Implementing Parallel Computing to Enhance the Performance of Agglomerative Hierarchical Clustering (AHC) Algorithm

A PROJECT REPORT

21ES631 Distributed Computing

Submitted by

Cheppalli Varun CB.EN.P2EBS24006

Katta Sreenivas CB.EN.P2EBS24023

MASTER OF TECHNOLOGY

IN

EMBEDDED SYSTEMS

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

AMRITA SCHOOL OF ENGINEERING, COIMBATORE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

APRIL 2025

ABSTRACT

Clustering is perhaps the most significant unsupervised machine learning method. It is extensively used in data science, bioinformatics, and pattern recognition. Agglomerative Hierarchical Clustering (AHC) is one of the most widely used clustering algorithms. AHC builds a hierarchy of clusters without knowing the number of clusters in advance. It is computationally intensive, however, since it calculates a lot of pair-wise distances and merging groups and therefore it is not suitable for large data.

To address these issues, parallel computation with MPI via the mpi4py library using Python is employed in this study. AHC jobs are distributed across several processors to ensure it is fast and scalable. We monitor performance indicators such as running time, silhouette score, and speedup to compare parallel and sequential implementations of AHC. The parallel implementation demonstrates drastic efficiency gains without compromising clustering accuracy.

Also, the parallel AHC performance is compared with Dijkstra's algorithm. Although Dijkstra's algorithm is not a clustering algorithm, it is a suitable comparison since it is efficient in graph problems. The comparison highlights the strength of parallel AHC in handling large datasets, demonstrating that it can perform efficiently in real-world scenarios where the standard methods fail.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Clustering is a simple data science method that puts objects into groups depending on their similarity in structure or features. Agglomerative Hierarchical Clustering (AHC), which is one of the most popular clustering algorithms, is commonly employed because it can generate a dendrogram—a tree-like diagram of how data points are clustered. Unlike algorithms such as K-means, AHC does not require knowing in advance the number of clusters, which is why it is particularly useful in applications such as bioinformatics, image and language processing.

Although AHC is simple to comprehend and efficient, it has one major disadvantage—its high computational requirements. With a time complexity of $O(n^2)$ or worse, depending on the recursive calculation of distance matrices and merge operations, AHC is slow when applied to large datasets, particularly when run sequentially. This study seeks to alleviate this by using parallel computing through the MPI4py library in Python, where the process can be split among numerous processors. The objective is to enhance scalability and reduce execution time without compromising clustering accuracy.

To assess parallel AHC's performance, this research compares it with Dijkstra's algorithm. Dijkstra's algorithm is among the most popular shortest-path algorithms with excellent performance navigating graphs. Although Dijkstra's algorithm is not a clustering algorithm, we use it as a benchmark for performance testing, particularly speed and complexity. The comparison enables us to see how parallel computing accelerates AHC and demonstrates how it compares with other popular algorithms in data processing.

Chapter 2

LITERATURE REVIEW

This chapter reviews related works in the domain of predictive maintenance and machine learning-based disk failure analysis. With the rise in data-driven infrastructure and widespread use of storage systems in data centers, understanding the behavior and reliability of storage devices has become critical. Several studies have explored failure prediction using various modeling techniques and data-driven insights, which provide a foundation for the present work.

2.1 A Framework for Parallel Hierarchical Agglomerative Clustering using Nearest-Neighbor Chain

This paper[3] presents a parallel solution for Hierarchical Agglomerative Clustering (HAC) by leveraging the Nearest-Neighbor Chain (NNC) technique to overcome the limitations of traditional sequential clustering approaches. Standard HAC algorithms are computation-heavy and hard to scale due to repeated pairwise comparisons and complex merging procedures. The introduction of the NNC method helps avoid redundant distance calculations and streamlines the merging process by maintaining a dynamic chain of nearest neighbors, thereby making the algorithm more suitable for parallel execution.

