Parallelising the NSW Search Algorithm

PROBLEM STATEMENT

The primary objective of this project is to implement a parallelized version of the Navigable Small World (NSW) search algorithm. This algorithm is very crucial in graph building and querying for K-NN. This project involves parallelising the NSW search algorithm making it efficient in the parallel computing environment.

NSW Background

- Navigable Small Worlds (NSW) are networks that have the attributes of small-world structures, which are characterized by short paths among nodes, and navigability.
- Navigable Small Worlds (NSW) networks are formed by sequentially adding elements as vertices, connecting them with their nearest neighbors. This structure, characterized by short paths and efficient navigation, is maintained through bidirectional links.
- NSWs use a variation of the greedy algorithm for efficient searching, where each new element is optimally connected to facilitate quick and efficient traversal.
- While NSWs can face challenges like getting trapped at local minima, their effectiveness measured by search path length and recall rate can be enhanced by adjusting the number of entry points.
- The small world properties are validated using experiments.

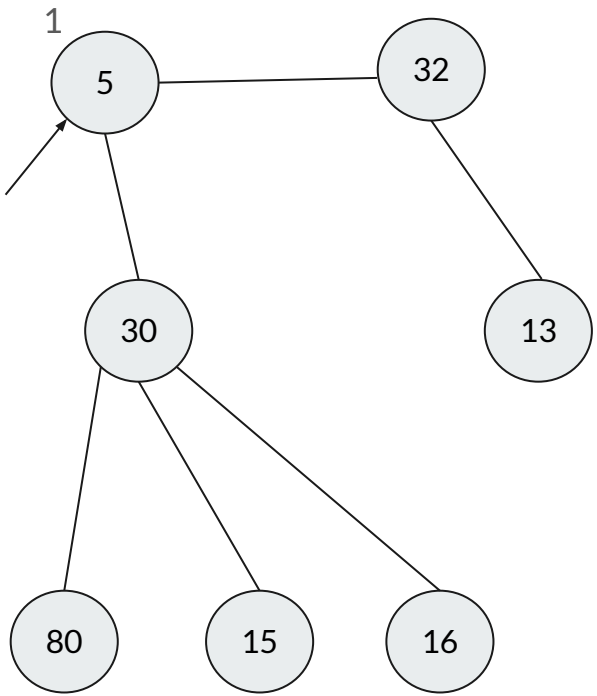
NSW Search algorithm

- This algorithm uses a greedy traversal algorithm which is similar to Dijkstra algorithm.
- The difference between Dijkstra algo and this search algorithm is that we won't have weights assigned to edges directly, but we will look at the distance between the current node and its neighbour node w.r.t., our query point and then we traverse accordingly.
- A neighbour for the current node is added to the candidate queue (which is a queue used to keep track of potential neighbors for search, traversal is stopped when this queue becomes empty), if our current node is at a greater distance than the neighbor node.
- If the neighbour node is at a larger distance from the query than the current node then we skip that neighbour node from adding to the candidate queue.

Setting for explaining the Search algorithm

- We have a **tempres priority queue**, which is local to the current entry point, which basically keeps the nodes which are found until then for adding to the **result** priority queue which is a global priority queue which is used to get the final K-NN.
- We have a **candidate priority queue**, which is local to the current entry point, which keeps track of the nodes for next traversal for the particular random entry. When this queue becomes empty we stop the traversal.
- We have a **result priority queue**, which is a shared global variable between all the threads after the traversal completes we add the elements in the **tempres pq** to this **result pq.**
- We have another global array of **atomic integers** which is used for maintaining the history of which nodes are previously visited or not which plays a major role in traversal of the graph while using the search algorithm.

Sequential NSW Search

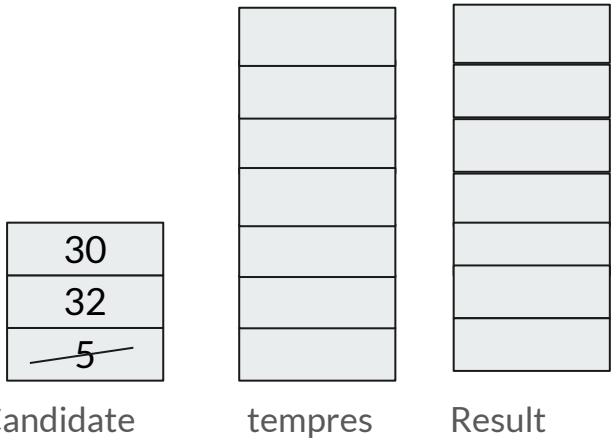


[The “1” which is present outside the node indicates that the node is visited]

