CS359 - Parallel Computing Project Report

Parallel Algorithm Design and implementation of K-Means Clustering

- Arnav Nirmal Jain - 220002018
- Bharat Kaurav - 220001017

INTRODUCTION

The project, titled **"Parallel Algorithm Design and Implementation of K-Means Clustering"**, explores the optimization of the K-Means clustering algorithm using parallel processing techniques to enhance computational efficiency. K-Means clustering is a widely used unsupervised machine learning algorithm for partitioning datasets into distinct clusters, but it is computationally expensive when applied to large datasets or high-dimensional data. Traditional implementations often struggle with scalability, as they require repeated calculations of distances and cluster assignments for all data points.

The goal of this project was to design and implement an optimized K-means clustering algorithm tailored for CPU architectures. By leveraging parallel processing, we aimed to significantly reduce computation time without compromising clustering accuracy. This optimization enhances the algorithm's suitability for large-scale applications, making it practical for real-world scenarios where computational resources are limited to CPUs.

This report details the findings and results of our work, beginning with an overview of the sequential K-Means clustering algorithm. It then outlines the design and implementation of our parallelized approach, highlighting the challenges of synchronization, load balancing, and the incorporation of data parallelism. Finally, we present our implementation's performance analysis, demonstrating parallelization's effectiveness in achieving faster and more scalable clustering.

PROBLEM STATEMENT

K-Means is an iterative algorithm that partitions a dataset into *K* clusters, where each data point belongs to exactly one cluster. The objective is to minimize the sum of squared distances between data points and their corresponding cluster centroids, ensuring that the resulting clusters are as homogeneous as possible. This algorithm is widely used across various industries, including market segmentation, image compression, and anomaly detection.

The sequential implementation of K-Means clustering exhibits a time complexity of ***O(N×K×I)***, where *N* is the number of data points, *K* is the number of clusters, and *I* is the number of iterations required for convergence. For large datasets, this computational cost can become a bottleneck.

Recognizing this, we identified parallelism as a promising approach to significantly reduce the convergence time of the K-Means algorithm. By leveraging parallel processing, the workload of assigning data points to clusters can be distributed across multiple threads or processes. Consequently, the time complexity of the parallel implementation is reduced to approximately $O(\frac{N}{P} \times K \times I)$, where *P* represents the number of processing units. This demonstrates that the parallel algorithm scales well with increasing computational resources.

The key challenges/subproblems addressed in this project include:

1. Developing an effective strategy to parallelize the K-Means algorithm while maintaining its accuracy.
2. Managing synchronization between processes to ensure consistent updates to cluster centroids.
3. Verifying the correctness of the parallel algorithm by comparing the clusters produced with those generated by the sequential implementation.
4. Achieving a notable reduction in the overall runtime of the parallel algorithm compared to the sequential version.

Through our efforts, we successfully demonstrated the potential of parallel computing to enhance the efficiency of K-Means clustering, address the outlined challenges, and significantly improve computational performance.

PROBLEM APPROACH

Sequential Clustering Algorithm

The traditional sequential K-Means algorithm is a clustering algorithm with K clusters and a dataset of N data points, where $X_i \in R^d$ represents the i-th data point in d-dimensional space, and $C_k \in R^d$ represents the centroid of the k-th cluster.

Initialization

1. Select K initial centroids C_1, C_2, ..., C_K randomly from the dataset.
2. Set a maximum number of iterations T and a convergence threshold ϵ for the centroids' movement.

Cluster Assignment (E-Step)

For each data point X_i (where i=1,2...,N):

1. Calculate the Euclidean distance between X_i and each centroid C_k, where k=1,2,...,K

$$d_{ik} = // X_i - C_k \|^2$$

2. Assign X_i to the nearest cluster a_i based on the minimum distance:

$$a_i = \text{arg min}_k d_{ik}$$

where a_i is the cluster index to which data point X_i is assigned

Centroid Update (M-Step)

For each cluster k (where k=1,2,...,K):

1. Identify the set S_k of all data points currently assigned to cluster k.
2. Update the centroid C_k by calculating the mean of all points in S_k:

$$C_k = \frac{1}{|S_k|} \sum_{i \in S_k} X_i$$

where $|S_k|$ is the number of points in S_k. If S_k is empty (i.e., no points are assigned to cluster k), reinitialize C_k to a random point.

Iterative Process

1. Repeat the E-Step and M-Step iteratively.
2. Stop if either:
    a. The change in centroids' positions (measured as the maximum Euclidean distance between old and new centroids) is less than a specified threshold ϵ, indicating convergence
    b. The number of iterations reaches T, the pre-set maximum.

