University of Camerino

School of Science and Technology

Master Degree in Computer Science (Class LM-18)

Parallel and Distributed Programming

K-Nearest Neighbors, a Parallel Approach using MPI

Students
Marco Quadrini **124080**
Matteo Scoccia **124600**

Supervisors
Prof. Andrea Polini

A.Y. 2023/2024

Contents

1. Introduction

The k-nearest neighbors (k-NN) algorithm, originally developed by Evelyn Fix and Joseph Hodges in 1951, is a non-parametric method widely applied in both classification and regression tasks. The algorithm operates by identifying the k nearest points to a given data point in a defined space, based on a distance metric, typically Euclidean distance. This project focuses on solving the problem of finding the k-nearest neighbors in a 3D Euclidean space, where each point is represented by three coordinates (x, y, z). The objective of this project is not only to implement the k-NN algorithm but also to optimize its execution by leveraging parallel processing capabilities. By using the Message Passing Interface (MPI) standard, which enables communication among processes in a distributed memory system, the solution can run efficiently on multi-core processors. This parallelized approach will allow the algorithm to handle larger datasets and reduce computation time significantly compared to a sequential implementation.
For each point in the dataset, the algorithm must compute the distance to every other point, sort these distances, and select the k closest points. The main challenge in parallelizing this process lies in distributing the data efficiently across multiple processors and managing the communication between them to ensure the correct nearest neighbors are identified for each point.
To achieve parallelization, the dataset will be split among several processes, each responsible for calculating distances for a subset of points. After computing the distances locally, the processes will communicate the results to gather the necessary global information.The performance improvement in this parallelized version will depend on factors such as the size of the dataset, the number of available cores, and the efficiency of the communication strategy. Ideally, the solution should achieve near-linear speedup as the number of processors increases, though communication overhead and load balancing are important considerations that may impact overall performance.

2. Background

In this section, we will look at the basic ideas behind the algorithms in the next chapters. In particular, we will talk about the data structures used and the techniques applied to run the project. By learning these ideas, we can understand how the algorithms work and solve different problems. Understanding these basics will help us deal with the challenges that come up later.

Data Structures

The project focuses on solving the 3D k-nearest neighbors (k-NN) algorithm, which requires calculating the distance between points in a 3D space. To represent these points, we defined a data structure called `Point`. This structure stores the coordinates of a point in space and contains three fields: `x`, `y`, and `z`, all of type `float`, as shown below:

```c
typedef struct {
    float x, y, z;
} Point;
```

Code 2.1: Struct definition for Point

In addition to representing the points, it was necessary to introduce another data structure to store the Euclidean distance between a point and its nearest neighbors. This structure, called `Distance`, holds two fields: `neighbor_distance`, a `float` representing the distance to a neighboring point, and `neighbor_index`, an `int` representing the index of that neighbor in the dataset. This structure helps keep track of both the distance and the corresponding neighbor:

```c
typedef struct {
    float neighbor_distance;
    int neighbor_index;
} Distance;
```

Code 2.2: Struct definition for Distance

These two structures, Point and Distance, are central to the implementation of the KNN algorithm, allowing us to represent points in space and calculate and store the distances between them. Where MPI was used, the corresponding MPI data types were

built.
To generate the points dataset in a 3D space, we used the function `generate_points`, shown below. This function takes as input an array of `Point` structures and an integer `n`, representing the number of points to generate. The values of the coordinates `x`, `y`, and `z` are randomly assigned, with values between 0 and 100, using the standard C library function `rand()`:

```c
void generate_points(Point *points, int n) {
    printf("\n---Generating %d points---\n", n);
    for (int i = 0; i < n; i++) {
        points[i].x = ((float) rand() / RAND_MAX) * 100;
        points[i].y = ((float) rand() / RAND_MAX) * 100;
        points[i].z = ((float) rand() / RAND_MAX) * 100;
    }
}
```

Code 2.3: Function to generate random points

The number of points to generate is provided as input, along with the number of nearest neighbors (`k`) to be found for each point. In the case of parallel KNN, the number of processors to use is also considered, as the algorithm utilizes the `MPI` to perform the computation across multiple processors. During execution, the algorithm divides the points among the processors to calculate distances and identify the nearest neighbors in parallel, optimizing computational efficiency.

Run Program

The program has three versions, in particular we developed a sequential version and two using the parallelization: one by exploiting the *Star Topology* and one using *Ring Topology*, but these will be explained in the following chapters 4,5 and 6. The sequential code can be easily compiled using the standard GCC compiler and run as a standalone application. On the other hand, the parallel version requires a specific compiler provided by the MPI standard, known as `mpicc` (see Code 2.4).

```
mpicc -o knn_parallel parallel.c -lm
```

Code 2.4: Compiling the MPI version of the project

Once the parallel code has been compiled, the resulting executable must be launched using the `mpirun` command, specifying the number of processors you wish to use for execution. Both the sequential and parallel versions require certain parameters to operate correctly:

1. **N**: The number of points to be generated.

2. **K**: The number of nearest neighbors to consider for each point.

The command to run the parallel version is shown in Code 2.5. In addition to `N` and `K`, we also need to specify the number of processors with which we want to execute the code.

```
mpirun -np <num_procs> ./knn_parallel <N> <K>
```

Code 2.5: Running the MPI executable

To evaluate performance more effectively, we have created an evaluation mode. To run this mode, you need to add the `-ev` flag to the command, as shown below. This mode will repeat the process five times and return the average of the results:

```
mpirun -np <num_procs> ./knn_parallel <N> <K> -ev
```

Code 2.6: Running the evaluation mode

To launch all possible combinations of parameters, reaching up to 65536 points and 64 processors, we have created a shell script. This script compiles the version of the KNN program using the `mpicc` compiler and executes it with varying values for the number of processes and nearest neighbors. For simplicity, we used a value of `N` which was always divisible by the number of processors, so that each had the same amount of points to work with.
The script iterates over a range of processor counts and values for `K`, enabling a comprehensive evaluation of the program's performance under different configurations. Below is the content of the script:

```
#!/bin/bash

mpicc -std=c99 -o knn_parallel parallel.c -lm

potenzaMassimaProcessi=6
potenzaMassimaK=16
massimoValoreK=20

for ((i=1; i<=potenzaMassimaProcessi; i++))
do
    p=$((2**i))
    for ((j=1; j<=potenzaMassimaK; j++))
    do
        n=$((2**j))
        for ((h=5; h<=massimoValoreK; h+=5))
        do
            k=$((h))
            mpirun ./knn_parallel $p $n $k -ev
        done
    done
done
```

Code 2.7: Shell script for executing k-NN with various configurations

Results and Performance Evaluation

The parallel implementations are compared to the sequential version with two metrics: speed-up and efficiency. The results will be described for each implementation by charts and data from Chapter [5] and [6].