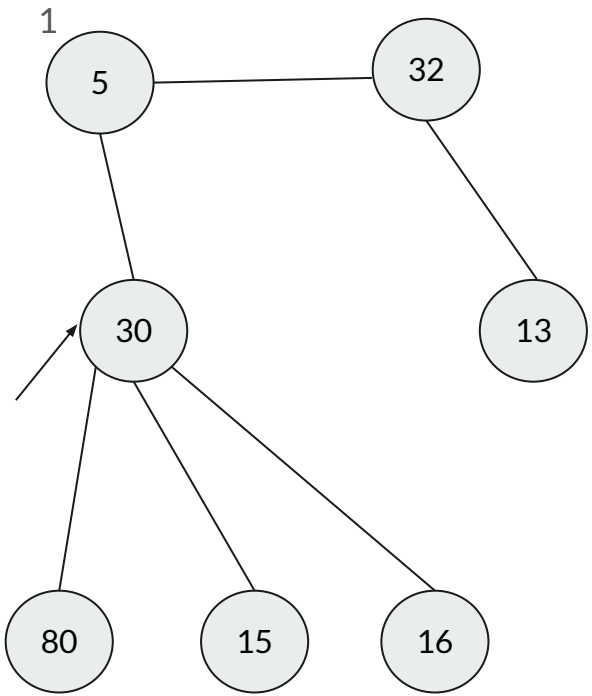
[Here Candidate, tempres, Result are priority queues]

Query Point = 20

We use Euclidean Distance: |x1-x2|



Sequential NSW Search

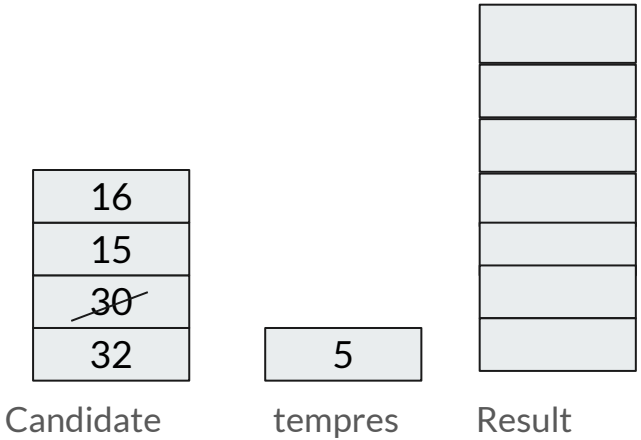


[The “1” which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

Query Point = 20

Node 80 is not added in “candidate” queue as dist(30,20) < dist(80,20)

