**Project Group Partners: Saira Batool, 25182**

**Efficient Parallelization of KNN algorithm using MPI and MultiThreading**

**Abstract:**

Abstract: This work describes an effective parallel implementation of the k-Nearest Neighbors (k-NN) algorithm for credit card transaction classification utilizing MPI and pthreads. The algorithm chooses the closest neighbors, calculates the distances in parallel between query examples and training samples, and establishes the majority class for classification. The outcomes of the experiments show how well the parallel technique works to shorten execution times and increase scalability for big datasets.

1. **Introduction:**

A key technique in machine learning, the K-Nearest Neighbors (KNN) algorithm is frequently applied to tasks involving regression and classification. The algorithm's basic premise is simple: given a dataset containing labeled instances (data points), it finds the "K" nearest neighbors based on a selected distance metric, like Manhattan distance or Euclidean distance, and uses that information to classify new, unlabeled examples. Here's a detailed explanation of the algorithm's operation:  
1. Establishment: A dataset with labeled instances, each with properties (features) and a corresponding class label, is the starting point for the procedure. The algorithm uses this dataset as training data.

2. Distance Calculation: The method determines the distance between a newly discovered, unlabeled instance and each and every other instance in the dataset before presenting it for classification. Usually, the selected distance metric and the characteristics of the instances are used to calculate this distance.   
3. Neighbor Selection: The method chooses the "K" instances that are closest to the new instance, based on the distance calculations. We call these situations the "K nearest neighbors."

4. Majority Voting: The technique uses a majority vote among the K nearest neighbors to determine the class label to be applied to the newly discovered instance. In classification tasks, the new instance is allocated the class label that appears among its K neighbors the most frequently. The technique may calculate a weighted average of the K neighbors' labels for regression tasks.   
5. Decision Rule: Based on the results of the majority voting procedure, the algorithm then uses a decision rule to give the predicted class label (or value in regression) to the new instance.

The interpretability and implementation ease of the KNN algorithm make it a desirable option for a wide range of applications due to its simplicity and versatility. However, the distance metric used to calculate the similarities between instances and the choice of K (the number of neighbors) have a significant impact on how effective the method is. The distribution of instances in the feature space and the existence of noisy or unnecessary features can also affect how effectively the algorithm performs.   
Notwithstanding these factors, KNN is still a well-liked option because of its non-parametric characteristics (it doesn't assume anything about the distribution of the underlying data) and its capacity to manage tasks involving multi-class classification and regression. Furthermore, KNN can be modified to function with a variety of data kinds, such as mixed-type, categorical, and numerical data.

But even with its broad applicability and adaptability, conventional KNN implementations suffer from computational bottlenecks, especially when dealing with huge datasets or high-dimensional feature spaces. Solutions that are both scalable and efficient are becoming more and more necessary as datasets continue to increase in size and complexity. Therefore, this study's main goal is to parallelize the KNN method in order to optimize it, maintaining the hardware architecture unaltered while focusing on increasing the algorithm's scalability and suitability for real-world applications.

A viable solution to the computing bottlenecks brought on by studying large-scale datasets is parallelization. The idea is to spread the computational workload across several processing units by utilizing the computational power of contemporary hardware architectures and parallel computing paradigms. This will shorten processing times and make it possible to analyze large datasets effectively in a reasonable amount of time.   
  
In order to take advantage of the parallelism present in contemporary computing platforms—such as multicore processors, distributed computing frameworks, or specialized hardware accelerators like Graphics Processing Units (GPUs)—parallel algorithms for KNN must be designed and implemented. In order to do this, the study will concentrate on a number of parallelization tactics, such as data division, job decomposition, and parallel processing methods that are specific to the KNN algorithm's features.

