Parallelizing K-means and BFS with OpenMP and MPI: Performance Analysis

Aheed Tahir Ali
21K-4517

Maaz Imam
21K-3218

Muhammad Usman
21K-4890

December 1, 2024

Abstract

Parallel computing plays a crucial role in addressing the computational demands of modern data-intensive applications. This project explores the parallelization of two fundamental algorithms, K-means clustering and Breadth-First Search (BFS), using both OpenMP and Message Passing Interface (MPI). The objective is to analyze the impact of parallelization on performance metrics such as computation time, communication time, and scalability across varying data sizes.

The project involves the implementation of parallel K-means clustering using OpenMP and MPI, as well as parallel BFS using the same frameworks. We examine the execution times for different data sizes, evaluating the efficiency of parallelization in terms of both computation and communication overhead. Additionally, we delve into the time complexities of the parallel algorithms and discuss their scalability in handling larger datasets.

This report presents a comprehensive analysis of the parallelized K-means and BFS algorithms, providing insights into the benefits and challenges of parallel computing in the context of these widely used data processing techniques.

1 Introduction

In today's data-driven world, the amount of information we deal with is growing rapidly, and so is the need for powerful computers. The usual way of doing things, where we handle data step by step, just can't keep up with the demands of tasks like machine learning and complex simulations. This is where parallel computing comes in.

We take a big problem, break it into smaller parts, and let different processors or computers work on them at the same time. This makes things much faster, especially for tasks that can be divided up easily.

This project is all about using this idea to improve two important algorithms: K-means clustering and Breadth-First Search (BFS). We're going to use two common ways of programming in parallel, called OpenMP and Message Passing Interface (MPI), to make these algorithms work better for handling large amounts of data.

Our main goal is to make these algorithms much more efficient and able to handle way bigger datasets than before, while still giving accurate and quick results. Imagine being able to analyze millions of customer records in just a few minutes to find out buying patterns or mapping complex social networks quickly.

By using parallel computing, we're trying to do more with data analysis than ever before. This project is a step towards getting quicker and deeper insights, and it opens doors to finding new and important things in different areas.

2 Background

2.1 K-means Clustering

K-means clustering is a widely used unsupervised machine learning algorithm that partitions a dataset into k distinct, non-overlapping subsets or clusters. The algorithm iteratively assigns data points to clusters based on their similarity to the cluster's centroid, aiming to minimize the sum of squared distances between data points and their assigned centroid.

2.2 Breadth-First Search (BFS)

Breadth-First Search is a graph traversal algorithm used to explore and analyze the structure of a graph. It systematically visits all the vertices of a graph in breadth-first order, level by level, starting from a chosen source vertex. BFS is fundamental for various applications, including pathfinding and network analysis.

In this project, we explore the parallelization of these algorithms using OpenMP and MPI, investigating the impact on performance metrics such as computation time, communication time, and scalability across different data sizes.

3 Implementation and Execution

For each algorithm, K-means clustering and BFS, we conducted separate implementations and executions using OpenMP and MPI to analyze the impact of parallelization techniques on performance.

3.1 K-means Clustering

3.1.1 OpenMP Implementation

The K-means clustering algorithm was parallelized using OpenMP directives to exploit shared-memory parallelism. Specifically, the computation of cluster centroids and assignment of data points to clusters were parallelized to enhance overall efficiency.

3.1.2 MPI Implementation

The MPI implementation of K-means clustering involved distributing data and computation across multiple nodes. Each node processed a subset of the data, and MPI communication was employed to exchange cluster information iteratively until convergence.

3.2 Breadth-First Search (BFS)

3.2.1 OpenMP Implementation

For BFS, OpenMP directives were utilized to parallelize the traversal of the graph. The algorithm was divided into sections that could be executed concurrently, allowing for efficient exploration of the graph's structure.

3.2.2 MPI Implementation

MPI was employed to parallelize BFS in a distributed-memory environment. Each node in the cluster processed a portion of the graph, and MPI communication facilitated the sharing of frontier information, ensuring a synchronized exploration of the graph.