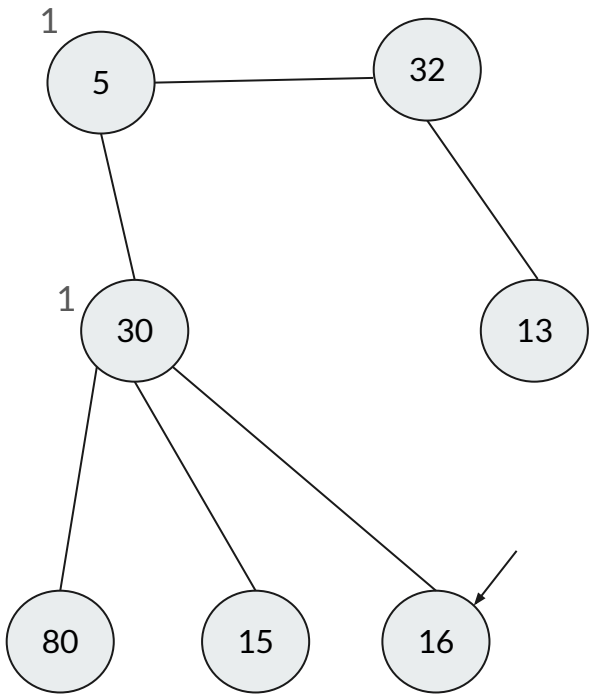


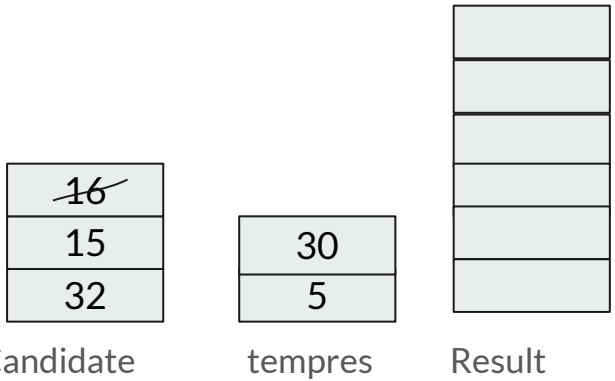
Sequential NSW Search

[The “1” which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

Query Point = 20



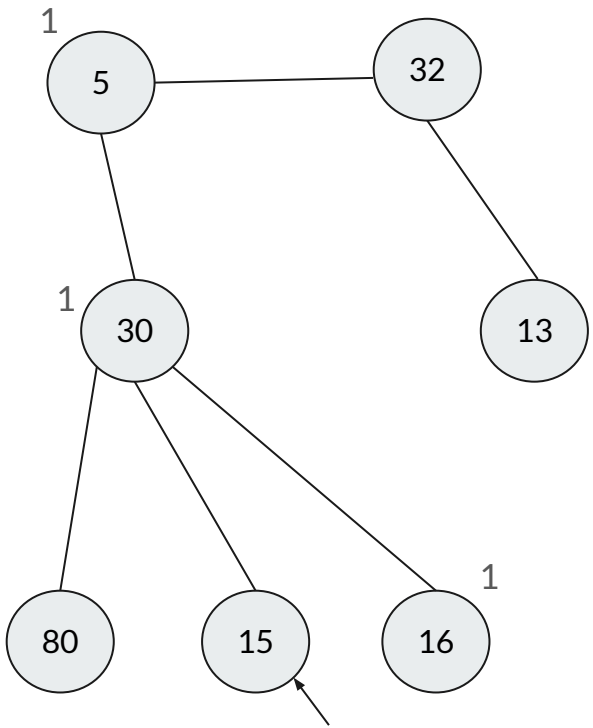


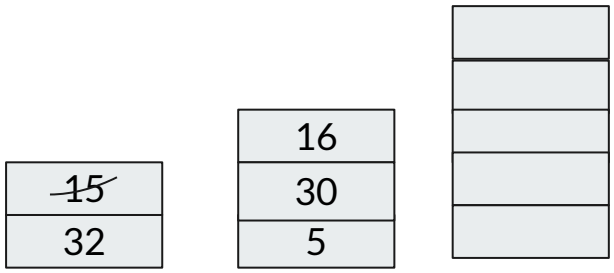
Sequential NSW Search

[The “1” which is present outside the node indicates that the node is visited]

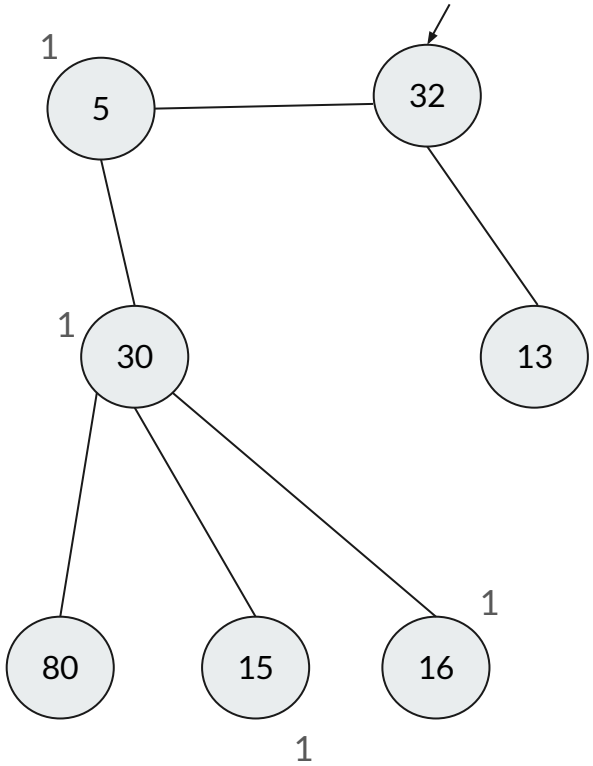
[Here Candidate, tempres, Result are priority queues]

Query Point = 20





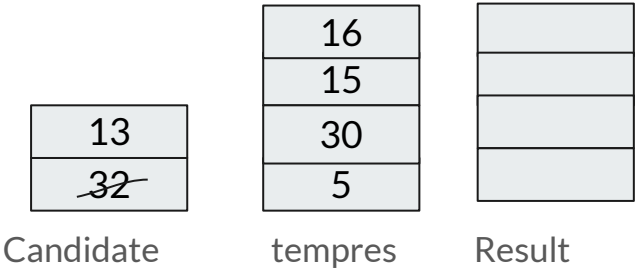
Sequential NSW Search



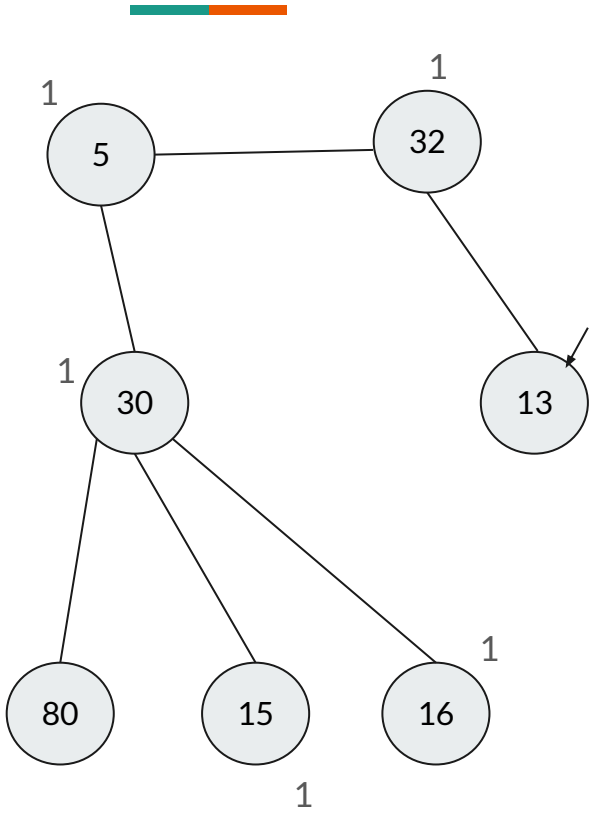
[The “1” which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