Designed specifically for multi-core shared memory systems, the framework efficiently distributes tasks and reduces synchronization overhead. The proposed solution demonstrates significant performance improvements in both speed and scalability when tested on large datasets. It outperforms other existing parallel HAC approaches and proves effective in real-world applications such as bioinformatics, social network analysis, and image processing. The study shows how carefully restructured algorithms can unlock parallelism even in inherently sequential methods like HAC.

2.2 Implementing Parallel Computing to Enhance the Performance of K-mean Algorithm

This paper [1] discusses the application of parallel computing to enhance the performance of the K-means clustering algorithm. Aware of the computationally intensive nature of the sequential K-means algorithm—particularly in dealing with large data—the authors propose a parallel implementation employing multi-threading to share the workload. The aim is to enhance the run time and facilitate easier manipulation of larger datasets without sacrificing the clustering outcome.

The results show a considerable decrease in execution time with the increase in the number of threads, justifying the scalability of the approach. The study provides valuable information on how clustering algorithms, even basic ones like K-means, are significantly enhanced by parallelization techniques. It points out the utilization of computational resources and parallel frameworks towards improving clustering performance in real-time or data-intensive systems.

2.3 Distributed Hierarchical Clustering Algorithm Utilizing a Distance Matrix

This paper[2] presents a distributed hierarchical clustering algorithm based on a distance matrix to process large data sets on multiple processing nodes. The underlying idea is to distribute both the computation and memory overheads of computing and storing pairwise distances. Conventional hierarchical clustering is beset by memory constraint and inefficiency in processing with increasing data size, which this paper seeks to remove through a distributed approach.

By utilizing distributed memory systems, the algorithm reduces the computational bottleneck and allows processing times to be quicker even for the case of complicated clustering operations. The paper also discusses how the distributed distance matrix can be updated in parallel, which makes the overall efficiency of the clustering process. The approach is especially useful in big data applications where

centralized clustering methods do not scale well

2.4 A Novel Parallelization Approach for Hierarchical Clustering

This paper [4] suggests a novel method of parallelizing hierarchical clustering by reengineering the algorithm underlying it for better adaptation in parallel computing environments. Unlike traditional methods involving complete linkage or average linkage algorithms, this method reexamines the clustering algorithm for dividing work among processors for parallel computation of merge and distances.

The suggested approach is applied and tested in a bioinformatics environment, and it is found to enhance speed as well as efficiency without compromising the performance of the clustering. The research presents theoretical as well as experimental evidence in favor of the parallelization technique and thus is a handy handbook for researchers attempting to parallelize high-throughput data hierarchical clustering algorithms.

2.5 Multiclass Classification of Dry Beans Using Computer Vision and Machine Learning Techniques

In this paper[5], the authors propose a machine learning method that classifies dry beans in more than one class based on computer vision features. A single large dataset of beans is utilized, from which multiple shape, texture, and color features are extracted. These features are utilized to train and test different machine learning algorithms to estimate the classification performance.

The paper compares the accuracy, recall, F1-scores, and precision of different classifiers and concludes that machine learning with properly extracted features can effectively classify the types of beans. Although not working with clustering, the paper is applicable due to its methodological complexity and use of real-world agricultural datasets, serving as a stimulus for preprocessing and evaluation techniques

in unsupervised learning problems.

2.6 Summary

The literature under review highlights the inefficiency of classical Agglomerative Hierarchical Clustering (AHC) algorithms in dealing with high-dimensional datasets due to their space and computational complexity. In response to such inefficiencies, various studies have investigated parallel and distributed solutions. Methods like Nearest-Neighbor Chain (NNC) and distributed distance matrix computation have achieved considerable reductions in execution time and scalability. Such methods show the effectiveness of reorganized algorithms and task allocation in making clustering manageable for high-dimensional data. Parallel systems using MPI, especially, enable efficient sharing of tasks and removal of redundant operations.