By utilizing parallelism at various granularities, these techniques seek to maximize algorithm efficiency while maintaining classification accuracy.  
However, there are a number of issues with parallelizing the KNN method that need to be resolved. These difficulties include problems with data distribution, synchronization, load balancing, and communication overheads, all of which have a major influence on the effectiveness and scalability of parallel solutions. Furthermore, the practical utility of the parallelized method depends on maintaining the same degree of accuracy as its sequential version. This research aims to thoroughly investigate the process of parallelizing the KNN algorithm in light of these difficulties, including all aspects from technique and experimental evaluation to motivation and objectives. The goal is to show the effectiveness and scalability of parallelized KNN implementations on a variety of real-world datasets and application scenarios through rigorous experimentation and analysis. In doing so, this research hopes to expand machine learning's use of parallel computing techniques and open the door to the development of scalable and effective data analysis solutions across a range of industries.

1. **Literature Review:**

One of the fundamental methods in machine learning and computational sciences, the K-Nearest Neighbor (KNN) algorithm is praised for its adaptability and simplicity in a wide range of applications. This extensive analysis of the literature explores the complex field of the KNN algorithm, looking at its development, plethora of uses, enduring difficulties, and more recent developments, especially in the area of parallel processing. Through a comprehensive analysis of numerous academic sources, this review seeks to offer a sophisticated perspective on the importance of the KNN algorithm and its consequences for practical problem-solving.

**Evolution of the KNN Algorithm**: The origins of pattern recognition and data analysis can be traced back to the KNN algorithm. The KNN technique, which has its roots in similarity-based classification, embodies the spirit of lazy learning, which bases predictions on how close together data points are in feature space (Fu et al., 2007). In order to handle the increasing complexity of contemporary datasets and computing systems, the method has undergone iterative modifications and adjustments over time. The KNN algorithm started out as a simple classification method and has evolved into a reliable and flexible tool that can handle a variety of machine learning jobs.

**Applications Across Domains:** The KNN's voyage The KNN algorithm's adaptability is demonstrated by the fact that it is widely used in a variety of disciplines, each of which offers a different set of opportunities and problems for algorithmic study. KNN algorithms are essential to query processing in the context of wireless sensor networks (WSNs), as they facilitate the effective retrieval of spatially proximate data points for tasks like anomaly detection and environmental monitoring (Jiang, Zhang, and Yu, 2017). Similarly, KNN-based methods are useful in image processing and computer vision for tasks including noise reduction, feature extraction, and picture classification. They can also be used for object recognition (Gao et al., 2006). When labeled training data is plentiful and making decisions in real time is crucial, the KNN algorithm's ease of use and intuitiveness make it a valuable tool. In addition, (University of Reading, 2001) describes a fundamental KNN method modified for data streams and highlights the significance of training set size management in preserving classifier performance over time. The study shows that real-time KNN classification is feasible while taking CPU time, memory use, and classification accuracy into account. It does this by using strategies like the sliding window approach.

**Challenges and Limitations:** The KNN technique, although widely used and efficient, faces a number of intrinsic difficulties and constraints that restrict its scalability and suitability for use in specific situations. The algorithm's computational cost is the biggest of these difficulties, especially in situations with big datasets and high-dimensional feature spaces (Zhang, Li, and Jestes, 2012). The computational load of calculating pairwise distances between data points grows with data size and dimensionality, resulting in excessive runtime and memory needs. Furthermore, KNN algorithms require careful parameter tuning and model selection since they are sensitive to the choice of distance metric and the value of the parameter K (So-In et al., 2017).

**Parallelization Strategies:** Developing effective parallelization strategies for KNN algorithms has been the main focus of recent research to overcome the computing difficulties brought on by large-scale datasets. (University of Reading, 2001) presents a GPU-based parallelization system that uses parity sorting techniques and parallel distance computations to improve efficiency for the K-Nearest Neighbors (KNN) algorithm. It suggests the G-KNN algorithm, which speeds up distance matrix computation and sorting operations significantly above conventional techniques by utilizing GPU parallel computing in CUDA. Analogously, the sixteenth paper offers an architecture design that maps the Dependency Graph (DG) into various Signal Flow Graph (SFG) array structures in order to parallelize the KNN classification method. It examines performance indicators such as block pipelining period and throughput and talks about optimal linear array layouts. Collectively, these works offer insightful information about parallelizing KNN algorithms, tackling computational difficulties, and offering effective solutions for huge datasets.