Speed-Up Calculation

The speed-up $S(n, p)$ is defined as the ratio of the execution time of the sequential algorithm $T_{\text{serial}}(n)$ to that of the parallel algorithm $T_{\text{parallel}}(n, p)$:

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

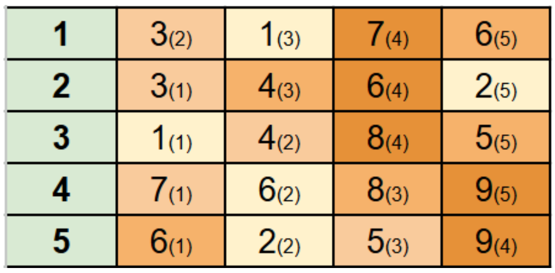
Efficiency Calculation

Efficiency $E(n, p)$ measures how effectively the processors are utilized and is calculated using the speed-up divided by the number of processors p:

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p) \times p}$$

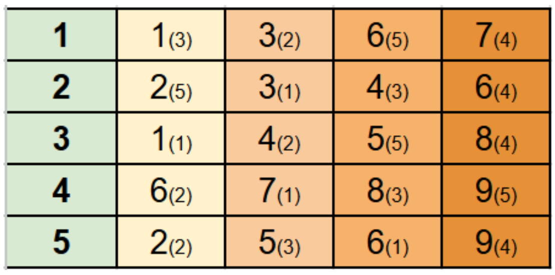
3. Sequential algorithm

The approach we used for the sequential implementation of our algorithm is pretty straightforward. The algorithm begins by generating the dataset using the function discussed in chapter [2]. After the dataset is generated, the program iterates through the array of the generated points, and for each one of these the distance with each of the other points in the dataset is computed.
The distance is stored in an array of type `NeighborDistance`, a structure that holds two fields: the distance from the current point and the index of the corresponding neighboring point. The dimension of this array is n-1 as for each point the distance with all the other n-1 points must be computed. After the distance array is filled, it is sorted using the quick sort algorithm, which has an average time complexity of $O(n \log n)$. Although divided in different arrays, what is being computed at this point is known as the *distance matrix*, but for each cell of the matrix we have the index of the point in addition to the distance. The diagonal is missing as the distance between the same points is not computed.



1	$3_{(2)}$	$1_{(3)}$	$7_{(4)}$	$6_{(5)}$
2	$3_{(1)}$	$4_{(3)}$	$6_{(4)}$	$2_{(5)}$
3	$1_{(1)}$	$4_{(2)}$	$8_{(4)}$	$5_{(5)}$
4	$7_{(1)}$	$6_{(2)}$	$8_{(3)}$	$9_{(5)}$
5	$6_{(1)}$	$2_{(2)}$	$5_{(3)}$	$9_{(4)}$

Figure 3.1: Example of distance arrays before sorting



1	$1_{(3)}$	$3_{(2)}$	$6_{(5)}$	$7_{(4)}$
2	$2_{(5)}$	$3_{(1)}$	$4_{(3)}$	$6_{(4)}$
3	$1_{(1)}$	$4_{(2)}$	$5_{(5)}$	$8_{(4)}$
4	$6_{(2)}$	$7_{(1)}$	$8_{(3)}$	$9_{(5)}$
5	$2_{(2)}$	$5_{(3)}$	$6_{(1)}$	$9_{(4)}$

Figure 3.2: Example of distance arrays after sorting

Once sorted, the first k elements of this array correspond to the k nearest points to the current point. This approach, although simple, has a high computational cost. For each of the n points in the dataset, the algorithm must compute $n-1$ distances, resulting in a time complexity of $O(n^2)$ for distance calculations. Furthermore, sorting the array for each point adds an additional cost, making the total time complexity $O(n^2 \log n)$. So, the time required for the computation as the number of point grows can be very large, and the same is for the required memory, as the space complexity is the same. One advantage of this approach is that it's independent the value of k. Regardless of the chosen value of k, the algorithm always performs the same set of computations. This means that increasing the value of k does not lead to additional computational overhead, as the sorted distance array is already available, and extracting the first k elements is not a costly operation.

```c
void find_k_nearest_neighbors(FILE *f, Point *points, int n, int k) {
    for (int i = 0; i < n; i++) {
        NeighborDistance distances[n - 1];
        int count = 0;

        // Compute all the distances from points[i] to all other points
        for (int j = 0; j < n; j++) {
            if (i != j) { // Skip the distance with itself
                distances[count].neighbor_distance =
                        euclidean_distance(points[i], points[j]);
                distances[count].neighbor_index = j;
                count++;
            }
        }

        // Sort the distances
        qsort(distances, n - 1, sizeof(NeighborDistance),
    compare_distances);

        // Distances are computed and sorted
        for (int m = 0; m < k; m++) {
            // Eventual additional computations with the k nearest
    neighbors
        }
    }
}
```

Code 3.1: Main computation of the sequential algorithm

Results

The results presented in Table 3.1 showcase the execution times of the KNN algorithm under varying conditions. The first column indicates the number of points N generated, while the second column specifies the number of nearest neighbors K considered in the computation. As the number of points increases from 1024 to 65536, a noticeable trend emerges: the execution time significantly increases, particularly for larger values of N. This behavior is expected due to the algorithm's inherent complexity, which grows with the size of the dataset.

On a positive note, the results also reveal that there is no substantial variations in execution time based on different K values for each N. As K increases, the time taken to compute distances and identify neighbors is the same.

Number of Points (N)	**K (Neighbors)**	**Execution Time (s)**
1024	5	0.198
	10	0.202
	15	0.206
	20	0.210
2048	5	0.860
	10	0.860
	15	0.872
	20	0.878
4096	5	3.746
	10	3.722
	15	3.706
	20	3.710
8192	5	15.644
	10	15.680
	15	15.700
	20	15.742
16384	5	66.708
	10	66.614
	15	66.684
	20	66.702
32768	5	281.996
	10	282.434
	15	283.104
	20	282.434
65536	5	1194.392
	10	1209.568
	15	1193.242
	20	1194.256

Table 3.1: Execution Time Results for KNN Algorithm

4. Parallel algorithm

The first parallel implementation we developed is based on a similar approach as the sequential version described in chapter 3. For this implementation, we decided to create two MPI data types to manage the communication between processes, and these data types correspond to the ones already mentioned before: point and distance.
Initially, the root process (rank 0) generates the entire dataset of points. Once the dataset is created, the root process broadcasts the dataset to all the other processes using the `MPI_Bcast` function. This function call — `MPI_Bcast(points, n, mpi_point_type, 0, MPI_COMM_WORLD)` — distributes the dataset to all processes so that each one has a complete copy. This enables each process to independently compute distances for a subset of points. The size of each subset is determined by $\frac{n}{p}$, where p is the total number of processes.
The process of finding the k-nearest neighbors for each subset closely mirrors the sequential implementation. Each process calculates the distances between its assigned points and all other points in the dataset. However, because each process handles only a fraction of the total points, the local distance matrix it computes has dimensions $points\_per\_process \times k$. This local distance matrix is stored in a `local_results` array, which will later be sent back to the root process.
Once all processes have completed their local computations, the root process gathers the results using `MPI_Gather`. The function call—`MPI_Gather(local_results, points_per_process × k, mpi_distance_type, global_results, points_per_process × k, mpi_distance_type, 0, MPI_COMM_WORLD)`—collects the local distance matrices from each process and combines them into the global result matrix. This matrix now contains the k-nearest neighbors for all points in the dataset.
The communication pattern used in this implementation corresponds to a *star topology*, where a main processor (the root one) sends data to each of the other processors which perform some sort of computation, and then gathers data back from them. Being based on the sequential implementation, this parallel version still suffers from the problems already mentioned before, such as inefficiency with large datasets and memory waste, but it can be faster using more processors. As the sequential version, this isn't affected by the dimension of k chosen by the user.