3.3 Execution

The implementations were executed on a cluster of nodes, and we recorded performance metrics for varying data sizes. Execution times, including computation and communication times, were measured to analyze the efficiency of parallelization. The following sections detail the results obtained and the insights gained from the experiments.

4 Experimental Setup

We tested the algorithms on a computer with an Intel Core i7 11th Gen processor allotted with 4 cores, 8GB of RAM, and running on a windows system. The development environment included Visual Studio Code as the integrated development environment, and MPI SDKs for C/C++ along with OpenMP were utilized for parallelization.

5 Analysis of K-means Clustering Implementations

5.1 Sequential Implementation

The sequential implementation of the K-means clustering algorithm was executed on the laptop's single processor core. For smaller datasets (100, 2000 data points), the algorithm demonstrated efficient performance, with execution times ranging from 0.001 to 0.003. However, as the dataset size increased to 100, 000, 200, 00 and 1,000, 000 data points, the algorithm took longer times starting off with 20 seconds and moving all the way up to ~25 minutes. This behavior indicates there's a lot of processing in handling larger dataset efficiently in a sequential setting.

Code	Data Points	Dimensions	Clusters	Time (in seconds)
Sequential	100	3	3	0.001
Sequential	2000	3	4	0.003
Sequential	100 000	5	7	20.111
Sequential	200 000	10	20	98.693
Sequential	1000 000	10	20	1491.921

5.2 OpenMP Implementation

Parallelizing K-means clustering using OpenMP introduced a notable improvement in performance. The algorithm demonstrated a clear reduction in execution times for all dataset sizes compared to the sequential implementation. Particularly for larger datasets (100,000, 200,000 and 1,000,000 data points), the OpenMP implementation showcased substantial speedup, with execution times ranging from 12.921 to 21.5 minutes. The scalability of the OpenMP implementation is evident, as it efficiently handles increased computational loads through parallelization. However, the execution times for the smaller datasets (100 and 2,000) increased which means OpenMP works best with larger datasets.

Code	Data Points	Dimensions	Clusters	Time (in seconds)
OpenMP	100	3	3	0.065
OpenMP	2000	3	4	0.0505
OpenMP	100 000	5	7	12.921
OpenMP	200 000	10	20	79.252
OpenMP	1000 000	10	20	1291.901

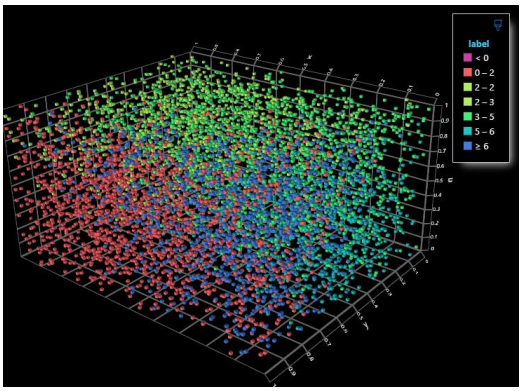

Figure 1: OpenMP Program Analyzing 100,000 Points with 5 Dimensions, Revealing the Intricacies of 7 Clusters in 3D Space.

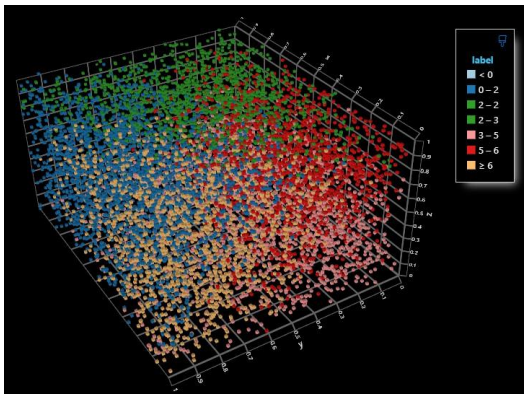

Figure 2: OpenMP Program Analyzing 200,000 Points with 10 Dimensions, Revealing the Intricacies of 20 Clusters in 3D Space.