Query Point = 20



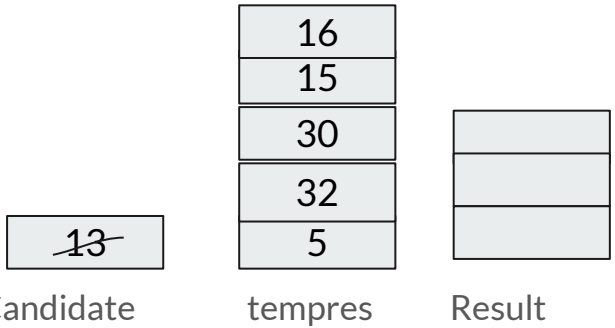
Sequential NSW Search



[The “1” which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

Query Point = 20

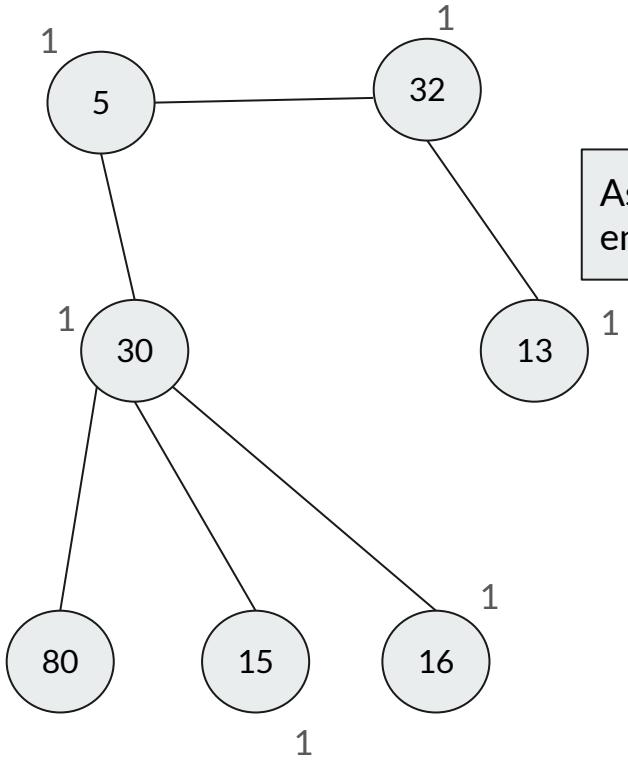


Sequential NSW Search

[The “1” which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

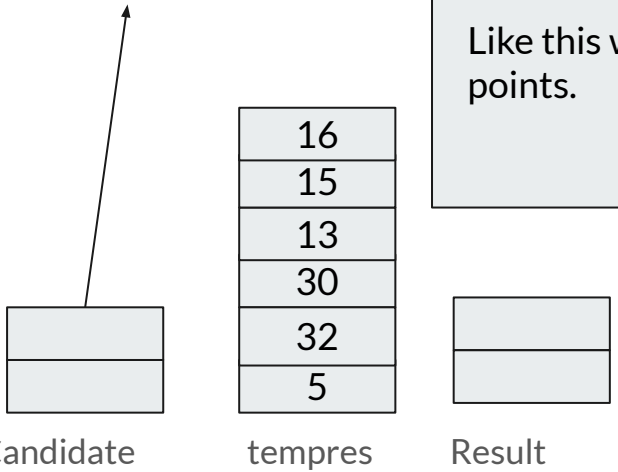
Query Point = 20



As “candidate” is zero, this entry index traversal is done

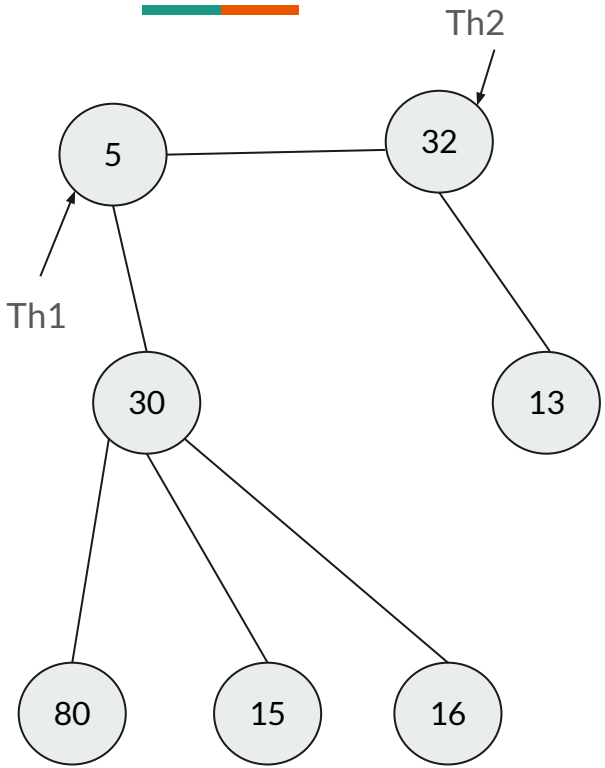
So now at the end push all the elements from “tempres” to “result".

Like this we do search for N entry points.



