Exercise 2
Parallel & Distributed Computer Systems

Doinakis Michail
e-mail: doinakis@ece.auth.gr
Github link : https://github.com/doinakis/knn-ring-mpi

In the second exercise we were asked to implement a search and find an algorithm for the k nearest neighbors (k-NN) of each point in a set of points X. This set was passed to us along with an index array, that enumerate the points, the number of points n , the number of dimensions and the number of neighbors k.

V0. Sequential

The sequential implementation finds for each point in a query set Y the k nearest neighbors in the corpus set X. The functions that were used are the following :

1. double quickselect(double *arr,int * ids, int n, int k)

I used the same quickselect function as in the previous assignment but its a little bit different. It takes as arguments a double array, witch is an array with distances in our case, the ids of the points in its current state, the size of the array n , and the number of the smallest element we want to calculate, and returns the kth smallest element. The twist here,compared to what I did in the previous exercise, is that I pass the array of the ids and I move them exactly like the quickselect moves to find the kth smallest element so in the end we cam keep track of each point initial id .

2. knnresult kNN(double * X, double * Y, int n , int m , int d , int k)

The second function implemented returns a struct variable of knnsresult type, which has an int array nidx containing the ids of the nearest neighbors, a double array ndist containing the distances the number of query points m and the number of the nearest neighbors k. So what this function does is calculate the Euclidean distance between to sets of points , X and Y, where Y are the query points.We were asked to use high-performance BLAS routines so I used for this purpose the function cblas_dgemm , which makes matrix multiplication of this type D←alpha*Y*X+beta*D. How we converted our matrices to fit in this standard it is shown in the code written. To calculate the k nearest neighbors we call the quickselect function for every row of the D matrix and we call it k-1 times for each row. This way we end up with the k nearest neighbors sorted in the ndist array.

V1. Synchronous

The v1 corresponds to the mpi implementation of the problem. Basically what we do is move the data along a ring (always receive from the previous and send to the next process), which has P processes. **This way we can have all the points checked at P-1 iterations!** The way I approached the mpi implementation is the following, every process that has an even id (0 is considered even), first sends its own points and then it receives a corpus of points from an odd process. The odd processes first receive the corpus points and then they are sending the points to the others. This way we make sure we wont end up in a deadlock.

1.knnresult distrAllkNN(double *X,int n,int d,int k)

This function also returns a knnresult type variable.We already covered how the communications are done so we are just gonna analyze what exactly we are doing in this fucntion. Before we start communicating each process sets its own ids depending on their rank and the number of processes that are initialized. Process with id = 0 contains the latest values of the ids. Also everyone is calculating the distances with itself, and store it to a result variable of knnresult type . After a while, when an even process receives its coprus points we call the kNN function and store it to a comps variable of knnresult type. Then we check for every element in each row of the comps.dist if its bigger than the last element of the result.ndist if it is then there is no other element to check since the comps.dist are sorted so we break the loop.If its smaller then we set the last element to be the smallest and then sort again the array using quickselect(i tried to implement this part with swaps, so that I dont have to call the quickselect k times, but I couldnt get it to work as intended). The same comparisons take place for the odd ids too, with the only difference that in this case first we receive and then we send the points we had. In order to do that though we need one more array (corpustosend) so we send the correct corpus every time, otherwise we would send the ones we just received.

V2. Asynchronous

In order to try to hide all the communication costs, caused by the blocking nature of MPI_Send() and MPI_Recv(), we decided to do the same implementation with mpi again but this time using non blocking commands, the MPI_Isend() and MPI_Irecv(),while we still make computations. At this type of communication we need to make sure that only one send and one receive takes place at the time, so at the end of our computations we still use the MPI_Wait() function to make sure the communication ended.

1.knnresult distrAllkNN(double *X,int n,int d,int k)

The idea is the same as in the synchronous communications. The only real difference is that now we do not need to do the trick with the even and odd process ids due to the use of MPI_Send() and MPI_Recv(). Other than that the code is basically the same as the synchronous one. You can see the comments on the code for more information.