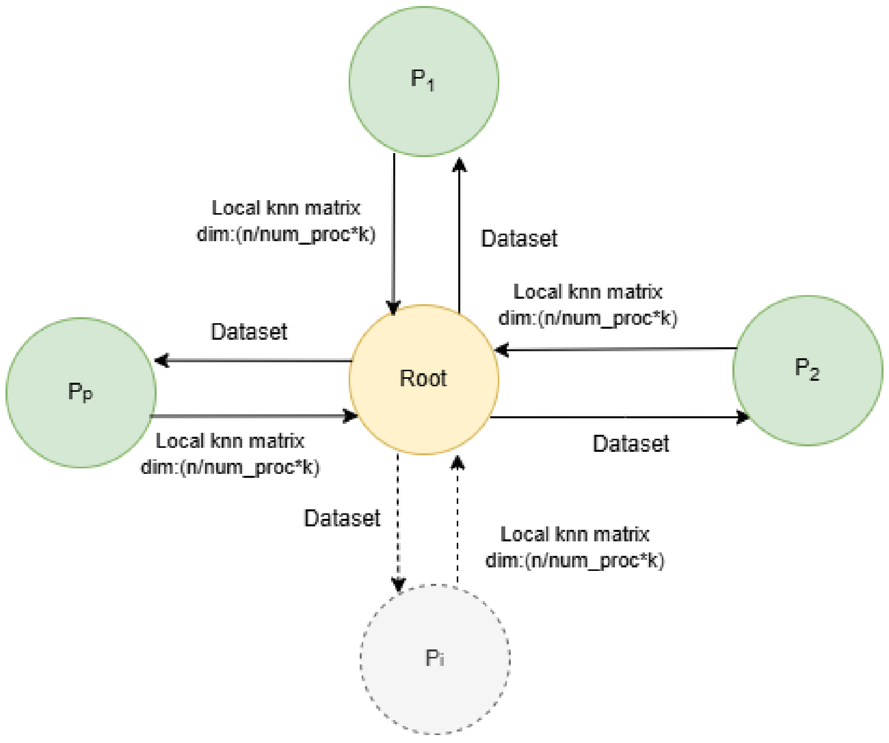
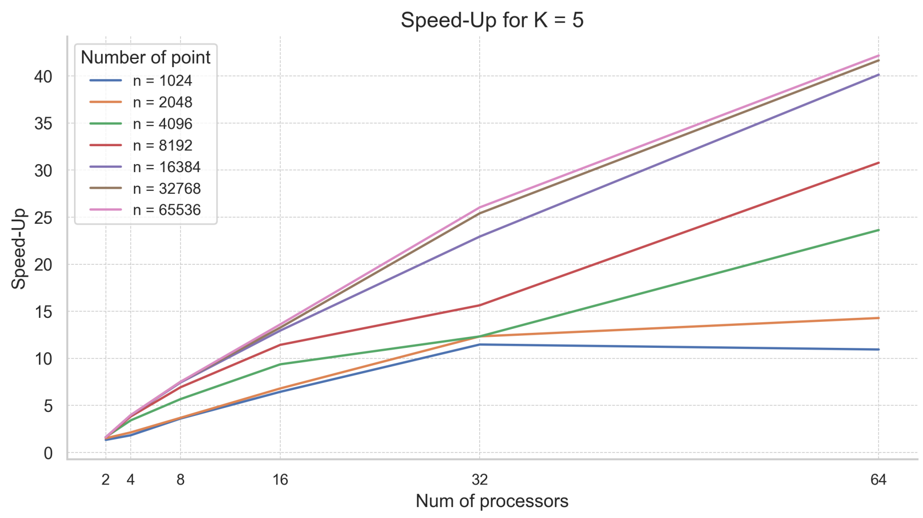
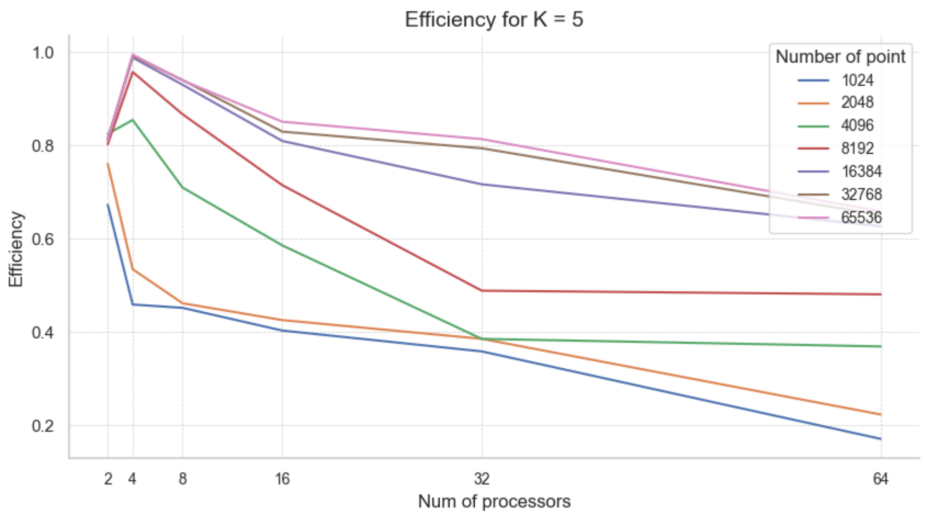

P1
Local knn matrix dim:(n/num_proc*k)
Dataset
Local knn matrix dim:(n/num_proc*k)
Dataset
Root
P2
Pp
Local knn matrix dim:(n/num_proc*k)
Dataset
Dataset
Local knn matrix dim:(n/num_proc*k)
Pi