Parallel NSW Search

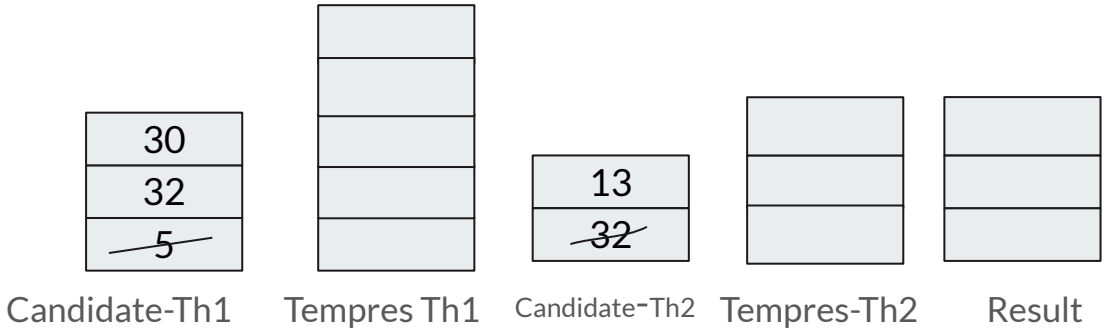
[For two threads]



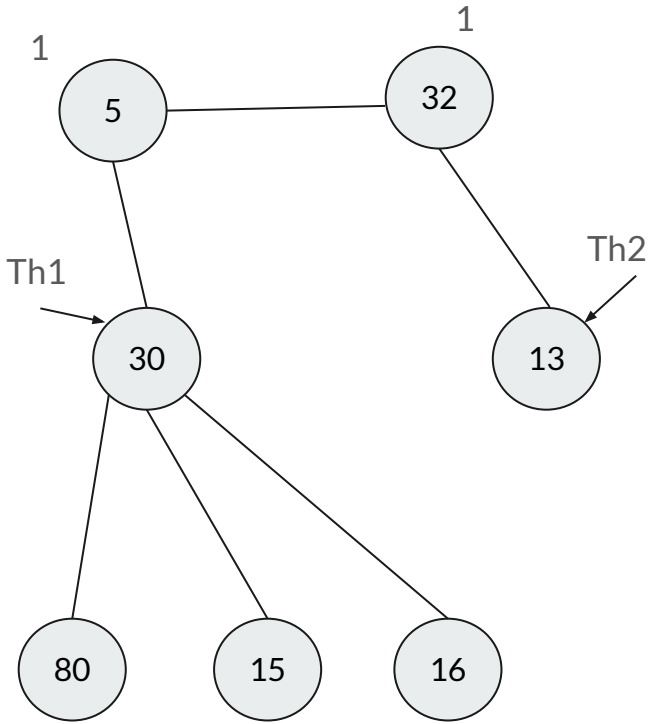
[The “1” which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues] Query Point = 20

This “Result” Priority queue is shared between the two threads.



Parallel NSW Search



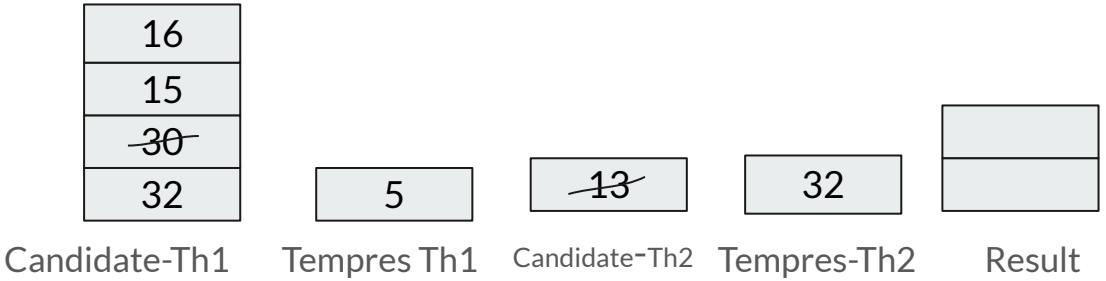
[The “1” which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

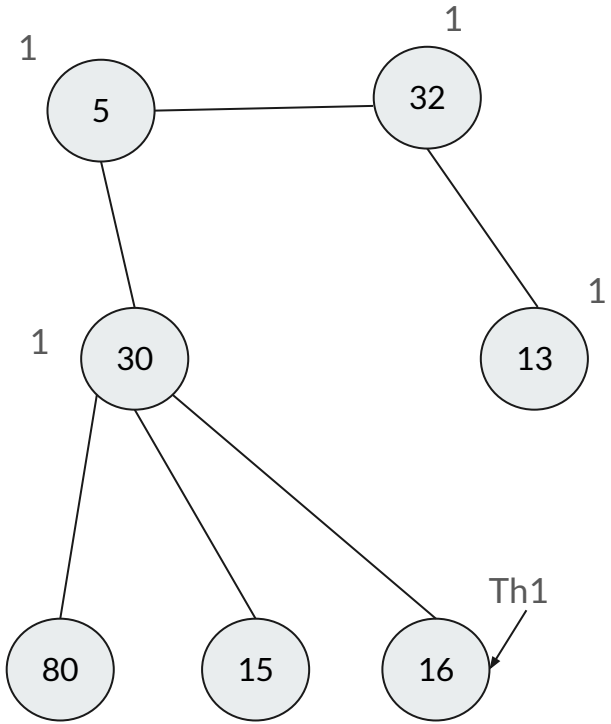
Query Point = 20

This “Result” Priority queue is shared between the two threads.

