k-Nearest Neighbors (k-NN) Implementation in Parallel

Epameinondas Bakoulas and Maria Sotiria Kostomanolaki

November 2024

Abstract

This document explains the implementation of the **k-Nearest Neighbors (kNN)** algorithm in C, both in serial and parallel execution. The focus will mainly be on the parallel execution using threads to improve the speed of the execution, assuming that $C = Q$, the number of points in the dataset is in the millions and the number of dimensions is in the hundreds. It will also display how we can efficiently manage the memory usage, by splitting the data into blocks.

1 Distance calculation

In a d-dimensional space, the **Euclidean distance matrix** D between two datasets C and Q is calculated as:

$$D = \sqrt{C.^2 - 2CQ^T + (Q.^2)^T}$$

where C is the **corpus** dataset with m points and d dimensions, and Q is the **query** dataset with n points and d dimensions. $C.^2$ and $Q.^2$ denote element-wise squaring of the matrices and summing along the columns. The function that implements the above is called `computeDistances`.

2 Sequential implementation

The `kNNsearch` function finds the k-nearest neighbors in the matrix C for each point in the matrix Q. First, it splits the Q matrix into *blocks* in order minimize memory usage. Then, it calculates the distances between the points in each block $(Q1, Q2, \dots)$ and the points in the corpus matrix C. Finally, it finds the k-nearest neighbors and their distances using the function `quickSelect`.

Our implementation aims to replicate the results of Matlab's `knnsearch` function, which is used to find the k-nearest neighbors. Thus, it will return 2 arrays `idx` and `dist`, where `idx` contains the indices of the nearest neighbors and `dist` contains the distances to the nearest neighbors. Since Matlab displays the nearest neighbors sorted, we also modified the `quickSelect` function to sort the k-nearest neighbors by distance.

3 Parallel implementation

We will assume $C = Q$ from now on and we will aim to find an *approximate* solution for the k-nearest neighbors problem for large datasets (number of points in the millions, and dimensions between 3 and 1000).

3.1 Initialization

We first create a struct called `Neighbor` that holds an *index* and a *distance*. The function `kNN` initializes an $n \times k$ array `nearestNeighbors` to store the nearest neighbors for each point. Each element in this array is initialized with a distance of *INFINITY* and an index of -1.

3.2 Splitting data into blocks

Controlling memory usage is crucial, so we split the data randomly into *blocks*. The randomess is achieved with the helper function `shuffleIndices` that takes an array of indices and shuffles them randomly. The first $n/numBlocks$ shuffled points are assigned to the first block, the next $n/numBlocks$ points to the second block, and so on.

For example, consider the matrix C ($n \times d$):

$$C = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 5 & 6 \\ 6 & 7 \\ 7 & 8 \\ 8 & 9 \\ 9 & 10 \\ 10 & 11 \\ 11 & 12 \\ 12 & 13 \\ 13 & 14 \\ 14 & 15 \\ 15 & 16 \end{bmatrix}$$

one possible split into 4 blocks is:

$$\text{Block 1} = \begin{bmatrix} 0 & 1 \\ 5 & 6 \\ 10 & 11 \\ 15 & 16 \end{bmatrix}, \quad \text{Block 2} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 12 & 13 \\ 14 & 15 \end{bmatrix}, \quad \text{Block 3} = \begin{bmatrix} 1 & 2 \\ 7 & 8 \\ 9 & 10 \\ 13 & 14 \end{bmatrix}, \quad \text{Block 4} = \begin{bmatrix} 3 & 4 \\ 6 & 7 \\ 8 & 9 \\ 11 & 12 \end{bmatrix}$$

3.3 Processing blocks

For each block, we calculate the distances between the points and update the nearest neighbors. Finding the k-Nearest Neighbors is achieved with the function `quickSelect`. It takes as input the neighbors of a point (an array of `Neighbors`) and quick selects the distances to find the k nearest neighbors. The blocks are processed in parallel. Each thread processes a block, until all blocks are processed.

After processing all of the blocks, the nearestNeighbors matrix (NN) will look like this (only first 4 points are shown, $k = 3$):

$$\text{NN} = \begin{bmatrix} (0, 0.0) & (5, 7.07) & (10, 14.14) \\ (1, 0.0) & (7, 8.49) & (9, 11.31) \\ (2, 0.0) & (4, 2.83) & (12, 14.14) \\ (3, 0.0) & (6, 4.24) & (8, 7.07) \end{bmatrix}$$

3.4 Improving the Solution

The solution is improved by finding distances between points in different blocks using a *subset* of the points. For example, if we combine pairs of blocks by choosing only 50% of the points in random (2 points from each block), we might get:

$$\text{Subset from Block 1} = \begin{bmatrix} 0 & 1 \\ 10 & 11 \end{bmatrix}, \quad \text{Subset from Block 2} = \begin{bmatrix} 2 & 3 \\ 14 & 15 \end{bmatrix}$$

We'll calculate the distance matrix using these points to update their neighbors. In the above example, we will find the distances between the points $[0, 1]$, $[2, 3]$, $[10, 11]$ and $[14, 15]$ (excluding the distances between the points in the same block) and update their nearest neighbors if we end up finding a new nearest neighbor.

Updating the nearest neighbors is achieved with the function `updateKNearestNeighbors`. It finds the (current) nearest neighbor with the biggest distance, and tries to replace it with the new neighbor. If the new neighbor is closer, it replaces the current one.