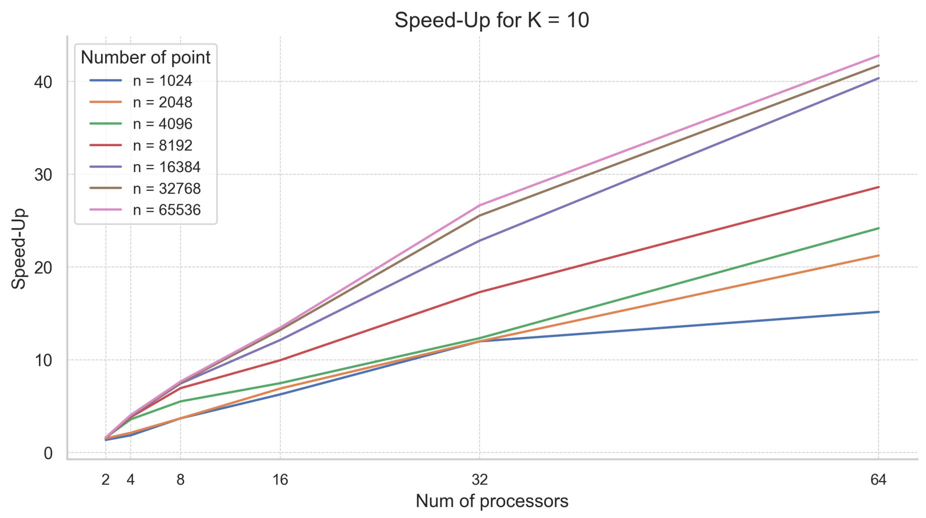
Figure 4.1: Process communication in this implementation

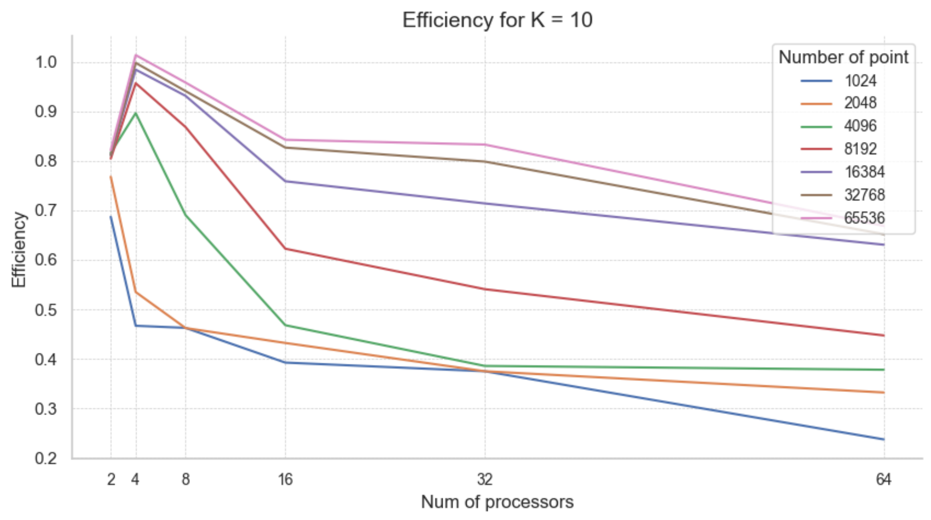
Results

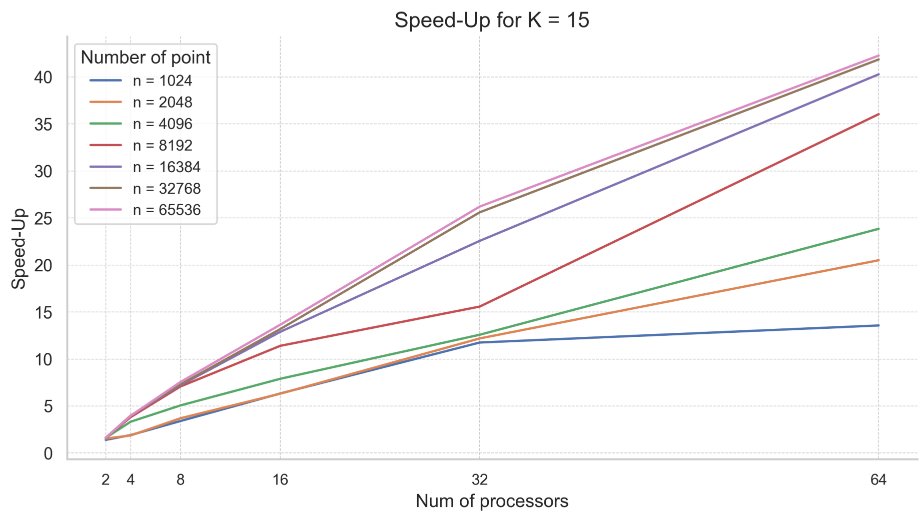
The results plot shown below are based on Speedup and Efficiency when comparing this parallel algorithm to its sequential counterpart.

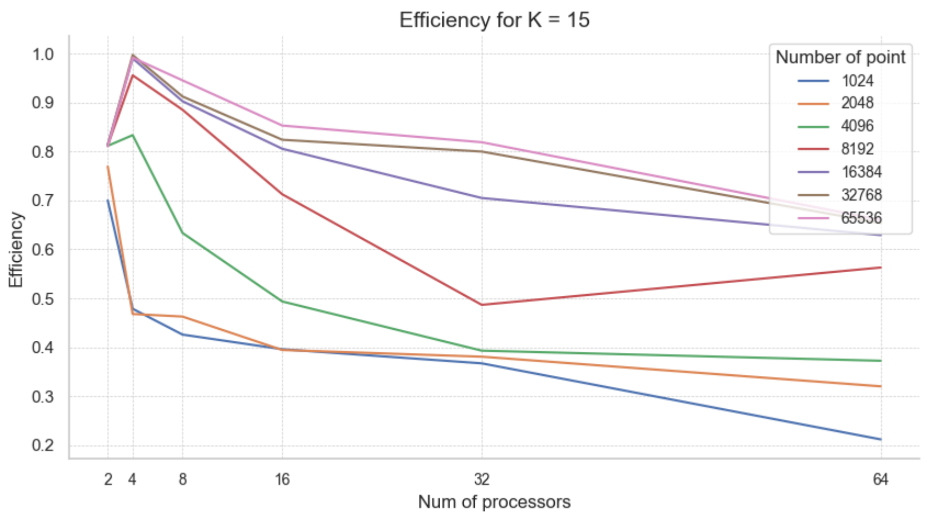

Speed-Up for K = 5
Number of point
n = 1024
n = 2048
n = 4096
n = 8192
n = 16384
n = 32768
n = 65536
Speed-Up
Num of processors
0
5
10
15
20
25
30
35
40
2
4
8
16
32
64

