Parallelization and Optimization of K-Nearest Neighbors (KNN) Search Algorithm

Maria Charisi
Eleni Koumparidou

Contents

1 Introduction to KNN Search Problem and Challenges **2**
1.1 Problem Definition . . . 2
1.2 Challenges . . . 2
1.3 Goals . . . 2
1.4 Testing . . . 2

2 Exact Approach of KNN-Search Problem **2**
2.1 Implementation . . . 3
2.2 PTHREADS specifics . . . 4
2.3 OpenMP Specifics . . . 4
2.4 OpenCilk Specifics . . . 4

3 Approximate Approach of KNN-Search Problem **5**
3.1 Parallelism in the Approximate Approach . . . 5
3.2 Threshold Calculation . . . 6

4 Testing **7**

Professor: Nikolaos Pitsianis
Department of Electrical and Computer Engineering
Parallel and Distributed Systems

1 Introduction to KNN Search Problem and Challenges

1.1 Problem Definition

A fundamental problem in computer science is the search for Nearest Neighbors between two distinct datasets, often referred to as the K-Nearest Neighbors (KNN) problem.

Given a set of query points and a collection of corpus points, the goal is to calculate the distance from each query point to every point in the corpus set, and subsequently identify the k nearest neighbors.

1.2 Challenges

While this problem is straightforward to solve when the dataset is small—both in terms of the number of points and the number of dimensions—significant challenges arise when the dataset scales. Specifically, as the number of points can exceed one million, and the number of dimensions can reach up to 1000, the computational cost and memory requirements escalate substantially. This makes it increasingly difficult for traditional serial algorithms to efficiently handle such large-scale problems, demanding advanced solutions with high memory and computational power.

1.3 Goals

Our aim is to provide efficient solutions to the K-Nearest Neighbors (KNN) search problem by leveraging parallelism through various parallel programming frameworks, including OpenMP, OpenCilk, and PTHREADS. These solutions aim to address both the exact and the approximate KNN search problem, ensuring effective handling of large-scale datasets. By utilizing parallel computing, we can significantly reduce the computational time required to process massive datasets, while maintaining an acceptable level of accuracy for approximate methods.

1.4 Testing

To evaluate the efficiency of the implemented algorithm, several key factors were considered, including execution speed, accuracy, and reliability. The performance of the code was tested using pre-existing benchmark datasets specifically designed for k-NN search, obtained from the ANN-Benchmarks repository.

2 Exact Approach of KNN-Search Problem

This approach guarantees highly accurate results by precisely determining the nearest neighbors using the Euclidean distance metric, ensuring optimal reliability and consistency in the computed outcomes.

2.1 Implementation

Distance Calculation using OpenBlas: A function named `distanceBlas` is created for computing a Distance Matrix (D) which contains the distances between a query and corpus set based on the following formula:

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

Where:

- C represents the corpus set.
- Q represents the query set.
- D is the resultant distance matrix.

This function is implemented with the use of the OpenBlas library, which provides high-performance implementations of basic linear algebra operations (BLAS), such as matrix multiplications.

Selection of Nearest Neighbors with QuickSelect: A naive approach would involve sorting the entire distance array for each query point, which incurs a time complexity of $O(n \log n)$. Given that the dataset may contain millions of points, this becomes infeasible for large-scale problems. To address this, we used the QuickSelect algorithm, which offers a more efficient solution. QuickSelect is a selection algorithm that partitions the distance array and recursively focuses only on the part containing the k smallest elements, thereby reducing the number of elements that need to be processed. The average time complexity of QuickSelect is linear, $O(n)$, making it much faster than sorting.

Multithreading Parallelism: To further enhance the performance of the algorithm mentioned previously, we implemented multithreading parallelism. The goal is to distribute the computational load across multiple threads, using the full potential of available CPU cores to speed up the execution. A function named `knnSearch` was developed to handle the multithreading aspect of the k-NN search algorithm. In this implementation, threads are dynamically allocated and initialized based on the number of available CPU cores. Each thread processes a small section of the Query set and a corresponding section of the Corpus set, enabling independent and simultaneous processing of multiple blocks of points.

