# k Nearest Neighbors - kNN

Savvas Tzanetis - 10889

November 20, 2024

# Contents

# Chapter 1

# Introduction

The k-Nearest Neighbors **(k-NN)** algorithm is a fundamental technique for finding the closest neighbors of a query set **Q** with respect to a data corpus **C**. However, its computational complexity becomes prohibitive for large datasets, especially when the corpus and query set have high dimensions. This report presents a parallel implementation of the k-NN search algorithm by leveraging parallelism using libraries like **OpenMP, OpenCILK** and **PTHREADS** as part of an assignment in the Parallel and distributed systems class of the department of Electrical and computer engineering of the Aristotle university of Thessaloniki. This algorithm, implemented in the **C++** language uses the **OpenBLAS** library for optimized matrix computations for high dimensional distance calculations and the correctness of the implementation is validated against MATLAB's **knnsearch**.

## 1.1 Problem Analysis

The k-NN problem aims to compute the indices and distances of the k nearest neighbors for each query point in Q with respect to C. This is done by using the euclidean formula to calculate the distances between the query points and the corpus points and then, using a quick-select algorithm we retrieve the k smallest distances and their corresponding indices. The exact formult for calculating the distances is:

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

where:

- C is a matrix set of Corpus data points.
- Q is a matrix set of Query data points.
- D is the matrix set of distances between each pair of points Q and C, where each row will contain distances between a query point and all corpus points.
- The operations are element-wise.

# Chapter 2

# A Serial Implementation

## 2.1 Matrix Input

We begin by reading the C and Q datasets from a .hdf5 file, compatible with the Matlab application, or by generating random data points for both datasets, specifying the the number of points and dimensions:

```
// Ask for matrices size, dimensions and number of neighbors
cout << "Enter the number of points for Corpus: ";
cin >> c;
cout << "Enter the number of Queries: ";
cin >> q;
cout << "Enter the number of dimensions: ";
cin >> d;

// Generate random C and Q matrices
C.resize(c, vector<float>(d));
Q.resize(q, vector<float>(d));
for (int i = 0; i < c; i++) {
    for (int j = 0; j < d; j++) {
        C[i][j] = rand() % 100;
    }
}
for (int i = 0; i < q; i++) {
    for (int j = 0; j < d; j++) {
        Q[i][j] = rand() % 100;
    }
}
```

## 2.2 Calculating the distances matrix

Computing the distances matrix is fairly easy using the previously mentioned formula that represents the euclidean distance for each query point relative to each corpus point. Below is the **calculateDistances** function:

```
int c_points = C.size();     // Number of points in C
int q_points = Q.size();     // Number of points in Q
int d = C[0].size();         // Number of dimensions in a point
vector<double> CSquared(c_points);          // Vector for C^2
vector<double> QSquared(q_points);          // Vector for Q^2
vector<double> CQT(c_points * q_points);    // Vector for C * Q^T

// Create a flat vector for C and Q (Used for cblas_dgemm)
vector<double> CFlat(c_points * d), QFlat(q_points * d);
for (int i = 0; i < c_points; ++i) {
    for (int j = 0; j < d; ++j) {
        CFlat[i * d + j] = C[i][j];
    }
}
for (int i = 0; i < q_points; ++i) {
    for (int j = 0; j < d; ++j) {
        QFlat[i * d + j] = Q[i][j];
    }
}

// Calculate C^2 and Q^2
for (int i = 0; i < c_points; i++) {
    double sum = 0.0;
    for (int j = 0; j < d; ++j) {
        sum += C[i][j] * C[i][j];
    }
    CSquared[i] = sum;
}
for (int i = 0; i < q_points; i++) {
    double sum = 0.0;
    for (int j = 0; j < d; ++j) {
        sum += Q[i][j] * Q[i][j];
    }
    QSquared[i] = sum;
}

// Calculate C*Q^T
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, c_points,
            q_points, d, 1.0, CFlat.data(), d, QFlat.data(), d,
            0.0, CQT.data(), q_points);

// Calculate D^2 using (C^2 - 2C*Q^T + Q^2T)
D.resize(c_points, vector<double>(q_points));
for (int i =0; i < c_points; i++) {
    for (int j = 0; j < q_points; j++) {
        D[i][j] = CSquared[i] + QSquared[j] - 2*CQT[i * q_points + j];
        if (D[i][j] < 0) {
            D[i][j] = 0;
        }
    }
}
```