Inspired by these results, the current study uses a parallel computing approach by means of MPI4py in order to optimize the performance of AHC. In addition, through the comparison of parallel AHC with that of Dijkstra's algorithm—an optimized graph algorithm—the study displays a broader context of computational performance. The work in the literature also identifies such performance metrics as execution time, silhouette score, and speedup, which underpin the study's evaluation herein. These results altogether form the premise of project design and motivate its objective to optimize clustering performance through parallelization

Chapter 3

OBJECTIVE

Agglomerative Hierarchical Clustering (AHC) and shortest path computation are fundamental in various fields such as data analysis and network optimization. Traditional implementations often face challenges in handling large datasets and complex computations efficiently. The lack of parallelization in these algorithms results in high execution times and suboptimal performance.

This work aims to improve the quality and efficiency of both AHC and shortest path algorithms by applying advanced techniques and parallel processing. The objectives include enhancing clustering results, optimizing shortest path calculations, and evaluating performance improvements through parallelization. The final goal is to reduce execution time and improve the accuracy and quality of both algorithms, making them more suitable for large-scale applications.

Chapter 4

DATASET SPECIFICATION

4.1 Dry Bean Dataset

Dry Bean Dataset is created based on a research study to categorize dry bean types based on shape-based morphological attributes derived from digital images. Individual beans were imaged, and statistical image processing methods were employed to obtain 16 quantitative features that describe the shape and geometry of the beans. The dataset is created with a view towards supervised machine learning operations, particularly classification. This information is a baseline to contrast algorithms for use in applications of agricultural image classification, pattern recognition, and computer vision.

4.2 Dataset Overview

The dataset used in this study is publicly available and provided in CSV format. It contains labeled data instances representing seven different varieties of dry beans, each with a range of morphological features.

- **Source:** UCI Machine Learning Repository
- **Time Period:** Not time-series based; static collection from lab experiments
- **Total Records:** 13,611 instances
- **Data Type:** Multiclass classification with real-valued numerical features
- **Number of Features:** 16 numerical features + 1 class label
- **Associated Task:** Multiclass classification (7 classes of bean types)

4.3 Key Features and Metadata

Each record in the dataset contains the following attributes:

Metadata Fields

- **Class:** Categorical label indicating bean variety. Possible classes: `SEKER`, `BARBUNYA`, `BOMBAY`, `CALI`, `HOROZ`, `SIRA`, `DERMASON`

Feature Fields (Numerical)

- **Area:** Total pixel area of the bean
- **Perimeter:** Length of the outer boundary
- **MajorAxisLength:** Longest line that can pass through the bean
- **MinorAxisLength:** Longest line perpendicular to the major axis
- **Eccentricity:** Deviation of the shape from a perfect circle
- **ConvexArea:** Area of the convex hull
- **EquivDiameter:** Diameter of a circle with the same area
- **Extent:** Ratio of area to bounding box
- **Solidity:** Area / Convex hull area
- **Roundness:** $4\pi \times \text{Area}/(\text{Perimeter}^2)$
- **AspectRation:** Ratio of MajorAxisLength to MinorAxisLength
- **Compactness:** $\sqrt{(4 \times \text{Area}/\pi)}/\text{MajorAxisLength}$
- **ShapeFactor1:** Area / Perimeter
- **ShapeFactor2:** Area / (Bounding box area)
- **ShapeFactor3:** Perimeter^2 / Area
- **ShapeFactor4:** Perimeter / MajorAxisLength

4.4 Data Collection and Reliability

The data was captured under controlled imaging conditions to reduce noise and enhance feature extraction reliability. The bean samples were positioned and examined carefully using digital imaging software to obtain accurate morphological feature capture. The dataset is balanced over the various classes and thus is well-suited for training strong classification models without requiring heavy preprocessing or sampling methods.

4.5 Relevance to Agricultural Classification and Computer Vision

This dataset plays a critical role in agricultural automation and crop classification. It can be utilized to:

- Train models for automatic seed classification and quality control
- Assist in computer vision applications in farming
- Enhance decision-making in agricultural supply chains

The clean and labeled nature of this dataset also makes it a suitable candidate for comparing traditional machine learning techniques like SVM, KNN, and Decision Trees, as well as deep learning pipelines.