Final Output

Once convergence is reached, the algorithm outputs:

1. The final centroids $C_1, C_2, \ldots, C_K$.
2. The cluster assignments $a_1, a_2, \ldots, a_N$ for each data point.
3. The sum of squared distances (inertia) as a measure of clustering quality

While effective, this approach is computationally expensive for large datasets, as the distance calculation between every data point and every centroid is performed in each iteration. This motivated us to explore parallelization to improve efficiency.

Why Parallelism Will Work For K-Means Clustering

Analyzing the K-Means clustering algorithm reveals several areas where parallelism can be applied:

1. Distance Calculations: The calculation of the distance between each data point and each centroid is independent for all data points. This independence provides an excellent opportunity for parallelization.

2. Cluster Assignment: The process of assigning each data point to the nearest centroid is also independent across data points.

However, there is a critical bottleneck in synchronization. After each iteration, all threads must communicate to update the global centroid values. If multiple threads attempt to update the same centroid concurrently, it could lead to corrupted values or false sharing. Proper synchronization mechanisms are necessary to avoid these issues and ensure consistency in the updated centroids.

The Implemented Parallel Clustering Algorithm:

We adopted a *data-parallelism* approach to implement parallelism in the K-means algorithm. Specifically, we divided the N data points evenly across the available threads. Any remaining data points will be allocated to the final thread if an even split is not feasible. Each thread independently handled its assigned points, calculating the closest centroid for each point and updating the point's cluster assignment. The main features of our approach included:

- **Initialization**: We selected the first K data points as initial centroids, setting a starting point for the clustering process.
- **Data Parallelism**: Each thread is assigned approximately $\frac{N}{number\ of\ threads}$ points, ensuring an efficient distribution of workload.

- **Thread Function Execution**: Each thread is iteratively assigned data points to clusters over a maximum of 100 iterations. At each step, the thread updated the global (shared) cluster centroids based on its assigned points. This iterative data-parallel approach ensured that cluster centroids were progressively refined.
- **Stopping Condition**: After each iteration, the L2-norm is computed for every cluster centroid by comparing its coordinates against those from the previous iteration. If the combined difference across all centroids (delta) falls below a threshold of 10^{-6}, all threads are terminated, signaling the end of the clustering process.

Precautions Taken for Efficient and Accurate Parallelization

To avoid issues related to concurrent data access and to optimize performance, we incorporated several synchronization and optimization strategies:

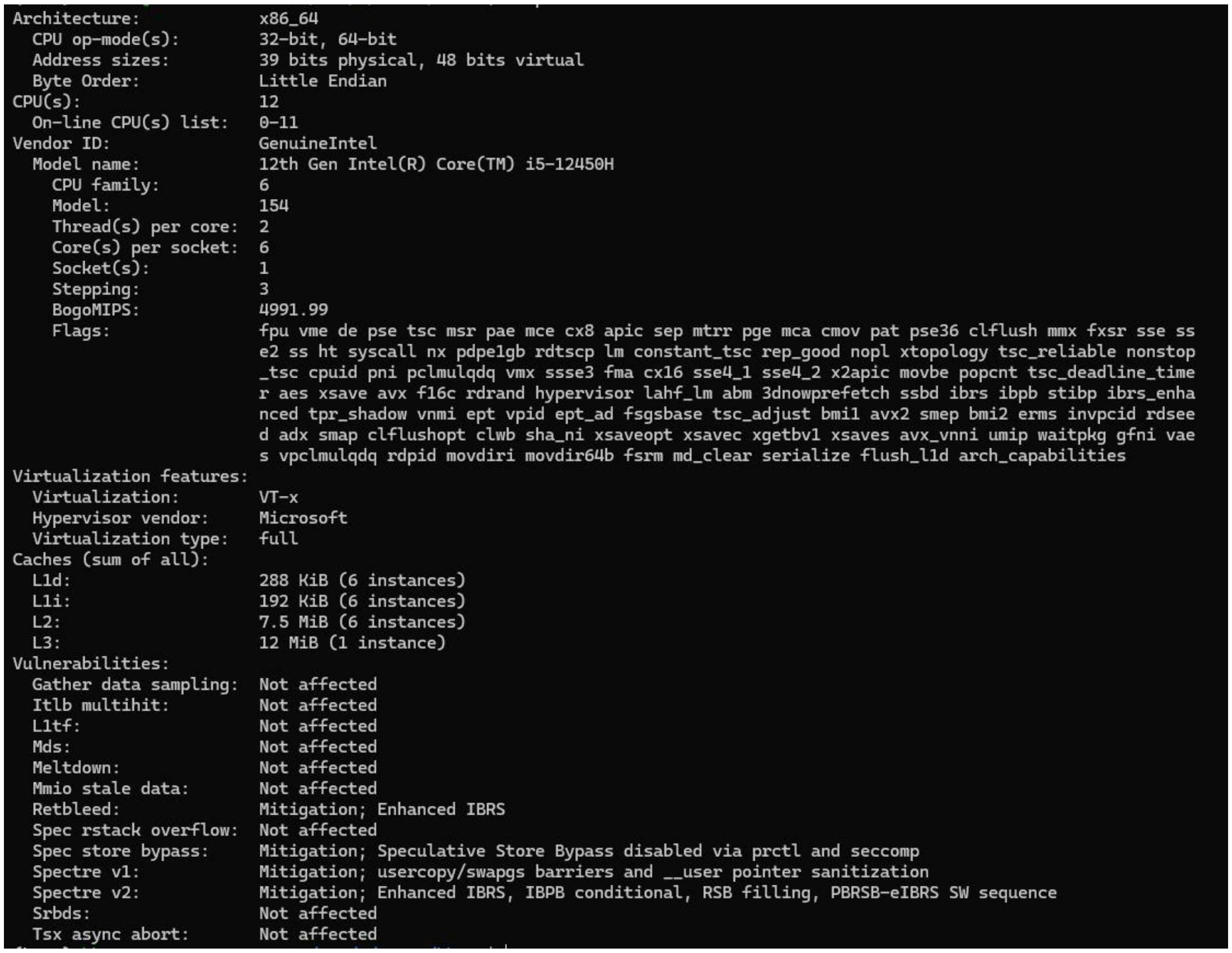
- **Mutual Exclusion and Critical Sections**: We implemented thread synchronization to ensure that the updates to the shared cluster centroid array occur without conflict. Using the `#pragma omp critical` construct, we prevented data races by only allowing one thread at a time to modify the centroid array, ensuring the integrity of centroid values.
- **Barrier Synchronization**: After each iteration, all threads synchronized at specific points to guarantee that they accessed the most recent centroid values. This was achieved using the `#pragma omp barrier` construct, placed after centroids and delta calculations. The first barrier synchronized centroid updates, and the second barrier ensured that the delta values, which determine convergence, are consistently viewed across all threads.
- **False Sharing Minimization**: To minimize false sharing, we carefully managed the shared variable updates, particularly for the centroids array. By using local variables where possible (e.g., storing intermediate delta values in a temporary variable), we limited the risk of cache invalidation, optimizing memory access efficiency.
- **Load Balancing Considerations**: Given the SPMD (Single Program, Multiple Data) structure, explicit load balancing was generally unnecessary. However, the last thread may handle a slightly larger workload if there are extra points that don't evenly divide among threads. This slight imbalance was acceptable within our parallelization strategy, as the workload difference remained minor.