After this improvement, the nearestNeighbors matrix (NN) will look like this:

$$\text{NN} = \begin{bmatrix} (0, 0.0) & (5, 7.07) & (2, 2.83) \\ (1, 0.0) & (7, 8.49) & (9, 11.31) \\ (2, 0.0) & (4, 2.83) & (0, 2.83) \\ (3, 0.0) & (6, 4.24) & (8, 7.07) \end{bmatrix}$$

The block pairs are processed in parallel. Each thread processes a pair of blocks, until all pairs are processed. If we process the pair (i, j), we do not have to process the pair (j, i), since the distances are symmetric.

3.5 Analyzing the end results

The last step is to find the *Recall* and *Queries per second* of the algorithm. Recall is equal to the percentage of the correct neighbors that we find. It is calculated by comparing the ground truth (exact) solution of the algorithm with the approximate solution.

$$\text{Recall} = \frac{\#\text{ correct neighbors}}{\text{total neighbors}}$$

Queries per second is the number of queries that the algorithm processes in one second.

$$\text{Queries per second} = \frac{\#\text{ points}}{\text{execution time (seconds)}}$$

4 Benchmarks

We will display the benchmarks produced from testing the dataset *SIFT-128-euclidean*. It has a total of $n = 1,000,000$ points and $d = 128$ dimensions, and we'll find the $k = 100$ nearest neighbors. Tested on a computer with CPU: i5-11400F (6 cores, 12 threads), 16GB RAM, and OS: Linux Mint 22. The following graph summarizes the benchmarks for different implementations:

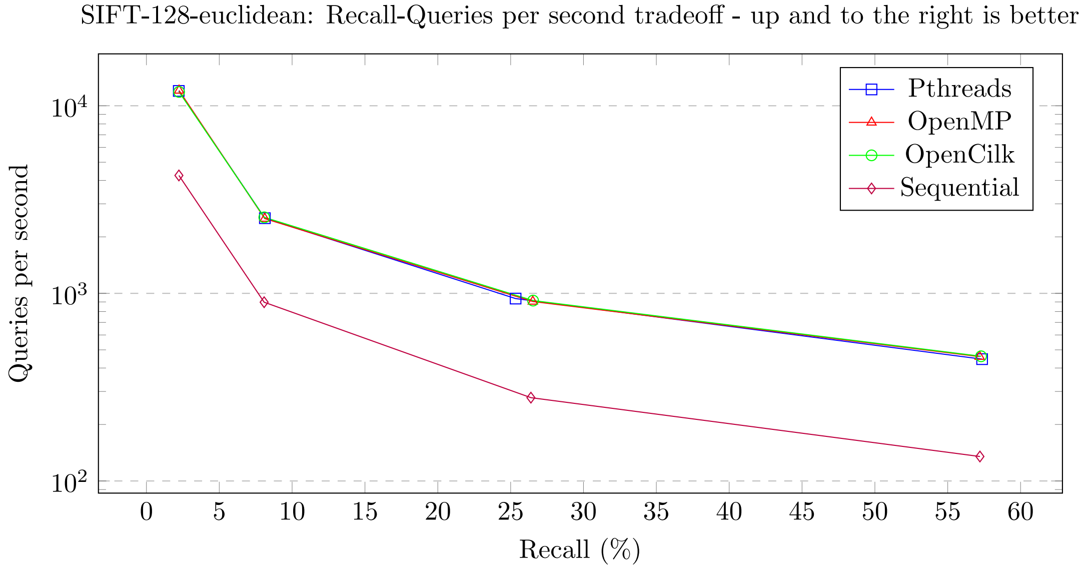

Figure 1: Queries per second vs Recall for different implementations (4 threads, 100 blocks)

All 3 parallel implementations produce very similar results. We can clearly see that the parallel implementations are much faster than the sequential one by a factor of **x2.8 - 3.4** using 4 threads.

To measure the recall we used only the first 10000 points of the dataset and we did not take into account the position of the neighbors in the `nearestNeighbors` array. We only checked if the neighbors were the same with the results produced by Matlab's `knnsearch` function. The correctness of this approach is guranteed by the fact that we introduced randomness in the algorithm, and also because n is very large.

What if we wanted to maximize the *Recall* for this dataset using the same algorithm? We tested it with OpenMP and the end results were Recall = 99.98% and Queries per second = 241. This result beats the last sequential result (Recall = 57.21%, Queries per second = 135) by a factor of ×1.7, both in Recall and Queries per second.

We can achieve even better performance by increasing the number of threads. Testing the same dataset with 8 threads and the results were slightly better, by a factor of $\times 1.1 - 1.2$ in Queries per second. This is expected due to hardware bottlenecks, but it does allow us to fully utilize the CPU and boost performance.

Comparing the benchmark results with `ann-benchmarks` for the same dataset, we can see that our algorithm is pretty competitive, taking into account that their tests were run on 31 threads. As it stands it only manages to beat `pg-vector`.

5 Algorithm correctness

The algorithm does produce correct results without bias, since there's randomness involved throughout the process. Block splitting is completely random, and picking points to create sub blocks is also random. In the case of large datasets, rerunning the algorithm with the same parameters will produce very similar *Recall* results with a very small deviation.

Data races may occur since `nearestNeighbors` is a global variable shared between the threads, but the error they might produce is marginal, compared to the overall result. The algorithm was tested with locks implemented to avoid data races, but it came at a high cost of speed. Thus, in this case, the speed of the algorithm takes a higher precedence.

6 Conclusion

You can find the source code of the implementation on Github: Go to repository

Detailed instructions on how to run the code and the results of the benchmarks can be found in the repository's README file.