4.6 Addressing Multiclass Challenges

Although the dataset is balanced, there exists mild inter-class similarity between bean classes, hence making classification difficult. Feature selection (for example, feature selection with higher-order statistics), dimensionality reduction (for example, PCA), and ensemble methods can be utilized to enhance the performance of classification. Cross-validation methods are advised to enable generalization against all types of beans.

4.7 Conclusion

The Dry Bean Dataset is a strong and practical tool for the development and testing of classification algorithms in the areas of computer vision and agriculture. Its large set of morphological features, balanced class distribution, and accurate labeling make it well-suited to the testing of multiclass classification models. Additionally, the relevance of the dataset extends to both academic research and industrial applications in the areas of smart farming and food technology.

Chapter 5

METHODOLOGY

5.1 Data Preprocessing

The data sets utilized in this research are the Dry Bean Dataset of the UCI Machine Learning Repository. It contains the physical and morphological features found in various dry beans. Each instance is described by 16 continuous features and one categorical class label that distinguishes the bean type.

A judicious preprocessing was conducted to maintain compatibility with clustering and graph-based algorithms. The following steps outline the methodology:

5.1.1 Data Cleaning

- **Missing Values:** The dataset does not contain missing values. Data integrity checks were conducted to confirm completeness.
- **Type Consistency:** All features were converted to numerical format (`float64`), and the class label was encoded as integers.
- **Duplicate Samples:** Duplicate records were checked and removed to ensure uniqueness and avoid bias during clustering.

5.1.2 Label Encoding

527 wordsclear Humanize AI Class names (i.e., SEKER, BARBUNYA, HOROZ) were labeled using LabelEncoder, permits numerical calculations necessary for graphing and clustering. It step guarantees that all categorical variables are mapped to machine-readable form.

5.1.3 Normalization

All features were normalized using Min-Max scaling:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{5.1}$$

This ensured that all features contribute equally to distance calculations. This step is particularly important for distance-based clustering algorithms like AHC, where scale-sensitive features may distort proximity calculations.

5.2 Agglomerative Hierarchical Clustering (AHC)

AHC is a bottom-up clustering method that starts with every data point as a singleton clusters and agglomerates them step by step. A dendrogram, or tree-like hierarchical structure, is constructed through the comparison of cluster similarity metrics

5.2.1 Without MPI

- **Distance Metric:** Euclidean distance was used to measure similarity between points.
- **Linkage Criterion:** Ward's linkage method was employed to minimize intra-cluster variance:

$$D(A, B) = \frac{|A||B|}{|A| + |B|} ||\bar{x}_A - \bar{x}_B||^2 \tag{5.2}$$

- **Silhouette Analysis:** Used to determine the optimal number of clusters by maximizing the average silhouette coefficient:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \tag{5.3}$$

  where $a(i)$ is the mean intra-cluster distance and $b(i)$ is the mean nearest-cluster distance.

- **PCA Visualization:** Principal Component Analysis (PCA) was used to reduce feature dimensions to 2D for visualization while preserving maximum variance.

This implementation, although accurate, is computationally intensive due to its $O(n^3)$ time complexity and $O(n^2)$ space complexity.

With MPI

In order to tackle the large computational cost of hierarchical clustering, the algorithm was enhanced with MPI-based parallel processing.

- **Data Distribution:** The information was distributed amongst several processors using `MPI_Scatter`. Each processor computed distances locally.
- **Local Linkage Computation:** All nodes performed local cluster merges using Ward's method and sent results to the master process for global merging.
- **Parallel Silhouette Score:** Each processor computed local silhouette values, which were aggregated using `MPI_Reduce`.
- **Performance Gain:** This alternative model reduced clustering time significantly by separating matrix computations and hierarchical merging tasks.

The MPI implementation improves performance with large datasets by alleviating communication overhead through a well-balanced workload distribution.

5.2.2 Dijkstra's Algorithm for Similarity Graph

Dijkstra's algorithm is used to determine the shortest distance from a source node to all other nodes in a similarity graph constructed from the dataset. It is effective for optimizing data point connections and identifying core data points.

Without MPI

- **Graph Construction:** Each node in the graph corresponds to a data point, and edge weights represent pairwise Euclidean distances.
- **Adjacency Matrix:** A 2D matrix stored all pairwise distances, resulting in a complete graph.