Dataset Information

For our project, we used multiple datasets that we obtained online, which contained approximately 10K, 50K, 100K, 200K, 400K, 500K, 600K, 800K, and 1M data points represented in three-dimensional space. This larger, higher-dimensional dataset allowed us to assess the performance and efficiency of our parallelized k-means implementation.

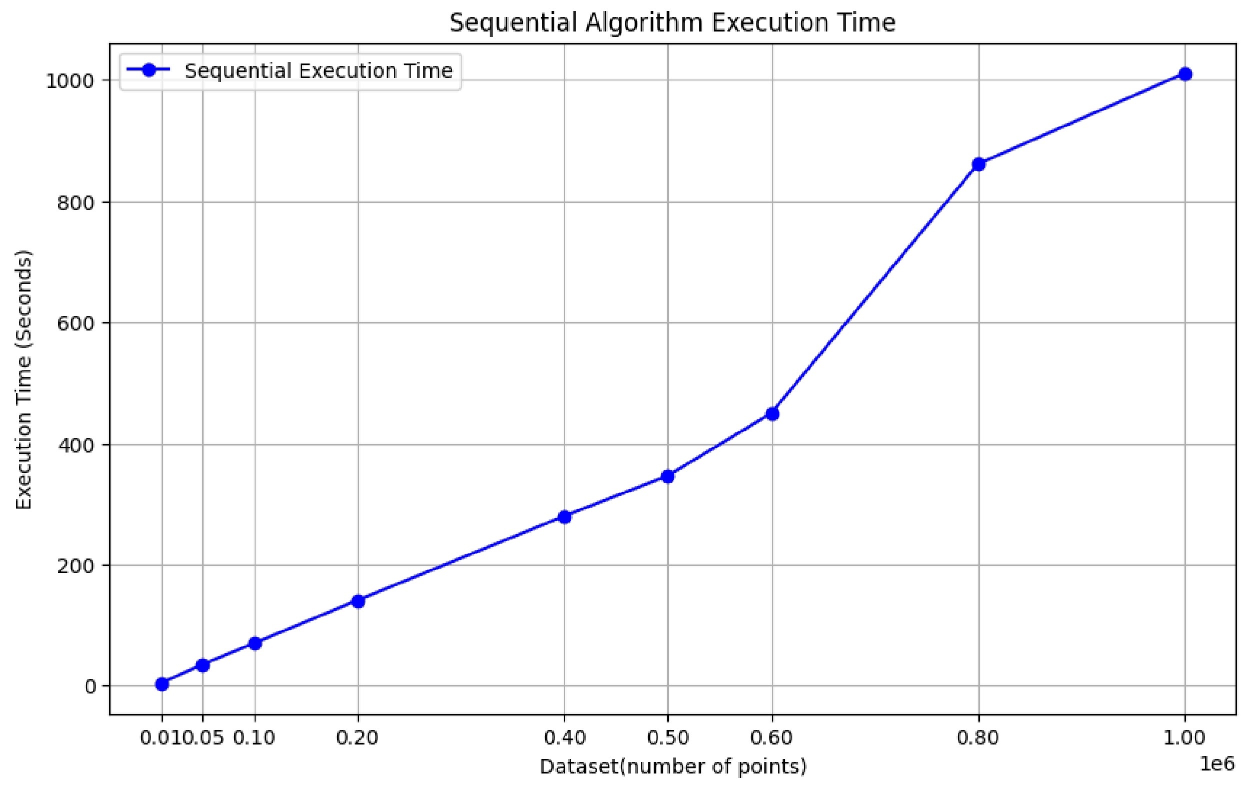
RESULT ANALYSIS

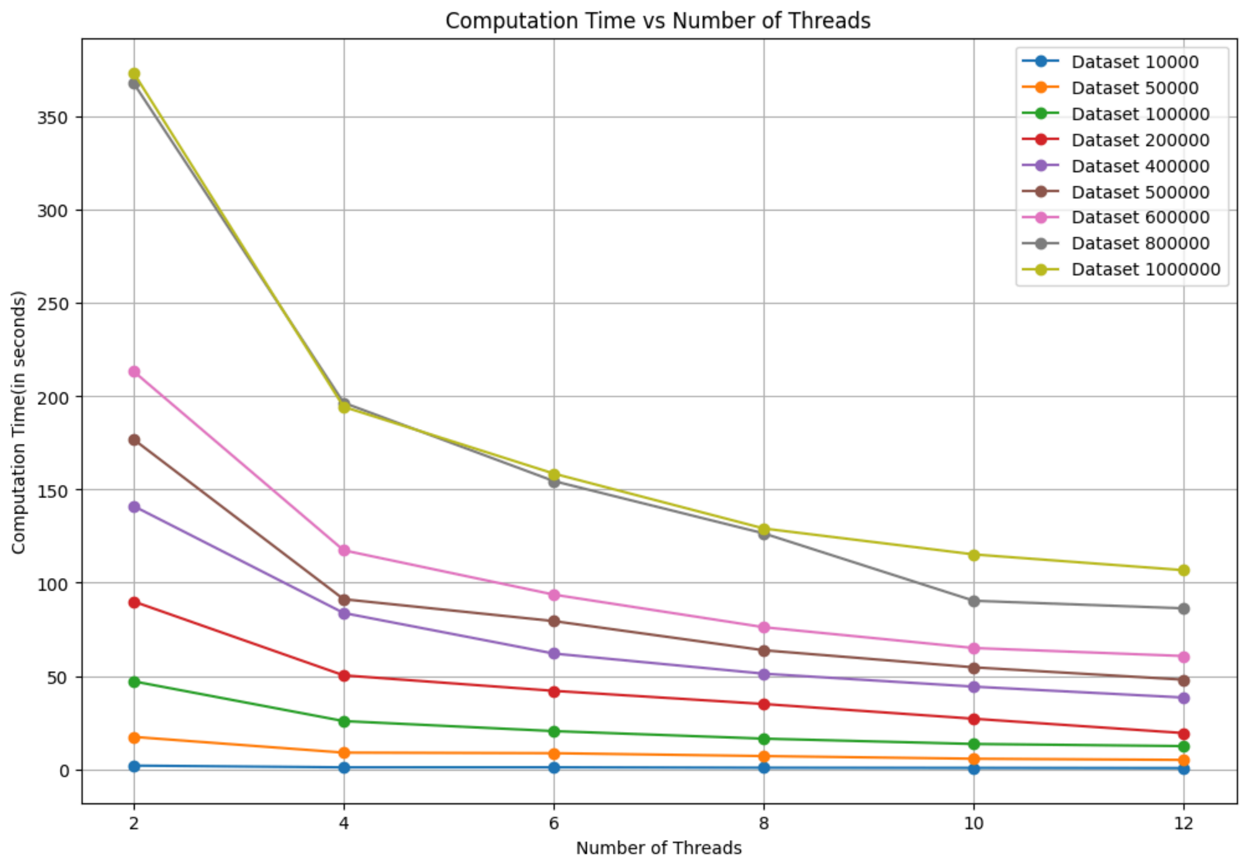
The following is the hardware description of the device we performed all our implementations on.



```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               GenuineIntel
  Model name:            12th Gen Intel(R) Core(TM) i5-12450H
    CPU family:          6
    Model:               154
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            3
    BogoMIPS:            4991.99
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse ss
                         e2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology tsc_reliable nonstop
                         _tsc cpuid pni pclmulqdq vmx ssse3 fma cx16 sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_time
                         r aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp ibrs_enha
                         nced tpr_shadow vnmi ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdsee
                         d adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 xsaves avx_vnni umip waitpkg gfni vae
                         s vpclmulqdq rdpid movdiri movdir64b fsrm md_clear serialize flush_l1d arch_capabilities
Virtualization features:
  Virtualization:        VT-x
  Hypervisor vendor:     Microsoft
  Virtualization type:   full
Caches (sum of all):
  L1d:                   288 KiB (6 instances)
  L1i:                   192 KiB (6 instances)
  L2:                    7.5 MiB (6 instances)
  L3:                    12 MiB (1 instance)
Vulnerabilities:
  Gather data sampling:  Not affected
  Itlb multihit:         Not affected
  L1tf:                  Not affected
  Mds:                   Not affected
  Meltdown:              Not affected
  Mmio stale data:       Not affected
  Retbleed:              Mitigation; Enhanced IBRS
  Spec rstack overflow:  Not affected
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled via prctl and seccomp
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:            Mitigation; Enhanced IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS SW sequence
  Srbds:                 Not affected
  Tsx async abort:       Not affected
```