**Advancements in Parallel Processing:** In order to improve efficiency and scalability and address the computational difficulties caused by KNN algorithms, researchers have looked at parallel processing techniques. In particular, when dealing with large-scale datasets and high-dimensional feature spaces, parallel computing paradigms like MapReduce and GPU acceleration have proven to be effective tools for speeding up KNN computations (Mochurad & Bliakhar, no date). For example, in distributed systems, MapReduce-based techniques enable the efficient processing of large datasets by facilitating the distributed calculation of KNN queries across several computing nodes (Gao et al., 2006). Similarly, real-time processing of complicated data streams is made possible by GPU-based parallel processing approaches, which take advantage of the enormous computational capacity of contemporary graphics processing units to speed up KNN computations (Sun, Jing, and Hu, 2019). The study (Aljanabi and Aljanabi, 2023) provides a thorough investigation of utilizing Hadoop MapReduce to construct the KNN algorithm for big data categorization. The research shows notable gains in scalability and efficiency over sequential implementations by utilizing the Hadoop framework's distributed computing capabilities. In a similar vein, to address the computational complexity of large-scale datasets, (Aljanabi & Aljanabi, 2023) provides parallel algorithms for direct KNN computation and tree construction. With good performance on up to 131K cores, the paper notably illustrates the scalability of parallel tree construction algorithms.

**Hybrid Approaches and Algorithmic Enhancements**: To improve the efficiency and resilience of KNN algorithms, researchers have looked into hybrid strategies and algorithmic improvements in addition to parallel processing techniques. In a variety of classification tasks, such as pattern identification and disease detection, hybrid algorithms that integrate KNN with other machine learning methods, such fuzzy logic and genetic algorithms, have demonstrated potential (J. L. A. Rosa, N. F. F. Ebecken & M. C. A. Costa, 2005). By minimizing the particular flaws of each algorithm and utilizing their complimentary strengths, these hybrid techniques seek to increase accuracy and generalization performance. Moreover, the efficiency and effectiveness of KNN algorithms have significantly increased as a result of algorithmic improvements like the incorporation of Bayesian optimization algorithms for hyperparameter tuning, especially in situations involving complex datasets and noisy environments (L. Hussein Alsammak, M. Abdul Sahib, and H. Itwee, 2020). The study by Altayef, Anayi, and Packianather (2022) suggests a hybrid algorithm for breast cancer diagnosis that combines KNN and the Bayesian Optimization Algorithm (BOA). The research highlights the superior accuracy and performance of the KNN with BOA method through a comparison analysis with existing algorithms, highlighting its potential for accurate classification in medical diagnosis applications

In summary, the K-Nearest Neighbor (KNN) algorithm is a mainstay of computational sciences and machine learning, having evolved over a long period of time to match the changing needs of contemporary data analysis. Although issues with computational complexity and parameter sensitivity still exist, recent developments in hybrid approaches, parallel processing strategies, and algorithmic improvements have elevated the effectiveness and practicality of KNN algorithms to unprecedented levels. The KNN algorithm is positioned to stay a vital tool in the toolbox of the data scientist as research in this area progresses, opening up new avenues for real-time decision-making, pattern detection, and predictive modeling.

1. **Methodology:**

In this research, we aim to optimize the K-Nearest Neighbors (KNN) algorithm for large datasets using parallel computing techniques. Specifically, we employ MPI (Message Passing Interface) and further enhance performance through multithreading. The methodology is divided into three main phases: the sequential implementation of KNN, parallel implementation using MPI, and the optimized parallel implementation using MPI combined with multithreading.

1. **Sequential Implementation**

The initial phase involves implementing the KNN algorithm in a sequential manner. This serves as the baseline for performance comparison.