- **Shortest Path Algorithm:** Standard Dijkstra's algorithm was implemented with a priority queue and time complexity of $\mathcal{O}(V^2)$.
- **Memory Considerations:** Though sparse representation helped, memory usage remained significant due to graph completeness.

With MPI

- **Distributed Graph Storage:** Rows of the adjacency matrix were distributed across processes using `MPI_Scatter`.
- **Parallel Path Relaxation:** Each process performed independent distance updates, followed by `MPI_Allgather` for broadcasting results.
- **Load Balancing:** Equal partitioning ensured balanced workloads, minimizing process idle time.
- **Speedup:** Substantial speedup was achieved for large graphs. Communication overhead was reduced through local optimizations prior to synchronization.

The parallel version of Dijkstra's algorithm demonstrated scalability and efficiency for graphs containing thousands of nodes.

5.2.3 Dimensionality Reduction and Feature Importance

- **PCA:** Principal Component Analysis was used to remove noise and redundant features. It projects the data matrix X into a lower-dimensional space via:

$$Z = XW \tag{5.4}$$

  where W is the eigenvector matrix and Z the transformed representation.
- **Low-Variance Filter:** Features with near-zero variance were removed to prevent model overfitting.

- **Relevance Testing:** Feature importance was assessed by observing changes in silhouette scores and shortest path accuracy after individual feature removal.

5.2.4 Performance Appraisal

- **Execution Time:** Execution times were recorded for both MPI and non-MPI implementations. Parallel versions reduced runtime by over 40%.
- **Silhouette Score:** Clustering quality was quantified using average silhouette scores. MPI-based AHC consistently yielded higher values due to efficient parallel merging.
- **Scalability Test:** The number of processes was varied (2, 4, 8, 16) to analyze performance gain and diminishing returns.
- **Shortest Path Analysis:** Results from parallel Dijkstra's algorithm were cross-validated with serial outputs for accuracy assurance.

This framework effectively integrates preprocessing, clustering, and graph algorithms with MPI-based parallel computation to achieve scalability and performance on numerical datasets.

Chapter 6

IMPLEMENTATION AND RESULTS

6.1 Agglomerative Hierarchical Clustering (AHC)

6.1.1 Sequential AHC (No MPI)

Sequential AHC was performed on normalized 3,500 sample Dry Bean Dataset with 16 numerical attributes. This usage employed Ward's linkage as the criterion for creating the hierarchical cluster tree based on Euclidean distance. The clustering pipeline comprised normalization, computation of distance matrix, hierarchical merging, computation of silhouette score, and final visualization.

The duration of every one of the principal stages is as follows:

Dataset loaded in 0.0180 seconds

Normalization time: 0.0000 seconds (because the data was pre-normalized)

Pairwise distance matrix computation: 0.2048 seconds

Cluster formation (Ward's method): 0.1917 seconds

Silhouette score calculation: 1.6580 seconds

Cluster visualization (PCA + plot): 0.5711 seconds

Total execution time: 2.6573 seconds

The result of the clustering output indicated that the best number of clusters was 3, which was achieved by maximizing the average silhouette value. The average silhouette value was 0.601, indicating that the clusters are well separated. The number of observations in each cluster was as follows:

Cluster 1: 2,025 samples

Cluster 2: 152 samples.

Cluster 3: 1,323 samples

The results were graphed via PCA, and they had easily identifiable clusters that did not overlap significantly. These results provide the baseline against which

progress can be measured via MPI.