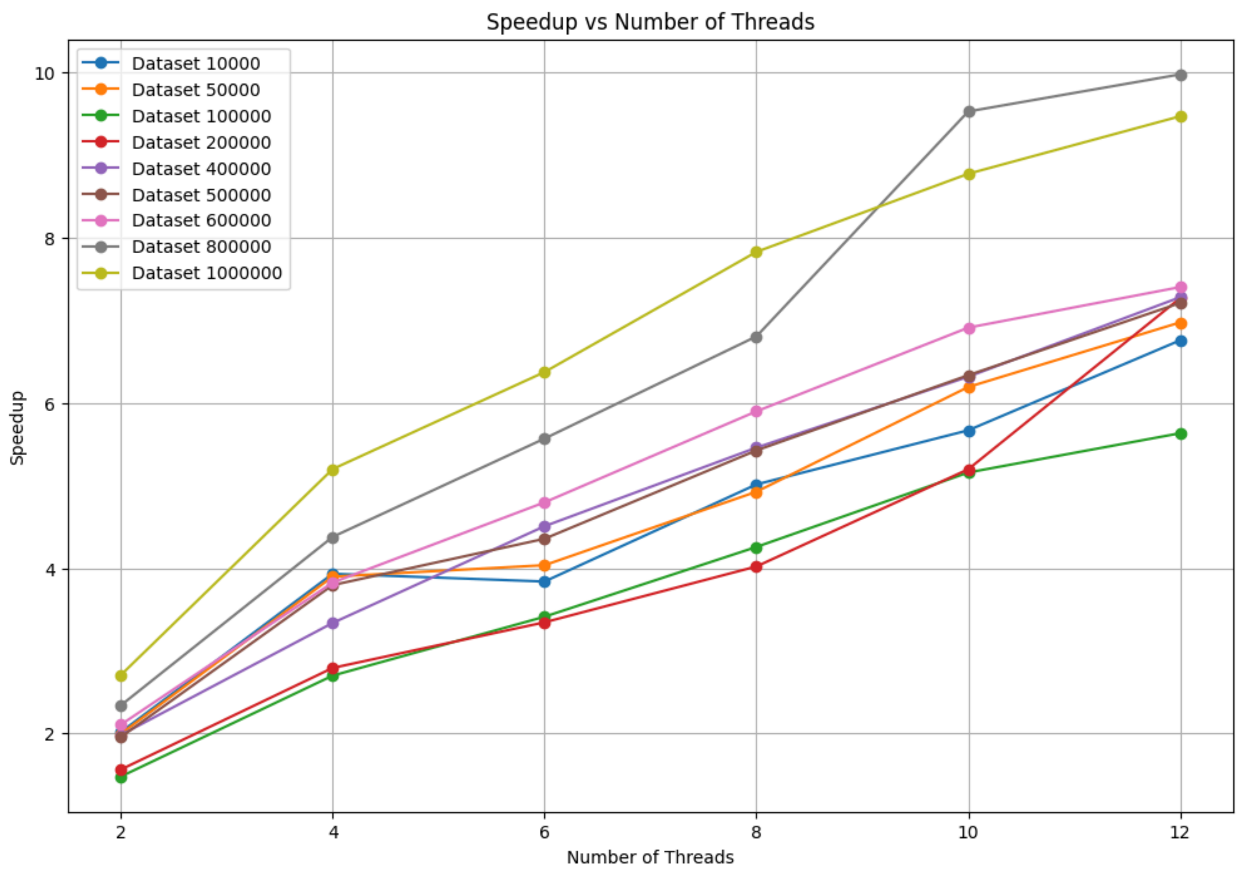
Below are the graphs that display the computation time taken by the sequential algorithm and parallel algorithm on various numbers of threads.





