KNN with threads, OpenMP, and MPI Implementation, Experiment and Analysis

1st Quoc Bao Tran
Department of Computer Science
Virginia Commonwealth University
University Richmond, Virginia 23284
tranq3@vcu.edu

***Abstract*—The k-Nearest Neighbors (KNN) algorithm is a fundamental machine learning technique, but its computational complexity poses challenges for large-scale datasets. This study implements and analyzes parallel versions of the KNN algorithm using POSIX threads, OpenMP, and MPI to address these scalability issues. We compare the performance of these implementations across various dataset sizes (small, medium, and large) and numbers of threads/processes (1 to 128) on the high-performance computing cluster athena.hprc.vcu.edu. Our results demonstrate significant speedups and improved scalability, particularly for larger datasets. We analyze runtime, speedup, and efficiency metrics, estimating the proportion of parallelizable code.**

***Index Terms*—k-Nearest Neighbors (KNN), OpenMP, MPI, POSIX threads**

I. Introduction

Machine learning algorithms are increasingly applied to massive datasets, pushing the limits of computational resources and highlighting the need for efficient, scalable implementations. The k-Nearest Neighbors (KNN) algorithm, despite its simplicity and effectiveness, faces significant challenges when scaled to large datasets due to its computational complexity of O(num_test × num_train × num_features) .

KNN operates by computing distances between each test instance and all training instances, followed by a majority vote among the k nearest neighbors to predict the class of the test instance. While conceptually straightforward, this process becomes computationally intensive as dataset sizes grow, limiting the algorithm's applicability to large-scale machine learning problems.

Parallel and distributed computing techniques offer promising solutions to address these scalability issues. By leveraging multi-core processors and distributed systems, we can potentially reduce computation time and enable the processing of larger datasets. However, the effectiveness of different parallelization strategies for KNN remains an area of active research.

This study aims to implement and analyze three parallel versions of the KNN algorithm:

1) A multi-threaded version using POSIX threads
2) A multi-threaded version using OpenMP
3) A distributed version using MPI (Message Passing Interface)

Our objective is not to improve the accuracy of the KNN algorithm but to enhance its scalability and reduce runtime through parallel and distributed computing techniques. We conduct a comprehensive analysis of these implementations, comparing their performance across various dataset sizes and numbers of threads/processes.

II. Methodology

We implemented three parallel versions of the KNN algorithm: a multi-threaded version using POSIX threads, a multi-threaded version using OpenMP, and a distributed version using MPI. Each implementation aims to improve the scalability of the KNN algorithm for large datasets. We also used the provided serial implementation as a baseline for comparison.

A. Serial Implementation

We began with the provided serial implementation of the KNN algorithm. This version calculates the distance between each test data point and all training data points sequentially. It serves as our baseline for performance comparisons and accuracy verification.

B. Parallel Implementations

1) POSIX Threads Version: In our multi-threaded approach, we leverage the POSIX thread (pthread) API. The workload is distributed across multiple threads, with each thread responsible for computing distances between a subset of test data points and all training data points. Specifically, each thread processes n test points, where n is calculated by dividing the total number of test data points by the number of threads used, the last thread will handle the remainder number of data points (threaded.cpp). Additionally, we also tried the second approach, in which, we divide the test data points by n_2, and divide the train data points by m_2, where $n_2 * m_2$ equal the number of threads used (threaded_both_2_loop.cpp). Unfortunately, the second approach uses too much memory and the runtime is too slow, and we have not figured out how to fix it. So we use the first approach for benchmarking.

2) OpenMP Version: We use the command:

```
#pragma omp parallel
```

for the OpenMP implementation, and we apply the same approach as the first one while doing POSIX threads. We

use OpenMP API to perform each test data point parallel (openmp.cpp).

3) MPI Version: We use the same idea with OpenMP version for this implementation by using $Gatherv()$ function to aggregate data from processes.

III. Experimental results

1) Datasets: We will run the multi-thread versions and the distributed versions on three datasets with different sizes. The small dataset contains 160 test instances and 1184 train instances. The medium dataset contains 740 test instances and 14708 train instances. The large dataset contains 3436 test instances and 61606 train instances. We run the KNN algorithm with k = 3 of the sequential version and parallel and distributed versions. The experiments are run on the Athena servers by using SLURM scripts (saved in $Slurm\ script$ folder), and the detailed data and results are stored in the $logs$ and the $excel$ folders.

2) Performance analysis: We illustrate the execution times in milliseconds (ms) for the sequential version of the KNN algorithm on small, medium, and large datasets in Table I. For our parallel implementations, we measured runtimes across varying numbers of threads (for POSIX threads and OpenMP) and processes (for MPI). Tables II and III illustrate these results for different dataset sizes and degrees of parallelism.

To evaluate the performance improvement of our parallel implementations, we calculate the speedup over the sequential version. The speedup is defined as:

$$speedup = \frac{T_s}{T_p}, \tag{1}$$

where T_s is the total runtime of the sequential version, and T_p is the total runtime of the parallel version.

Tables V and VI demonstrate the speedup of multi-thread versions (POSIX Threads and OpenMP) and multi-process version (MPI) over the sequential version, respectively. Table IV illustrates the speedup of multi-process version (MPI) running on only one node instead of multiple nodes over the sequential version.

To visualize the speedup trends, Figure 1 and Figure 2 show the speedup of the two parallel versions (multi-thread and multi-process) over the sequential version, respectively.

All the experiments have been done 3 times and the results are average runtimes to get more accurate estimates.