Global reductions and all-to-all

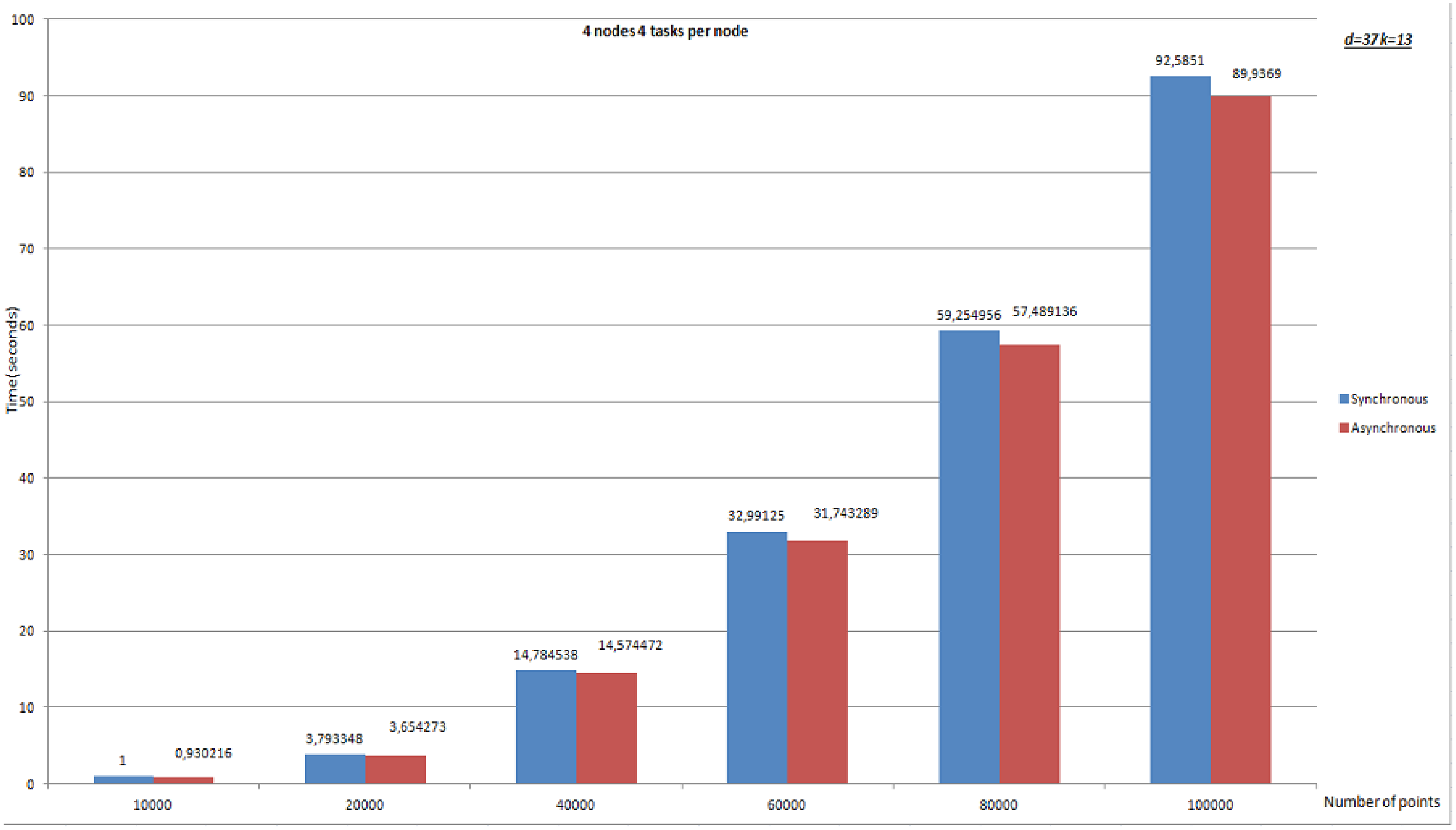
The global reduction part was pretty easy to add in the asynchronous part. At the end of the function we just declare two variables min_distance and max_distance for each process to have and two other variables to store the min and max value. Then we call MPI_Reduce for the minimum and the maximum and the values return to the process with id 0 and then we can work with them (for example print them). The reason that this part of the code is in comments is because it may affect the performance of the code in a bad way, so if you want to try it you can un-comment it.

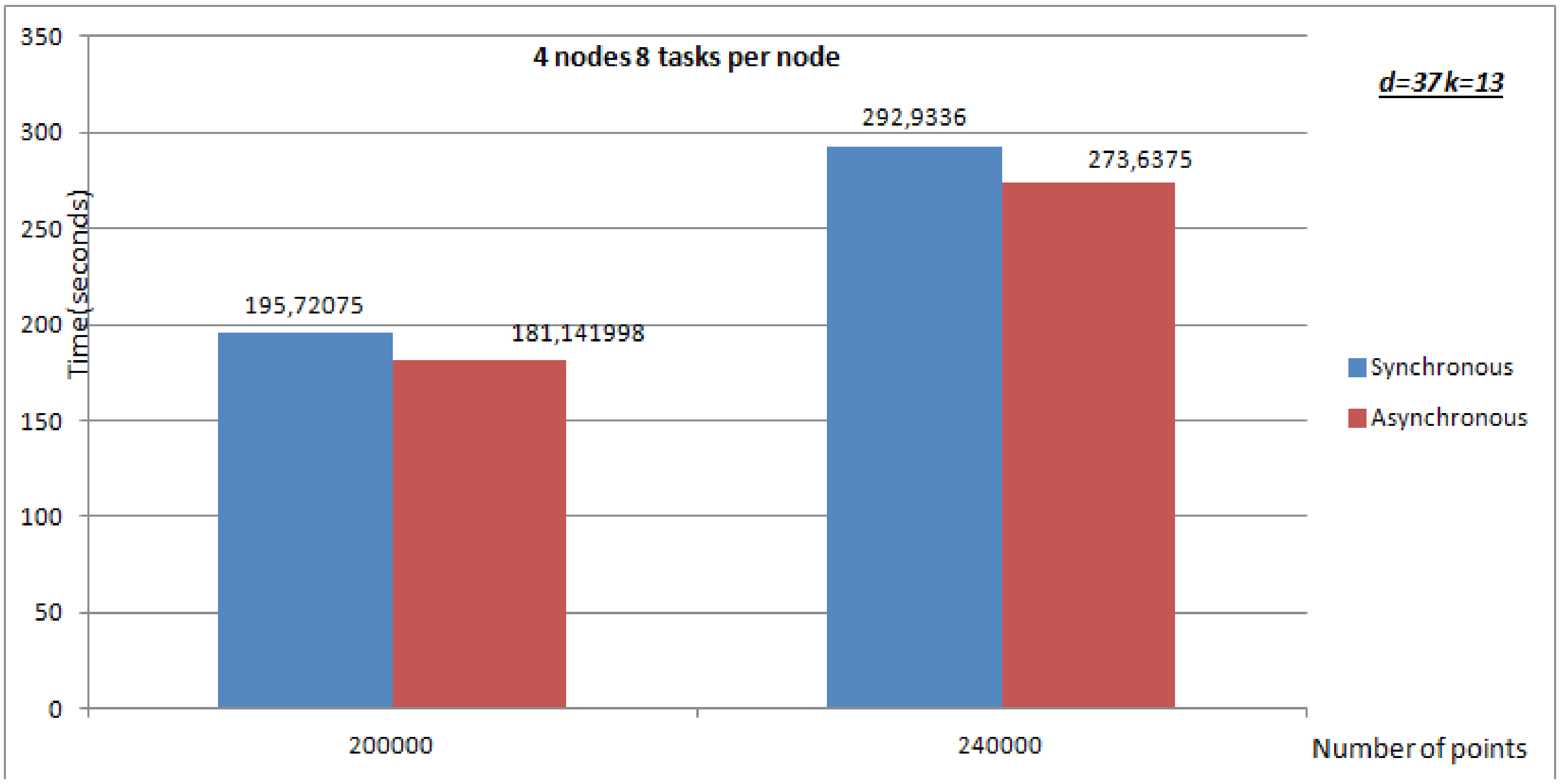
Run Times

The tests I ran were the following:4 nodes 1task per node, 4 nodes 2 tasks per node, 4 nodes 4 tasks per node and 4 nodes with 8 tasks per node always with the same d and k but with different number of points. As we can see from the diagrams, for low tasks per node we can’ t see significant difference between the synchronous and asynchronous implementation, but as we increase the number of points and the number of communication being done we can see an improvement up to 10 seconds. That’s probably because the time of the calculation is not high enough to hide the communication cost. Where we see the biggest advantage of the asynchronous code, is when we have to do lots of communication in last case with 8 tasks per node. As a result we understand that the asynchronous implementation can be much faster if the number of points is too high and the number of mpi processes is high as well. Also, the reason I couldn’t ran bigger test was due to the limitation of the cluster. When I increased the number of points to a certain degree, if the calculations took more than 5 minutes my jobs aborted. So the asynchronous implementation definitely benefits us,even for not than many points and processes, but it would be more obvious for bigger problems.

Notes

I had a problem with the online tester. Although locally it was running fine and I got correct answers at every implementation, when I tried to validate it online ,the sequential ran fine, but the mpi ones both had an error of **time limit exceeded.** The diagrams show the average time that took the processes to finish each time.Also, I couldn’t fit all the diagrams inside the report so I included 2 here and the other 2 can be found at my github link : https://github.com/doinakis/knn-ring-mpi/blob/master/diagrams.md


4 nodes 4 tasks per node
d=37 k=13
Time(seconds)
100
90
80
70
60
50
40
30
20
10
0
1
0,930216
3,793348
3,654273
14,784538
14,574472
32,99125
31,743289
59,254956
57,489136
92,5851
89,9369
10000
20000
40000
60000
80000
100000
Number of points
Synchronous
Asynchronous


4 nodes 8 tasks per node
d=37 k=13
Time(seconds)
350
300
250
200
150
100
50
0
195,72075
181,141998
292,9336
273,6375
200000
240000
Number of points
Synchronous
Asynchronous