As expected from any multithreaded programming method, the computation time of the algorithm decreases as the number of threads increases so much so that for 12 threads, the overall computation time for various datasets is close enough to zero. The difference between the computation time of different datasets on a fixed number of threads is different because of the unequal distribution of points for each of the respective threads.

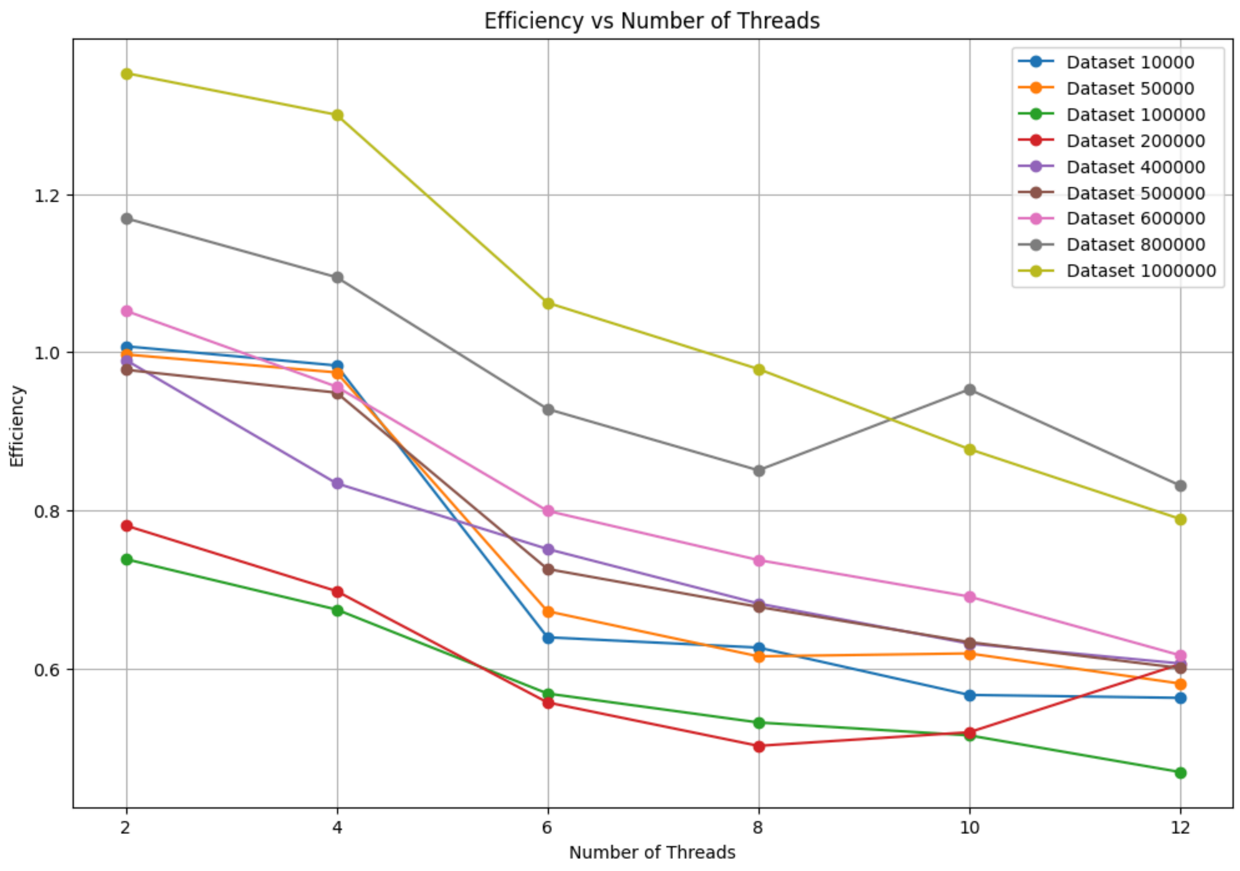
Below is the graph displaying the speedup of our implementation achieved on various datasets with different numbers of threads.