Explaining the code provided, we create 'flat' versions of the C and Q matrices as these are the preferred way of storing the matrices for using OpenBLAS library, we create vectors for storing the $C^2$ and $Q^2$ and then using the OpenBLAS library with **cblas_dgemm** we calculate a flat version of the distances matrix $D^2$, which we reverse back to a standard matrix layout later. The square root for each element in D is calculated later in the algorithm to save time.

## 2.3 Finding the k-Nearest Neighbors

Lastly, using the **quick-select** algorithm we sort the distances in each row of the distance matrix in **O(n)** time, identifying the k smallest values and returning them, along with their indexes.

```
for (int i = 0; i < Q.size(); i++) {
    vector<pair<int,double>> point_pairs;
    for (int j = 0; j < C.size(); j++) {
        point_pairs.emplace_back(j, D[j][i]);
    }

    quickSelect(point_pairs, k);

    idx[i].resize(k);
    dist[i].resize(k);
    for (int j = 0; j < k; j++) {
        idx[i][j] = point_pairs[j].first;
        dist[i][j] = sqrt(point_pairs[j].second);
    }
}

for (int i = 0; i < Q.size(); ++i) {
    for (int j = 0; j < k; ++j) {
        idx[i][j] += 1;
    }
}

return {idx, dist};
```

# Chapter 3

# Optimization Techniques

It is apparent that for a large number of data points as well as a high dimensional space, this technique can become increasingly slower, for that reason we will have to implement some optimizations to calculate an approximate result as well as leverage parallel optimization using **OpenMP, OpenCILK** and **PTHREADS.**

## 3.1 Dimension Reduction

An effective way for reducing the computational time, especially in higher dimensions, is to lower the dimensionality of the Corpus and Query datasets using the Johnson-Lindenstrauss Lemma. The Johnson-Lindenstrauss Lemma states that for any set of n points in a high-dimensional space, it is possible to project these points into a much lower-dimensional space, such that the pairwise distances between the points are approximately preserved. With the use of projection-based dimensionality reduction, we are able to make these approximations effectively. Specifically, for a set of c points in a d-dimensional space, we can project these points onto a subspace with dl dimensions, where:

$$dl = 3 * \frac{log(c)}{e^2}$$

From the Johnson-Lindenstrauss Lemma, we know that the distances between these points will be distorted by no more than a factor of $1 \pm e$.

```
// Perform k-NN search
int cp = C.size();
int qp = Q.size();
int dim = C[0].size();
int dl = 3 * (log(cp) / (e*e));

if((cp > 1000 && qp > 1000) && dl < dim) {
    vector<vector<float>> CS, QS;
    randomProjection(C, Q, dl, CS, QS);
    auto [idx, dist] = knnSearchParallel(CS, QS, k);
}
```

```
int c_points = C.size();
int q_points = Q.size();
int d = C[0].size();

vector<double> CSTemp(c_points * dl);
vector<double> QSTemp(q_points * dl);

vector<double> CFlat(c_points * d), QFlat(q_points * d);
for (int i = 0; i < c_points; ++i) {
    for (int j = 0; j < d; ++j) {
        CFlat[i * d + j] = C[i][j];
    }
}
for (int i = 0; i < q_points; ++i) {
    for (int j = 0; j < d; ++j) {
        QFlat[i * d + j] = Q[i][j];
    }
}


vector<double> projections(dl * d);
for (int i = 0; i < dl * d; i++) {
    projections[i] = (rand()% 2 == 0 ? -1 : 1) / sqrt((double)dl);
}

cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, c_points,
            dl, d, 1.0, CFlat.data(), d, projections.data(), dl,
            0.0, CSTemp.data(), dl);
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, q_points,
            dl, d, 1.0, QFlat.data(), d, projections.data(), dl,
            0.0, QSTemp.data(), dl);

CS.resize(c_points, vector<double>(dl));
for (int i = 0; i < c_points; ++i) {
    for (int j = 0; j < dl; ++j) {
        CS[i][j] = CSTemp[i * dl + j];
    }
}

QS.resize(q_points, vector<double>(dl));
for (int i = 0; i < q_points; ++i) {
    for (int j = 0; j < dl; ++j) {
        QS[i][j] = QSTemp[i * dl + j];
    }
}
```