5.3 MPI Implementation

In the MPI implementation, the K-means clustering algorithm was parallelized across multiple processes, providing insights into the performance in a distributed-memory environment. The communication and computation times, along with the total execution time, were recorded for various dataset sizes and process configurations.

For smaller datasets (100 and 2,000 data points), the communication time was minimal, indicating efficient coordination among processes. However, as the dataset size increased, particularly for 100,000, 200,000 and 1,000,000 data points, the communication time became more prominent. This suggests that the overhead of inter-process communication becomes a critical factor in the overall performance for larger datasets.

Code	Data Points	Dimensions	Clusters	Comm. Time (sec)	Comp. Time (sec)	Total Time (sec)
MPI	100	3	3	0.000 02	0.000 8	0.001 43
MPI	2 000	3	4	0.066	0.000 3	0.076 3
MPI	100 000	5	7	1.320	0.027	1.647
MPI	200 000	10	20	6.701	0.032	7.502
MPI	1 000 000	10	20	34.591	3.030	41.322

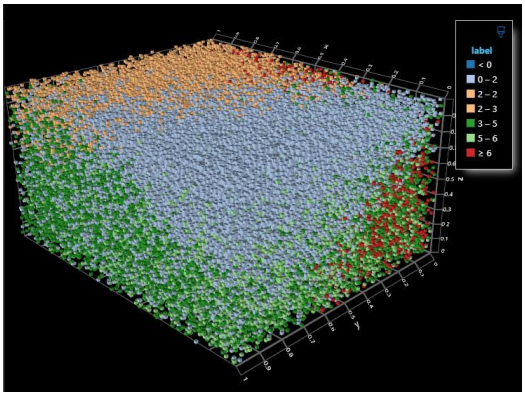

Figure 3: MPI Program Analyzing 100,000 Points with 5 Dimensions, Visualizing the Complexity of 7 Clusters in 3D Space.

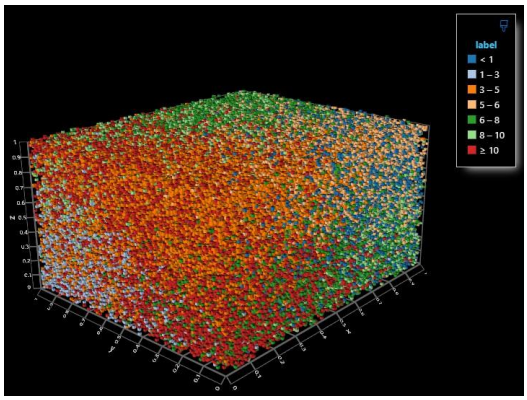

Figure 4: MPI Program Analyzing 1,000,000 Points with 10 Dimensions, Visualizing the Complexity of 20 Clusters in 3D Space.

5.4 Summary

The parallelization of the K-means clustering algorithm using OpenMP and MPI significantly improves its performance and scalability. OpenMP, leveraging shared-memory parallelism, demonstrates efficient speedup, particularly for larger datasets. On the other hand, MPI, designed for distributed-memory environments, exhibits scalability but introduces communication overhead, which becomes more prominent with increasing dataset sizes.

The choice between OpenMP and MPI depends on the specific characteristics of the dataset and the available computing infrastructure, balancing shared-memory and distributed-memory considerations.

6 Analysis of Breadth-First Search Implementations

6.1 Sequential Implementation

The sequential implementation of the BFS algorithm was executed on the laptop's single processor core. It was noted that when employing a graph of 100,000 edges we got a fairly large time of 31.927 seconds however when the data size was increased to 200,000 and 1,000,000 the system crashed, and no computation time was received this is due to the significant size of the input which was unable to be computed on our system. This test serves as an example of the limitation of sequential programming.

Edges	Computation Time (sec)
100 000	31.927
200 000	NaN
1 000 000	NaN

6.2 OpenMP Implementation

The OpenMP implementation of the Breadth-First Search (BFS) algorithm was executed on the specified laptop configuration, utilizing different numbers of threads. The results reveal insights into the performance of the parallel BFS algorithm under shared-memory parallelization.

