CS5300 : PARALLEL AND CONCURRENT PROGRAMMING
PARALLELISING NSW SEARCH FOR NN SETTING
FINAL PROJECT REPORT

Anudeep Rao Perala	Asli Nitej Reddy Busireddy	Purab Ganta
CS21BTECH11043	CS21BTECH11011	CO21BTECH11005

Contents

1 Problem Statement

1.1 K-Nearest Neighbor Search

The K-Nearest Neighbor (K-NN) Search Problem is a fundamental challenge in the field of data science and machine learning. It involves finding the K closest points (neighbors) to a given query point in a multi-dimensional space. This problem is crucial in various applications such as recommendation systems, classification tasks, and clustering.

1.2 Question Background

- HNSW (Hierarchical Navigable Small World) is an efficient method for finding approximated K nearest neighbors in high-dimensional spaces. This was developed to address the problem of the **curse of dimensionality** which makes many traditional nearest neighbor search techniques computationally inefficient as the dimensionality of the dataset increases. This HNSW algorithm builds on another graph based structure called as NSW network.
- The HNSW algorithm is similar to skip-list where in place of linked list as layers, in HNSW we have NSW networks as the hierarchial layer components.

1.3 Description about NSW Networks

Navigable Small World (NSW) networks play a crucial role in the context of Nearest Neighbor searches, especially in high-dimensional spaces. The concept, inspired by the small-world phenomenon observed in social networks, refers to the way nodes (or data points) are interconnected in a manner that allows for efficient navigation and search. In the context of Nearest Neighbors, NSW networks are used to create a graph structure that enables quick and efficient searching for the closest points to a given query point.

- **Features of NSW Networks:**
  - In NSW networks, data points are treated as nodes in a graph. Each node is connected to several other nodes, not necessarily the closest ones. This structure forms a network that is neither completely regular nor entirely random.
  - The term 'small world' refers to the property of the network where any node can be reached from any other node in a relatively small number of steps, despite the network's potentially large size.
  - NSW networks allow for dynamic connections between nodes. This flexibility enables the network to adapt to the complex geometry of the data, improving search efficiency.
- **Key Components:**
  - Graph: The graph is composed of nodes and edges, where each node represents a data point in the dataset and edges connect nodes that are close based on the defined distance metric.
  - Distance Metric: This is a function(ex: Euclidean distance) which is used to determine the similarity between data points. The choice of distance metric is very important for the quality of the connections in the graph and in deciding the relation between the datapoints for commenting on the terms **nearest** and **closest**.
- **Main Procedures:**
  - Graph Initialization: Start with an empty graph.
  - Graph Construction: Begin with a random node. For each subsequent data point, find its closest nodes in the graph (using a greedy algorithm), and then connect the data point to a fixed number of the closest nodes. Repeat this for all input data points.

- Search: To search for the nearest neighbor of a query point, begin at a randomly chosen node. Move iteratively to the node that's closest to the query point(using a greedy approach). This process will generally lead to a local minimum, where no neighboring node in the graph is closer to the query point than the current node. This node is considered an approximation of the nearest neighbor. It's important to note that due to the nature of the algorithm, the result might not always be the exact nearest neighbor but an approximation.

- **Input:**
  - D: A dataset of N data points.
  - K: An integer parameter denoting the number of neighbors to be connected to each node during graph construction. This helps control the density and connectivity of the graph.
  - *Distance Function*: A distance metric used to measure the similarity between data points.

- **Output:**
  - A graph G, represented as $G = (V, E)$
  - V: Set of vertices, representing the data points in D so $|V| = N$
  - E: Set of edges, representing the connections between data points.

2 Project Idea

In our project we want to Parallelise the Search algorithm in the NSW network, which play a crucial role in both construction of graph and for querying the input datapoint for K-Nearest Neighbors.

3 Parallelizing the Search Algorithm

- Here we have tried two versions of parallelization for the search algorithm they are:

  1. Static Allocation of Data to threads (SAM)
  2. Dynamic Allocation of Data to threads (DAM)

- The basic idea for parallelization in these both the methods goes like this, in the greedy algorithm for searching we have entry index from where we start our search for K-NN and we return the K-NN after we have searched for all the entry indexes.

- Now what we want to do is that for each thread we would be giving entry index to each thread, then that thread would find its K-NN and then add that to a priority queue which is a global variable shared by all the threads in the process of finding the K-NN.