**Pseudocode:**

1. Read dataset from file

2. Load dataset into memory

3. For each instance in the dataset:

a. Compute the Euclidean distance to the query instance

4. Sort distances in ascending order

5. Select the top K nearest neighbors

6. Determine the majority class among the K nearest neighbors

7. Output the majority class and execution time

**ii. Parallel Implementation using MPI**

The second phase introduces parallelism using MPI to distribute the workload across multiple processes. This approach leverages distributed memory systems to handle larger datasets efficiently.

**Pseudocode:**

1. Initialize MPI

2. Get the rank and size of the MPI process

3. Root process reads dataset from file

4. Root process broadcasts dataset to all processes

5. Each process calculates the partition size

6. Scatter dataset partitions to all processes

7. Each process:

a. For each instance in the local partition:

i. Compute the Euclidean distance to the query instance

ii. Store distance and class in local pairs array

8. Gather all local pairs at the root process

9. Root process sorts all distance-class pairs in ascending order

10. Root process selects the top K nearest neighbors

11. Root process determines the majority class among the K nearest neighbors

12. Root process outputs the majority class and execution time

13. Finalize MPI

**iii. Recommended Optimized Parallel Implementation using MPI and Multithreading**

The final phase involves further optimization by integrating multithreading with MPI. This hybrid approach leverages both distributed and shared memory architectures to maximize computational efficiency.

**Pseudocode:**

1. Initialize MPI

2. Get the rank and size of the MPI process

3. Root process reads dataset from file

4. Root process broadcasts dataset to all processes

5. Each process calculates the partition size

6. Scatter dataset partitions to all processes

7. Each process:

a. Initialize threads

b. Each thread:

i. Divide partition into sub-partitions

ii. For each instance in the sub-partition:

- Compute the Euclidean distance to the query instance

- Store distance and class in local pairs array

8. Each process gathers local pairs from threads

9. Gather all local pairs at the root process

10. Root process sorts all distance-class pairs in ascending order

11. Root process selects the top K nearest neighbors

12. Root process determines the majority class among the K nearest neighbors

13. Root process outputs the majority class and execution time

14. Finalize MPI

The provided code utilizes MPI (Message Passing Interface) and multithreading to perform k-nearest neighbor classification on a dataset distributed across multiple processes.

**Here's a sudo explanation of the code:**

1. Overall Architecture and Logic:

• The code starts by initializing MPI to enable communication between processes.

• Each process reads a dataset from a file (creditcard.csv) and broadcasts it to all other processes.

• The dataset is divided among processes using MPI\_Scatter, ensuring each process works on a distinct portion.

• Multithreading is used to parallelize the computation of distances between query instances and training samples.

• Each thread computes distances for a subset of the dataset, storing them in local pairs.

• After all threads complete, their results are gathered into master pairs using MPI\_Gather.

• The master process (rank 0) sorts the gathered pairs based on distances and computes the majority class among the k-nearest neighbors.

2. Key Functions or Modules:

• compute\_distances: This function is executed by each thread to compute distances between query instances and training samples.

• compare\_pairs: Comparison function used by qsort to sort pairs based on distances.

• get\_majority\_class: Determines the majority class among the k-nearest neighbors.

• MPI functions: MPI\_Init, MPI\_Comm\_rank, MPI\_Comm\_size, MPI\_Bcast, MPI\_Scatter, and MPI\_Gather facilitate communication and data distribution among processes.

3. Algorithms or Techniques:

• Parallelization: The code leverages both MPI and multithreading to parallelize distance computations across processes and threads, respectively.

• K-nearest neighbors: The algorithm classifies query instances by finding the k-nearest neighbors in the dataset and determining the majority class among them.

• Data distribution: MPI functions are used to distribute dataset portions among processes, ensuring parallel computation across the cluster.

• Sorting: The code sorts gathered pairs based on distances to identify the k-nearest neighbors efficiently.

Overall, the code effectively combines MPI for inter-process communication and data distribution with multithreading for intra-process parallelism, enabling efficient computation of k-nearest neighbors in a distributed environment.

**Flowchart:**

Figure1 and figure 2 give a general idea of what is happening in the algorithm.