Edges	Threads	Computation Time (sec)
100 000	4	0.000 326
200 000	4	0.000 173
1 000 000	4	0.000 146

The OpenMP implementation showcases efficient parallelization for BFS. Not only did the computation times significantly decrease but also that the system was now able to compute the larger data sizes of 200,000 and 1,000,000 which the sequential program was unable to. This behavior aligns with the shared-memory parallelism provided by OpenMP, allowing for effective utilization of multiple threads and resulting in improved performance.

6.3 MPI Implementation

The MPI implementation of the BFS algorithm was conducted with varying numbers of processes, exploring the impact of distributed-memory parallelization on performance.

Edges	Processes	Computation Time (sec)
100 000	4	0.000 452
100 000	8	0.000 322
200 000	4	0.001 235
200 000	8	0.001 554
1 000 000	4	0.012 322
1 000 000	8	0.025 001

The MPI implementation demonstrates the impact of distributed-memory parallelization on BFS. Smaller datasets show efficient parallelization with decreasing computation times as the number of processes increases. However, for the larger data set, an increase in the number of processes does not yield a proportional reduction in computation time, suggesting potential overhead in inter-process communication.

6.4 Summary

The parallelization of BFS using OpenMP and MPI provides valuable insights into the algorithm's performance characteristics. OpenMP, leveraging shared-memory parallelism, demonstrates efficient parallelization with decreasing computation times as the number of threads increases. On the other hand, MPI, designed for distributed-memory environments, shows effective parallelization for smaller datasets, but the overhead of inter-process communication becomes more apparent for larger datasets.

The choice between OpenMP and MPI for BFS depends on the dataset size and the balance between shared-memory and distributed-memory considerations.

7 Conclusion

In this project, we explored the parallelization of two fundamental algorithms, K-means clustering and Breadth-First Search (BFS), using OpenMP and MPI. The experiments were conducted on a laptop with an Intel Core i7-11 Gen processor, 8 GB RAM, running Windows, and implemented using Visual Studio Code with MPI SDKs for C/C++ and OpenMP.

7.1 K-means Clustering

The parallelization of K-means clustering using OpenMP showcased significant improvements in performance, with clear speedup observed across various dataset sizes. The scalability of OpenMP was evident, particularly for larger datasets, where execution times decreased substantially compared to the sequential implementation.

The MPI implementation of K-means clustering in a distributed-memory environment demonstrated scalability, but the trade-off between computation and communication times became prominent for larger datasets. The inter-process communication overhead was more noticeable as the dataset size increased, impacting the overall efficiency of the algorithm.

7.2 Breadth-First Search

For Breadth-First Search, the OpenMP implementation demonstrated efficient parallelization with decreasing computation times as the number of threads increased. This aligns with the shared-memory parallelism of OpenMP, allowing effective utilization of multiple threads for improved performance.

The MPI implementation of BFS showcased effective parallelization for smaller datasets, but the overhead of inter-process communication became more apparent for larger datasets. While smaller datasets exhibited efficient parallelization with decreasing computation times as the number of processes increased, this trend did not hold consistently for larger datasets.

7.3 Overall Insights

The project provided valuable insights into the strengths and limitations of OpenMP and MPI for parallelizing diverse algorithms. OpenMP demonstrated robust performance in shared-memory environments, showcasing scalability and efficiency, particularly for computation-intensive tasks like K-means clustering. On the other hand, MPI exhibited effective parallelization in distributed-memory settings but introduced communication overhead that became more pronounced with larger datasets.

In conclusion, the choice between OpenMP and MPI depends on the nature of the algorithm, dataset size, and the characteristics of the computing environment. The project highlighted the importance of considering both computation and communication aspects when parallelizing algorithms for optimal performance in different parallel computing scenarios.

References

1. https://github.com/rexdwyer/MPI-K-means-clustering
2. https://www.youtube.com/watch?v=T_BVqSya1Is
3. https://marketplace.visualstudio.com/items?itemName=msrvida.vscode-sanddance