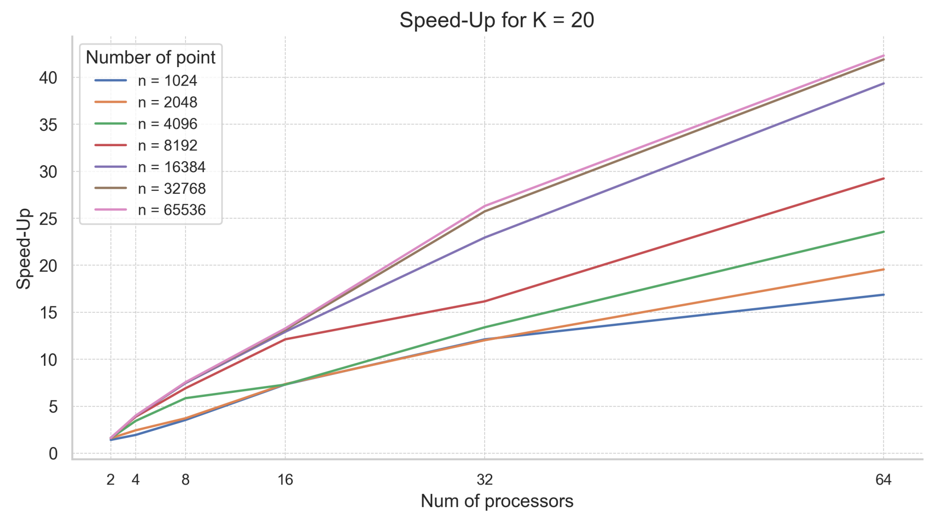

Efficiency for K = 5
Number of point
1024
2048
4096
8192
16384
32768
65536
Efficiency
Num of processors
0.2
0.4
0.6
0.8
1.0
2
4
8
16
32
64

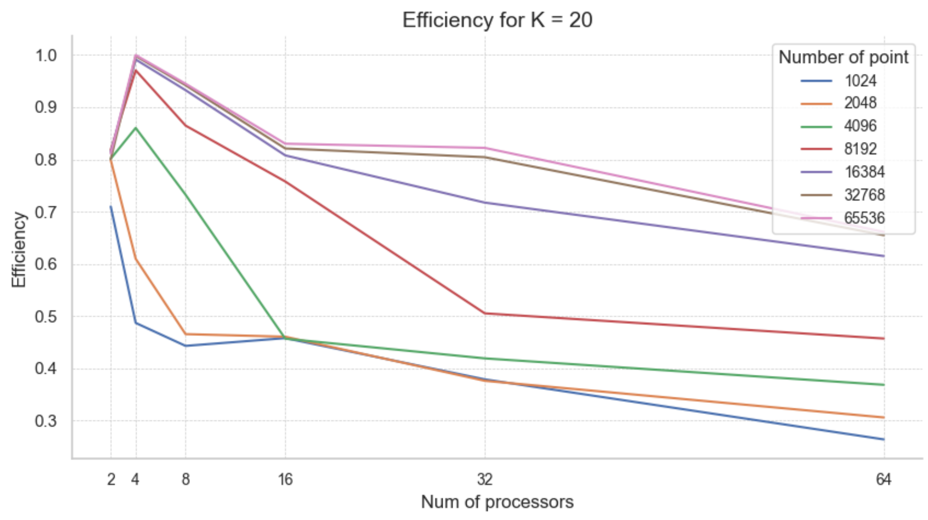

Speed-Up for K = 10
Number of point
n = 1024
n = 2048
n = 4096
n = 8192
n = 16384
n = 32768
n = 65536
Speed-Up
0
10
20
30
40
2
4
8
16
32
64
Num of processors


Efficiency for K = 10
Number of point
1024
2048
4096
8192
16384
32768
65536
Efficiency
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
2
4
8
16
32
64
Num of processors


Speed-Up for K = 15
Number of point
n = 1024
n = 2048
n = 4096
n = 8192
n = 16384
n = 32768
n = 65536
Speed-Up
Num of processors


Efficiency for K = 15
Number of point
1024
2048
4096
8192
16384
32768
65536
Efficiency
Num of processors


Speed-Up for K = 20
Number of point
n = 1024
n = 2048
n = 4096
n = 8192
n = 16384
n = 32768
n = 65536
Speed-Up
0
5
10
15
20
25
30
35
40
2
4
8
16
32
64
Num of processors


Efficiency for K = 20
Number of point
1024
2048
4096
8192
16384
32768
65536
Efficiency
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
2
4
8
16
32
64
Num of processors

5. Sequential improved

After the first sequential algorithm, whose implementation was straightforward but inefficient, we developed another sequential version using a different logic in order to improve performances.
In this improved version, we optimize the process by eliminating the need for sorting altogether. Instead of calculating all distances and then sorting them, we maintain a fixed-size array of the nearest neighbors as we compute the distances. This array is updated dynamically: if a newly computed distance is smaller than the current largest distance in the *k-nearest neighbors* list, we replace it with the new distance and re-position it within the list to maintain sorted order. This ensures that at any given point, we are only storing the k smallest distances and do not have to sort a large array of *n-1* distances.
The algorithm begins by generating the dataset using the function discussed in Chapter 3. After the dataset is generated, the program iterates through the array of points. For each point, the distance to all other points in the dataset is computed, but instead of storing all the distances, we only retain the k smallest ones. This is done using the function `insert_distance_if_needed`, which checks if a new distance should replace the largest distance in the list of current *k-nearest neighbors*.

```c
// Insert a neighbor into the K nearest neighbors list if it is closer
void insert_distance_if_needed(Distance *nearest_neighbors, int K,
    Distance new_neighbor) {
    if (new_neighbor.neighbor_distance < nearest_neighbors[K-1].
    neighbor_distance) {
        nearest_neighbors[K-1] = new_neighbor;

        // Move the new element to its correct position to maintain sorted
     order
        for (int i = K - 2; i >= 0; i--) {
            if (nearest_neighbors[i].neighbor_distance > nearest_neighbors
    [i + 1].neighbor_distance) {
                Distance temp = nearest_neighbors[i];
                nearest_neighbors[i] = nearest_neighbors[i + 1];
                nearest_neighbors[i + 1] = temp;
            } else {
                break;
            }
        }
    }
}

// Calculate distances and retain K nearest neighbors
void calculate_and_retain_k_nearest(Point *points, Distance *results, int
    N, int K) {
    for (int i = 0; i < N; i++) {
        Distance *nearest_neighbors = &results[i * K];

        for (int k = 0; k < K; k++) {
            nearest_neighbors[k].neighbor_distance = INFINITY;
            nearest_neighbors[k].neighbor_index = -1;
        }

        // Calculate the distance of the current point to all other points
        for (int j = 0; j < N; j++) {
            if (i != j) {
                Distance new_neighbor;
                new_neighbor.neighbor_distance = euclidean_distance(&
    points[i], &points[j]);
                new_neighbor.neighbor_index = j;

                // Insert this new neighbor into the K nearest neighbors
    list if it is closer
                insert_distance_if_needed(nearest_neighbors, K,
    new_neighbor);
            }
        }
    }
}
```

Code 5.1: Main computation of the improved sequential algorithm

As shown in the code, for each point, the distances to all other points are calculated using the `euclidean_distance` function. If the newly computed distance is smaller than the largest distance in the current k-nearest neighbors list, the new distance is inserted into the list, and the list is adjusted to maintain order. This makes the algorithm more efficient.