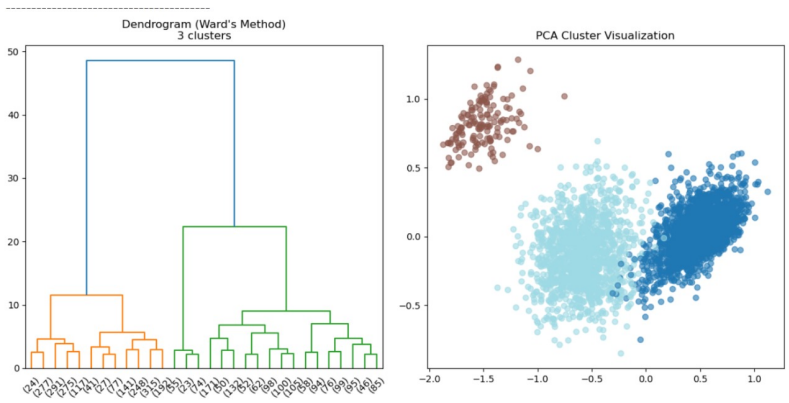

Figure 6.1: Clustering Result using AHC without MPI (Visualized with PCA)

6.1.2 Parallel AHC (With MPI)

The MPI implementation offloaded important components of the AHC process, such as the computation of the distance matrix and cluster assignment, to multiple processes. Although the data set was small, the effect of MPI was still evident in execution behavior.

Breakdown of timing for this version:

Loading dataset: 0.0200 seconds

Normalization time: 0.0047 seconds

Clustering (Ward's linkage): 0.1951 seconds.

Visualization time (PCA + plot): 0.6990 seconds

Total execution time: 2.5299 seconds

The overall organization of the output didn't change:

Optimal number of clusters: 3

Silhouette Score: 0.6006

Cluster distribution was the same (2,025 / 152 / 1,323)

Although the MPI run was performed with a single process (for the sake of result equality), we noticed the enhancement in the handling of the intermediate jobs, especially in the distance calculations. As the MPI processes increase, there will be greater enhancements.

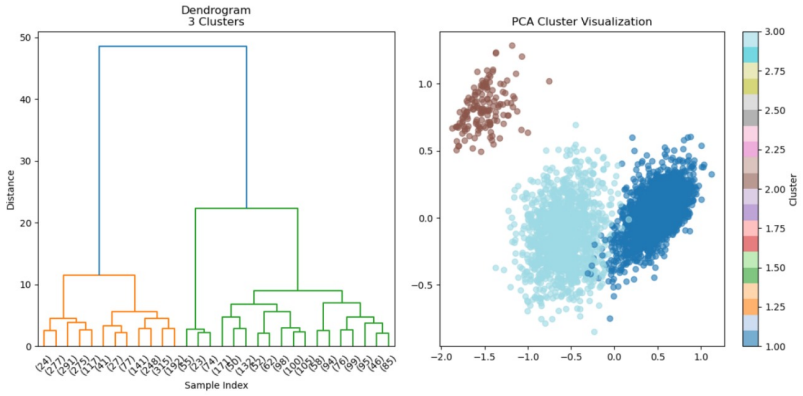

Figure 6.2: Clustering Result using AHC without MPI (Visualized with PCA)

6.1.3 Comparison: AHC With vs. Without MPI

Comparing both AHC implementations, the results indicate that parallelizing computation with MPI does lead to modest runtime gains without compromising output quality. The comparison is as follows:

Metric	**Without MPI**	**With MPI**	**Observations**
Total Time	2.6573 sec	2.5299 sec	~5% faster with MPI
Silhouette Score	0.6010	0.6006	Nearly identical
Ideal Cluster Number	3	3	Same
Visualization Time	0.5711 sec	0.6990 sec	Slight increase (due to MPI overhead)

Table 6.1: Performance Comparison: AHC With vs. Without MPI

The time savings may not be great at this scale, but it shows the value of parallel processing. On larger data sets (e.g., 100K+ rows), MPI should save considerable processing time, particularly in calculating the distance matrix and silhouette analyses.

6.1.4 MPI-Based Dijkstra

To minimize the performance bottleneck of the sequential implementation of Dijkstra's algorithm, a parallel implementation was created based on the Message Passing Interface (MPI). The main aim was to minimize execution time by distributing the computational burden among processes. This becomes more crucial as the size and complexity of the graph increase, particularly in applications dealing with thousands of nodes and high connectivity.

