Aristotle University of Thessaloniki
School of Electrical & Computer Engineering

Parallel and Distributed Computer Systems 2nd Assignment

Tsoumplekas Georgios, 9359, gktsoump@ece.auth.gr

Meta Louis-Kosmas, 9390, louismeta@ece.auth.gr

Github URL: https://github.com/lkmeta/KNN-search-algorithm

0. Abstract

This report is about the implementation of a kNN algorithm in both serial and parallel ways (for a distributed memory system). For a given set of points (in either a Euclidean space or not) the kNN algorithm calculates the k points nearest to each of these points $x \in X$. This is particularly useful for predictions in many real-life applications such as finance (forecasting stock market), medicine (predicting whether a patient will suffer from a particular disease), and security (face identification from a database). Moreover, kNN has been widely used in machine learning for classification purposes, although it not the fastest or the most accurate algorithm compared to SVMs or Neural Networks. However, the fact that it can be easily implemented and there is no need to train the algorithm makes it still attractive for real-life applications. In this particular assignment we implement 3 versions of the algorithm: the first one is a serial implementation, the second one is a parallel one using MPI to enable communications between processes with each one having its own memory and the third one is a variation of the second implementation where the points of each process are stored in a special data structure (Vantage Point Tree) for easier searching of the nearest neighbors.

1. V0 implementation

In this sequential version, we implemented the kNN algorithm the classic way where, for each point in a query set Y, we look for the k nearest neighbors in the corpus set X. In order to find the nearest neighbors of each point we have to calculate the all-to-all distance matrix. However, the size of this matrix grows rapidly when we have a lot of points and it is not practical to store the whole matrix. As a result, in order to save space, we examine each query point separately: we create a distance matrix just for this point and apply k-select in it to find its k nearest neighbors. Then, we repeat this procedure for every point in the query set. In order to increase the calculations' speed, we used the BLAS cblas_dgemm function to calculate the distance matrix of each point. Whenever sorting was needed, quicksort was preferred for bigger arrays and insertion sort was preferred for smaller ones since it is more efficient when the number of elements to be sorted is small.

2. V1 implementation

In this implementation we used the serial implementation of the knn algorithm to run on multiple processes in order to save memory and achieve faster execution time. The basic concept is that we break the query set into smaller parts and each process finds the k nearest neighbors of the points in the query sub-set that was assigned to it. In order to achieve workload balance, we split the query set in such a way that each process has the same or almost the same number of points (sizes differ at most by one element). At first, each process calculates the neighbors of its local points using as corpus points the local points themselves. Then, considering that the processes form a communication ring where each one communicates only with its previous (rank-1) and its next (rank+1) process, they send their points to the next one and receive new points form the previous one. The new points consist the new corpus sub-set in which knn is applied for the local points. The previously found nearest neighbors and the recently found ones are then combined and only the k ones closer to the examined point are kept. Then, each process sends the previously arrived points to the next process and the procedure is repeated until all corpus sub-sets have passed through every process. This implementation has the benefit of asynchronous communications: while we wait for the next points to arrive, we can calculate the knn with the ones we already have, thus saving a lot of time. Moreover, there is no need for huge amounts of memory to be allocated in one machine (even the query and corpus sets are big) since each process only has to store its local points which are a fraction of the total number of points.

3. V2 implementation

The V2 implementation uses the parallel and distributed memory methods of the V1 implementation and incorporates into them a specialized data structure, the Vantage Point Tree (VPT) to save the local points of each process. For the root of the VPT we implemented a method that chooses the most appropriate point from a sample of points in such a way that is as far from the rest of the points as possible and although it does not always give the best point (due to the small sample size chosen) it gives better results than picking a random point. We also set an appropriate number of minimum points each leaf node may contain in order to increase efficiency.

This whole process helps decrease execution time since, due to the structure of the VPT, we have to check less points to see if they are possible neighbors (thus calculating less distances between points). In the best-case scenario, searching is of logarithmic time complexity (just going down the tree, no intersections) and in the worst case it is just as bad as using a distance matrix (checking all corpus set points for each query set point).

Apart from that, the implementation is similar to that of V1 using MPI for asynchronous communications between processes. Only difference regarding this part is that, apart from sending points from each process to the next one we also send two matrixes accompanying those points: the first one contains the distances of those points' neighbors and the second one the indexes of those neighbors. This is done because we now consider the local points that create the VPT as the corpus subset points and the transferred points as the query subset points.