4 Static Allocation Method (SAM)

4.1 Working of SAM

- SAM parallelization involves dividing the overall workload into discrete tasks or chunks statically, meaning the division of work is determined before the parallel execution begins.

- In SAM, each thread receives an approximately equal portion of the workload.

- SAM is pretty effective in the situations where we have full information about data which we are working on for parallelization.

- Working of SAM in this parallelisation:

  – In this method of parallelization we have an array of entry indexes, here for the thread i the array indexes $i, i + 1 * N, i + 2 * N, ...$ (where N is the total number of threads) is assigned.

4.2 Drawbacks of SAM

While this approach can improve efficiency, especially for large datasets, it assumes that the workload is evenly distributed across tasks. If some tasks are inherently more computationally intensive than others, this could lead to imbalances where some threads finish much earlier than others, leading to under utilization of resources.

5 Dynamic Allocation Method (DAM)

5.1 Working of DAM

- This approach dynamically assigns tasks to threads during runtime, rather than dividing them up statically before execution. DAM is useful in situations where predicting the exact workload of each task is not trivial.

- This approach ensures more effective load balancing across threads. In situations where tasks have unpredictable execution times, DAM helps prevent situations where some threads remain idle while others are overloaded.

- Working of DAM in this parallelisation:

  – We have a Counter class which basically has an integer, as soon as a thread becomes free it then accesses this global Counter class variable which assigns an entry index to the accessing thread and after that the thread works on that entry index.

- Basically the Counter class variable assigns an entry index to that thread that first requests for the value, the assigning is stopped after the Counter class integer variable has reached maximum number of entries.

5.2 Drawbacks of DAM

While DAM provides better load balancing, it introduces overhead due to the dynamic allocation process. This includes the time taken for threads to fetch new tasks and the synchronization required to manage access to the shared task pool.

6 Experiments

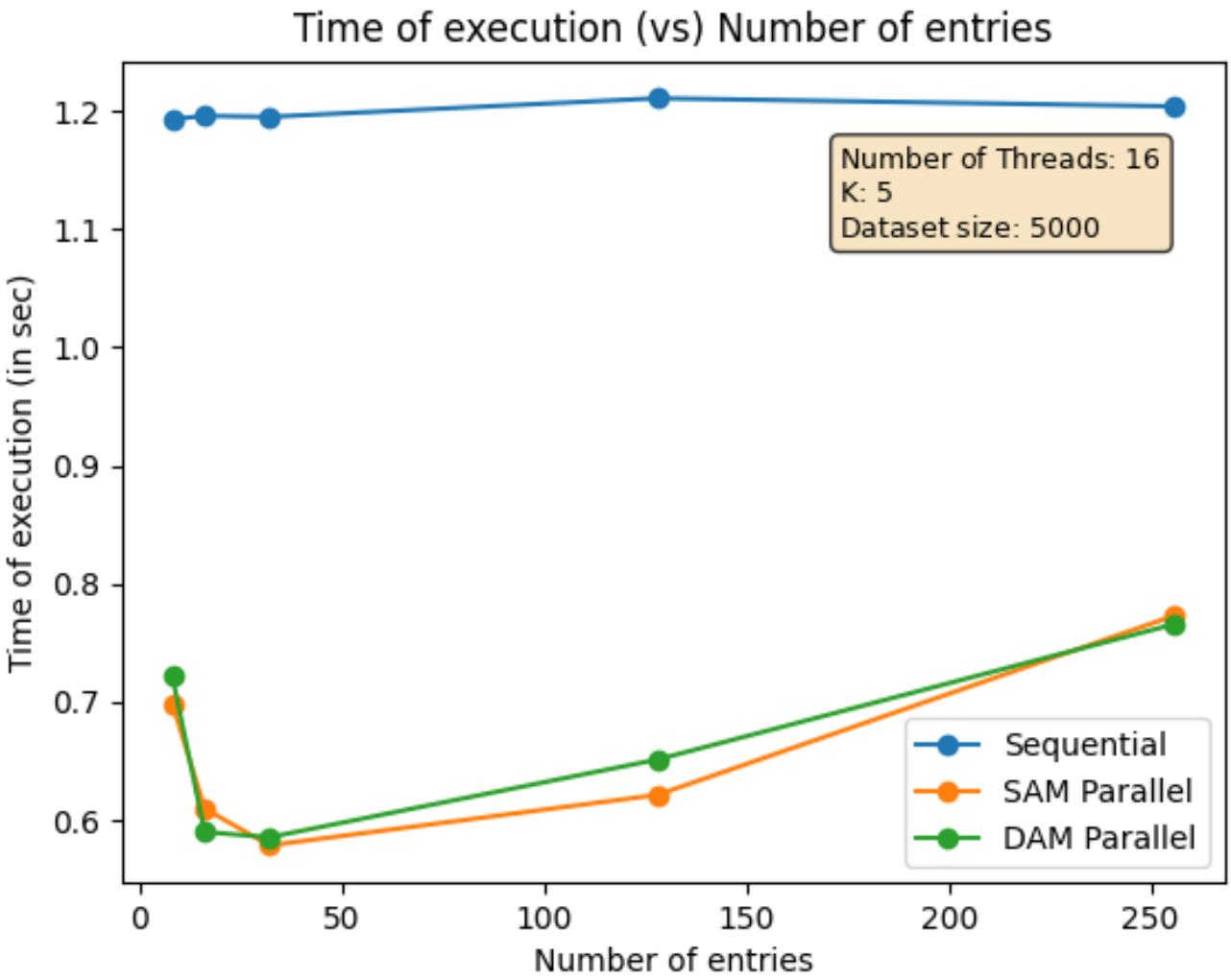
6.1 Experiments on Search Parallelization