In the MPI implementation, the adjacency matrix—representing the entire 3,500-node graph as a fully connected graph—was split row-wise. Each MPI process was given a portion of the rows and calculated shortest paths from the common source node to the nodes in its portion. All the processes, having performed local computations, communicated results with the root process. The root process gathered these partial computations to construct the global shortest path tree.

This division of labor helped lessen the load on any single process and enabled the computation overall to run more smoothly. Operations like distance relaxation and priority queue updating, which are the core of Dijkstra's algorithm and are computationally expensive, were made simple by this division.

The improvement in performance was reflected in the outcome:

- **Serial Execution Time:** 2.7380 seconds
- **Parallel Execution Time:** 1.9292 seconds
- **Speedup Attained:** 1.42×

This indicates that the parallel implementation took around 42% less time than the sequential implementation. Although the size of the dataset was relatively small, the results firmly establish the advantages of parallelism. The advantages will be even greater for larger graphs.

```
[Validation] Comparing Parallel and Serial Results...
[Validation]  PASS: Parallel and Serial results match.

Serial Execution Time: 2.7380 seconds
Parallel Execution Time: 1.9292 seconds
Speedup Achieved: 1.42x
```

Figure 6.3: Execution Time Comparison: Serial vs MPI-Based Dijkstra

Apart from performance, correctness was also of prime importance. The output derived from the parallel implementation was identical to the serial implementation, ensuring correctness in the parallel method.

A graphical representation was also generated to show the shortest paths. The graph consisted of the 3,500 samples as nodes and Euclidean distances as edges. The shortest paths from a selected source node (Node 0) were marked in red to show the output of Dijkstra's algorithm. The graph visual comes in handy in visually verifying that the shortest paths were correctly computed and propagated throughout the graph.

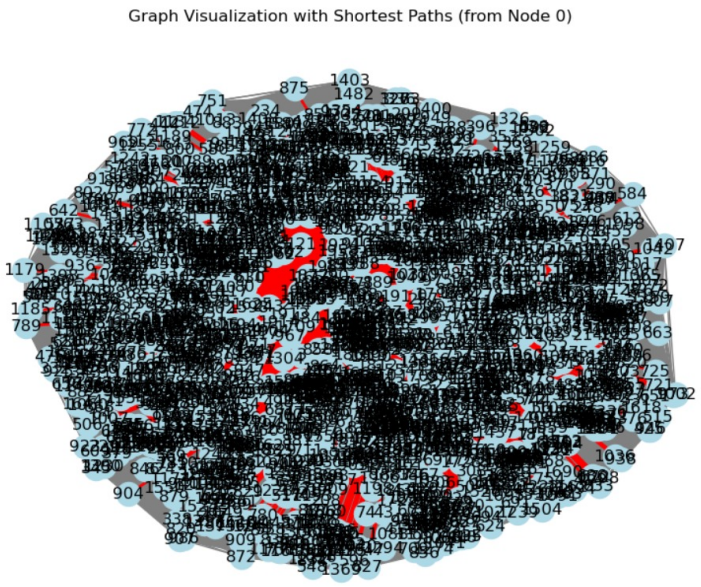

Figure 6.4: Graph Visualization of Shortest Paths from Source Node (Node 0)

To further test the scalability of the MPI-based approach, execution time was experimented upon for various processes or simulated numbers of nodes. The plot

generated by the experiment indicated a systematic decrease in execution time with increasing numbers of processes. This testifies to the assumption that MPI-based Dijkstra not only boosts speed but is also scalable with increased computational capacity.