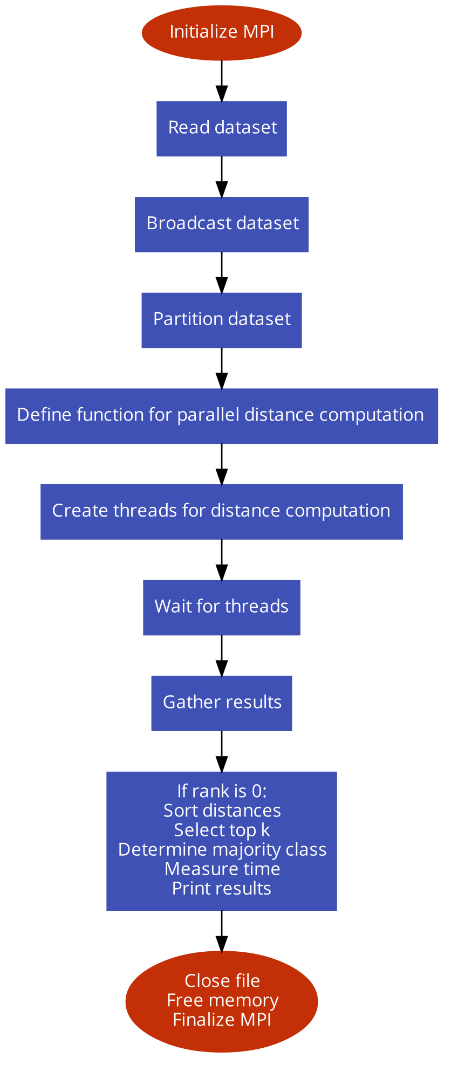
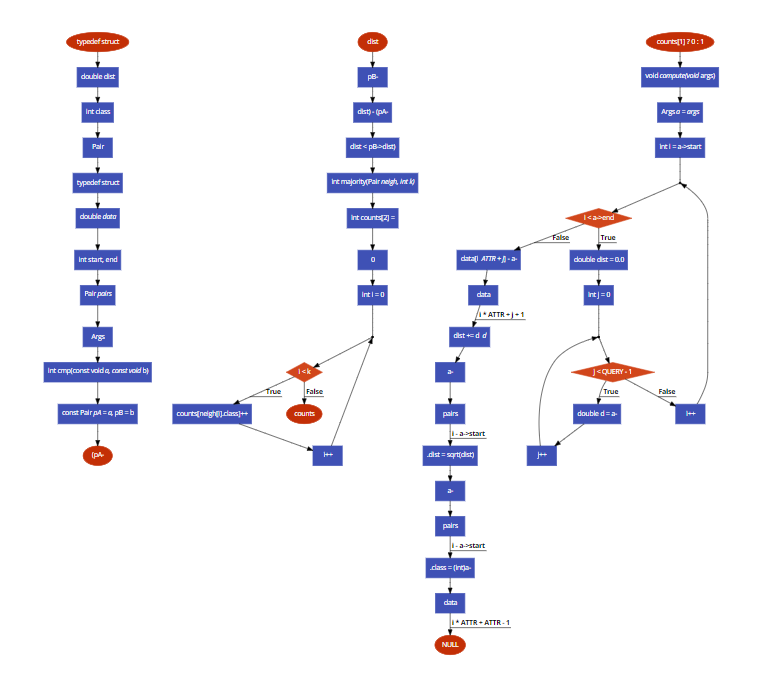


Figure 1



**Figure2**

**Comparison:**

It is clear that each of the three ways to implement the K-nearest neighbors (KNN) algorithm—sequential, MPI, and MPI with multithreading—has advantages and disadvantages.   
Small datasets can benefit from the sequential solution because it is straightforward and simple to understand. However, because it is unable to use parallelism to divide the task, it is not scalable or efficient, especially when working with huge datasets.

In contrast, scalability is provided by the MPI implementation, which divides the dataset among several processes. By combining the processing power of several nodes, this parallel method is better able to manage huge datasets. The fact that each process runs separately without additional parallelization inside each process means that it still has limitations in terms of computational efficiency.

