K-Nearest Neighbors, a Parallel Approach

Parallel and Distributed Programming

Students:
Marco Quadrini
Matteo Scoccia

30th Set 2024

Supervisor:
Prof. Andrea Polini

What is the KNN Search Algorithm?

- First introduced by Evelyn Fix and Joseph Hodges in 1951
- Formalized by Thomas Cover and Peter Hart in 1967
- It is a method for finding the **"k"** nearest data points to a query point
- It is one of the fundamental search methods

How KNN Search Algorithm Works?

- Take in input a point and the number of neighbors (**“k”**)
- Compute the distance between the point and all other points
- Select the **“k”** closest points to the input point based on the chosen distance metric

The Project

- KNN requires computing the distance between one point and all other points
- For large datasets it becomes computationally expensive and memory-intensive
- **Goal**: implement parallel KNN algorithms using MPI
- Analyze **speed-up** and **efficiency** in various scenarios with large datasets.

General Approach

- Calculate Euclidean distance between two points
- Generation of points in a random way
- Implementing 4 different approaches: 2 sequential and 2 parallel
- Using MPI library for parallel approach

Evaluating Metrics

The parallel implementations are compared to their sequential counterpart focusing on two metrics:

- **Speed-Up:** $$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}$$

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

Sequential Implementation

- Generate the dataset in a random way
- For each point, the algorithm computes the distance to all other points in the dataset.
- Distances from each point are stored in an array of **NeighborDistance ->** neighbor matrix

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

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

Sequential Implementation

- Each distance array is sorted and the first k elements are retrieved
- Straightforward approach but inefficient (costly in terms of time and space)
- Doesn’t depend on the values of k chosen

Sequential Implementation Results

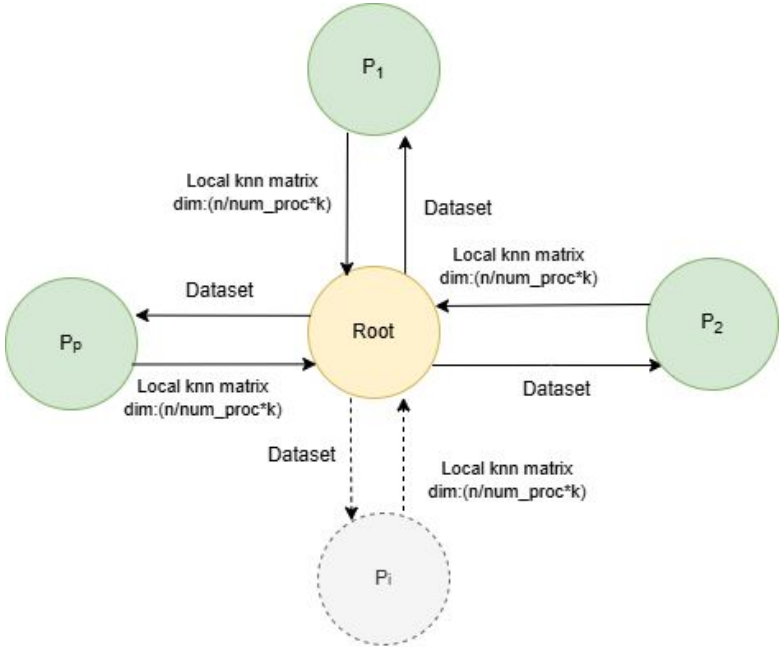
Number of Points (N)	K (Neighbors)	Execution Time (s)
1024	5	0.198
	10	0.202
	15	0.206
	20	0.210
2048	5	0.860
	10	0.860
	15	0.872
	20	0.878
4096	5	3.746
	10	3.722
	15	3.706
	20	3.710
8192	5	15.644
	10	15.680
	15	15.700
	20	15.742
16384	5	66.708
	10	66.614
	15	66.684
	20	66.702
32768	5	281.996
	10	282.434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