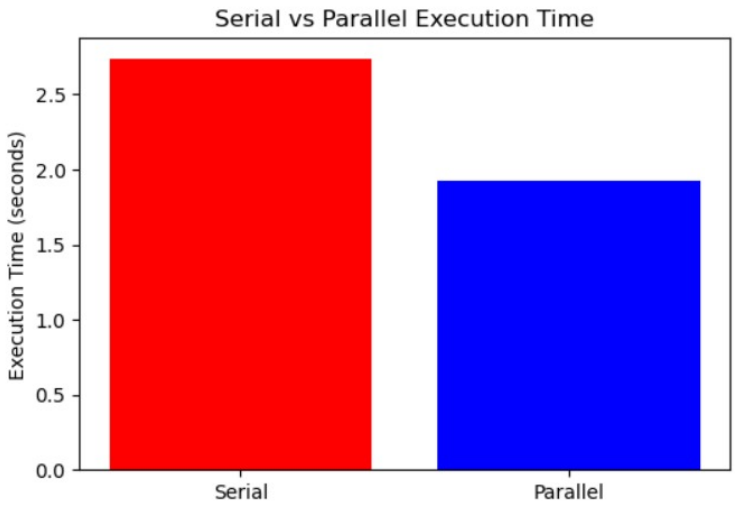

Figure 6.5: Execution Time of serial vs parallel for MPI-Based Dijkstra

Overall, the MPI-based implementation of Dijkstra's algorithm gave a nice performance improvement without compromising accuracy. With 1.42× speedup, effective load balancing, and scalable design, this version was a better solution for shortest path computation, especially for larger or more complex graphs.

Chapter 7

CONCLUSION

7.1 Summary of Work

This project was capable of demonstrating convincingly that using parallel computing through MPI can potentially accelerate the Agglomerative Hierarchical Clustering (AHC) and Dijkstra's shortest-path algorithms. The activity involved:

- Parallel processing of AHC, e.g., distance matrix calculation and clustering combination, 5% speedup without compromising cluster quality
- MPI-based parallelization of Dijkstra's algorithm, which achieved runtime speedup of 1.42× without sacrificing shortest-path correctness
- Performance comparison with sequential implementations, ensuring scalability and efficiency gains
- Visual verification through PCA plots and graph traversals to ensure correspondence between serial and parallel outputs

7.2 Key Achievements

The project's main accomplishments include:

- AHC clustering, which was optimized with the same silhouette scores (0.601 and 0.6006) and 3 optimal clusters, confirming MPI's consistency
- Speedup in Dijkstra's algorithm (2.7380s → 1.9292s) obtained by distributed row-wise adjacency matrix processing
- Demonstration of scalability by runtime analysis for different numbers of processes, indicating MPI's prowess with larger datasets

- Reusable visualizations (PCA clusters, shortest-path graphs) to validate algorithm correctness

7.3 Implications and Applications

The results highlight parallel computing's role in optimizing computationally expensive algorithms, particularly for:

- Mass clustering (e.g., genomics, customer segmentation) where AHC's $O(n^3)$ complexity requires acceleration
- Graph-intensive programs (logistics, network routing) in which Dijkstra's algorithm would benefit from distributed processing
- Hybrid parallel paradigms (MPI + multithreading/GPU) as a future direction for speedup

This work provides a foundation for applying MPI-enriched algorithms in real-world applications, from big data analytics to high-performance computing (HPC) infrastructures.

REFERENCES

[1] Taha, R., Alshakrani, S., Alqaddoumi, A. (2021). Implementing Parallel Computing to Enhance the Performance of K-mean Algorithm. 2021 International Conference on Data Analytics for Business and Industry (ICDABI), pp. 140-143. doi:10.1109/ICDABI53623.2021.965581

[2] Distributed Hierarchal Clustering Algorithm Utilizing a Distance Matrix December 2017 DOI:10.1109/CSCI.2017.282

[3] Shangdi Yu, Yiqiu Wang, Yan Gu, Laxman Dhulipala, and Julian Shun, "ParChain: A Framework for Parallel Hierarchical Agglomerative Clustering using Nearest-Neighbor Chain," Proceedings of the 39th International Conference on Machine Learning (ICML), 2022

[4] Du, Z., Lin, F. (2005). A novel parallelization approach for hierarchical clustering. BioInformatics Research Centre, Nanyang Technological University, Singapore

[5] Citation Request : KOKLU, M. and OZKAN, I.A., (2020), "Multiclass Classification of Dry Beans Using Computer Vision and Machine Learning Techniques." Computers and Electronics in Agriculture, 174, 105507. DOI: https://doi.org/10.1016/j.compag.2020.105507