Results

The results presented in Table 3.1 showcase the execution times of this improved KNN algorithm. We could notice a significant improvement in terms of execution times using this new approach, as this version is very efficient when k is small compared to n.

Number of Points (N)	**K (Neighbors)**	**Execution Time (s)**
1024	5	0.090
	10	0.080
	15	0.050
	20	0.050
2048	5	0.160
	10	0.170
	15	0.180
	20	0.190
4096	5	0.660
	10	0.670
	15	0.690
	20	0.720
8192	5	2.670
	10	2.740
	15	2.780
	20	2.820
16384	5	10.680
	10	10.740
	15	10.760
	20	10.920
32768	5	42.140
	10	42.230
	15	42.400
	20	42.810
65536	5	167.010
	10	167.150
	15	167.430
	20	167.920

Table 5.1: Execution Time Results for Improved Sequential KNN Algorithm

6. Ring implementation

The second parallel implementation we developed uses a ring topology for process communication. We also changed the approach we used in the first parallel version, shifting to the improved version we already tested with the improved sequential algorithm. Additionally, the distribution and collection of data among processes are handled differently than in the first parallel implementation. We again use two custom MPI data types for `point` and `distance`.
The process begins with the root process (rank 0) generating the dataset. Once the dataset is created, it is divided into smaller subsets, each allocated to a different process. This is achieved through `MPI_Scatter` using the custom `mpi_point_type`, where each process receives a chunk of the dataset. The size of each subset is calculated as $\frac{N}{p}$, where N is the total number of points and p is the number of processes. This allows each process to work independently on its assigned subset.
To facilitate the ring communication pattern, we define two variables, `next` and `previous`, which indicate the rank of the next and previous processes in the ring, respectively. Each process then sends its local subset of points to the next process and receives a subset from the previous process in the ring. This exchange is done using the `MPI_Sendrecv` function call with the custom `mpi_point_type`:

```
MPI_Sendrecv(my_points, points_per_process, mpi_point_type,
             next, 0, received_points, points_per_process,
             mpi_point_type, previous, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
```

Each process computes the distances between its local points and the points received from the previous process. The function `calculate_and_retain_k_nearest` is used to compute these distances and retain only the k nearest neighbors for each point. The local results are stored in a `local_results` array, which has dimensions *points_per_process* $\times$ k. So in this implementation, each process computes the distances between its local points and the points received from the previous process in the ring. However, unlike the sequential and the star pattern parallel implementations. Like the second sequential version, this implementation directly inserts each new distance into an array of k-nearest neighbors if it qualifies. This reduces the need for storing and sorting large amounts of unnecessary data, thereby optimizing both memory usage and computational efficiency.

This process is repeated p times, where p is the number of processes, to ensure that each process has computed the distances between its local points and all other points in the dataset. After all iterations are completed, the local results are gathered back to the root process using `MPI_Gather` with the custom `mpi_distance_type`:

```
MPI_Gather(local_results, points_per_process * k, mpi_distance_type,
            global_results, points_per_process * k,
            mpi_distance_type, COORDINATOR, MPI_COMM_WORLD);
```

This step collects all the local results into the `global_results` array, which contains the k-nearest neighbors for all points in the dataset.

The ring topology used in this implementation ensures that each process communicates only with its direct neighbors. This version of the algorithm has proven to be significantly faster than the previous one, but the main reason is in the improved computation procedure rather than in the ring communication pattern.

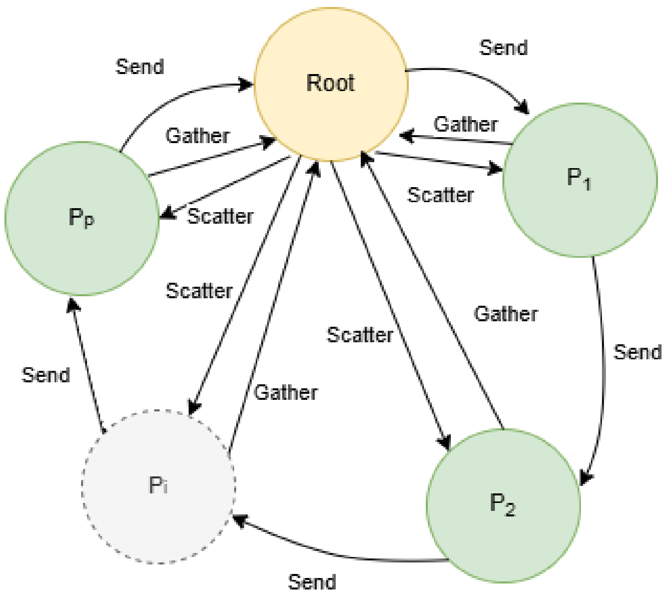
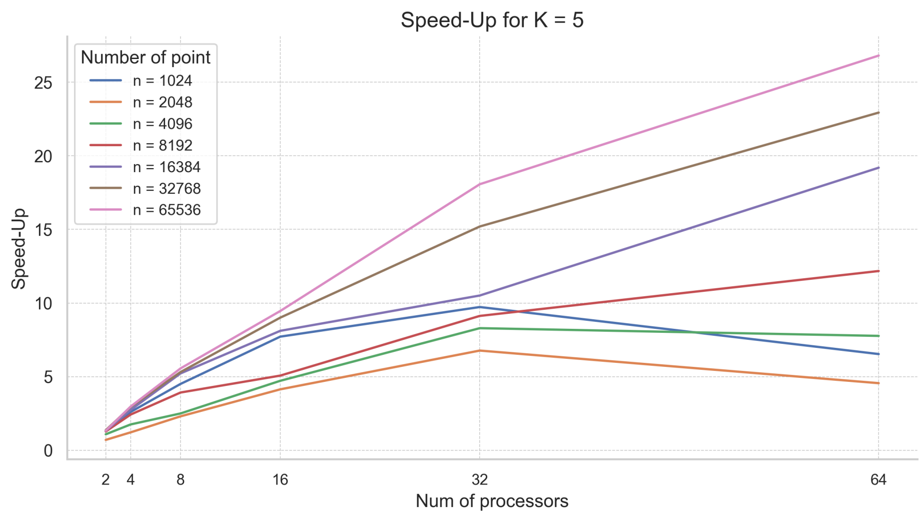
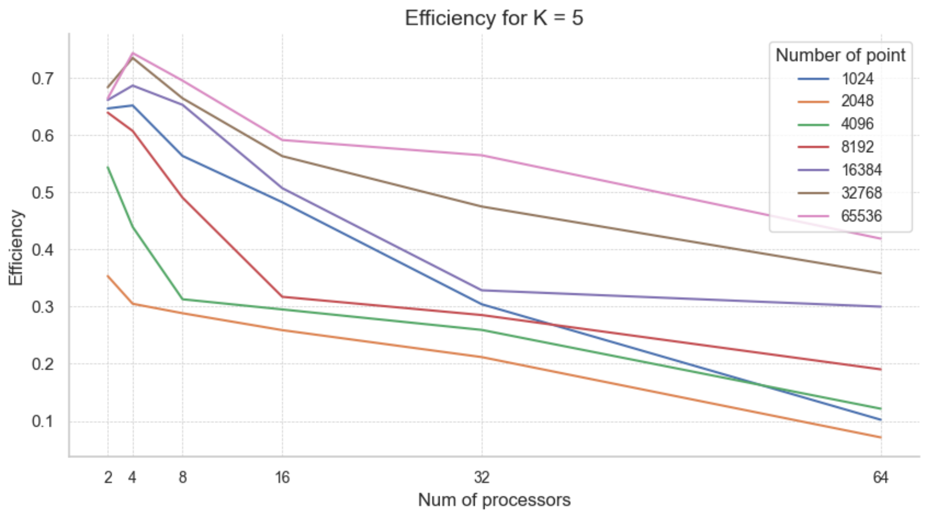
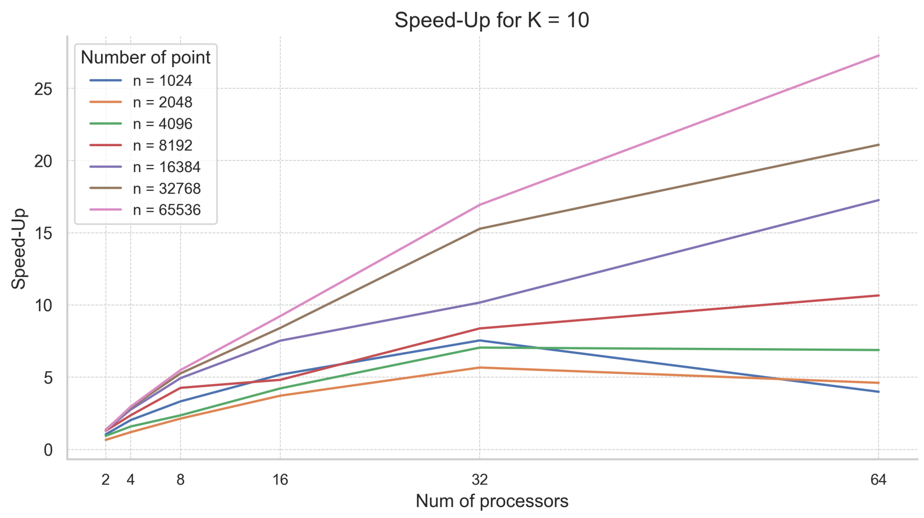