Node 80 is not added in the candidate queue, as dist(30,20) < dist(80,20)



Parallel NSW Search

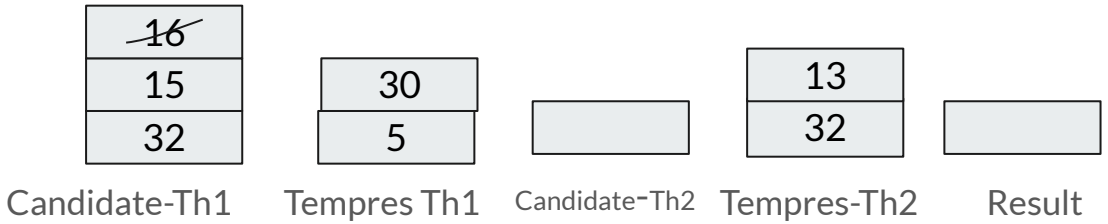


[The “1” which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

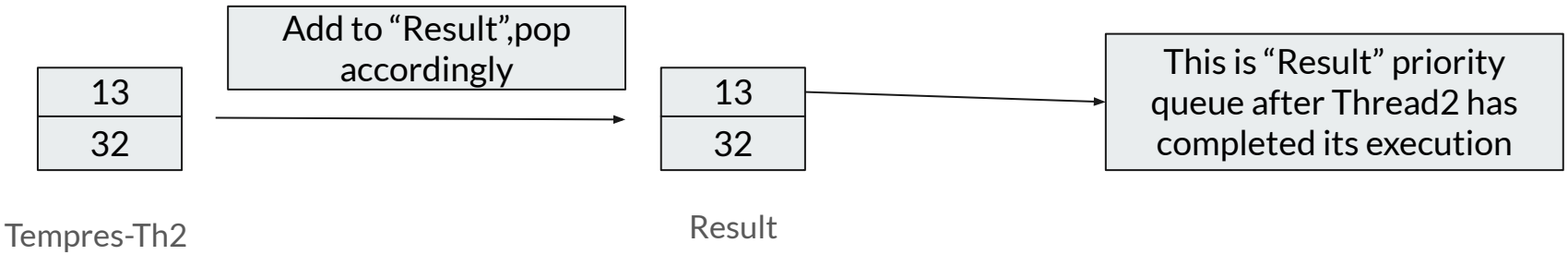
Query Point = 20

This “Result” Priority queue is shared between the two threads.

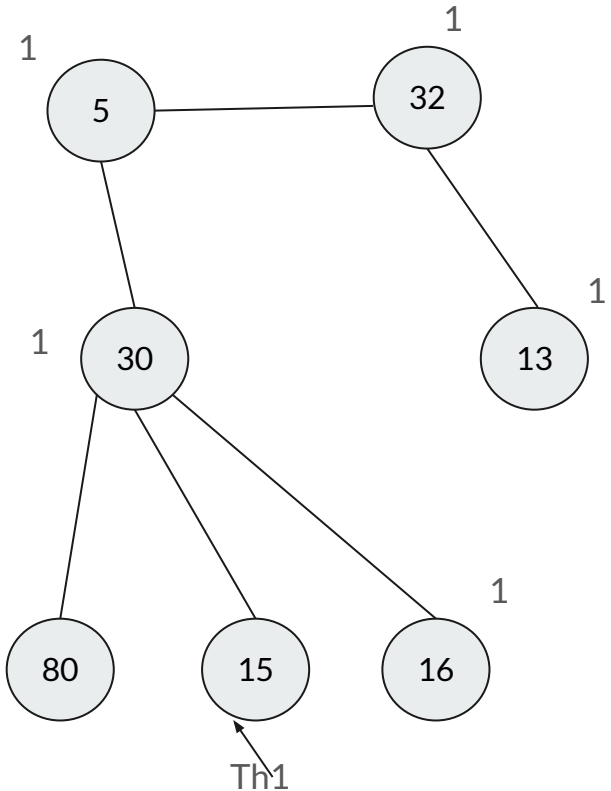


Parallel NSW Search

Now As “candidates-Th2” is empty add all elements in tempres-Th2 to “Result” priority queue.



Parallel NSW Search



[The "1" which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

Query Point = 20

This "Result" Priority queue is shared between the two threads.