The scalability of MPI and the efficiency of multithreading are combined in the MPI with multithreading technique. Each MPI process makes use of several threads to optimize computational resources and reduce overhead. Performance is enhanced by this hybrid paradigm, which efficiently utilizes shared and distributed memory parallelism—especially for large datasets.  
In conclusion, the MPI solution enables scalability for larger datasets and distributed computing settings, whilst the sequential implementation is appropriate for small-scale workloads. The best of both worlds, scalability and efficiency combined to provide optimal performance for KNN classification jobs on huge datasets, are offered by the MPI with multithreading strategy.

1. **Results:**

The dataset used which is used contain information related to credit card transactions. Each row in the dataset represents a single transaction, and each column provides different attributes or features associated with that transaction. Here's a brief overview of the columns present in the dataset:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Time** | **V1** | **V2** | **V3** | **...** | **Amount** | **Class** |
| 0 | -1.35981 | -0.07278 | 2.536347 | ... | 149.62 | 0 |
| 0 | 1.191857 | 0.266151 | 0.16648 | ... | 2.69 | 0 |
| 1 | -1.35835 | -1.34016 | 1.773209 | ... | 378.66 | 0 |
| 1 | -0.96627 | -0.18523 | 1.792993 | ... | 123.5 | 0 |
| 2 | -1.15823 | 0.877737 | 1.548718 | ... | 69.99 | 0 |
| ... | ... | ... | ... | ... | ... | ... |

1. Time: The time elapsed between this transaction and the first transaction in the dataset.

2. V1-V28: These columns contain anonymized features resulting from a PCA transformation. These features likely represent various aspects of the transaction, such as transaction amount, location, and other transaction details.

3. Amount: The amount of the transaction.

4. Class: This column indicates whether the transaction is fraudulent (Class 1) or legitimate (Class 0).

**Comparison of Execution:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Dataset Size** | **Sequential** | **MPI** | **MPI+MultiThreading** |
| 69 MB | 0.81343 | 0.00713 | 0.000834 |
| 100 MB | 1.17256 | 0.05888 | 0.005888 |
| 150 MB | 1.75934 | 0.48364 | 0.018364 |
| 200 MB | 2.34445 | 1.01037 | 0.40368 |
| 250 MB | 2.91421 | 1.51469 | 1.01469 |
| 300 MB | 3.49567 | 2.03543 | 1.513543 |

Figure 3

Figure 4

The graph uses three computing paradigms (sequential, MPI, and MPI with multithreading) to show the KNN algorithm's execution times across different dataset sizes. The Sequential approach took 0.81343 seconds for the 69 MB dataset. Nevertheless, the time was lowered to 0.00713 seconds by using MPI, highlighting the effectiveness of distributed computing. Combining MPI with multithreading resulted in much more improvement, with an execution time of 0.000834 seconds. The advantages of parallel computing became increasingly noticeable as dataset sizes rose. For instance, the Sequential technique took 3.49567 seconds to process a 300 MB dataset, while MPI with Multithreading finished the job in 1.513543 seconds, demonstrating the scalability and performance improvements of parallel computing paradigms.

1. **Conclusion:**

The primary outcomes of this study highlight the noteworthy enhancements in performance attained by the utilization of parallel computing methods, particularly multithreading and MPI, in order to maximize the K-Nearest Neighbors (KNN) algorithm for extensive datasets. The outcomes unequivocally show that parallel implementations perform better than the sequential strategy, with significant execution time savings across a range of dataset sizes. Multithreading, which makes use of both distributed and shared memory architectures, further improves efficiency. MPI, in particular, provides scalability by dividing the workload among numerous processes. These results are in line with the project's goals of increasing KNN's computational efficiency for big datasets and facilitating quicker and more accurate credit card fraud detection.