We have two experiments for this NSW network and we are performing search for 100 values on an NSW graph.

6.1.1 Experiment - 1

Constraints:

- Number of Threads: 16
- K: 5
- Dataset size: 5000



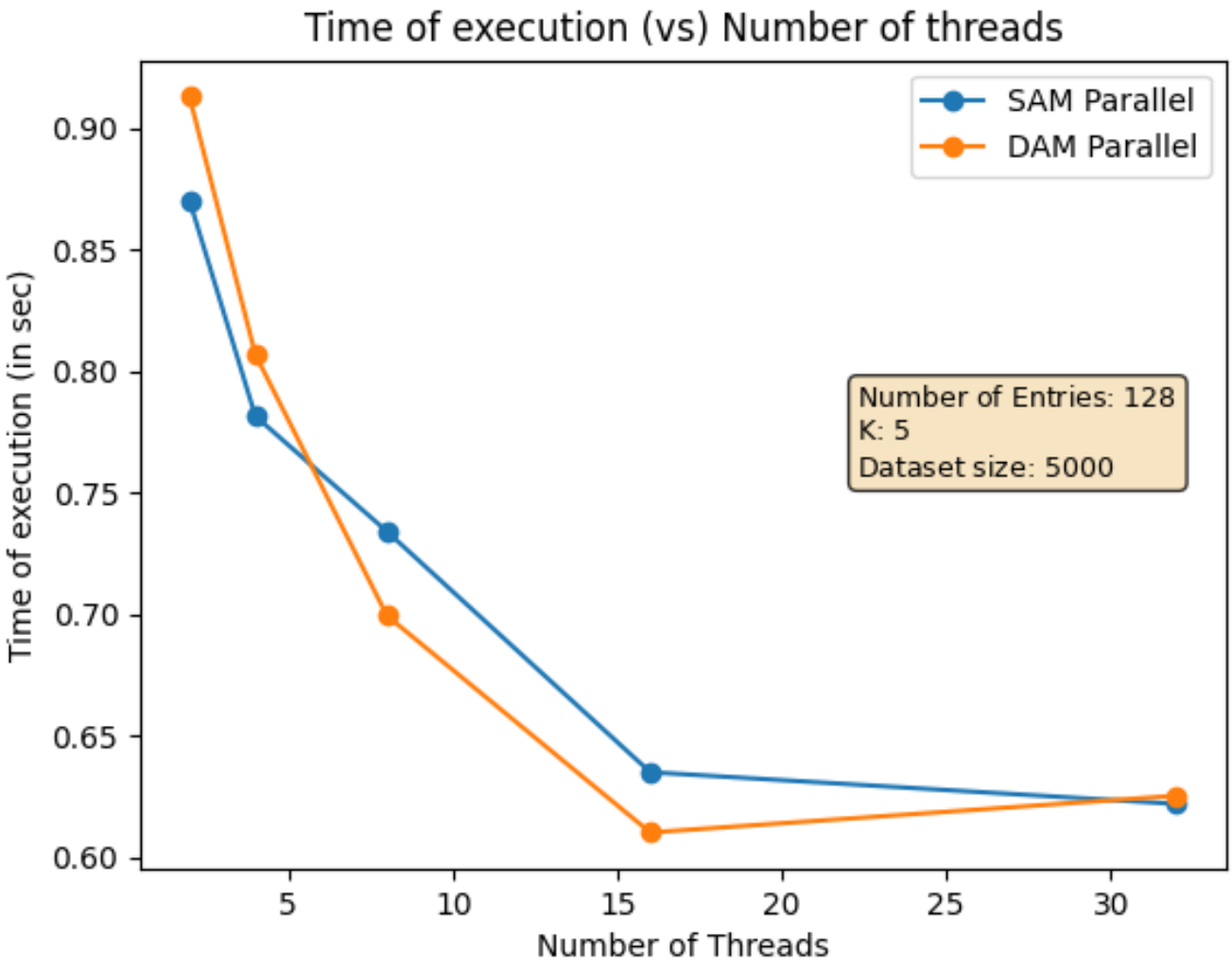
Observations:

- Both SAM and DAM parallelization methods show a significant reduction in execution time compared to the sequential approach of about 2 times. This demonstrates the benefit of parallelising the search algorithm.
- As the number of entries increases, the execution time for both parallel methods (SAM and DAM) increases, which is expected since more data points require more computation.
- As the number of entries increases, DAM parallel seems to show a slight advantage over SAM parallel, suggesting better handling of larger workloads through its dynamic task distribution.
- The initial drop in execution time for parallel methods (particularly noticeable with DAM) which could be due to the overhead of setting up parallel threads because of larger number of entries. As the number of entries increases, the overhead becomes less significant compared to the total computation time.

6.1.2 Experiment - 2

Constraints:

- Number of Entries: 128
- K: 5
- Dataset size: 5000



Observations:

- For both SAM and DAM, as the number of threads increases, the execution time decreases. This shows that parallelisation has effectively reduced the time required for the nearest neighbor search.
- Initially, with a lower number of threads, the SAM and DAM methods show similar execution times. As the number of threads increases, the DAM method appears to be more effective at reducing execution time than SAM, suggesting better efficiency in dynamic task allocation compared to static allocation. The overhead of accessing of global variable in DAM becomes insignificant as we increase the size of the dataset.
- The graph suggests there might be a point of overhead or saturation where adding more threads does not result in a significant decrease in execution time. This could be because the workload is not large enough to justify the use of additional threads.