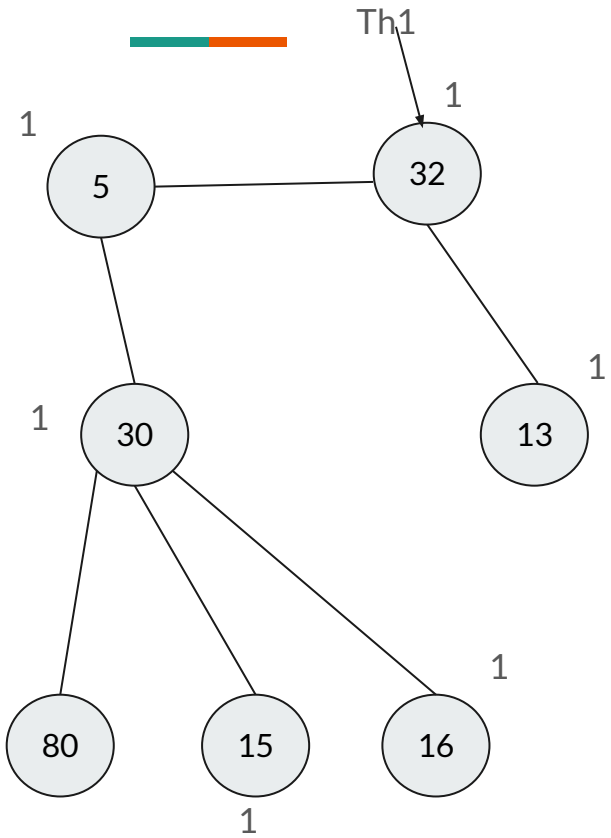
~~15~~
32

Candidate-Th1

16
30
5

Tempres Th1

Parallel NSW Search



[The "1" which is present outside the node indicates that the node is visited]

[Here Candidate, tempres, Result are priority queues]

Query Point = 20

This "Result" Priority queue is shared between the two threads.

As 32 is already visited we skip so this "candidate" queue is empty.

So we add "TempresTh1" elements to "Result" priority queue.

~~32~~

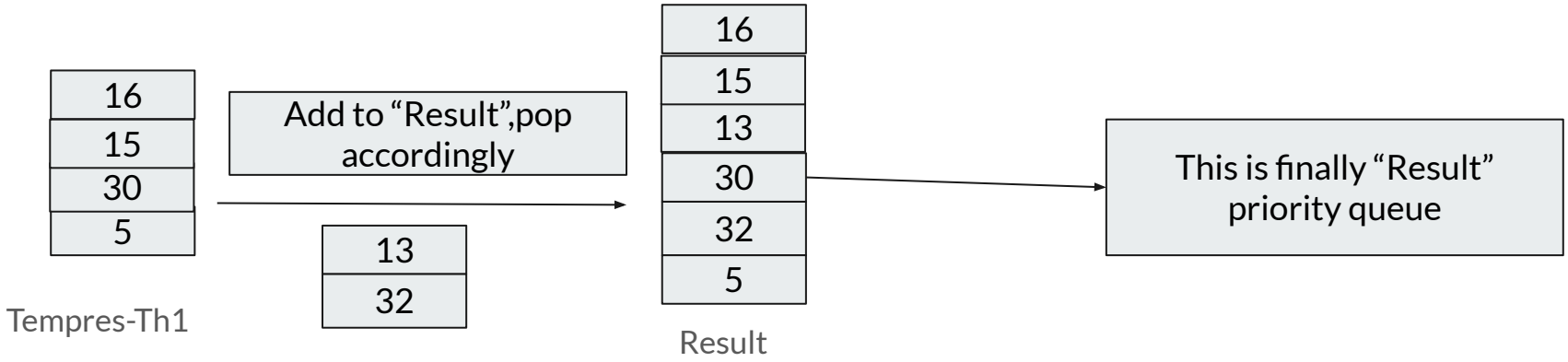
Candidate-Th1

16
15
30
5

Tempres Th1

Parallel NSW Search

Now add all elements in “tempres-Th1” to “Result” priority queue.



Like this threads work parallelly and update the “Result” Priority queue which is a shared variable for all the threads.

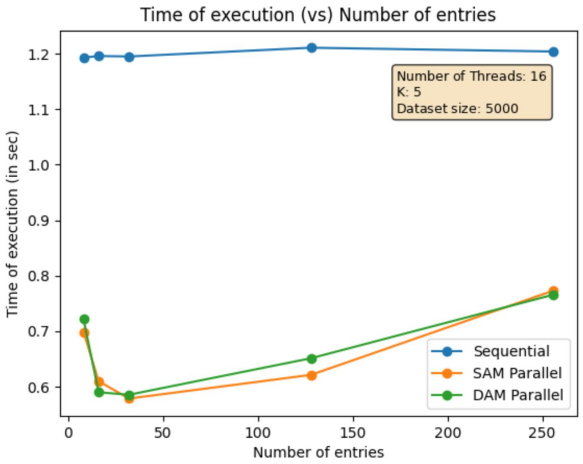
Experiments

We have two types of experiments:

1. Experiments on Search time.
2. Experiments on Building the NSW Graph.

Experiments on Search Time

Experiment - 1:



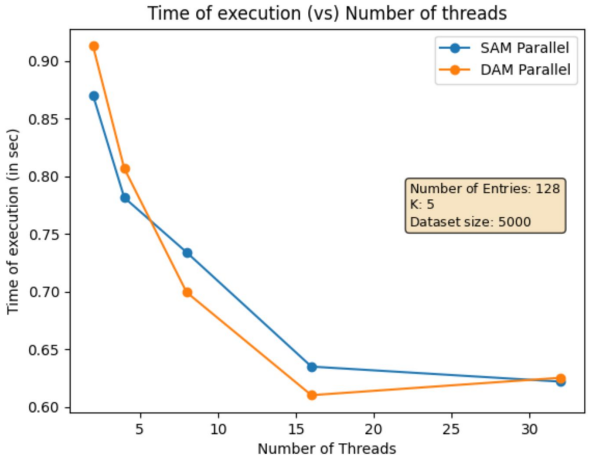
Constraints:
Number of Threads: 16
K: 5
Datasize: 5000

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.