Within each thread, the `distanceBlas` function is used to compute the Euclidean distances between the corresponding points in the Query and Corpus sets. This method significantly reduces memory allocation requirements. Rather than computing and storing a massive Distance Matrix with dimensions equal to QUERY $\times$ CORPUS, the approach dynamically allocates only a small block of the matrix for each thread. This block contains the distances relevant to the specific subset of the Query and Corpus that the thread is responsible for. Once the thread completes its task, the allocated memory is deallocated, ensuring efficient memory management and avoiding the overhead of holding a large distance matrix in memory at once. Afterward, the QuickSelect algorithm is applied to identify the k-nearest neighbors for that specific block.

However, dividing the Corpus set into smaller sections means that the k-nearest neighbors identified within each block are locally nearest neighbors rather than the smallest overall

neighbors across the entire corpus. As a result, the k-neighbors found by each thread only represent the closest points within their respective block of the Corpus set.

To address this, after gathering all k-nearest neighbors from each block, the algorithm invokes QuickSelect once again to find the true k-nearest neighbors across the entire corpus. This ensures that the k-neighbors for each query point are the closest points overall, considering all blocks processed by different threads.

2.2 PTHREADS specifics

In PTHREADS implementation, multithreading is initialized with the `pthread_create` function. For each block of the Query and Corpus sets, a new thread is created to handle the respective computation. The function `knnSearchThread` is invoked within each thread, processing a specific block of data.

```
pthread_create(&threads[i*cThreads+j], NULL, knnSearchThread,
                (void*)&thread_data[i*cThreads+j]);
```

Once all the threads have completed their respective tasks, the results are combined after `pthread_join` is called to ensure that no other thread is still processing.

```
pthread_join(threads[i*cThreads+j], NULL);
```

2.3 OpenMP Specifics

In the OpenMP version, parallelism is initialized using the #pragma `omp parallel for` directive, which allows the parallel execution of loops. In this case, the parallelism is applied twice: first to divide the Query set into chunks and then to divide the Corpus set into smaller sections for processing. The `num_threads` clause is used to define how many threads should be utilized for each loop.

```
#pragma omp parallel for num_threads(qThreads)
    for(int i=0; i < qThreads; i++)

#pragma omp parallel for num_threads(cThreads)
    for(int j=0; j<cThreads; j++)
```

2.4 OpenCilk Specifics

In OpenCilk implementation, the `cilk_for` directive is used to create parallel loops. Similar to OpenMP, the `cilk_for` loop is applied twice: once to partition the Query set and again to partition the Corpus set.

```
cilk_for (int i = 0; i < qThreads; i++)
    cilk_for (int j = 0; j < cThreads; j++)
```

3 Approximate Approach of KNN-Search Problem

To enhance performance on large datasets, we implemented an approximate version of the algorithm. Our approach involves building a VP-Tree for the corpus dataset and subsequently querying this VP-Tree to identify the nearest neighbors for each input query.

A VP-Tree, or Vantage-Point Tree, is a binary tree structure optimized for searching within metric spaces. The key idea behind a VP-Tree is to recursively partition the dataset based on distances from a selected reference point, referred to as the vantage point.

At each level of the tree:

- A vantage point is chosen from the dataset.
- The distances from this vantage point to all other points are calculated.
- The points are divided into two subsets: those within a specified radius (forming the left subtree) and those outside the radius (forming the right subtree).

This recursive partitioning creates a hierarchical structure that allows efficient pruning of the search space during queries. To find the k-nearest neighbors of a query point using a VP-Tree, the search begins at the root node. The process involves calculating the distance between the query point and the vantage point associated with the current node. Based on this distance and the radius of the current node, the search proceeds as follows:

- **Distance greater than the radius plus the threshold:** In this case, the query point is too far to be within the left subtree. The search continues in the right subtree.
- **Distance less than the radius minus the threshold:** The query point is close enough to potentially find neighbors in the left subtree. The search proceeds in the left subtree.
- **Distance within the threshold range:** The query point lies near the boundary of the current node's radius. In this case, both subtrees are searched to ensure no neighbors are missed.

During the search process, we maintain a record of the indices and distances of the vantage points that have been evaluated. Potential neighbors are stored in a max-priority queue, where elements are ordered based on their distances to the query point. Once the search concludes, the priority queue contains the k nearest neighbors of every query.

3.1 Parallelism in the Approximate Approach

To further enhance performance in our approximate solution, parallelism was incorporated into key sections of the code. Specifically, the construction of the VP-Tree leverages parallelism within the recursive process, enabling the simultaneous partitioning of data and reducing the overall computational time.