4. Experimental Results

All of the experimental results mentioned below occurred after testing our implementations in the AUTh High Performance Computing (HPC) infrastructure. The parallel implementations were executed on 2,4,8,16 and 20 cores on different nodes checking for 10,20,50 and 100 neighbors each time. Due to the big amount of data and diagrams produced, results here will be presented for 2 matrices (a smaller and a bigger one) out of the 10 different used. However, the results can be generalized for them as it can be seen by examining all diagrams for all matrices contained here.

First of all, regarding the serial implementation, it is clear that an increase in the number of neighbors k causes an increase in the execution time (see fig.1). This result makes sense since there are more elements to be sorted after the quickselect has been applied to each point. Although fig.7 may seem contradictory to that conclusion this is not the case since the decrease of execution time is very small and we can consider it as a marginal case where execution time stays the same.

Regarding V1, we can see that for both matrices this implementation shows a really good reaction regarding the acceleration when we increase the number of processes. Each time we double the number of processes the execution times becomes two times smaller as shown in fig.2 and fig.8. Additionally, fig.3 and fig.9 support this result, too. As we can see there is almost a 1:1 ratio between acceleration and the number of processes used. Last but not least, it is also important to notice that for a certain number of processes used, when we increase the number of neighbors k to be found the execution time stays almost the same as shown in fig.10 which is useful when the number of neighbors k is big.

Regarding V2, we can also see that an increase in the number of parallel processes used decreases the execution time. However, as we can see in fig.5 and fig.11 the decrease ratio is not as big as the one we had in the V1 implementation. This can be seen more clearly in fig.4 and fig.12 where the acceleration/number of processes is now smaller than 1:1 meaning that this implementation cannot react as well as V1 when increasing the processes. We can assume that this happens due to the bottleneck effect since V2 is already running a much faster than V1 due to the optimization occurring from the use of the VPT. Lastly, it is clear this time that an increase in the number of neighbors k causes an increase of the execution time as shown in fig.13. This result is expected since an increase in wanted neighbors translates in having to search more leaves of the VPT in order to find them, thus making the VPT not as effective for bigger values of k (but still more effective than searching the whole distance matrix). A solution to that problem could be that the minimum number of points B we have in each leaf increases proportionally to the number of elements k.

Finally, comparing V1 and V2 execution times (see fig.6, fig.14) we come to the expected result that the VPT implementation is significantly faster than searching the whole distance matrix. The execution time of V1 decreases much faster than that of V1 when we increase the number of processes, however it is still not enough to become as fast as V2. This is also backed up by the theoretical analysis since searching the whole distance matrix is the worst case scenario of the VPT and for every other case it is faster than that.