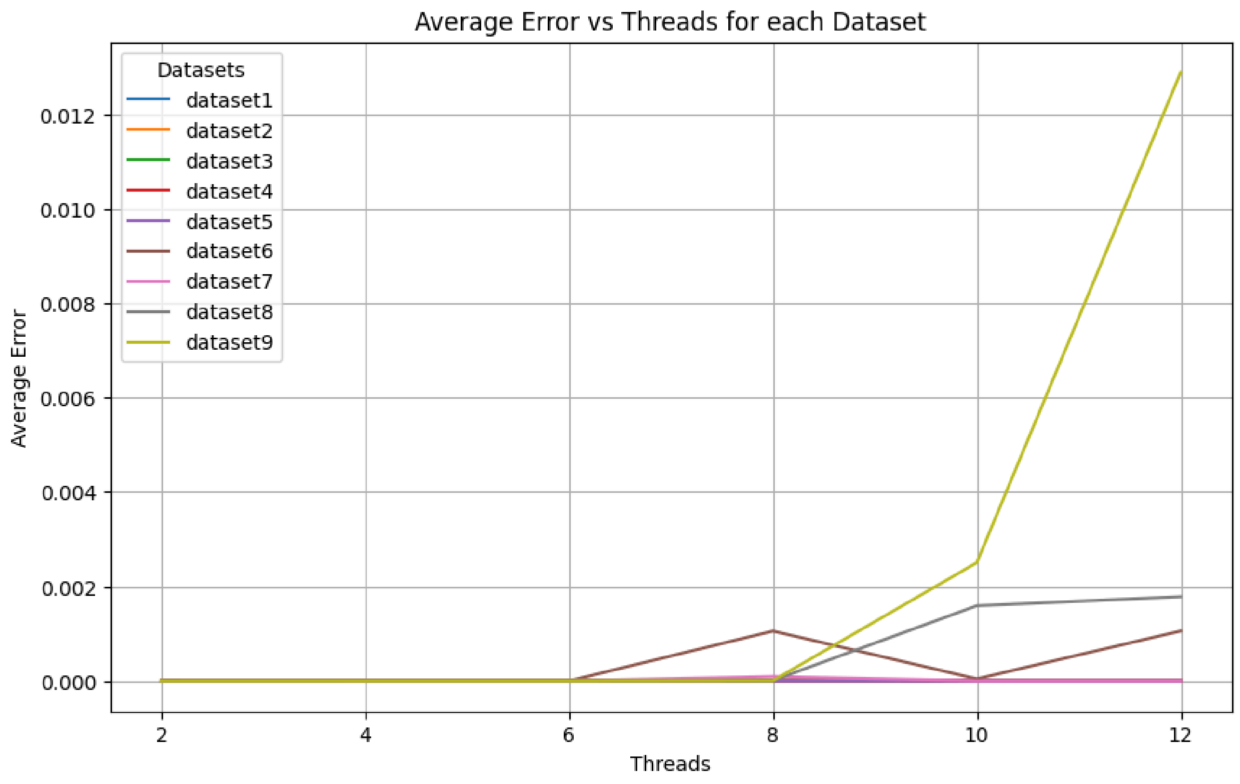
Theoretically, if the problem size is too low, the speedup decreases as the number of cores increases. This is because the cost of overheads will be far more than the actual cost of computation, thereby giving a low speedup. But for larger problem sizes, the overhead cost is negligible and the work could be distributed among many processors, thus the speedup increases as the number of processors increases. The best speedups are obtained for the larger-sized threads (sizes 800,000 and 1 million). The worst performance is for the dataset-sized 10,000.
Linear speedup is not achievable, in general, because of contention for shared resources, the time required to communicate between processors and between processes, and the inability to structure the software so that an arbitrary number of processors can be kept usefully busy.

Superlinear speedups are observed in some runs at single-threaded runs due to out-of-control system execution properties and/or latencies.

Below is the graph displaying the efficiency of our implementation achieved on various datasets with different numbers of threads.