Parallel Implementation

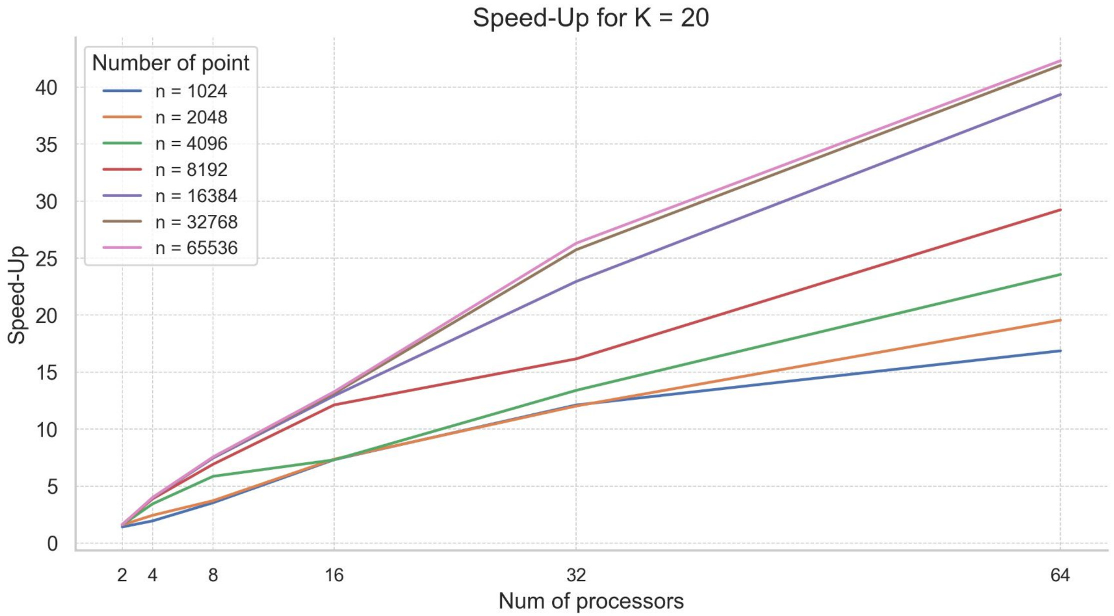
- The root process (rank 0) generates the entire dataset of points
- The root process broadcasts the dataset to all other processes (Mpi_Bcast)
- Custom Mpi data types for point and distance
- Each process compute distances and find the k-nearest neighbors for a subset of points
- The process stores its sub-distance matrix locally