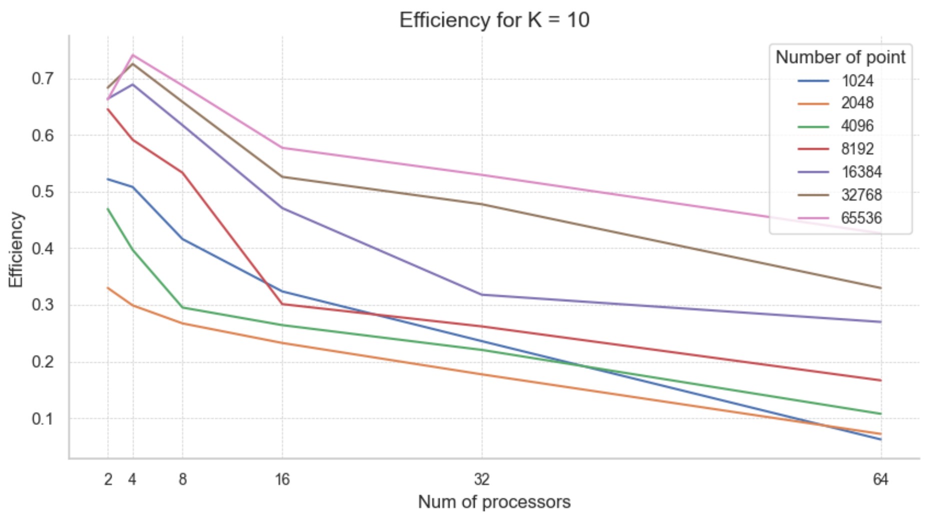
Figure 6.1: Process communication in this implementation

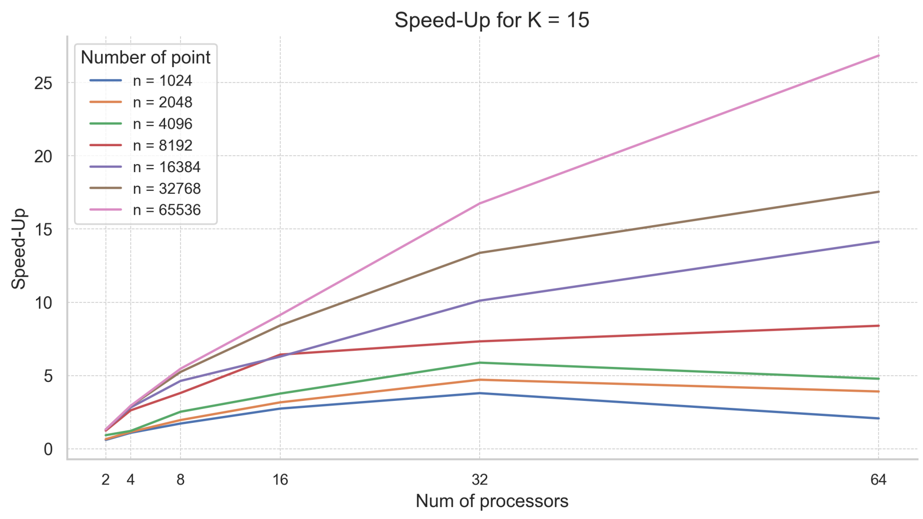
Results


Speed-Up for K = 5
Number of point
n = 1024
n = 2048
n = 4096
n = 8192
n = 16384
n = 32768
n = 65536
Speed-Up
Num of processors
2
4
8
16
32
64
0
5
10
15
20
25