Experiments on Search Time

Experiment - 2:



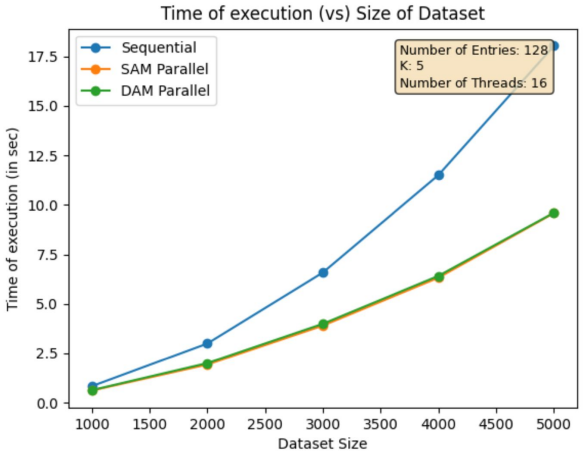
Constraints:
Number of Entries: 128
K: 5
Datasize: 5000

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.

Experiments on Graph Building

Experiment - 1:



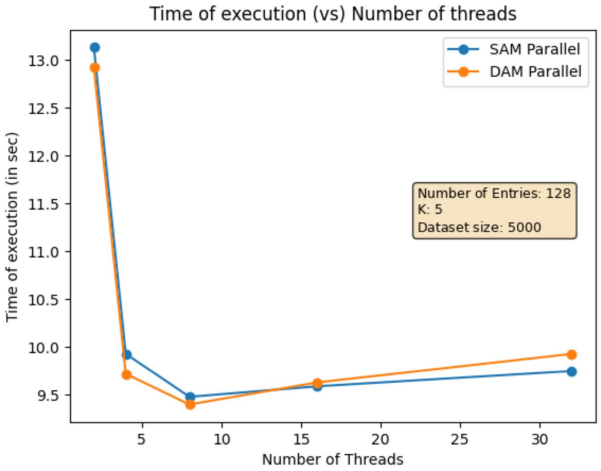
Constraints:
Number of Entries: 128
K: 5
Number of threads: 16

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.

Experiments on Graph Building

Experiment - 2:



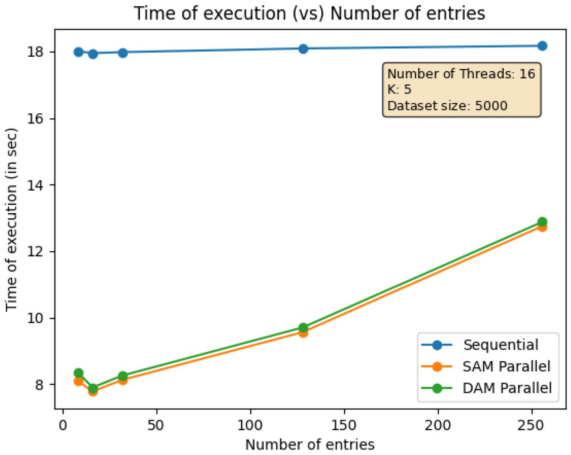
Constraints:
Number of Entries: 128
K: 5
Dataset size: 5000

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.

Experiments on Graph Building

Experiment - 3:



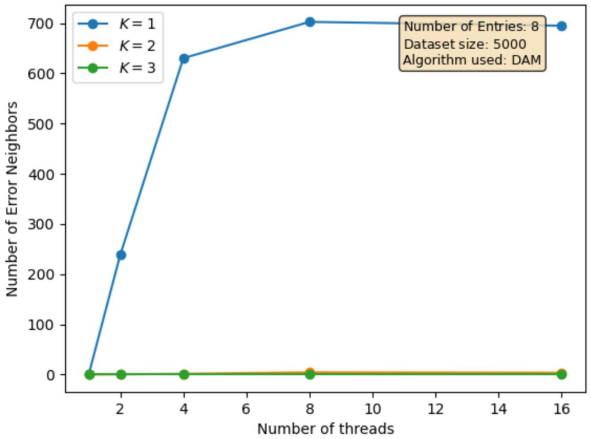
Constraints:
Number of Threads: 16
K: 5
Dataset size: 5000

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.

Experiments on Graph Building

Experiment - 4:



Constraints:
Number of Entries: 8
Dataset size: 5000
Algorithm: DAM

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.

Improvements

- Can use thread Pools.
- Can make the result PQ concurrent rather than using the temp result so that when we are at the wrong random entry point which does not suffice our Nearest Neighbors we can exit that entry point fast rather than traversing that entry point.
- Need to experiment on different values of entry points as using more entry points is of no use.