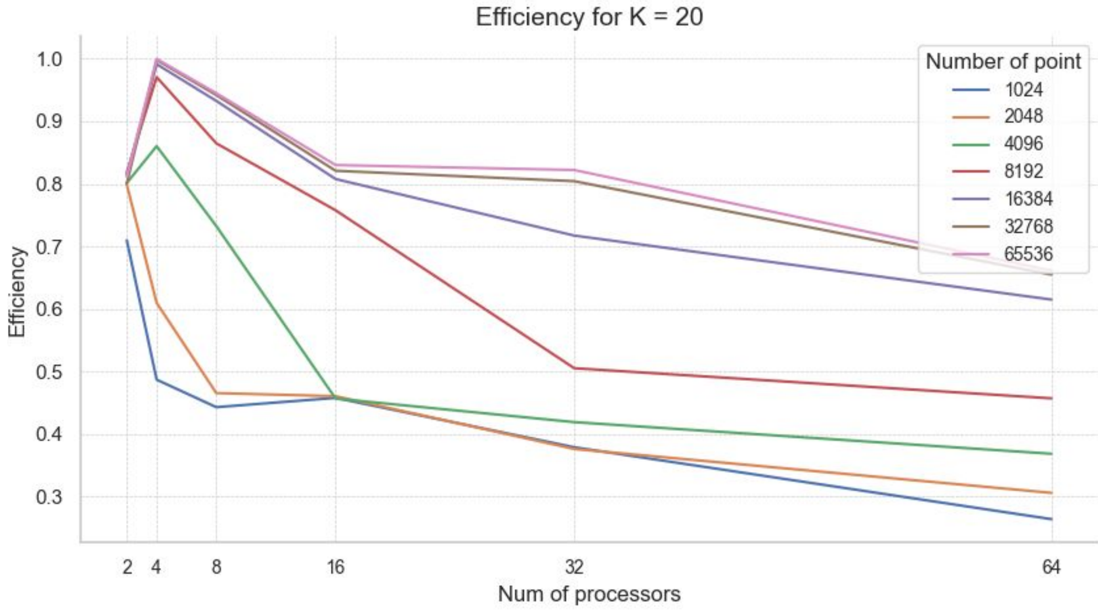
Parallel Implementation

- The root process gathers the results in a matrix (Mpi_Gather)
- This matrix contains the **k-nearest** neighbors for all points in the dataset.
- Resulting communication pattern: **Star Topology**



Parallel Implementation Results





Improved Sequential Implementation

- Generate the dataset in a random way
- Only K distances for each point are retained
- For each point, distances from each other point are computed
- If the computed distance fits in the currently retained distances (smaller) it’s inserted, otherwise discarded
- The array is naturally sorted

Improved Implementation Results

- Improved time and space computational cost
- Improved execution time:

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

Ring Implementation

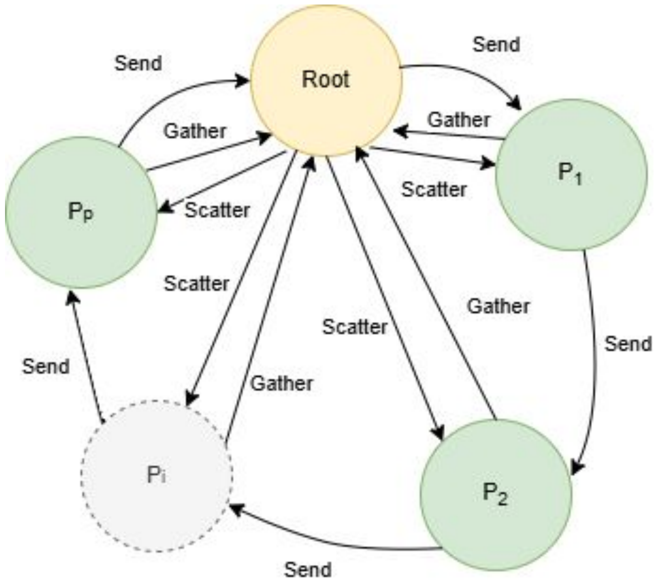
- The root process (rank 0) generates the entire dataset of points
- The dataset is divided into smaller subsets, each allocated to a different process (Mpi_Scatter)
- Each process perform its computation on its subset of points (improved approach)
- Each process then sends its local subset of points to the next process and receives a subset from the previous process in the ring (Mpi_Sendrecv)