Dataset	Runtime (ms)
Small	18.00
Medium	798.33
Large	11659.67

TABLE I
AVERAGE RUNTIME (MS) OF SEQUENTIAL KNN IMPLEMENTATION (K = 3)

TABLE II
AVERAGE RUNTIME (MS) OF PARALLEL KNN IMPLEMENTATIONS (POSIX THREADS AND OPENMP)

Threads	POSIX Threads			OpenMP		
	Small	Medium	Large	Small	Medium	Large
1	19.00	869.00	11,288.00	19.00	810.33	12,488.67
2	19.00	838.00	11,821.00	17.00	500.67	6,587.00
4	19.00	857.00	11,654.00	16.33	303.67	3,710.67
8	10.00	487.00	6,278.00	8.00	295.33	3,650.00
16	10.00	497.00	6,289.00	10.00	313.33	3,646.00
32	6.00	296.00	3,490.00	11.33	335.00	4,032.67
64	7.00	304.00	3,535.00	14.67	331.00	4,471.00
128	7.00	278.00	3,598.00	14.00	412.33	4,888.00

TABLE III
AVERAGE RUNTIME (MS) OF PARALLEL KNN IMPLEMENTATION (MPI) ON ANY NUMBER OF NODES

Processes	Small	Medium	Large
1	13.67	769.67	12,178.33
2	36.33	436.00	6,201.33
4	51.33	239.67	3,125.67
8	66.33	513.00	1,776.33
16	87.00	100.00	1,201.33
32	74.33	166.00	764.67
64	115.00	269.33	887.33
128	5,449.33	4,313.33	7,259.67

TABLE IV
AVERAGE RUNTIME (MS) OF PARALLEL KNN IMPLEMENTATION (MPI) ON ONLY 1 NODE (WE HAVE NOT BEEN ABLE TO RECEIVE THE RESULTS OF 32, 64, 126 PROCESSES YET BECAUSE OF THE BUSYNESS AT THE ATHENA CLUSTER)

Processes	Small	Medium	Large
1	18.00	598.67	11,920.33
2	33.00	282.00	5,783.33
4	20.00	156.00	3,004.00
8	35.67	101.67	1,751.00
16	25.33	79.67	926.00

TABLE V
SPEEDUP OF PARALLEL KNN IMPLEMENTATIONS (POSIX THREADS AND OPENMP)

Threads	POSIX Threads			OpenMP		
	Small	Medium	Large	Small	Medium	Large
1	0.95	0.92	1.03	0.95	0.99	0.93
2	0.95	0.95	0.99	1.06	1.59	1.77
4	0.95	0.93	1.00	1.10	2.63	3.14
8	1.80	1.64	1.86	2.25	2.70	3.19
16	1.80	1.61	1.85	1.80	2.55	3.20
32	3.00	2.70	3.34	1.59	2.38	2.89
64	2.57	2.63	3.30	1.23	2.41	2.61
128	2.57	2.87	3.24	1.29	1.94	2.39

TABLE VI
SPEEDUP OF PARALLEL KNN IMPLEMENTATION (MPI) ON ANY NUMBER OF NODES

Processes	Small	Medium	Large
1	1.32	1.04	0.96
2	0.50	1.83	1.88
4	0.35	3.33	3.73
8	0.27	1.56	6.56
16	0.21	7.98	9.71
32	0.24	4.81	15.25
64	0.16	2.96	13.14
128	0.003	0.19	1.61

TABLE VII
SPEEDUP OF PARALLEL KNN IMPLEMENTATION (MPI) ON 1 NODE ONLY

Processes	Small	Medium	Large
1	1.00	1.33	0.98
2	0.55	2.83	2.02
4	0.90	5.12	3.88
8	0.50	7.85	6.66
16	0.71	10.02	12.59

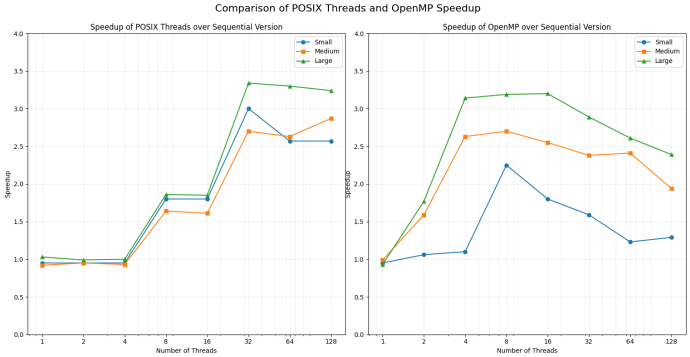

Fig. 1. Speedup of multi-thread POSIX and OpenMP Threads versions over sequential version

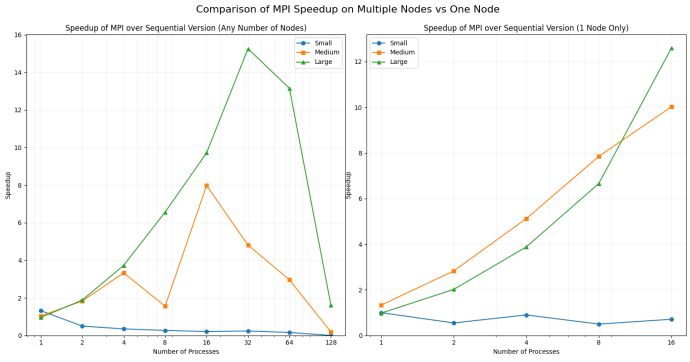

Fig. 2. Speedup of multi-process version (MPI) over sequential version on multiple nodes and on one node

IV. CONCLUSION

Our parallel implementations of the KNN algorithm have demonstrated significant improvements in computational efficiency for both multi-thread and multi-process approaches. Especially for the MPI implementation, it is noticed that the time to perform KNN will be more stable and faster while performing in only one node.

Looking ahead, we plan to extend our research by exploring GPU-based parallelization techniques, which may offer even greater performance enhancements for this algorithm.