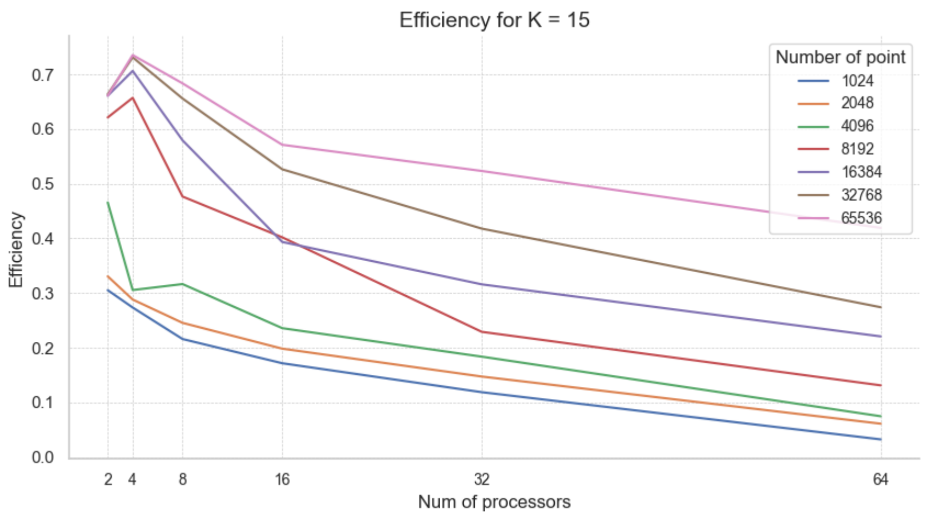

Efficiency for K = 5
Number of point
1024
2048
4096
8192
16384
32768
65536
Efficiency
Num of processors
2
4
8
16
32
64
0.1
0.2
0.3
0.4
0.5
0.6
0.7


Speed-Up for K = 10
Number of point
n = 1024
n = 2048
n = 4096
n = 8192
n = 16384
n = 32768
n = 65536
Speed-Up
0
5
10
15
20
25
2
4
8
16
32
64
Num of processors

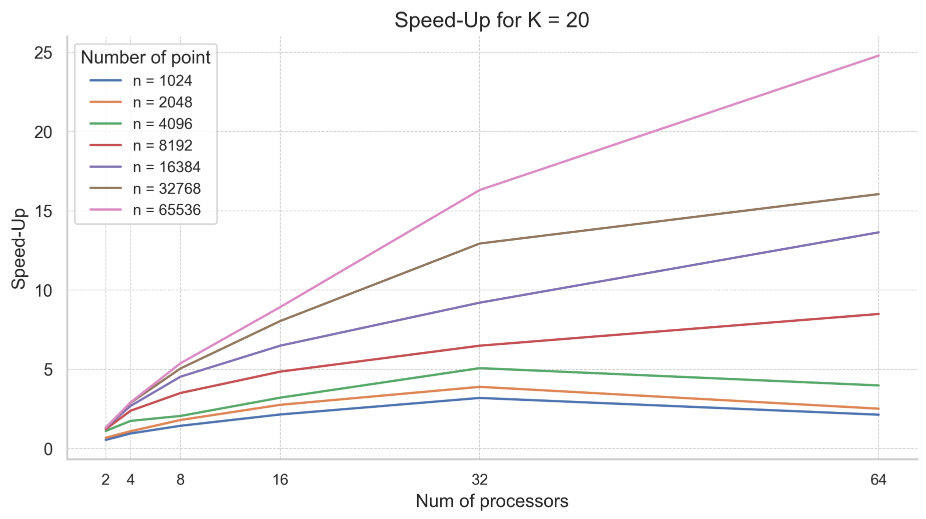

Efficiency for K = 10
Number of point
1024
2048
4096
8192
16384
32768
65536
Efficiency
0.1
0.2
0.3
0.4
0.5
0.6
0.7
2
4
8
16
32
64
Num of processors

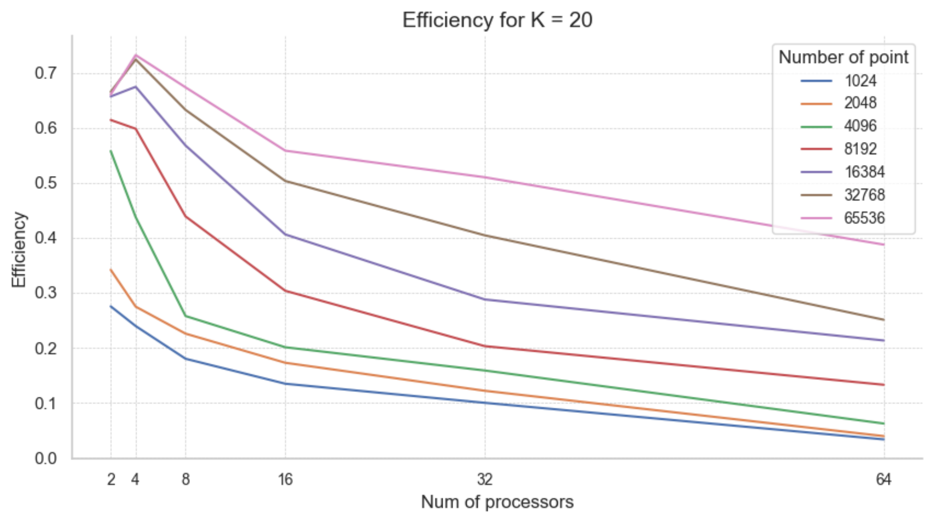


Speed-Up for K = 15
Number of point
n = 1024
n = 2048
n = 4096
n = 8192
n = 16384
n = 32768
n = 65536
Speed-Up
0
5
10
15
20
25
2
4
8
16
32
64
Num of processors



Efficiency for K = 15
Number of point
1024
2048
4096
8192
16384
32768
65536
Efficiency
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
2
4
8
16
32
64
Num of processors


Speed-Up for K = 20
Number of point
n = 1024
n = 2048
n = 4096
n = 8192
n = 16384
n = 32768
n = 65536
Speed-Up
0
5
10
15
20
25
2
4
8
16
32
64
Num of processors


Efficiency for K = 20
Number of point
1024
2048
4096
8192
16384
32768
65536
Efficiency
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
2
4
8
16
32
64
Num of processors

7. Conclusions

In this report, we implemented and compared four versions of the K-Nearest Neighbors (KNN) algorithm: two sequential versions, a parallel version using a star communication pattern, and a parallel version using a ring topology, then we assessed their performance in handling large datasets.
The **first sequential approach** calculates the distances between each point and all others, sorting the results to find the K-nearest neighbors. While simple, this method becomes computationally expensive as N increases due to its $O(n^2 \log n)$ complexity.
An **improved sequential version** optimizes this by dynamically maintaining a sorted list of K-nearest neighbors during distance calculation. This reduces the time complexity and improves performance, making it more efficient for larger datasets.
The **first parallel implementation** uses a star communication pattern, where the root process broadcasts the dataset and gathers results. This reduces computation time but introduces communication overhead, particularly with larger datasets and many processes.
The **ring topology** uses `MPI_Sendrecv`, and each process communicates with its neighbors, exchanging points in a ring pattern.
The results gathered from our testing show that as the number of processors increases, the execution time reduces significantly, and even if the speed-up doesn't reflect a perfect linear relationship, the results are quite promising. In conclusion, parallelism greatly improves the scalability of KNN, particularly with optimized data handling and efficient communication patterns.
Full results of the executions are available on the GitHub repository of our project (https://github.com/matteoscoccia/knn-project).