6.2 Experiments on Graph Building

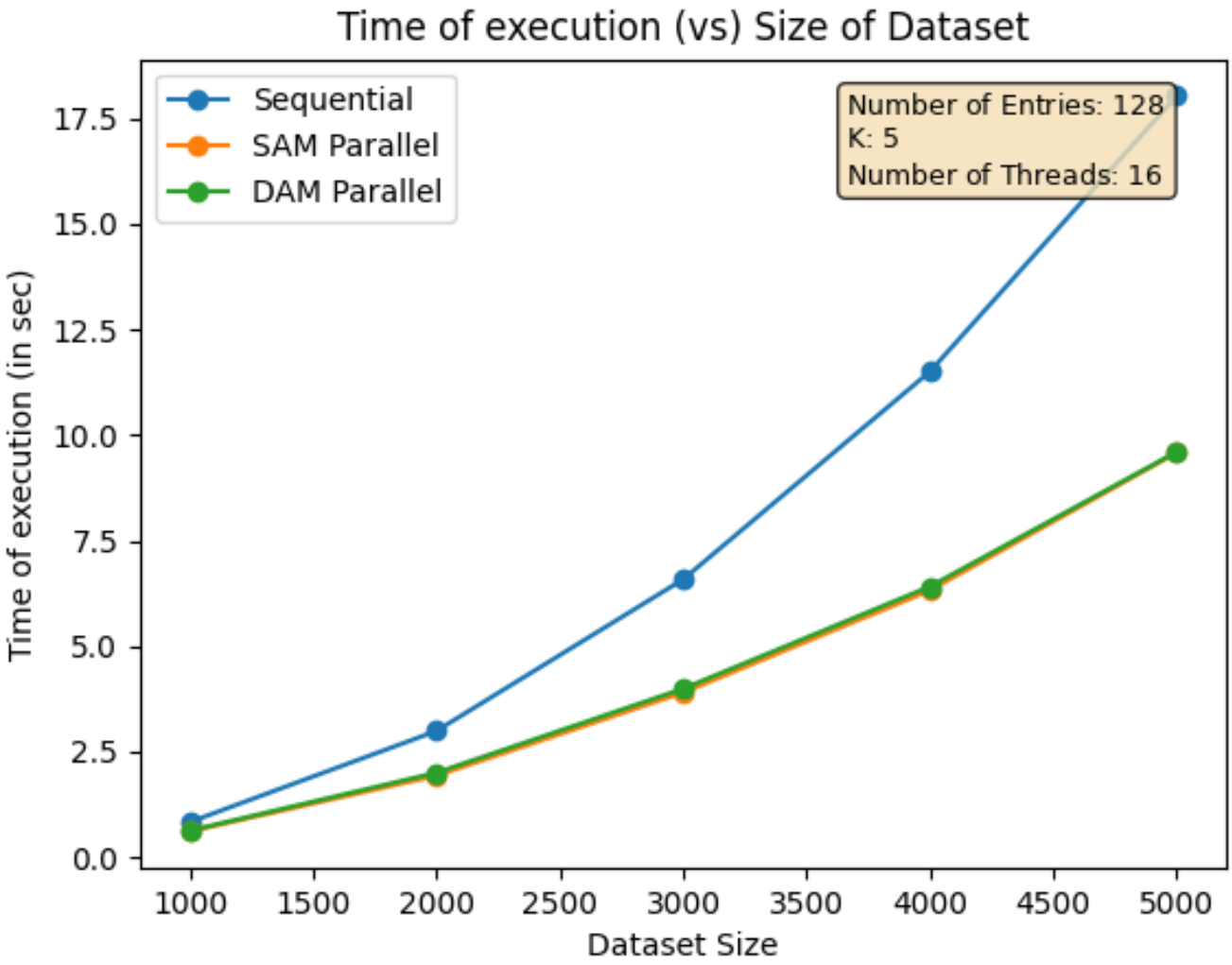
We have four experiments for the building of NSW network.

6.2.1 Experiment - 1

Constraints:

- Number of Entries: 128

- K: 5
- Number of Threads: 16



Observations:

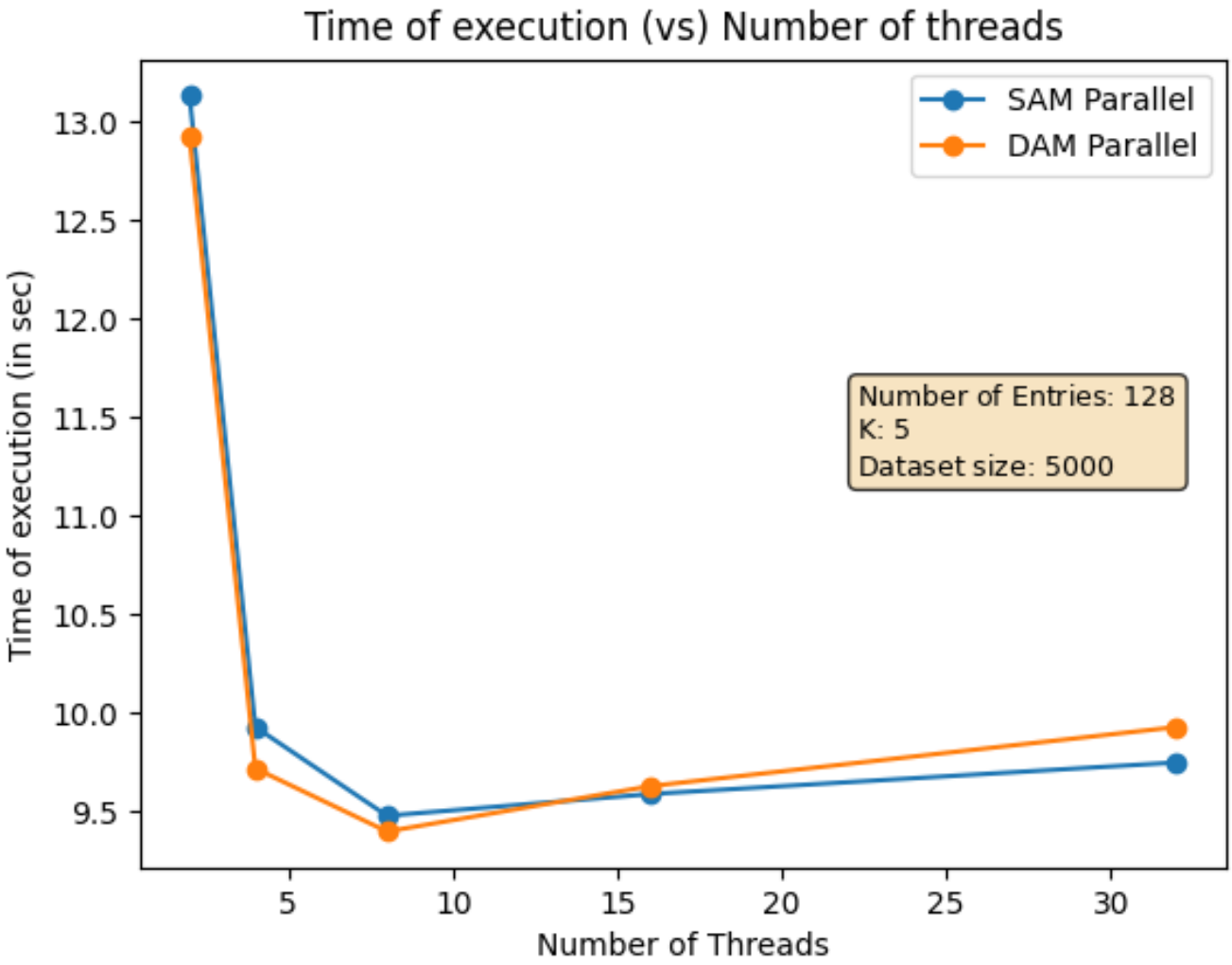
- Both parallel methods (SAM and DAM) have lower execution times compared to the sequential method across all dataset sizes of about 2 times, demonstrating the advantage of parallel processing in building the NSW graph.
- The sequential method's execution time increases at a much faster rate than the parallel methods indicating the inability of the sequential algorithm in handling larger datasets efficiently.
- The initial execution times for the parallel methods are close to one another, indicating that the overhead of managing parallel tasks is comparable between SAM and DAM. As the dataset size increases, DAM's dynamic allocation might be handling the increased complexity more effectively.

6.2.2 Experiment - 2

Constraints:

- Number of Entries: 128

- K: 5
- Dataset size: 5000



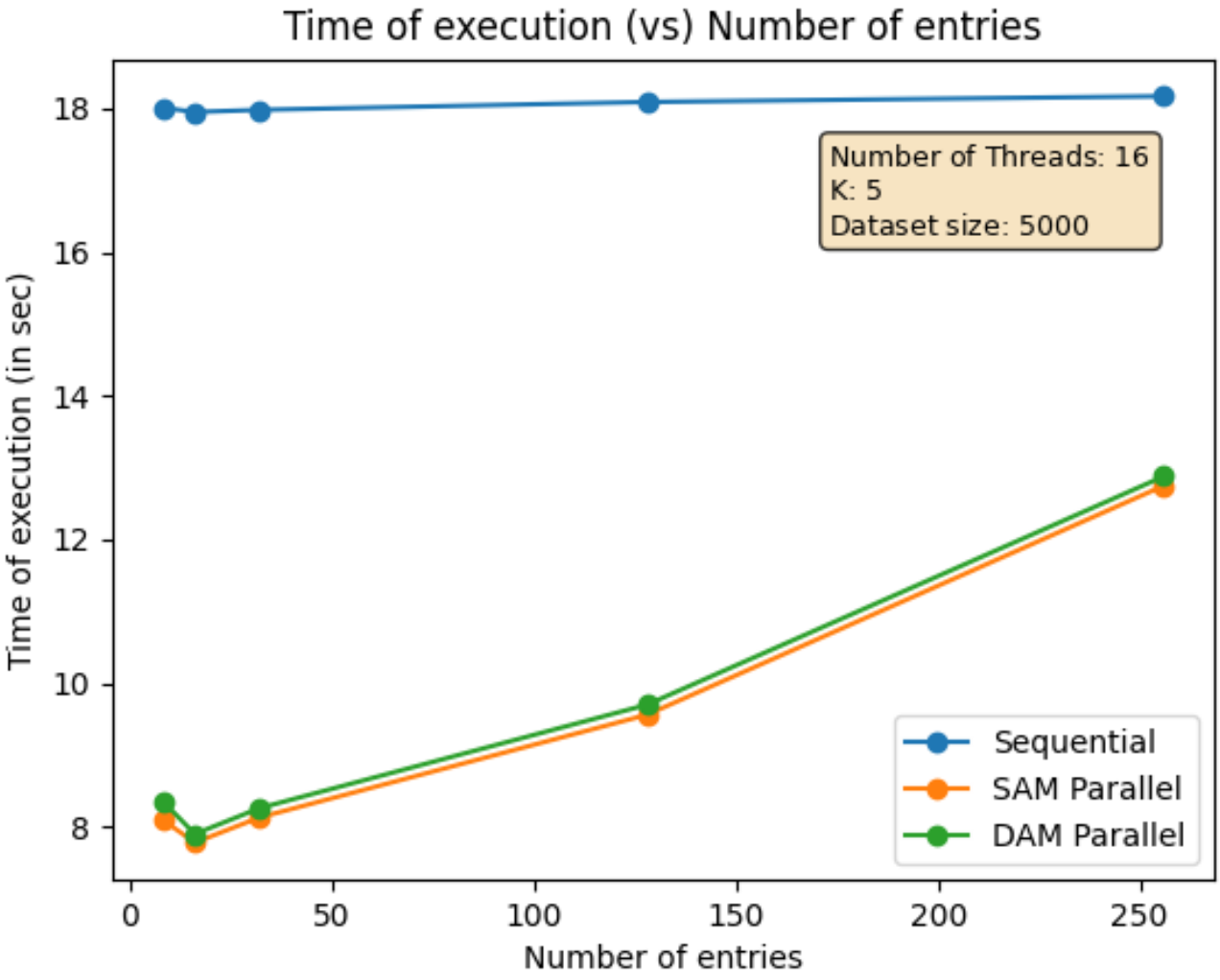
Observations:

- We can see that there is a sharp decrease in execution time for both methods as the number of threads increases from 1 to around 5. This indicates that both SAM and DAM benefit significantly from the initial addition of threads.
- After the initial decrease, the execution time for both methods plateaus. It appears that increasing the number of threads beyond a certain point (around 5-10) does not result in significant further decreases in execution time. After that the time increases as the number of threads increases which is because of the overhead of thread creations playing significant role in increasing the time for building the graph. This is because as the size of the dataset beign small using high number of threads doesn't help in this case.
- Both methods reach an optimal performance at around 10 threads, after which adding more threads has a negligible impact on decreasing execution time.
- The graph clearly shows us that there is an overhead associated with adding more threads, which could be due to the costs of context switching, thread management, or synchronization issues that do not contribute to further performance gains.

6.2.3 Experiment - 3

Constraints:

- Number of Entries: 128
- K: 5
- Dataset size: 5000



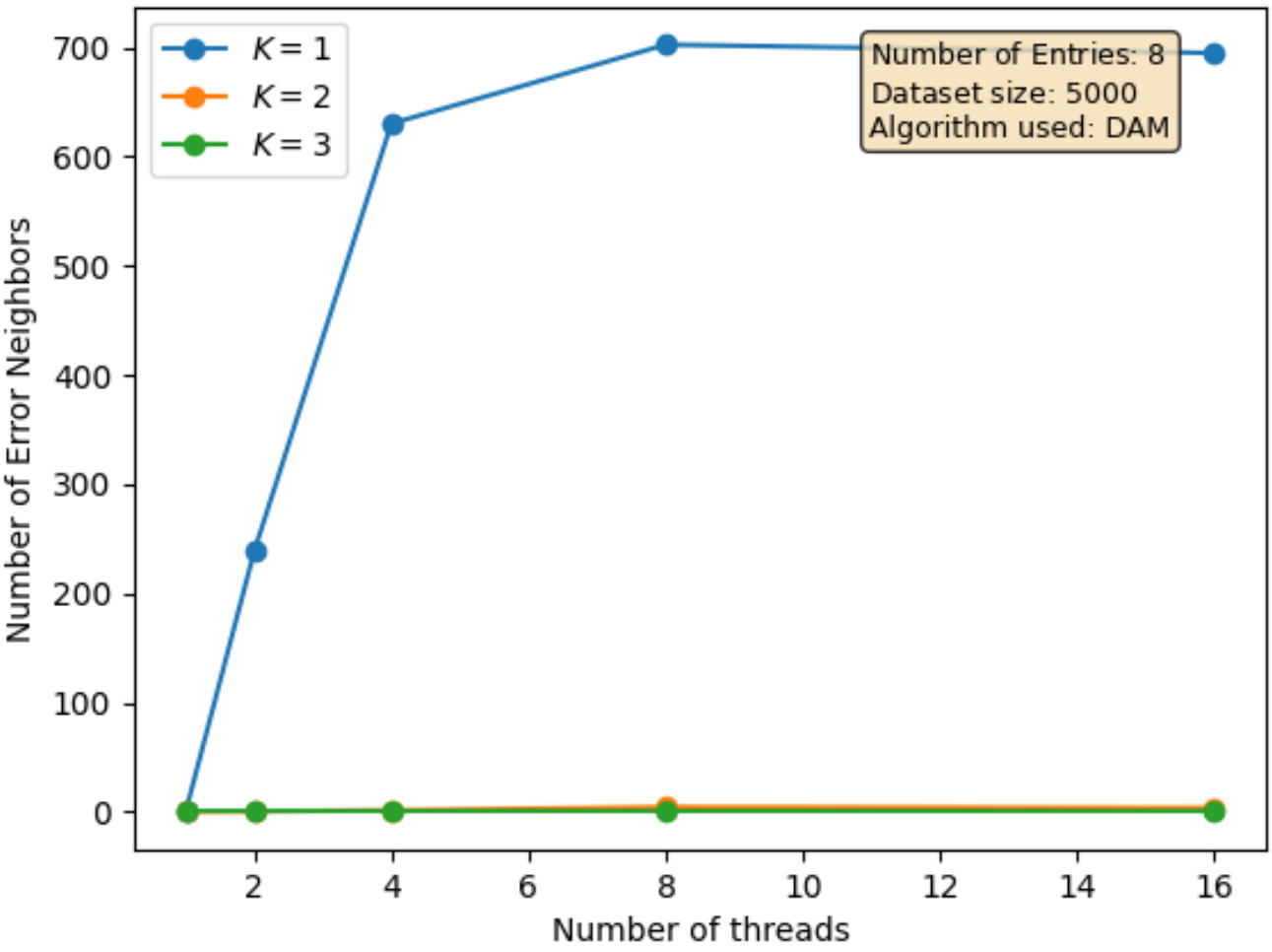
Observations:

- Both parallel methods (SAM and DAM) demonstrate a significant reduction in execution time compared to the sequential method of about 1.5 times. This shows the efficiency gained from parallelisation in building the NSW graph.
- The performance of the sequential algorithm doesn't seem to change a lot as the number of entries are increased this is because of the nature of the algorithm where our required work gets done with a small number of entries for such a small dataset size of 5000.
- Here the time of execution increases for SAM and DAM as the number of entries are increased which is as expected here this overhead can be decreased a bit by using the concept of thread pools as C++ doesn't support thread pools we were unable to implement thread pools. Using thread pools we can remove the overhead for creating threads for each entry index.

6.2.4 Experiment - 4

Constraints:

- Number of Entries: 128
- K: 5
- Dataset size: 5000



Observations:

- This graph indicates us the comparision of the graph formed by Sequential algorithm vs the graph formed by DAM parallelisation.
- We can clearly see that for $K = 1$ as the number of threads increases the number of error neigbhours increases this is because of the local minima halting of the random point entry.
- And because of this error neighbhours our whole graph build tend to go wrong.
- As the value of K increases this Error is decreased drastically because as we have $K > 1$ we search for more values and we dewise the KNN from a set rather than just for 1 point.
- Now the error values for $K = 2$ and $K = 3$ which are nearly zero indicates that the implemented parallel version for search algorithm is correct.

7 Progress so far

There exist no parallel version for the parallelisation of the NSW search algorithm in this setting of Nearest Neighbours. Our idea is that we want to parallelise the insertion and search for the NSW graph and then we want to parallelise the HNSW algorithm. There already exists parallelisation for HNSW algorithm but that follows a different approach from our approach to parallelise the HNSW algorithm.