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        #pragma omp task
            buildVPTree(&Left, &(*node)->left,
                        totalDistance, totalNodes,&L_kindex);
        #pragma omp task
            buildVPTree(&Right, &(*node)->right,
                        totalDistance, totalNodes, &R_kindex);
    }
}
```

Moreover, instead of performing the search on the VP-Tree sequentially for each query, we implemented a parallelized approach. The query set was divided into chunks, and the nearest neighbors for each chunk were calculated simultaneously. Here is an example using OpenMP framework:

```
#pragma omp parallel for
for (size_t i = 0; i < Q.rows; i++){
    using namespace std;
    priority_queue<Neighbor, vector<Neighbor>, Compare> pq;
    searchVPTree(Corpus, Q.data + i * Q.cols, (int)Q.cols, (int)k, pq, threshold);
    std::vector<Neighbor> neighbors;
    while (!pq.empty()) {
    neighbors.push_back(pq.top());
    pq.pop();
    }
    allNeighbors[i] = std::move(neighbors);
}
```

3.2 Threshold Calculation

A crucial aspect of the code is the calculation of the threshold, which represents the trade-off between accuracy and performance. If the threshold is large, more subtrees will be examined, leading to higher accuracy at the cost of slower execution. Conversely, if the threshold is small, fewer subtrees will be examined, improving performance but potentially missing some nearest neighbors.

To calculate the threshold, we first compute the mean distance between two corpus points. This is achieved by leveraging the computations performed during the `VPtree` function. Specifically, while building the VP-Tree, a significant number of distances between corpus points are calculated, which can be reused to efficiently estimate the mean distance.

```
//calculate mean distance between corpus points
double localTotalDistance = 0.0;
int localTotalNodes = 0;

#pragma omp parallel for reduction(+:localTotalDistance, localTotalNodes)
    for (int i = 0; i < (int)distances.rows; i++) {
        localTotalDistance += distances.data[i];
        localTotalNodes++;
    }

// Update the shared variables outside the parallel region
pthread_mutex_lock(&mtx);
*totalDistance += localTotalDistance;
*totalNodes += localTotalNodes;
pthread_mutex_unlock(&mtx);
```

After this, to determine the final threshold, we multiplied the mean distance by a multiplier. This multiplier is generated by the following function, which takes the number of corpus points as input. Depending on the desired trade-off between accuracy and performance, the function can be adjusted accordingly.

```
double thresholdFunc(double x) {
    double m = (0.1 - 0.5) / (1000000 - 100);
    double b = 0.5 - m * 100;
    return m * x + b;
}
```

4 Testing

To test our code, we used benchmark datasets from the following GitHub repository: ANN-Benchmarks. Specifically, we utilized the MNIST dataset, which contains 60,000 corpus points, 10,000 queries, and 784 dimensions, as well as the SIFT dataset, which consists of 1,000,000 corpus points, 10,000 queries, and 128 dimensions.The metrics we used for the testing are the following:

- **Recall** refers to the accuracy of the algorithm, specifically the proportion of nearest neighbors that were correctly identified. It is a measure of how many of the true nearest neighbors were successfully calculated by the algorithm.
- **Queries/sec (Queries per second)** refers to the performance of the algorithm. It is calculated as the total number of queries processed divided by the total execution time required by the algorithm. A higher value indicates better performance, as it signifies that more queries are processed in less time.

Version	Metric	MNIST	SIFT
V0-OpenMP	Recall	100%	—
	Queries/second	273	—
V0-PTHREADS	Recall	100%	—
	Queries/second	448	—
V1-OpenMP	Recall	4.02%	3.07%
	Queries/second	507	383
V1-PTHREADS	Recall	4.02%	3.07%
	Queries/second	530	388

Table 1: V0 refers to Exact Approach and V1 to Approximate Approach

As seen in the table above, the exact approach of the problem yields a 100% recall, demonstrating perfect accuracy in identifying the nearest neighbors. However, it is unable to execute for the SIFT dataset due to the large size of the distance matrix, which exceeds memory and computational constraints. On the other hand, the approximate approach, while offering significantly lower accuracy (recall), is able to execute on both datasets and achieves better performance. The approximate method also provides flexibility, as its results can be modified simply by adjusting the threshold, allowing for a trade-off between accuracy and performance.

For testing, we used a laptop with 8 GB of RAM and four cores. However, the program was executed within a virtual machine running Arch Linux, which had access to only three cores and 4 GB of RAM. By utilizing a more powerful machine, with greater resources such as additional cores and more memory, better performance can be achieved.