A) MiniBooNE_PID: n=130064, d=50

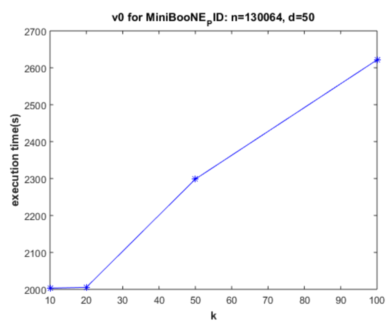

Fig.1

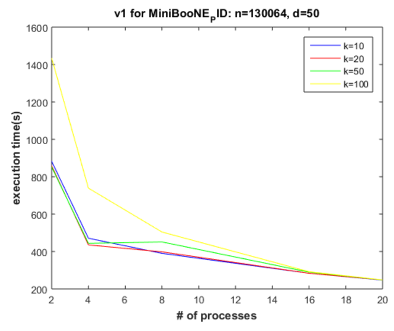

Fig.2

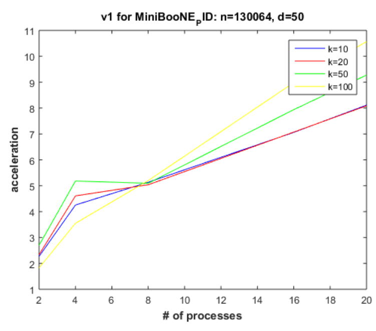

Fig.3

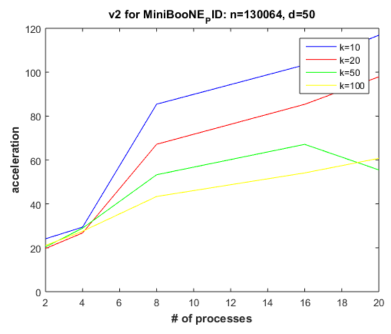

Fig.4

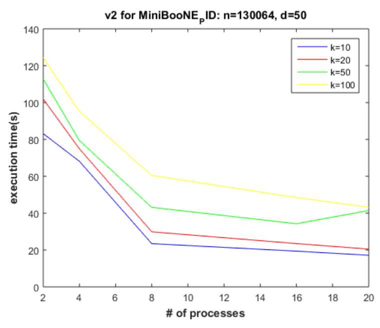

Fig.5

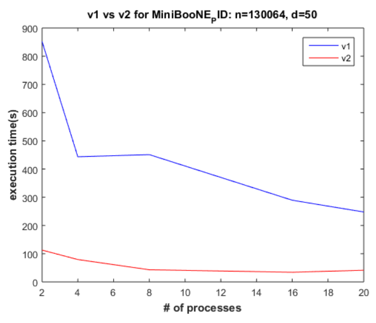

Fig.6

B) CNNIBN: n=33117 , d=17

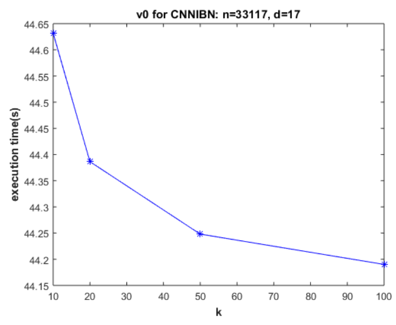

Fig.7

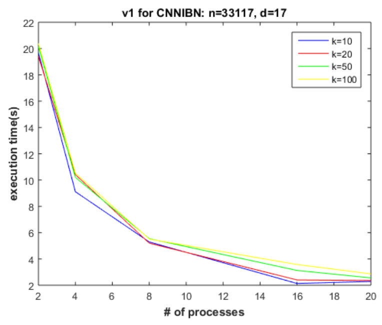

Fig.8

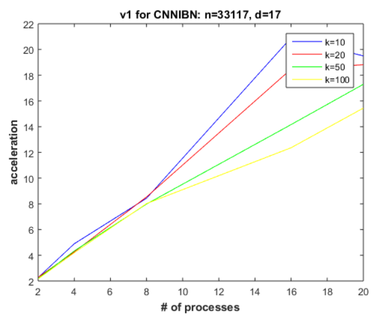

Fig.9

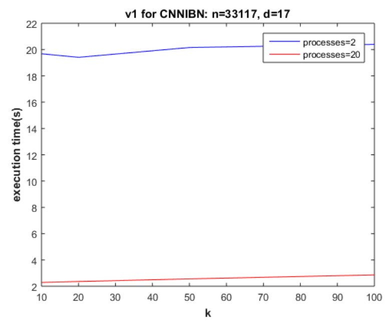

Fig.10

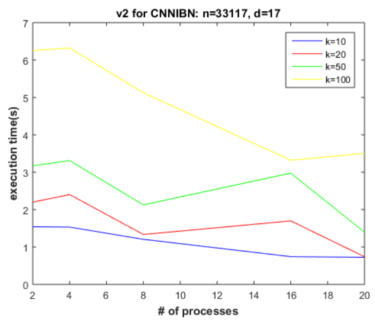

Fig.11

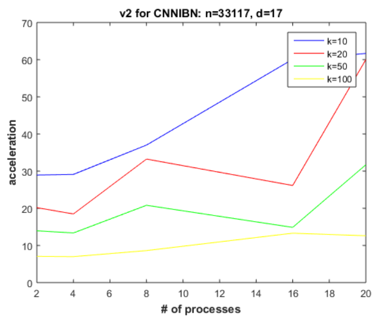

Fig.12

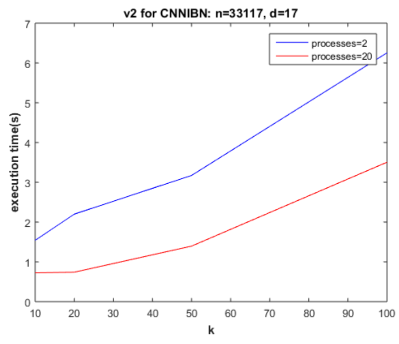

Fig.13

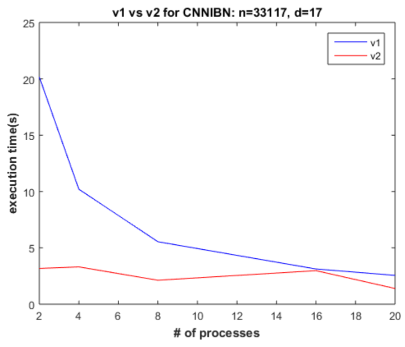

Fig.14

<u>Note:</u> Further explanation and analysis for each implementation mentioned above is available as comments in the source code.