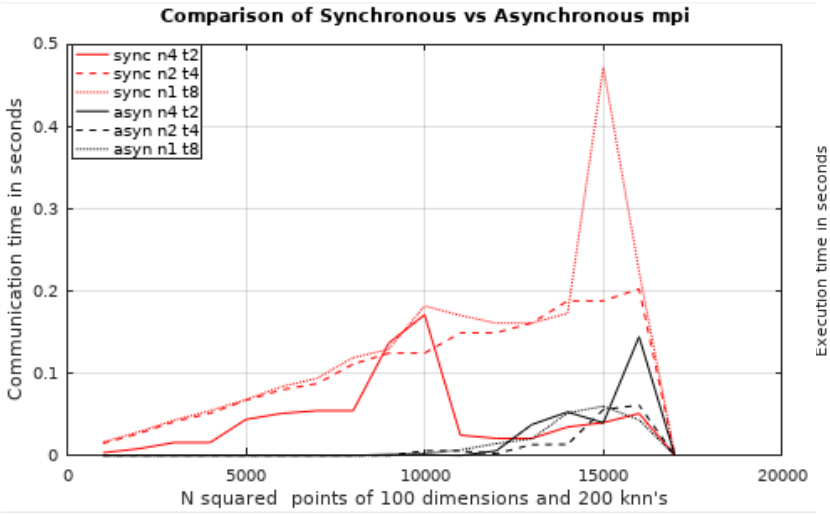
Lastly, we wish to keep the maximum and minimum distances that have been found in all the processes. We will use the general function MPI_Allreduce which in essence performs a reduction and immediately after an MPI_Bcast. To reduce this to one transmission for both min and max we create a struct which contains both called <<Limits>> and a custom MPI_Op function to handle the reduce of min and max simultaneously.

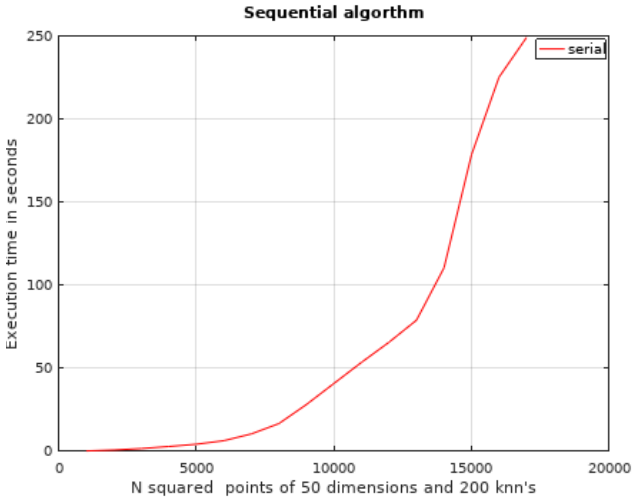
Results

BLAS SPEED UP (50 dimensions):

Dimensions/Samples^2	100^2	1000^2	10000^2
100	10.1794 times	37.4938 times	59.7479 times







Final Comments

- We can observe that in all the graphs there are a lot of spikes due to stochastic events.
- The sequential algorithm is very slow compared to the MPI implementations which use eight (8) processes with different combinations of nodes and task per node. Also, asynchronous has a slight advantage which should be magnified as the data becomes bigger, especially in the case where we execute the program in different nodes.
- We can confirm asynchronous has manage to attenuate the communicational cost in general and more notably in the case of four (4) different nodes where transmitting baud rate is significantly slower.