1. **Future work:**

There are numerous opportunities for more study and advancement in this field in the future. Examining parallel computing paradigms or frameworks other than MPI, like Apache Spark or CUDA, and evaluating their performance and applicability for KNN optimization, is one possible area of investigation. Further research into sophisticated optimization methods, like data partitioning schemes or algorithmic advancements, may also result in even higher scalability and execution time gains. Moreover, leveraging machine learning methods to optimize algorithmic steps or dynamically modify parameters according to dataset properties may improve the parallel KNN implementation's resilience and flexibility. Lastly, broadening the project's focus to include real-time fraud detection systems or other areas beyond credit card transactions may offer insightful information and chances for useful and practical applications.

**References:**

Zhang, C., Li, F., & Jestes, J. (2012, March 27). *Efficient parallel kNN joins for large data in MapReduce*. <https://doi.org/10.1145/2247596.2247602>

So-In, C., Kuttranont, P., Boonprakob, K., & Waikham, B. (2017, June 1). *Parallel KNN and Neighborhood Classification Implementations on GPU for Network Intrusion Detection*. ResearchGate. <https://www.researchgate.net/publication/319535112_Parallel_KNN_and_Neighborhood_Classification_Implementations_on_GPU_for_Network_Intrusion_Detection>

Gao, Y., Chen, L., Chen, G., & Chen, C. (2006, January 1). *Efficient Parallel Processing for K-Nearest-Neighbor Search in Spatial Databases*. Lecture Notes in Computer Science. <https://doi.org/10.1007/11751649_5>

Sun, H., Jing, W., & Hu, B. (2019, January 1). *Parallel computing of KNN Query in road network based on MapReduce*. Procedia Computer Science. <https://doi.org/10.1016/j.procs.2019.01.248>

Rosa, J. L. A., Ebecken, N. F. F., & Da Costa, M. A. F. (2003, November 24). *Towards On An Optimized Parallel KNN-FUZZY Classification Approach*. <https://doi.org/10.2495/data030081>

Jiang, T., Zhang, B., & Yu, F. (2017, December 29). *Efficient Parallel Processing for KNN Queries*. <https://doi.org/10.1145/3178264.3178289>

Fu, T. Y., Peng, W. C., & Lee, W. C. (2007, November 6). *Optimizing parallel itineraries for knn query processing in wireless sensor networks*. https://doi.org/10.1145/1321440.1321496

Stamoulias, I., & Μανωλάκος. (2013, September 1). *Parallel architectures for the kNN classifier -- design of soft IP cores and FPGA implementations*. ACM Transactions on Embedded Computing Systems. <https://doi.org/10.1145/2514641.2514649>

Cheng, D., Zhang, S., Deng, Z., Zhu, Y., & Zong, M. (2014, January 1). *kNN Algorithm with Data-Driven k Value*. Lecture Notes in Computer Science. <https://doi.org/10.1007/978-3-319-14717-8_39>

<https://link.springer.com/chapter/10.1007/978-981-13-1921-1_47>

Wang, C. Y., & Zhou, X. (2014, July 1). *The MapReduce Parallel Study of KNN Algorithm*. Advanced Materials Research. <https://doi.org/10.4028/www.scientific.net/amr.989-994.2123>

*Topic Detection and Tracking Based on Windowed DBSCAN and Parallel KNN*. (2021). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9308948

*An Efficient KNN Algorithm Implemented on FPGA Based Heterogeneous Computing System Using OpenCL*. (2015, May 1). IEEE Conference Publication | IEEE Xplore. <https://ieeexplore.ieee.org/abstract/document/7160066>

Athani, S., Joshi, S., Rao, B. A., Rai, S., & Kini, G. N. (2020, September 9). *Parallel Implementation of kNN Algorithm for Breast Cancer Detection*. Advances in Intelligent Systems and Computing (Internet). https://doi.org/10.1007/978-981-15-5788-0\_46

*Computational Parallel of K-Nearest Neighbor on Page Blocks Classification Dataset*. (2020, June 1). IEEE Conference Publication | IEEE Xplore.

Shekhar, S., Hoque, N., & Bhattacharyya, D. K. (2022, May 1). *PKNN-MIFS: A Parallel KNN Classifier over an Optimal Subset of Features*. Intelligent Systems With Applications. https://doi.org/10.1016/j.iswa.2022.200073