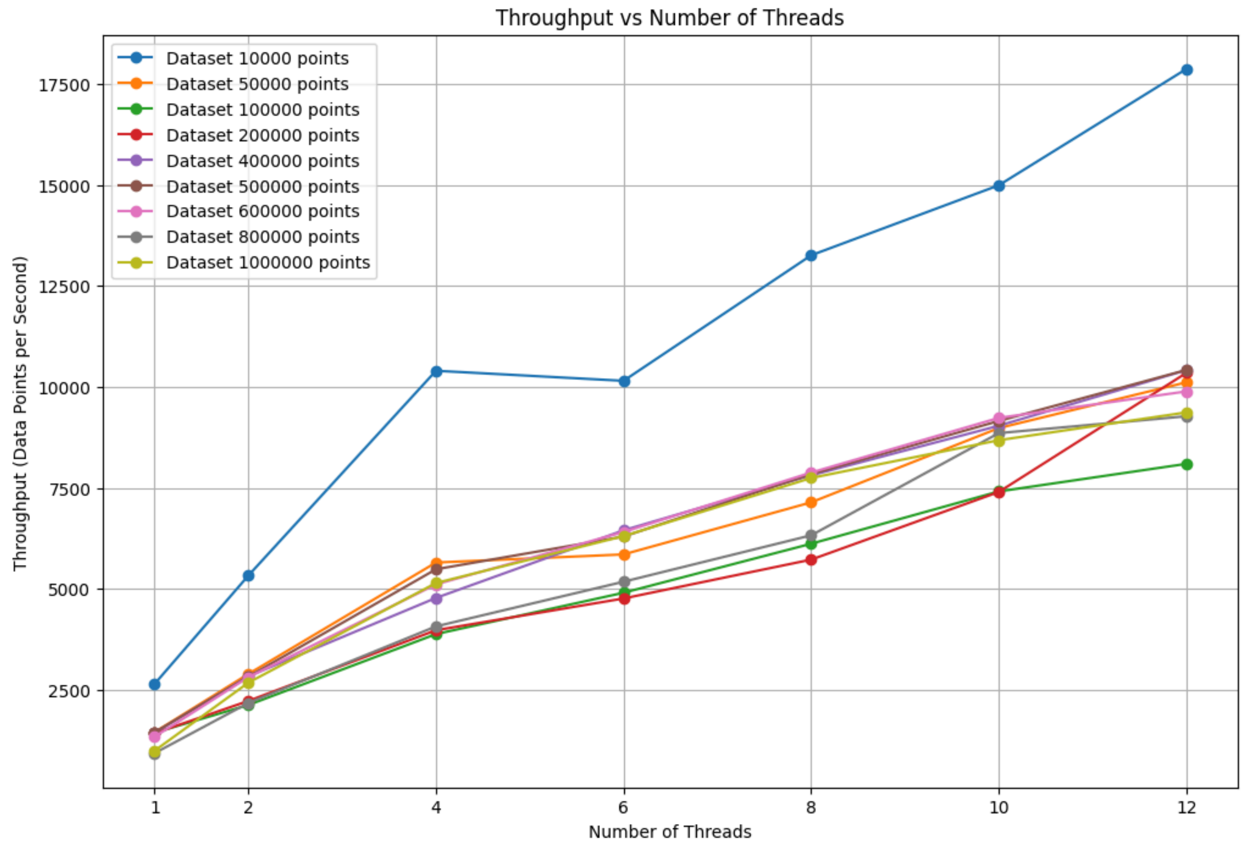
The efficiency curves decreased for every instance run with an increase in the number of threads. This is again due to the diminishing returns property on including an extra thread - a consequence of Amdahl's Law. The efficiency increases if the problem size is increased keeping the number of processing elements constant. And this fact is clearly visible for 12 threads. The efficiency at num_threads = 1 goes above 1.0 in certain runs due to superlinear speedups perhaps due to out-of-control system latencies during particular executions. The efficiency curves demonstrate the best efficiency at the larger-sized datasets (800,000 and 1 million) and the worst performance at smaller-sized ones (10000).



The graph above represents the mean distance between the i-th centroid calculated sequentially and the i-th centroid calculated in parallel, plotted against the number of threads used. The results indicate that for a lower number of threads, the parallel algorithm closely mirrors the output of the sequential algorithm, yielding almost identical results.

As the number of threads increases, a minor error—on the order of 0.01—is observed in datasets with a large number of data points. However, this error remains within an acceptable range, demonstrating that the parallel implementation maintains a high level of accuracy while achieving significant computational efficiency. These results confirm the successful implementation of the parallel K-Means clustering algorithm.

Below is the graph displaying the throughput of our implementation achieved on various datasets with different numbers of threads.



The graph illustrates a clear relationship between throughput and the number of threads. Specifically, as the number of threads increases, the throughput also rises. This behavior can be attributed to the parallel nature of the computation.

When the number of threads is increased, the workload is distributed across multiple threads, enabling the processing of a larger number of data points simultaneously. This parallelism minimizes idle time and enhances the efficiency of operations, resulting in a higher throughput.

Thus, the increase in throughput can be directly linked to the improved utilization of computational resources facilitated by multithreading.

CONCLUSION

By the end of this project, we implemented and optimized a parallelized version of the k-means clustering algorithm. As an outcome, there was a notable reduction in computation time, allowing the algorithm to handle massive datasets more efficiently. Through our evaluation metrics—time efficiency, throughput, and accuracy comparison—we assessed the performance gains achieved through parallelism and ensured that clustering accuracy remained consistent with the serial implementation. This approach holds promise for future expansions, including support for higher-dimensional data, making it a scalable solution suitable for large and complex datasets.

REFERENCES

1. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Chandramohan-Fall-2012-CSE633.pdf
2. https://medium.com/@TRikace/parallel-k-means-clustering-with-reducer-function-9be00135edce
3. https://in.mathworks.com/help/stats/kmeans.html
4. https://dl.acm.org/doi/abs/10.1145/3469213.3470413

GITHUB CODE

- https://github.com/arnavjain2710/Parallel-K-Means-Clustering-Algorithm