Ring Implementation

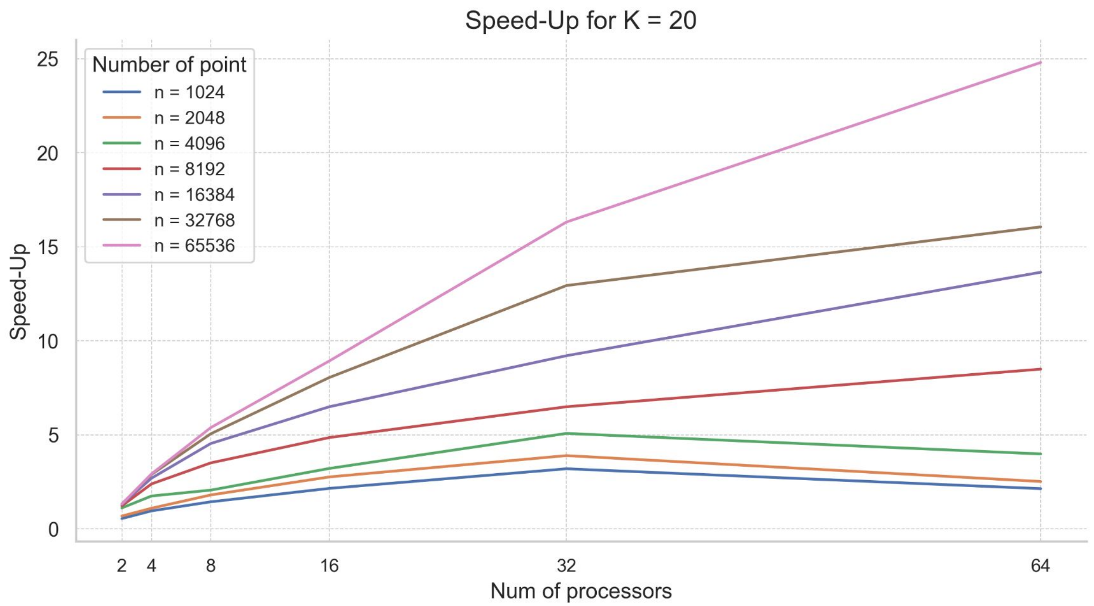
- Each process computes the distances between its local points and the points received from the previous process

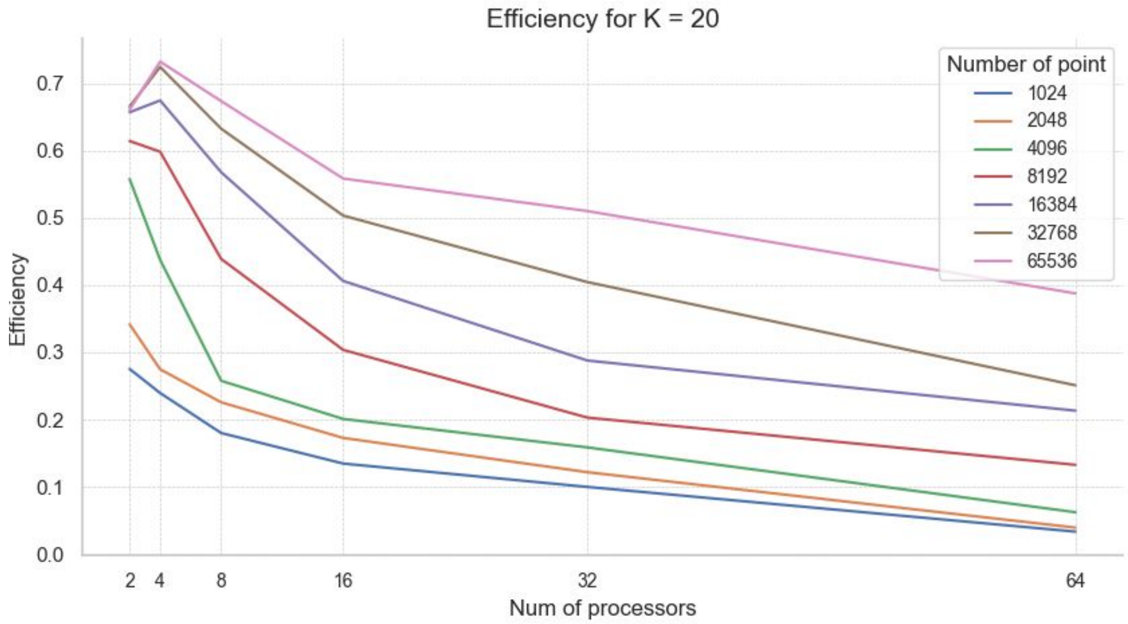
- The local results are gathered back to (Mpi_Gather)

- Based on **Ring Topology**



Ring Implementation Results





Conclusions

- 4 approaches: 2 sequentials and 2 parallel
- Used MPI for effective parallelization
- Improved performances of a sequential algorithm using parallel computation
- Performance evaluation using Speedup and Efficiency metrics
- Promising Speedup results, even though Linear Speedup was not achieved

Thanks for your attention!