## 3.2 Parallelism

Using the previously mentioned libraries for parallelization, we split the Queries dataset into smaller sub-Queries, based on how many cores the machine running the routine has, using this formula:

$$subQ = numProcessors * 0.6 + 1$$

where **subQ** is the number of sub-Queries and **numProcessors** is the number of logical processors of the system, using 60% of these logical processors plus 1. The number of points in each sub-Query is calculated with:

$$chunkSize = \frac{Qpoints + subQ - 1}{subQ}$$

In order to make sure all queries are being processed. After that, we calculate we execute our serial **knnSearch** function for each sub-Query with the Corpus data set and finally combine the results.

```
#pragma omp parallel for
for (int i = 0; i < num_subQs; ++i) {
    int start_idx = i * chunk_size;
    int end_idx = min(start_idx + chunk_size, q_points);
    subQs[i].assign(Q.begin() + start_idx, Q.begin() + end_idx);
}

#pragma omp parallel for
for (int i = 0; i < num_subQs; ++i) {
    auto [subidx, subdist] = knnSearch(C, subQs[i], k);
    for (int j = 0; j < subQs[i].size(); ++j) {
        idx[i * chunk_size + j] = subidx[j];
        dist[i * chunk_size + j] = subdist[j];
    }
}

#pragma omp parallel for
for (int i = 0; i < q_points; ++i) {
    for (int j = 0; j < k; ++j) {
        idx[i][j] += 1;
    }
}

return {idx, dist};
```

For the **OpenMP** and **OpenCILK** implementations, parallelism is also used for computing the $C^2$ and $Q^2$ vectors in the **calculateDistances** function, by parallelizing the needed **for** loops, using **#pragma omp parallel for** and **cilk_for** respectively.

# Chapter 4

# Results and Correctness Verification

In this chapter we will discuss the performance of our **k-NN** search algorithm in comparison to Matlab's **knnsearch** function. The system the testing of the algorithm and the comparison of its results to Matlab's was executed on, is consisting of a **6 Core - 12 Thread AMD 4650U** processor with a clock-speed of **2.1Ghz base, 4.0Ghz boost**, and **12Gb RAM** inside of a **WSL2 Ubuntu** image.

## 4.1 Results Accuracy

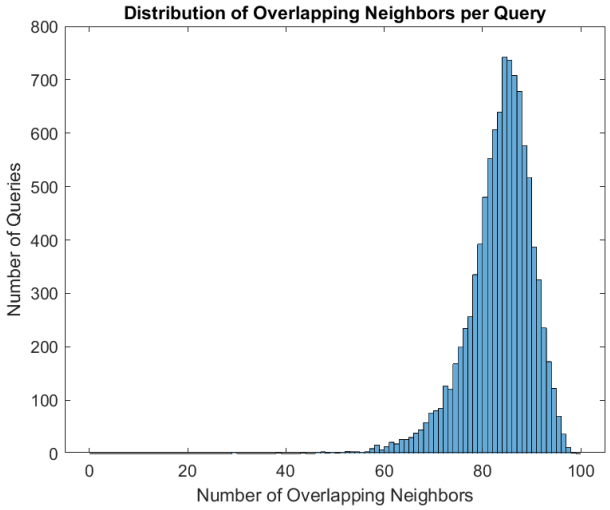
The results of the algorithm in comparison to Matlab's **knnsearch**, are 100% accurate for small datasets, while for larger datasets, using comparison metrics such as:

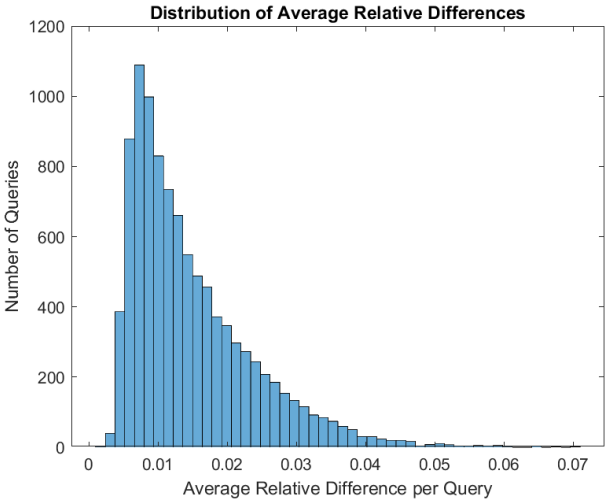
- Overlapping of neighbors to find how many neighbors of the approximate results are also present in the exact results.
- Distance comparison assessing how closely the distances from our implementation match those from Matlab's
- Precision at k, determining the proportion of our k neighbors that are also among MATLAB's top k neighbors for each query.

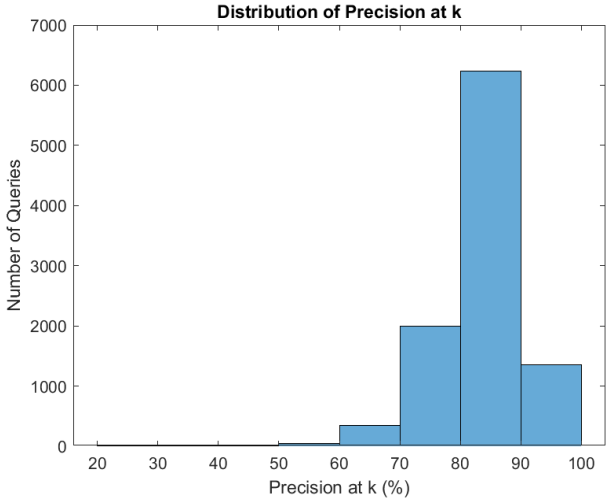
the results using the **OpenMP** implementation of our algorithm using the **mnist-784-euclidean** dataset from the **ANN Benchmarks** database for comparison, are:

- Average number of overlapping neighbors per query: 82.99 out of 100
- Overall average relative difference in distances: 1.48%
- Percentage of distances within 5.00% tolerance: 98.83%
- Average precision at k: 82.99%

These results were calculated using a Matlab script found in the project folder.




## 4.2 Performance

Lastly the performance of this algorithm can be calculated with two metrics:

- Raw execution time.
- Queries per second.

For the previously mentioned system, the dataset **mnist-784-euclidean** that was also used for accuracy metrics, runs with a mean time of 16 seconds, while Matlab's **knnsearch** runs in approximately 30 seconds.

- The parallel implementation presented is approximately **1.88x** times faster than Matlab's function.
- The parallel implementation presented handles approximately **625** queries/second.

From the aforementioned results, it is safe to conclude that this parallel implementation achieves significant speedup while maintaining high accuracy.

# Chapter 5

# Tools and Sources Used

## 5.1 Tools Used

- The **C++** programming language, as well as its standard libraries.
- The **OpenBLAS** library for fast matrix multiplications.
- The **HDF5** library for reading and writing **.h5** files, compatible with the Matlab application
- The libraries **OpenMP, OpenCILK, PTHREADS**.
- **WSL2** for running the code with Ubuntu from a Windows 11 machine.
- **GitHub**, as well as **GitHub Copilot**.

## 5.2 Sources

- ANN Benchmarks:
  - `https://github.com/erikbern/ann-benchmarks`
- Johnson-Lindenstrauss Lemma:
  - `https://en.wikipedia.org/wiki/Random_projection`
  - `https://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma`